Université Joseph Fourier, Grenoble I
Institut Fourier, UFR de Mathématiques
Licence de Mathématiques, niveau L3

Mathématiques Algorithmiques Élémentaires en C++

Une introduction sous forme de travaux dirigés
avec compléments de cours

Michael.Eisermann@fourier.ujf-grenoble.fr
www-fourier.ujf-grenoble.fr/~eiserm

Version préliminaire du 22 juin 2009
Merci de me signaler toute sorte de correction, critique, ou commentaire.

Table des matières

Références bibliographiques

1. Programmation en C++

Il y a de nombreux livres d'introduction au langage C++. Ceux qui suivent ne sont qu'une indication ; vous en trouverez d'autres dans le rayon « C++ » de votre bibliothèque universitaire.

[1] H. Garreta, *Le langage et la bibliothèque C++*, Édition Ellipses, Paris 2000.
☞ Une introduction concise et très informative.

[2] C. Delannoy, *Programmer en langage C++*, Éditions Eyrolles, Paris 2004.
☞ Très bon livre pour commencer le C++, très pédagogique.

[3] J.R. Hubbard, *Programmer en C++*, Série Schaum, Dunod, Paris 2002.
☞ Une introduction très accessible et riche d'exemples.

[4] B. Stroustroup, *Le langage C++*, CampusPress, Paris 1999 (3e édition), ou plus récent
☞ Le livre de référence, écrit par le concepteur du langage. Complet, incontournable pour les initiés, mais trop riche pour débuter. Inestimable pour ses réflexions philosophiques.

2. Algorithmique générale

Comme pour la programmation, les introductions à l'algorithmique sont nombreuses. Les suivantes m'ont particulièrement plu ; vous en trouverez d'autres dans le rayon « algorithmique ».

[5] R. Séroul, *Informatique pour mathématiciens*, InterEditions, Paris 1995.
☞ Passage en douceur entre mathématiques « statiques » et « dynamiques », c'est-à-dire algorithmique. Destiné aux étudiants en licence, ce livre présente une multitude d'exemples fondamentaux, réconciliant pensée mathématique et programmation. Illustré en langage Pascal.

[6] L. Albert *et al*, *Cours et exercices d'informatique*, Vuibert, Paris 1998.
☞ Une sympathique introduction au monde algorithmique, écrite pour des élèves en classes préparatoires et 1er et 2nd cycles universitaires. Illustré en langage Caml.

[7] R. Sedgewick, *Algorithmes en langage C*, Dunod, Paris 2000.
☞ Introduction générale aux aspects algorithmiques, illustrés en langage C. Ceci inclut une discussion détaillée des méthodes de tri, dont l'auteur est un expert.

[8] D.E. Knuth, *The art of computer programming*, Addison Wesley, 1969.
☞ La référence classique en algorithmique. Bien que la programmation ait beaucoup évoluée entre-temps, cette œuvre monumentale reste d'actualité après une quarantaine d'années.

3. Algèbre algorithmique

Quant à l'algèbre algorithmique je me suis servi des sources suivantes, que je recommande sans aucune retenue. Pour vrai dire, l'objectif de ce cours est atteint s'il donne envie de lire la suite :

[9] P. Naudin, C. Quitté, *Algorithmique algébrique*, Masson, Paris 1992.
☞ Une introduction très complète à l'algorithmique algébrique, illustré en langage Ada. Cette œuvre offre une impressionnante collection d'exercices, la plupart corrigés !

[10] V. Shoup, *A computational introduction to number theory and algebra*, `www.shoup.net`
☞ Une introduction très proche dans l'esprit de la nôtre, mais plus complète. Elle servira parfaitement de suite et approfondissement et va bien avec la *Number Theory Library* (NTL), écrite par Shoup en C++. Les deux sont disponibles sur le web.

[11] J. von zur Gathen, J. Gerhard, *Modern Computer Algebra*, Cambridge Univ. Press, 2003.
☞ Une excellente introduction à l'algorithmique mathématique en vue du calcul formel, qui couvre un vaste terrain et tente de réconcilier théorie et pratique.

[12] H. Cohen, *Computational algebraic number theory*, Springer-Verlag, Heidelberg 1993
☞ Une riche source d'algorithmes en théorie des nombres, bien présentée et détaillée, par un des auteurs du système PARI (calcul en théorie des nombres). Niveau avancé.

[13] A.M. Cohen *et al.*, *Some Tapas of Computer Algebra*, Springer-Verlag, Berlin 1999.
☞ Un tour d'horizon de résultats classiques et d'applications récentes, regroupés en divers sujets, allant de l'élémentaire au plus pointu.

4. Aspects algorithmiques particuliers

[14] P. Ribenboim, *The new book of prime number records*, Springer-Verlag, New York 1996

[15] R. Crandall, C. Pomerance, *Prime numbers — a computational perspective*, Springer-Verlag, New York 2001
☞ L'algorithmique des nombres premiers et la factorisation de grands nombres entiers sont devenues un domaine hautement spécialisé. Les deux livres précédents développent un beau panorama tout en offrant des approfondissements mathématiques.

[16] J.-P. Delahaye, *Le fascinant nombre π*, Pour la Science, Paris 1997
☞ Encore un sujet fascinant, assez spécialisé mais aussi amusant.

5. Mathématiques

Les mathématiques requises sont pour la plupart élémentaires, dans les chapitres plus avancés elles correspondent au niveau de la licence (bac+3). Il existe évidemment de nombreuses œuvres qui présentent les mathématiques à ce niveau, n'en citons que quelques unes :

[17] R.L. Graham, D.E. Knuth, O. Patashnik, *Mathématiques concrètes*, Vuibert, Paris 2003
☞ Une introduction en douceur aux notions mathématiques utilisées dans Knuth, *The art of computer programming*. Elle se veut volontairement concrète, basique et très détaillée.

[18] L. Schwartz, *Algèbre 3ème année*, Dunod, Paris 2003
☞ Introduction à la trilogie groupes–anneaux–corps, cette œuvre récente est bien adaptée à un cours d'algèbre au niveau licence (bac+3).

[19] D. Guin, *Algèbre*, Éditions Belin, Paris 1997
☞ Sympathique introduction à l'algèbre niveau licence/maîtrise ; tome 1 : groupes, anneaux, modules ; tome 2 : corps et théorie de Galois.

[20] M. Artin, *Algebra*, Prentice Hall, 1991
☞ Cette introduction à l'algèbre entame la trilogie groupes–anneaux–corps en partant des exemples concrets, dont elle fournit une riche collection.

6. Analyse numérique

Le calcul numérique n'est que brièvement abordé aux derniers chapitres de ces notes. Les œuvres suivantes peuvent servir d'introduction à l'analyse numérique.

[21] H. Boualem, R. Brouzet, *La planète $\mathbb{R}$. Voyage au pays des nombres réels*, Dunod, Paris 2002
☞ L'objet fondamental pour toute l'analyse est le corps des nombres réels. Ce sympathique bouquin arrive à poser ces fondements d'une manière solide et très agréable à lire.

[22] J.-P. Demailly, *Analyse numérique et équations différentielles*, EDP Sciences, 1996
☞ Issu de l'enseignement en premier cycle, cet ouvrage introductif essaie de faire le pont entre analyse « pure » et « appliquée ».

Quelques symboles fréquemment utilisés

$\mathbb{N}$	l'ensemble des nombres naturels $0, 1, 2, 3, \ldots$
$\mathbb{Z}$	l'anneau des entiers $0, \pm 1, \pm 2, \pm 3, \ldots$
$\mathbb{Q}$	le corps des nombres rationnels
$\mathbb{R}$	le corps des nombres réels
$\mathbb{C}$	le corps des nombres complexes
$\mathbb{Z}[i]$	l'anneau $\mathbb{Z} + i\mathbb{Z} \subset \mathbb{C}$ des entiers de Gauss avec $i^2 = -1$
$\lvert x \rvert$	valeur absolue de x, c'est-à-dire $\lvert x \rvert = x$ pour $x \geq 0$ et $\lvert x \rvert = -x$ pour $x \leq 0$
$\lfloor x \rfloor$	arrondi de $x \in \mathbb{R}$ vers $-\infty$, défini par $\lfloor x \rfloor := \max\{n \in \mathbb{Z} \mid n \leq x\}$
$\lceil x \rceil$	arrondi de $x \in \mathbb{R}$ vers $+\infty$, défini par $\lceil x \rceil := \min\{n \in \mathbb{Z} \mid n \geq x\}$
$[x]$	partie entière de $x \in \mathbb{R}$ défini par $[x] := \lfloor x \rfloor$ si $x \geq 0$ et $[x] := \lceil x \rceil$ si $x \leq 0$
$\lfloor x \rceil$	entier le plus proche de x ; dans le cas ambigu $x = 2n \pm \frac{1}{2}$ on pose $\lfloor x \rceil = 2n$
$\log_b x$	logarithme de x à base b
$\log_2 x$	logarithme de x à base 2
$\ln x$	logarithme de x à base e
$[a, b]$	l'intervalle des nombres réels entre a et b (les extrémités a et b incluses)
$[\![a, b]\!]$	l'intervalle $\{a, a+1, \ldots, b-1, b\}$ des entiers entre a (inclus) et b (inclus)
$[\![a, b[\![$	l'intervalle $\{a, a+1, \ldots, b-1\}$ des entiers entre a (inclus) et b (exclus)
$]\!]a, b]\!]$	l'intervalle $\{a+1, \ldots, b-1, b\}$ des entiers entre a (exclus) et b (inclus)
$]\!]a, b[\![$	l'intervalle $\{a+1, \ldots, b-1, b-1\}$ des entiers entre a (exclus) et b (exclus)
$\mathbb{N}_+, \mathbb{Z}_+$	l'ensemble des entiers positifs $1, 2, 3, \ldots$
$\mathbb{Q}_+$	l'ensemble des nombres rationnels positifs
$\mathbb{R}_+$	l'ensemble des nombres réels positifs
$\mathbb{Z}_n$	l'anneau quotient $\mathbb{Z}/n\mathbb{Z}$, y compris le cas particulier $\mathbb{Z}_0 = \mathbb{Z}/0\mathbb{Z} \cong \mathbb{Z}$
π_n	la projection canonique $\pi_n \colon \mathbb{Z} \to \mathbb{Z}_n$ définie par $a \mapsto \overline{a} = a + n\mathbb{Z}$
$[a]_n$	la classe $\overline{a} = [a]_n = \pi_n(a)$ de l'entier a dans le quotient $\mathbb{Z}/n\mathbb{Z}$
$a \operatorname{div} b$	division euclidienne de $a \in \mathbb{Z}$ par $b \in \mathbb{Z}^*$ avec quotient $q = a \operatorname{div} b \in \mathbb{Z}$ et …
$a \bmod b$	… reste $r = a \bmod b$, définis par $a = qb + r$ et $0 \leq r < \lvert b \rvert$
$v \leftarrow e$	affectation : la variable v prend la valeur de l'expression e.
$a \leftrightarrow b$	échanger les valeurs des variables a et b
1101_{dec}	représentation décimale du nombre $1 \cdot 10^3 + 1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0 = 1101$
1101_{bin}	représentation binaire du nombre $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$
$\operatorname{len} n$	longueur de la représentation binaire de $n \in \mathbb{Z}$ (c'est-à-dire le nombre de bits), $\operatorname{len} n := \min\{\ell \in \mathbb{N} \mid \lvert n \rvert < 2^\ell\}$, donc $\operatorname{len} 0 = 0$, $\operatorname{len} 1 = 1$, $\operatorname{len} 2 = \operatorname{len} 3 = 2$, $\operatorname{len} 4 = \cdots = \operatorname{len} 7 = 3$, $\operatorname{len} 8 = \cdots = \operatorname{len} 15 = 4$, etc.
A^*	le sous-ensemble des éléments non-nuls d'un anneau A
$A^\times$	le groupe multiplicatif des éléments inversibles dans un anneau A
$a \mid b$	a divise b dans l'anneau A en question, càd il existe $q \in A$ tel que $aq = b$
$a \sim b$	a et b sont associés dans A, càd il existe $u \in A^\times$ tel que $ua = b$
A/I	l'anneau quotient d'un anneau A par un idéal $I \subset A$

Partie A

Concepts de base et premières applications

Vous apprécierez davantage la programmation
si vous la comparez à une création littéraire
destinée à être lue.
Donald E. Knuth

CHAPITRE I

Une brève introduction à la programmation en C++

Objectifs

- Apprendre les éléments du langage C++ afin de programmer quelques petits exemples.
- Se familiariser avec quelques concepts fondamentaux de la programmation : variable, type, instruction, condition, boucle, fonction, paramètre, copie vs référence.

Ce premier chapitre sert d'introduction au langage de programmation C++, dans le but de pouvoir mettre en œuvre des algorithmes de calcul. Même si vous n'avez jamais programmé, ne vous effrayez pas : nous commençons par des exemples faciles. Les notions de base présentées ici suffiront pour nos besoins modestes dans ce cours. À long terme il vous sera sans doute utile d'élargir et d'approfondir vos connaissances en programmation : vous êtes vivement invités à consulter le rayon « C++ » de votre bibliothèque universitaire afin d'y trouver l'œuvre qui vous convienne le mieux. Vous pouvez également vous renseigner sur internet, par exemple sur le site `www.cplusplus.com`.

Comment démarrer ? Dans toute la suite nous allons travailler sous le système GNU/Linux. Avant tout, loguez-vous sur une machine et ouvrez une fenêtre `xterm` pour y entrer des commandes. Afin de sauvegarder vos futurs fichiers vous pouvez créer un répertoire de travail, nommé `mae` pour fixer les idées, puis certains sous-répertoires. Il suffit pour ce faire d'utiliser la commande `mkdir` (*make directory*), en l'occurrence en tapant `mkdir mae`. Ensuite vous pouvez vous placer dans votre répertoire `mae` en tapant `cd mae` (*change directory*).

Où trouver les fichiers d'exemples ? Vous pouvez copier la plupart des fichiers d'exemples pour ne pas les retaper ; vous les trouvez tous dans le répertoire `~eiserm/mae`. Vous pouvez créez un lien symbolique (*link*) par `ln -s ~eiserm/mae source`. Vérifier par `ls` (*list*) que vous avez bien créé un lien qui s'appelle `source`. Le lien s'utilise comme un sous-répertoire ; par exemple, `ls source/chap01` montre la liste des fichiers disponibles pour le chapitre 1. Afin de travailler avec ces fichiers, surtout pour les modifier, vous les recopiez dans votre répertoire en tapant `mkdir chap01`, puis `cp -v source/chap01/* chap01`. Il sera utile de suivre la même démarche pour les chapitres à venir.

Sommaire

1. Éditer et compiler des programmes

Placez-vous dans le répertoire souhaité, par exemple en tapant `cd ~/mae`, puis tapez la commande `emacs somme.cc` suivi d'un signe '`&`'. L'éditeur de texte `emacs` s'ouvre alors dans une fenêtre. Ouvrez un fichier, disons « somme.cc » et vous pouvez commencer à taper votre texte.

1.1. Un premier programme. Recopiez le texte suivant en utilisant la touche de tabulation après chaque passage à la ligne, de façon à obtenir une indentation automatique. Tout texte après le symbole '`//`' jusqu'à la fin de la ligne sert de commentaire. (Il sert à votre information uniquement ; ne le retapez pas afin d'économiser du temps.)

Programme I.1 Un programme impeccable `somme+.cc`

```
#include <iostream>           // fichier en-tête iostream pour l'entrée-sortie
using namespace std;          // accès direct aux objets et fonctions standard

int ma_fonction( int n )      // définition d'une fonction à un paramètre n
{                             // cette accolade commence le corps de la fonction
  return n*n;                 // cette instruction calcule et renvoie la valeur n^2
}                             // cette accolade termine le corps de la fonction

int main()                    // définition de la fonction principale
{                             // cette accolade commence le corps de la fonction
  cout << "Calcul de la somme de ma fonction f(k) pour k=a,..,b" << endl;
  cout << "Donnez les valeurs des bornes a et b svp : ";
  int min, max;               // définition de deux variables appelées min et max
  cin >> min >> max;          // lecture des deux valeurs du clavier
  int somme= 0;               // définition et initialisation de la variable somme
  for( int k= min; k <= max; k= k+1 )             // boucle allant de min à max
    {                                             // début du bloc de la boucle
      cout << somme << "+" << ma_fonction(k);     // afficher les deux valeurs
      somme= somme + ma_fonction(k);              // recalculer la somme
      cout << " = " << somme << endl;             // afficher la nouvelle somme
    }                                             // fin du bloc de la boucle
  cout << "La somme vaut " << somme << endl;      // afficher le résultat final
}                             // cette accolade termine le corps de la fonction
```

Pour sauvegarder votre texte cliquez sur `save` dans le menu `file`. Pour lancer la compilation cliquez sur `compile` dans le menu `tools`, puis remplacez `make -k` par la commande `g++ somme.cc` au bas de la fenêtre emacs. La fenêtre se scinde en deux parties ; l'une contient votre programme, l'autre vous montre l'évolution de la compilation et les erreurs éventuelles. En cas de succès, la compilation se termine par le message `Compilation finished at ...` Vous avez obtenu un fichier exécutable, nommé `a.out`, dans votre répertoire de travail. Revenez dans la fenêtre initiale `xterm` et lancez la commande `a.out` (il faut éventuellement taper `./a.out`).

Exercice/P 1.1. Essayez, par exemple, de calculer $\sum k^3$ à la place de $\sum k^2$. Puis, si vous êtes courageux ou impatient, essayez de calculer $\sum k!$. (Pour ce faire, il faudra introduire une boucle pour calculer $n!$ dans `ma_fonction`. Devinez-vous déjà comment le faire ? Si non, veuillez patienter ... et lire la suite.)

Remarque 1.2. Notez qu'il y a toujours *un seul* fichier `a.out` dans votre répertoire de travail. Conserver des exécutables n'est pas toujours une bonne idée : vérifiez la taille de ce fichier en tapant `ls -l`. Si vous voulez conserver un exécutable, vous pouvez le renommer à l'aide de la commande `mv` (*move*), par exemple en tapant `mv a.out somme`, avant de compiler un autre programme dans le même répertoire. De manière alternative, la compilation avec `g++ somme.cc -o somme` compile le fichier source `somme.cc` et produit un exécutable nommé `somme`.

1.2. Un programme erroné. Vous pouvez maintenant éditer votre second programme (volontairement faux afin de provoquer des erreurs de compilation). Cliquez sur `files` puis sur `open` et entrez le nom `faux.cc` du fichier à ouvrir dans la ligne de commande. Tapez alors le texte du programme I.2 et lancez la compilation. Vous obtenez, entre autre, les messages suivants :

```
faux.cc: In function ‘int main()’:
faux.cc:10: ‘cout’ undeclared (first use this function)
faux.cc:10: ‘i’ undeclared (first use this function)
faux.cc:11: ‘endl’ undeclared (first use this function)
faux.cc:11: parse error before ‘int’
faux.cc:12: non-lvalue in assignment
```

Si vous placez la souris sur la ligne `faux.cc:10` et que vous cliquez deux fois avec le bouton du milieu, le curseur se placera automatiquement sur la ligne correspondante du texte source.

Programme I.2 Un programme erroné `faux.cc`

```
 1  int carre( int n )
 2  {
 3    return n*n;
 4  }
 5
 6  int main()
 7  {
 8    for( int p=0; p<10; p++ )
 9      {
10        cout << carre(i) << endl
11          int k;
12        5= p;
13      }
14  }
```

Essayons de rectifier le programme ci-dessus. Le flot `cout` est déclaré par exemple dans le fichier `iostream` : nous devons ajouter la directive `#include <iostream>` en début de fichier. Recompilez, puis ajouter `using namespace std;` pour voir la différence.

Dans la ligne 10, la variable `i` n'est pas déclarée : il convient de remplacer `i` par `p`. L'erreur suivante provient de l'absence du point virgule après l'instruction

`cout << carre(i) << endl`

(Remarquez à ce propos la mauvaise indentation de la ligne suivante).

La dernière erreur vient de ce que le signe ‘`=`’ est le signe d'affectation et non le signe d'égalité. On ne peut affecter de valeur à une constante : on dit que ce n'est pas une valeur à gauche (*left value*). Une fois les erreurs corrigées la compilation se déroule normalement.

1.3. Structure globale d'un programme en C++. Comme on a déjà vu dans le programme I.1, un programme en C++ se compose de certains éléments typiques. Avant de parler des détails dans le paragraphes suivants, donnons un aperçu de sa structure globale :

Variables: Les variables représentent les données et l'état du logiciel. Pendant l'exécution, leurs valeurs subissent certains changements, prescrits par les instructions du programme. Le comportement et la capacité d'une variable sont déterminés par son *type* (voir §2).

☞ La définition d'une variable peut, en C++, être faite n'importe où dans le code (mais, bien entendu, avant l'utilisation). Cela permet de la définir aussi près que possible de l'endroit où elle est utilisée : on améliore ainsi la lisibilité du code.

Instructions: Les actions décrites dans le code source du programme sont nommées *instructions*. Il s'agit par exemple des calculs, des affectations, des opérations d'entrée-sortie, etc. Les instructions conditionnelles, avant tout, seront discutées en §3. Un programme consiste ainsi d'une liste d'instructions, séparées par des point-virgules.

☞ On a souvent intérêt à regrouper plusieurs instructions en un *bloc* en les mettant entre deux accolades ‘{’ et ‘}’. Vous pouvez considérer un bloc comme une seule instruction, dite *complexe* car elle est composée de plusieurs instructions plus élémentaires. Le point-virgule obligatoire après chaque instruction est facultatif après un bloc $\{\ldots\}$.

Fonctions: Pour augmenter la lisibilité d'un logiciel, il est en général indispensable de le découper en sous-programmes, appelés *fonctions*. Il s'agit d'un bloc d'instructions (le *corps* de la fonction) qui porte un nom (figurant à la *tête* de la fonction). On peut ainsi appeler la fonction de n'importe où dans le programme.

☞ Vous pouvez regarder un programme comme une longue liste d'instructions, tout comme un roman, aussi long qu'il soit, consiste une suite de mots. Cependant, à partir d'une certaine taille, il est nécessaire d'introduire plus de structure : des paragraphes, des sections, des chapitres, ... En C++ une façon de ce faire est le regroupement en fonctions (puis en classes et modules).

☞ Une fonction peut avoir des *paramètres* et peut renvoyer une *valeur*, ce qui fait l'objet du §4. Le programme principal est une fonction dont le nom doit être `main`.

Directives: Ce sont des instructions spéciales (précédées du caractère dièse `#`) qui sont traitées avant la compilation. Le préprocesseur modifie le texte source du programme en y incluant, par exemple, le contenu des fichiers en-tête. L'usage des fichiers en-tête est une façon archaïque mais éprouvée pour déclarer les fonctions d'une bibliothèque que l'on utilisera dans le programme. C'est le cas pour le fichier `iostream`, qui déclare l'entrée-sortie standard (voir §5).

Commentaires: Il est souhaitable, voire nécessaire, d'inclure des commentaires pour expliquer le fonctionnement du programme (voir §7). Ces commentaires doivent être compris entre `/*...*/`. On peut aussi commenter une ligne en la faisant débuter par `//`.

Soulignons finalement la fameuse distinction entre *définition* et *déclaration* :

Définitions: Tout objet utilisé (variable, constante, fonction) doit être défini *exactement une fois* dans le programme, ni plus ni moins. La définition d'une variable réserve la mémoire nécessaire ; la définition d'une constante spécifie sa valeur ; la définition d'une fonction génère le code associé.

Déclarations: Tout objet (variable, constante, fonction) doit être déclaré *au moins une fois* avant d'être utilisé dans le programme. Ceci permet au compilateur de connaître déjà le type de l'objet afin d'y faire référence, alors que l'objet lui-même peut être défini plus tard.

Toute définition est aussi une déclaration, mais non réciproquement.

Programme I.3 Déclaration vs définition `affiche.cc`

```
1   #include <iostream>                // directive pour inclure iostream
2   using namespace std;               // accès direct aux fonctions standard
3
4   void affiche( int i );             // déclaration de la fonction affiche
5
6   int main()                         // définition de la fonction main
7   {
8     int n;                           // définition de la variable n
9     cout << "donnez un entier : ";    // première utilisation du flot cout
10    cin >> n;                        // première utilisation de cin et n
11    affiche(n);                      // appel de la fonction affiche
12  }
13
14  void affiche( int i )              // définition de la fonction affiche
15  {
16    cout << "la valeur est " << i << endl;
17  }
```

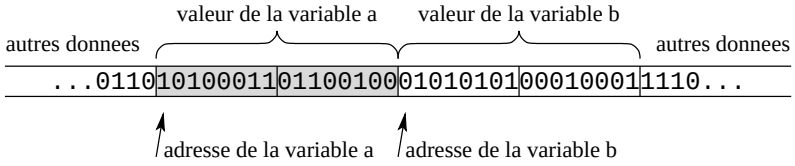
Question/P 1.3. Pourquoi la déclaration de la fonction `affiche()` est-elle nécessaire dans la ligne 4 du programme I.3 ? Pourrait-on s'en passer ? Que se passe-t-il si vous la mettez en commentaire ? Le comparer avec le programme I.1 : où s'effectue la déclaration des fonctions dans ce dernier ?

Remarque 1.4. La déclaration séparée d'une fonction, si elle n'est pas toujours nécessaire dans de petits programmes, est indispensable si l'on veut appeler une fonction avant de l'avoir définie, comme c'est le cas dans le programme I.3. La déclaration séparée devient obligatoire si deux fonctions s'appellent mutuellement ou si l'on veut appeler une fonction dans des modules distincts (voir à ce propos la « programmation modulaire »).

2. Variables et types primitifs

De manière générale, un programme travaille de l'information : pendant l'exécution, cette information est stockée dans la mémoire et subit des changements suivant les instructions du programme. (Relire le programme I.1 pour un exemple.) En C++, l'information est organisée par des *variables*. Chaque variable est d'un certain *type* et porte un *nom* (son *identificateur*).

Mémoire. Sur un ordinateur, l'unité élémentaire de l'information est le *bit* qui ne prend que les deux valeurs 0 et 1. La mémoire de l'ordinateur peut être vue comme une très longue suite de bits. Une séquence de huit bits consécutifs est appelée un *octet* (ou *byte* en anglais) : il peut représenter $2^8 = 256$ valeurs différentes. De manière analogue, deux octets peuvent représenter $2^{16} = 65536$ valeurs, etc. Regardons par exemple comment sont stockées deux variables `a` et `b`, de taille 2 octets disons :



À titre d'illustration, suivons l'humble vie de la variable `a` durant l'exécution de la fonction suivante :

```
int test()
{                       // Au début de ce bloc la variable a n'existe pas encore
  short int a;          // Création de la variable a: allocation de deux octets
  a= 3 + ( 4 * 5 );     // Évaluation d'une expression, puis affectation du résultat
  return a;             // Renvoyer (copier) la valeur de a  à l'instance appelante
}                       // Destruction de la variable a devenue inutile, puis retour
```

La variable `a` est créée lors de sa *définition* ; le compilateur lui réserve alors deux octets à une certaine *adresse* dans la mémoire. Le *nom* « `a` » fait ensuite référence à cette adresse. Les instructions du programme ne changent que la *valeur* stockée à cet endroit, qui, lui, ne bouge plus. Quand la variable n'est plus utilisée, à la sortie du bloc (ou fonction ou programme) où elle est définie, elle est *détruite*. La mémoire occupée jusqu'ici est rendue au recyclage, pour ensuite stocker d'autres données. Dans notre exemple la *valeur* de la variable `a` est renvoyée comme résultat de la fonction `test()` à la fonction appelante :

```
int b;                  // Création de la variable b: allocation de deux octets
b= test();              // Appel de test(), puis affectation (stockage) du résultat
```

☞ Soulignons que pendant toute sa vie, la variable `a` occupe deux octets exactement, jamais plus. La raison est simple : c'est lors de sa création qu'il faut décider de son *type*, disons `short int`, ce qui fixe en particulier la taille. *Le type reste le même durant toute la vie d'une variable !* Dans notre cas, la taille de deux octets limite forcément la capacité à 2^{16} valeurs seulement. L'interprétation des valeurs possibles et les opérations disponibles sont également définies par le type, comme discuté plus bas.

☞ Comme on le voit dans ces exemples, l'affectation suit la syntaxe `variable= expression`. La sémantique est la suivante : d'abord l'expression est évaluée, puis la valeur qui en résulte est affectée à la variable. Pour ceci le résultat de l'expression doit être du même type que la variable, sinon le compilateur essaiera d'effectuer une conversion (implicite) ou émettra un message d'erreur (en cas d'incompatibilité).

Noms. Le nom d'une variable sert à l'identifier : il permet en particulier de trouver l'emplacement dans la mémoire, d'accéder à la valeur stockée et de la modifier.

Un nom peut être constitué de lettres, de chiffres et du caractère blanc souligné « _ » (*underscore* en anglais). Un nom ne doit pas commencer par un chiffre et doit éviter les mots clés réservés du langage.

☞ Notez que le C++ fait la différence entre majuscules et minuscules : les deux noms `toto` et `Toto` ne représentent donc pas la même variable. En C++ les noms de variables peuvent être aussi longs que l'on désire, toutefois le compilateur ne tiendra compte que des 32 premiers caractères.

☞ Vous pouvez donner n'importe quel nom pourvu qu'il respecte les règles précédentes. Veillez tout de même à ce qu'un nom de variable soit en rapport avec l'utilité quelle aura dans votre programme : cela aide à une meilleure compréhension de celui-ci, surtout dans un code source long et complexe (voir §7).

Types. Le C++ est un langage *typé* : le type d'une variable détermine sa capacité (sa taille dans la mémoire et les valeurs possibles) ainsi que son comportement (les opérations disponibles dans le langage).

Une des particularités du C/C++ est que ce langage se veut proche de la machine. Il ne fournit donc que des types directement existant sur le microprocesseur. Cette approche a pour avantage une grande rapidité à l'exécution. Pour en profiter, par contre, il faut comprendre un peu comment travaille la machine.

Le C++ fournit un très petit nombre de *types primitifs*, dont la capacité et le comportement sont prédéfinis par le compilateur. Les types primitifs sont omniprésents dans la programmation en C++. On discutera par la suite chacun de ces types, sa capacité et ses opérations, ainsi que ses limitations :

`bool:` booléen, ne prend que deux valeurs : `true` (=1) et `false` (=0).
`char:` caractère (taille 1 octet typiquement), options : `signed, unsigned`
`int:` nombre entier (taille 2 ou 4 octets typiquement), options : `signed, unsigned`
`short int:` nombre entier (taille 2 octets typiquement), options : `signed, unsigned`
`long int:` nombre entier (taille 4 octets typiquement), options : `signed, unsigned`
`long long int:` nombre entier (taille 8 octets typiquement), options : `signed, unsigned`
`float:` nombre à virgule flottante simple précision (typiquement 4 octets)
`double:` nombre à virgule flottante double précision (typiquement 8 octets)
`long double:` nombre à virgule flottante triple précision (typiquement 12 octets)

☞ Afin d'éviter d'éventuelles déceptions, soyons clairs : les types du C++ sont très loin des concepts idéalisés des mathématiques. Par exemple, le type `int` ne modélise que les « petits entiers ». De manière analogue, le type `float` n'a rien à voir avec les nombres réels : la mémoire finie fixée implique que l'on ne peut stocker que des développements binaires *tronqués* ce qui produit forcément des erreurs d'arrondi.

2.1. Le type int. Le type `int` a été conçu pour modéliser les entiers (plus précisément les « petits entiers », voir la capacité plus bas). En C++ on pourrait écrire par exemple :

```
int p,q;          // définition de deux variables p et q de type int
p= 10;            // affectation d'une valeur (ici une constante)  : p vaut 10
q= 5*(p+1);       // évaluation d'une expression, puis affectation : q vaut 55
p= q;             // affectation (ici copie d'une valeur)          : p vaut 55
q= q+1;           // affectation (ici augmentation par 1)          : q vaut 56
```

Après exécution, les variables `p` et `q` ont les valeurs `55` et `56`, respectivement. Définition et affectation peuvent être combinées. Ainsi les lignes suivantes produisent le même résultat qu'avant :

```
int p(10);        // définition de p avec initialisation à la valeur 10
int q= 5*(p+1);   // définition de q et affectation de la valeur 55
p= q;             // affectation (à noter que p est déjà défini)
q= q+1;           // augmentation (à noter que q est déjà défini)
```

Capacité du type int. À cause des limitations de taille, une variable de type `int` ne peut stocker que des valeurs entre $-2^{31} = -2147483648$ et $2^{31} - 1 = 2147483647$ incluses. Il existe des variantes `short` et `long` et `long long`, dont chacune peut être `signed` (l'option par défaut) ou `unsigned`. Le tableau suivant en donne les capacités correspondantes (implémentées par GNU C++ sur un processeurs à 32 bits ; elles peuvent varier d'un compilateur à un autre, et même d'une machine à une autre).

type	taille		valeurs	plage des valeurs possibles	
	octets	bits	possibles	minimum	maximum
`unsigned short int`	2	16	2^{16}	0	65535
`signed short int`	2	16	2^{16}	-32768	32767
`unsigned int`	4	32	2^{32}	0	4294967295
`signed int`	4	32	2^{32}	-2147483648	2147483647
`unsigned long int`	4	32	2^{32}	0	4294967295
`signed long int`	4	32	2^{32}	-2147483648	2147483647
`unsigned long long int`	8	64	2^{64}	0	18446744073709551615
`signed long long int`	8	64	2^{64}	-9223372036854775808	9223372036854775807

☞ Si l'on veut faire des calculs avec des entiers plus grands, les types primitifs ne suffiront plus. Dans ce cas il faut une solution sur mesure ; on discutera une implémentation au chapitre II.

Comportement du type int. Les valeurs prises par une variable de type `int` sont interprétées comme des nombres entiers, signés ou non, suivant le schéma ci-après :

représentation binaire sur 2 octets = 16 bits	interprétation ‘`unsigned`’	interprétation ‘`signed`’
$0000000000000000_{\text{bin}}$	0_{dec}	0_{dec}
$0000000000000001_{\text{bin}}$	1_{dec}	1_{dec}
$0000000000000010_{\text{bin}}$	2_{dec}	2_{dec}
$0000000000000011_{\text{bin}}$	3_{dec}	3_{dec}
…	…	…
$0111111111111110_{\text{bin}}$	32766_{dec}	32766_{dec}
$0111111111111111_{\text{bin}}$	32767_{dec}	32767_{dec}
$1000000000000000_{\text{bin}}$	32768_{dec}	-32768_{dec}
$1000000000000001_{\text{bin}}$	32769_{dec}	-32767_{dec}
…	…	…
$1111111111111100_{\text{bin}}$	65532_{dec}	-4_{dec}
$1111111111111101_{\text{bin}}$	65533_{dec}	-3_{dec}
$1111111111111110_{\text{bin}}$	65534_{dec}	-2_{dec}
$1111111111111111_{\text{bin}}$	65535_{dec}	-1_{dec}

En C++ ce schéma correspond aux valeurs possibles d’une variable de type `short int`, aussi appelé `short` tout simplement.

Il admet deux variantes :
`unsigned short` allant de 0 à 65535, puis `signed short` allant de -32768 à 32767.

Peut-être la deuxième moitié de l’interprétation `signed` vous semble un peu arbitraire, voire bizarre. Mais elle devient très naturelle si l’on calcule modulo 2^{16} : ainsi $32768 \equiv -32768$, puis $32769 \equiv -32767$, etc … finalement $65534 \equiv -2$ et $65535 \equiv -1$.

Les types `int` et `long int` sont représentés de la même manière mais sur 4 octets, et le type `long long int` sur 8 octets.

Opérateurs arithmétiques. Les `int` admettent les opérateurs de comparaison usuels : égal `==`, différent `!=`, inférieur `<`, inférieur ou égal `<=`, supérieur `>`, supérieur ou égal `>=`. Ce sont des opérateurs binaires qui comparent deux `int` et renvoient la valeur booléenne qui en résulte. Les opérateurs arithmétiques `+`, `-`, `*` ont leur sens usuel pour les `int`, avec une exception importante : si la capacité est dépassée, seul le reste modulo 2^{32} est retenu ! (Voir la représentation interne esquissée plus haut.) Autrement dit :

☞ *Les variables de type `int` avec les opérations `+`, `-`, `*` modélisent non les entiers mais les entiers modulo 2^{32}.* ✍

Quant à la division des `int`, l’opérateur `/` calcule la partie entière du quotient, alors que l’opérateur `%` en calcule le reste. À noter donc que `17/3` vaut `5`, et `17%3` vaut `2`. Si l’on veut calculer la fraction comme une valeur à virgule flottante, expliquée plus bas, il faut d’abord transformer les deux opérants en `float` puis utiliser la division prévue pour les `float`.

Opérateurs mixtes. Au delà de l’opérateur d’affectation `=`, les opérateurs `+=`, `-=`, `*=`, `/=`, `%=` composent l’affectation avec un opérateur arithmétique : `a+=b` équivaut à `a=a+b`, puis `a-=b` équivaut à `a=a-b` etc. Pour afficher les valeurs `0,5,10,...,95,100`, par exemple, on peut ainsi écrire :

```
for( int i=0; i<=100; i+=5 ) cout << i << " ";
```

Dans de telles boucles on utilise également l’incrémentation `++` ou la décrémentation `--`. La version préfixe `++i` équivaut à `i=i+1` ou encore à `i+=1`. La version postfixe `i++` change la variable `i` de la même façon, mais renvoie l’ancienne valeur de `i` comme résultat, tandis que la version préfixe en renvoie la nouvelle. Donc `int i=5; cout << i++; cout << i;` affiche `56`, tandis que la variante `int i=5; cout << ++i; cout << i;` affiche le résultat `66`.

Exercice/P 2.1. En reprenant le programme I.1 comme modèle, écrire un programme qui lit au clavier un nombre naturel *n* et calcule la factorielle *n*!. Jusqu’à quelle valeur de *n* le calcul est-il correct ?

Exercice/P 2.2. Vous pouvez calculer et afficher les puissances successives 2^0, 2^1, 2^2, 2^3 … et ainsi déterminer vous-mêmes la capacité du type `unsigned short int` avec la boucle suivante :

```
for ( unsigned short int entier=1, bits=0; entier!=0; entier*=2, ++bits )
  cout << "2^" << bits << "=" << entier << endl;
```

La boucle s’arrête lorsque la variable `entier` vaut `0`, ce qui arrive après 16 itérations. Expliquez ce phénomène. D’une manière analogue, deviner puis vérifier le résultat du calcul suivant :

```
unsigned int p=0;       cout << "p = " << p << ", p-1 = " << p-1 << endl;
signed   int q=(p-1)/2; cout << "q = " << q << ", q+1 = " << q+1 << endl;
```

Vous pouvez ainsi déterminer *empiriquement* les limites du type `int`, et, après modification, des autres types entiers. Une implémentation de cette idée est disponible dans le fichier `limites.cc`.

2.2. Le type bool. Le type `bool` modélise les variables booléennes. Une telle variable ne peut prendre que deux valeurs : vrai (=1) ou faux (=0). Ainsi deux constantes de type `bool` sont prédéfinies : `true` et `false`. Les opérations de la logique booléenne sont réalisées par la négation `!b`, la conjonction « et » `a&&b`, et la disjonction inclusive « ou » `a||b`, où `a` et `b` sont deux expressions de type `bool` et les opérateurs renvoient à nouveau une valeur de type `bool`. On dispose aussi des comparaisons usuelles, égalité `a==b` et inégalité `a!=b`.

Le type `bool` est particulièrement important pour les instructions conditionnelles, expliquées en §3 plus bas. L'opérateur conditionnel en est une variante : il suit la syntaxe

⟨booléen⟩ `?` ⟨expression⟩ `:` ⟨alternative⟩ `;`

Si ⟨booléen⟩ est vrai, cet opérateur retourne la valeur de ⟨expression⟩, autrement il retourne la valeur de ⟨alternative⟩. Ainsi la fonction

```
int max( int a, int b ) { return ( a>=b ?  a :  b ); }
```

calcule le maximum de `a` et `b`.

Question 2.3. Déterminer les valeurs des variables booléennes après les définitions suivantes :

```
bool a= (  1 < 2 );
bool b= ( -1 > 0 );
bool c= ( a && b );
bool d= ( a || b ) != ( 3 == 4 );
```

Dans les trois premières lignes, les parenthèses ne sont pas nécessaires mais facilitent la lecture. Dans la dernière ligne les parenthèses deviennent importantes. De manière générale, si vous avez le moindre doute sur la priorité de différents opérateurs, mettez des parenthèses qui expriment ce que vous voulez dire.

Remarque 2.4. Dans une expression logique, un entier `i` est implicitement converti en `bool`. Explicitement `bool(i)` vaut `false` si `i` vaut zéro, et `true` sinon. Par exemple `!5||4` vaut `true`. Réciproquement, une valeur booléenne `b` peut être converti en un entier : `int(b)` vaut `0` si `b` vaut `false`, et `int(b)` vaut `1` si `b` vaut `true`. Ceci est fait, par exemple, pour l'affichage : `cout << false << true;` affiche `01`.

2.3. Le type char. Une variable de type `char` contient un caractère, par exemple une des lettres `a...z` ou `A...Z`, mais aussi des chiffres `0...9` ou bien d'autres symboles comme `!"#$%&()*+,-.` etc. A priori, il n'y a rien de numérique dans cette notion. Pourtant, dans un ordinateur, tout caractère est codé par un entier. Il existe ainsi une table de « traduction » entre valeur entière et caractère : le plus souvent c'est la table ASCII (*American Standard Code for Information Interchange*). Le compilateur se charge de faire la transition entre les deux formes d'écriture :

Programme I.4 Conversion entre `int` et `char` — `intchar.cc`

```
1  #include <iostream>                  // déclarer l'entrée-sortie standard
2  using namespace std;                 // accès direct aux fonctions standard
3
4  int main()
5  {
6    int i= 65;                         // 65 est le code ASCII du caractère 'A'
7    char c= char(i);                   // conversion explicite de int en char
8    cout << c << ":" << i << endl;     // affichage du caractère et de son code
9    c= 'B';                            // le caractère 'B' correspond au code 66
10   i= c;                              // conversion implicite de char en int
11   cout << c << ":" << i << endl;     // affichage du caractère et de son code
12   i= 67;                             // 67 est le code ASCII du caractère 'C'
13   cout << char(i) << ":" << i << endl; // conversion et affichage à la fois
14   c= 'D';                            // le caractère 'D' correspond au code 68
15   cout << c << ":" << int(c) << endl;  // conversion et affichage à la fois
16 }
```

Comme on le voit dans cet exemple, la conversion d'une variable `i` de type `int` dans le type `char` est assez naturelle : on écrit simplement `char(i)`. Réciproquement, `int(c)` convertit la variable `c` de `char` en `int`. La conversion est effectuée implicitement si nécessaire.

Capacité et comportement du type char. Le type `char` occupe un octet, ce qui veut dire que le code ASCII ne peut prévoir que 256 caractères (en fait seuls les 128 premiers sont standardisés). De manière interne, le type `char` est représenté comme un entier de taille 1 octet. On peut donc effectuer tous les calculs que nous avons vus plus haut pour les entiers. Par exemple la différence `'c'-'a'` correspond à l'entier 2, et `'a'+7` donne `'h'`, le 8ème caractère de l'alphabet. Logiquement il existe les deux variantes `signed char` et `unsigned char`. Bref, la seule différence entre `char` et `int`, outre la taille, est le comportement à l'entrée-sortie.

Remarquons finalement qu'en C++ une constante littérale de type `char` s'écrit comme `'a'`. Par contre une chaîne de caractères s'écrit comme `"ceci est une chaîne"`. On les a déjà utilisées pour l'affichage des messages, et on les reconsidérera au §6.2.

Exercice/P 2.5. Écrire un programme qui affiche les codes ASCII de 32 à 255 avec les caractères associés dans un joli tableau à 16 colonnes et 14 lignes. Pour le passage à la ligne après un groupe de 16 caractères, vous pouvez regarder le reste modulo 16. (Il s'agit d'un petit bricolage qui illustre un constat fondamental : bien formater la sortie peut prendre un bon moment. Vous trouvez une solution dans le fichier `ascii.cc`.)

Remarque 2.6. Comme le code ASCII fut développé aux États Unis pour les États Unis, les caractères codés par 32…127 ne contiennent que l'alphabet latin standard, sans accents ni caractères spéciaux. Vous voyez le résultat dans l'affichage de l'exercice précédent. Pour coder l'alphabet latin élargi, on se sert de la plage 128…255 (dont l'interprétation semble pourtant moins standardisée). On en a déjà profité dans nos programmes, par exemple dans les chaînes de caractères destinées à l'affichage. Pour d'autres alphabets encore, on peut utiliser *unicode*, un jeu de caractères international standardisé. La taille d'un caractère unicode est forcément plus grande, à savoir 16 bits. Consultez `www.unicode.org` pour en savoir plus.

2.4. Le type float. Les variables du type `float`, aussi appelées *nombres à virgule flottante*, ont été conçues pour modéliser des nombres réels d'une manière approchée, voir plus bas. Le programme I.5 ci-dessous en donne un exemple. Les types à virgule flottante sont toujours signés : il est donc inutile d'utiliser les mots clés `signed` et `unsigned` dans un tel cas.

Programme I.5 Solution numérique d'une équation quadratique `quadratique.cc`

```
1   #include <iostream>             // déclarer l'entrée-sortie standard
2   #include <cmath>                // déclarer les fonctions mathématiques
3   using namespace std;            // accès direct aux fonctions standard
4
5   int main()
6   {
7     cout << "Entrez les coefficients de ax^2+bx+c svp:" << endl;
8     double a,b,c;
9     cout << "a = "; cin >> a;
10    cout << "b = "; cin >> b;
11    cout << "c = "; cin >> c;
12    cout << "L'équation est " << a << "x^2 + " << b << "x + " << c << endl;
13    double d= b*b - 4*a*c;        // NB: Un programme sérieux devrait...
14    double r= sqrt(d);            // ... tester si d n'est pas négatif !
15    double x1= (-b+r)/(2*a);      // ... tester si a est non nul !
16    double x2= (-b-r)/(2*a);      // ... tester si a est non nul !
17    cout << "Voici les deux solutions :" << endl;
18    cout << "x1 = " << x1 << endl << "x2 = " << x2 << endl;
19  }
```

À noter que la représentation des nombres réels n'est possible qu'avec une précision (très) limitée. Dans ce genre de calculs approchés, on aura donc toujours affaire à des erreurs d'arrondi. Les résultats sont à interpréter avec la plus grande prudence !

Exercice/P 2.7. Tester le programme I.5 sur l'exemple $x^2 + 10^{10}x + 1$. Il affichera, sans honte, deux solutions qui sont manifestement fausse. C'est scandaleux ? Certes, mais le C++ n'y peut rien. Comment expliquer ce phénomène ? (Lire la suite.)

Capacité du type float. Comme pour les petits entiers du type `int`, comprendre le comportement du type `float` nécessite (malheureusement) la connaissance de la représentation interne. Le tableau suivant en donne un résumé (valable pour notre compilateur GNU C++) :

type	taille		précision	capacité (grandeurs possibles)	
	mantisse	exposant	(erreur relative)	minimum	maximum
`float`	24 bits	8 bits	$2^{-24} \approx 5 \cdot 10^{-8}$	$2^{-128} \approx 10^{-38}$	$2^{127} \approx 10^{38}$
`double`	53 bits	11 bits	$2^{-53} \approx 1 \cdot 10^{-16}$	$2^{-1024} \approx 10^{-308}$	$2^{1023} \approx 10^{308}$
`long double`	64 bits	15 bits	$2^{-64} \approx 5 \cdot 10^{-20}$	$2^{-16384} \approx 10^{-4932}$	$2^{16383} \approx 10^{4932}$

Exemple 2.8. Comme les arrondis sont un problème très fréquent, nous prenons ici le temps de le discuter un peu plus en détail. Le principe est exemplifié dans le schéma ci-après. Étant donné un nombre réel (par exemple $31,9$) on le transforme d'abord en représentation binaire. Puis la virgule est rendue flottante de sorte que tous les bits '1' soient placés à droite de la virgule ; en compensation on introduit un facteur 2^e convenable. Finalement la mantisse est tronquée à la taille prescrite, disons 24 bits pour une variable de type `float`. Seule ce développement tronqué et l'exposant sont stockés dans la mémoire :

$$\begin{aligned} 31,9_{\text{dec}} &= 11111.11100110011001100110011\ldots_{\text{bin}} \\ &= .11111111001100110011001100110011\ldots_{\text{bin}} \cdot 2^5 \\ &\approx .\underbrace{11111111001100110011001}_{\text{mantisse de 24 bits}}{}_{\text{bin}} \cdot \underbrace{2^5}_{\text{décalage}} \end{aligned}$$

Exercice/M 2.9. Si vous savez le faire, vérifiez la transformation de $31,9$ en binaire. En particulier essayez de vous convaincre que la représentation binaire est périodique, comme indiqué. Sinon, vous pouvez reprendre cet exemple au chapitre IV où l'on discutera de tels changements de base.

Comportement du type float. Une variable de type `float` occupe typiquement 4 octets, soit 32 bits, dont 24 bits pour la mantisse et 8 bits pour l'exposant. La longueur limitée de la mantisse entraîne forcément des erreurs d'arrondi. Même dans notre innocent exemple $31,9$ la représentation binaire est trop longue pour tenir entièrement dans 24 bits (en base 2 elle est périodique donc infinie). Pour en comprendre les effets possibles, souvent inattendus, tester le programme suivant puis expliquer le résultat. (Le vérifier en affichant la différence `somme-1.0`. Quel est le développement binaire de $0,1_{\text{dec}}$?)

Programme I.6 Un résultat surprenant ? `compte.cc`

```
1   #include <iostream>      // déclarer l'entrée-sortie standard
2   using namespace std;     // accès direct aux fonctions standard
3
4   int main()
5   {
6     float somme= 0.0;
7     for ( int i=1; i<=10; ++i ) somme+= 0.1;
8     cout << "La somme vaut " << somme << "." << endl;
9     if ( somme == 1.0 ) cout << "Le compte est bon." << endl;
10    else cout << "Vous êtes accusé de détournement de fonds." << endl;
11  }
```

Les nombres à virgule flottante ne représentent qu'un ensemble fini de nombres de manière exacte ; tous les autres nombres réels ne peuvent qu'être stockés de manière tronquée. Typiquement il faut s'attendre à une erreur relative de $2^{-24} \approx 0.000005\%$, ce qui n'est pas mal, mais tout de même une erreur.

L'usage naïf des nombres à virgule flottante nuit à la correction des résultats.
Consommez avec modération.

Exercice/M 2.10. Esquisser le fonctionnement de l'addition pour les nombres à virgule flottante. Dans la représentation décrite plus haut, que donne l'addition de 2^{60} et 1 ? de 1 et 2^{-60} ?

Exercice/P 2.11. Pour explorer les types `float` et `double` puis `long double` vous pouvez regarder le programme `precision.cc`. Il provoque des erreurs et détermine ainsi la longueur de la mantisse et de l'exposant.

Opérations du type float. Les variables du type `float` admettent les opérateurs de comparaison usuels : égal `==`, différent `!=`, inférieur `<`, inférieur ou égal `<=`, supérieur `>`, supérieur ou égal `>=`. Ce sont des opérateurs binaires qui comparent deux `float` et renvoient la valeur booléenne qui en résulte.

Les opérateurs arithmétiques `+`, `-`, `*`, `/` ont leur sens usuel pour les `float`. De plus, les fonctions usuels sont disponibles après inclusion du fichier en-tête `cmath`. Par exemple : `fabs`, `sqrt`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, etc.

Exercice/P 2.12. Les erreurs d'arrondi peuvent se produire dans les calculs les plus simples, même avec des *nombres rationnels* ! Ainsi le calcul suivant effectué avec les `float` ne donne pas 0, comme il serait mathématiquement correct, mais produit une erreur d'arrondi. Le tester puis expliquer son résultat :

```
float a= 10.0 / 3.0;  // calcul exact : a vaut 10/3
float b=    a - 3.0;  //                b vaut 1/3
float c=    b * 3.0;  //                c vaut 1
float d=    c - 1.0;  //                d vaut 0
cout << d << endl;    // Que vaut le résultat approximatif ?
```

Effectuer aussi le calcul similaire `a=10.0/4.0; b=a-2.0; c=b*2.0; d=c-1.0;` Cette fois-ci le résultat est-il exact ? Comment expliquer ce phénomène ?

Exercice/P 2.13. Incroyable mais vrai : l'addition des `float` n'est même pas associative ! Pour `a=-1e30; b=1e30; c=1.0;` comparer `(a+b)+c` et `a+(b+c)`. Expliquer la différence.

Exercice/P 2.14. Écrire un programme qui lit au clavier un nombre n puis calcule la somme $\sum_{k=1}^{k=n} \frac{1}{k^2}$ de deux manières différentes : d'abord dans le sens des indices croissants, puis dans le sens inverses. Les résultats sont-ils identiques ? Comment expliquer la différence ? Quelle méthode est préférable ? (Pour varier, vous pouvez comparer les résultats calculés avec `float`, `double`, et `long double`. Même exercice avec $\sum_{k=1}^{k=n} \frac{1}{k}$.)

☞ Les erreurs d'arrondi, leur propagation dans les calculs, et les techniques pour éviter le pire, forment toute une théorie faisant partie de l'analyse numérique. Avant d'entamer une programmation numérique sérieuse, consultez un des livres spécialisés à ce sujet, et privilégiez des méthodes éprouvées au lieu de solutions ad hoc.

Conversion de type. Il est parfois nécessaire de changer explicitement le type d'une valeur, on parle alors d'une *conversion* comme déjà vu plus haut. Ainsi une valeur `f` de type `float` peut être converti en entier par `int(f)` ; le résultat est la partie entière de `f`. (À noter qu'il s'agit de *tronquer* et non *d'arrondir* la valeur.) La conversion est implicite dans une affectation comme `int i= f`.

Réciproquement, rappelons que pour deux valeurs `p` et `q` de type `int` l'expression `p/q` donne un entier, à savoir la partie entière du quotient. Si l'on veut calculer une valeur à virgule flottante approchée on écrit explicitement `float(p)/float(q)` ou bien implicitement `float(p)/q`. Dans ce cas la division de deux variables de type `float` donne un résultat de type `float`, comme souhaité.

2.5. Constantes. Comme nous le voyons dans les exemples, l'usage des *constantes littérales* est très fréquent. Une constante littérale du type `char` s'écrit entre apostrophes, par exemple `char c='a'`. Plus généralement, une chaîne de caractères, comme on l'a déjà vue pour la sortie, s'écrit entre guillemets, par exemple `cout << "Bonjour!"`. Les constantes du type `int` s'écrivent comme `83` (décimal) ou bien `0123` (octal) ou bien `0x53` (hexadécimal). Les constantes à virgule flottante sont notées comme `0.23` ou `.23` ou `2.3e-1` (de type `double` par défaut) ou bien `0.23f` (de type `float`).

Constantes nommées. Si une constante apparaît à plusieurs reprises dans le programme, il est souvent utile de définir une *constante nommée*. Pour ce faire le mot-clé `const` peut être ajouté à la déclaration d'un tel objet pour en faire une constante plutôt qu'une variable. Par exemple :

```
const int version=7;      // variable figée de type int
const float pi=3.1415f;   // variable figée de type float
```

De telles constantes peuvent être utilisées comme des variables usuelles, par exemple l'expression `2*pi*rayon` calcule $2\pi r$ avec $\pi = 3,1415$. Évidemment une constante, une fois définie, ne peut plus servir de valeur à gauche, c'est-à-dire on ne peut pas affecter de valeurs : l'instruction `pi=3` produira un message d'erreur du compilateur. (Cette restriction est la raison d'être des constantes.) Pour la même raison, une constante doit être initialisée lors de sa définition, car une affectation ultérieure est impossible. Il est donc impossible de définir une constante `const int c;` sans spécifier sa valeur.

2.6. Références. Une *référence* sur un objet introduit un synonyme, un alias de l'objet référencé. En termes de mémoire, on ne crée pas une copie, mais une référence sur l'adresse de l'objet. En termes d'identificateurs, on déclare un nouveau nom pour un vieil objet. (Si, si, ceci peut être très utile.)

```
int a= 0;          // définition d'une variable nommée a
int & b= a;        // référence nommée b sur la variable a
a= 1; cout << "a=" << a << ", b=" << b << endl;   // a=1, b=1
b= 2; cout << "a=" << a << ", b=" << b << endl;   // a=2, b=2
```

Dans l'exemple précédent on définit d'abord une variable `a` ; le compilateur lui réserve donc une partie de la mémoire à une certaine adresse. Ensuite on crée une référence qui s'appelle `b` ; ce nom fait maintenant référence à la même adresse que le nom `a`. En particulier le compilateur ne crée pas de nouvel objet, on déclare juste un alias.

☞ Le signe `&` est utilisé dans la définition du nom `b` — est dans la définition seule !. Il indique que `b` ne désigne pas une nouvelle variable mais un synonyme de la variable `a`. Après sa définition, le nom `b` s'utilise comme le nom d'une variable ordinaire. Cependant, quand vous testez ce bout de code, vous verrez que tout changement sur `b` est répercuté sur `a`, et réciproquement. Les deux se comportent alors comme un seul objet, adressable sous deux noms différents.

☞ Une référence doit être initialisée lors de sa définition : il est impossible de définir une référence `int &c;` sans spécifier la variable sur laquelle elle fait référence ! De même, une fois l'identification entre `a` et `b` est faite, on ne peut plus la changer : il s'agit effectivement d'un seul et même objet. (Que se passe-t-il par contre si l'on enlève le signe `&` ci-dessus ?)

Références constantes. Tout ce que nous venons de dire s'applique également aux constantes :

```
const int a= 10;   // définition de la constante a à valeur 10
const int &b= a;   // définition d'une référence constante sur a
```

Dans l'exemple suivant, `a` est une variable alors que `b` est constante :

```
int a;             // définition d'une variable a
const int &b= a;   // définition d'une référence constante sur a
```

C'est une situation intéressante, car on ne peut pas changer la valeur en utilisant le nom `b` ; ceci n'est possible que par le nom `a`. On réalise ainsi une restriction d'accès qui est fréquente et très utile pour les paramètres de fonctions (voir §4 plus bas).

```
const int a= 10;   // définition de la constante a à valeur 10
int &b= a;         // définition d'une référence non constante --> erreur
```

Ce dernier exemple est refusé par le compilateur. Pourquoi ?

3. Instructions conditionnelles

Les instructions conditionnelles permettent de ramifier l'exécution des instructions suivant l'état des variables. On parle ainsi de *branchement*, dont une variante est *l'itération*. Dans ce but le C++ offre les instructions suivantes :

3.1. L'instruction if. Branchement suivant un booléen. *Syntaxe :*

`if (` ⟨expression⟩ `)` ⟨instruction⟩`;`
`if (` ⟨expression⟩ `)` ⟨instruction⟩ `else` ⟨alternative⟩`;`

Sémantique : Si l'expression booléenne donne la valeur `true`, alors l'instruction suivante est exécutée. Dans la deuxième forme, la valeur `false` entraîne l'exécution de l'alternative.

Remarque 3.1. La comparaison `==` est facilement confondue avec l'affectation `=`. De telles fautes de frappe sont très courantes et elles s'avèrent en général catastrophiques pour le fonctionnement du programme ! (Heureusement un bon compilateur émettra un avertissement.) Changeons par exemple le programme I.7 en remplaçant la condition `(b==0)` par l'affectation `(b=0)`. Pourquoi le comportement du programme change-t-il radicalement ?

Question 3.2. Où est l'erreur dans le code suivant : `if x<y min=x else min=y;` Comment le rectifier ?

Question 3.3. Où est l'erreur dans le code suivant ? Comment le corriger ?

```
if ( a <= b <= c ) cout << "suite croissante" << endl;
if ( a >= b >= c ) cout << "suite décroissante" << endl;
```

Programme I.7 Division de deux entiers `division.cc`

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6     cout << "\nDonnez deux entiers a et b (avec b non nul) svp : ";
7     int a, b;
8     cin >> a >> b;
9     cout << "Vous avez entré a=" << a << " et b=" << b << endl;
10    if ( b==0 ) cerr << "La division par 0 n'est pas définie\n" << endl;
11    else cout << "Leur quotient entier vaut a/b=" << (a/b) << endl
12              << "et le reste vaut a%b=" << (a%b) << endl << endl;
13  }
```

3.2. L'instruction switch. Branchement suivant un entier. *Syntaxe :*

`switch(` ⟨expression⟩ `)`
{
`case` ⟨constante_1⟩ `:` ⟨instructions_1⟩ `break;`
...
`case` ⟨constante_n⟩ `:` ⟨instructions_n⟩ `break;`
`default:` ⟨instructions par défaut⟩
}

Sémantique : D'abord l'expression est évaluée en un entier. Si cette valeur figure dans la liste des constantes, alors les instructions suivantes sont exécutées. (Elles sont typiquement terminées par `break` pour terminer et sortir du bloc.) Si la valeur n'y figure pas, les instructions qui suivent le mot clé `default` sont exécutées. Le programme I.8 en donne un exemple.

Programme I.8 Évaluation d'une expression arithmétique `switch.cc`

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6     cout << endl << "Donnez une expression binaire svp : ";
7     float x, y;
8     char op;
9     cin >> x >> op >> y;
10    cout << "Calcul de " << x << op << y << " ... ";
11    switch( op )
12      {
13      case '+': cout << "résultat : " << x+y << endl; break;
14      case '-': cout << "résultat : " << x-y << endl; break;
15      case '*': cout << "résultat : " << x*y << endl; break;
16      case '/': if (y!=0) cout << "résultat : " << x/y << endl;
17                else      cerr << "division par 0" << endl;
18                break;
19      default:  cerr << "seules + - * / sont permises" << endl;
20      }
21    cout << "Au revoir.\n" << endl;
22  }
```

☞ On a tout intérêt à remplacer, dans la mesure du possible, une suite de `if` par un `switch` : en effet `switch` est plus lisible (et plus rapide à l'exécution) qu'une longue succession de `if` .

Exercice/P 3.4. Écrire une fonction `jour` qui prend comme paramètres un entier *j* compris entre 1 et 7, disons, et qui affiche le jour de la semaine : « lundi », « mardi », …, « dimanche ».

3.3. Boucle while. Itération selon la syntaxe suivante :

`while (` ⟨condition⟩ `)` ⟨instruction⟩`;`
`do` ⟨instruction⟩ `while (` ⟨condition⟩ `);`

Sémantique : L'instruction est itérée tant que la condition est vraie. Dans la première forme, la condition est testée *avant* l'exécution de l'instruction. Dans la seconde forme, la condition est testée *après* l'exécution de l'instruction, de sorte que l'itération soit exécutée au moins une fois.

3.4. Boucle for. Boucle selon la syntaxe suivante :

`for(` ⟨initialisation⟩`;` ⟨condition⟩`;` ⟨progression⟩ `)` ⟨instruction⟩`;`

Sémantique : Au début de la boucle est effectuée l'initialisation. Tant que la condition est vraie, l'instruction est exécutée, ensuite la progression est effectuée, et la boucle recommence avec le test de la condition. Ainsi la sémantique des boucles suivantes est la même :

`for(` ⟨initialisation⟩`;` ⟨condition⟩`;` ⟨progression⟩ `)` ⟨instruction⟩`;`
⟨initialisation⟩`; while(` ⟨condition⟩ `) {` ⟨instruction⟩`;` ⟨progression⟩`; }`

On pourrait donc se passer de la boucle `for`, mais son utilisation améliore souvent la lisibilité.

☞ Il est possible de définir une variable dans l'initialisation d'une boucle, par exemple
`for( int i=0; i<10; ++i ) cout << i << endl;`
Ceci a été utilisé dans le programme I.1. Par convention, la portée de la variable `i` est restreinte à la boucle.

Exemple 3.5 (Chronométrage d'une boucle)**.** Dans une boucle typique, chaque itération ne prend qu'une fraction d'une seconde, mais elle n'est pas instantanée. Ceci se fait sentir pour un grand nombre d'itérations.

Pour avoir un exemple concret, le programme `chrono.cc` chronomètre le calcul de la somme $1+2+\cdots+n$ par la formule $\frac{1}{2}n(n+1)$, puis par une boucle allant de 1 à n. Si l'on choisit par exemple $n=10^9$, la deuxième variante nécessite l'exécution d'un milliard d'itérations. Afin de vous faire une intuition sur la performance des ordinateurs, lisez puis testez ce petit programme.

Il sera instructif de chronométrer de la même manière vos programmes plus élaborés.

3.5. Les instructions break et continue. Pour gérer les boucles et les itérations il y a deux instructions supplémentaires : `break` sort immédiatement de la boucle alors que `continue` termine l'itération en cours et recommence avec la prochaine itération, en passant par la progression et la condition d'arrêt.

Comme on a vu plus haut, la commande `break` s'utilise très naturellement dans les branchements du type `switch` ; sauf exception elle y est logiquement nécessaire. Dans les boucles l'usage est plus rare, mais peut parfois faciliter la programmation (à consommer avec modération).

Exemple 3.6. Le programme suivant lit une suite d'entiers positif. Tout entier négatif est rejeté, puis la lecture continue. L'entier `0` sert à signaler la fin de la suite.

Programme I.9 Exemple de `break` et `continue` `break.cc`

```
1  #include <iostream>          // déclarer l'entrée-sortie standard
2  using namespace std;         // accès direct aux fonctions standard
3
4  int main()
5  {
6    cout << "Entrez une suite d'entiers positifs (0 pour terminer) :\n";
7    for( int i=1; ; ++i )      // Ici pas de condition d'arrêt
8      {
9        cout << "L'entier no " << i << " : ";
10       int n;                 // définition d'une variable locale
11       cin >> n;              // lecture d'une valeur du clavier
12       if ( n == 0 ) break;   // sortir immédiatement de la boucle
13       if ( n < 0 ) continue; // terminer l'itération en cours
14       cout << "L'entier no " << i << " vaut " << n << endl;
15     }
16   cout << "Au revoir\n" << endl;
17 }
```

4. Fonctions et paramètres

Pour augmenter la lisibilité d'un programme il est en général indispensable de le découper en sous-programmes, appelés *fonctions*. Nous allons voir tout au long de ce cours que cette technique apporte un réel confort dans la vie du programmeur. Ainsi il est indispensable de comprendre ce concept fondamental *en détail*, en particulier le passage des *paramètres* et le renvoie des *résultats*.

4.1. Déclaration et définition d'une fonction. En C++ déclaration et définition d'une fonction sont, respectivement, de la forme

`type_de_retour nom_de_la_fonction(` ⟨liste des paramètres⟩ `);`
`type_de_retour nom_de_la_fonction(` ⟨liste des paramètres⟩ `) {` ⟨instructions⟩ `}`

On précise ainsi au compilateur le type du résultat retourné, le nom de la fonction, ainsi que le nombre des paramètres et leur type. Voici un exemple :

Programme I.10 Exemple d'une fonction avec paramètres

```
int calcul( int x, char op, int y )
{
  switch( op )
    {
    case '+': return x+y;  // effectuer une addition
    case '-': return x-y;  // effectuer une soustraction
    case '*': return x*y;  // effectuer une multiplication
    default:  exit(1);     // erreur (sortie du programme)
    }
}
```

Les paramètres et la valeur renvoyée réalisent la communication entre la fonction et le monde extérieur : ils constituent *l'interface* de la fonction. Dans notre exemple, l'instruction `int r= calcul(2,'+',3);` appelle la fonction `calcul` et déclenche ainsi les actions suivantes :

Passage des paramètres: D'abord les paramètres sont passés à la fonction `calcul`. Pour l'appel `calcul(2,'+',3);` ceci équivaut aux définitions `int x=2; char op='+'; int y=3;` en entrée. Les paramètres reçoivent ainsi leurs valeurs dictées par *l'instance appelante*.

Exécution de la fonction: Ensuite est exécutée la fonction proprement dite. Ici on utilise les paramètres comme des variables usuelles. Leur seule particularité est qu'elles viennent d'être initialisées lors de l'appel de la fonction.

Renvoie du résultat: L'instruction `return` ⟨expression⟩`;` permet de renvoyer une valeur à l'instance appelante. À noter que ceci provoque la sortie immédiate de la fonction.

☞ Une fonction de type `void` ne retourne pas de valeur ; elle se termine donc par `return;` (facultatif) et ne peut pas comporter d'instruction `return` ⟨expression⟩. Toute autre fonction doit obligatoirement renvoyer une valeur du type spécifié.

4.2. Mode de passage des paramètres. Suivant le contexte et les besoins de l'application, les paramètres d'une fonction lui sont passés soit *par copie* soit *par référence*.

Passage par référence: Lors du passage par référence la fonction appelée reçoit une référence sur l'objet original. (Pour les références voir §2.6.) Toute modification de la variable-paramètre est effectuée sur l'original, et persiste après la sortie de la fonction. Ce mode de passage est signalé dans la liste des paramètres par le symbole `&` entre le type et le nom du paramètre.

Passage par copie: Lors du passage par copie, la fonction ne reçoit qu'une copie de l'objet original, et cette copie mène une vie indépendante. Ainsi toute modification de la variable-paramètre est effectuée sur la copie et n'est visible que dans la fonction. À la sortie de la fonction la copie est détruite et ne laisse aucune trace ; l'objet original, quant à lui, reste inchangé. (Ce mode de passage est l'option par défaut en C++.)

Programme I.11 Passage par copie vs passage par référence

```
int puissance4( int x )     // passage par copie
{
  x= x*x;       // calculer d'abord le carré...
  x= x*x;       // ... puis la puissance 4
  return x;     // renvoyer la valeur calculée
}

int incrementer( int& x )  // passage par référence
{
  int y= x;     // stocker la valeur initiale de x
  x= x+1;       // incrémenter la variable x
  return y;     // renvoyer l'ancienne valeur de x
}
```

Exercice/P 4.1. Essayez de comprendre en détail le fonctionnement du programme `passage.cc`. Que prédiriez-vous comme résultat ? Le compiler puis l'exécuter afin de vérifier votre prévision. Modifier le programme afin de varier le passage des paramètres. Quels résultats prédiriez-vous ? Les vérifier.

☞ La valeur renvoyée par une fonction est un objet d'un certain type donné. Si l'on veut qu'une fonction transmette *plusieurs valeurs* comme résultat, sans pour autant définir un type complexe à cet effet, il suffit de lui passer des paramètres par référence (c'est-à-dire modifiables) pour stocker les résultats :

```
void eudiv( const int& a, const int& b, int& q, int& r );
```

On suppose dans cet exemple que `eudiv` effectue une division euclidienne : le quotient et le reste sont stockés dans les variables `q` et `r`, nécessairement passés par référence. (Expliquer pourquoi.)

Question 4.2. Analyser les fonctions suivantes. Quel est leur résultat ? Expliquer la nécessité du signe '`&`'.

```
void swap( int& a, int& b ) { int c=a; a=b; b=c; }
void noswap( int a, int b ) { int c=a; a=b; b=c; }
```

Exercice/P 4.3. Écrire une fonction `trier` qui prend comme paramètres trois entiers a, b, c et les échange de sorte qu'à la fin on ait $a \leq b \leq c$. Choisir un mode de passage convenable : copie ou référence ?

Passage par référence constante. Pour un gros objet il est souvent plus efficace de le passer par référence que par copie. Pensez à une image numérique de plusieurs méga-octets : la création d'une copie nécessite de la mémoire et du temps non négligeables, tandis qu'une référence n'introduit qu'un synonyme sans aucun travail de copie. Cependant, d'éventuels changements seront effectués sur l'original, ce qui peut ou non être souhaitable. Pour cette raison, les paramètres peuvent être déclarés constants :

Passage par référence constante: L'argument est passé par *référence* afin d'éviter une copie inutile, et déclaré `const` pour éviter toute modification. (Le passage par référence en absence de `const` est considéré comme l'intention de modifier la variable.)

Passage par copie constante: Il est possible de passer un objet par copie tout en le déclarant `const`. Ceci combine l'inconvénient d'une copie avec la restriction d'une constante. Bien que théoriquement possible ce mode de passage n'a donc pas d'intérêt pratique.

Surcharge des fonctions. En C++ il est possible de définir des fonctions différentes portant le même nom, à condition qu'elles se distinguent par leurs paramètres. Voici un premier exemple :

```
void eudiv( const int& a, const int& b, int& q );
void eudiv( const int& a, const int& b, int& q, int& r );
```

Dans cet exemple c'est le nombre des paramètres qui diffère : le compilateur saura appeler la bonne fonction pour `eudiv(10,3,quot)` et pour `eudiv(10,3,quot,reste)` car le contexte détermine son choix. De même dans le deuxième exemple, où le type des paramètres diffère :

```
void swap( int& a, int& b ) { int c=a; a=b; b=c; }
void swap( float& a, float& b ) { float c=a; a=b; b=c; }
```

Dans les deux cas, le compilateur saura choisir la bonne fonction à partir des paramètres passés. Ceci n'est plus le cas dans l'exemple suivant, qui lui est erroné :

```
int   sqrt( int a ) { ... }
float sqrt( int a ) { ... }
```

L'intention du programmeur était sans doute de fournir deux variantes pour le calcul d'une racine carrée : la première pour calculer la partie entière, de type `int`, la deuxième pour calculer une valeur approchée, de type `float`. Pour le compilateur, par contre, il est impossible de deviner lors de l'appel `sqrt(2)` laquelle des deux fonction est souhaitée. Il refuse donc de compiler ce code.

Remarque 4.4. Vous pouvez donner des valeurs par défaut aux derniers paramètres d'une fonction. Dans ce cas, si vous appelez cette fonction avec un paramètre manquant, sa valeur par défaut sera utilisée pour l'initialisation. Par exemple, supposons que la fonction `rationnel( int numer, int denom= 1 )` construit le nombre rationnel `numer/denom`. L'appel `rationnel(3,2)` donne $\frac{3}{2}$, alors que l'appel `rationnel(3)` donne $\frac{3}{1} = 3$.

4.3. Les opérateurs. Un opérateur n'est rien d'autre qu'une fonction — avec un nom et une notation particulière. L'avantage de l'écriture sous forme d'opérateur est une meilleure lisibilité : au lieu d'écrire `add(a,b)` on préfère la notation `a+b`, au lieu de `mult(a,b)` on préfère `a*b`, et au lieu de `assign(a,b)` on préfère `a=b`. Dans ce but le C++ offre une vingtaine d'opérateurs, qui prennent un ou deux (voire trois) paramètres. Parmi les opérateurs que nous avons déjà vus, on distingue les opérateurs arithmétiques, les opérateurs d'affectation, d'incrémentation, de comparaison, de logique, et d'entrée-sortie.

Surcharge d'opérateurs. Comme vu plus haut, les opérateurs arithmétiques `+`, `-`, `*`, `/` sont définis pour les types entiers comme `short int`, `int`, `long int` etc, mais aussi pour les types numériques comme `float`, `double`, `long double` etc. Évidemment ces opérateurs sont lourdement surchargés, car ils prennent un sens différent dans chaque instance. À chaque appel le compilateur choisira l'opérateur approprié au vu du type des opérandes données. Dans tous les cas, un tel opérateur prend deux arguments du même type et renvoie un certain résultat du dit type.

Mode de passage. Regardons les opérateurs fournis par le C++ sous l'angle du passage par copie ou par référence. Comme les opérateurs arithmétiques ne changent pas les valeurs de leurs paramètres, on s'attend à une déclaration comme

```
type operator + ( type a, type b );      ou bien
type operator + ( const type& a, const type& b );
```

Il en est de même pour les autres opérateurs arithmétiques `-`, `*`, `/`, `%`. Les opérateurs de comparaison `==`, `!=`, `<`, `<=`, `>`, `>=` pourraient être déclarés de manière similaire :

```
bool operator == ( const type& a, const type& b );
```

L'opérateur d'affectation `=` prend lui aussi deux paramètres du même type, mais cette fois-ci une variable à gauche et une valeur quelconque à droite. Il ne change pas le paramètre à droite, mais il modifie bien sûr le paramètre à gauche en y affectant sa nouvelle valeur :

```
type& operator = ( type& variable, const type& valeur );
```

Il en est de même pour les variantes `+=`, `-=`, `*=`, `/=`, `%=`. Les opérateurs d'incrémentation `++` et de décrémentation `--` prennent un seul paramètre, et bien sûr ils le modifient :

```
type& operator ++ ( type& variable );
```

Ceci correspond à notre fonction `incrementer()` dans le programme I.11.

C'est ainsi que l'on devrait définir ces opérateurs. Bien sûr ils sont déjà prédéfinis par le compilateur pour les types primitifs `int`, `float`, etc. On verra plus tard, quand nous définissons nos propres types en C++, comment implémenter des opérateurs qui suivent les modèles ci-dessus. (Par exemple les grands entiers au chapitre II, les permutations au chapitre VI, ou les polynômes au chapitre XII).

4.4. Portée et visibilité. Une variable n'existe que dans le bloc dans lequel elle est définie, et ceci seulement après sa définition. On dit alors que la variable est *locale* : elle est créée lors de sa définition, et détruite lors de la sortie du bloc. La partie entre création et destruction est appelée la *portée* de la variable. Une variable définie en dehors de tout bloc est dite *globale*. Le programme I.12 en donne un exemple.

Pour un exemple un peu plus complexe, essayez de comprendre le programme I.13. Notez qu'une variable peut en cacher une autre : s'il y a deux variables de même nom et de portées imbriquées, la variable locale a priorité sur la variable globale. Tester le programme pour le vérifier.

Programme I.12 Portée des variables `portee.cc`

```
 1  int g;        // définition de la variable globale g
 2
 3  int main()
 4  {             // début du bloc de la fonction main()
 5    g = 0;      // ok : nous sommes dans la portée de g
 6    i = 1;      // erreur : nous ne sommes pas dans la portée de i
 7    int i;      // la portée de i commence par sa définition
 8    i = 2;      // ok : nous sommes maintenant dans la portée de i
 9    {           // début du sous-bloc
10      g = 3;    // ok : nous sommes toujours dans la portée de g
11      i = 4;    // ok : nous sommes toujours dans la portée de i
12      j = 5;    // erreur : nous ne sommes pas dans la portée de j
13      int j;    // la portée de j commence par sa définition
14      j = 6;    // ok : nous sommes maintenant dans la portée de j
15      i = 7;    // ok : nous sommes toujours dans la portée de i
16      g = 8;    // ok : nous sommes toujours dans la portée de g
17    };          // fin du sous-bloc et fin de la portée de j
18    j = 9;      // erreur : nous ne sommes plus dans la portée de j
19    i = 10;     // ok : nous sommes toujours dans la portée de i
20    g = 11;     // ok : nous sommes toujours dans la portée de g
21  }             // fin du bloc et fin de la portée de i
22
23  void une_autre_fonction()
24  {             // début du bloc de la fonction
25    g = 12;     // ok : nous sommes dans la portée de g
26  }             // fin du bloc de la fonction
```

Programme I.13 Visibilité des variables `visible.cc`

```
 1  #include <iostream>              // déclarer l'entrée-sortie standard
 2  using namespace std;             // accès direct aux fonctions standard
 3
 4  int i=-1, j=-2, a=0, b=0;        // définition globale de i,j,a,b
 5
 6  void affiche( int a, int b )     // définition de la fonction affiche(a,b)
 7  {                                // début du bloc et de la portée de a et b
 8    cout << a << " " << b << endl; // ici les paramètres a et b sont locaux
 9  }                                // fin du bloc et de la portée de a et b
10
11  int main()                       // définition de la fonction main()
12  {                                // début du bloc de la fonction main()
13    affiche(i,j);                  // ici i est globale et j est globale
14    for ( int i=3; i<=5; ++i )     // définition locale de la variable i
15      {                            // début du sous-bloc de la boucle
16        affiche(i,j);              // ici i est locale et j est globale
17        int j=i*i;                 // définition d'une variable locale j
18        affiche(i,j);              // ici i est locale et j est locale
19      };                           // fin du sous-bloc et de i,j locales
20    affiche(i,j);                  // ici i est globale et j est globale
21  }                                // fin du bloc de la fonction main()
```

☞ Il est possible d'adresser la variable globale en faisant précéder son nom de l'opérateur de portée ' : : '. (Remplacer `i` par `::i` dans la ligne 16 ou 17 ou 18, par exemple.)

4.5. La fonction main. La fonction `main` est le point d'entrée de votre programme, elle est donc obligatoire et un peu particulière. Elle aussi peut prendre des paramètres, c'est-à-dire des options dans une ligne de commande : l'instance appelante est le système d'exploitation de votre ordinateur, qui passe ces options à votre logiciel. Ainsi votre programme pourra utiliser une liste de paramètres passée par le système d'exploitation et lui retourner une valeur, dit code de retour, attestant de sa bonne exécution.

Programme I.14 La fonction `main` peut prendre des paramètres `main.cc`

```
1  #include <iostream>
2  using namespace std;
3
4  int main( int nombre_de_parametres, char* liste_des_parametres[] )
5  {
6    for( int i=0; i<nombre_de_parametres; ++i )
7      cout << "paramètre " << i << " = " << liste_des_parametres[i] << endl;
8    return 0;
9  }
```

☞ Comme on le voit dans cet exemple, la fonction `main` reçoit deux paramètres : le premier est le nombre d'options, le deuxième est la liste des options, dont chacun est une chaîne de caractères.

☞ Un programme C/C++ rend toujours un code de retour de type `int`. Le système d'exploitation considérera que le programme s'est bien déroulé si la valeur de retour est nulle (la valeur par défaut). Pour toute autre valeur, le système diagnostiquera une erreur. Ce mécanisme est bien utile lorsque vous utilisez un script shell : ce dernier pourra selon la valeur de retour lancer une action adaptée.

5. Entrée et sortie

5.1. Les flots standard. Un programme doit en général communiquer avec le monde extérieur. Typiquement votre logiciel affichera des données ou des messages sur l'écran, et l'utilisateur entrera des données ou des commandes au clavier. Pour cet effet quatre flots d'entrée-sortie (*input-output streams* en anglais) sont déclarés après inclusion du fichier `iostream` :

Le flot `cin` correspond à l'entrée standard (typiquement le clavier)
Le flot `cout` correspond à la sortie standard (typiquement l'écran).
Le flot `cerr` est utilisé pour envoyer des messages d'erreur (typiquement sur l'écran)
Le flot `clog` est utilisé pour protocoler l'avancement du programme (typiquement sur l'écran)

L'opérateur `<<` permet d'envoyer (écrire) une valeur dans un flot de sortie.
L'opérateur `>>` permet d'extraire (lire) une valeur dans un flot d'entrée.

Envoyer le caractère spécial '`\n`' commence une nouvelle ligne.
Envoyer le caractère spécial '`\r`' retourne au début de la même ligne.
L'instruction `cout << flush;` vide la mémoire tampon et force l'affichage immédiat.
L'instruction `cout << endl;` vide la mémoire tampon et commence une nouvelle ligne.

5.2. Écrire dans un flot de sortie. L'inclusion du fichier en-tête `iomanip` permet de formater les flots. Dans le programme I.15 ci-dessous, par exemple, `setprecision(int n)` définit le nombre de décimales affichées, alors que `setw(int n)` (pour *set width* en anglais) définit la largeur du champ.

Programme I.15 Formater les flots avec `iomanip` `iomanip.cc`

```
1  #include <iostream>    // déclarer l'entrée-sortie standard
2  #include <iomanip>     // manipulateurs pour formater les flots
3  #include <cmath>       // fonctions mathématiques comme sqrt()
4  using namespace std;   // accès direct aux fonctions standard
5
6  int main()
7  {
8    for( int n=0; n<=100; n+=10 )
9      cout << "n=" << setw(3) << n << ", n^2=" << setw(5) << (n*n) << endl;
10   cout << "sqrt(2) = " << setprecision(10) << sqrt(2.0) << endl;
11 }
```

Exercice/P 5.1. Pour augmenter la précision d'un calcul on pourrait être tenté d'afficher plus de chiffres, par exemple comme dans le code suivant :

```
double x= sqrt(2);
cout << setprecision(50) << x << endl;
```

Trouver l'erreur logique dans cette approche. Combien de chiffres sont valables ? Pour simplifier vous pouvez remplacer `sqrt(2)` par `1.0/3.0`. Expliquer ce phénomène.

Exercice/P 5.2. Deviner puis vérifier ce que donnent les instructions suivantes :

```
int i=5; cout << i << i++ << i-1 << endl;
```

Expliquer pourquoi une telle écriture est fortement déconseillée. Trouver une écriture plus claire. ▼

5.3. Lire d'un flot d'entrée. En principe la lecture d'un flot d'entrée est aussi simple que l'écriture dans un flot de sortie. Il y a quand même une différence intrinsèque : lors de l'écriture le logiciel connaît déjà les données à afficher, mais pendant la lecture on ignore ce qui va être lu (logique, non ?). Il faut donc prévoir plusieurs cas, y inclus des entrées erronées.

Exercice/P 5.3. Écrire une fonction qui lit une suite d'entiers positifs terminée par 0 et qui retourne le maximum, le minimum, et la moyenne. Pensez aux cas limites, voire erronés : Que faire avec une entrée négative ? Que faire avec une liste vide, c'est-à-dire de longueur zéro ?

☞ La difficulté de programmer l'entrée-sortie augmente avec la complexité des données. Pour ne pas perdre trop de temps et pour procéder directement au noyau mathématique, nos projets comporteront, comme bout de code initial, une entrée-sortie prête à utiliser. C'est commode mais peu réaliste.

5.4. Écrire et lire dans les fichiers. La communication entre programme et monde extérieure peut se faire par d'autres voies que l'écran ou le clavier. Le programme I.16 ouvre le fichier `lecture.txt` pour lire une liste d'entiers. En parallèle il ouvre le fichier `ecriture.txt` pour écrire les valeurs absolues. Facile, non ? Notons que l'on peut interroger l'état d'un flot par les fonctions `eof()` – *end of file* = fin du flot, `good()` – opération réussie, `fail()` – opération échouée, `bad()` – erreur grave = flot corrompu.

Programme I.16 Lire et écrire dans des fichiers `fichiers.cc`

```
1  #include <iostream>     // déclarer l'entrée-sortie standard
2  #include <fstream>      // manipuler les flots associés aux fichiers
3  using namespace std;    // accès direct aux fonctions standard
4
5  int main()
6  {
7    clog << "J'ouvre le fichier \"lecture.txt\" pour lecture ... ";
8    ifstream entree("lecture.txt");   // ifstream = input file stream
9    if ( !entree ) { clog << "echec." << endl; return 1; };
10   clog << "ok." << endl;
11
12   clog << "J'ouvre le fichier \"ecriture.txt\" pour écriture ... ";
13   ofstream sortie("ecriture.txt");  // ofstream = output file stream
14   if ( !sortie ) { clog << "echec." << endl; return 1; }
15   clog << "ok." << endl;
16
17   int nombre_in, nombre_out;
18   while( !entree.eof() )
19     {
20       entree >> nombre_in;
21       if( entree.eof() ) break;
22       nombre_out= abs(nombre_in);
23       sortie << nombre_out << " ";
24       clog << "J'ai lu " << nombre_in << " --> j'ai écrit " << nombre_out << endl;
25     }
26   clog << "Je ferme les fichiers \"lecture.txt\" et \"ecriture.txt\"." << endl;
27   entree.close(); sortie.close();
28 }
```

Ici le flot `clog` n'est utilisé que pour « protocoler » l'avancement du travail ; on pourrait aussi bien le supprimer. Mis à part ouverture et fermeture, les fichiers s'utilisent comme les flots standard `cin` et `cout`.

On pourrait imaginer que pour `cin` et `cout` l'ouverture s'effectue automatiquement lors du lancement du logiciel, ainsi que la fermeture quand le programme se termine.

Dévier l'entrée-sortie. Le programme I.16 précédent peut être réécrit comme suit :

Programme I.17 Lire et écrire les flots standard `devier.cc`

```
#include <iostream>
using namespace std;

int main()
{
  int nombre_in, nombre_out;
  while( !cin.eof() )
    {
      cin >> nombre_in;
      if( cin.eof() ) break;
      nombre_out= abs(nombre_in);
      cout << nombre_out << " ";
      clog << "J'ai lu " << nombre_in << " --> j'ai écrit " << nombre_out << endl;
    }
}
```

Par défaut `cin` lit du clavier et `cout` écrit sur l'écran. On peut néanmoins dévier ces flots standard en appelant le programme comme suit :

```
a.out < input.txt
```

Ainsi le flot d'entrée `cin` lit le fichier `input.txt`. Si celui-ci contient le texte `+1 -2 +3 -4 +5 -6`, par exemple, alors le programme affiche `1 2 3 4 5 6` sur l'écran. Bien sûr, on peut aussi dévier la sortie :

```
a.out > output.txt
```

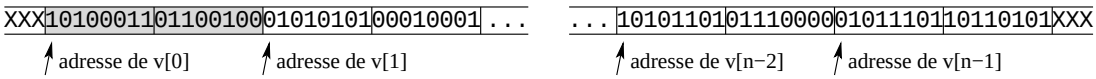
Dans ce cas le flot de sortie `cout` écrit dans le fichier `output.txt`. (Son ancien contenu est écrasé ; à utiliser avec prudence.) On peut finalement dévier entrée et sortie à la fois :

```
a.out < input.txt > output.txt
```

Ainsi notre programme transforme les données du fichier d'entrée `input.txt` et écrit le résultat dans le fichier de sortie `sortie.txt`. Génial, non ?

6. Tableaux

Il arrive fréquemment que l'on veuille stocker une famille de données de même type, par exemple une suite finie de nombres, une liste de noms, d'adresses, etc. Pour cela le C/C++ prévoit comme construction primitive les *tableaux*. Un tableau de longueur n est une famille $v_0, v_1, \ldots, v_{n-1}$ de n variables du même type, indexées par $0, 1, \ldots, n-1$. Sur ordinateur ceci se réalise par n variables stockées à n adresses consécutives dans la mémoire. Pour chacune il faut réserver la mémoire nécessaire (selon le type) :



Au lieu des tableaux primitifs du C/C++, il sera plus commode et plus sûr d'utiliser des implémentations sophistiquées comme les vecteurs ou les chaînes de caractères. Le principe est le même, mais ces classes offrent plus de confort et de fonctionnalité.

6.1. La classe vector. La bibliothèque STL (*Standard Template Library*) fournit la classe générique `vector` avec une fonctionnalité assez naturelle : un vecteur d'entiers de type `int` est défini par

```
vector<int> mon_vecteur;
```

Jusqu'ici c'est un vecteur vide, de longueur 0. Sa taille peut être adaptée par

```
mon_vecteur.resize(10);
```

Pour définir un vecteur et spécifier sa taille, on utilise la définition

```
vector<int> mon_vecteur(10);
```

On détermine la taille par la fonction `mon_vecteur.size()` et on accède à l'élément numéro `i` par `mon_vecteur[i]`. Le programme I.18 illustre d'autres opérations disponibles.

Programme I.18 Exemple d'utilisation de la classe `vector` — `vector.cc`

```
 1  #include <iostream>              // déclarer l'entrée-sortie standard
 2  #include <vector>                // définir la classe générique vector
 3  using namespace std;             // accès direct aux fonctions standard
 4
 5  ostream& operator << ( ostream& out, const vector<int>& vec )
 6  {
 7    out << "( ";                          // parenthèse ouvrante pour faire joli
 8    for ( size_t i=0; i<vec.size(); ++i ) // boucle parcourant les indices
 9      out << vec[i] << " ";               // affichage de la valeur vec[i]
10    out << ")";                          // parenthèse fermante pour faire joli
11    return out;                          // on rend le flot comme il se doit
12  }
13
14  int main()
15  {
16    cout << "\nVoici quelques opérations sur des vecteurs:\n" << endl;
17    vector<int> mon_vecteur(5);    // définir un vecteur de longueur 5
18    cout << "initialisation(5) : " << mon_vecteur << endl;
19    mon_vecteur.resize(20,99);     // rallonger le vecteur en remplissant par 99
20    cout << "adaptation(20,99) : " << mon_vecteur << endl;
21    mon_vecteur.resize(10);        // raccourcir le vecteur à la longueur 10
22    cout << "adaptation(10) :    " << mon_vecteur << endl;
23    for ( int i=0; i<10; ++i )     // boucle parcourant les indices du vecteur
24      mon_vecteur[i]= 10+i;        // affecter une valeur à la place indexée i
25    cout << "affectation :       " << mon_vecteur << endl;
26    vector<int> une_copie;         // définir un deuxième vecteur (encore vide)
27    cout << "un vecteur vide :   " << une_copie << endl;
28    une_copie= mon_vecteur;        // affectation (par recopiage des éléments)
29    cout << "copie du vecteur :  " << une_copie << endl << endl;
30    mon_vecteur.push_back(90);     // rajouter la valeur 90 à la fin
31    cout << "prolongation :      " << mon_vecteur << endl;
32    mon_vecteur.pop_back();        // raccourcir en effaçant la dernière place
33    cout << "troncature :        " << mon_vecteur << endl;
34    mon_vecteur.clear();           // effacer le vecteur tout entier
35    cout << "délétion complète : " << mon_vecteur << endl << endl;
36    cout << "copie retenue :     " << une_copie << endl << endl;
37  }
```

De la même façon on utilise un vecteur de n'importe quel autre type, comme `vector<bool>` ou `vector<char>` ou `vector<double>`. Pour cette raison on appelle la classe `vector` une *classe générique* (ou *template* en anglais, ou *patron de classe*). À noter que la classe `vector` n'est définie qu'après inclusion du fichier en-tête correspondant par la directive `#include <vector>`.

☞ La classe générique `vector` de la STL ne fournit pas d'opérateurs d'entrée-sortie. Ceci n'est pas bien grave : pour afficher un vecteur du type `vector<int>` on pourrait aisément écrire une fonction

```
void affiche( ostream& out, const vector<int>& vec );
```

Nous préférons un *opérateur* qui poursuit le même but. Il est implémenté et utilisé dans le programme I.18. Rappelons à ce propos qu'un opérateur n'est rien d'autre qu'une fonction avec une écriture particulière.

☞ Le fichier `vectorio.cc` implémente l'entrée-sortie des vecteurs d'une manière un peu plus générale. Essayez de comprendre son fonctionnement.

Attention aux indices ! Avant d'utiliser un vecteur on doit obligatoirement spécifier sa taille et ainsi réserver la mémoire nécessaire. Le compilateur ne peut pas deviner quels indices seront utilisés durant l'exécution du programme. Or, en dehors des indices légitimes résérvés on accéderait aux données voisines, qui appartiennent à d'autres variables voire d'autres programmes.

Pour cette raison la lecture d'une case non définie donne des résultats erronés, et l'écriture peut détruire des données irréparablement. Un tel logiciel se compile sans problème, mais lors de l'exécution il s'arrêtera brutalement signalant une *erreur de segmentation*. Cette situation est, hélas, assez fréquente. En particulier on se trompe facilement du dernier indice : un vecteur de taille 10 n'a pas d'élément indexé par 10 !

☞ *Un vecteur de taille n est toujours indexé par* $0, 1, \ldots, n-1$.
Lire un élément d'indice illégitime donne des résultats imprévisibles ;
son écriture risque d'écraser d'importantes données stockées à cet endroit ! ✍

6.2. La classe string. Très souvent, même dans les petits programmes, on doit manipuler les chaînes de caractères. La classe `string` de la bibliothèque standard permet de ce faire avec la plus grande facilité. Elle peut être vue comme une spécialisation des tableaux, avec une fonctionnalité sur mesure pour les chaînes de caractères. Sa déclaration se fait par la directive `#include <string>`.

Une variable du type `string` peut être définie par

```
string s;
```

On peut affecter une constante littérale par

```
s= "un exemple";
```

L'opérateur `+` est surchargé et réalise la concaténation :

```
s= "voici " + s + " !";
```

Après ces instructions `s` contient le texte « `voici un exemple !` ». On accède à la *n*ième lettre par `s[n]` , c'est-à-dire on utilise l'indexation connue des vecteurs. Par exemple, `s[0]` vaut `'v'` , et l'affectation `s[0]= 'V'` redéfinit la première lettre, de sorte que `s` contienne le texte « `Voici un exemple !` ». La classe `string` offre beaucoup d'autres fonctions, dont le programme I.19 ne montre que quelques exemples.

Programme I.19 Exemple d'utilisation de chaînes de caractères `string.cc`

```
#include <iostream>        // déclarer l'entrée-sortie standard
#include <string>          // déclarer la classe string
using namespace std;       // accès direct aux fonctions standard

int main()
{
  cout << "\nBonjour et bienvenue.\nComment t'appelles-tu ? ";
  string nom, s;           // définition de deux chaînes (encore vides)
  cin >> nom;              // lire un mot (délimité par des espaces)
  cout << "Salut " << nom << " !" << endl;
  getline(cin,s);          // effacer le reste de la ligne d'entrée

  string t("Ceci est une chaîne de caractères.");
  cout << t << endl;       // la chaîne de caractères initiale
  cout << string(t,9,14) << t.size() << " caractères.\n";
  cout << t << endl;       // la chaîne de caractères inchangée
  s= t + " On peut y rajouter du texte.";
  cout << s << endl;       // la chaîne de caractères rallongée
  s.insert(54,"ou insérer ou glisser ");
  cout << s << endl;       // la chaîne de caractères après insertion
  s.replace(68,7,"remplacer");
  cout << s << endl;       // la chaîne de caractères après remplacement
  s.erase(65,13);
  cout << s << endl;       // la chaîne de caractères après délétion

  cout << "Retape la permière phrase stp : " << endl;
  getline(cin,s);          // lire toute une ligne, espaces inclus
  cout << "Tu as entré la phrase\n\"" << s << "\"" << endl;
  if ( s == t ) cout << "Sans aucune faute de frappe, bravo !" << endl;
  else cout << "Ceci n'est pas la phrase\n\"" << t << "\"" << endl;
  cout << "première sous-chaîne \"ou\" : pos=" << s.find("ou") << endl;
  cout << "dernière sous-chaîne \"ou\" : pos=" << s.rfind("ou") << endl;
  cout << "première ponctuation : pos=" << s.find_first_of(".,:;!?") << endl;
  cout << "dernière ponctuation : pos=" << s.find_last_of(".,:;!?") << endl;
  cout << "Au revoir, " << nom << ".\n" << endl;
}
```

Conversion entre string et flot. Il sera parfois commode de lire ou d'écrire dans une chaîne de caractères comme on écrit ou lit dans un flot. Ceci est possible comme illustrés dans le programme suivant : `istringstream` prend une chaîne de caractères et en fait un flot d'entrée (*input string stream*), alors que `ostringstream` fournit un flot de sortie qui écrit dans une chaîne de caractères (*output string stream*).

Programme I.20 Conversion entre string et flot `sstream.cc`

```
#include <iostream>            // entrée-sortie standard
#include <sstream>             // conversion entre string et flot
using namespace std;           // accès direct aux fonctions standard

int main()
{
  // Définir la chaîne à analyser, puis lire ses termes
  string t= " test 12.5 + 87.5 ";
  cout << "string initial : \"" << t << "\"" << endl;
  istringstream iss(t);
  string mot; double x,y; char op;
  iss >> mot >> x >> op >> y;

  // Afficher les termes, puis les écrire dans une chaîne de caractères
  cout << mot << endl << x << endl << op << endl << y << endl;
  ostringstream oss;
  oss << " ### " << mot << " ### " << x << " ### " << op << " ### " << y << " ### ";
  string s= oss.str();
  cout << "string terminal : \"" << s << "\"" << endl;
}
```

7. Quelques conseils de rédaction

Le code source d'un bon programme sera très souvent relu, analysé, modifié, corrigé, élargi, amélioré, réutilisé, etc. Pour cette raison une bonne structuration et des commentaires détaillés sont indispensable. Ils seront le meilleur guide pour tout futur lecteur, et ne serait-ce que pour le programmeur lui-même, qui essaie de retrouver le sens de son programme écrit quelques mois auparavant…

⇒ Il convient de bien présenter le code source et de le commenter sans retenue.

⇒ Il ne faut surtout pas croire que le seul lecteur sera le compilateur, au contraire !

⇒ Lors de la rédaction d'un programme il faut écrire pour l'homme et non pour la machine.

Contempler à ce propos le mot savant de Knuth cité au début de ce chapitre.

7.1. Un exemple affreux. Le langage C++ se prête facilement à la programmation de code obscure. Regardons le programme I.21, qui est correct mais humainement incompréhensible. Le compilateur C++ accepte ce code sans aucun problème et en produit un logiciel exécutable.

Programme I.21 Un programme incompréhensible — à éviter ! `obscure.cc`

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

int main()
{
  const int a=10000; int b=52514; vector<int> c(b); int d=0, e=0, f, g, h;
  for( ; (f=b-=14); d=1, cout << setw(4) << setfill('0') << g+e/a << flush )
    for( g=e%=a; (h=--f*2); e/=h ) e=e*f+a*( d ? c[f] : a/5 ), c[f]=e%--h;
}
```

Pouvez-vous deviner ce que fait ce logiciel ? Si l'on vous disait que ces trois lignes calculent 15000 chiffres de π, seriez-vous convaincu ? Feriez-vous confiance en un tel programme ? Seriez-vous capable de calculer ainsi 20000 chiffres de π ? La réponse est non, non, non, et non !

Ces questions correspondent d'ailleurs à des critères de qualité très importantes : l'utilisabilité, la compréhensibilité, la vérifiabilité, et la réutilisablitilé. Pour ces raisons il faut à tout prix éviter de code source comme ci-dessus : le programme I.21 n'est qu'une curiosité pour amuser les étudiants, mais pour tout autre objectif il est inutilisable.

Remarque 7.1. Le code source du programme I.21 est une adaptation en C++ d'un programme de J. Arndt et C. Haenel, donné dans leur livre π *unleashed*, Springer-Verlag, Berlin 2001. Cette méthode de calculer π a été découverte par S. Rabinowitz en 1991. On l'expliquera au chapitre IV, où l'on développe aussi une preuve de sa correction. Vous serez ainsi capable de l'implémenter d'une façon plus digne.

7.2. Le bon usage. Pour obtenir des programmes compréhensibles, même lors de la création de programmes assez brefs, il faut respecter certaines règles. Les remarques suivantes sont loin d'être exhaustives ; elles essaient simplement d'anticiper quelques difficultés et mauvaises habitudes répandues. Même si ce baratin ne vous parle pas trop lors de la première lecture, il serait bon de le relire de temps en temps.

Utiliser des noms de variables et fonctions qui aient un sens. Vous pouvez opter pour des noms courts ou longs. Il est préférable d'utiliser un nom court (une unique lettre par exemple) pour un compteur de boucle, ou un autre usage relativement ponctuel. Par contre si votre variable doit être utilisée en divers points de votre programme, il est alors préférable d'opter pour un nom plus long et surtout plus explicite. En effet, si à la seule lecture de la variable on sait à quoi elle correspond, cela sera une aide précieuse.

Commenter soigneusement tout fichier source. Dans les exemples présentés ici, qui se veulent didactiques, les commentaires expliquent le fonctionnement du langage C++. Dans un programme réel, par contre, de telles explications seraient inappropriées : on supposera que le lecteur envisagé connaît déjà le langage, inutile alors de lui expliquer « ceci est un commentaire » ou « cela est une variable ». Il faut, par contre, expliquer tout ce qui n'est pas immédiatement évident.

Commenter toute fonction et tout bloc sémantique. Pour ne pas encombrer le corps d'une fonction avec de longs commentaires, on peut écrire un commentaire détaillé immédiatement avant la fonction : quelles sont les données d'entrée et de sortie ? les hypothèses ? les garanties ? les limitations ? la méthode utilisée ? Pour des implémentations plus complexes il est avantageux de se référer à une documentation externe, par exemple « voir Knuth §5.3.1 », ou une spécification comme « voir ISO C++ 14882 : §27.3 ». Ensuite des commentaires courts d'une ligne suffiront dans le corps, en renvoyant éventuellement aux plus amples explications précédentes.

Créer une documentation du programme indépendante des fichiers sources. Afin de garder à jour, *dans un seul fichier*, et le code source et la documentation, il existe des systèmes de documentation comme Doxygen (`www.doxygen.org`) ou JavaDoc (`java.sun.com/javadoc`). Ils produisent une documentation à partir des commentaires du code source.

Optimiser la lisibilité du code source. Pour cela il est souhaitable de bien formater le fichier source :

- Ne pas écrire plus d'une instruction par ligne (sauf raisons contraires).
- Mettre les accolades ouvrantes et fermantes seules sur une ligne.
- Indenter correctement (ceci est automatique sous `emacs` avec la touche de tabulation).
- Laisser des lignes blanches afin de regrouper les blocs sémantiques.

☞ Bien sûr des exceptions à ce format sont possibles, et différents styles se sont établis. L'essentiel est d'en choisir un et de l'utiliser d'une manière cohérente. (Dans ces notes je n'ai pas toujours respecté ces règles, dans le souci d'une meilleure mise en page. Ne faites donc pas comme je fais, faites plutôt comme je dis. `;-)`

Relire votre code source. Plusieurs fois. Avec du recul. Pour un projet sérieux il faut soigneusement vérifier chaque fonction : le code écrit traduit-il bien votre intention ? Peut-il être plus clair ? plus efficace ? Vérifier, améliorer, rerédigez en même temps les commentaires qui l'accompagnent.

Concluons par le méta-conseil formulé par Bjarne Stroustrup :

☞ *« Ne suivez les conseils que lorsqu'ils vous semblent logiques.*
Il n'existe pas de substitut à l'intelligence,
pas plus qu'à l'expérience, au bon sens et au bon goût. » ✍

Spécifier chaque fonction et son interface. Quand vous introduisez une fonction, même courte, prenez soin de spécifier pour quel usage elle est faite, quelle donnée elle requiert dans chaque paramètre, et quelle(s) donnée(s) elle renvoie. Mettez le tout dans un joli commentaire :

```
//--------------------------------------------------------------------
// Calcul du coefficient binomial
// Entrée : deux entiers n et k
// Sortie : renvoie le coefficient binomial (n,k)
//
// Pour rappel : le coefficient binomial (n,k) est le nombre
// des sous-ensembles de cardinal k d'un ensemble de cardinal n.
// On suppose donc 0 <= k <= n, et la fonction renvoie 0 sinon.
//
// On utilise ici le type Integer qui modélise les entiers,
// positifs ou négatifs ou nuls, sans restriction de taille.
//--------------------------------------------------------------------
Integer binomial( Integer n, Integer k )
{ ... }
```

L'utilisation de la fonction est ainsi clarifiée. En s'y conformant, le fonctionnement interne sera relativement facile à corriger, adapter, ou optimiser : il suffira de modifier convenablement le code de la fonction (voir chapitre II, §1.3). Ceci est un changement *local* et sans aucun risque, pourvu que la fonction et tous ses utilisateurs respectent la spécification.

☞ Lors de la conception d'un programme non trivial, la spécification et l'interface de chaque fonction doivent être claires et précises dès le début. Il est assez pénible, dans un état avancé du projet, de chercher puis modifier tous les usages d'une fonction, dispersés dans le code. Ce genre de changements *globaux* est coûteux et provoque souvent des erreurs — à éviter.

Introduire des fonctions auxiliaires. Si les mêmes actions apparaissent à plusieurs endroits dans votre programme, songez à en faire une fonction. De manière semblable, si une fonction devient trop longue et difficile à comprendre, essayez de la couper en sous-problèmes de taille modérée (si possible ni trop petits ni trop grands). Ainsi on encapsule des tâches bien précises et les rend accessibles aux vérifications et tests séparés. De plus, munies de commentaires comme ci-dessus, la lecture sera plus aisée qu'un long enchaînement d'instructions sans structure.

Finalement, pour un plus grand programme, il est recommandable de couper le fichier source en plusieurs modules (sous-programmes dans des fichiers séparés) de taille convenable, qui regroupent certaines familles de fonctions appartenant à un thème commun. Ceci a été fait, par exemple, pour la bibliothèque standard, qui est regroupée par thème : `iostream`, `iomanip`, `fstream`, `cmath`, `vector`, `string`, etc.

Introduire des variables auxiliaires. Si votre programme calcule deux fois la même quantité, songez à introduire une variable auxiliaire afin d'y stocker la précieuse valeur intermédiaire. Ceci devient obligatoire si le calcul est coûteux. Voici un mauvais exemple :

```
cout << "le résultat vaut " << calcul_long_et_laborieux(a,b,c) << endl;
if ( calcul_long_et_laborieux(a,b,c) > 0 )
  cout << "ce résultat est positif" << endl;
```

Supposons que votre fonction longue et laborieuse nécessite une heure pour calculer la réponse souhaitée. Alors le programme ci-dessus y mettra *deux* heures ! Il va sans dire que la deuxième heure est un gaspillage injustifiable. Utilisez plutôt la variante suivante :

```
int resultat= calcul_long_et_laborieux(a,b,c);
cout << "le résultat vaut " << resultat << endl;
if ( resultat > 0 ) cout << "ce résultat est positif" << endl;
```

Vérifier les résultats. Ne vous fiez jamais aveuglement à un programme, même si c'est le vôtre ! Chaque fois que cela vous est possible, recoupez les résultats intermédiaires ou finaux de votre programme avec des résultats dont vous êtes sûr. Ces tests de bon sens, souvent peu chers, détectent parfois des erreurs cachées qui autrement se montreraient seulement plus tard, et souvent de manière plus vicieuse.

8. Exercices supplémentaires

8.1. Exercices divers. Les exercices suivants sont à peu près dans l'ordre de difficulté croissante.

Exercice/P 8.1. Écrire un programme qui lit au clavier une durée en secondes et la convertit en jours, heures, minutes et secondes. Même exercice pour la conversion réciproque.

Exercice/P 8.2. Compléter le programme I.5 esquissé plus haut pour qu'il résolve correctement *tous* les cas possibles d'une équation de degré ≤ 2.

Exercice 8.3. Trouver les solutions $a,b,c \in \mathbb{N}$ des équations $ab \equiv 1 \pmod{c}$, $bc \equiv 1 \pmod{a}$, $ac \equiv 1 \pmod{b}$. En quoi l'ordinateur peut-il y être utile ? Trouve-t-il de solutions ? Dans quelle mesure peut-il résoudre le problème ? Formuler une conclusion précise des expériences sur ordinateur.

Exercice/P 8.4. Soit $f : \mathbb{N} \to \mathbb{N}$ définie par $f(n) = n/2$ si n est pair, et $f(n) = 3n+1$ si n est impair. Écrire un programme qui lit un entier positif n et affiche les termes successifs de la suite récurrente définie par $u_0 = n$ et $u_{k+1} = f(u_k)$. Empiriquement on observe qu'une telle suite arrive toujours à la valeur 1, indépendamment de la valeur initiale n. Après quelques essais, on pourrait conjecturer que *toute* valeur initiale $n \in \mathbb{N}$ mène à $f^k(n) = 1$ pour un certain rang k. Cette conjecture, dite *conjecture de Syracuse*, reste toujours ouverte. Voilà une question qui est facile à formuler mais incroyablement difficile à résoudre.

Exercice/P 8.5. Écrire une fonction qui calcule l'angle entre deux coordonnées sphériques. Ajouter un programme qui lit au clavier deux coordonnées géographiques et affiche leur distance sur terre. Par exemple, quelle est la distance entre $45°N, 5°E$ et $50°N, 8°E$?
Remarque. — Vous pouvez développer les formules nécessaires vous-mêmes. Essayez ensuite d'écrire une entrée-sortie confortable, qui vérifie aussi la validité des données entrées.

8.2. Un jeu de devinette.

Exercice/P 8.6. Écrire un programme qui réalise le jeu « trop petit, trop grand » : le logiciel choisit un nombre aléatoire que l'utilisateur doit deviner. L'utilisateur propose un nombre et le programme répond « trouvé » ou « trop petit » ou « trop grand », puis on réitère si nécessaire. ▼

Programme I.22 Début d'un programme de devinette `devinette0.cc`

```
#include <iostream>             // déclarer l'entrée-sortie standard
#include <cstdlib>              // pour déclarer random() et srandom()
#include <ctime>                // pour déclarer time()
using namespace std;            // accès direct aux fonctions standard

int main()
{
  cout << "\nBienvenue au jeu \"trop petit, trop grand\" !" << endl;
  cout << "\nEntrez un nombre maximal svp : ";
  int max; cin >> max;
  srandom( time(NULL) );      // initialiser le générateur aléatoire
  int secret= random()%max+1; // produire un nombre aléatoire entre 1 et max
  cout << "J'ai choisi un nombre secret entre 1 et " << max << "." << endl;

  // *** compléter le programme en implémentant ici les règles du jeu.
  cout << "Mon nombre secret vaut " << secret << "." << endl;
}
```

Exercice/P 8.7. En reprenant le jeu précédent, écrire un programme qui devine un nombre choisi par l'utilisateur. Celui-ci répond, toujours honnêtement, en tapant 't' ou 'p' ou 'g', respectivement.

8.3. Calendrier grégorien.

Exercice/P 8.8. Écrire un programme qui lit au clavier une date du type `j/m/a`, teste sa validité et affiche le jour de la semaine ainsi que le nombre des jours qui se sont écoulés depuis le `01/01/0001`, date aussi commode que fictive. Combien de jours avez-vous aujourd'hui ?

Réciproquement, écrire une fonction `date(int n, int& j, int& m, int& a)` qui calcule la date `j/m/a` à partir de son numéro `n`, cumulatif depuis le `01/01/0001`. (Expliquer le signe `&` dans la liste des paramètres.) Quand fêterez-vous votre 10000ème journiversaire ? ▼

8.4. Vecteurs.

Exercice/P 8.9. Écrire une fonction `inverse` qui prend comme paramètre un vecteur, passé par référence, et qui le remplace par le vecteur dans l'ordre inverse. La tester avec un programme qui lit au clavier une suite d'entiers positifs terminée par 0 et qui affiche la suite inverse. *Indication.* — On pourrait utiliser une fonction `swap(a,b)` qui échange les valeurs des variables `a` et `b`.

Exercice/P 8.10. Écrire une fonction `pascal` qui prend un vecteur $(v_0, v_1, \ldots, v_n)$, passé par référence, et le remplace par le vecteur $(v_0, v_0+v_1, \ldots, v_{n-1}+v_n, v_n)$. *Attention.* — Le nouveau vecteur est plus grand. Vaut-il mieux le calculer de gauche à droite ou de droite à gauche ? Testez votre fonction par un programme qui engendre ainsi successivement les lignes du triangle de Pascal.

Exercice/P 8.11. Pour un vecteur d'entiers $v = (v_1, \ldots, v_n)$ on définit le vecteur dérivé $v' = (v'_1, \ldots, v'_n)$ par $v'_1 = |v_1 - v_n|$ et $v'_i = |v_i - v_{i-1}|$ pour $i = 2, \ldots, n$. Écrire une fonction `deriver` qui prend comme argument un vecteur v, passé par référence, et le remplace par le vecteur v'. Le tester par un programme qui lit au clavier un vecteur v puis affiche les dérivés $v, v', v'', \ldots, v^{(k)}$ jusqu'à $k = 20$ disons.

Exercice/M 8.12. En jouant avec le programme de l'exercice précédent, on tombe sur des observations inattendues : pour un vecteur (v_1, v_2, v_3, v_4) donné, arrive-t-on toujours au vecteur nul ? Même question pour un vecteur de longueur $n = 2^k$. Comment expliquer ce phénomène ? Que se passe-t-il pour un vecteur de longueur $n \neq 2^k$? Tombe-t-on toujours sur le même cycle ? Que vaut la période en fonction de n ?

Exercice 8.13. Voici un challenge : écrire un programme qui lit une liste d'entiers et détermine la médiane. (Définir d'abord ce que c'est.) Il vous faudra éventuellement une méthode de tri, ce qui sera l'objectif du chapitre V. Bien-sûr, la question devient triviale dès que la liste est triée. Voyez-vous une autre méthode ?

8.5. Chaînes de caractères.

Exercice/P 8.14. Écrire une fonction `palindrome` qui prend comme paramètre une chaîne de caractères et détermine s'il s'agit d'un palindrome. Par exemple, « anna » est un palindrome, « nana » ne l'est pas. Quel mode de passage convient le mieux : par copie, par référence ou par référence sur une constante ? Tester votre fonction avec un programme qui lit un mot au clavier puis affiche son verdict.

Exercice/P 8.15. Écrire une fonction `anagramme` qui prend comme paramètres deux chaînes de caractères et qui détermine si les deux forment un anagramme. Par exemple « marie » et « aimer » forment un anagramme, mais non « tester » et « rester ». Quel mode de passage convient le mieux ? Tester votre fonction avec un programme qui lit deux mots au clavier puis affiche s'il s'agit d'un anagramme ou non.

Exercice/P 8.16. Pour un minimum de sécurité on exige qu'un mot de passe ait entre 6 et 8 caractères, qu'il contienne au moins une lettre et au moins un chiffre. Écrire une fonction qui teste si ces conditions sont vérifiées. (Si vous voulez, rajoutez qu'au moins un des caractères soit ni lettre ni chiffre.) Écrire un programme qui lit au clavier un mot de passe proposé par l'utilisateur et qui lui rappelle les règles si nécessaire.

Exercice/P 8.17. Écrire une fonction `minuscule` qui prend comme paramètre une chaîne de caractères, passée par référence, et qui convertit tout en lettres minuscules. Écrire une fonction analogue `majuscule`.

8.6. Topologie du plan. Voici un challenge plus osé, si vraiment vous vous ennuyez. Cet exercice vous propose d'analyser puis de programmer quelques questions de la topologie du plan. Tout sera affine par morceaux pour éviter d'éventuelles pathologies sauvages, mais surtout pour être accessible au calcul sur ordinateur. Malgré son apparence élémentaire, les questions mathématiques soulevées sont assez intéressantes.

Deux points $p, q \in \mathbb{R}^2$ définissent un *segment* $[p,q] \subset \mathbb{R}^2$ que l'on peut paramétrer par la fonction $s\colon [0,1] \to \mathbb{R}^2$, $s(t) = (1-t)p + tq$. Plus généralement, toute famille $C = (p_0, p_1, \ldots, p_n)$ de points $p_k \in \mathbb{R}^2$ spécifie une *courbe polygonale* $c\colon [0,n] \to \mathbb{R}^2$ définie par $c(k+t) = (1-t)p_k + tp_{k+1}$ pour $k = 0, 1, \ldots, n-1$

et $0 \leq t \leq 1$. C'est une application continue, affine sur chaque intervalle $[k, k+1]$, réliant les points donnés $c(k) = p_k$. Son image est donc la réunion des segments $[p_0, p_1]$, $[p_1, p_2]$, …, $[p_{n-1}, p_n]$. Dans la suite on supposera que la courbe c est *fermée* dans le sens que le point de départ $c(0) = p_0$ et le point d'arrivée $c(n) = p_n$ coïncident.

Vous pouvez imaginez les fonctions C++ suivantes dans un logiciel de dessin. Pour simplifier nous n'allons regarder que la partie calculatoire. La liste C peut être implémentée comme un vecteur de longueur $2n+2$, à coefficients entiers de type `int`, ou un type flottant si vous ne craignez pas des erreurs d'arrondi. Pour l'entrée-sortie des vecteurs, vous pouvez vous servir du fichier `vectorio.cc`.

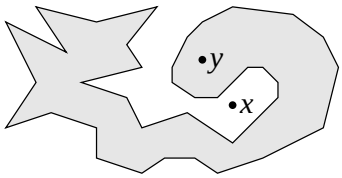


FIG. 1. Un polygone dans le plan : qui est « in », qui est « out » ?

Voici quelques tâches que l'on voudrait implémenter ; vous pouvez en rajouter d'autres :

(1) Déterminer si le polygone est simple, c'est-à-dire les segments ne se recoupent pas.

(2) Déterminer la longueur du polygone, puis l'aire de la région qu'il borde.

(3) Déterminer si le polygone est convex (ou étoilé, mais c'est plus difficile).

(4) Déterminer si globalement le parcours du polygone tourne à gauche ou à droite.

(5) Déterminer si un point donné $x \in \mathbb{R}^2$ est à l'intérieur ou à l'extérieur du polygone.

Pour ces questions il faudra d'abord préciser soigneusement ce que cela veut dire. Ensuite la solution mathématique n'est pas toujours évidente, mais peut-être vous y trouvez un certain plaisir de rechercher. Soyez assuré, il existe des solutions élégantes et efficaces…

Quelques réponses et indications

[Exercice 5.2, affichage tordu] À la surprise générale, les instructions

```
int i=5; cout << i << i++ << i-1 << endl;
```

affichent `654` et non `555`. Pour comprendre ce résultat, il faut savoir que l'opérateur de sortie est évalué de droite à gauche, donc contrairement au sens de lecture. (Ne me démandez pas pourquoi ; c'est juste une convention.) Il renvoie comme résultat une référence sur le flot de sortie, ce qui permet d'enchaîner plusieurs opérateurs. Le parenthésage implicite équivaut à l'écriture explicite

```
int i=5; (((cout << i) << i++) << i-1) << endl;
```

Cette écriture, bien que plus précise, reste peu lisible. Il est préférable de la couper en plusieurs instructions, dont chacune est facilement compréhensible.

[Exercices 8.6 et 8.7, devinette « trop petit, trop grand »] Ce jeu a deux facettes intéressantes : quant à l'apprentissage du C++ il oblige à réfléchir sur la communication entre utilisateur et logiciel. Quant à l'aspect algorithmique, c'est une préparation ludique à la recherche dichotomique (discutée au chapitre V). Si vous avez le temps, ne vous privez pas de ce plaisir ; sinon, vous trouvez une proposition de solution dans les programmes `devinette.cc` et `deviner.cc`.

[Exercices 8.8, calendrier grégorien] Cet exercice nécessite un peu de préparation. Précisons que l'on applique ici uniformément les règles du calendrier grégorien, bien qu'elles ne prissent effet que le 15 octobre 1582. Tout d'abord, comment déterminer si une année donnée est bissextile ? (C'est cette règle raffinée qui est due au Pape Grégoire, d'où le nom.) Ensuite, comment calculer le nombre des jours entre le `01/01/0001` et le `01/01/a`, début de l'année courante ? (Pour ceci il faut appliquer la règle des années bissextiles. À titre d'exemple, si le `01/01/0001` est le jour numéro `1`, alors le `01/01/2001` est le jour numéro `730486`.) Finalement, comment calculer le nombre des jours écoulés pendant l'année courante, c'est-à-dire du `01/01/a` au `j/m/a` ?

On pourra écrire une fonction `int numero( int jour, int mois, int annee )` qui réalise ce calcul et renvoie `0` si la date n'est pas valide. Pour le calcul réciproque on pourra écrire une fonction `void date( int num, int& jour, int& mois, int& annee )`. Si après réflexion vous ne trouvez pas de meilleure solution, vous pouvez regarder le fichier `gregorien.cc`. Bonne lecture !

Rien n'est fait
quand il reste à faire.

PROJET I

Tester la conjecture d'Euler concernant $x^4+y^4+z^4=w^4$

Objectifs

- Résoudre une question mathématique par une énumération exhaustive.
- Reconnaître, puis contourner, des problèmes typiques des types `int` et `double`.
- Comprendre les limitations mathématiques et informatiques inhérentes d'une telle approche.

Le problème et son histoire. Comme vous savez, l'équation de Pythagore $x^2+y^2=z^2$ admet une infinité de solutions $(x,y,z)\in\mathbb{Z}_+^3$ telles que x,y,z soient premiers entre eux. La plus petite est $(3,4,5)$, puis on a $(5,12,13)$, et on sait même produire toutes les solutions de manière systématique.

En 1769 Euler montra que l'équation $x^3+y^3=z^3$, par contre, n'admet aucune solution $(x,y,z)\in\mathbb{Z}_+^3$. Il s'agit du premier cas du grand théorème de Fermat, qui dit que $x^n+y^n=z^n$ avec $n\geq 3$ n'admet pas de solutions $(x,y,z)\in\mathbb{Z}_+^3$. Après sa découverte, Euler conjectura que $x^4+y^4+z^4=w^4$ n'admet pas de solutions $(x,y,z,w)\in\mathbb{Z}_+^4$, et plus généralement que $z_1^n+z_2^n+\cdots+z_{n-1}^n=z_n^n$ avec $n\geq 3$ n'admet pas de solutions dans $\mathbb{Z}_+^n$. Il venait d'établir le cas $n=3$. Il a fallu deux siècles environ pour en connaître la réponse pour $n=5$ (L.J. Lander et T.R. Parkin en 1966), puis pour $n=4$ (N.D. Elkies et R. Frye en 1988).

☞ Pour ne pas gâcher le plaisir de la découverte, ne cherchez pas tout de suite la réponse sur internet.

1. Préparation : calcul d'une racine

Afin de programmer ce problème, il sera utile de disposer des fonctions $a\mapsto a^4$ et $a\mapsto\lfloor\sqrt[4]{a}\rfloor$. Le type en C++ à utiliser pour les entiers sera nommé `Integer` dans la suite. Pour introduire cet alias en C++, il suffit d'écrire `typedef int Integer;` au début de votre programme. Ceci définit le type `Integer` comme synonyme avec `int`. Vous n'utilisez ensuite que le type `Integer` ; si jamais vous voulez changer de `int` à `long long int` il suffit de mettre à jour la ligne `typedef`.

Exercice/P 1.1. Écrire une fonction `Integer puissance4( const Integer& a )` qui calcule a^4 avec deux multiplications seulement. Pour quelle plage de paramètres votre fonction sera-t-elle correcte ? Expliquer le signe '`&`' devant le paramètre. Quelles alternatives conviennent ici ?

Réciproquement on cherche une fonction `racine4` qui calcule $\lfloor\sqrt[4]{a}\rfloor$, c'est-à-dire l'unique nombre naturel r tel que $r^4\leq a<(r+1)^4$. Pour ce faire deux méthodes naïves viennent à l'esprit : la première est correcte mais lente, la deuxième est rapide mais fausse :

Exercice/P 1.2. Écrire une fonction `Integer racine4lente_mais_correcte( const Integer& a )` qui calcule $\lfloor\sqrt[4]{a}\rfloor$ par une boucle parcourant $r=0,1,2,\ldots$. Justifier votre fonction dans les commentaires. À quelle plage de paramètres s'applique-t-elle ? Expliquer en quoi elle est inefficace quand r est grand. (Revoir le chronométrage de l'exemple 3.5. On discutera une nette amélioration au projet III.)

Exercice/P 1.3. Dans un monde idéal, sans erreur d'arrondi, on utiliserait sans hésitation

```
Integer racine4rapide_mais_fausse( const Integer& a )
  { return Integer( exp( log(double(a))/4 ) ); }
```

Vérifier les résultats de `racine4rapide_mais_fausse( puissance4(a) )` pour quelques petites valeurs de `a`. Lesquels sont corrects ? Expliquez ce phénomène. (Revoir §2.4.) Pourquoi et dans quels cas une petite erreur d'arrondi peut-elle entraîner une grosse erreur dans le résultat final ?

Comme la valeur cherchée est un entier, il est hors de question d'accepter une quelconque imprécision dans le résultat : on exige la valeur exacte ! Voici une façon de s'en tirer :

Exercice/P 1.4. En tenant compte des deux exercices précédents, écrire une fonction `racine4` qui soit à la fois *correcte* et *efficace* et *claire*. À partir de la valeur initiale `r = racine4rapide_mais_fausse(a)`, on corrige `r`, le cas échéant, afin de *garantir* l'inégalité $r^4 \leq a < (r+1)^4$: tant que r est trop petit on l'augmente, tant que r est trop grand on le diminue. Implémentez cette idée, et justifiez la correction dans les commentaires. À quelle plage de paramètres votre fonction s'applique-t-elle ?

2. Énumération exhaustive

On se propose de tester la conjecture d'Euler dans la mesure du possible.

Exercice/M 2.1 (préparation)**.** D'abord pour $n=4$ on essaiera une recherche exhaustive de toutes les solutions $z_1^4+z_2^4+z_3^4=z_4^4$ avec la restriction $1 \leq z_1 \leq z_2 \leq z_3 \leq N$. Jusqu'à quel N peut-on aller avec le type `int` ? `long long int` ? Montrer que le nombre des cas à traiter est $\binom{N+2}{3}$. Est-ce un polynôme en N ? de quel degré ? Est-il réaliste de chercher toutes les solutions jusqu'à $N=10^2$? $N=10^3$? $N=10^4$? $N=10^5$? $N=10^6$? Spécifier une borne N qui vous semble raisonnable.

Exercice/P 2.2 (implémentation)**.** Écrire une fonction `void euler4( Integer min, Integer max )` qui affiche toutes les solutions $z_1^4+z_2^4+z_3^4=z_4^4$ avec $1 \leq z_1 \leq z_2 \leq z_3$ et $\text{min} \leq z_3 \leq \text{max}$. Naïvement on pourrait penser à 4 boucles imbriquées, mais la fonction `racine4` permet de se ramener à 3 boucles. (Le projet V expliquera comment les réduire à 2 boucles seulement en utilisant des méthodes de tri.)

☞ *Étapes intermédiaires :* Il sera utile que le programme affiche de temps en temps l'avancement du travail. (Pas trop souvent car l'affichage, lui aussi, prend du temps. . ..) On pourrait ainsi écrire

```
cout << "en train de tester " << ... << "\r" << flush;
```

De même il sera intéressant d'afficher toute solution dès qu'elle est trouvée :

```
cout << "solution trouvée :  " << ... << endl;
```

L'affichage de ces messages permettra, le cas échéant, d'interrompre le programme avec la touche `CTRL c` sans perdre pour autant toute l'information.

Exercice/P 2.3 (généralisation)**.** Refaire le développement précédent pour l'équation $z_1^5+z_2^5+z_3^5+z_4^5=z_5^5$ avec la restriction $1 \leq z_1 \leq z_2 \leq z_3 \leq z_4 \leq N$. Jusqu'à quel N peut-on aller avec le type `int` ? `long long int` ? Combien de cas doivent être traités ? Est-ce un polynôme en N ? de quel degré ? Cette approche est-elle réaliste pour $N=300$? $N=1000$? $N=3000$? $N=10000$? Écrire des fonctions `puissance5` et `racine5` puis une fonction `euler5`.

Exercice/P 2.4 (consolidation)**.** Quand votre programme semble marcher, vérifiez à nouveau sa correction logique (non seulement syntaxique : c'est déjà fait par le compilateur). Explicitez dans les commentaires une spécification pour chaque fonction, en précisant à quelle plage de paramètres elle s'applique. Essayez de justifier que chaque fonction satisfait sa spécification, en remontant de l'élémentaire au complexe (*bottom-up*). Certifiez-vous finalement que votre logiciel est 100% fiable ?

☞ *Rédaction finale :* Après s'être convaincu de sa correction, nettoyer le code source et rédiger la version finale des commentaires. Commencer par quelques lignes de commentaires comportant le nom du programme, l'auteur, la date, ainsi qu'une brève description. Essayer de produire un code qui soit à la fois correct et efficace, concis et compréhensible. Relire à ce propos les conseils du §7.

Exercice/P 2.5 (conclusion)**.** Exécutez la version finale de votre logiciel et rédigez-en un protocole. Combien de temps prend-il ? Quel en est le résultat ? Formulez soigneusement une conclusion : que peut-on dire des conjectures d'Euler ? Quelles questions sont laissées en suspense ?

☞ *Compilation finale :* Afin d'optimiser un peu, compilez avec `g++ -O3 -static`.

- L'option `-O3` demande au compilateur d'optimiser la traduction en langage machine : la compilation sera plus complexe et plus lente, mais l'exécution sera un peu plus rapide.
- L'option `-static` fait inclure les bibliothèques d'une manière dite « statique » : une copie est collée à votre logiciel, ce qui augmente sa taille mais l'accélère un peu.

Pour tester empiriquement ces différentes options, vous pouvez comparer la taille et la rapidité des logiciels qui en résultent. Pour savoir plus sur les options d'optimisation, consultez le manuel en ligne de g++ en tapant `info gcc Invoking Optimize` dans une fenêtre xterm.

Die ganzen Zahlen hat der liebe Gott gemacht,
alles andere ist Menschenwerk.
Leopold Kronecker

CHAPITRE II

Implémentation de grands entiers en C++

Objectifs

- Reconsidérer les algorithmes d'arithmétique connus de l'école.
- Comprendre leur formulation en C++, nécessairement très détaillée.
- Expérimenter empiriquement les premiers phénomènes de complexité.

La problématique. Très souvent en programmation, les types primitifs ne modélisent pas ce que l'on veut. Par exemple le type `int` ne représente que les « petits » entiers : sa mémoire étant fixée à 4 octets (32 bits), la plage des valeurs possibles (avec signe) va de -2^{31} à $2^{31}-1$, ce qui correspond à l'intervalle $[\![-2147483648, 2147483647]\!]$. Pour illustration, reprenons un exemple déjà rencontré au chapitre I :

Exemple 0.1. On veut écrire un programme qui affiche les valeurs $n!$ pour $n = 1, 2, \ldots, 50$.
À titre d'exemple, on souhaite que le programme suivant calcule correctement la factorielle :

```
Integer factorielle= 1;
for( Integer facteur= 1; facteur <= 50; ++facteur )
{
  factorielle*= facteur;
  cout << "La factorielle " << facteur << "! vaut " << factorielle << endl;
}
```

Comme on a vu au chapitre I, les types primitifs du C++ n'y suffisent pas, et l'utilisation naïve du type `int` créera des résultats catastrophiques. Pour résoudre ce genre de problème assez fréquent, on est mené à construire des nouveaux types, traditionnellement appelés *classes*, qui sont mieux adaptés au problème. Dans notre cas on voudrait disposer d'une classe, appelée `Integer`, qui implémente les grands entiers avec une utilisation intuitive.

L'approche générale. Les types primitifs n'admettent qu'une allocation de mémoire « statique » (c'est-à-dire de taille fixée d'avance), ce qui restreint forcément la plage des valeurs. Pour les grands entiers il est avantageux d'allouer la mémoire de manière « dynamique », en adaptant la taille d'un tableau au cours du programme selon les besoins du calcul. On discutera au §1 une telle solution « faite maison », modélisée directement sur la numération décimale, et le §2 discute brièvement des questions de complexité.

Le chapitre III présente une solution « professionnelle », issue de la bibliothèque GMP. N'importe quelle des deux implémentations permettra de résoudre le problème de l'exemple 0.1 d'une manière facile est naturelle ; elles se distinguent seulement par leur niveau d'optimisation : une journée de programmation pour notre classe `Naturel` contre plusieurs années de développement pour la bibliothèque `GMP`.

Sommaire

1. Une implémentation « faite maison » des nombres naturels

1.1. Numération décimale à position. Comme le C++ ne prévoit pas de type primitif pour les grands nombres entiers, nous allons implémenter nous-mêmes un tel type, appelé `Naturel`, dans le fichier `naturel.cc`. C'est un excellent exercice de programmation. Si cependant vous êtes impatient, il suffira de survoler ce chapitre pour passer directement au chapitre III qui prépare l'usage pratique.

Avertissement. — Le seul but de notre implémentation est d'illustrer le principe.
Elle restera dilettante dans le sens qu'aucune optimisation sérieuse ne sera faite.
À part cette négligence, toute autre implémentation suivrait une approche similaire.

On reprend ici une idée vielle de 2000 ans, qui est aussi simple que géniale : la numération décimale à position. Afin de réaliser ce programme en C++, on utilisera un *tableau* pour stocker une suite de chiffres. (Voir le chap. I, §6.1 pour l'utilisation des tableaux sous forme de la classe générique `vector`.) Ainsi notre implémentation sera une traduction directe des algorithmes scolaires bien connus.

À noter toutefois qu'une telle implémentation doit être très précise : en particulier on doit manipuler les vecteurs avec soin, réserver toujours de la mémoire de taille suffisante, puis rallonger ou raccourcir le cas échéant. On discutera tour à tour les différentes fonctions de base, commençant par la représentation des données et l'entrée-sortie, ensuite l'incrémentation/decrémentation et la comparaison, puis l'addition/soustraction, la multiplication, et finalement la division euclidienne.

Rappel de notation. Pour l'axiomatique des nombres naturels et leurs principales propriétés, nous renvoyons à l'annexe de ce chapitre. En voici un résultat fondamental :

Définition 1.1. Nous fixons un nombre naturel $b \geq 2$, que nous appelons *la base*. (Dans tout ce paragraphe on pourra choisir $b = 10$ pour fixer les idées.) Par rapport à la base b, toute suite finie $(a_n, \ldots, a_1, a_0)$ de nombres naturels $a_k \in \mathbb{N}$ représente un nombre naturel, à savoir

$$\langle a_n, \ldots, a_1, a_0 \rangle_b := a_n b^n + \cdots + a_1 b^1 + a_0 = \sum_{k=0}^{n} a_k b^k.$$

Si $a_n \neq 0$ et $0 \leq a_k < b$ pour tout $k = 0, 1, \ldots, n$, nous appelons la suite $(a_n, \ldots, a_1, a_0)$ une *représentation normale* par rapport à la base b, ou aussi un *développement* en base b.

Proposition 1.2. *Tout nombre naturel $a \in \mathbb{N}$ peut être écrit de manière unique comme $a = \langle a_n, \ldots, a_1, a_0 \rangle_b$ avec $0 < a_n < b$ et $0 \leq a_k < b$ pour tout $k = 0, 1, \ldots, n-1$. L'application $(a_n, \ldots, a_1, a_0) \mapsto a = \sum_{k=0}^{n} a_k b^k$ établit donc une bijection entre nombres naturels a et développements $(a_n, \ldots, a_1, a_0)$ en base b. Sous cette bijection on appelle a_k le kième chiffre de a dans son développement en base b.* □

Dans le cas $b = 10$ on appelle *chiffres décimaux* les symboles `0,1,2,3,4,5,6,7,8,9`, que l'on identifie avec les dix premiers nombres naturels. Ainsi tout nombre naturel a peut être représenté par une suite de chiffres décimaux $(a_n, \ldots, a_1, a_0)$, appelée le *développement décimal* de a. À noter, en particulier, qu'avec cette convention le nombre naturel 0 sera représenté par la suite vide.

Exercice/M 1.3. Étant donnés deux nombres naturels $a' = \langle a'_n, \ldots, a'_1, a'_0 \rangle_b$ et $a'' = \langle a''_n, \ldots, a''_1, a''_0 \rangle_b$, leur somme $a = a' + a''$ peut être représentée comme $\langle a'_n + a''_n, \ldots, a'_1 + a''_1, a'_0 + a''_0 \rangle_b$. Seul petit problème : cette représentation n'est en général pas normale. Pour ce faire on doit encore propager les retenues. L'algorithme II.1 ci-dessous fait exactement ceci. Essayez de justifier sa correction.

Algorithme II.1 Normalisation par rapport à une base b

Entrée: Une représentation $(a_m, \ldots, a_1, a_0) \in \mathbb{N}^{m+1}$ d'un entier a en base $b \geq 2$
Sortie: La représentation normale $(a_n, \ldots, a_1, a_0) \in \mathbb{N}^{n+1}$ de a en base b

Initialiser $retenue \leftarrow 0$, $n \leftarrow m$
pour k **de** 0 **à** n **faire** $a_k \leftarrow a_k + retenue$, $retenue \leftarrow a_k \operatorname{div} b$, $a_k \leftarrow a_k \bmod b$
tant que $retenue > 0$ **faire** $n \leftarrow n+1$, $a_n \leftarrow retenue \bmod b$, $retenue \leftarrow retenue \operatorname{div} b$
tant que $n \geq 0$ et $a_n = 0$ **faire** $n \leftarrow n-1$
retourner $(a_n, \ldots, a_1, a_0)$

1.2. Implémentation en C++. On définit le type `Chiffre` pour représenter un chiffre, ou un petit entier à deux chiffres au plus. Le type `Naturel` est un vecteur de chiffres. Techniquement il est avantageux de tout formuler sous forme d'une *classe*, mais c'est ici un détail sans importance : notre implémentation restera assez facile à comprendre, même sans explication détaillée de la programmation « orientée objet ».

Programme II.1 Définition du type `Naturel`, conversion de `int`, et opérateur de sortie

```
typedef int Indice;                          // type pour stocker un indice
typedef short int Chiffre;                   // type pour stocker un chiffre (ou deux)
const Chiffre base= 10;                      // base entre 2 et 16, ici on choisit 10
const char chiffre[]= "0123456789abcdef";    // chiffres pour l'affichage

class Naturel
{
public:
  vector<Chiffre> chiffres;    // La suite des chiffres dans la base donnée

  void raccourcir()            // Raccourcir en effaçant d'éventuels zéros terminaux
    { while( !chiffres.empty() && chiffres.back()==0 ) chiffres.pop_back(); };

  void normaliser()            // Normaliser pour n'avoir que de chiffres 0,...,b-1
    {
      Chiffre retenue= 0;
      for( Indice i=0; i<chiffres.size(); ++i )
        { chiffres[i]+= retenue; retenue= chiffres[i]/base; chiffres[i]%= base; };
      while( retenue > 0 ) { chiffres.push_back( retenue % base ); retenue/= base; };
      raccourcir();
    };

  Naturel( int n= 0 )          // Constructeur par conversion d'un petit entier
    { for( ; n>0; n/=base ) chiffres.push_back( Chiffre( n % base ) ); };

  Naturel& operator= ( const Naturel& n )  // Affectation (par copie des chiffres)
    { chiffres= n.chiffres; return *this; };

  void clear()                 // Remettre à zéro (= suite vide)
    { chiffres.clear(); };

  bool est_zero() const        // Tester si zéro (= suite vide)
    { return chiffres.empty(); };

  Indice size() const          // Déterminer la taille (= longueur de la suite)
    { return chiffres.size(); };

  Chiffre operator[] ( Indice i ) const    // Lire le chiffre à la position i
    { return ( i<0 || i>=chiffres.size() ? Chiffre(0) : chiffres[i] ); };
};

// Conversion valeur -> symbole pour la sortie
char symbole( Chiffre valeur )
{ return ( valeur>=0 && valeur<base ? chiffre[valeur] : '?' ); }

// Opérateur de sortie (afficher les chiffres, ou bien "0" pour une suite vide)
ostream& operator<< ( ostream& out, const Naturel& n )
{
  Indice i= n.chiffres.size();
  if ( i == 0 ) return ( out << 0 );
  for( --i; i>=0; --i ) out << symbole( n.chiffres[i] );
  return out;
}
```

Avec le début de notre implémentation on peut déjà écrire le programme II.2 suivant :

Programme II.2 Un exemple d'utilisation `naturel-exemple.cc`

```
1  #include <iostream>          // déclarer l'entrée-sortie standard
2  #include "naturel.cc"        // inclure notre implémentation de la classe Naturel
3  using namespace std;         // accès direct aux fonctions standard
4
5  int main()
6  {
7    Naturel a;                 // définition d'une variable (initialisée à zéro)
8    Naturel b(123);            // définition avec initialisation à la valeur 123
9    cout << "a = " << a << ", b = " << b << endl;
10   a= b;                      // affectation (copie des chiffres de b vers a)
11   cout << "a = " << a << ", b = " << b << endl;
12   a.clear();                 // effacer la valeur stockée (remettre à zéro)
13   cout << "a = " << a << ", b = " << b << endl;
14   a= 42;                     // affecter une valeur (conversion implicite)
15   cout << "a = " << a << ", b = " << b << endl;
16   cout << "développement de longueur " << a.size();
17   for( int i=0; i<=10; ++i ) cout << ", a[" << i << "]=" << a[i];
18   cout << endl;
19 }
```

Représentation interne: Tout d'abord, on veut que tout nombre naturel puisse être représenté comme une valeur de type `Naturel`. Ceci est réalisé ici en stockant les chiffres du développement décimal dans un vecteur appelé `chiffres`. Par convention on stocke un développement décimal $(a_n,\dots,a_1,a_0)$ comme un vecteur `a[0],a[1],...,a[n]` de longueur $n+1$, dans le sens de *poids croissant*. C'est juste une convention commode, mais une fois cette décision est prise, il faut s'y conformer dans les fonctions ultérieures, sans aucune exception.

Normalisation: Par précaution on a déjà implémenté deux fonctions auxiliaires : `raccourcir()` qui supprime d'éventuels zéro terminaux, ainsi que `normaliser()` qui propage d'éventuelles retenues comme expliqué plus haut afin de n'avoir que des chiffres $0,\dots,9$. Ces deux fonctions pourraient servir dans des fonctions ultérieures qui doivent renvoyer un résultat normalisé.

Constructeur: On peut définir une variable `a` de type `Naturel` par `Naturel a;` elle sera initialisée à zéro (la valeur par défaut dans le constructeur). La définition `Naturel b(123)` entraîne une initialisation à la valeur décimale 123 : la suite des chiffrés stockée sera donc `3,2,1` (sic !). À noter que l'opérateur de sortie se chargera d'un affichage dans l'ordre usuel.

Opérateur d'affectation: On voudra utiliser l'opérateur d'affectation `=` avec la syntaxe et la sémantique usuelle : il prend deux arguments, une variable de type `Naturel` à gauche et une valeur de type `Naturel` à droite, puis il affecte la valeur à la variable. Dans notre cas, l'instruction `a= b;` copie le développement décimal stocké dans `b` dans la variable `a`.

Conversion de type: La construction par conversion permet d'affecter une valeur de type `int` à une variable de type `Naturel` : on pourra écrire `a= Naturel(42)` pour une conversion explicite ou bien `a= 42` pour une conversion implicite. Les deux passent par le constructeur fourni ci-dessus, et produisent un objet anonyme temporaire de type `Naturel` contenant les chiffres `2,4` du développement décimal de 42 ; celui-ci est ensuite copié dans `a` par l'opérateur d'affectation.

Réinitialisation: La fonction `clear()` permet d'effacer le développement décimal ; le résultat (la suite vide) représente zéro. Réciproquement la fonction `est_zero()` permet de déterminer si un nombre vaut zéro : il suffit de tester si le développement est de longueur 0. À noter que l'on suppose toujours une représentation normalisée ; par exemple $\langle 0,0,0\rangle_{\text{dec}}$ n'est pas une représentation normale de zéro !

Accès aux chiffres: On fournit aussi un opérateur d'indexation *sécurisé* : pour un indice légitime on renvoie le chiffre correspondant stocké dans le vecteur, et pour tout indice en dehors de cette plage on renvoie 0, la seule valeur mathématiquement raisonnable. (Le justifier.) On peut ainsi accéder aux chiffres du développement décimal sans se soucier de la longueur exacte de la représentation interne. (Quant à l'indexation des vecteurs, relire les avertissements du chapitre I, §6.1.)

Entrée-sortie: Les opérateurs d'entrée `>>` et de sortie `<<` permettent de lire et d'écrire des valeurs de type `Naturel` ; ils traduisent alors entre la représentation externe (l'écriture décimale usuelle) et la représentation interne (qui peut en différer). On n'a reproduit ci-dessus que la sortie, qui est plus facile. Vous trouverez l'opérateur d'entrée dans le fichier `naturel.cc`.

☞ Les chiffres sont affichés dans l'ordre usuel $a_n, \ldots, a_1, a_0$, alors qu'ils sont stockés dans l'ordre inverse. Ceci illustre un principe important : le stockage interne et l'écriture externe sont indépendants. C'est la tâche des opérateurs d'entrée-sortie de traduire entre les deux.

Exercice/P 1.4. Avec ces quelques lignes de code, notre implémentation est déjà opérationnelle. Bien évidemment, il manque encore l'arithmétique, que nous allons implémenter dans la suite. Si vous êtes courageux vous pouvez essayer de programmer une solution vous-même, ou au moins esquisser une implémentation. Vous verrez que ce n'est pas trivial. Voici ce que l'on souhaite faire :

Incrémentation et decrémentation: On voudra utiliser l'incrémentation `++` et la decrémentation `--` avec la syntaxe usuelle, afin de « compter » avec le type `Naturel`.

Opérateurs de comparaison: On voudra utiliser les opérateurs de comparaison `==`, `!=`, ainsi que `<`, `<=`, `>`, `>=` comme d'habitude.

Opérateurs arithmétiques: On voudra utiliser les opérateurs arithmétiques `+`, `-`, `*`, `/`, `%` avec la syntaxe et la sémantique usuelles.

Opérateurs mixtes: Finalement, les opérateurs mixtes `+=`, `-=`, `*=`, `/=`, `%=` devront s'utiliser d'une manière naturelle, de sorte que `a+=b` équivaille à `a=a+b` etc.

1.3. Incrémenter et decrémenter. Comme première opération arithmétique nous commençons par le processus de *compter*, c'est-à-dire augmenter $n \mapsto n+1$ ou diminuer $n \mapsto n-1$ (pour $n > 0$). Essayer de comprendre leur fonctionnement (sur quelques exemples) puis de justifier leur correction.

Programme II.3 Incrémentation et decrémentation

```
// Incrémenter : la fonction renvoie toujours 'true' (= exécution sans erreur).
bool incrementer( Naturel& n )
{
  for( Indice i=0; i<n.size(); ++i )
    if ( n.chiffres[i] == Chiffre(base-1) ) n.chiffres[i]= Chiffre(0);
    else { ++(n.chiffres[i]); return true; };
  n.chiffres.push_back( Chiffre(1) );
  return true;
}

// Incrémentation sous forme d'opérateur
Naturel& operator++ ( Naturel& n )
{ incrementer(n); return n; }

// Decrémenter : si n=0 la fonction renvoie 'false' (= erreur).
bool decrementer( Naturel& n )
{
  for( Indice i=0; i<n.size(); ++i )
    if ( n.chiffres[i] == Chiffre(0) ) n.chiffres[i]= Chiffre(base-1);
    else { --(n.chiffres[i]); n.raccourcir(); return true; };
  return false;
}

// Decrémentation sous forme d'opérateur
Naturel& operator-- ( Naturel& n )
{
  if ( !decrementer(n) )
    cerr << "Erreur : on ne peut decrémenter 0." << endl;
  return n;
}
```

1.4. Comparaison. Comme deuxième opération nous implémentons la comparaison entre deux nombres naturels a et b. Par convention, le résultat vaut soit 0 si $a = b$, soit $+1$ si $a > b$, soit -1 si $a < b$. Nous montrons ici deux solutions possibles :

Exercice/P 1.5. La première méthode de comparaison diminue a et b jusqu'à ce qu'au moins un des deux vaut 0. C'est clairement une méthode correcte, et elle semble raisonnable pour les petits entiers, disons jusqu'à 1000. Mais cette méthode devient très inefficace pour les grands entiers. Si l'on veut comparer 10^{50} et 10^{60}, par exemple, quel temps de calcul prédirez-vous ?

Exercice/P 1.6. La deuxième méthode est celle connue de l'école : on compare d'abord les longueurs, puis en cas de coïncidence on compare les chiffres un par un dans l'ordre du poids décroissant. Montrer que l'algorithme donné est correct, c'est-à-dire pour toutes données d'entrée a, b il renvoie la réponse 0 ou ± 1 comme spécifiée ci-dessus. Attention : on utilise à nouveau l'hypothèse que les représentations stockées pour a et b soient toujours normalisées. Sinon, que se passerait-il ?

Programme II.4 Comparaison de deux nombres de type `Naturel`

```
int comparaison_lente( Naturel a, Naturel b )
{
  while( !a.est_zero() && !b.est_zero() ) { decrementer(a); decrementer(b); }
  if ( a.est_zero() && b.est_zero() ) return 0;
  if ( a.est_zero() ) return -1; else return +1;
}

int comparaison_scolaire( const Naturel& a, const Naturel& b )
{
  if ( a.chiffres.size() > b.chiffres.size() ) return +1;
  if ( a.chiffres.size() < b.chiffres.size() ) return -1;
  for( Indice i= a.chiffres.size()-1; i>=0; --i )
    {
      if ( a.chiffres[i] > b.chiffres[i] ) return +1;
      if ( a.chiffres[i] < b.chiffres[i] ) return -1;
    }
  return 0;
}

bool operator== ( const Naturel& a, const Naturel& b )
{ return ( comparaison_scolaire(a,b) == 0 ); }

bool operator!= ( const Naturel& a, const Naturel& b )
{ return ( comparaison_scolaire(a,b) != 0 ); }

bool operator< ( const Naturel& a, const Naturel& b )
{ return ( comparaison_scolaire(a,b) < 0 ); }

bool operator<= ( const Naturel& a, const Naturel& b )
{ return ( comparaison_scolaire(a,b) <= 0 ); }

bool operator> ( const Naturel& a, const Naturel& b )
{ return ( comparaison_scolaire(a,b) > 0 ); }

bool operator>= ( const Naturel& a, const Naturel& b )
{ return ( comparaison_scolaire(a,b) >= 0 ); }
```

Nous fournissons aussi la comparaison sous forme des opérateurs usuels `==`, `!=`, `<`, `<=`, `>`, `>=`, ce qui permet une écriture commode, comme `a<b` au lieu de `comparaison_scolaire(a,b)<0`. Sans cette implémentation explicite, le type `Naturel` n'admettrait pas de telle comparaison abrégée, car le compilateur ne saurait pas deviner le sens de l'instruction `a<b`. À nous donc de préciser ce que nous souhaitons réaliser.

1.5. Addition et soustraction. L'addition peut être implémentée par deux méthodes différentes :

Exercice/P 1.7. La première méthode réalise l'addition par une incrémentation itérée. On initialise la somme s par $s \leftarrow a$. Tant que $b > 0$ on diminue b et augmente s. Finalement, lors que $b = 0$, la variable s contient la somme cherchée. Expliquer pourquoi cette méthode, aussi élégante qu'elle soit, n'est pas recommandable : dans quelles situations est-elle raisonnable, dans quelles est-elle catastrophique ?

Exercice/P 1.8. La deuxième approche effectue l'addition scolaire de $a = \sum_{i=0}^{\ell_a} a_i 10^i$ et $b = \sum_{i=0}^{\ell_b} b_i 10^i$ en additionnant chiffre par chiffre, du plus petit au plus haut poids. Vérifier les détails de cette implémentation afin de justifier sa correction : renvoie-t-elle toujours la représentation normalisée de la somme cherchée ?

☞ Voici quelques remarques et indications. On part de l'égalité

$$\left(\sum_{i=0}^{m} a_i 10^i\right) + \left(\sum_{i=0}^{m} b_i 10^i\right) = \sum_{i=0}^{m} (a_i + b_i) 10^i,$$

qui se traduit immédiatement en une boucle parcourant $i = 0, 1, 2, \ldots, m$. Évidemment le résultat n'est en général pas normalisé : comme à l'école, la principale complication est la retenue. On pourrait effectuer une normalisation tout à la fin, en appelant la fonction `normaliser()`. Il semble plus efficace de propager la retenue déjà dans la boucle : c'est possible si l'on la parcourt dans le bon sens, de $i = 0$ à $i = m$.

La longueur de la somme est $m = \max(\ell_a, \ell_b)$, ou bien $m + 1$ s'il reste une retenue à la fin. Lors de la boucle, si $\ell_a > \ell_b$, il faut rajouter des chiffres zéro à la fin de b ; de même, si $\ell_a < \ell_b$, il faut rajouter des chiffres zéro à la fin de a. On s'en tire avec une astuce : les deux cas sont assurés ici par l'usage de l'opérateur `[]` sécurisé, implémenté plus haut. (C'est légèrement inefficace, mais le code est plus élégant.)

Programme II.5 Addition de deux nombres de type `Naturel`

```
bool addition_lente( const Naturel& a, Naturel b, Naturel& somme )
{ for( somme= a; !b.est_zero(); incrementer(somme), decrementer(b) ); return true; }

bool addition_scolaire( const Naturel& a, const Naturel& b, Naturel& somme )
{
  // Réserver de la mémoire pour recevoir le résultat
  somme.clear();
  Indice taille= max( a.size(), b.size() );
  somme.chiffres.resize( taille, Chiffre(0) );

  // Calculer la somme chiffre par chiffre en tenant compte des retenues
  Chiffre retenue= 0;
  for( Indice i=0; i<taille; ++i )
    {
      Chiffre temp= a[i] + b[i] + retenue;
      if ( temp < base ) { somme.chiffres[i]= temp; retenue = 0; }
      else { somme.chiffres[i]= temp - base; retenue = 1; };
    }

  // Rajouter éventuellement un chiffre supplémentaire pour la retenue
  if ( retenue ) somme.chiffres.push_back(retenue);
  return true;
}

Naturel operator+ ( const Naturel& a, const Naturel& b )
{ Naturel somme; addition_scolaire( a, b, somme ); return somme; }

Naturel& operator+= ( Naturel& a, const Naturel& b )
{ a= a+b; return a; }
```

Exercice/P 1.9. Si vous voulez vous pouvez implémenter les deux approches (dites lente et scolaire) pour effectuer la soustraction $a - b$. Par convention on renvoie `false` si le résultat se révèle négatif. Vous trouverez une solution possible dans `naturel.cc`.

1.6. Multiplication. Nous implémentons la multiplication par deux méthodes différentes :

Exercice/P 1.10. La première méthode réalise la multiplication par une addition itérée. On initialise le produit p par $p \leftarrow 0$. Tant que $b > 0$ on diminue b et ajoute $p \leftarrow p + a$. Finalement, lorsque $b = 0$, la variable p contient le produit cherché. Expliquer pourquoi cette méthode, aussi élégante qu'elle soit, n'est pas recommandable : dans quelles situations est-elle raisonnable, dans quelles est-elle catastrophique ?

Exercice/P 1.11. Comme deuxième méthode nous implémentons la multiplication scolaire. Exécuter la fonction à la main sur un exemple, disons $456 \cdot 789$. Vérifier les détails de notre implémentation afin de justifier sa correction : renvoie-t-elle toujours la représentation normalisée du produit cherché ? A-t-on réservé un vecteur de taille suffisante ? trop grande ? Risque-t-on des problèmes de capacité insuffisante en utilisant le type `short int` pour `Chiffre` ? Quelle est la plus grande valeur qui puisse apparaître dans la variable `retenue` ? Donner un exemple où ce maximum est atteint.

☞ Voici quelques remarques et indications. Tout d'abord, la multiplication par zéro joue un rôle particulier et sera traitée à part. (Pourquoi ?) Sinon on part de l'égalité

$$\left(\sum_{i=0}^{m} a_i 10^i\right) \cdot \left(\sum_{j=0}^{n} b_j 10^j\right) = \sum_{i=0}^{m} \sum_{j=0}^{n} (a_i b_j) 10^{i+j},$$

qui se traduit en deux boucles imbriquées, parcourant $i = 0, 1, 2, \ldots, m$ et $j = 0, 1, 2, \ldots, n$, afin de sommer tous les produits $a_i b_j 10^{i+j}$. Bien évidemment en base 10 le facteur 10^{i+j} veut dire qu'il faut ajouter le produit $a_i b_j$ à la position $i + j$.

À nouveau le résultat brut n'est pas encore normalisé. Il y a au moins deux manières différentes de ce faire : on pourrait par exemple appeler la fonction `normaliser()` tout à la fin. Cette approche est pourtant dangereuse, car elle nécessite de stocker toute la somme $\sum_{i+j=k} a_i b_j$ dans `produit[k]`. Ceci pourrait déborder la capacité du type `short int`. (Essayer de construire un exemple de ce genre.)

Pour cette raison on a pris le soin de formuler une méthode qui traite la retenue directement dans la boucle intérieure, et qui assure ainsi de ne pas déborder le type `short int`.

Programme II.6 Multiplication de deux nombres de type `Naturel`

```
bool multiplication_lente( const Naturel& a, Naturel b, Naturel& produit )
{ for( produit= 0; !b.est_zero(); --b, produit+= a ); return true; }

bool multiplication_scolaire( const Naturel& a, const Naturel& b, Naturel& produit )
{
  produit.clear();
  if ( a.size() == 0 || b.size() == 0 ) return true;
  produit.chiffres.resize( a.size() + b.size(), Chiffre(0) );
  for( Indice i=0; i<a.size(); ++i )
    {
      Chiffre aux, retenue= 0;
      for( Indice j=0; j<b.size(); ++j )
        {
          aux= produit.chiffres[i+j] + a.chiffres[i] * b.chiffres[j] + retenue;
          produit.chiffres[i+j]= aux % base;
          retenue= aux / base;
        }
      produit.chiffres[i+b.size()]= retenue;
    }
  produit.raccourcir();
  return true;
}

Naturel operator* ( const Naturel& a, const Naturel& b )
{ Naturel produit; multiplication_scolaire( a, b, produit ); return produit; }

Naturel& operator*= ( Naturel& a, const Naturel& b )
{ a= a*b; return a; }
```

1.7. Division euclidienne. Essayons finalement d'implémenter la division euclidienne de a par b. Pour rappel : pour tout $a,b \in \mathbb{N}$ avec $b \neq 0$ il existe une unique paire $(q,r) \in \mathbb{N} \times \mathbb{N}$ de sorte que $a = bq + r$ et $0 \leq r < b$. Nous présentons deux méthodes pour calculer cette paire (q,r) :

Exercice/P 1.12. La première méthode réalise la division euclidienne par une soustraction itérée. On initialise les variables q et r par $q \leftarrow 0$ et $r \leftarrow a$. Tant que $r \geq b$ on pose $r \leftarrow r - b$ et $q \leftarrow q + 1$. Ainsi l'égalité $a = bq + r$ est préservée par chaque itération. Finalement, lorsque $r < b$, nous avons trouvé la paire (q,r) cherchée. Expliquer pourquoi cette méthode, aussi élégante qu'elle soit, n'est pas recommandable : dans quelles situations est-elle raisonnable, dans quelles est-elle catastrophique ?

Exercice/P 1.13. Comme deuxième approche nous implémentons la division euclidienne par (une variante de) la méthode scolaire : on construit le développement décimal du quotient en appliquant intelligemment la division lente ci-dessus. Pour commencer vous pouvez exécuter la fonction à la main pour la division de $a = 567$ par $b = 19$, par exemple. Essayez de comprendre le fonctionnement de cet algorithme, puis justifiez sa correction. Vous trouvez un développement très accessible dans Shoup [**10**],§3.3, et une discussion plus approfondie dans Knuth [**8**], vol. 2, §4.3.

Programme II.7 Division euclidienne de deux nombres de type `Naturel`

```
bool division_lente( const Naturel& a, Naturel b, Naturel& quot, Naturel& reste )
{
  if ( b.est_zero() ) return false;
  for( quot= 0, reste= a; reste>=b; ++quot, reste-= b );
  return true;
}

bool division_scolaire( const Naturel& a, const Naturel& b, Naturel& quot, Naturel& reste )
{
  quot.clear(); reste.clear();
  if ( b.est_zero() ) return false;
  if ( b.size() > a.size() ) { reste= a; return true; };
  quot.chiffres.resize( a.size() - b.size() + 1, Chiffre(0) );
  for( Indice i= a.size()-1; i>=0; --i )
    {
      // Calculer reste = base*reste + a[i] puis (peu de) soustractions itérées
      reste.chiffres.push_back( Chiffre(0) );
      for( Indice j= reste.size()-1; j>0; --j )
        reste.chiffres[j]= reste.chiffres[j-1];
      reste.chiffres[0]= a.chiffres[i];
      reste.raccourcir();
      while( reste>=b ) { reste= reste-b; ++quot.chiffres[i]; }
    }
  quot.raccourcir();
  return true;
}

Naturel operator/ ( const Naturel& a, const Naturel& b )
{
  Naturel q,r;
  if ( !division(a,b,q,r) ) cerr << "Erreur : division non définie." << endl;
  return q;
}

Naturel operator% ( const Naturel& a, const Naturel& b )
{
  if ( b.est_zero() ) return a;
  Naturel q,r; division(a,b,q,r); return r;
}

Naturel& operator/= ( Naturel& a, const Naturel& b ) { a= a/b; return a; }
Naturel& operator%= ( Naturel& a, const Naturel& b ) { a= a%b; return a; }
```

2. Questions de complexité

2.1. Le coût des calculs. Le but des implémentations précédentes était de reprendre les algorithmes que vous appliquez depuis toujours, et de les expliciter en langage de programmation. Ceci n'est pas immédiat et nécessite un certain effort, puis des tests et des vérifications soigneuses. Après s'être convaincu de la correction de nos fonctions, on arrive à la question d'efficacité :

Exercice/P 2.1. Lire puis exécuter le programme `naturel-exemple.cc` qui effectue quelques calculs illustratifs. Vérifier dans la mesure du possible la corrections sur des petits exemples. Puis, sur des exemples de plus en plus grands, comparer la performance des algorithmes « scolaires » et « lents ».

Formuler avec précision vos observations : jusqu'à quelle taille les algorithmes lents sont-ils raisonnables ? Dans quels cas s'avèrent-ils catastrophiques ? Comment expliquer ces phénomènes ? Est-ce que les algorithmes scolaires, eux-aussi, ralentissent sur des exemples très grands ? de manière significative ? L'addition est-elle plus rapide que la multiplication ? Pourquoi ?

La morale de cette histoire est que les algorithmes lents et scolaires, bien qu'ils mènent tous au même résultat, se comportent très différemment au niveau de leur *complexité*, c'est-à-dire leur temps d'exécution diffère drastiquement en fonction de l'entrée. Le choix d'un mauvais algorithme entraîne que l'on ne peut pas effectuer certains calculs en temps utile, alors que c'est facile avec un algorithme bien adapté.

2.2. Complexité linéaire vs quadratique.

Exercice 2.2. Soient a et b deux nombres naturels à ℓ décimales. Expliquer pourquoi le temps nécessaire pour effectuer l'addition lente est proportionnel à 10^ℓ environ. On dit ainsi que cet algorithme est de complexité exponentielle. Il en est de même pour les autres algorithmes qualifiés « lents » ci-dessus.

☞ Lorsqu'un algorithme est de complexité exponentielle, il n'est utilisable que pour des exemples minuscules. Dans notre exemple, chaque fois que l'on augmente la longueur ℓ par un, le temps de calcul est multiplié par 10. Pour des problèmes de grandeur nature, où $\ell \approx 100$ disons, une telle méthode est inutilisable.

Exercice 2.3. Soient a et b deux nombres naturels à ℓ décimales. Expliquer pourquoi le temps nécessaire pour effectuer l'addition scolaire est proportionnel à ℓ seulement. On dit ainsi que cet algorithme est de *complexité linéaire*. Chaque fois que ℓ est doublé, le coût double lui aussi.

Exercice 2.4. Soient a et b deux nombres naturels à ℓ décimales. Expliquer pourquoi le temps nécessaire pour effectuer la multiplication scolaire est proportionnel à ℓ^2. On dit ainsi que cet algorithme est de *complexité quadratique*. Chaque fois que ℓ est doublé, le coût est multiplié par 4.

☞ Lorsqu'un algorithme est de complexité quadratique, il n'est utilisable que pour des problèmes moyennement grands. On chronométra l'addition et la multiplication plus amplement au chapitre III.

Exercice 2.5. Soit a un nombre naturel à ℓ décimales. Quel est le coût de l'incrémentation $a \mapsto a+1$ dans le meilleur des cas ? dans le pire des cas ? en moyenne ? Mêmes questions pour la comparaison scolaire de deux nombres a et b de longueur $\leq \ell$.

Exercice 2.6. Expliquer pourquoi la division euclidienne, comme la multiplication, est d'une complexité quadratique quand on utilise l'algorithme scolaire. Plus précisément, pour calculer $(a,b) \mapsto (q,r)$ avec $a = qb + r$ et $0 \leq r < b$ il faut nm itérations, où n et m sont les longueurs de b et q, respectivement.

3. Exercices supplémentaires

3.1. Numération romaine. L'exercice suivant est classique. Techniquement il porte sur les chaînes de caractères, mathématiquement il met en relief la simplicité de la numération décimale à position.

Exercice/P 3.1. Écrire une fonction `string romain(int n)` qui transforme un nombre n (entre 1 et 3999) en numération romaine, en utilisant les « chiffres » `I,V,X,L,C,D,M`. (Rappeler d'abord les règles de cette écriture.) Écrire une fonction `int romain(string s)` qui réalise la traduction inverse. On pourra commencer par une fonction `int romain(char c)` qui traduit les symboles `I,V,X,L,C,D,M` en leur valeur respective `1,5,10,50,100,500,1000`.
Remarque. — Ce système, bien que démodé, est toujours en usage : regarder par exemple les numéros des chapitres ou des toutes premières pages de ces notes.

3.2. Partitions. Cet exercice est complexe, mais bénéfique pour la culture mathématique. Une *partition* d'un nombre naturel n est une famille $p = (p_1, p_2, \dots, p_\ell)$ d'entiers positifs, décroissante dans le sens $p_1 \geq p_2 \geq \cdots \geq p_\ell \geq 1$, et ayant pour somme $p_1 + p_2 + \cdots + p_\ell = n$. On appelle n le *degré* et ℓ la *longueur* de la partition. Par exemple, les partitions de degré $n = 0, 1, 2, 3, 4$ sont exactement les suivantes :

$n = 0$: $()$ — la partition vide
$n = 1$: (1)
$n = 2$: (2), $(1,1)$
$n = 3$: (3), $(2,1)$, $(1,1,1)$
$n = 4$: (4), $(3,1)$, $(2,2)$, $(2,1,1)$, $(1,1,1,1)$

On organise ici les partitions dans l'ordre deg-invlex : on compare d'abord le degré (la plus petite somme d'abord), puis dans le même degré on utilise l'ordre lexicographique inverse (les plus grands vecteurs d'abord).

Exercice/P 3.2. En C++ on peut réaliser les partitions par les définitions suivantes :

```
typedef vector<int> Partition;  // type pour stocker une partition
```

Écrire un opérateur de sortie convenable pour les partitions, puis ajouter une fonction qui calcule le degré. Implémenter une fonction qui compare deux partitions p et q dans l'ordre deg-invlex : elle renvoie `+1` si $p > q$, et `-1` si $p < q$, puis `0` si $p = q$. Ajouter ensuite les quatre opérateurs `<`, `<=`, `>`, `>=`.

Finalement, implémenter l'incrémentation et la décrémentation dans l'ordre deg-invlex :

```
Partition& operator ++ ( Partition& p );  // incrémentation
Partition& operator -- ( Partition& p );  // decrémentation
```

Le défi est ici de développer un algorithme correct pour trouver le successeur et le prédécesseur d'une partition donnée. Vous pouvez tester votre implémentation en énumérant les partitions comme suit :

```
Partition debut(1,1), fin(5,1);
for( Partition p=debut; p<=fin; ++p ) cout << p << endl;
for( Partition q=fin; q>=debut; --q ) cout << q << endl;
```

Comme une application parmi beaucoup d'autres, regardons les matrices $n \times n$ ayant pour polynôme caractéristique $(X - \lambda)^n$. Combien y en a-t-il à conjugaison près ? Les énumérer pour $n = 1, 2, 3, 4, 5, \dots$.

“Can you do addition?” the White Queen asked.
“What’s one and one and one and one and one
and one and one and one and one and one?”
“I don’t know,” said Alice. “I lost count.”
Lewis Carroll, *Through the Looking Glass*

COMPLÉMENT II

Fondement de l’arithmétique : les nombres naturels

Objectifs

- Retracer le fondement axiomatique des nombres naturels (d’après Dedekind et Landau).
- Établir quelques résultats fondamentaux (bon ordre, division euclidienne, numération en base *b*)

Comme ce cours se veut élémentaire, le chapitre II vous propose d’implémenter les nombres naturels sur ordinateur, d’abord l’addition/soustraction puis la multiplication/division en numération décimale. On pourrait s’y étonner : pourquoi commencer par des concepts aussi basiques ? Tout d’abord, c’est un bon exercice de programmation. Mais aussi d’un point de vue mathématique il y a de bonnes raisons pour reconsidérer sérieusement les algorithmes de l’arithmétique élémentaire. Cet annexe essaie de mettre en relief leur rôle important, et de poser les fondements mathématiques nécessaires.

Sommaire

1. Apologie de l’arithmétique élémentaire

Avec toute modestie, je souhaite vous présenter quelques raisons pour reconsidérer sérieusement les algorithmes de l’arithmétique élémentaire. Ceci souligne aussi leur importance historique et culturelle.

1.1. Motivation pédagogique. On utilisera les opérations arithmétiques des entiers partout dans la suite de ce cours. Il me semble donc honnête de commencer par le début, et de poser les fondements avant d’édifier les étages supérieurs. D’autant plus que ceci vous permettra de mieux situer votre travail, et de comprendre comment votre ordinateur stocke et travaille les grands entiers. Certes, on verra des découvertes algorithmiques plus excitantes encore, mais peut-on apprécier des méthodes sophistiquées si l’on méconnaît les méthodes de base ? Soyons honnêtes : en général, l’élémentaire est un bon exercice ; en le retravaillant on trouve souvent des questions et des approfondissement auparavant inaperçus. Le projet 1 en donne en bel exemple : qui aurait soupçonné que la bonne vielle multiplication admettait des solutions nettement plus efficaces ?

1.2. Motivation culturelle. L’arithmétique en numération décimale fournit les exemples les plus classiques d’algorithmes. Familière à tous et fréquemment utilisée au quotidien, l’arithmétique fait partie de notre bagage culturel. La traduction en C++ vous rappellera peut-être à quel point c’est un outil non-trivial.

Le comparer, par exemple, à la *numération linéaire* |, ||, |||, ||||, |||||, … Pour vous amusez, vous pouvez expliciter les règles de comparaison, addition, soustraction, multiplication et division euclidienne dans cette notation. C’est facile mais très inefficace pour les nombres plus grands. (Pourquoi ?) Un peu plus confortable pour les petits nombres, la *numération romaine* I, II, III, IV, V, … n’est guère plus efficace.

Pour illustration citons une correspondance, datant du moyen âge, entre un riche marchand et un intellectuel. Le premier cherchait conseil auprès de son ami pour savoir où il serait préférable d’envoyer son enfant étudier la science des nombres et du calcul, afin de lui garantir la meilleure formation possible. Ce dernier répondit :

> Si tu désires l'initier à la pratique de l'addition ou de la soustraction, n'importe quelle université française, allemande ou anglaise fera l'affaire. Mais, pour la maîtrise de l'art de multiplier ou de diviser, je te recommande plutôt de l'envoyer dans une université italienne.

À cette époque, le dur apprentissage de l'arithmétique élémentaire était strictement réservé à l'élite intellectuelle, au prix de trois ou quatre années d'études universitaires. Cette situation changerait dramatiquement avec l'apparition de la numération décimale, qui entraînerait une « démocratisation » successive de l'arithmétique élémentaire.

1.3. Motivation historique. On ne peut pas surestimer l'importance qui eurent le développement puis la large diffusion de « notre » numération décimale. Ne citons que l'éloge formulée par Pierre-Simon de Laplace :

> C'est à l'Inde que nous devons la méthode ingénieuse d'exprimer tous les nombres au moyen de dix symboles, chaque symbole ayant une valeur de position ainsi qu'une valeur absolue. Idée profonde et importante, elle nous apparaît maintenant si simple que nous en méconnaissons le vrai mérite. Mais sa réelle simplicité, la grande simplicité qu'elle a procurée à tous les calculs, met notre arithmétique au premier plan des inventions utiles, et nous apprécierons d'autant plus la grandeur de cette œuvre que nous nous souviendrons qu'elle a échappé au génie d'Archimède et d'Apollonius, deux des plus grands hommes qu'ait produits l'antiquité.

Des livres entiers ont été consacrés à la fascinante histoire de cette révolution culturelle, technologique et scientifique. Pour commencer je vous conseille Knuth [**8**], vol. 2, §4.1.

L'histoire a aussi laissé des traces étymologique : le mot « algorithme » est dérivé d'Al-Khwarizmi, mathématicien persan et auteur de deux importants manuscrits, écrits environ de l'année 825 de notre ère, sur la résolution d'équations (*algèbre*) et l'arithmétique en numération décimale, empruntée des Indiens. Ainsi le mot latin « algorismus » puis « algorithmus » est devenu le mot français « algorithme ». Il signifiait initialement l'ensemble des techniques d'Al-Khwarizmi, notamment le calcul en numération décimale. Les traductions et vulgarisations de ses œuvres à partir du XIIème siècle jouèrent un rôle essentiel dans l'apparition de la numération à position en Europe.

1.4. Motivation mathématique. Alfred North Whitehead, dans son livre *An Introduction to Mathematics*, souligne plus généralement l'importance d'une notation adéquate pour résoudre des problèmes mathématiques :

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. Before the introduction of the Arabic notation, multiplication was difficult, and the division even of integers called into play the highest mathematical faculties. Probably nothing in the modern world would have more astonished a Greek mathematician than to learn that, under the influence of compulsory education, the whole population of Western Europe, from the highest to the lowest, could perform the operation of division for the largest numbers. (…) Our modern power of easy reckoning with decimal fractions is the almost miraculous result of the gradual discovery of a perfect notation.

1.5. Motivation algorithmique. D'après ce qui précède, l'arithmétique semble un bon point de départ pour notre exploration de l'algorithmique élémentaire. Et c'est un point auquel on peut revenir, avec des outils plus avancés : durant les 40 dernières années, suite à l'avènement des ordinateurs, l'arithmétique des grands entiers a vue une recherche approfondie, couronnée d'énormes progrès avec des algorithmes de plus en plus efficaces. (Il en est de même d'ailleurs pour l'arithmétique des polynômes de grand degré ou des matrices de grande taille). Bien que notre discussion restera assez basique, sachez qu'en algorithmique les algorithmes scolaires, qui n'ont pas changés depuis le moyen âge, ne sont pas le dernier mot.

2. Les nombres naturels

Pour la culture mathématique, mais aussi pour entreprendre une implémentation sérieuse sur ordinateur, il semble utile de (re)considérer de plus près l'objet mathématique que sont les « nombres naturels » et ses principales propriétés que l'on souhaite modéliser. Comme disait E. Landau, la subtilité des nombres naturels $0, 1, 2, 3, \ldots$ réside dans les mystérieux trois petits points. À titre d'avertissement, ils prennent un tout autre sens dans « A,B,C,… » ou dans « lundi, mardi, mercredi, … ». Nous allons donc éclaircir ce mystère par une définition rigoureuse.

☞ *Avertissement : l'abus de l'abstraction peut nuire à la compréhension.*
En cas de contre-indication, survoler le développement suivant pour passer à la suite. ✍

2.1. Qu'est-ce que les nombres naturels ? Nous avons tous assez jeunes appris à compter et à nous servir des nombres naturels. Devenue quasi sous-consciente, cette technique nous semble effectivement très naturelle, au moins très habituelle. Après réflexion on constate un phénomène surprenant : bien que tout le monde s'en serve et sache citer des *exemples*, comme « zéro, un, deux, trois, … », il n'est pas évident comment *définir* avec précision ce que sont les nombres naturels. (Faites l'expérience autour de vous, matheux et non-matheux confondus.)

Dans l'histoire des mathématiques, cette question de définition rigoureuse a été soulevée assez tard, et c'est seulement vers la fin du XIXe siècle qu'une réponse satisfaisante fut décortiquée. Nous résumons de manière très sommaire ce développement, qui fut un des premiers exemples de l'approche axiomatique qui caractérise l'ère moderne.

Selon vos préférences et votre expérience mathématique, cette approche vous semblera fascinant ou ennuyant, d'une élégance absolue ou d'une abstraction excessive. À vous de juger. En tout cas, ce développement est un bénéfique exercice de style. Il fait partie de la culture mathématique de l'avoir travaillé au moins une fois dans la vie.

2.2. L'approche axiomatique. Les nombres naturels sont l'abstraction mathématique du processus de *compter*. Le nombre initial est noté 0, puis à chaque nombre naturel n on associe son successeur $\mathbf{S}n$. Ceci permet déjà de construire et de nommer quelques petits exemples :

$$1 := \mathbf{S}0,\ 2 := \mathbf{S}1,\ 3 := \mathbf{S}2,\ 4 := \mathbf{S}3,\ 5 := \mathbf{S}4,\ 6 := \mathbf{S}5,\ 7 := \mathbf{S}6,\ 8 := \mathbf{S}7,\ 9 := \mathbf{S}8,\ 10 := \mathbf{S}9,\ \ldots$$

On sous-entend que ce processus continue infiniment sans jamais boucler, et que tout nombre naturel est ainsi atteint. La définition suivante en est la formulation mathématique d'après R. Dedekind (*Was sind und was sollen die Zahlen ?*, 1888).

Définition 2.1. Les nombres naturels forment un ensemble $\mathbb{N}$ muni d'un élément initial $0 \in \mathbb{N}$ et d'une application $\mathbf{S}\colon \mathbb{N} \to \mathbb{N}$ satisfaisant aux axiomes suivants :

(1) Pour tout n dans $\mathbb{N}$ on a $\mathbf{S}n \neq 0$, autrement dit 0 n'appartient pas à l'image de $\mathbf{S}$.

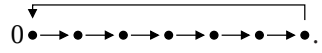
(2) Pour tout $n \neq m$ dans $\mathbb{N}$ on a $\mathbf{S}n \neq \mathbf{S}m$, autrement dit $\mathbf{S}$ est injectif.

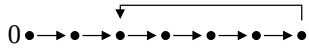
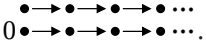
(3) Si $E \subset \mathbb{N}$ est un sous-ensemble contenant 0 et vérifiant $\mathbf{S}(E) \subset E$, alors $E = \mathbb{N}$.

La dernière condition est aussi appelée *l'axiome de récurrence*.

Remarque 2.2. Cette définition ne précise pas ce qui est un nombre naturel : la nature des éléments de $\mathbb{N}$ n'est pas essentielle, tout ce qui compte est l'application $\mathbf{S}\colon \mathbb{N} \to \mathbb{N}$ qui modélise le processus de compter. L'élément 0 est le point de départ ; c'est le seul élément qui n'est pas successeur d'un autre.

Graphiquement, les axiomes spécifient une application $\mathbf{S}\colon \mathbb{N} \to \mathbb{N}$ comme  .

L'axiome (1) exclut des applications comme  ou  .

L'injectivité (2) exclut  et l'axiome (3) exclut  .

Chacun des exemples suivants vous montre un modèle des nombres naturels, c'est-à-dire un triplet $(\mathbb{N}, 0, \mathbf{S})$ satisfaisant aux axiomes. (Si, si, il y a plusieurs possibilités mais on verra plus bas qu'elles sont essentiellement les mêmes.)

Exemple 2.3. Le modèle le plus évident consiste des suites finies répétant un symbole fixé, disons 1. Ici la suite vide 0 sert d'élément inital, avec $\mathbf{S}0 = 1$. Pour une suite non vide on pose $\mathbf{S}1\cdots1 = 1\cdots11$ en rajoutant un symbole de plus. C'est facile à définir, et l'ensemble des suites ainsi construites satisfait visiblement aux axiomes ci-dessus.

Exemple 2.4. En s'inspirant de la numération binaire on pourrait procéder comme suit. On choisit un ensemble à deux éléments $\{0,1\}$, puis on regarde les suites finies formées de 0 et 1 commençant par 1. La suite vide $()$ sert d'élément inital et on pose $\mathbf{S}() = 1$. Pour une suite non-vide on pose $\mathbf{S}1\ldots011\ldots11 = 1\ldots100\ldots00$ puis $\mathbf{S}11\ldots11 = 100\ldots00$. (Bien sûr il faudra expliciter un peu plus cette définition de $\mathbf{S}$.) Cette construction est un peu fastidieuse mais légitime : le résultat vérifie aux axiomes. Vous pouvez aussi vous amuser à réécrire cette définition pour la numération décimale.

Exemple 2.5. Vous pouvez bien sûr utiliser les mots « zéro, un, deux, trois, … » pour représenter les nombres naturels, comme vous le faites depuis votre enfance. Or, il ne suffit pas d'aller jusqu'à « neuf cent quatre-vingt-dix-neuf » ou un autre nombre « assez grand ». La difficulté est d'expliciter une règle qui associe à chaque n son successeur $\mathbf{S}n$: pour des nombres de plus en plus grands il faut faire face au problème d'inventer des noms, si possible d'une manière systématique et efficace. Une telle tentative sera sans doute similaire aux exemples précédents mais plus pénible encore.

Les trois exemples précédents sont acceptables d'un point de vue pragmatique, mais ils laissent à désirer d'un point de vue mathématique. L'exemple suivant est sans doute le plus élégant :

Exemple 2.6. John von Neumann proposa en 1923 un modèle des nombres naturels qui est devenu classique. L'élément initial est l'ensemble vide $\emptyset = \{\}$, puis on pose $0 := \emptyset$, $1 := \{\emptyset\}$, $2 := \{\emptyset,\{\emptyset\}\}$, $3 := \{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\},\ldots$ Ici le successeur d'un ensemble n est défini comme l'ensemble $\mathbf{S}n := n\cup\{n\}$. Ainsi chaque n est l'ensemble des nombres plus petits que n. Comme vous voyez, cette construction se base uniquement sur la théorie des ensembles (que nous admettons ici). Vous pouvez vérifier aisément que cette construction satisfait aux axiomes souhaités.

Exemple 2.7. Bourbaki construit les nombres naturels comme les cardinaux des ensembles finis, approche qui repose également sur la théorie des ensembles. Un nombre naturel est ici une classe d'équivalence d'ensembles équipotents. Ainsi $0 := \mathrm{card}(\emptyset) = \{\emptyset\}$ est la classe de tous les ensembles vides (en fait, il n'y en a qu'un seul), $1 := \mathrm{card}(\{\emptyset\})$ est la classe de tous les ensembles contenant un élément exactement, $2 := \mathrm{card}(\{\emptyset,\{\emptyset\}\})$ est la classe de tous les ensembles contenant deux éléments exactement, etc. Bien que plus abstraite, cette approche mène, bien sûr, au même but.

2.3. Constructions récursives. Comme technique fondamentale et omniprésente, nos axiomes permettent d'établir la *construction récursive*. C'est exactement ce théorème auquel nous faisons appel quand nous construisons une suite itérative qui commence par x_0 puis procède « par récurrence » : $x_{k+1} = f(x_k)$ pour $k = 0,1,2,3,\ldots$. En voici une formulation explicite :

Théorème 2.8 (Dedekind, 1888)**.** *Soient X un ensemble, $x_0 \in X$ un élément, et $f\colon X \to X$ une application. Alors il existe une unique application $g\colon \mathbb{N} \to X$ vérifiant $g(0) = x_0$ et $g(\mathbf{S}n) = f(g(n))$ pour tout $n \in \mathbb{N}$.*

L'unicité découle de l'axiome de récurrence (exercice facile mais bénéfique). L'existence d'une telle application g est beaucoup moins évidente : elle nécessite une *construction* !

Remarque 2.9 (La construction naïve ne marche pas)**.** La première idée qui vient à l'esprit est la construction suivante : pour 0 on pose $g_0\colon \{0\} \to X$, $g(0) = x_0$. Ayant construit $g_n\colon \{0,\ldots,n\} \to X$ on définit $g_{\mathbf{S}n}\colon \{0,\ldots,n,\mathbf{S}n\} \to X$ par $g_{\mathbf{S}n}(k) = g_n(k)$ pour tout $k \leq n$ puis on prolonge par $g_{\mathbf{S}n}(\mathbf{S}n) = f(g_n(n))$. On obtiendrait ainsi une suite $(g_n)_{n\in\mathbb{N}}$, et la réunion $g = \bigcup_{n\in\mathbb{N}} g_n$ nous fournirait l'application cherchée $g\colon \mathbb{N} \to X$. Le problème avec cette démarche est qu'elle est *circulaire* : elle utilise le principe de construction récursive pour justifier ce même principe. (Le détailler.) Ce n'est donc pas une preuve.

La construction naïve échoue mais illustre bien à quel point le principe de récursion nous est naturel et utile. D'autant plus est-il important de *prouver* que ce principe est toujours applicable. Voici l'argument qui marche :

PREUVE D'EXISTENCE. Formellement, une application $g\colon A \to B$ d'un ensemble de départ A vers un ensemble d'arrivé B est une relation binaire $g \subset A \times B$ telle que pour tout $a \in A$ il existe un et un seul élément $b \in B$ avec $(a,b) \in g$. Existence et unicité permettent de définir b comme *l'image* de $a \in A$ par l'application g, notée $g(a)$. On dit aussi que l'application g envoie a sur $b = g(a)$.

Dans notre cas on part d'une application $f\colon X \to X$ et d'un élément initial $x_0 \in X$. Notre objectif est de montrer l'existence d'une application $g\colon \mathbb{N} \to X$ vérifiant $g(0) = x_0$ et $g(\mathbf{S}n) = f(g(n))$ pour tout $n \in \mathbb{N}$.

Une partie $R \subset \mathbb{N} \times X$ sera appelée *inductive* si $(0,x_0) \in R$ et que pour tout $(n,t) \in R$ on a $(\mathbf{S}n, f(t)) \in R$. On constate que le produit $\mathbb{N} \times X$ tout entier est inductif. En plus, si $(R_\lambda)_{\lambda\in\Lambda}$ est une famille de parties inductives $R_\lambda \subset \mathbb{N} \times X$, alors leur intersection $R = \bigcap_{\lambda\in\Lambda} R_\lambda$ est à nouveau une partie inductive.

Soit $g \subset \mathbb{N} \times X$ l'intersection de toutes les parties inductives. D'après ce qui précède, c'est une partie inductive, et la plus petite par construction. Il nous reste à montrer qu'il s'agit d'une application. Regardons donc l'ensemble

$$E = \{n \in \mathbb{N} \mid \text{il existe un et un seul } x \in X \text{ tel que } (n,x) \in g\}.$$

Nous allons prouver par récurrence que $E = \mathbb{N}$. Vérifions d'abord que $0 \in E$. On sait déjà que $(0,x_0) \in g$ puisque g est inductive. S'il existait une deuxième paire $(0,x) \in g$ avec $x \neq x_0$, alors on pourrait l'enlever et $g \smallsetminus \{(0,x)\}$ serait encore inductive mais plus petite que g, ce qui est absurde. (Le détailler.) On a donc bien $0 \in E$, comme souhaité.

Vérifions ensuite que $n \in E$ entraîne $\mathbf{S}n \in E$. L'hypothèse $n \in E$ nous dit qu'il existe un unique $x \in X$ tel que $(n,x) \in g$. Ceci entraîne que $(\mathbf{S}n, f(x)) \in g$, puisque g est inductive. S'il existait une deuxième paire $(\mathbf{S}n,y) \in g$ avec $y \neq f(x)$, alors on pourrait l'enlever et $g \smallsetminus \{(\mathbf{S}n,y)\}$ serait encore inductive mais plus petite que g, ce qui est absurde. (Le détailler.) On a donc bien $\mathbf{S}n \in E$, comme souhaité.

On conclut que $E = \mathbb{N}$ par l'axiome de récurrence. Autrement dit, $g \subset \mathbb{N} \times X$ est bien une application de $\mathbb{N}$ vers X. Le fait que g soit inductive se reformule maintenant comme $g(0) = x_0$ et $g(\mathbf{S}n) = f(g(n))$ pour tout $n \in \mathbb{N}$. □

Le théorème de construction récursive a d'innombrables applications. En voici la première : la définition 2.1 caractérise les nombres naturels de manière univoque (exercice) :

Corollaire 2.10 (Unicité des nombres naturels)**.** *Soit N' un ensemble, soit $0' \in N'$ un élément et soit $\mathbf{S}'\colon N' \to N'$ une application. Si le triplet $(N',0',\mathbf{S}')$ vérifie aux axiomes de la définition 2.1, alors il existe une unique bijection $\psi\colon \mathbb{N} \to N'$ telle que $\psi(0) = 0'$ et $\psi\mathbf{S} = \mathbf{S}'\psi$. On dit ainsi que $(\mathbb{N},0,\mathbf{S})$ et $(N',0',\mathbf{S}')$ sont canoniquement isomorphes.* □

Ce résultat est très rassurant : il vous laisse toute liberté de choisir votre modèle préféré des nombres naturels, la seule restriction étant bien sûr qu'il doit satisfaire aux axiomes ci-dessus. Si jamais quelqu'un d'autre choisit un autre modèle, ce dernier est forcément isomorphe au vôtre : seuls les « noms » des éléments ont changé, et la bijection ψ en fait la traduction. À titre d'analogie : on compte essentiellement de la même manière dans toute les langues bien que les mots utilisés soient différents.

2.4. Addition. Toute la structure et la richesse de la théorie des nombres sont contenues dans les trois axiomes de la définition 2.1, qui est la distillation de plusieurs millénaires d'activité mathématique. Cette définition précise ce que l'on entend par compter. Esquissons maintenant comment *construire* l'arithmétique des nombres naturels pour ensuite *prouver* toutes les propriétés de base, si utiles dans la pratique :

Proposition 2.11 (Addition)**.** *Il existe une et une seule application $+\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ définie par $m + 0 := m$ puis par récurrence par $m + \mathbf{S}n := \mathbf{S}(m+n)$ pour tout $m,n \in \mathbb{N}$. Cette opération $+$, appelée addition, est associative et commutative, admet 0 comme élément neutre bilatéral, et $m + k = n + k$ implique $m = n$.*

Remarquons que $n + 1 = \mathbf{S}n$ pour tout $n \in \mathbb{N}$, ce qui permet de remplacer la fonction $\mathbf{S}$ par la notation $n \mapsto n + 1$ dans la suite. Soulignons que les propriétés énoncées ne sont pas des hypothèses, mais peuvent être déduites des définitions et des axiomes. À titre d'illustration, nous esquissons ici une preuve de la proposition. Dans la suite les vérifications d'énoncés similaires seront laissées au lecteur.

DÉMONSTRATION. Existence et unicité découlent du principe de récurrence (exercice). Montrons d'abord l'associativité. Soit $E = \{n \in \mathbb{N} \mid a + (b + n) = (a + b) + n \text{ pour tout } a,b \in \mathbb{N}\}$. On constate que $0 \in E$ car $a + (b + 0) = a + b = (a + b) + 0$. Ensuite, $n \in E$ entraîne $\mathbf{S}n \in E$ car $a + (b + \mathbf{S}n) = a + \mathbf{S}(b + n) = \mathbf{S}(a + (b + n)) = \mathbf{S}((a + b) + n) = (a + b) + \mathbf{S}n$. Ainsi on conclut que $E = \mathbb{N}$, donc l'addition est associative.

Nous avons $n+0=n$ par définition ; montrons que $0+n=n$. Soit $E=\{n\in\mathbb{N}\mid 0+n=n\}$. On a $0+0=0$, donc $0\in E$. Pour $n\in E$ on a $0+n=n$, donc $0+\mathbf{S}n=\mathbf{S}(0+n)=\mathbf{S}n$, ce qui veut dire que $\mathbf{S}n\in E$. On conclut que $E=\mathbb{N}$, autrement dit 0 est l'élément neutre bilatéral.

De manière analogue, nous avons $n+1=\mathbf{S}n$ par définition ; montrons que $1+n=\mathbf{S}n$. Soit $E=\{n\in\mathbb{N}\mid 1+n=\mathbf{S}n\}$. On a $0\in E$ car $1+0=1=\mathbf{S}0$. Ensuite, $n\in E$ entraîne $\mathbf{S}n\in E$ car $1+\mathbf{S}n=\mathbf{S}(1+n)=\mathbf{SS}n$.

La commutativité en découle comme suit : soit $E=\{n\in\mathbb{N}\mid a+n=n+a \text{ pour tout } a\in\mathbb{N}\}$. On a déjà montré que $0\in E$ ainsi que $1\in E$. Finalement $n\in E$ entraîne $\mathbf{S}n\in E$ car $a+\mathbf{S}n=a+(1+n)=(a+1)+n=n+(a+1)=n+(1+a)=(n+1)+a=\mathbf{S}n+a$. Ceci prouve la commutativité.

La vérification que $m+k=n+k$ implique $m=n$ est laissée en exercice. □

2.5. Multiplication. De manière analogue on établit les propriétés de la multiplication :

Proposition 2.12 (Multiplication)**.** *Il existe une et une seule application* $\cdot\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ *définie par* $m\cdot 0:=0$ *puis par récurrence* $m\cdot\mathbf{S}n:=m\cdot n+m$ *pour tout* $m,n\in\mathbb{N}$. *Cette opération* $\cdot$*, appelée multiplication, est associative, commutative, et admet* 1 *comme élément neutre bilatéral. Si* $k\neq 0$*, alors* $m\cdot k=n\cdot k$ *implique* $m=n$*. La multiplication est distributive sur l'addition, c'est-à-dire* $k\cdot(m+n)=(k\cdot m)+(k\cdot n)$. □

Proposition 2.13 (Exponentiation)**.** *Il existe une et une seule application* $\hat{}\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ *définie par* $a^0:=1$ *puis par récurrence* $a^{\mathbf{S}n}:=a^n\cdot a$ *pour tout* $a,n\in\mathbb{N}$. *Cette opération* $\hat{}$*, appelée exponentiation, jouit des propriétés* $a^1=a$ *et* $a^{m+n}=a^m\cdot a^n$ *et* $(a^m)^n=a^{m\cdot n}$. *Si* $a\notin\{0,1\}$*, alors* $a^m=a^n$ *implique* $m=n$. □

Notation. La multiplication $a\cdot b$ est souvent abrégée par ab. Comme d'habitude, l'associativité permet d'économiser certaines parenthèses : on écrit $a+b+c$ pour $(a+b)+c$ ou $a+(b+c)$, et de même abc pour $(ab)c$ ou $a(bc)$. Finalement, on écrit $ab+c$ pour $(a\cdot b)+c$, et ab^n pour $a\cdot(b^n)$. On sous-entend que l'exponentiation est prioritaire sur la multiplication, et la multiplication est prioritaire sur l'addition.

2.6. Ordre. Pour travailler commodément avec les nombres naturels il nous faut encore de l'ordre. On définit la relation $\leq$ en posant $m\leq n$ si et seulement s'il existe $k\in\mathbb{N}$ tel que $m+k=n$. Il s'agit effectivement d'un ordre, c'est-à-dire que $n\leq n$ (réflexivité), $m\leq n$ et $n\leq m$ impliquent $m=n$ (antisymétrie), puis $k\leq m$ et $m\leq n$ impliquent $k\leq n$ (transitivité). Les vérifications sont laissées en exercice.

Proposition 2.14. *L'ensemble* $\mathbb{N}$ *muni de la relation* $\leq$ *est bien ordonné : tout sous-ensemble* $X\subset\mathbb{N}$ *non vide admet un plus petit élément, c'est-à-dire il existe* $m\in X$ *tel que* $m\leq x$ *pour tout* $x\in X$.

DÉMONSTRATION. Soit $M:=\{n\in\mathbb{N}\mid n\leq x \text{ pour tout } x\in X\}$. C'est l'ensemble des minorants de X. Évidemment on a $0\in M$. Par hypothèse X est non-vide, il existe donc au moins un élément $x\in X$: ainsi on a $\mathbf{S}x\notin M$ et donc $M\neq\mathbb{N}$. Par conséquent il existe un $m\in M$ de sorte que $\mathbf{S}m\notin M$. (Sinon on aurait $M=\mathbb{N}$ par l'axiome de récurrence.) Si $m\notin X$, alors $\mathbf{S}m$ serait encore un minorant, ce qui n'est pas le cas. Donc $m\in M\cap X$ est un plus petit élément de X, comme énoncé. □

Remarque 2.15. Un ensemble bien ordonné est en particulier totalement ordonnée : pour tout m,n on a soit $m<n$, soit $m=n$, soit $m>n$. (Exercice : il suffit de regarder le plus petit élément de $\{m,n\}$.)

Notation. Par commodité on définit la relation stricte $m<n$ par $m\leq n$ et $m\neq n$. On définit aussi les ordres inverses, $n\geq m$ par $m\leq n$, et $n>m$ par $n<m$.

Proposition 2.16. *L'addition et la multiplication respectent l'ordre dans le sens que* $m\leq n$ *implique* $m+k\leq n+k$ *ainsi que* $mk\leq nk$*. Il en est de même pour l'inégalité stricte :* $m<n$ *implique* $m+k<n+k$*, et aussi* $mk<nk$ *pourvu que* $k\neq 0$*. Si* $m<n$ *et* $k\neq 0$ *alors* $m^k<n^k$ *; si de plus* $a\notin\{0,1\}$*, alors* $a^m<a^n$. □

Définition 2.17. La soustraction $n-m$ n'est définit que si $n\geq m$. Dans ce cas il existe $k\in\mathbb{N}$ tel que $m+k=n$. Ce nombre k étant unique, on peut donc définir $n-m:=k$.

2.7. Divisibilité. Ajoutons que l'on peut définir la division de manière analogue à la soustraction :

- On définit la relation $\mid$ en posant $m\mid n$ si et seulement s'il existe $k\in\mathbb{N}$ tel que $mk=n$. Il s'agit effectivement d'un ordre, c'est-à-dire que $n\mid n$ (réflexivité), $m\mid n$ et $n\mid m$ impliquent $m=n$ (antisymétrie), puis $k\mid m$ et $m\mid n$ impliquent $k\mid n$ (transitivité).
- L'ordre $\mid$ n'est pas total : on n'a ni $2\mid 3$ ni $3\mid 2$. Par conséquent, $\mathbb{N}$ n'est pas bien ordonné par rapport à l'ordre $\mid$, car $\{2,3\}$ n'a pas de plus petit élément. (On y reviendra quand on parlera du pgcd.)

– Toutefois, 1 est le plus petit élément de $\mathbb{N}$ par rapport à l'ordre $|$, car $1 \mid n$ pour tout n, et 0 en est le plus grand car $n \mid 0$ pour tout n.
– Les éléments minimaux de $\mathbb{N} \smallsetminus \{1\}$ par rapport à la divisibilité $|$ sont appelés irréductibles (ou premiers) ; il s'agit effectivement des nombres premiers dans le sens usuel (le détailler).
– La multiplication respecte l'ordre $|$ dans le sens que $m|n$ implique $mk \mid nk$, mais l'addition ne la respecte pas : on a $1 \mid 2$ mais non $1+1 \mid 2+1$.
– La division n/m n'est définit que si $m \mid n$. Dans ce cas il existe $k \in \mathbb{N}$ tel que $mk = n$. À l'exception du cas $m = n = 0$ le nombre k est unique, et on peut donc définir $n/m := k$.

2.8. Division euclidienne. Après le bref résumé des fondements, nous sommes en mesure de déduire un résultat célèbre, qui nous intéressera tout particulièrement dans la suite :

Proposition 2.18 (Division euclidienne dans $\mathbb{N}$)**.** *Soient $a, b \in \mathbb{N}$ deux nombres naturels avec $b \neq 0$. Alors il existe une et une seule paire $(q,r) \in \mathbb{N} \times \mathbb{N}$ vérifiant $a = b \cdot q + r$ et $0 \leq r < b$.*

DÉMONSTRATION. *Existence.* — On fixe un nombre naturel $b \neq 0$. Soit E l'ensemble des nombres naturels $a \in \mathbb{N}$ qui s'écrivent comme $a = b \cdot q + r$ avec $q, r \in \mathbb{N}$ et $0 \leq r < b$. Nous allons montrer par récurrence que $E = \mathbb{N}$. Évidemment $0 \in E$ car $0 = b \cdot 0 + 0$ et $0 < b$. Montrons que $a \in E$ implique $a+1 \in E$. L'hypothèse $a \in E$ veut dire qu'il existe $q, r \in \mathbb{N}$ tels que $a = b \cdot q + r$ et $r < b$. On distingue deux cas :
– Si $r + 1 < b$, alors $a + 1 = b \cdot q + (r+1)$ est de la forme exigée, donc $a + 1 \in E$.
– Si $r + 1 = b$, alors $a + 1 = b \cdot (q+1) + 0$ est de la forme exigée, donc $a + 1 \in E$.
On conclut que tout nombre naturel $a \in \mathbb{N}$ s'écrit comme $a = b \cdot q + r$ avec $q, r \in \mathbb{N}$ et $0 \leq r < b$.

Unicité. — Supposons que $b \cdot q + r = b \cdot q' + r'$ avec $r, r' < b$. Quitte à échanger (q,r) et (q',r') nous pouvons supposer $r \leq r'$. On obtient ainsi $b \cdot (q - q') = r' - r$. On constate que $r' - r < b$. Si $q - q' \geq 1$ on aurait $b \cdot (q - q') \geq b$, ce qui est impossible. Il ne reste que la possibilité $q - q' = 0$, ce qui entraîne $r' - r = 0$. On conclut que $q = q'$ et $r = r'$, comme énoncé. □

2.9. Numération à position. Voici un premier exploit de nos efforts :

Corollaire 2.19 (Numération en base b)**.** *On fixe un nombre naturel $b \geq 2$. Alors tout nombre naturel $a \geq 1$ s'écrit de manière unique comme $a = a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0$ où $a_0, a_1, \ldots, a_{n-1}, a_n$ sont des nombres naturels vérifiant $0 \leq a_k < b$ pour tout $k = 0, \ldots, n$ ainsi que $a_n \neq 0$.* □

Exercice 2.20. Déduire le corollaire de la proposition.

Remarque 2.21. De manière abrégée on écrit aussi $\langle a_n, a_{n-1}, \ldots, a_1, a_0 \rangle_b := a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0$. Lorsque b est assez petit, on peut représenter chaque a_i avec $0 \leq a_i < b$ par un symbole distinctif, appelé *chiffre*. Pour nos besoins, la base b vaut au plus seize, et les chiffres 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F nous serviront d'abréviations commodes pour les seize premiers nombres naturels 0, **S**0, **SS**0, **SSS**0, …. En système décimal ceci permet d'écrire $\langle 1,7,2,9 \rangle_{\text{dec}}$, puis de supprimer les virgules et d'écrire $\langle 1729 \rangle_{\text{dec}}$ ou 1729 tout court, si le contexte laisse comprendre sans équivoque que nous travaillons en base dix. On arrive ainsi à exprimer, de manière unique, tous les nombres naturels à l'aide de très peu de symboles seulement par une notation très courte. C'est simple et efficace, mais il fallait y penser !

Exercice 2.22. Après cette longue marche partant des axiomes, nous sommes finalement arrivés à la notation décimale usuelle de nombres naturels. Cette promenade n'est pas vaine : nous nous sommes ainsi rassurés des fondements de l'arithmétique que nous voulons implémenter. Au chapitre II nous en avons considéré deux implémentations : celle qui traduit littéralement les définitions de cet annexe, puis celle, bien plus efficace, qui travaille directement sur les décimales suivant les algorithmes connus de l'école. Si vous voulez, vous pouvez maintenant *justifier* que ces algorithmes sont corrects, c'est-à-dire *prouver* qu'ils rendent le résultat exigé par la définition.

Corollaire 2.23 (Numération en base mixte)**.** *Soit $(b_k)_{k \in \mathbb{N}}$ une suite infinie de nombres naturels avec $b_k \geq 2$ pour tout $k \in \mathbb{N}$. Alors tout nombre naturel $a \geq 1$ s'écrit de manière unique comme*

$$a = a_n b_{n-1} \cdots b_0 + a_{n-1} b_{n-2} \cdots b_0 + \cdots + a_1 b_0 + a_0$$

avec $a_n \neq 0$ et $0 \leq a_k < b_k$ pour tout $k = 0, \ldots, n$. □

Exemple 2.24. La numération en base mixte est plus répandue que l'on ne pense. La durée de 2 semaines, 3 jours, 10 heures, 55 minutes et 17 secondes, par exemple, représente

$$2\cdot 7\cdot 24\cdot 60\cdot 60+3\cdot 24\cdot 60\cdot 60+10\cdot 60\cdot 60+55\cdot 60+17=1508117$$

secondes. Existence et unicité d'une telle écriture sont quotidiennement utilisées, prérequis sans lesquels ce système n'aurait pas pu s'établir dans l'histoire humaine. Nous y reviendrons au chapitre IV.

3. Construction des nombres entiers

Dans $\mathbb{N}$ certaines équations comme $5+x=0$ n'ont pas de solution. Pour remédier à ce défaut il est naturel de « compléter » $\mathbb{N}$ par des éléments nouveaux, que l'on appellera « nombres négatifs », afin de pouvoir résoudre toute équation de la forme $a=b+x$ avec $a,b\in\mathbb{N}$.

Remarque 3.1. La manière ad hoc de le faire est de considérer l'ensemble $\{\pm\}\times\mathbb{N}$ puis d'identifier $+n$ avec n ainsi que $+0$ avec -0. Ainsi on rajoute à l'ensemble $\mathbb{N}=\{0,1,2,3,\ldots\}$ les éléments $-1,-2,-3,\ldots$. On notera l'ensemble qui en résulte par $\mathbb{Z}$. Jusqu'ici tout marche bien : on peut par exemple décréter que $5+(-5)=0$. La grande difficulté est d'étendre les structures de $\mathbb{N}$ à $\mathbb{Z}$ de manière cohérente : pour cela il faudra construire une addition, une multiplication, puis une soustraction, une division, un ordre, etc… qui étendent les structures de $\mathbb{N}$. Tout cela est possible, mais les preuves détaillées sont assez fatigantes. Le problème est que la construction ad hoc ne facilite pas la transition de $\mathbb{N}$ à $\mathbb{Z}$, et la distinction des cas $n\in\mathbb{N}$ et $n\in\mathbb{Z}\smallsetminus\mathbb{N}$ mène à des preuves inutilement compliquées. Bien que fastidieuse pour les preuves, c'est cette approche qui est souvent favoriser pour une implémentation sur ordinateur. Ne méprisez donc pas cette idée un peu naïve : elle est sous-optimale pour la théorie, mais elle marche très bien dans l'application pratique. C'est d'ailleurs cette approche que l'on utilise quotidiennement dans les calculs.

La construction mathématique suivante est plus élégante mais aussi un peu plus abstraite. Elle mérite toutefois de l'attention car elle nous mène aux bonnes preuves et servira de modèle dans d'autres cas.

Reprenons l'équation $a=b+x$ où $a,b\in N$. On aimerait que les nombres entiers, encore à construire, fournissent une solution x à chaque équation de ce type. Une fois construit, tout nombre entier s'écrirait donc comme différence $a-b$ pour deux nombres naturels $a,b\in\mathbb{N}$. Notons toutefois qu'une telle écriture n'est pas unique : on a $a-b=c-d$ si et seulement si $a+d=c+b$.

Proposition 3.2. *Sur $\mathbb{N}\times\mathbb{N}$ on définit la relation $\sim$ par $(a,b)\sim(c,d)$ si et seulement si $a+d=c+b$. C'est une relation d'équivalence. On note $\mathbb{Z}:=(\mathbb{N}\times\mathbb{N})/_\sim$ l'ensemble quotient.*

Proposition 3.3. *Sur $\mathbb{Z}$ on peut définir une opération $+\colon\mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$ par $[a,b]+[c,d]:=[a+c,b+d]$. Cette opération, appelée* addition, *est associative, commutative, et admet $[0,0]$ pour élément neutre bilatéral. Tout élément $[a,b]\in\mathbb{Z}$ admet $[b,a]\in\mathbb{Z}$ pour opposé : $[a,b]+[b,a]=[a+b,a+b]=[0,0]$.* □

Proposition 3.4. *Sur $\mathbb{Z}$ on peut définir une opération $\cdot\colon\mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$ par $[a,b]\cdot[c,d]:=[ac+bd,ad+bc]$. Cette opération, appelée* multiplication, *est associative, commutative, et admet $[1,0]$ comme élément neutre bilatéral. La multiplication est distributive sur l'addition, c'est-à-dire $x\cdot(y+z)=(x\cdot y)+(x\cdot z)$ pour tout $x,y,z\in\mathbb{Z}$. La multiplication est sans diviseur de zéro, c'est-à-dire $x\cdot y=0$ implique $x=0$ ou $y=0$.* □

Proposition 3.5. *Sur $\mathbb{Z}$ on peut définir une relation $\leq$ en posant $[a,b]\leq[c,d]$ si et seulement si $a+d\leq c+b$. Ceci définit un ordre total sur $\mathbb{Z}$. Pour tout $x,y,z\in\mathbb{Z}$ on a que $x\leq y$ implique $x+z\leq y+z$, et en plus $x\leq y$ et $0\leq z$ impliquent $x\cdot z\leq y\cdot z$.*

Proposition 3.6. *L'application $\phi\colon\mathbb{N}\to\mathbb{Z}$, $a\mapsto[a,0]$ est injective et permet donc d'identifier $\mathbb{N}$ avec son image $\phi(\mathbb{N})$. Cette application respecte l'addition, $\phi(a+b)=\phi(a)+\phi(b)$, la multiplication, $\phi(a\cdot b)=\phi(a)\cdot\phi(b)$, et l'ordre : $\phi(a)\leq\phi(b)$ si et seulement si $a\leq b$. Avec cette identification on retrouve finalement les nombres naturels $\mathbb{N}$ comme sous-ensemble des nombres entiers $\mathbb{Z}$, comme souhaité.* □

Proposition 3.7 (division euclidienne)**.** *Pour tout $a,b\in\mathbb{Z}$, $b>0$, il existe une unique paire $(q,r)\in\mathbb{Z}\times\mathbb{Z}$ de sorte que $a=bq+r$ et $0\leq r<b$.* □

Comme pour les nombres naturels, il est vivement conseillé d'avoir travaillé la construction des nombres entiers au moins une fois dans sa vie. Si vous cherchez un défi, vous pouvez ensuite construire le corps $\mathbb{Q}$ des nombres rationnels selon les mêmes lignes, en partant de l'équation $a=b\cdot x$ (cf. chap. XII).

Even fairly good students, when they have obtained the solution of the problem and written down neatly the argument, shut their books and look for something else. Doing so, they miss an important and instructive phase of the work. (…) No problem whatever is completely exhausted.
George Pólya (1887 - 1985), *How to Solve It*

PROJET II

Multiplication rapide selon Karatsuba

1. Peut-on calculer plus rapidement ?

D'après notre brève révision des algorithmes scolaires, on pourrait poser une question provocatrice : peut-on calculer plus rapidement ? de manière significative ?

Une chose est claire : l'addition de a et b à n décimales produit un résultat à n ou $n+1$ décimales ; déjà pour l'écrire il faut donc au moins une boucle de longueur n, et on ne peut pas espérer de faire mieux.

Quant à la multiplication, par contre, ce n'est pas si évident : la multiplication de a et b à n décimales produit un résultat à $2n$ ou $2n-1$ décimales, ce qui entraîne une complexité *au moins linéaire*. Avec l'algorithme scolaire on a une solution *au plus quadratique*. A priori il reste donc une marge pour l'optimisation.

D'après la légende, Kolmogorov posa la question de la multiplication vers 1962 dans son séminaire, convaincu que même le meilleur algorithme sera forcément de complexité quadratique (comme la multiplication scolaire). Un des participants, A. Karatsuba, découvrit aussitôt une méthode bien meilleure.

2. L'algorithme de Karatsuba

L'idée de Karatsuba est une incarnation du paradigme « diviser pour régner », aussi simple que géniale : on suppose donnés deux nombres naturels a et b sous forme de leur développement décimal, chacun à $2n$ chiffres au plus. On les décompose comme

$$a = a_0 + a_1 10^n \qquad \text{et} \qquad b = b_0 + b_1 10^n,$$

où a_0 et b_0 sont compris dans l'intervalle $[\![0, 10^n[\![$, autrement dit, a_0 et b_0 contiennent les n chiffres « bas », alors que a_1 et b_1 contiennent les chiffres « hauts ». Ce découpage peut être vu comme une division euclidienne par 10^n, mais en base 10 c'est juste une action de « copier-coller ». Ce décalage d'indices est peu coûteux (de complexité linéaire). Bien évidemment, le produit ab de type $2n \times 2n$ est égal à

$$ab = (a_0 b_0) + (a_0 b_1 + a_1 b_0)10^n + (a_1 b_1)10^{2n}$$

ce qui nécessite a priori quatre multiplications de type $n \times n$. Jusqu'ici rien de surprenant. Voici l'astuce de Karatsuba : on peut arriver au même résultat avec trois multiplications seulement ! On calcule

$$s \leftarrow a_0 b_0, \quad t \leftarrow a_1 b_1, \quad u \leftarrow (a_0 + a_1)(b_0 + b_1) - s - t.$$

Puis on constate que $u = a_0 b_1 + a_1 b_0$, donc le produit cherché est

$$ab = s + u10^n + t10^{2n}$$

À nouveau les multiplications par 10^n et 10^{2n} ne sont que des décalages d'indices peu coûteux. Au total on n'effectue que trois multiplications de type $n \times n$. Certes, on a trois additions/soustractions supplémentaires, mais quand n est grand le gain d'une multiplication l'emportera. Mieux encore : on peut appliquer cette méthode de manière récursive pour encore économiser de la même manière sur les multiplications du type $n \times n$, et ainsi de suite, jusqu'à une taille raisonnablement petite pour la multiplication scolaire.

3. Analyse de complexité

Regardons de plus près la complexité de cet algorithme. Soit $c\colon \mathbb{N} \to \mathbb{N}$ le coût de la multiplication selon Karatsuba, mesuré en nombre d'opérations sur les chiffres. Alors on a la majoration

$$c(2n) \leq 3c(n) + \alpha n$$

Ici $3c(n)$ est le coût des 3 multiplications de taille n, puis αn est le coût linéaire des additions/soustractions.

Proposition 3.1. *Soit $c\colon \mathbb{N} \to \mathbb{N}$ une fonction croissante qui vérifie $c(2n) \leq 3c(n) + \alpha n$ ainsi que $c(1) \leq \beta$. Alors on a la majoration $c(2^k) \leq 3^k(\alpha+\beta)$ pour tout $k \in \mathbb{N}$. Celle-ci entraîne, pour tout $n \in \mathbb{N}$, la majoration $c(n) < 3(\alpha+\beta)n^\varepsilon$ avec exposant $\varepsilon = \log 3/\log 2 \approx 1,585$.*

DÉMONSTRATION. On a $c(1) \leq \beta$, et on calcule $c(2) \leq 3\beta + \alpha$, puis $c(2^2) \leq 3^2\beta + (3+2)\alpha$, puis $c(2^3) = 3^3\beta + (3^2 + 3\cdot 2 + 2^2)\alpha$, etc. Par récurrence on établit la majoration

$$c(2^k) \leq 3^k\beta + \sum_{i=0}^{k-1} 3^{k-1-i}2^i\alpha = 3^k(\alpha+\beta) - 2^k\alpha \leq 3^k(\alpha+\beta).$$

L'égalité au milieu découle de la formule $\sum_{i=0}^{k-1} a^{k-1-i}b^i = \frac{a^k-b^k}{a-b}$ spécialisée en $a=3$ et $b=2$. Pour tout $n \geq 2$ on a $n \leq 2^k$ avec $k = \lceil \log_2 n \rceil < 1 + \log_2 n$, donc $3^k < 3^{1+\log_2 n} = 3n^{\log_2 3}$. Par notre hypothèse de monotonie de la fonction c, on conclut que $c(n) \leq c(2^k) < 3(\alpha+\beta)n^{\log_2 3}$. □

Corollaire 3.2. *Le temps pour multiplier deux nombres naturels à n décimales selon la méthode de Karatsuba est majoré par γn^ε avec un exposant $\varepsilon = \log_2 3 \approx 1,585$ et une constante $\gamma > 0$. Seule la constante γ dépend des détails de l'implémentation, de la vitesse de l'ordinateur, etc. Quand n est grand, cette méthode est donc une amélioration significative de la méthode quadratique.* □

4. Implémentation et test empirique

La méthode de Karatsuba est facile à implémenter mais il faut faire attention aux détails (voir le programme II.8 plus bas). Des expériences montrent que la méthode scolaire est plus rapide pour les petits nombres, mais à long terme c'est Karatsuba qui gagne : chaque fois que la longueur double, le temps d'exécution est multiplié par 4 pour la méthode scolaire, mais seulement par 3 pour Karatsuba !

Exercice/P 4.1. Comparer empiriquement la performance de la multiplication scolaire et la méthode de Karatsuba, à l'aide du programme `karatsuba-test.cc`. Qu'observez-vous ?

☞ *Ce projet est encore très incomplet...* ✍

Programme II.8 Multiplication rapide selon Karatsuba

```
void decouper( const Naturel& n, Indice e, Naturel& n1, Naturel& n0 )
{
  // Cas exceptionnel: tout rentre dans la partie basse
  n0.clear(); n1.clear();
  Indice taille= n.size();
  if ( e >= taille ) { n0= n; return; };

  // Copier la partie haute (comme n est normalisé, n1 l'est aussi)
  n1.chiffres.resize( taille-e );
  for( Indice i=0, j=e; j<taille; ++i, ++j ) n1.chiffres[i]= n.chiffres[j];

  // Déterminer la partie basse en supprimant d'éventuels zéros terminaux
  Indice i= e-1;
  while( i>=0 && n.chiffres[i]==Chiffre(0) ) --i;
  n0.chiffres.resize(i+1);
  for( ; i>=0; --i ) n0.chiffres[i]= n.chiffres[i];
}

bool karatsuba( const Naturel& a, const Naturel& b, Naturel& produit )
{
  // Déléguer les petites multiplications à la méthode scolaire
  if ( a.size() < 40 || b.size() < 40 )
    { multiplication_scolaire( a, b, produit ); return true; };

  // Déterminer une taille de coupure convenable
  Indice m= max( a.size(), b.size() );
  if ( m%2 == 1 ) ++m;
  Indice n= m/2;

  // Découper a et b en deux moitiés
  Naturel a0, a1, b0, b1, s, t, u;
  decouper( a, n, a1, a0 );
  decouper( b, n, b1, b0 );

  // Multiplication récursive selon Karatsuba
  karatsuba( a0, b0, s );
  karatsuba( a1, b1, t );
  karatsuba( a0+a1, b0+b1, u );
  u-= s; u-= t;

  // Mettre s et t bout à bout dans produit
  produit.chiffres.clear();
  produit.chiffres.resize( a.size()+b.size(), Chiffre(0) );
  for( Indice i=0; i<s.size(); ++i ) produit.chiffres[i]  = s.chiffres[i];
  for( Indice i=0; i<t.size(); ++i ) produit.chiffres[m+i]= t.chiffres[i];

  // Ajouter u au milieu du produit
  Indice i= n; Chiffre retenue= 0;
  for( Indice j=0; j<u.size(); ++j, ++i )
    {
      produit.chiffres[i]+= u.chiffres[j] + retenue;
      if ( produit.chiffres[i] < base ) { retenue= 0; }
      else { produit.chiffres[i]-= base; retenue = 1; };
    }

  // Stocker une éventuelle retenue à la fin, puis raccourcir le résultat
  if ( retenue > 0 )
    {
      while ( produit.chiffres[i] == Chiffre(base-1) ) produit.chiffres[i++]= 0;
      ++produit.chiffres[i];
    };
  produit.raccourcir(); return true;
}
```

Numbers have neither substance, nor meaning, nor qualities.
They are nothing but marks, and all that is in them
we have put into them by the simple rule of straight succession.
Hermann Weyl (1885 - 1955), *Mathematics and the Laws of Nature*

CHAPITRE III

La bibliothèque GMP

Ce chapitre présente une solution « industrielle » du problème des grands entiers en C++ : la bibliothèque GMP. Par rapport à notre implémentation « faite maison » elle se distingue seulement par son niveau d'optimisation (une journée de programmation pour notre classe `Naturel` contre plusieurs années de développement pour la bibliothèque `GMP`.) En contrepartie nous sommes obligés d'accepter une telle bibliothèque comme une boîte noire : on apprendra facilement son interface mais non son fonctionnement intérieur. (Vous pouvez lire sa documentation en ligne si cela vous intéresse.)

Sommaire

1. Une implémentation « professionnelle » des nombres entiers

On discutera dans ce chapitre l'utilisation d'une classe `Integer`, issue de la bibliothèque GMP (*GNU Multiple Precision Library*), qui modélise les « grands entiers », c'est-à-dire, les nombres entiers sans limitation de taille. Contrairement au chapitre précédent, nous ne tentons pas ici une implémentation nous-mêmes, mais nous nous servirons d'une bibliothèque toute faite.

1.1. Avant-propos. Les bibliothèques les plus courantes offrent des solutions à certains problèmes omniprésents dans la programmation. Il est très commode de s'en servir pour ne pas réinventer la roue. Ainsi, utiliser une bibliothèque de haute qualité offre des avantages importants :

- Les fonctions sont soigneusement implémentées et bien testées,
- en particulier elles sont plus fiables et plus efficaces qu'une solution ad hoc,
- elles fournissent une fonctionnalité assez complète et standardisée,
- le code de votre application ainsi créé sera plus concis et plus lisible.

De manière générale, le partage de bibliothèques de qualité permet de mutualiser des implémentations éprouvées, de minimiser des réimplémentations inutiles, et de se concentrer sur l'essentiel.

☞ *Bref, les bibliothèques c'est bon, mangez-en !* ✍

En contrepartie, avouons qu'il existe aussi des inconvénients :

- L'utilisation d'une bibliothèque demande, bien évidemment, comme investissement initial la lecture du « mode d'emploi » plus ou moins complexe. D'où l'intérêt des bibliothèques bien construites et d'utilisation intuitive !
- Pour ce cours de programmation l'autre inconvénient est d'ordre pédagogique : on utilise souvent une bibliothèque comme une boîte noire, c'est-à-dire, on apprend sa fonctionnalité externe (en particulier l'interface), mais on n'apprend pas comment elle est construite (on particulier on ignore souvent la structure des données et les algorithmes utilisés).

Afin de remédier à ces défauts dans le cas des grands entiers, nous le chapitre II discute les éléments d'une implémentation assez complète dans notre exemple `Naturel`. On ferra pareil dans des chapitres plus avancés : permutations au chap. VI, quelques anneaux au chap. XII, puis des polynômes au chap. XIII, par exemple. Bien qu'il existe de bibliothèques adéquates, on peut ainsi apprendre en détail le développement mathématique, souvent intéressant en lui-même. Pour une application non pédagogique on utilisera plutôt une bibliothèque professionnelle, si possible.

La bibliothèque STL. Le C++ vient déjà avec une bibliothèque standard : elle fait partie du C++ et n'est donc souvent pas remarquée comme bibliothèque à part entière. La bibliothèque la plus connue est sans doute la STL. Elle fut développée par *Hewlett-Packard Company* à partir de 1994, puis par *Silicon Graphics Computer Systems* à partir de 1996. Elle fut standardisée et acceptée comme partie intégrante du langage C++ en 1998 ; tout système conforme à ce standard fournit donc une version de la STL.

La STL offre une classe générique pour modéliser les vecteurs (`vector`) que nous avons regardée au chapitre I. La STL offre aussi d'autres structures de données qui sont fréquemment utilisées et très utiles, suivant l'application envisagée : les listes (`list`), les listes bidirectionnelles (`deque`), les piles (`stack`), les files (`queue`), les files de priorité (`priority_queue`), les ensembles (`set`), et d'autres encore.

Pour en savoir plus vous pouvez consultez la page `www.sgi.com/stl`. Il existe également d'excellentes introductions à la STL ; consultez votre bibliothèque universitaire.

La bibliothèque GMP. Le *GNU Multiple Precision Project* a développé la bibliothèque GMP pour des calculs efficaces en précision arbitraire en C/C++. Elle offre des classes pour les nombres entiers et rationnels, et les nombres à virgule flottante en précision arbitraire. Vous pouvez consultez le site officiel `www.swox.com/gmp` où vous trouverez une ample documentation. La bibliothèque GMP se trouve également sur le site de GNU, `www.gnu.org/manual/gmp`. Avec votre installation sous GNU/Linux vous pouvez également taper `info gmp C++` pour lire la documentation locale en ligne.

Que faut-il pour modéliser les entiers ? En tenant compte de nos expériences précédentes, précisons nos exigences. On souhaite disposer d'un type `Integer` qui modélise les entiers dans le sens suivant :

Représentation et entrée-sortie: Tout d'abord, on veut que tout nombre entier puisse être représenté comme une valeur de type `Integer`. Les opérateurs d'entrée `>>` et de sortie `<<` permettent de lire et d'écrire ces valeurs ; ils traduisent alors entre la représentation externe (l'écriture décimale) et la représentation interne (que nous ignorons).

Opérateurs d'affectation: On voudra utiliser l'opérateur d'affectation `=` avec la syntaxe et la sémantique usuelle : il prend deux arguments, une variable de type `Integer` à gauche et une valeur de type `Integer` à droite, puis il affecte la valeur à la variable.

Conversion de type: La conversion devra permettre d'affecter une valeur du type `int` à une variable `var` de type `Integer`. On pourra donc écrire `var=Integer(1)` pour une conversion explicite ou bien `var=1` pour une conversion implicite.

Opérateurs de comparaison: On voudra utiliser les opérateurs de comparaison `==`, `!=`, ainsi que `<`, `<=`, `>`, `>=` avec la syntaxe usuelle, à savoir : ils prennent deux arguments de type `Integer` et renvoient un résultat de type `bool`, qui correspond à la comparaison dans $\mathbb{Z}$.

Opérateurs arithmétiques: On voudra utiliser les opérateurs arithmétiques `+`, `-`, `*`, `/`, `%` avec la syntaxe usuelle, à savoir : ils prennent deux arguments de type `Integer` et renvoient un résultat de type `Integer`. Quant à leur sémantique, on exige que le résultat corresponde au résultat de l'opération respective sur $\mathbb{Z}$.

Opérateurs mixtes: Les opérateurs `+=`, `-=`, `*=`, `/=`, `%=` devront s'utiliser d'une manière naturelle, de sorte que `a+=b` équivaille à `a=a+b` etc. De même pour les opérateurs `++` et `--`, pour lesquels on exige que `++a` équivaille à `a=a+1` etc. L'intérêt pourra être une meilleure lisibilité, et éventuellement une plus grande performance. (Cela dépend du niveau d'optimisation.)

Ces opérations élémentaires décrivent alors la base requise pour travailler avec des entiers sur ordinateur. D'autres fonctions seraient également souhaitables, comme `factorielle`, `binomial`, `racine`, `pgcd`, `ppcm`, `est_premier`, `factoriser` etc. On en développera quelques-unes dans la suite, mais tout développement ultérieure sera basé sur les opérations élémentaires ci-dessus.

1.2. Les entiers de la bibliothèque GMP. Comme on a vu au chapitre II, implémenter l'arithmétique des nombres entiers est faisable mais très laborieux. Heureusement il existe déjà de telles implémentations, soigneusement testées et optimisées. Nous utiliserons par la suite la classe `mpz_class` de la bibliothèque GMP (*GNU Multiple Precision Library*). Elle satisfait à toutes nos exigences ci-dessus, et de plus les calculs s'effectuent très rapidement. Son utilisation est assez intuitive :

Programme III.1 Exemple d'utilisation de la classe `Integer` `gmp-exemple.cc`

```
1  #include <iostream>         // déclarer l'entrée-sortie standard
2  using namespace std;        // accès direct aux fonctions standard
3
4  // *** Attention : à compiler avec g++ -lgmpxx
5  #include <gmpxx.h>          // accéder à la bibliothèque gmp pour C++
6  typedef mpz_class Integer;  // Integer semble plus parlant que gmp_class
7
8  int main()
9  {
10   cout << "Entrez deux entiers svp : ";
11   Integer a, b;
12   cin >> a >> b;
13   cout << "a   = " << a   << endl << "b   = " << b   << endl
14        << "a+b = " << a+b << endl << "a-b = " << a-b << endl
15        << "a*b = " << a*b << endl;
16   if ( b != 0 ) cout << "a/b = " << a/b << endl
17                      << "a%b = " << a%b << endl;
18   else cout << "La division par zéro n'est pas définie." << endl;
19 }
```

☞ On accède à la bibliothèque GMP pour le C++ en incluant la directive `#include <gmpxx.h>`, qui effectue les *déclarations* nécessaires. Ensuite la compilation s'effectue avec l'option `-lgmpxx` pour établir le lien vers les *définitions* de cette bibliothèque. Sinon, le compilateur émettra une longue liste d'erreurs, réclamant `undefined reference to ...`

☞ Comme vous voyez dans le programme III.1, nous avons renommé la classe `mpz_class` en `Integer`, ce qui semble plus parlant. Si jamais vous voulez remplacer `mpz_class` par une future classe `nouveau_modele`, il suffit de modifier une seule ligne. À condition, bien sûr, que `nouveau_modele` implémente les entiers conformément à notre spécification.

Remarque 1.1. Pour utiliser le type `Integer` avec des constantes, vous pouvez bien évidemment utiliser les constantes littérales du type `int` et les convertir en `Integer`. Pour les constantes plus grandes ce n'est plus jouable. (Essayez d'expliquer pourquoi). On fait alors appel aux chaînes de caractères :

```
Integer a(12345);                      // ok : constructeur à partir d'un int
Integer b("98765432109876543210");     // ok : constructeur à partir d'une chaîne
a = Integer("98765432109876543210");   // ok : conversion explicite
a = "12345678901234567890123456789l"; // ok : conversion implicite
a = 1234567890123456789012345678901234567890;   // constante littérale trop longue !
```

Dans le même esprit, où est l'erreur logique dans le calcul suivant ? Comment le rectifier ?

```
Integer f= 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19*20;
```

Quels sont les principes de cette implémentation ? Rappelons que le chapitre I, partant du type `int`, fut lourdement *orientée machine* : il fallait d'abord comprendre la réalisation du type `int` sur la machine, et ensuite en se demandait dans quelles limites ce type pourrait bien servir pour nos calculs. L'approche *orientée objet* procède dans le sens inverse : on spécifie d'abord les propriétés et opérations nécessaires pour modéliser une classe d'objets (mathématiques ou autres), et ensuite on cherche à les satisfaire par une implémentation convenable. C'est cette deuxième approche que nous poursuivons ici, et notre implémentation du type `Naturel` en était un premier exemple.

En gros, la classe `mpz_class` est implémentée comme notre exemple `Naturel`, en particulier elle utilise un tableau de taille adaptable pour stocker les chiffres d'un nombre entier. La principale différence entre `mpz_class` et notre candidat `Naturel` est la *performance*. En effet, les programmeurs de la bibliothèque GMP ont fait un énorme effort d'optimisation :

– Au niveau algorithmique, la bibliothèque GMP implémente les meilleurs algorithmes connus à ce jour. De nombreuses variantes ont été testées et optimisées, puis les meilleures ont été retenues.

☞ Si le choix de l'algorithme le mieux adapté dépend des données, ce choix se fait au moment de l'exécution, typiquement en fonction de la taille des données. Pour la multiplication notamment on choisira entre scolaire et Karatsuba, puis d'autres encore que nous n'avons pas présentés ici.

– Les fonctions les plus fréquentes (comme l'addition ou la soustraction, ou d'autres routines de base) sont codées en langage machine et non en C/C++, afin d'utiliser le plus efficacement les instructions fournies par le microprocesseur.

  ☞ Cette approche est très laborieuse car on doit réimplémenter ces fonctions sur chaque type de processeur. Ceci n'est justifié que dans les rares cas ou le gain espéré est significatif.

– La représentation interne n'est pas en base 10 mais en base 2, plus précisément en base 2^{32}, pour profiter pleinement de la structure du microprocesseur. (La conversion en base 10 se fait uniquement à l'entrée-sortie, considérée comme peu fréquente et négligeable devant les calculs.)

  ☞ Même pour la GMP le principe de base reste incontournable : plus les nombres sont grands, plus ils occupent de mémoire, et plus les opérations s'effectuent lentement. Mis à part ce constat, la représentation interne ne nous regardera pas dans la suite : l'essentiel est que `mpz_class` fonctionne suivant la spécification ci-dessus.

Tests empiriques. Les résultats sont assez impressionnants : vous pouvez regardez le programme `gmp-chrono.cc` pour comparer la performance de la classe `mpz_class` et notre candidat `Naturel`. Remarquez à ce propos que notre addition semble compétitive (à un facteur constant près), mais la complexité quadratique de la multiplication scolaire se révèle catastrophique pour les nombres de plus en plus grands. La GMP, par contre, utilise les meilleurs algorithmes connus, et arrive ainsi à une complexité quasi-linéaire.

1.3. Exemple pratique : calcul de coefficients binomiaux. Après les opérations de base, regardons une fonction un peu moins élémentaire : le coefficient binomial $\binom{\cdot}{\cdot} : \mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$, défini par $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ si $0 \le k \le n$, et $\binom{n}{k} = 0$ sinon. Même pour de tels calculs très simples, il y a en général plusieurs méthodes possibles. Elles sont basées sur les mêmes opérations élémentaires, elles aboutissent toutes au résultat cherché, mais elles passent par des calculs intermédiaires bien différents.

Très souvent nous devons choisir la méthode la plus efficace parmi celles qui sont à notre disposition. Dans notre exemple, il y a au moins quatre méthodes différentes qui viennent à l'esprit :

Exercice/P 1.2. La définition $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ se traduit littéralement en une méthode de calcul. L'implémenter en deux fonctions `factorielle` et `binomial1`. Quel mode de passage des paramètres convient le mieux ? Combien d'opérations arithmétiques effectuent-elles, multiplications et divisions confondues, pour calculer $\binom{n}{k}$? Quel est le plus grand entier qui apparaisse dans les calculs intermédiaires ? Est-ce une méthode efficace pour calculer $\binom{1000000}{2}$?

Exercice/P 1.3. La fraction simplifiée $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$ suggère une deuxième méthode : on calcule d'abord le numérateur, puis on divise par le dénominateur. Expliquer brièvement en quoi cette méthode est avantageuse, puis l'implémenter en une fonction `binomial2`.

Exercice/P 1.4. La formule $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{1\cdots(k-1)k}$ peut être lue comme $\binom{n}{k} = \binom{n}{k-1}\frac{n-k+1}{k}$ avec condition initiale $\binom{n}{0} = 1$. Ceci donne lieu à une troisième méthode de calcul : une boucle où multiplications et divisions sont effectuées en alternance. Expliquer brièvement en quoi cette méthode pourrait être avantageuse, puis l'implémenter en une fonction `binomial3`.

Exercice/P 1.5. La propriété $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ donne lieu à une récurrence qui évite toute multiplication, en se basant sur les valeurs initiales $\binom{n}{0} = \binom{n}{n} = 1$. L'implémenter en une fonction récursive `binomial0`. Comme avant, veillez que $\binom{n}{k} = 0$ pour $k < 0$ ou $k > n$.

Exercice/P 1.6. Écrire un programme qui lit au clavier deux entiers `n` et `k`, puis affiche les résultats des fonctions `binomial3`, `binomial2`, `binomial1`, `binomial0` (dans cet ordre-ci). Tester soigneusement les fonctions pour quelques petites valeurs, puis pour des exemples plus grandes. Les résultats coïncident-ils comme il se doit ? Si oui, on veut alors comparer leur performance :

(1) Calculer $\binom{10}{5}$, $\binom{20}{10}$, $\binom{30}{15}$, ... Qu'observez-vous ? Comment expliquer le ralentissement de la fonction `binomial0` ? (Vous pouvez arrêter le programme avec `CTRL c`.)

(2) En excluant `binomial0`, calculer $\binom{10}{10}$, $\binom{100}{10}$, $\binom{1000}{10}$, $\binom{100000}{10}$, ... Qu'observez-vous ? Comment expliquer le ralentissement de la fonction `binomial1` ?

(3) En utilisant seulement `binomial3` et `binomial2`, calculer $\binom{2000}{1000}$, $\binom{4000}{2000}$, $\binom{8000}{4000}$, ... Au début il n'y a pas de différence significative, puis `binomial2` devient plus lente que `binomial3`. Comment expliquer cette différence ?

Justifiez ainsi la conclusion suivante : bien que les quatre fonctions calculent toutes le coefficient binomial cherché, leur temps d'exécution peut différer. Pour les petits nombres `n` et `k` c'est `binomial2` qui gagne, tandis que pour les grands nombres la fonction `binomial3` semble la plus efficace.

Bien évidemment on ne peut comparer que les méthodes que l'on connaît. Question naturelle : existe-t-il des méthodes encore meilleures ou des astuces d'optimisation ? Stratégie générale : plus on sait sur la fonction à calculer, plus de méthodes se présentent ! En voici un exemple :

Exercice/P 1.7. Vérifier d'abord que votre fonction `binomial3` calcule aisément $\binom{1000000}{10}$ mais elle bloque sur $\binom{1000000}{999990}$. Comment expliquer ce comportement ? En quoi la symétrie $\binom{n}{k} = \binom{n}{n-k}$ peut-elle être utilisée pour optimiser le calcul ? L'implémenter en une fonction `binomial4`. Vérifier qu'ainsi les calculs intermédiaires n'excèdent jamais la valeur $k\binom{n}{k}$.

2. Évaluation d'expressions algébriques

Jusqu'ici on ne peut entrer des entiers qu'en numération décimale. Or, entrer un grand entier comme $10^{100}+949$ par son écriture décimale de 101 chiffres n'est pas très commode. Il sera plus naturel justement d'utiliser une expression semblable à $10^{100}+949$.

> La lecture des données en entrée constitue souvent la partie la plus négligée d'un programme. Ce dernier devant communiquer avec une personne, il doit faire face aux fantaisies, aux conventions et aux erreurs apparemment aléatoires de celle-ci. Toute tentative pour forcer la personne à se comporter d'une façon plus adaptée à la machine est souvent considérée (avec raison) comme offensive. (Bjarne Stroustrup, *Le langage C++*)

Le but de ce paragraphe est de développer une fonction qui soit capable de lire et d'évaluer de telles expressions, et qui soit raisonnablement commode à utiliser. Ceci n'introduit aucun nouveau concept mathématique, il s'agit surtout d'un exercice de programmation.

2.1. Notations. Il y a plusieurs conventions possibles pour l'écriture d'une expression algébrique : on choisira ici le compromis d'une notation qui soit à la fois commode à utiliser et facile à programmer.

Notation infixe: On est habitué à l'écriture $10^{100}+949$ qui correspond à `(10^100)+949` où le symbole `^` représente l'opérateur de puissance. C'est ce que l'on appelle la notation *infixe* parce que les opérateurs binaires figurent entre leurs opérandes.

Notation préfixe: Pour les fonctions on utilise traditionnellement la notation *préfixe* comme $\phi(a)$ ou $f(x,y)$. Ici la fonction précède les paramètres.

Notation postfixe: Dans certains domaines mathématiques (comme la théorie des groupes) on préfère la notation *postfixe* comme a^{ϕ} ou $a\phi$. C'est aussi le cas pour la factorielle $n!$ où le paramètre précède la fonction.

Évidemment toutes ces notations sont équivalentes entre elles dans le sens que l'on peut traduire l'une à l'autre : au lieu d'écrire `(10^100)+949` en notation infixe, on peut écrire `+ ^ 10 100 949` en notation préfixe ou encore `10 100 ^ 949 +` en notation postfixe.

Question 2.1. Pourquoi la notation infixe nécessite-t-elle des parenthèses alors que les notations préfixe et postfixe peuvent s'en passer ? Expliquer comment transformer les différentes écritures linéaires en un arbre, et réciproquement comment transformer un arbre en chacune des écritures linéaires. On ne poursuivra pas cette approche ici, mais elle sera indispensable pour correctement stocker/analyser/évaluer des expressions d'une manière plus approfondie.

Dans la suite on développera une fonction `eval_postfix` qui évalue une expression en notation postfixe. On obtient ainsi une sorte de calculette, quoique modeste, qui permet de commodément calculer avec des grands entiers. On mettra les fonctions de calcul dans le fichier `integer.cc` et les fonctions d'entrée/sortie dans le fichier `eval-postfixe.cc`.

☞ Au-delà de son but à court terme, ce projet s'inscrit dans un développement « durable » tout le long ce semestre : nous commençons ici le travail sur le fichier `integer.cc` qui augmentera à fur et à mesure (si vous le maintenez comme souhaité, bien sûr). Chaque fois que vous programmez une fonction d'intérêt général pour la classe `Integer`, elle devrait être placée dans `integer.cc`. Idéalement vous inclurez ce fichier dans vos programmes ultérieurs, pour avoir toute la facilité d'une mini-bibliothèque (bien écrite et bien testée, si vous suivez les règles de l'art).

2.2. Évaluation en notation postfixe. Précisons d'abord ce que nous voulons faire. On souhaite disposer d'une fonction d'entrée avec (au moins) les possibilités suivantes :

- On peut entrer un entier en numération décimale.
  Exemple. — l'entrée `123` produit la valeur 123.
- On peut entrer toute une expression, délimitée de parenthèses ‘`(`’ et ‘`)`’.
  Exemple. — l'entrée `( 5 ! )` donne la valeur 120.
- On peut empiler plusieurs entiers, séparés d'espaces. La commande ‘`del`’ efface le dernier entier (il le dépile). Le passage à la ligne (la touche `return`) fait afficher la pile.
  Exemple. — l'entrée `( 123 456 789 del <return>` affiche `( 123 456` de sorte que l'on puisse contrôler le résultat intermédiaire et continuer l'entrée.
- On peut entrer le nom d'une fonction ou un opérateur arithmétique comme `+`, `-`, `*`, `/`, `%` en notation postfixe. Une telle fonction dépile ses arguments puis empile son résultat.
  Exemple. — l'entrée `( 123 456 + <return>` affiche `( 579 .`
- D'autres fonctions seront faciles à ajouter à fur et à mesure.
  Exemple. — l'entrée `( 100 20 binomial )` calculera le coefficient binomial $\binom{100}{20}$.

Exercice/P 2.2. Vous trouvez le début d'une implémentation dans le fichier `eval_postfixe.cc`. Lire le code source, le compiler puis le tester. Regarder en particulier l'usage de la pile.

Compléter l'implémentation en incluant les opérations arithmétiques : l'addition `+`, la soustraction `-`, la multiplication `*`, la division euclidienne `/`, le reste de la division `%`. En chaque instance on récupère les deux opérandes de la pile, puis on y remet le résultat du calcul. Veillez à l'ordre des opérandes ainsi qu'à d'éventuelles erreurs, par exemple la division par zéro.

Exercice/P 2.3. Écrire une fonction `binomial` dans `integer.cc`, issue de vos expériences en §1.3, et la rendre disponible dans `eval_postfixe.cc` via la commande `binomial`.

Exercice/P 2.4. Écrire une fonction

```
Integer puissance( Integer base, Integer exp )
```

dans `integer.cc`, et la rendre disponible dans `eval_postfixe.cc` via l'opérateur `^`. Ajouter

```
Integer puissance( Integer base, Integer exp, const Integer& mod )
```

qui calcule la puissance modulaire, c'est-à-dire le reste modulo `mod`. (Discuter le mode de passage des paramètres.) La rendre disponible via l'opérateur ternaire `^%`. *Remarque.* — Cette approche vous semblera peut-être redondante. Essayons donc de le justifier : En quoi la puissance modulaire peut-elle être plus efficace que la puissance ordinaire suivie d'une réduction modulaire ? Si possible, optimisez vos fonctions, puis tester leur performance. (On y reviendra dans le projet VIII.)

Exercice/P 2.5. Le programme `postfixe.cc` permet d'évaluer une expression passée par la ligne de commande. Après compilation avec `g++ postfixe.cc -lgmpxx -o postfixe` on peut ainsi écrire

```
$> postfixe "( 1 5 !  + )"
```

ce qui calcule $1+5! = 121$. (Il s'agit donc d'une calculette en miniature.) Ajouter d'autres fonctions qui vous semblent intéressantes : pgcd, ppcm, factorisation, test de primalité, etc.

"I only took the regular course." "What was that?" enquired Alice.
"Reeling and Writhing, of course, to begin with," the Mock Turtle replied:
"and then the different branches of Arithmetic —
Ambition, Distraction, Uglification, and Derision."
Lewis Carroll, *Alice's Adventures in Wonderland*

PROJET III

Calcul de la racine *n*ième

Objectifs

- Étudier deux algorithmes importants : la recherche dichotomique et la méthode de Newton
- Développer une preuve de correction pour un algorithme non trivial.

Ce projet vous propose d'implémenter deux méthodes afin de calculer la racine *n*ième entière, $\lfloor \sqrt[n]{a} \rfloor$, pour des nombres naturels a assez grands. Pour cela nous n'aurons besoin que des opérations élémentaires $+, -, *, /$ implémentées au préalable ; nous nous servirons ici de la bibliothèque GMP. (En principe notre implémentation du type `Naturel` marcherait aussi, mais elle serait plus lente.)

Pour un problème difficile on est déjà content de trouver *une* solution. S'il peut être résolu par plusieurs méthodes, on a tout intérêt à en choisir la plus efficace. Dans ce projet on regardera la situation suivante :

Lemme 0.1. *Soit* $f\colon \mathbb{N} \to \mathbb{N}$ *une application vérifiant* $0 = f(0) \le f(1) \le f(2) \le \dots$ *avec* $\sup f = +\infty$. *Alors pour tout* $y \in \mathbb{N}$ *il existe un unique* $x \in \mathbb{N}$ *de sorte que* $f(x) \le y < f(x+1)$. □

En appliquant ce lemme à $f(x) = x^n$, on voit que le nombre cherché est $x = \lfloor \sqrt[n]{y} \rfloor$. C'est bon à savoir qu'il existe et qu'il est unique, mais comment le trouver efficacement ?

Sommaire

1. Recherche dichotomique

Exercice 1.1. Commençons par la solution la plus évidente. Montrer que l'algorithme III.1 est correct : Pourquoi s'arrête-t-il ? Pourquoi renvoie-t-il la valeur cherchée ? Vérifier qu'il nécessite $x+1$ évaluations de la fonction f.

Algorithme III.1 Encadrement $f(x) \le y < f(x+1)$ par une recherche linéaire

Entrée: une fonction croissante $f\colon \mathbb{N} \to \mathbb{N}$ comme ci-dessus et un nombre naturel $y \in \mathbb{N}$
Sortie: l'unique $x \in \mathbb{N}$ tel que $f(x) \le y < f(x+1)$.

```
r ← 0                                  // On commence par r = 0 donc f(r) ≤ y.
tant que f(r+1) ≤ y faire r ← r+1      // On assure f(r) ≤ y après chaque itération.
retourner r                            // On sait finalement f(r) ≤ y < f(r+1), donc r = x.
```

Le coût linéaire de l'algorithme III.1 est prohibitif pour des exemples réalistes, où x peut être très grand. À l'instar de la recherche dichotomique discutée au chapitre V, l'algorithme suivant met en œuvre une variante astucieuse, qui améliore considérablement la performance :

Exercice 1.2. Prouver que l'algorithme III.2 est correct : montrer d'abord la terminaison, puis la correction en suivant les commentaires dans la description de la méthode.

Exercice 1.3. Vérifier pour $x = 0$ que l'algorithme III.2 n'effectue qu'une seule évaluation de f. Pour $x \ge 1$ déterminer le nombre exact d'évaluations de f effectuées dans la première puis la seconde boucle.
Indication. — Vous pouvez commencer par les cas $x = 1, \dots, 8$ et ainsi vérifier le tableau suivant.

Algorithme III.2 Encadrement $f(x) \leq y < f(x+1)$ par une recherche dichotomique

Entrée: une fonction croissante $f\colon \mathbb{N} \to \mathbb{N}$ comme ci-dessus et un nombre naturel $y \in \mathbb{N}$
Sortie: l'unique $x \in \mathbb{N}$ tel que $f(x) \leq y < f(x+1)$.

```
r ← 0,  s ← 1                          // On commence par r = 0 donc f(r) ≤ y.
tant que f(s) ≤ y faire r ← s,  s ← 2s  // On assure ainsi que f(r) ≤ y < f(s).
tant que s − r > 1 faire               // Tant que l'intervalle n'est pas réduit à un point …
  m ← ⌊(r+s)/2⌋                        // … on divise l'intervalle au milieu et …
  si f(m) ≤ y alors r ← m sinon s ← m   // … on assure à nouveau que f(r) ≤ y < f(s).
fin tant que                           // finalement s = r + 1 et f(r) ≤ y < f(s)
retourner r                            // On conclut que r = x.
```

Conclusion. — Pour des valeurs minuscules ($x \leq 5$) l'algorithme linéaire est au moins aussi efficace que son concurrent dichotomique, pour $x = 2$ et $x = 4$ il est même légèrement supérieur. Mais déjà à partir de $x = 6$ la recherche dichotomique s'amortit :

Évaluations de f	$x=0$	$x=1$	$x=2$	$x=3$	$x=4$	$x=5$	$x=6$	$x=7$	$x=8$
linéaire	1	2	3	4	5	6	7	8	9
dichotomique	1	2	4	4	6	6	6	6	8

Pour x grand la recherche dichotomique est nettement plus efficace que la recherche linéaire. Pour $x = 10^6$, par exemple, elle ne nécessite que 40 évaluations, alors que l'algorithme linéaire en nécessite un million. Pour $x = 10^9$ c'est 60 contre un milliard !

Exercice 1.4. On peut appliquer les méthodes précédentes à la fonction $f\colon \mathbb{N} \to \mathbb{N}$, $f(x) = xb$ et une valeur $a \in \mathbb{N}$, afin de trouver l'unique $q \in \mathbb{N}$ vérifiant $qb \leq a < (q+1)b$. On obtient ainsi le quotient q de la division euclidienne de a par b (avec reste $r = a - qb$). Détailler pourquoi la recherche linéaire revient à la méthode des soustractions itérées, alors que la recherche dichotomique correspond à la division scolaire en numération binaire. (Voir chapitre II, §1.7.)

2. La méthode de Newton-Héron

La recherche dichotomique s'applique à toute fonction croissante $f\colon \mathbb{N} \to \mathbb{N}$. Peut-on faire mieux dans le cas spécifique où la fonction est donnée par $f(x) = x^n$? Pour ceci on s'inspire du résultat suivant, que vous reconnaissez de votre cours d'analyse :

Théorème 2.1 (Calcul de la racine *n*ième d'après Newton-Héron)**.** *Soit $n \geq 2$ un entier et $a > 0$ un nombre réel. Pour toute valeur initiale $u_0 > 0$ la suite récurrente $u_{k+1} := \frac{1}{n}\left((n-1)u_k + a/u_k^{n-1}\right)$ converge vers la racine $r = \sqrt[n]{a}$. Pour $k \geq 1$ on a convergence monotone $u_k \searrow r$. Symétriquement pour $v_k = a/u_k^{n-1}$ on a $v_k \nearrow r$. On obtient ainsi des encadrements explicites $v_k \leq r \leq u_k$ de plus en plus fins :*

$$v_1 \leq v_2 \leq v_3 \leq \cdots \leq r \leq \cdots \leq u_3 \leq u_2 \leq u_1$$

Ajoutons que pour u_k proche de la racine r la convergence est exponentielle : à chaque itération le nombre de décimales valables double à peu près. C'est la convergence exponentielle qui fait de ce théorème un outil très puissant : si vous avez calculé un encadrement $[v_k, u_k]$ à $\approx 10^{-2}$ près, disons, l'itération suivante ne laissera qu'un écart $\approx 10^{-4}$, celle d'après $\approx 10^{-8}$, puis $\approx 10^{-16}$ etc.

Pour la racine *n*ième entière d'un nombre naturel il reste à mettre en œuvre un algorithme qui donne le résultat exact en n'utilisant que l'arithmétique des nombres entiers. Pour ceci on imite la récursion ci-dessus en remplaçant les nombres réels par les entiers :

Proposition 2.2. *Soient $y, x_0 \in \mathbb{Z}_+$ deux entiers positifs. On définie une suite récurrente $x_k \in \mathbb{Z}_+$ par*

$$x_{k+1} := \left\lfloor \left((n-1)x_k + \left\lfloor y/x_k^{n-1} \right\rfloor\right)/n \right\rfloor .$$

On a alors le comportement suivant :

- *Si $x_k^n \leq y$ alors $x_{k+1} \geq x_k$.*
- *Si $x_k^n > y$ alors $x_{k+1} < x_k$ mais toujours $(1 + x_{k+1})^n > y$.*

On conclut qu'une valeur initiale x_0 vérifiant $x_0^n \geq y$ donne une suite initialement décroissante, $x_0 > x_1 > \cdots > x_{k-1} > x_k \leq x_{k+1} \ldots$, et que $x_k = \left\lfloor \sqrt[n]{y} \right\rfloor$ est la valeur cherchée.

Exercice 2.3. Écrire un programme qui affiche la suite récurrente définie dans la proposition, afin de vérifier empiriquement ces affirmations, et pour motiver l'algorithme III.3 ci-dessous. Montrer la correction de cet algorithme (en admettant la proposition) : établir d'abord la terminaison, puis la correction du résultat en suivant les commentaires dans la description de la méthode.

Algorithme III.3 Racine *n*ième entière d'après Newton-Héron

Entrée: deux entiers $y \geq 1$ et $n \geq 2$

Sortie: l'unique entier $r \geq 1$ vérifiant $r^n \leq y < (r+1)^n$

choisir une valeur initiale x tel que $x^n \geq y$ // voir la remarque 3.1 plus bas
répéter
  $r \leftarrow x, \quad x \leftarrow \left\lfloor \frac{1}{n}\left((n-1)x + \left\lfloor \frac{y}{x^{n-1}} \right\rfloor\right) \right\rfloor$ // variante entière de la récursion de Newton-Héron
jusqu'à $x \geq r$ // condition d'arrêt motivée ci-dessus
retourner r

Exercice/M 2.4. Si vous êtes courageux, vous pouvez essayez de prouver la proposition.
Indication. — Dans la version réelle, on itère la fonction $f\colon \mathbb{R}_+ \to \mathbb{R}_+$ donnée par $f(x) = \frac{1}{n}\left((n-1)x + yx^{1-n}\right)$. Elle est strictement croissante sur $[\sqrt[n]{y}, +\infty[$, elle vérifie $\sqrt[n]{y} < f(x) < x$ pour tout $x > \sqrt[n]{y}$, ainsi que $f(\sqrt[n]{y}) = \sqrt[n]{y}$, ceci est donc le seul point fixe et il est attractif. (Le détailler.) Dans la version entière nous itérons $g\colon \mathbb{Z}_+ \to \mathbb{Z}_+$ définie par $g(x) = \left\lfloor \left((n-1)x + \left\lfloor y/x^{n-1} \right\rfloor\right)/n \right\rfloor$. Vérifier que $g(x) = \lfloor f(x) \rfloor$ pour tout $x \in \mathbb{Z}_+$.

Remarque 2.5 (complexité)**.** On remarque que la décroissance dans les entiers est légèrement plus rapide que dans la version réelle, grâce aux arrondis. Ceci permet d'étendre le résultat sur l'excellente convergence de la méthode de Newton à notre calcul dans les entiers.

3. Implémentation et tests empiriques

Remarque 3.1 (approximation grossière)**.** Au début de l'algorithme III.3 il faut choisir une valeur initiale x tel que $x^n \geq y$. Bien sûr on pourrait prendre $x = y$, mais pour accélérer il vaut mieux choisir une valeur proche de la racine $\sqrt[n]{y}$ mais bien entendu plus grande que celle-ci. Il y a plusieurs méthodes de le faire de manière efficace. En s'inspirant de la recherche dichotomique on pourrait écrire :

```
Integer x=1; while ( puissance(x,n) < y ) x*=2;
```

Il existe une variante plus rapide, qui profite du fait que l'entier y soit stocké en base 2. Ainsi il est facile de déterminer sa longueur $\ell = 1 + \lfloor \log_2 |y| \rfloor$, qui n'est rien d'autre que le nombre de chiffres binaires :

```
int len( const Integer& y ) { return mpz_sizeinbase(y.get_mpz_t(),2); };
int log2( const Integer& y ) { return len(y) - 1; };
```

Ensuite on calcule aisément $x = 2^{1+\lfloor \lfloor \log_2 y \rfloor / n \rfloor}$; en système binaire il ne s'agit que d'un décalage :

```
Integer x= Integer(2) << ( log2(y)/n );              // décalage bit par bit
```

Vérifier que $x \approx \sqrt[n]{y}$ tout en assurant $x^n > y$, comme exigé dans l'algorithme. Vérifier que cette astuce s'applique également à la recherche dichotomique, où il faut trouver r, s tels que $r \leq \sqrt[n]{y} < s$:

```
Integer r= Integer(1) << ( log2(y)/n ), s= r << 1;   // décalages bit par bit
```

Exercice/P 3.2. Implémenter la recherche dichotomique (algorithme III.2) et la méthode de Newton-Héron (algorithme III.3) afin de calculer $\sqrt[n]{y}$. Ici y sera de type `Integer`, tandis que pour n le type `int` suffira. (Pourquoi ?) Tester les deux fonctions sur des exemples de plus en plus grands. (Il sera intéressant de faire afficher les étapes intermédiaires du calcul.) Les résultats sont-ils identiques, comme il se doit ?

Exercice/P 3.3. Comment déterminer quelle méthode est plus efficace ? Comme la comparaison des coûts exacts s'avère difficile, on fait recours aux tests empiriques :

– Calculer $\left\lfloor \sqrt[3]{n!} \right\rfloor$ pour $n = 1, \ldots, 1000$.
– Calculer $\left\lfloor \sqrt[n]{n!} \right\rfloor$ pour $n = 1, \ldots, 1000$.

Vous pouvez mesurer le temps d'exécution avec le programme `racine.cc`. Formulez vos observations, puis essayez d'en tirer une conclusion ou une règle heuristique pour choisir la meilleure méthode.

4. Critères de qualité d'un logiciel

En guise de conclusion, et afin de clarifier les termes, explicitons les qualités que l'on souhaiterait de tout logiciel. Plus l'application envisagée est importante, plus les critères suivants deviennent cruciaux. Pensez par exemple à un logiciel de pilotage d'un avion ou la conduite d'un métro sans conducteur.

Fiabilité: Un programme est *fiable* s'il fait toujours exactement ce pour quoi il est fait, en parfaite conformité avec ses spécifications. Cette qualité est indispensable. Pour une application importante on n'acceptera pas un logiciel qui marche 9 fois sur 10.

Clarté: Dans le souci de fiabilité, on doit insister sur un code source *clair et net*. Un programme embrouillé ou incompréhensible est inutilisable : d'abord on n'arrive pas à se convaincre de sa fiabilité, puis il sera difficile, voire impossible, de le maintenir ou modifier. Que vaut un programme qui semble marcher mais on ne comprend pas pourquoi ?

Efficacité: Un programme *efficace* s'exécute rapidement et utilise économiquement la mémoire et toute autre ressource de l'ordinateur. Évidemment, cette qualité est assez importante dans la pratique : que vaut une réponse correcte si elle arrive trop tard ?

☞ L'efficacité ne doit être recherchée qu'après satisfaction de deux exigences précédentes, qui sont primordiales : que vaut une réponse rapide si elle est fausse ?

Robustesse: Un programme fiable travaille correctement dans les situations prévues par sa spécification. Il est *robuste* si de plus il réagit de manière raisonnable dans des situations imprévues. Par exemple, si l'utilisateur entre des données hors domaine, le logiciel, au lieu de dérailler, les refuse poliment et propose une manière de rectifier la situation. Après la fiabilité et l'efficacité, la robustesse sera très appréciée par l'utilisateur.

Réutilisabilité: Comme la programmation soigneuse nécessite un investissement important, il convient de *réutiliser*, si possible, le code source déjà développé. Bien sûr on ne veut réutiliser que de code qui soit fiable, clair, et efficace. Si ces qualités sont satisfaites, tout programmeur appréciera la possibilité de s'en servir. (Reconnaissons à ce propos que nous avons déjà profité des superbes bibliothèques STL et GMP.) Pour un développement durable, il est donc important de prévoir les réutilisations possibles dans d'autres contextes.

Documentation: Pour qu'un logiciel soit utile dans un contexte plus large, l'utilisateur souhaitera une documentation indépendante de l'implémentation. Elle s'adresse à l'utilisateur envisagé, que ce soit un programmeur qui réutilise certaines composantes, ou un usager qui se sert du logiciel comme application toute faite. Dans la pratique une utilisation « intuitive » et une documentation de qualité sont souvent cruciales.

Il va sans dire qu'on devrait appliquer ces critères à tout logiciel, non seulement dans ce cours. Avouons cependant que ces exigences draconiennes sont rarement satisfaites, soit par manque de temps, soit tout simplement par négligence.

☞ En tenant compte des critères ci-dessus, essayez de réexaminer et d'optimiser votre implémentation. Est-elle fiable ? (Le vérifier sur beaucoup d'exemples triviaux et non-triviaux. Enfin, peut-on *prouver* sa correction ?) Le code source est-il clair et bien commenté ? (Essayez de lire et vérifier la solution de quelqu'un d'autre.) Le logiciel s'exécute-t-il rapidement ? (Majorer si possible la complexité théorique, puis effectuez de nombreux tests empiriques.) Est-il robuste dans des circonstances imprévues ? (Soyez méchant et essayez de provoquer une erreur grave, puis invitez quelqu'un d'autre à le faire.) Réagit-il toujours de manière aimable envers l'utilisateur ? (Le tester sur un non-spécialiste.) L'usage dans d'autres programmes sera-t-il facile ? (À revoir dans des applications ultérieures, plus tard pendant ce semestre.)

Algorithms + Data Structures = Programs
Niklaus Wirth

CHAPITRE IV

Numération positionnelle et conversion de base

Objectifs

- Réviser la numération en base $b \geq 2$ et l'algorithme de changement de base.
- Implémenter une application surprenante : le calcul de e et de π à une précision arbitraire.
- Préparer le terrain pour le calcul arrondi (chap. XV) et le calcul arrondi fiable (chap. XVI)

Ce chapitre explique d'abord comment convertir la représentation d'un nombre de la base 10 à la base 2, puis comment convertir entre deux bases quelconques. L'idée est simple, mais les applications sont étonnantes : avec très peu d'analyse, cette méthode permet de calculer quelques milliers de décimales de $e = 2,71828\ldots$ Le projet traitera ensuite le calcul de $\pi = 3,14159\ldots$

Sommaire

1. Numération positionnelle

1.1. Un premier exemple : la conversion de la base 10 à la base 2. Rappelons d'abord la division euclidienne des entiers : pour tout $a, b \in \mathbb{Z}$ avec $b \neq 0$ il existe une unique paire $(q, r) \in \mathbb{Z} \times \mathbb{Z}$ telle que $a = bq + r$ et $0 \leq r < |b|$. Dans la suite on utilisera la notation $a \operatorname{div} b := q$ pour le quotient, et $a \bmod b := r$ pour le reste d'une telle division euclidienne.

Exemple 1.1. Comment trouver la représentation binaire du nombre $11,6875_{\text{dec}}$? C'est facile pour la partie entière $11_{\text{dec}} = \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0 = \texttt{1011}_{\text{bin}}$. Ce développement s'obtient par une division euclidienne itérée selon le schéma suivant :

$$\begin{aligned} 11 \bmod 2 &= \mathbf{1} \quad \text{et} \quad 11 \operatorname{div} 2 = 5 \\ 5 \bmod 2 &= \mathbf{1} \quad \text{et} \quad 5 \operatorname{div} 2 = 2 \\ 2 \bmod 2 &= \mathbf{0} \quad \text{et} \quad 2 \operatorname{div} 2 = 1 \\ 1 \bmod 2 &= \mathbf{1} \quad \text{et} \quad 1 \operatorname{div} 2 = 0 \end{aligned}$$

Pareil pour la partie fractionnaire $0,6875_{\text{dec}} = \mathbf{1} \cdot 2^{-1} + \mathbf{0} \cdot 2^{-2} + \mathbf{1} \cdot 2^{-3} + \mathbf{1} \cdot 2^{-4} = \texttt{0.1011}_{\text{bin}}$. Ici on multiplie par 2 au lieu de diviser, on extrait la partie entière et continue avec la partie fractionnaire :

$$\begin{aligned} 0,6875 \cdot 2 &= \mathbf{1},3750 \\ 0,3750 \cdot 2 &= \mathbf{0},7500 \\ 0,7500 \cdot 2 &= \mathbf{1},5000 \\ 0,5000 \cdot 2 &= \mathbf{1},0000 \end{aligned}$$

Exercice/M 1.2. Vérifier la conversion de $31,9_{\text{dec}}$ en binaire donnée en chapitre I, page 12. Cette fois-ci on obtient une représentation binaire qui est périodique. (Pourquoi ?)

1.2. Représentation en base b. Les bases 10 et 2 n'ont rien de particulier, à part leur usage fréquent, et nous regarderons dans la suite une base b quelconque, avec $b \in \mathbb{N}$, $b \geq 2$. Explicitons d'abord les deux algorithmes sous-jacents aux exemples précédents.

Algorithme IV.1 Représentation d'un nombre naturel $a \geq 0$ en base b

Entrée: Un nombre naturel $x \in \mathbb{N}$ et un entier $b \geq 2$

Sortie: L'unique suite $(x_{-n}, \dots, x_0)$ dans $[\![0, b[\![$ telle que $x = \sum_{k=-n}^{0} x_k b^{-k}$ et $x_{-n} \neq 0$

Initialiser $n \leftarrow -1$
tant que $x > 0$ **faire** $n \leftarrow n+1$, $x_{-n} \leftarrow x \bmod b$, $x \leftarrow x \operatorname{div} b$
retourner $(x_{-n}, \dots, x_0)$

Exercice/M 1.3. Vérifier que l'algorithme IV.1 est correct. Plus explicitement : étant donnés $x \in \mathbb{N}$ et $b \geq 2$, pourquoi produit-il une suite $(x_{-n}, \dots, x_0)$ dans $[\![0, b[\![$ vérifiant $x = \sum_{k=-n}^{0} x_k b^{-k}$? Pourquoi cette suite est-elle unique ? Ceci justifie d'appeler x_k le $k$ième chiffre de x dans le développement en base b.

Algorithme IV.2 Représentation d'un nombre réel $r \geq 0$ en base b

Entrée: Un nombre réel $x \geq 0$ et deux entiers $b \geq 2$, $p \geq 0$

Sortie: L'unique suite $(x_{-n}, \dots, x_0, \dots, x_p)$ dans $[\![0, b[\![$ avec $x_{-n} \neq 0$ vérifiant $x = \sum_{k=-n}^{p} x_k b^{-k} + \varepsilon_p$ avec un reste $0 \leq \varepsilon_p < b^{-p}$

Trouver d'abord la représentation $(x_{-n}, \dots, x_0)$ de la partie entière $\lfloor x \rfloor$.
pour k **de** 1 **à** p **faire** $x \leftarrow x - \lfloor x \rfloor$, $x \leftarrow bx$, $x_k \leftarrow \lfloor x \rfloor$
retourner $(x_{-n}, \dots, x_0, \dots, x_p)$

Exercice/M 1.4. Vérifier que l'algorithme IV.2 est correct : étant donnés $x \geq 0$ et $b \geq 2$ et $p \geq 0$, pourquoi produit-il une suite $(x_{-n}, \dots, x_0, \dots, x_p)$ dans $[\![0, b[\![$? Pourquoi a-t-on $x = \sum_{k=-n}^{p} x_k b^{-k} + \varepsilon_p$ avec un reste $0 \leq \varepsilon_p < b^{-p}$? Pourquoi cette suite est-elle unique ?

Exercice/M 1.5. Montrer que l'algorithme IV.2 est *stable* dans le sens suivant : augmenter la précision de p à $p+1$ *ajoute* un chiffre sans changer les chiffres précédents. On peut ainsi, pour $p \to \infty$, obtenir un *développement infini* en base b. Définir ce que c'est et expliquer pourquoi il n'y a plus unicité. Quels nombres admettent plus d'un développement infini ? L'algorithme ci-dessus n'en produit qu'un, lequel ?

Remarque 1.6. Dans l'algorithme IV.2 nous ignorons comment est donné le nombre réel x, c'est-à-dire comment il est représenté concrètement sur machine. La seule chose qu'il faut savoir est d'effectuer deux opérations élémentaires : multiplication par b, puis séparation des parties entière et fractionnaire. Bien qu'élémentaires, ces opérations ne sont pas toujours faciles : pour vous en convaincre, essayez d'appliquer l'algorithme IV.2 à $\sqrt{7}$ ou à $\ln 2$ ou à $\int_0^1 e^{-x^2/2} dx$ ou tout autre nombre réel qui vous vient à l'esprit.

Par conséquent, dans toutes les applications suivantes, nous supposons que x est lui-même donné dans une numération positionnelle, c'est-à-dire par une somme $x = \sum x_k v_k$ où les x_k sont les « chiffres », pondérés par des valeurs positionnelles v_k, choisies convenablement selon le contexte. Ce cadre est suffisamment général pour être intéressant, et en même temps il se prête bien au calcul.

1.3. Conversion de base. Ce paragraphe explique comment convertir un développement en base b en une autre base c. La partie entière se traite comme ci-dessus et ne pose aucun problème. Nous nous concentrerons donc sur la partie fractionnaire uniquement. Autrement dit, nous allons implémenter des calculs avec des *nombres à virgule fixe*, c'est-à-dire avec des développements de la forme

$$x_0, x_1 x_2 x_3 \dots x_m \quad \text{en base } b.$$

Une telle représentation n'est rien d'autre que la donnée d'une suite de chiffres $x_0, x_1, \dots, x_m$. Afin d'avoir une notation commode nous introduisons l'application $\langle \cdot \rangle_b : \mathbb{Z}^{m+1} \to \mathbb{Q}$ qui associe à chaque suite finie $x = (x_0, x_1, \dots, x_m)$ le nombre rationnel $\langle x \rangle_b := \sum_{k=0}^{m} x_k b^{-k}$ ainsi représenté. C'est une application $\mathbb{Z}$-linéaire, dont l'image consiste de tous les nombres rationnels de la forme z/b^{-m} avec $z \in \mathbb{Z}$.

Définition 1.7. On dit que $q \in \mathbb{Q}$ est *représenté* par $x \in \mathbb{Z}^{m+1}$ en base b si $\langle x \rangle_b = q$. Une telle représentation est appelée *normale* (par rapport à b) si $0 \leq x_k < b$ pour tout $k = 1, 2, \ldots, m$.

Pour simplifier nous n'exigeons pas que $0 \leq x_0 < b$; l'entier x_0 joue le rôle de la partie entière et peut prendre n'importe quelle valeur dans $\mathbb{Z}$. Ce sont les chiffres après la virgule qui nous intéressent ici.

☞ *Dans tout ce chapitre nous allons utiliser ces développements dits* à virgule fixe. ✍

Notons d'abord que la normalisation est toujours possible : si par exemple x_m ne vérifie pas $0 \leq x_m < b$, on effectue une division euclidienne $x_m = qb + x'_m$ avec quotient $q = x_m \operatorname{div} b$ et un reste $x'_m = x_m \bmod b$ vérifiant $0 \leq x'_m < b$. On remplace alors x_m par x'_m et ajoute la retenue q à x_{m-1}. En itérant ce procédé pour $k = m-1, \ldots, 1$ on obtient le résultat suivant :

Proposition 1.8 (normalisation)**.** *Toute représentation $x \in \mathbb{Z}^{m+1}$ peut être normalisée par rapport à la base b, c'est-à-dire qu'il existe une représentation normale $\bar{x} \in \mathbb{Z}^{m+1}$ telle que $\langle \bar{x} \rangle_b = \langle x \rangle_b$.*

L'algorithme IV.3 ci-dessous construit une telle représentation normale à partir de x.

Algorithme IV.3 Normalisation par rapport à une base b

Entrée: Un vecteur $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\bar{x} = \langle x \rangle_b$ en base $b \geq 2$

Sortie: Le vecteur normal $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\bar{x}$ en base $b \geq 2$

```
pour k de m à 1 faire                  // l'indice k parcourt de droite à gauche
    x_{k-1} ← x_{k-1} + (x_k div b)    // ajouter la retenue à x_{k-1}
    x_k ← x_k mod b                    // réduire x_k modulo b
fin pour
retourner (x_0, x_1, ..., x_m)
```

Nous allons prouver la proposition 1.8 en montrant que l'algorithme est correct :

Lemme 1.9 (correction)**.** *L'algorithme IV.3 est correct. Plus explicitement, la valeur représentée $\langle x \rangle_b$ ne change pas durant l'exécution de l'algorithme, et le vecteur renvoyé x est normalisé.*

DÉMONSTRATION. Évidemment l'algorithme se termine (toujours après m itérations). Vérifions d'abord l'invariance. Par définition on a $\langle x \rangle_b = \sum_{k=0}^{m} x_k b^{-k}$. La division euclidienne donne $x_k = b \cdot q + x'_k$ tel que $0 \leq x'_k < b$ et $q \geq 0$. Si l'on pose $x'_{k-1} = x_{k-1} + q$ on voit aisément que $\langle x \rangle_b = \langle x' \rangle_b$. (Le détailler.)

Montrons par récurrence que le résultat est normalisé. Supposons qu'avant l'itération k les chiffres $x_{k+1}, \ldots, x_m$ sont normalisé, c'est-à-dire que $0 \leq x_j < b$ pour tout $j = k+1, \ldots, m$. Ces chiffres-là ne changent pas lors de l'itération k, et après le chiffre x_k vérifie $0 \leq x_k < b$ par construction. □

C'est toujours une excellente idée de majorer les calculs intermédiaires : explosent-ils ou restent-ils bornés ? Plus concrètement, on peut se demander si les petits entiers (de type `int`) suffiront pour implémenter l'algorithme. Le lemme suivant en donne la réponse :

Lemme 1.10 (majoration)**.** *Dans l'algorithme IV.3, si tous les x_k satisfont initialement $0 \leq x_k < cb$, avec une constante $c \in \mathbb{N}$, alors la retenue ne dépasse jamais $2c$.*

DÉMONSTRATION. Pour $k > m$ la retenue est nulle, il n'y donc rien à montrer. Supposons que la retenue ajoutée à x_k a été inférieure à $2c$. Alors $0 \leq x_k < c(b+2)$. La division euclidienne $x_k = bq + x'_k$ donne donc $0 \leq x'_k < b$ et $0 \leq q < 2c$ car $b \geq 2$. On conclut par récurrence □

Le lemme suivant confirme que tronquer une représentation en base b donne la partie entière.

Lemme 1.11 (unicité)**.** *Si $x \in \mathbb{Z}^{m+1}$ est normale par rapport à la base b, alors $x_0 \leq \langle x \rangle_b < x_0 + 1$. Si $x, y \in \mathbb{Z}^{m+1}$ sont normales par rapport à la base b, alors $\langle x \rangle_b = \langle y \rangle_b$ entraîne $x = y$.*

DÉMONSTRATION. Évidemment $\langle x \rangle_b = \sum_{k=0}^{m} x_k b^{-k} \geq x_0$ car tous les termes de la somme $\sum_{k=1}^{m} x_k b^{-k}$ sont positifs ou nuls. D'autre part $\sum_{k=1}^{m} x_k b^{-k} \leq \sum_{k=1}^{m} (b-1) b^{-k} = 1 - b^{-m} < 1$.

Supposons maintenant que $x, y \in \mathbb{Z}^{m+1}$ sont normales et que $\langle x \rangle_b = \langle y \rangle_b$. Tout d'abord on constate que $\lfloor \langle x \rangle_b \rfloor = x_0$ et $\lfloor \langle y \rangle_b \rfloor = y_0$, ce qui implique $x_0 = y_0$. Ensuite $\lfloor b \langle x \rangle_b \rfloor = b x_0 + x_1$ et $\lfloor b \langle y \rangle_b \rfloor = b y_0 + y_1$, ce qui implique $x_1 = y_1$. On conclut ainsi par récurrence. □

L'argument de la preuve précédente n'est rien d'autre que le développement en base b vu dans l'algorithme IV.2. Après cette préparation, l'algorithme IV.4 formalise la méthode de l'exemple 1.1 :

Algorithme IV.4 Changement de la base b à la base c à une précision donnée n

Entrée: Un vecteur $x = (x_0, x_1, \dots, x_m) \in \mathbb{Z}^{m+1}$ et trois entiers $b \geq 2$, $c \geq 2$, $n \geq 0$
Sortie: Un vecteur $y = (y_0, y_1, \dots, y_n) \in \mathbb{Z}^{n+1}$, normal par rapport à la base c
Garanties: La valeur $\langle y \rangle_c$ est une approximation optimale dans le sens que $\langle y \rangle_c \leq \langle x \rangle_b < \langle y \rangle_c + c^{-n}$

pour j **de** 0 **à** n **faire**
  normaliser x par rapport à la base b // ceci fait appel à l'algorithme précédent
  $y_j \leftarrow x_0, \quad x_0 \leftarrow 0$ // transférer la partie entière x_0 dans y_j
  $x \leftarrow c \cdot x$ // multiplier chaque chiffre de x par c
fin pour
retourner $(y_0, y_1, \dots, y_n)$

Proposition 1.12 (correction)**.** *L'algorithme IV.4 est correct, c'est-à-dire qu'il convertit la représentation $x \in \mathbb{Z}^{m+1}$ en base b en une représentation normale $y \in \mathbb{Z}^{n+1}$ en base c telle que $\langle y \rangle_c \leq \langle x \rangle_b < \langle y \rangle_c + c^{-n}$.*

DÉMONSTRATION. Évidemment l'algorithme se termine (toujours après m itérations).

Vérifions d'abord que les chiffres $y_1, \dots, y_n$ ainsi produits sont normalisés dans le sens que $0 \leq y_j < c$. Ceci n'est pas forcément vrai pour y_0 qui reçoit la partie entière initiale. Mais pour $j = 1, \dots, n$ nous savons que $0 \leq \langle x \rangle_b < c$. La normalisation de x assure que $x_0 \leq \langle x \rangle_b < x_0 + 1$ d'après le lemme 1.11.

Notons $x^{(j)}$ et $y^{(j)}$ les représentations au début de la $j$ème itération. On vérifie aisément par récurrence que l'algorithme préserve l'égalité $\langle x \rangle_b = \langle y^{(j)} \rangle_c + c^{-j} \langle x^{(j)} \rangle_b$. (C'est le point clé. Le détailler.) On en déduit l'encadrement $\langle y^{(j)} \rangle_c \leq \langle x \rangle_b < \langle y^{(j)} \rangle_c + c^{-j}$, ce qui prouve la correction de l'algorithme. □

Remarque 1.13 (approximation)**.** En général on ne peut pas espérer tomber exactement sur $\langle y \rangle_c = \langle x \rangle_b$, même avec une précision n arbitrairement grande. Pour le voir convertir par exemple $\frac{1}{3} = \langle 0,1 \rangle_3$ vers $\langle 0,3,3,3,\dots \rangle_{10}$ en base 10, ou bien $\langle 0,1 \rangle_{10}$ en base 2. Mais en choisissant la précision n suffisamment grande, l'encadrement $\langle y \rangle_c \leq \langle x \rangle_b < \langle y \rangle_c + c^{-n}$ garantit une approximation aussi fine que souhaitée.

Remarque 1.14 (complexité)**.** L'algorithme IV.4 est de complexité *quadratique* : il nécessite mn itérations pour convertir la représentation initiale x de longueur m en la représentation finale y de longueur n.

Exercice/P 1.15. Implémenter les algorithmes IV.3 et IV.4 en deux fonctions

```
void normaliser( vector<int>& chiffres, int base );
void convertir( vector<int> chiffres1, int base1,
                vector<int>& chiffres2, int base2, int precision );
```

Justifier le mode de passage des paramètres. L'utilisation du type `int` risque-t-elle de provoquer des erreurs de dépassement de capacité ? Tester votre fonction avec un programme qui lit au clavier une représentation en base b de longueur m et la convertit en base c avec précision n. (Pour l'entrée-sortie des vecteurs vous pouvez vous servir du fichier `vectorio.cc`.)

☞ Les bases fréquemment utilisées sont 2, 8, 10, 16, et on aura rarement besoin d'une base $b > 16$. Il semble donc une bonne idée d'utiliser les chiffres `0...9` puis `A...F` pour représenter les entiers $0, \dots, 15$. Adapter votre programme pour qu'il utilise cette notation.

2. Représentation factorielle

La représentation en base b est pratique et omniprésente, mais il y a d'autres représentations intéressantes qui sont parfois mieux adaptées. Nous regarderons dans la suite le *développement factoriel*. En imitant l'approche du §1.3, nous introduisons l'application $\langle \cdot \rangle_! : \mathbb{Z}^{m+1} \to \mathbb{Q}$ qui associe à chaque suite finie $x = (x_0, x_1, \dots, x_m)$ le nombre rationnel $\langle x \rangle_! := \sum_{k=0}^{m} \frac{x_k}{(k+1)!}$. Par exemple $\langle 2,1,1,1 \rangle_! = 2 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!}$ est le début de la série $\sum_{k=0}^{\infty} \frac{1}{k!}$ qui converge vers $e = 2,71828\dots$.

Remarque 2.1 (base mixte)**.** La représentation factorielle utilise ce que l'on appelle une *base mixte*. De tels systèmes sont très courants, par exemple pour parler d'une durée de temps : ainsi la durée de 2 semaines, 3 jours, 10 heures, 55 minutes et 17 secondes équivaut à $\mathbf{2} + \frac{\mathbf{3}}{7} + \frac{\mathbf{10}}{7 \cdot 24} + \frac{\mathbf{55}}{7 \cdot 24 \cdot 60} + \frac{\mathbf{17}}{7 \cdot 24 \cdot 60 \cdot 60}$ semaines.

Exercice/M 2.2. L'application $\langle\cdot\rangle_! : \mathbb{Z}^{m+1} \to \mathbb{Q}$ est $\mathbb{Z}$-linéaire et son image consiste de tous les nombres rationnels $\frac{z}{(m+1)!}$ avec $z \in \mathbb{Z}$. En particulier, tout nombre rationnel $q \in \mathbb{Q}$ admet une telle représentation avec m convenable. Expliquer pourquoi le développement en base b mène aux représentations périodiques pour certains nombres rationnels, alors que dans le système factoriel tout tel développement se terminera. Écrire deux algorithmes analogues aux algorithmes IV.1 et IV.2 qui développent un nombre réel $r \geq 0$ en base factorielle.

Définition 2.3. On dit que $x \in \mathbb{Z}^{m+1}$ est *normale* par rapport à la base factorielle si $0 \leq x_k \leq i$ pour tout $k = 1, \ldots, m$. (Comme avant nous ne posons aucune restriction sur la valeur x_0.)

Heureusement, comme en base b, la normalisation en base factorielle est toujours possible :

Lemme 2.4 (normalisation)**.** *Toute représentation $x \in \mathbb{Z}^{m+1}$ peut être normalisée par rapport à la base factorielle, c'est-à-dire qu'il existe une représentation normale $\overline{x} \in \mathbb{Z}^{m+1}$ telle que $\langle\overline{x}\rangle_! = \langle x\rangle_!$. L'algorithme IV.5 ci-dessous calcule une telle représentation normale à partir de x.*

Algorithme IV.5 Normalisation par rapport à la base factorielle

Entrée: Un vecteur $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\overline{x} = \langle x\rangle_!$ en base factorielle
Sortie: Le vecteur normal $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\overline{x}$ en base factorielle

pour k **de** m **à** 1 **faire** // l'indice k parcourt de droite à gauche
  $x_{k-1} \leftarrow x_{k-1} + x_k \operatorname{div} (k+1)$ // ajouter la retenue à x_{k-1}
  $x_k \leftarrow x_k \bmod (k+1)$ // réduire x_k modulo $k+1$
fin pour
retourner $(x_0, x_1, \ldots, x_m)$

Exercice/M 2.5. Prouver l'algorithme IV.5 : pourquoi la valeur $\langle x\rangle_!$ ne change-t-elle pas ?

Le lemme suivant affirme que tronquer une représentation factorielle donne la partie entière, et on en déduit que la représentation normale en base factorielle est unique :

Lemme 2.6 (unicité)**.** *Si $x \in \mathbb{Z}^{m+1}$ est normale par rapport à la base factorielle, alors $x_0 \leq \langle x\rangle_! < x_0 + 1$. Si $x, y \in \mathbb{Z}^{m+1}$ sont normales par rapport à la base factorielle, alors $\langle x\rangle_! = \langle y\rangle_!$ entraîne $x = y$.*

Exercice/M 2.7. Montrer le lemme précédent en prouvant que $\sum_{k=1}^{m} \frac{k}{(k+1)!} = 1 - \frac{1}{(m+1)!}$. Ensuite, pour l'unicité, expliciter comment la valeur $\langle x\rangle_!$ détermine x_0, puis les chiffres $x_1, x_2, x_3, \ldots$.

Reste à établir la conversion entre développement factoriel et développement en base b :

Proposition 2.8 (conversion)**.** *L'algorithme IV.6 convertit la représentation factorielle $x \in \mathbb{Z}^{m+1}$ en une représentation $y \in \mathbb{Z}^{n+1}$ en base b telle que $\langle y\rangle_b \leq \langle x\rangle_! < \langle y\rangle_b + b^{-n}$.*

Algorithme IV.6 Changement de base factorielle en base b avec précision n

Entrée: Un vecteur $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ et deux entiers $b \geq 2$, $n \geq 0$
Sortie: Un vecteur $y = (y_0, y_1, \ldots, y_n) \in \mathbb{Z}^{n+1}$, normal par rapport à la base b
Garanties: La valeur $\langle y\rangle_b$ est une approximation optimale dans le sens que $\langle y\rangle_b \leq \langle x\rangle_! < \langle y\rangle_b + b^{-n}$

pour j **de** 0 **à** n **faire**
  normaliser x par rapport à la base factorielle // ceci fait appel à l'algorithme précédent
  $y_j \leftarrow x_0$, $x_0 \leftarrow 0$ // transférer la partie entière x_0 dans y_j
  $x \leftarrow b \cdot x$ // multiplier chaque chiffre de x par b
fin pour
retourner $(y_0, y_1, \ldots, y_n)$

Exercice/M 2.9. Prouver l'algorithme IV.6 : expliquer pourquoi les chiffres $y_1, \ldots, y_n$ sont normalisés dans le sens que $0 \leq y_k < b$, puis prouver l'inégalité $\langle y\rangle_b \leq \langle x\rangle_! < \langle y\rangle_b + b^{-n}$. (On pourra suivre la preuve de la proposition 1.12) Pourquoi ne peut-on en général pas espérer tomber sur l'égalité $\langle y\rangle_b = \langle x\rangle_!$ même avec une précision n arbitrairement grande ?

2.1. Calcul de e à 10000 décimales. Comme application on se propose de calculer $e = \sum_{i=0}^{\infty} \frac{1}{i!}$ en base b à une précision p donnée près. Plus explicitement, on cherche une suite de chiffres $(t_0, t_1, \ldots, t_p)$ telle que le nombre rationnel $t = \sum_{k=0}^{p} t_k b^{-k}$ ainsi représenté vérifie $t < e < t + b^{-p}$.

Exercice/M 2.10. Expliquer pourquoi cette spécification définit le vecteur $(t_0, t_1, \ldots, t_p)$ de manière univoque. Comment définir les « chiffres de e » ? Y a-t-il unicité ? Quel est le rapport entre les chiffres t_k et les chiffres de e ?

Quant au calcul, voici une démarche possible, qui n'est rien d'autre qu'un changement de base :

- Tout d'abord, le vecteur $x = (2, 1, 1, \ldots, 1, 1) \in \mathbb{Z}^{m+1}$ représente une valeur approchée $s := \langle x \rangle_!$ vérifiant $s < e < s + \frac{2}{(m+2)!}$. (Le montrer.)
- En convertissant x de base factorielle en base b, comme ci-dessus, on obtient $y \in \mathbb{Z}^{n+1}$ qui représente une valeur approchée $t := \langle y \rangle_b$ vérifiant $t \leq s < t + b^{-n}$.

Ce procédé produit alors un développement $y \in \mathbb{Z}^{n+1}$ en base b qui représente une valeur approchée vérifiant $t < e < t + \varepsilon$ avec une erreur $\varepsilon = \frac{2}{(m+2)!} + b^{-n}$.

Exercice/M 2.11. Pour calculer $p = 10000$ chiffres valables de e nous considérons $n = 10010$, disons. Trouver m tel que $\frac{2}{(m+2)!} \leq b^{-n}$, afin de garantir $\varepsilon < 2b^{-n}$. On fixe ainsi les deux paramètres m et n utilisés dans le procédé ci-dessus. Expliquer pourquoi le développement y ainsi trouvé donne presque n chiffres exacts de e : au pire, le dernier chiffre devrait être augmenté par 1, avec éventuelle propagation des retenues sur les chiffres précédents. Malgré cette ambiguïté concernant les derniers chiffres, pourquoi peut-on espérer d'ainsi obtenir au moins 10000 chiffres valables de e ?

Avant de mettre en œuvre ce développement de e en C++, assurons-nous que le type `int` suffit pour tous les calculs intermédiaires effectués dans l'algorithme IV.6.

Proposition 2.12 (majoration)**.** *Soit $x \in \mathbb{Z}^{m+1}$ normal par rapport à la base factorielle. En effectuant la multiplication par b puis la normalisation, tous les calculs intermédiaires sont majorés par bm.*

DÉMONSTRATION. Commençons par un vecteur normal $x \in \mathbb{Z}^{m+1}$, c'est-à-dire $0 \leq x_k \leq k$. La multiplication par b produit un vecteur $x' = bx$ avec $x'_k = bx_k \leq bk$. Jusqu'ici le plus grand nombre qui puisse apparaître est bm. Montrons que la normalisation suivante ne produit pas de nombres plus grands. On a $bx_m \leq bm < b(m+1)$, donc la retenue $a_m = bx_m \operatorname{div} (m+1)$ est majorée par $b-1$. Pour la position d'avant on obtient ainsi $bx_{m-1} + a_m < bm$, donc la retenue a_{m-1} est également majorée par $b-1$. Par récurrence on conclut que chaque retenue a_k est majorée par $b-1$ et que $bx_{k-1} + a_k < bk$. Ainsi tous les calculs intermédiaires sont majorés par bm. □

Pour le calcul de e, le vecteur initial $x = (2, 1, 1, \ldots, 1) \in \mathbb{Z}^{m+1}$ est normal par rapport à la base factorielle. Sous la condition que bm n'excède pas la capacité de `int`, nous pouvons alors implémenter le calcul de e en utilisant le type `int`.

Exercice/P 2.13. Écrire un programme qui calcule les 10000 premiers chiffres du développement décimal de e. Que valent les chiffres aux positions 9991 à 10000 ?

3. Quelques conseils pour une bonne programmation

Tout travail fait, contemplons la citation « algorithms + data structures = programs » donnée au début du chapitre. Plus concrètement, explicitons quelques recommandations de bon sens pour le développement d'un travail informatique. Dans le cadre de ce cours ces règles ne sont que des conseils, mais elles deviennent primordiales pour tout projet important, en particulier si différentes tâches sont réparties sur une équipe. Dans ce but nous proposons une liste de six étapes essentielles :

1. ***Spécification:*** Cette étape préliminaire consiste à définir ce que l'on *veut* faire. Ainsi on doit détailler le domaine d'application et le résultat exigé, ce qui constitue la *spécification* de la fonction recherchée. Il faut en particulier s'assurer que la spécification correspond bien à l'objectif envisagé, qu'elle est univoque et ne contient pas de contradictions.

   ☞ Pour un projet important cette phase aboutit souvent à la rédaction d'un *cahier des charges* qui fixe l'objectif et détaille la spécification sous forme d'un contrat. Remarquons que la description supplémentaire de ce que la fonction *ne fait pas* peut également servir à clarifier l'objectif.

2. ***Structuration des données et choix des algorithmes:*** Le langage utilisé, le compilateur et les bibliothèques fournissent certaines *structures de données* avec leurs opérations spécifiques. Ceci est le "bas" du programme, c'est-à-dire le niveau le plus élémentaire.

   Pour résoudre le problème spécifié dans l'étape précédente, il faut maintenant décider de la *représentation informatique* des objets en question et décrire les opérations dont on a besoin. Autrement dit, il s'agit de formuler un *modèle informatique* du problème posé et de la solution envisagée. Ceci est le "haut" du programme, c'est-à-dire le niveau le plus complexe.

3. ***Programmation modulaire:*** Le principe consiste à partager un problème donné en plusieurs sous-problèmes que l'on traite successivement, pour arriver finalement aux opérations élémentaires, déjà implémentées. Idéalement c'est une traduction directe de la structuration élaborée dans l'étape précédente. L'implémentation du programme peut ainsi se faire de haut en bas (*top-down*) ou de bas en haut (*bottom-up*).

4. ***Preuve de correction:*** Le programme étant écrit, on le *prouve*, c'est-à-dire qu'on montre qu'il vérifie la spécification. Cette étape développe alors un raisonnement précis et complet, souvent issu de l'analyse faite dans les étapes précédentes. Avouons que la preuve de correction est plus ou moins facile pour les programmes simples, mais elle devient un défi inextricable pour des programmes complexes (ou mal organisés).

   ☞ L'idéal serait la preuve automatique des programmes, ce qui est possible avec certains langages de programmation récents. Évidemment une telle approche demande une programmation beaucoup plus rigoureuse : le programmeur doit fournir le code du programme avec le code de sa preuve. Le compilateur ne peut en général pas *inventer* une preuve de correction, mais il peut très bien *vérifier* une preuve, convenablement formalisée. En tout cas, le C++ en est loin.

5. ***Tests:*** Si tout ce qui précède a été fait correctement, cette étape est superflue. Néanmoins il est en général prudent de s'assurer sur des *exemples* que l'on a atteint le but. À noter que les tests ne peuvent en général porter que sur très peu d'exemples, la difficulté étant de n'oublier aucun cas particulier. En général, la phase de tests s'effectue de bas en haut : on teste d'abord les fonctions élémentaires pour finir avec la fonction principale.

6. ***Présentation:*** Il convient de bien présenter le code source et de le commenter sans retenue. Ce n'est pas seulement une préoccupation esthétique ! Ce souci d'explication du programme assure sa compréhension, puis sa diffusion et son évolution éventuelle. Dans ce but il sera également souhaitable d'établir une documentation, détaillée et complète, indépendante de l'implémentation.

☞ Très souvent un projet de programmation n'est pas un processus linéaire mais *cyclique*. Si la preuve ou les tests (étapes 4 et 5) exhibent des erreurs, on est forcé de réviser l'étape 3 (pour une erreur de programmation) voire recommencer l'étape 2 (pour une erreur de conception plus fondamentale). Si les tests montrent que le programme s'exécute trop lentement ou plus généralement mobilise trop de ressources, on sera mené à reconsidérer l'étape 3 (pour optimiser l'implémentation) voire l'étape 2 (pour choisir des structures et/ou des algorithmes plus efficaces). Si finalement on découvre des ambiguïtés ou même des contradictions dans la spécification, on est forcé de reconsidérer l'étape 1.

Que j'aime à faire apprendre
un nombre utile aux sages !
Glorieux Archimède, artiste, ingénieur,
Toi de qui Syracuse aime encore la gloire,
Soit ton nom conservé par de savants grimoires.

PROJET IV

Calcul de π à 10000 décimales

Objectifs

- Calculer les chiffres de π à une précision donnée.
- Développer une preuve de correction pour cette implémentation.

Ce projet présente une méthode simple, publiée par S. Rabinowitz et S. Wagon en 1995, pour calculer π à une précision donnée près. Cette méthode est suffisamment efficace pour nos besoins, mais surtout elle a le mérite d'être très élémentaire : il ne s'agit que d'un changement de base ! Sur un ordinateur standard on arrive ainsi facilement à calculer les 10000 premiers chiffres de π.

Ce projet se décompose en deux parties : la première discute l'analyse mathématique, la deuxième concerne l'implémentation. Il va sans dire que les deux, une analyse détaillée et une implémentation soigneuse, sont essentielles pour que votre futur programme soit correct. Si vous travaillez soigneusement ces deux étapes, non seulement votre logiciel produira des décimales, mais vous saurez même *prouver* qu'il s'agit des décimales de π.

☞ *Remarque.* — Si vous préférez aller directement aux sources, vous pouvez consulter l'article original de Stanley Rabinowitz et Stan Wagon, *A spigot algorithm for the digits of* π, American Mathematical Monthly 102 (1995), no. 3, pages 195–203. Les auteurs ajoutèrent à la fin une implémentation hâtive en Pascal, dont vous trouverez une traduction en C++ dans le fichier `rabinowitz.cc`. Malheureusement ce programme n'est pas entièrement correct (voir les commentaires dans notre fichier source). Ceci ne diminue en rien le travail mathématique des auteurs, mais souligne l'importance d'une implémentation soigneuse.

Sommaire

1. Quel est le but de ce projet ?

Avant d'entrer dans le vif du sujet, précisons ce que l'on envisage à faire. Tout d'abord on choisit une base $b \geq 2$, disons $b = 10$ pour fixer les idées. Dans le développement en base b on peut écrire $\pi = \sum_{k=0}^{\infty} \pi_k^{(b)} b^{-k}$ avec des chiffres $\pi_k^{(b)} \in [\![0, b[\![$ pour tout $k \geq 1$. Dans la pratique, c'est-à-dire en temps limité, on ne peut calculer qu'un nombre fini de chiffres. On fixe alors un nombre p de chiffres à calculer, ce que nous appelons la précision souhaitée. Le but de ce projet est d'écrire une fonction

```
approx_pi( int base, int precision, vector<int>& chiffres )
```

qui calcule les chiffres $(t_0, t_1, \ldots, t_p)$ d'un nombre rationnel $t = \sum_{k=0}^{p} t_k b^{-k}$ de sorte que l'on puisse garantir $t < \pi < t + b^{-p}$. Afin de garantir sa correction on développera une preuve mathématique.

Exercice/M 1.1 (optionnel). Rappelez pourquoi tout nombre réel admet une représentation en base b, puis expliciter les réels qui en admettent plus d'une. Expliquez pourquoi la représentation est unique pour le nombre π. (Vous pouvez admettre que π est irrationnel.) On peut donc parler du $k$ième chiffre de π dans le développement en base b, ce que nous avons noté $\pi_k^{(b)}$. Expliquer pourquoi la spécification de $(t_0, t_1, \ldots, t_p)$ donnée ci-dessus définit ce vecteur de manière univoque. Quel est le rapport entre les chiffres t_k et les chiffres $\pi_k^{(b)}$?

2. Une représentation convenable de π

Notre point de départ est la présentation de π par la série suivante :

$$\frac{\pi}{2} = \sum_{k=0}^{\infty} \frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)} = 1 + \frac{1}{3} + \frac{1\cdot 2}{3\cdot 5} + \frac{1\cdot 2\cdot 3}{3\cdot 5\cdot 7} + \dots \tag{1}$$

Votre futur programme convertira tout simplement cette série en un développement en base b.

Exercice/M 2.1 (optionnel). Dans tout ce projet vous pouvez considérer la formule précédente comme définition de π. Pour ceux qui se demandent s'il s'agit bien du nombre π usuel, voir par exemple P. Eymard & J.-P. Lafon, *Autour du nombre π*, §2.7 (Édition Hermann, Paris 1999). Voici une preuve élégante, contribution de Vincent Munnier (Licence de Mathématiques à l'Institut Fourier en 2004) :

On considère les intégrales de Wallis, $I_k = \int_0^{\pi/2} \sin^k t\, dt$. On trouve $I_0 = \frac{\pi}{2}$ et $I_1 = 1$, puis, par une intégration par parties, la relation de récurrence $I_{k+1} = \frac{k}{k+1} I_{k-1}$. On obtient ainsi $I_{2k+1} = \frac{k!^2 \cdot 2^{2k}}{(2k+1)!}$. Nous constatons que le terme général de notre série est justement $2^{-k} I_{2k+1}$. Nous avons alors :

$$\sum_{k=0}^{\infty} \frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)} = \sum_{k=0}^{\infty} \int_0^{\pi/2} 2^{-k} \sin^{(2k+1)} t\, dt = \int_0^{\pi/2} \sum_{k=0}^{\infty} 2^{-k} \sin^{(2k+1)} t\, dt$$
$$= \int_0^{\pi/2} \frac{2\sin t}{2-\sin^2 t} = \int_0^{\pi/2} \frac{2\sin t}{1+\cos^2 t} = \big[-2\arctan(\cos t)\big]_0^{\pi/2} = \frac{\pi}{2}$$

Comme exercice, vérifier les détails de ce calcul. Pourquoi peut-on interchanger la somme et l'intégrale ?

3. Développement en base de Wallis et conversion en base b

Comme valeur approchée de π on considère la somme finie

$$s = \sum_{k=0}^{m} 2 \frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)}. \tag{2}$$

Exercice/M 3.1. Montrer que $s < \pi < s + 2^{1-m}$.

En imitant l'approche du chapitre IV, nous introduisons l'application $\langle\cdot\rangle_\pi : \mathbb{Z}^{m+1} \to \mathbb{Q}$ qui associe à chaque suite finie $x = (x_0, x_1, \dots, x_m)$ le nombre rationnel en « base de Wallis »

$$\langle x\rangle_\pi := \sum_{k=0}^{m} x_k \frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)} \tag{3}$$

Par exemple $\langle 2,2,2,2\rangle_\pi = 2 + 2\cdot\frac{1}{3} + 2\cdot\frac{1\cdot 2}{3\cdot 5} + 2\cdot\frac{1\cdot 2\cdot 3}{3\cdot 5\cdot 7}$ est le début de la série présentée ci-dessus.

Définition 3.2. On dit que $x \in \mathbb{Z}^{m+1}$ est *normale* par rapport à la base de Wallis si $0 \le x_k \le 2k$ pour tout $k = 1, \dots, m$. (Comme avant nous ne posons aucune restriction sur la valeur x_0.)

Exercice/M 3.3. Toute représentation $x \in \mathbb{Z}^{m+1}$ peut être normalisée par rapport à la base de Wallis, c'est-à-dire qu'il existe une représentation normale $\bar{x} \in \mathbb{Z}^{m+1}$ telle que $\langle\bar{x}\rangle_\pi = \langle x\rangle_\pi$. Montrer que l'algorithme IV.7 ci-dessous calcule une telle représentation normale à partir de x.

Algorithme IV.7 Normalisation par rapport à la base de Wallis

Entrée: Un vecteur $x = (x_0, x_1, \dots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\bar{x} = \langle x\rangle_\pi$ en base de Wallis

Sortie: Le vecteur normal $x = (x_0, x_1, \dots, x_m) \in \mathbb{Z}^{m+1}$ représentant $\bar{x}$ en base de Wallis

pour k **de** m **à** 1 **faire**	// l'indice k parcourt de droite à gauche
$q_k \leftarrow x_k \operatorname{div} (2k+1)$	// calculer la retenue
$x_{k-1} \leftarrow x_{k-1} + k q_k$	// ajouter la retenue à x_{k-1}
$x_k \leftarrow x_k \bmod (2k+1)$	// réduire x_k modulo $2k+1$
fin pour	
retourner $(x_0, x_1, \dots, x_m)$	

Exercice/M 3.4. Si $x \in \mathbb{Z}^{m+1}$ est normale par rapport à la base de Wallis, alors $x_0 \le \langle x\rangle_\pi < x_0 + 4$. Montrer cette majoration, par exemple en comparant avec une série géométrique convenable.

Exercice/M 3.5 (optionnel). Contrairement aux représentations analysées en chapitre IV, on ne peut pas garantir la majoration $\langle x\rangle_\pi < x_0 + 1$. (Trouver un contre-exemple.) En particulier notre preuve que la représentation normale est unique n'est plus valable pour la base de Wallis. (Trouver également un contre-exemple.) Vous pouvez optimiser notre majoration en montrant que $\langle x\rangle_\pi < x_0 + 2$. Plus précisément, montrer que $\langle 0,2,4,6,\ldots,2m\rangle_\pi = 2 - 2^{m+1}/\binom{2m+1}{m} \to 2$.

Reste à établir la conversion entre développement en base de Wallis et développement en base b :

Exercice/M 3.6. Montrer que l'algorithme IV.8 ci-dessous convertit la représentation $x \in \mathbb{Z}^{m+1}$ en base de Wallis en une représentation $y \in \mathbb{Z}^{n+1}$ en base b telle que $\langle y\rangle_b \le \langle x\rangle_\pi < \langle y\rangle_b + 4b^{-n}$. Les chiffres $y_1,\ldots,y_n$ produits dans la boucle sont-ils forcément normalisés ? Quelle majoration peut-on garantir ? Motiver ainsi la normalisation de y à la fin.

Algorithme IV.8 Changement de base de Wallis en base b avec précision n

Entrée: Un vecteur $x = (x_0, x_1, \ldots, x_m) \in \mathbb{Z}^{m+1}$ et deux entiers $b \ge 2$, $n \ge 0$
Sortie: Un vecteur $y = (y_0, y_1, \ldots, y_n) \in \mathbb{Z}^{n+1}$, normal par rapport à la base b
Garanties: Les nombres représentés sont proches, càd $\langle y\rangle_b \le \langle x\rangle_\pi < \langle y\rangle_b + 4b^{-n}$

```
pour j de 0 à n faire
  normaliser x par rapport à la base de Wallis   // voir l'algorithme précédent
  y_j ← x_0                                      // y_j n'est pas forcément la partie entière de ⟨x⟩_π, mais …
  x_0 ← 0                                        // en soustrayant y_j on assure au moins que 0 ≤ ⟨x⟩_π < 4
  x ← b·x                                        // multiplier tous les chiffres par b
fin pour
normaliser y par rapport à la base b, retourner y
```

Exercice/M 3.7. Supposons que l'algorithme IV.8 est appliqué à un vecteur initial $x \in \mathbb{Z}^{m+1}$ qui est normal par rapport à la base de Wallis. Montrer que les valeurs intermédiaires qui peuvent intervenir pendant les calculs sont majorées par $4bm$. (Vous pouvez prendre le lemme 2.12 pour modèle.) En utilisant le type `unsigned int` à 32 bits et une base $b \in [\![2,16]\!]$, jusqu'à quelle longueur m et quelle précision p peut-on aller ? Même question pour le type `unsigned short` à 16 bits.

4. De l'analyse mathématique à l'implémentation

Nous pouvons maintenant calculer π à une précision donnée près :
- Tout d'abord, le vecteur $x = (2,2,2,\ldots,2,2) \in \mathbb{Z}^{m+1}$ représente une valeur approchée $s := \langle x\rangle_\pi$ de π vérifiant $s < \pi < s + 2^{1-m}$ (voir exercice 3.1).
- En convertissant x de la base de Wallis à la base b, on obtient $y \in \mathbb{Z}^{n+1}$ qui représente une valeur approchée $t := \langle y\rangle_b$ de s telle que $t \le s < t + 4b^{-n}$ (voir exercice 3.6).

Ce procédé produit alors un développement $y \in \mathbb{Z}^{n+1}$ en base b qui représente une valeur approchée t de π telle que $t < \pi < t + \varepsilon$ avec un écart $\varepsilon = 2^{1-m} + 4b^{-n}$.

☞ Si l'on veut arriver à p chiffres valables de π, il faut choisir une précision initiale $\tilde{p}$ légèrement plus grande que la précision souhaitée p. En voici l'argument détaillé :

Exercice/M 4.1. Étant donné $\tilde{p} \in \mathbb{N}$ expliquer pourquoi le choix $n = \tilde{p} + 3$ et $m \ge 2 + \tilde{p}\log_2 b$ permet de calculer $y \in \mathbb{Z}^{n+1}$ tel que $t = \langle y\rangle_b$ satisfasse $t < \pi < t + \varepsilon$ avec un écart $\varepsilon < b^{-\tilde{p}}$. Ensuite, tronquer y à $\tilde{y} = (y_0, y_1, \ldots, y_{\tilde{p}})$ donne presque $\tilde{p}$ chiffres valables de π : au pire, le dernier chiffre devrait être augmenté de 1, avec éventuelle propagation des retenues sur les chiffres précédents. Malgré cette ambiguïté concernant les derniers chiffres, pourquoi peut-on espérer obtenir au moins $p = 10000$ chiffres valables de π en partant d'une longueur initiale de $\tilde{p} = 10010$?

☞ Approche pragmatique : on construit d'abord un vecteur $\tilde{y}$ de longueur $\tilde{p}$ avec précision $b^{-\tilde{p}}$, puis on supprime, un par un, les chiffres terminaux dont on ne peut être sûr. Disons, par exemple, les décimales de π aux positions 760 à 769 sont `3499999983`. Si l'on calcule jusqu'à la position 765, on n'est sûr que des chiffres jusqu'à la position 760. En général, il faut supprimer tous les 9 terminaux ainsi que le chiffre d'avant. (Le justifier.) Même s'il restera finalement moins que les p chiffres souhaités, au moins le programme est honnête. Si besoin en est, l'utilisateur pourra le relancer avec une précision p plus grande.

Exercice/P 4.2. Écrire une fonction `approx_pi` qui prend comme paramètres la base b (comprise entre 2 et 16) et la précision souhaitée p (comprise entre 1 et 10^6) et qui construit le vecteur $(t_0, t_1, \ldots, t_p)$ des p premières décimales de π. Veillez à supprimer, un par un, les chiffres terminaux dont on ne peut garantir la correction. Que valent les chiffres aux positions 9991 à 10000 du développement décimal de π ?

☞ Voici quelques indications :
- On n'a besoin que de deux vecteurs x et y. Ne pas oublier de normaliser y à la fin.
- Il sera utile que le programme affiche l'avancement du travail, par exemple la précision achevée (la taille de y déjà atteinte), éventuellement les chiffres y_k déjà obtenus.
- Pour l'affichage on utilisera les chiffres usuels `0123456789ABCDEF`, voir l'exercice 1.15. L'affichage regroupé en blocs de dix chiffres, avec dix blocs par ligne, semble assez lisible sur l'écran.
- Par curiosité et pour une comparaison/vérification rapide de vos résultats, vous pouvez faire une statistique des chiffres jusqu'à la position p donnée.
- Quand votre programme marche et vous vous êtes convaincu de sa correction, nettoyez le code source et rédigez la version finale des commentaires. Essayez de produire un code dont vous serez fier et qui vous plaira toujours d'ici un an.

5. Quelques points de réflexion

Structuration du projet. Reconsidérez le projet IV (ou III ou I) sous l'angle des remarques faites au chapitre IV, §3. Où la spécification s'est-elle faite dans ce projet ? La structuration des données ? L'implémentation fut-elle menée de haut en bas ou de bas en haut ? Arrive-t-on à prouver que le programme est correct ? En quoi les tests étaient-ils décisifs ? Le programme final arrive-t-il à un niveau « publiable » ? Vaut-il la peine de peaufiner la documentation et la présentation du code source ?

☞ Pour un exemple extrême, reconsidérez le programme I.21, à trois lignes seulement :

```
const int a=10000; int b=52514; vector<int> c(b); int d=0, e=0, f, g, h;
for( ; (f=b-=14); d=1, cout << setw(4) << setfill('0') << g+e/a << flush )
  for( g=e%=a; (h=--f*2); e/=h ) e=e*f+a*( d ? c[f] : a/5 ), c[f]=e%--h;
```

Avec beaucoup d'effort on pourrait exhiber de vagues ressemblance avec l'algorithme de ce projet. Mais sans aucun contexte il semble tombé du ciel. (Ou vient-il plutôt de l'enfer ?) Bien que syntaxiquement correct, ce programme est extrêmement mal présenté et l'absence de toute documentation le rend inutilisable.

Complexité quadratique. Notre approche est suffisamment efficace pour calculer 10000 décimales de π, encore un record mondial vers la fin des années 1950. De plus notre calcul ne nécessite que quelques minutes ! Le programme arrive-t-il aussi bien à calculer 10^5 décimales ? puis 10^6 décimales ? Comment expliquer le ralentissement observé ?

Notre algorithme est d'une complexité dite *quadratique* en fonction de la précision p : afin de calculer un par un les n chiffres de y, on itère une boucle sur $1, \ldots, n$. Dans chaque itération la multiplication puis la normalisation de x exécute elle-même une boucle sur $m, \ldots, 1$. Le nombre total des instructions est donc proportionnel au produit mn, qui vaut à peu près $\text{const} \cdot p^2$. Dix fois plus de chiffres nécessitent donc cent fois plus de temps !

Entre-temps des méthodes beaucoup plus efficaces on été développées, et on est ainsi arrivé en 1999 à calculer plus de $2 \cdot 10^{11}$ décimales de π. Pour en savoir plus vous pouvez consultez *Le fascinant nombre* π de Jean-Paul Delahaye (Pour la Science, Paris 1997) ou π *unleashed*, de Jörg Arndt et Christoph Haenel (Springer-Verlag, Berlin 2001).

Généralisation. Notre approche marche plus généralement pour tout nombre réel σ qui se présente sous forme d'une série $\sigma = \sum_{k=0}^{\infty} x_k v_k$ dans un « système positionnel généralisé ». Pour cela on suppose que les « chiffres » x_k sont des entiers, et que les « valeurs positionnelles » v_k vérifient un certain type de récursion : on suppose $v_0 = 1$ et $v_k = v_{k-1} \frac{p_k}{q_k}$ avec deux nombres naturels $p_k, q_k \geq 1$ vérifiant $\frac{p_k}{q_k} \leq \theta$ pour tout k et une constante $\theta < 1$. Ceci englobe nos trois exemples précédents : la base b (avec $p_k = 1$ et $q_k = b$), la base factorielle (avec $p_k = 1$ et $q_k = k+1$), et la base de Wallis (avec $p_k = k$ et $q_k = 2k+1$).

Si vous voulez, vous pouvez étudier cette nouvelle situation, légèrement généralisée, en y étendant le développement précédent. Si jamais vous tombez sur une constante mathématique qui se présente sous cette forme, vous aurez déjà un algorithme quadratique, ce qui permettra de calculer quelques milliers de décimales.

Partie B

Tri et permutations

Cherchez et vous trouverez,
. . . car qui cherche trouve.
Matthieu **7** 7-8 et Luc **11** 9-10

CHAPITRE V

Recherche et tri

Objectif. Comprendre les techniques de base pour organiser des données ordonnées.

Ce chapitre discute quelques méthodes de recherche et de tri. Les applications sont abondantes ; imaginez par exemple à quel point un dictionnaire serait difficile à utiliser si les mots clés n'étaient pas ordonnés ! Ils le sont, heureusement, car l'éditeur a pris le soin de les *trier*. Vous en profitez en appliquant une méthode efficace de *recherche*. Le projet à la fin de ce chapitre illustre une application moins évidente : la recherche des solutions d'une équation diophantienne (par exemple $x^4+y^4+z^4=w^4$ avec $x,y,z,w\in\mathbb{Z}_+$).

Sommaire

Exemple 0.1. Les deux problèmes fondamentaux, tri et recherche, peuvent se formuler ainsi :

(1) Trier une liste donnée afin d'établir l'ordre $a_1 \le a_2 \le \cdots \le a_n$.

(2) Chercher un élément b dans une liste ordonnée $(a_1 \le a_2 \le \cdots \le a_n)$.

Ajoutons que le tri résout bien d'autres problèmes concernant les listes, a priori non triées :

(3) Trouver les éléments doubles dans une liste (problème de multiplicité).

(4) Vérifier si deux listes sont les mêmes à permutation près (problème d'anagramme).

(5) Trouver la médiane d'une longue liste de valeurs réelles (problème de rang).

Remarque 0.2. D.E. Knuth dans *The Art of Computer Programming* discute diverses méthodes de tri dans le tome 3, intitulé *Sorting and Searching*. Tout au début il fait le constat suivant, toujours valable de nos jours :

> Computer manufacturers estimate that over 25% of the running time on their computers is currently being spent on sorting (. . .) We may conclude that (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms are in common use. The real truth probably involves some of all three alternatives. In any event we can see that sorting is worthy of serious study, as a practical matter.

Ce chapitre tente d'expliquer puis de comparer différentes méthodes de recherche (§1) et de tri (§2–§3). Les exercices mathématiques permettront d'établir la correction des méthodes proposées et de comparer leur performance. Cette partie peut sembler un peu théorique mais elle est très bénéfique pour comprendre ce qu'est l'algorithmique. La morale à retenir : préférer les bons algorithmes aux solutions naïves !

☞ Si vous êtes impatient ou insouciant, vous pouvez lire la recherche dichotomique (algorithme V.2) puis le tri fusion (algorithme V.6) et le tri rapide (algorithme V.7), afin de passer directement à l'implémentation (§4). En §5 nous regardons brièvement quelles solutions offre la STL.

1. Recherche linéaire vs recherche dichotomique

Dans la pratique les données à chercher ou à trier peuvent être des nombres ou des chaînes de caractères ou bien d'autres objets encore. Les méthodes présentées ici s'étendent sans aucune modification aux éléments d'un ensemble ordonné E quelconque. Ceci veut dire que E est muni d'une relation d'ordre $<$ de sorte que pour tous $a, b \in E$ on ait ou $a < b$ ou $a = b$ ou $b < a$ (trichotomie), et que $a < b$ et $b < c$ implique $a < c$ (transitivité). Étant donné un tel ordre $<$ on définit les relations $\leq$, $>$, $\geq$ comme d'habitude, afin d'avoir une écriture plus commode.

1.1. Recherche linéaire. Pour commencer nous allons considérer le problème de la recherche d'un élément dans une liste $A = (a_1, a_2, \ldots, a_n)$. Étant donné un élément b, on souhaite déterminer s'il appartient à la liste A, et si oui, trouver le premier indice k tel que $a_k = b$. La méthode évidente consiste à parcourir toute la liste jusqu'à atteindre l'élément cherché :

Algorithme V.1 Recherche linéaire

Entrée: Un élément b et une liste $A = (a_1, a_2, \ldots, a_n)$.

Sortie: Le premier indice k tel que $a_k = b$, ou bien *non trouvé* si b n'appartient pas à A.

```
pour k de 1 à n faire
  si a_k = b alors retourner k
fin pour
retourner non trouvé
```

Exercice/M 1.1. Expliquer pourquoi la recherche linéaire nécessite n itérations dans le pire des cas, une itération seulement dans le meilleur des cas, et $\frac{n+1}{2}$ itérations en moyenne (en supposant que l'on choisit b dans A au hasard). Pourquoi cette recherche est-elle appelée *linéaire* ?

1.2. Recherche dichotomique. La recherche linéaire se révèle assez inefficace si le nombre d'éléments dans la liste est élevé. La recherche peut être considérablement accélérée si la liste est ordonnée dans le sens que $a_1 \leq a_2 \leq \cdots \leq a_n$. La méthode suivante est un exemple classique du paradigme *diviser pour régner* : on compare d'abord l'élément cherché b avec a_m au milieu de la liste, puis on continue la recherche sur la moitié adéquate de la liste.

Algorithme V.2 Recherche dichotomique

Entrée: Un élément b et une liste ordonnée $A = (a_1, a_2, \ldots, a_n)$.

Sortie: Le premier indice k tel que $a_k = b$, ou bien *non trouvé* si b n'appartient pas à A.

```
i ← 1,  j ← n                          // Si A contient b, alors le premier indice est entre i et j.
tant que i < j faire
  m ← ⌊(i+j)/2⌋                        // calculer l'indice au milieu, arrondi arbitrairement
  si b ≤ a_m alors j ← m sinon i ← m+1  // continuer avec la moitié gauche ou droite
fin tant que
si a_i = b alors retourner i sinon retourner non trouvé
```

Exemple 1.2. Pour voir le fonctionnement de l'algorithme V.2 sur un exemple concret, faites le tourner « à la main » sur la liste ordonnée suivante :

$$(1) \qquad (12, 17, 20, 34,\ 37, 37, 36, 42,\ \ 48, 57, 61, 64,\ 67, 70, 77, 94).$$

Chercher ainsi la première occurrence de 61, puis 37 (qui y figure deux fois), finalement 50 (qui n'y est pas). Pour un exemple plus réaliste vous pouvez ensuite regarder une liste dont la taille n'est pas une puissance de 2, et vérifier que le principe s'applique pareil.

Exercice/M 1.3. Prouver que l'algorithme V.2 est correct. Pour ce faire montrer d'abord qu'il se termine : la différence $j - i$ est un entier positif ou nul qui diminue à chaque itération jusqu'à ce que $i = j$. Vérifier ensuite que chaque itération préserve la propriété suivante : si A contient b, alors le premier indice k tel que $a_k = b$ se trouve dans l'intervalle $[\![i, j]\!]$. Conclure.

Exemple 1.4. Comme dans l'exemple précédent, regardons une liste de longueur 16 mais contenant cette fois-ci les nombres binaires $0000_{\text{bin}},\dots,1111_{\text{bin}}$. Vérifiez que la première itération de la recherche dichotomique divise cette liste en deux partie, à savoir $(0000_{\text{bin}},\dots,0111_{\text{bin}})$ et $(1000_{\text{bin}},\dots,1111_{\text{bin}})$. On détermine ainsi le premier bit. La deuxième itération détermine le deuxième bit, et ainsi de suite. Pour cette raison la recherche dichotomique est aussi appelée *recherche binaire*, ou *binary search* en anglais.

Exercice/M 1.5. Pour $n = 2^k$ vérifier que l'algorithme V.2 nécessite exactement k itérations : initialement l'intervalle $[\![i, j]\!]$ contient 2^k éléments, et chaque itération divise la longueur par deux. Plus généralement, montrer le théorème suivant. Justifier ainsi pourquoi la recherche dichotomique est préférable à la recherche linéaire, voire indispensable si n est grand.

Théorème 1.6. *Pour trouver un élément dans une liste ordonnée de longueur n, la recherche dichotomique définie par l'algorithme V.2 nécessite* $\lceil \log_2 n \rceil$ *ou* $\lfloor \log_2 n \rfloor$ *itérations seulement.*

Remarque 1.7. Comme une variante de la recherche dichotomique vous pouvez reprendre le jeu « trop petit, trop grand » esquissé en chapitre I, exercices 8.6 et 8.7. Pour trouver un nombre dans $[\![1, 127]\!]$ explicitez une stratégie qui ne nécessite que 6 questions. Explicitez ensuite une stratégie dans le cas général. Expliquez pourquoi il s'agit ici d'une recherche « trichotomique » plutôt que « dichotomique ».

1.3. Attention aux détails ! Rappelons que le but d'un algorithme est de préciser sans aucune ambiguïté le procédé à exécuter. L'avantage d'une formulation précise est, bien évidemment, que l'on puisse établir sa correction une fois pour toutes, pour ensuite l'utiliser la conscience tranquille. Pour cette raison il faut faire attention au moindre détail ; toute erreur, même minuscule, peut s'avérer catastrophique dans des futures implémentations :

Exercice/M 1.8. Dans l'algorithme V.2, si l'on remplaçait la condition $i < j$ par la condition $i \leq j$, l'algorithme modifié serait-il correct ? Donner une preuve ou un contre-exemple.

Exercice/M 1.9. Dans l'algorithme V.2, si l'on remplaçait la comparaison $b \leq a_m$ par $b < a_m$, l'algorithme modifié serait-il correct ? Donner une preuve ou un contre-exemple.

Exercice/M 1.10. Dans l'algorithme V.2, si l'on remplaçait $m \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$ par $m \leftarrow \left\lceil \frac{i+j}{2} \right\rceil$, l'algorithme modifié serait-il correct ? Donner une preuve ou un contre-exemple.

2. Trois méthodes de tri élémentaires

Étant donnée une famille $A = (a_1, a_2, \dots, a_n)$ d'éléments dans un ensemble ordonné E, le but d'un tri est de déterminer une permutation $\sigma\colon \{1,2,\dots,n\} \to \{1,2,\dots,n\}$ qui mette les éléments en ordre croissant, c'est-à-dire $a_{\sigma(1)} \leq a_{\sigma(2)} \leq \cdots \leq a_{\sigma(n)}$.

Dans ce qui suit, nous regarderons la variante suivante : on commence par une famille A stockée sous forme d'un tableau, et on souhaite le permuter de sorte que le tableau A' qui en résulte soit ordonné. Afin d'économiser la mémoire et le temps de copie, c'est un seul tableau qui est utilisé, qui au début représente A et qui finit par représenter A'. Ainsi on ne construira pas explicitement la permutation σ, mais directement le résultat de son action sur le tableau A.

2.1. Les plus petits exemples. Avant de discuter des méthodes plus générales, il semble utile de regarder le tri de deux, trois, ou quatre éléments.

Exercice 2.1. Commençons par le plus petit exemple, le tri de deux éléments, de type `int` disons :

```
void trier( int& a, int& b ) { if ( a > b ) { int c=a; a=b; b=c; }; };
```

Expliquer le fonctionnement de cette fonction en distinguant les deux configurations initiales possibles, à savoir $a \leq b$ et $a > b$. La fonction est-elle correcte ? Expliquer l'importance du symbole ' `&` ' dans la liste des paramètres.

Remarque 2.2. Comme l'opération d'échanger deux éléments sera omniprésente dans la suite, il est plus commode de définir une fonction qui effectue ce travail répétitif. On pourrait donc écrire :

```
void echanger( int& a, int& b ) { int c=a; a=b; b=c; };
void trier( int& a, int& b ) { if ( a > b ) echanger(a,b); };
```

Dans cet exemple l'écriture est plus longue, mais elle devient économique à partir de trois éléments.

Exercice 2.3. Après deux, regardons le tri de trois éléments. En voici un candidat :

```
void trier( int& a, int& b, int& c )
{ if ( a > b ) echanger(a,b);
  if ( b > c ) echanger(b,c);
  if ( a > b ) echanger(a,b); };
```

Expliquer le fonctionnement de cette fonction en distinguant toutes les six configurations initiales possibles, allant de $a \leq b \leq c$ jusqu'à $c < b < a$. La fonction est-elle correcte ?

Exercice 2.4. Remarquons que la fonction précédente nécessite toujours 3 comparaisons et effectue entre 0 et 3 échanges. Peut-on trouver une fonction plus efficace ? Voici un candidat :

```
void trier( int& a, int& b, int& c )
{ if ( a > b ) echanger(a,b);
  if ( b > c ) { echanger(b,c); if ( a > b ) echanger(a,b); }; };
```

Expliquer le fonctionnement de cette fonction puis montrer sa correction. Vérifier que cette fonction nécessite le même nombre d'échanges, mais seulement entre 2 et 3 comparaisons. Quel est le nombre moyen de comparaisons si toutes les six configurations initiales sont équiprobables ?

Remarque 2.5. Pour $n \geq 4$ éléments on souhaiterait sans doute une méthode générale de tri, ce que nous discuterons dans les paragraphes suivants. Néanmoins on peut s'intéresser aux méthodes *optimales* de tri, pour une taille n donnée. Une autre façon de ce voir est de regarder un tournoi de n joueurs, que l'on veut classer dans l'ordre de performance (la liste triée) avec un nombre minimum de matchs (comparaisons). Si cette question vous intéresse, consultez Knuth [**8**], §5.3.1.

Exercice 2.6. Écrire une fonction optimale pour trier quatre éléments. Est-ce possible en effectuant au plus 4 comparaisons ? au plus 5 comparaisons ? (Contempler l'inégalité $2^4 < 4! < 2^5$.)

Exercice 2.7. Essayez d'écrire une fonction optimale pour trier cinq éléments. Est-ce possible en effectuant au plus 6 comparaisons ? au plus 7 comparaisons ? (Contempler $2^6 < 5! < 2^7$.)

Exercice 2.8. Avant de continuer, expliquez par quelle méthode vous triez dans la vie quotidienne. Par exemple, comment triez-vous une main de cartes ? Appliquez votre méthode à la famille

$$(7,6,1,3,1,7,2,4) \tag{2}$$

Il y a des fortes chances que votre méthode soit une des trois suivantes :

2.2. Tri par sélection. C'est sans doute la méthode de tri la plus élémentaire. On cherche le plus petit élément a_m parmi $a_1, a_2, \ldots, a_n$, puis on le place au début de la liste. Cette opération est répétée pour la liste restante, jusqu'à ce que tous les éléments soient dans l'ordre.

Exercice 2.9. L'algorithme V.3 réalise un tri par sélection. Montrer sa correction, c'est-à-dire montrer qu'il produit bien une suite ordonnée comme énoncé.

Algorithme V.3 Tri par sélection (*selection sort* en anglais)

Entrée: Une famille $(a_1, a_2, \ldots, a_n)$ d'éléments d'un ensemble ordonné.
Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

pour i **de** 1 **à** $n-1$ **faire**
  $m \leftarrow i$
  pour j **de** $i+1$ **à** n **faire**
    si $a_j < a_m$ **alors** $m \leftarrow j$
  fin pour
  échanger $a_i \leftrightarrow a_m$
fin pour

Exercice/M 2.10. Essayons d'estimer plus précisément le coût de cet algorithme. Supposons que chaque itération de la boucle intérieure nécessite un temps α_1, et que chaque itération de la boucle extérieure ajoute un coût β_1. Vérifier que le coût total de l'algorithme V.3 est alors $c_1(n) = \frac{1}{2}\alpha_1 n(n-1) + \beta_1(n-1)$. Pour n grand, expliquer pourquoi c'est le terme dominant $\frac{1}{2}\alpha_1 n^2$ qui l'emporte.

2.3. Tri par transposition. Une variante du tri par sélection est le tri par transposition, aussi appelé le *tri bulle*. On regarde si deux éléments voisins sont dans le mauvais ordre, si oui on les échange. Cette opération est itérée jusqu'à ce que tous les éléments soient dans l'ordre.

Exercice 2.11. L'algorithme V.4 réalise un tri par transposition. Montrer sa correction. Combien d'itérations faut-il ? Vérifier que son coût est $c_2(n) = \frac{1}{2}\alpha_2 n(n-1) + \beta_2(n-1)$. Pouvez-vous trouver des améliorations ?

Algorithme V.4 Tri par transposition (tri bulle ou *bubble sort* en anglais)

Entrée: Une famille $(a_1, a_2, \ldots, a_n)$ d'éléments d'un ensemble ordonné.

Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

pour i **de** 1 **à** $n-1$ **faire**
  pour j **de** 1 **à** $n-i$ **faire**
    si $a_j > a_{j+1}$ **alors** échanger $a_j \leftrightarrow a_{j+1}$
  fin pour
fin pour

2.4. Tri par insertion. C'est le tri du joueur de cartes. On considère que les éléments de la liste à trier sont donnés un par un. Avant de recevoir l'élément a_i on a déjà trié les éléments $a_1, \ldots, a_{i-1}$. On insère alors l'élément a_i à la position appropriée. Après cette opération la liste $\overline{a_1}, \ldots, \overline{a_i}$ est ordonnée, et on itère jusqu'à ce que tous les éléments soient dans l'ordre.

Exercice 2.12. L'algorithme V.5 réalise un tri par insertion. Montrer sa correction. Combien d'itérations faut-il en moyenne ? Vérifier que son coût moyen est $c_3(n) = \frac{1}{4}\alpha_3 n(n-1) + \beta_3(n-1)$. Peut-on déjà conclure que cet algorithme est supérieur aux précédents ? Quel est l'intérêt des comparaisons empiriques ?

Algorithme V.5 Tri par insertion (*straight insertion sort* en anglais)

Entrée: Une famille $(a_1, a_2, \ldots, a_n)$ d'éléments d'un ensemble ordonné.

Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

pour i **de** 2 **à** n **faire**
  $a \leftarrow a_i$, $j \leftarrow i-1$
  tant que $j > 0$ et $a_j > a$ **faire** $a_{j+1} \leftarrow a_j$, $j \leftarrow j-1$
  $a_{j+1} \leftarrow a$
fin pour

2.5. En sommes-nous contents ? Les méthodes de tri discutées dans ce paragraphe sont assez simples et elles suffisent pour trier les petites listes. Pourtant, dans la pratique il faut souvent trier des listes assez grandes, disons de 10^6 éléments, voire beaucoup plus. (On rencontrera de tels exemples dans le projet V.)

Pour tester dans quelles limites nos méthodes sont praticables, vous pouvez les tester sur des données réelles, par exemple sur des listes « aléatoires ». Pour ceci vous pouvez utiliser la fonction suivante :

```
#include <cstdlib>
void aleatoire( vector<int>& vec  )
{ for ( int i=0; i<vec.size(); ++i ) vec[i]= random()%vec.size(); };
```

Exercice/P 2.13. Implémenter une ou plusieurs des méthodes de tri présentées ci-dessus. Laquelle des méthodes vous semble la plus simple à implémenter et/ou la plus efficace ? Les tester sur des listes aléatoires. Arrive-t-on ainsi à trier une liste de longueur 10^2 ? 10^3 ? 10^4 ? 10^5 ? 10^6 ?

Exemple 2.14. Regardons un annuaire comme les pages blanches (`www.pages-blanches.fr`) qui stocke quelques millions d'entrées, disons 10^8 éléments. Une méthode de tri de complexité quadratique y mettra environ 10^{16} itérations. Même à une vitesse foudroyante de 10^8 itérations par seconde, l'annuaire ne sera prêt que dans trois ans – et déjà dépassé.

Exercice 2.15. Pour un exemple encore plus impressionnant pensez à Google (`www.google.fr`) qui se dit de stocker quelques milliards d'entrées. Estimer la durée d'un tri avec une des méthodes ci-dessus.

3. Diviser pour trier

Les exemples précédents nous mène à la conclusion qu'une méthode de complexité quadratique n'est pas acceptable pour trier de grands fichiers. Les méthodes présentées ci-dessus sont donc inappropriées. Existe-t-il des méthodes de tri qui soient plus efficaces ? Heureusement que oui ! Ceci fait l'objet de ce paragraphe.

L'idée géniale de la recherche dichotomique est de diviser le problème en deux sous-problèmes de taille moitié. C'est le paradigme de « diviser pour régner ». Vu la réussite de cette approche on essaiera de faire pareil pour le tri. Cette idée a donné lieu à un bon nombre d'algorithmes de tri, dont on ne présentera que deux : le tri fusion et le tri rapide.

3.1. Le tri fusion. L'idée du tri fusion est simple : on divise en deux moitiés la liste à trier, on trie chacune d'elles, puis on fusionne les deux moitiés pour reconstituer la liste complète triée.

Algorithme V.6 Tri fusion (*merge sort* en anglais)

Entrée: Une famille $A = (a_1, a_2, \ldots, a_n)$ d'éléments d'un ensemble ordonné.

Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

si $n \leq 1$ **alors retourner** A	// Si $n = 0$ ou $n = 1$ il n'y a rien à trier.
$p \leftarrow \lfloor n/2 \rfloor$, $q \leftarrow \lceil n/2 \rceil$	// décomposer n en deux moitiés p et q
$B = (b_1, \ldots, b_p) \leftarrow (a_1, \ldots, a_p)$	// produire une copie de la moitié gauche
$C = (c_1, \ldots, c_q) \leftarrow (a_{p+1}, \ldots, a_n)$	// produire une copie de la moitié droite
trier la liste B et la liste C	// on sait déjà trier des listes de taille p et q
fusionner B et C dans une liste triée A	// voir exercice 3.1 ci-dessous
retourner A	

Exercice 3.1. Compléter l'algorithme en explicitant comment fusionner les listes B et C dans une seule liste triée A avec n itérations seulement. On pourra traiter B et C comme deux « files d'attente ». Votre algorithme ainsi complété est-il correct ? Comment assure-t-il la terminaison ?

Exemple 3.2. Pour voir comment fonctionne l'algorithme, appliquez-le au tri de la liste (2).

Exercice 3.3. En appliquant le tri fusion à quatre équipes, montrer qu'il existe un tournoi qui ne nécessite que cinq matchs pour établir le classement. Expliquer pourquoi on ne peut pas faire mieux.

3.2. Analyse de complexité. En quoi l'algorithme V.6 est-il intéressant ? Pour y répondre il faut analyser son coût, c'est-à-dire le temps nécessaire pour son exécution. Soit $c(n)$ le coût de l'algorithme V.6 quand on l'applique à une liste de longueur n. Ceci comprend : (1) le coût pour diviser la liste en deux moitiés, (2) le coût pour trier les deux parties, et (3) le coût pour fusionner les deux listes en une seule.

Les étapes (1) et (3) nécessitent chacune n itérations ; leur coût est linéaire en n, disons αn avec une certaine constante $\alpha \geq 0$. L'étape (2) nécessite le tri de la moitié gauche B de taille $p = \lfloor \frac{n}{2} \rfloor$ et de la moitié droite C de taille $q = \lceil \frac{n}{2} \rceil$. S'ajoute un coût constant $\beta \geq 0$ pour le calcul de p et de q etc. Au total, pour $n \geq 2$, on arrive à un coût

$$c(n) = c(\lfloor \frac{n}{2} \rfloor) + c(\lceil \frac{n}{2} \rceil) + \alpha n + \beta \tag{3}$$

Exercice/M 3.4. Pour les puissances $n = 2^k$ vérifier les premières valeurs de $c(n)$:

n	1	2	4	8	16	32
$c(n)$	0	$2\alpha + \beta$	$8\alpha + 3\beta$	$24\alpha + 7\beta$	$64\alpha + 15\beta$	$160\alpha + 31\beta$

On devine déjà que la croissance est moins que quadratique. Plus précisément, pour $n = 2^k$ vous pouvez montrer par récurrence que $c(n) = \alpha n \log_2 n + \beta(n-1)$. Par une variante de cette récurrence, essayez de prouver le théorème suivant :

Théorème 3.5. *Soient $\alpha, \beta \geq 0$. Avec la valeur initiale $c(1) = 0$ la relation de récurrence* (3) *définit une fonction $c \colon \mathbb{Z}_+ \to \mathbb{R}$. Pour tout $n \in \mathbb{Z}_+$ cette fonction vérifie l'inégalité*

$$\alpha n \lfloor \log_2 n \rfloor + \beta(n-1) \quad \leq \quad c(n) \quad \leq \quad \alpha n \lceil \log_2 n \rceil + \beta(n-1).$$

Remarque 3.6. Si $\alpha, \beta > 0$, alors le terme dominant dans cette majoration est $\alpha n \lceil \log_2 n \rceil$, qui croît un peu plus rapidement que le terme linéaire $\beta(n-1)$. On conclut que pour n grand le tri fusion a un coût d'ordre $n \log_2 n$. Ceci est une amélioration énorme vis-à-vis un coût quadratique n^2. Le justifier, par exemple en traçant les deux fonctions ; les évaluer pour $n = 10^2, 10^3, 10^4, 10^5, 10^6$.

Remarque 3.7. Le tri fusion est fait sur mésure pour les listes. Dans ce cas la repartition en deux sous-listes et la fusion en une liste finale ne sont que des manipulations très efficaces de pointeurs et ne nécessitent pas de copier les objets eux-mêmes. En particulier, le tri fusion appliqué sur les listes ne nécessite pas de mémoire temporaire auxiliaire. (Essayez de le détailler si vous connaissez les listes.)

Quand on l'applique sur des vecteurs, par contre, le tri fusion nécessite beaucoup de copie inutiles, ce qui est coûteux en temps et en mémoire. L'algorithme est toujours correct, mais les vecteurs ne sont visiblement pas la structure des données optimale pour cette méthode. Dans une application réaliste il faut parfois tenir compte de cette restriction et du couplage étroit entre algorithme et structure de données.

3.3. Le tri rapide. Le tri fusion discutée ci-dessus est une méthode lucide et éprouvée. Son seul inconvenient est la copie des deux moitiés dans des listes auxiliaires. Le tri rapide ci-dessous évite de telles copies, il est plus rapide *en moyenne*, mais malheureusement sa performance est moins prévisible.

À nouveau l'idée de cet algorithme consiste à séparer en deux la liste initiale. Cette fois-ci on réorganise la liste afin d'obtenir deux parties $B = (a_1, \dots, a_q)$ et $C = (a_p, \dots, a_n)$ de sorte que $a_i \leq a_j$ pour tout $i \in [\![1, q]\!]$ et $j \in [\![p, n]\!]$. La séparation en deux listes se fait à l'aide d'un *pivot* a_m, choisi arbitrairement : on place dans la première partie de la liste les éléments plus petits que le pivot et dans la seconde les éléments plus grands que le pivot. Il suffit ensuite de trier les parties B et C séparément pour arriver à une liste complètement triée.

Algorithme V.7 Tri rapide (*quicksort* en anglais)

Entrée: Une famille $(a_1, a_2, \dots, a_n)$ d'éléments d'un ensemble ordonné.
Sortie: La même famille, permutée de sorte que $a_1 \leq a_2 \leq \cdots \leq a_n$.

```
si n > 1 alors
  m ← ⌊(n+1)/2⌋,  pivot ← a_m          // choisir un pivot, ici on prend l'élément au milieu
  p ← 1,  q ← n                        // p parcourt de gauche à droite, q de droite à gauche
  répéter
    tant que a_p < pivot faire p ← p+1 // trouver le premier élément a_p trop grand
    tant que a_q > pivot faire q ← q−1 // trouver le dernier élément a_q trop petit
    si p > q alors terminer la boucle  // condition d'arrêt anticipé
    si p < q alors échanger a_p ↔ a_q  // échanger a_p et a_q pour qu'ils soient dans l'ordre
    p ← p+1,  q ← q−1                  // faire avancer les indices
  jusqu'à p > q
  trier (a_1, ..., a_q) puis (a_p, ..., a_n)   // appliquer le tri rapide à chacune des deux parties
fin si
```

Exemple 3.8. Pour avoir une idée de son fonctionnement, faites tourner le tri rapide sur les listes de longueur 2 (deux configurations initiales) puis sur les listes de longueur 3 (six configurations initiales). Finalement, pour un exemple plus réaliste, appliquet-le à la liste (2).

Exercice/M 3.9. Montrer que l'algorithme V.7 est correct, c'est-à-dire expliquer pourquoi il produit bien une liste ordonnée comme il le prétend. *Indication.* — Évidemment la partie délicate est la boucle qui réorganise la liste en séparant la partie basse et la partie haute.

(1) Montrer d'abord que chaque itération de la boucle incrémente p et/ou décrémente q. Par conséquent la boucle se termine après un nombre fini d'itérations en assurant $p > q$.

(2) Reste à comprendre les opérations effectuées sur la liste : montrer par récurrence que la boucle garantie toujours l'inégalité $a_i \leq a_j$ pour (i, j) vérifiant $1 \leq i < p$ et $q < j \leq n$.

Comme on finit par $p > q$, on obtient alors deux sous-listes $(a_1, \dots, a_q)$ et $(a_p, \dots, a_n)$ de sorte que $a_i \leq a_j$ pour tout $i \in [\![1, q]\!]$ et $j \in [\![p, n]\!]$, comme souhaité. Conclure.

3.4. Analyse de complexité. L'approche « diviser pour régner » n'est efficace que si les deux sous-problèmes sont de tailles à peu près égales. Pour le tri rapide il est donc souhaitable que le pivot soit la *médiane*. Malheureusement nous ne disposons pas de méthode rapide pour trouver la médiane ; on choisit donc le pivot au milieu, en espérant le mieux. (On pourrait aussi bien prendre a_1 ou a_n ou a_m avec m un indice aléatoire entre 1 et n.) Il se peut, bien sûr, que le pivot soit mal choisi ; dans le pire des cas une des deux sous-listes ne contient qu'un seul élément et l'autre en contient $n-1$. Heureusement, ce cas arrive assez rarement. (Dans un certain sens le tri rapide préfère les données aléatoires, mais il peut bloquer sur des listes avec un méchant ordre initial.) Plus précisément on peut montrer :

Théorème 3.10. *Le tri rapide donné par l'algorithme V.7 est correct dans le sens qu'il trie toute liste donnée d'une longueur n quelconque. Dans le pire des cas cet algorithme nécessite* $\frac{1}{2}n^2$ *itérations environ, mais en moyenne il n'en nécessite que* $2n\ln n$. □

Malgré ses inconvénients, beaucoup de programmeurs préfèrent le tri rapide, car il est facile à implémenter et évite toute copie dans des listes auxiliaires. Avec une implémentation optimisée, il est (en moyenne !) entre 30% et 50% plus rapide que d'autres méthodes de tri. Pour une analyse détaillée et des améliorations possibles voir Knuth [**8**], §5.2.2, ou Sedgewick [**7**], chapitre 9.

4. Fonctions génériques

4.1. Les calamités de la réécriture inutile. Dans les exemples du paragraphe 2 nous avons traité le tri d'entiers. Dans un autre contexte vous voulez peut-être trier des chaînes de caractères, puis des nombres de type `double`, etc... Après réflexion, le problème de tri est toujours le même ! Cependant, pour notre implémentation en C++, nous devons spécifier le type. Que faire ?

Une façon répandue de réaliser ces différentes fonctions est, hélas, l'usage de « copier-coller » puis le remplacement « à la main » de `int` par `string`, et la semaine prochaine de `string` par `double` etc. Supposons que la semaine d'après vous découvrez une erreur cachée dans votre méthode de tri. Vous corrigez alors trois variantes : la variante `int`, la variante `string` puis la variante `double`. Après réflexion vous voulez améliorer votre méthode de tri, donc vous changez à nouveau trois variantes... Il va sans dire que cette approche se révèlera infernale !

4.2. L'usage des fonctions génériques. Heureusement le C++ prévoit un concept qui permet de mieux organiser la programmation, économiser du temps, et éviter les fautes de frappes d'un recopiage erroné. Dans notre exemple, on veut éviter d'écrire plusieurs fonctions quasi identiques :

```
void echanger(    int& a,    int& b ) {    int c=a; a=b; b=c; }
void echanger( string& a, string& b ) { string c=a; a=b; b=c; }
void echanger( double& a, double& b ) { double c=a; a=b; b=c; }
```

Il sera beaucoup plus économique d'écrire une seule fonction générique :

```
template <typename T>
void echanger( T& a, T& b ) { T c=a; a=b; b=c; }
```

Ici le mot réservé `template` signale qu'il s'agit non d'une fonction, mais d'un *modèle de fonction*, aussi appelé une *fonction générique*. Le mot réservé `typename` précède le type indéterminé, en l'occurrence appelé `T`. Ensuite dans la liste des paramètres et les instructions de la fonctions, on utilise `T` comme si c'était un type usuel. Si vous appelez la fonction `echanger` avec deux paramètres de type `int` le compilateur prendra le modèle ci-dessus et créera aussitôt la fonction concrète `void echanger(int& a,int& b)` comme souhaitée, simplement en remplaçant `T` par `int`. Malin, non ? La fonction concrète ainsi générée est appelée une *instance* du modèle.

Les versions génériques de nos fonctions de tri s'écrivent alors ainsi :

```
template <typename T>
void trier( T& a, T& b ) { if ( a > b ) echanger(a,b); }
```

```
template <typename T>
void trier( T& a, T& b, T& c )
{ if ( a > b ) echanger(a,b);
  if ( b > c ) { echanger(b,c); if ( a > b ) echanger(a,b); }; }
```

Remarque 4.1. Nous avons déjà fait la connaissance de la programmation générique en chapitre I : la classe `vector` est en fait une classe générique dans le sens expliqué plus haut. Sa déclaration est donc

```
template <typename T>
class vector;
```

Si l'on veut l'utiliser pour les éléments de type `int`, par exemple, le compilateur prend le modèle `vector` et en construit la classe souhaitée `vector<int>` simplement en remplaçant `T` par `int`. Avec le même effort de programmation, la classe générique peut ainsi être utilisée dans des situations les plus variées.

Exercice/P 4.2. En s'inspirant de l'opérateur de sortie `<<` défini dans le programme I.18, définir un opérateur de sortie générique qui soit capable d'afficher un vecteur d'un type `T` quelconque. (Pour une solution voir `vectorio.cc`).

4.3. Implémentation de recherche et tri en C++. Si vous voulez, vous pouvez mettre en œuvre les méthodes de recherche et de tri discutées ci-dessus en implémentant une ou plusieurs des fonctions suivantes en C++. Essayez dès le début d´écrire des fonctions génériques comme expliqué en §4. Si pour certaines fonctions vous n'y arrivez pas, vous pouvez avoir recours aux solutions offertes par la bibliothèque STL, discutées dans le paragraphe suivant.

Exercice/P 4.3. Écrire une fonction générique qui teste si le vecteur passé en paramètre est trié :

```
template <typename T>
bool est_trie( const vector<T>& vec );
```

Expliquer pourquoi ce mode de passage est le mieux adapté pour cette tâche.

Exercice/P 4.4. Écrire une fonction générique qui effectue une recherche dichotomique :

```
template <typename T>
int recherche_dichotomique( const vector<T>& vec, const T& element );
```

Expliquer à nouveau pourquoi ce mode de passage est le mieux adapté.

Pour les méthodes de tri vous pouvez choisir entre le tri rapide et le tri fusion :

Exercice/P 4.5. Implémenter le tri rapide en définissant la fonction générique suivante :

```
template <typename T>
void tri_rapide( vector<T>& vec, int gauche, int droite )
```

Ensuite, pour l'appel initial du tri rapide, on pourrait fournir la fonction suivante :

```
template <typename T>
void tri_rapide( vector<T>& vec ) { tri_rapide( vec, 0, vec.size()-1 ); }
```

À noter que les appels récursifs passent le vecteur tout entier comme paramètre par référence ; ce sont les indices `gauche` et `droite` qui spécifie la partie à trier. Expliquer pourquoi ce mode de passage est le mieux adapté pour éviter tout recopiage inutile.

Exercice/P 4.6. Implémenter le tri fusion en définissant la fonction suivante :

```
template <typename T>
void tri_fusion( vector<T>& vec )
```

Contrairement au tri rapide le tri fusion nécessitera des copies dans des vecteurs auxiliaires.
Remarque. — Dans le tri fusion ainsi que le tri rapide il est en général plus efficace de trier les toutes petites listes (disons $n \leq 3$) « à la main » comme en §2.1 Voyez-vous pourquoi ?

Exercice/P 4.7. Vérifiez votre implémentation puis testez sa correction sur beaucoup d'exemples. Si vous voulez tester systématiquement un grand nombre d'exemples, vous pouvez écrire une fonction qui le fait pour vous : d'abord on crée un vecteur aléatoire, ensuite on le trie puis vérifie le résultat par la fonction de l'exercice 4.3. Sur un tel vecteur ordonné on peut finalement tester la recherche dichotomique en cherchant un par un les éléments du vecteur.

Exercice/P 4.8. Mesurer la performance de votre implémentation : combien de temps faut-il pour trier une liste de taille 10^2 ? 10^3 ? 10^4 ? 10^5 ? 10^6 ? Comparer avec une méthode de tri élémentaire. Êtes-vous content ?

5. Solutions fournies par la STL

5.1. Algorithmes de recherche et tri. Comme le tri et la recherche sont omniprésents dans la programmation, la bibliothèque STL en fournit un bon nombre de fonctions génériques. Elles sont disponibles après inclusion du fichier en-tête correspondant par la directive `#include <algorithm>` ; le programme V.1 ci-dessous en illustre l'usage.

Programme V.1 Tri et recherche avec les algorithmes de la STL `tri.cc`

```
#include <iostream>     // pour déclarer l'entrée-sortie standard
#include <algorithm>    // pour définir les fonctions de tri et de recherche
#include <vector>       // pour définir la class générique vector
#include <cstdlib>      // pour déclarer la fonction random()
using namespace std;

template <typename T>
ostream& operator << ( ostream& out, const vector<T>& vec )
{
  out << "( ";
  for ( size_t i=0; i<vec.size(); ++i ) out << vec[i] << " ";
  out << ")";
  return out;
}

void aleatoire( vector<int>& vec  )
{
  for ( size_t i=0; i<vec.size(); ++i ) vec[i]= random() % vec.size();
}

int main()
{
  cout << "\nBienvenue aux vecteurs aléatoires !" << endl;
  cout << "Entrez la longueur svp : ";
  int longueur;
  cin >> longueur;
  vector<int> vec(longueur);
  aleatoire(vec);
  cout << "\nvoici un vecteur aléatoire :  " << vec << endl;
  sort( vec.begin(), vec.end() );
  cout << "\naprès un tri rapide :         " << vec << endl;
  random_shuffle( vec.begin(), vec.end() );
  cout << "\naprès une mélange aléatoire : " << vec << endl;
  stable_sort( vec.begin(), vec.end() );
  cout << "\naprès un tri fusion :         " << vec << endl;
  cout << "\nEntrez un nombre à chercher svp : ";
  int element;
  cin >> element;
  bool trouve= binary_search( vec.begin(), vec.end(), element );
  if ( trouve ) cout << "L'élément " << element << " appartient à la liste.\n";
  else cout << "L'élément " << element << " n'appartient pas à la liste.\n";
  cout << "Au revoir.\n" << endl;
}
```

Pour trier un vecteur `vec`, la STL prévoit entre autre les deux fonctions suivantes :

```
sort( vec.begin(), vec.end() );         // tri rapide (quick sort)
stable_sort( vec.begin(), vec.end() );  // tri fusion (merge sort)
```

La première effectue un tri rapide, la deuxième un tri fusion. Comme expliqué plus haut le tri fusion est légèrement moins rapide et nécessite de la mémoire auxiliaire, mais il *garantit* un temps d'exécution majoré par $\alpha n \log_2 n$. Le tri rapide, par contre, est un peu plus rapide *en moyenne*, mais très lent dans le pire des cas. À choisir en fonction des applications envisagées.

Ajoutons qu'il existe une fonction « inverse » qui mélange un vecteur de manière aléatoire :

```
random_shuffle( vec.begin(), vec.end() );
```

La fonction `binary_search` effectue une recherche dichotomique d'un élément `a` dans un vecteur ordonné :

```
binary_search( vec.begin(), vec.end(), a );
```

Elle renvoie le booléen `true` si `a` est un élément du vecteur, et `false` sinon. Il existe aussi des variantes qui renvoient le premier ou le dernier indice `k` tel que `vec[k]` vaut `a`. Pour en savoir plus sur les algorithmes de la STL, consulter la documentation sur le site `http://www.sgi.com/stl/`.

Le plus simple de tous ces algorithmes est l'échange de deux éléments, déjà rencontré en §4 :

```
template <typename T>
void swap( T& a, T& b ) { T c=a; a=b; b=c; };
```

Notons cependant que la fonction `swap` peut encore être optimisée : si les objects à échanger sont grands, il est plus efficace d'échanger des pointer au lieu des données elles-mêmes. Pour cette raison la STL implémente des fonctions `swap` spécialisée et optimisée sur mesure pour des vecteurs, des listes, des chaînes de catactères, etc. Lorsqu'une version spécialisée est disponible, c'est elle qui sera utilisée. Seulement par défaut utilise-t-on la version générique ci-dessus.

5.2. La classe générique set. La bibliothèque STL fournit une classe générique `set` pour modéliser un *ensemble fini* d'objets d'un type donné. Dans l'exemple qui suit on regarde une variable `ensemble` du type `set<int>`, mais tout s'adapte à n'importe quel autre type au lieu de `int`. Voici les principales opérations que l'on veut effectuer sur un tel ensemble :

(1) Ajouter un élément `a` : ceci se réalise par la fonction `ensemble.insert(a)`.

(2) Enlever un élément `a` : ceci se réalise par la fonction `ensemble.erase(a)`.

(3) Tester si `a` est un élément de `ensemble` : ceci peut se réaliser par `ensemble.count(a)`.

Remarque 5.1. On voit déjà que l'implémentation des ensembles est étroitement liée aux questions de tri et de recherche. Bien sûr on peut implémenter un ensemble comme une liste non ordonnée de n éléments. Pourtant, ceci entraînerait que le test « `a` appartient à `ensemble` » doit parcourir toute la liste, ce qui nécessite un temps proportionnel à n (voir §1). Pour des applications réalistes cette méthode n'est pas praticable.

Dans le souci d'efficacité il est donc nécessaire de stocker les éléments de manière ordonnée. Dans ce cas le test d'appartenance se réalise en temps $\alpha \log_2 n$, par une recherche dichotomique (voir §1). Pour que les opérations `insert` et `erase` soient aussi efficaces, on stocke les éléments non dans un vecteur mais dans une structure mieux adaptée : un arbre binaire.

En faisant abstraction des détails d'une telle implémentation, la conclusion est la suivante :

Théorème 5.2. *On peut implémenter un ensemble fini de sorte que les opérations élémentaires ci-dessus s'effectuent en un temps majoré par $\alpha \log_2 n$, où n est la taille de l'ensemble.* □

Soulignons à nouveau l'importance d'une implémentation efficace : Pour traiter un ensemble d'un million d'éléments, disons, on préfère attendre quelques secondes plutôt que quelques heures !

Le programme V.2 illustre l'utilisation de la classe `set`. Comme vous constaterez, les éléments d'un ensemble sont toujours stockés de manière ordonnée, et chaque élément y apparaît au plus une fois (ce qui sont les principales différences entre un ensemble et une liste).

Pour modéliser des familles avec répétitions possibles on utilise la classe générique `multiset`. Remplacer `set` par `multiset` dans le programme V.2, le tester puis expliquer les différences dans le comportement du logiciel. Le souci d'harmoniser les fonctions pour `set` et `multiset` explique aussi le nom, a priori bizarre, de la fonction `set.count(a)`.

5.3. Les itérateurs. Il y a un concept annexe qui est d'une grande importance pratique : pour afficher un ensemble on souhaite le parcourir du début à la fin. Comme indiqué plus haut, les éléments ne sont pas stockés dans un vecteur, en particulier l'utilisateur ne peut pas les adresser par indice. La solution : on parcourt un ensemble à l'aide d'un *itérateur*. Ceci est illustré dans le programme V.2 : l'itérateur `it` est

Programme V.2 Ensemble d'entiers modélisé par `set<int>` `set.cc`

```
1   #include <iostream>     // déclarer l'entrée-sortie standard
2   #include <set>          // pour définir la class générique set
3   using namespace std;
4
5   template <typename T>
6   ostream& operator << ( ostream& out, const set<T>& ens )
7   {
8     out << "{ ";
9     typename set<T>::const_iterator it;
10    for ( it= ens.begin(); it!=ens.end(); ++it )
11      out << *it << " ";
12    out << "}";
13    return out;
14  }
15
16  int main()
17  {
18    cout << "\nBienvenue aux ensembles ! On teste ici la classe set.\n"
19         << "Entrez un entier positif pour l'ajouter, négatif pour effacer, "
20         << "0 pour terminer." << endl;
21    set<int> ensemble;
22    for(;;)
23      {
24        cout << "ensemble = " << ensemble << endl;
25        cout << "Entrez un entier svp : ";
26        int element;
27        cin >> element;
28        if ( element == 0 ) break;
29        if ( element  > 0 ) ensemble.insert( element );
30        else                ensemble.erase( -element );
31      }
32    for(;;)
33      {
34        cout << "Entrez un nombre à chercher svp : ";
35        int element;
36        cin >> element;
37        if ( element == 0 ) break;
38        if ( ensemble.count( element ) > 0 )
39          cout << "L'élément " << element << " appartient à l'ensemble.\n";
40        else
41          cout << "L'élément " << element << " n'appartient pas à l'ensemble.\n";
42      }
43    cout << "Au revoir.\n" << endl;
44  }
```

une sorte de pointeur sur un élément de notre ensemble. On peut l'incrémenter par `++it` et décrémenter par `--it`, pour avancer ou reculer. Puis on peut accéder à l'élément vers lequel il pointe par `*it`. Pour parcourir du début à la fin, l'ensemble `ens` fournit les itérateurs `ens.begin()` et `ens.end()`. Comme vous pouvez le constater, la construction d'une boucle devient ainsi facile et naturelle.

Mais vérifiez tout : ce qui est bon, retenez-le !
1 Thess. **5** 21

PROJET V

Applications du tri aux équations diophantiennes

Objectif. Le tri peut se révéler un outil magnifique même dans des situations inattendues. Dans ce projet nous allons appliquer le tri aux équations $a^3+b^3=c^3+d^3$ et $a^4+b^4=c^4+d^4$. Finalement on reprend la conjecture d'Euler concernant l'équation $a^4+b^4+c^4=d^4$. Il y aura des surprises…

Ce projet nous permet entre autre de tester nos méthodes de tri et de recherche sur des exemples réalistes, et avec des algorithmes efficaces le temps d'exécution restera raisonnable. On constatera que la mémoire disponible peut également être une ressource précieuse.

Sommaire

1. Le taxi de Ramanujan

L'anecdote suivante est devenue légendaire : Un jour quand le mathématicien G.H. Hardy rendit visite à son collègue et ami S. Ramanujan, il mentionna le numéro du taxi avec lequel il était venu, en ajoutant « *C'est sans doute un nombre sans aucune propriété intéressante.* » « *Au contraire !* » répondit Ramanujan « *C'est le plus petit nombre qui se décompose de deux manières différentes en somme de deux cubes d'entiers positifs.* » Peut-être vous connaissez cette histoire mais vous avez oublié le numéro du taxi. Comment le retrouver rapidement ?

— ∗ —

Nous nous proposons ici de chercher les plus petits nombres qui s'écrivent de deux manières différentes comme somme de deux cubes d'entiers positifs. Autrement dit nous cherchons les petites solutions de l'équation $a^3+b^3=c^3+d^3$ avec $a,b,c,d\in\mathbb{Z}_+$ et $\{a,b\}\neq\{c,d\}$. Voici trois démarches qui se présentent :

Exercice 1.1. On peut effectuer une recherche exhaustive qui parcourt les quadruplets (a,b,c,d) vérifiant $1\leq a<c\leq d<b\leq N$ (le justifier). Vérifier que cette méthode nécessite $\binom{N+1}{4}$ itérations.

Exercice 1.2. En s'inspirant du projet **I**, expliciter une méthode qui ne nécessite que $\binom{N}{3}$ itérations grâce à une fonction auxiliaire $n\mapsto\lfloor\sqrt[3]{n}\rfloor$. En quoi cette approche est-elle préférable à la méthode précédente ?

Exercice 1.3. Détaillez une méthode utilisant le tri : pour $1\leq a\leq b\leq N$ on construit une liste des valeurs a^3+b^3, on la trie, puis on cherche les doublons. Combien de mémoire est necessaire pour stocker la liste ? Combien d'itérations sont nécessaire pour le tri puis la recherche ?

Remarque 1.4. Comme quatrième approche vous pouvez effectuer une recherche sur internet. Esquisser en quoi cette dernière approche, elle aussi, est basée sur des méthodes de tri et de recherche.

Exercice/P 1.5. Choisissez la méthode qui vous semble la plus efficace et/ou la plus simple à réaliser, puis justifier votre choix. Implémentez-la afin de trouver le numéro du taxi de Ramanujan. Ce résultat serait-il également accessible par les autres méthodes ? Combien de solutions y a-t-il pour $N=500$? $N=1000$? $N=5000$? Jusqu'où peut-on aller ?

Exercice 1.6. Trouver le plus petit nombre qui s'écrive de *trois* manières différentes comme somme de deux cubes d'entiers positifs. (À l'heure actuelle on connaît la réponse pour les multiplicités 2, 3, 4, et 5 seulement.)

Exercice 1.7. Pouvez-vous trouver le plus petit nombre qui s'écrive de deux manières différentes comme somme de deux bicarrés d'entiers positifs, $a^4+b^4=c^4+d^4$? (Dans ce cas on ne connaît aucun exemple de multiplicité ≥ 3.)

1.1. Équations diophantiennes, recherche et tri. En généralisant l'exemple précédent on arrive à la formulation suivante : étant données deux fonctions $P,Q\colon \mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$, comment énumérer rapidement toutes les solutions de l'équation $P(a,b)=Q(c,d)$ avec $(a,b,c,d)\in[\![1,N]\!]^4$? Bien sûr on peut parcourir tous les quadruplets (a,b,c,d) et tester si l'équation $P(a,b)=Q(c,d)$ est satisfaite. Ceci peut s'implémenter par quatre boucles imbriquées, et nécessite N^4 itérations au total. Peut-on faire mieux ?

Exercice 1.8. Expliciter une méthode bénéficiant d'une méthode efficace de tri :

(1) On dresse une liste de tous les triplets $(P(a,b),a,b)$, puis on la trie par rapport à P.

(2) On dresse une liste de tous les triplets $(Q(c,d),c,d)$, puis on la trie par rapport à Q.

(3) On parcourt les listes simultanément en cherchant les coïncidences $P(a,b)=Q(c,d)$.

Estimer la mémoire nécessaire et le temps d'exécution. Pourquoi est-il hors de question d'utiliser un tri de complexité quadratique ici ?

Exercice 1.9. Expliciter une variante bénéficiant des méthodes efficaces de tri et de recherche :

(1) On dresse une liste de tous les triplets $(Q(c,d),c,d)$, puis on la trie par rapport à Q.

(2) On parcourt tous les triplets $(P(a,b),a,b)$ en cherchant les coïncidences avec la liste.

Estimer la mémoire nécessaire et le temps d'exécution. Pourquoi est-il hors de question d'utiliser une recherche linéaire ici ?

2. L'équation $a^4+b^4+c^4=d^4$ reconsidérée

Regardons à nouveau l'équation $a^4+b^4+c^4=d^4$ vue en projet I. L'approche ci-dessus s'applique avec $P(a,b)=a^4+b^4$ et $Q(c,d)=d^4-c^4$. Nos nouvelles méthodes sont plus efficaces que l'approche du projet I et nous permettrons de chercher des solutions dans un domaine beaucoup plus large ! Le but ultime sera de trouver toutes les solutions $(a,b,c,d)\in[\![1,N]\!]^4$ avec $N=10^6$. Comme dans le projet I nous posons

```
typedef unsigned long long int Integer;
```

Exercice/P 2.1. Chercher les solutions jusqu'à $N=2000$ par la méthode esquissée ci-dessus : stocker les valeurs d^4-c^4 avec $1\leq c<d\leq N$ dans un vecteur de type `vector<Integer>`, le trier, puis chercher les valeurs a^4+b^4 avec $1\leq a\leq b\leq N$ dans la liste.

☞ *Étapes intermédiaires :* Comme d'habitude il sera utile que le programme affiche les différentes étapes (construction, tri, recherche) et l'avancement du travail (pour les boucles extérieures). Si vous avez bien programmé, votre logiciel s'exécute en quelques secondes. Félicitations !

Remarque 2.2. Pour $N=2000$ il faut construire un vecteur de longueur $\frac{1}{2}N(N-1)=1999000$. Comme on connaît sa taille d'avance, il est utile de *réserver* la mémoire tout au début, par l'instruction suivante :

```
vector<Integer> imageQ; imageQ.reserve(max*(max-1)/2);
```

Ceci définit un vecteur vide mais prévoit déjà la mémoire indiquée. Ensuite vous pouvez remplir le vecteur par la fonction `imageQ.push_back()`, tout en évitant la réallocation inutile pendant la construction.

Remarque 2.3. Notre approche marche encore pour $N=5000$, peut-être même jusqu'à $N=10000$, mais au delà les limites de cette méthode se font sentir : le tester. il se trouve qu'ici la mémoire devient le facteur limitant. De combien de mémoire vive dispose la machine sur laquelle vous travaillez ? Quand la mémoire commence à manquer il se peut que le tri fusion n'aboutit plus alors que le tri rapide marche encore (expliquer pourquoi).

2.1. Restrictions modulaires. Jusqu'ici notre méthode est très générale est s'applique à des fonctions $P,Q\colon\mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$ quelconques. Pour aller plus loin nous utilisons le fait que $P(a,b)=a^4+b^4$ et $Q(c,d)=d^4-c^4$ sont des *polynômes* et de plus *homogènes*.

Exercice/M 2.4. Vérifier que toute solution $(a,b,c,d)\in\mathbb{Z}^4$ de l'équation $P(a,b)=Q(c,d)$ entraîne une famille infinie de solutions (ka,kb,kc,kd) avec $k\in\mathbb{Z}$. Justifier que notre recherche peut se restreindre aux solutions *primitives*, c'est-à-dire celles qui vérifient $\operatorname{pgcd}(a,b,c,d)=1$.

Exercice/M 2.5. Vérifier que toute solution $(a,b,c,d)\in\mathbb{Z}^4$ de l'équation $P(a,b)=Q(c,d)$ induit une solution $(\bar{a},\bar{b},\bar{c},\bar{d})\in\mathbb{Z}_n^4$ de l'équation $\bar{P}(\bar{a},\bar{b})=\bar{Q}(\bar{c},\bar{d})$ modulo n. Par contraposé, comment l'inégalité $\bar{P}(\bar{a},\bar{b})\neq\bar{Q}(\bar{c},\bar{d})$ peut-elle simplifier la recherche des solutions dans $\mathbb{Z}^4$?

Exercice/M 2.6. Montrer que l'application $\mathbb{Z}_4 \to \mathbb{Z}_4$, $x \mapsto x^4$ a pour l'image l'ensemble $\{0,1\}$. En déduire que, quitte à permuter a,b,c, toute solution $(a,b,c,d) \in \mathbb{Z}^4$ vérifie $a \equiv b \equiv 0 \pmod 2$. Pour une solution primitive on peut de plus conclure que $c \equiv d \equiv 1 \pmod 2$.

Exercice/M 2.7. Montrer que l'application $\mathbb{Z}_5 \to \mathbb{Z}_5$, $x \mapsto x^4$ a pour l'image l'ensemble $\{0,1\}$. Quitte à permuter a,b,c, toute solution $(a,b,c,d) \in \mathbb{Z}^4$ vérifie donc $a \equiv b \equiv 0 \pmod 5$. Pour une solution primitive on peut de plus conclure que $c \not\equiv 0 \not\equiv d \pmod 5$.

Avec cette technique on arrive au résultat suivant, que l'on admettra (cf. M. Ward, *Euler's problem on sums of three fourth powers*, Duke Mathematical Journal 15 (1948) 827–837) :

Théorème 2.8 (dû à M. Ward, 1948)**.** *Si $(a,b,c,d) \in \mathbb{Z}^4$ est une solution primitive de l'équation $a^4+b^4+c^4=d^4$, alors $d \not\equiv 0 \pmod 5$ et $d \equiv 1 \pmod 8$. Quitte à permuter les variables a,b,c on a $a \equiv 0 \pmod 8$ et $b \equiv 0 \pmod{40}$ ainsi que $c \equiv \pm d \pmod{1024}$.* □

☞ *Optimisation :* L'intérêt pratique de ce théorème est qu'il suffit désormais de parcourir les couples (a,b) et (c,d) vérifiant les dites congruences. En C++ ceci peut se réaliser par les boucles suivantes :

```
for ( Integer a=8; a<=max; a+=8 )
  for ( Integer b=40; b<=max; b+=40 ) { ... };
for ( Integer d=1; d<=max; d+=8 ) if ( d%5 != 0 )
  { for ( Integer c=d%1024; c<d; c+=1024 ) { ... };
    for ( Integer c=1024-d%1024; c<d; c+=1024 ) { ... }; };
```

Cette astuce écarte plus de 99% de couples $(a,b) \in [\![1,N]\!]^2$ car on n'en retient qu'une fraction $\frac{1}{320}$; mieux encore, il écarte 99,99% des couples $(c,d) \in [\![1,N]\!]^2$ car on n'en retient qu'une fraction $\frac{1}{10240}$ (le justifier).

Exercice/P 2.9. Chercher toutes les solutions jusqu'à $N = 100000$ par la méthode améliorée. Vérifier que l'on construit un vecteur de longueur 971529 seulement.

Si vous voulez aller encore un peu plus loin, vous pouvez écarter 99% des couples (c,d) restants en appliquant d'autres cribles modulaires : on fixe un nombre n et on regarde quels couples (c,d) apparaissent dans les solutions de l'équation $\overline{a}^4+\overline{b}^4=\overline{d}^4-\overline{c}^4$ modulo n. Si (c,d) est impossible modulo n, alors il est impossible dans $\mathbb{Z}$. Le programme V.3 montre une implémentation de cette idée en C++.

Exercice/P 2.10. Chercher toutes les solutions jusqu'à $N = 500000$ par la méthode améliorée. Vérifier que l'on construit un vecteur de longueur 418127 seulement.

☞ *Avertissement.* — Le choix du type `long long int` entraîne que nous travaillons modulo 2^{64} : si le logiciel trouve une solution, ceci ne veut dire que $a^4+b^4+c^4 \equiv d^4 \pmod{2^{64}}$. Interprété correctement, c'est un résultat précieux, car une fois trouvée, il est très facile de confirmer ou réfuter une solution :

Pour une solution prospective a,b donnée, chercher les valeurs c,d associéés, puis afficher le quadruplet (a,b,c,d) comme « candidat ». Ensuite on peut effectuer un calcul dans $\mathbb{Z}$ pour vérifier si (a,b,c,d) est effectivement une solution. (On pourra utiliser `mpz_class` pour le calcul avec des grands entiers.)

Remarque 2.11. Pour résumer, nous avons raffiné successivement nos méthodes : l'idée de la méthode 1, notre point de départ, est de stocker les valeurs $P(c,d)$ dans un vecteur trié. La méthode 2 restreint les couples (c,d) par certaines congruences (théorème 2.8). La méthode 3 généralise cette idée et implémente plusieurs cribles modulaires (programme V.3). Pour souligner l'utilité de ces raffinements successifs, le tableau ci-dessus présente le nombre des couples (c,d) à considérer avec $1 \le c < d \le N$. Grâce aux restrictions modulaires exploitées par la méthode 3, la construction du vecteur se fait assez rapidement, et le tri suivant se passe également sans problème. C'est la dernière phase, la recherche des coïncidences, qui est la plus coûteuse en temps. Pour $N = 10^6$ il faut compter environ une heure d'exécution.

$N=$	5000	10000	50000	100000	500000	1000000
méthode 1	12497500	49995000	1249975000	4999950000	124999750000	499999500000
méthode 2	2194	9268	241627	971529	24388874	97605862
méthode 3	61	183	3623	15373	418127	1672259

Remarque 2.12. On peut encore optimiser la mémoire nécessaire pour l'énumération exhaustive, en utilisant une *file de priorité* au lieu d'un tableau. Pour en savoir plus, consultez l'article de Daniel J. Bernstein, *Enumerating solutions to $p(a)+q(b)=r(c)+s(d)$*, Mathematics of Computation 70 (2001), 389–394.

Programme V.3 Cribles modulaires pour le problème d'Euler `crible.cc`

```
#include <iostream>
#include <vector>
using namespace std;
typedef unsigned long long int Integer;

// Calculer  P(a,b) = a^4 + b^4  modulo mod (sans débordement)
Integer lePolynomeP( Integer a, Integer b, Integer mod )
{
  a%=mod; a*=a; a%=mod; a*=a; a%=mod;
  b%=mod; b*=b; b%=mod; b*=b; b%=mod;
  return (a+b) % mod;
}

// Calculer  Q(c,d) = d^4 - c^4  modulo mod (sans débordement)
Integer lePolynomeQ( Integer c, Integer d, Integer mod )
{
  c%=mod; c*=c; c%=mod; c*=c; c%=mod;
  d%=mod; d*=d; d%=mod; d*=d; d%=mod;
  return ( c<=d ? d-c : d-c+mod );
}

// Pour tenir compte des restrictions modulaires, la fonction suivante
// construit un vecteur  image  du type  vector<bool>  et de longueur mod
// de sorte que  image[x]==true  ssi x est dans l'image de P modulo mod.
vector<bool> imagePmod( const int& mod )
{
  cout << "construction de l'image de P mod " << mod << " ... " << flush;
  vector<bool> image(mod,false);
  for ( int a=0; a<mod; ++a )
    for ( int b=0; b<mod; ++b )
      image[ lePolynomeP(a,b,mod) ]= true;
  cout << "statistique : " << flush;
  Integer possibles=0;
  for ( int c=0; c<mod; ++c )
    for ( int d=0; d<mod; ++d )
      if ( image[ lePolynomeQ(c,d,mod) ] ) ++possibles;
  cout << (100*possibles)/(mod*mod) << "% retenus" << endl;
  return image;
}

// Construction des images modulo 5^4,13^2,3^4,29^2,7^3,11^2,19^2,31^2.
// (Dans l'ordre d'efficacité, d'après de nombreux tests empiriques.)
// Les images sont précalculées comme des vecteurs constants,
// ce qui permettra d'accéder rapidement à ces informations.
const vector<bool>
  image625= imagePmod(625), image169= imagePmod(169),
  image81 = imagePmod(81),  image841= imagePmod(841),
  image343= imagePmod(343), image121= imagePmod(121),
  image361= imagePmod(361), image961= imagePmod(961);

// La fonction  crible(c,d)  teste si le couple (c,d) passe les cribles.
// Si non, elle renvoie false et on peut supprimer (c,d) dans la suite.
bool crible( Integer c, Integer d )
{
  return image625[lePolynomeQ(c,d,625)] && image169[lePolynomeQ(c,d,169)]
      && image81 [lePolynomeQ(c,d,81) ] && image841[lePolynomeQ(c,d,841)]
      && image343[lePolynomeQ(c,d,343)] && image121[lePolynomeQ(c,d,121)]
      && image361[lePolynomeQ(c,d,361)] && image961[lePolynomeQ(c,d,961)];
}
```

Partie C

Arithmétique des entiers

All is well that ends well.
William Shakespeare

CHAPITRE VIII

Algorithme, correction, complexité

Objectifs

Un des objectifs de ce cours est de développer une notion de plus en plus précise de *complexité* de calcul. Pour ceci il faut préciser la *méthode* utilisée. Afin de choisir parmi les méthodes admises il faut d'abord une *spécification*. On est ainsi mené à regarder de plus près la notion *d'algorithme*. Ce chapitre discutera ce concept, en soulignant les trois points clés : *terminaison*, *correction*, *complexité*.

Sommaire

1. Qu'est-ce qu'un algorithme ?

Exemple 1.1. En utilisant l'arithmétique des entiers, on veut calculer le coefficient binomial $\binom{n}{k}$:

Spécification VIII.1 Calcul du coefficient binomial $\binom{n}{k}$

Entrée: deux entiers n et k
Sortie: le coefficient binomial $\binom{n}{k}$

En voici quatre méthodes candidates à résoudre ce problème.

Méthode VIII.2 Calcul du coefficient binomial $\binom{n}{k}$

si $k<0$ ou $k>n$ **alors retourner** 0 **sinon retourner** $\binom{n-1}{k-1}+\binom{n-1}{k}$

Cette méthode est carrément fausse. Voyez-vous pourquoi ?

Méthode VIII.3 Calcul du coefficient binomial $\binom{n}{k}$

si $k=0$ ou $k=n$ **alors retourner** 1 **sinon retourner** $\binom{n-1}{k-1}+\binom{n-1}{k}$

Cette méthode est fausse dans le sens qu'elle ne se termine pas toujours. (Expliquer pourquoi.) D'un autre coté, quand elle s'arrête, elle renvoie le résultat correct. (Le détailler.)

Méthode VIII.4 Calcul du coefficient binomial $\binom{n}{k}$

si $k<0$ ou $k>n$ **alors retourner** 0
si $k=0$ ou $k=n$ **alors retourner** 1
retourner $\binom{n-1}{k-1}+\binom{n-1}{k}$

Cette méthode se termine et donne le résultat correct. (Le montrer.)

Méthode VIII.5 Calcul du coefficient binomial $\binom{n}{k}$

si $k<0$ ou $k>n$ **alors retourner** 0
$b\leftarrow 1,\quad k\leftarrow \min(k,n-k)$
pour i **de** 1 **à** k **faire** $b\leftarrow \big(b\cdot(n-i+1)\big)/i$
retourner b

Cette méthode est également correcte, mais plus efficace que la précédente. (Pourquoi ?)

1.1. Algorithme = spécification + méthode. Comme on ne développera pas la théorie abstraite des algorithmes, nous nous contentons ici d'une définition pragmatique :

Définition 1.2. Un algorithme décrit un procédé, susceptible d'une réalisation mécanique, pour résoudre un problème donné. Il consiste en une *spécification* (ce qu'il doit faire) et une *méthode* (comment il le fait) :

- La **spécification** précise les données d'entrée avec les *préconditions* que l'on exige d'elles, ainsi que les données de sortie avec les *postconditions* que l'algorithme doit assurer. Autrement dit, les préconditions définissent les données auxquelles l'algorithme s'applique, alors que les postconditions résument le résultat auquel il doit aboutir.
- La **méthode** consiste en une suite finie d'instructions, dont chacune est soit une *instruction primitive* (directement exécutable sans explications plus détaillées) soit une *instruction complexe* (qui se réalise en faisant appel à un algorithme déjà défini). En particulier chaque instruction doit être exécutable de manière univoque, et ne doit pas laisser place à l'interprétation ou à l'intuition.

Exemple 1.3. Un algorithme décrit souvent le calcul d'une application $f : X \to Y$.

- La donnée d'entrée est d'un certain type, elle appartient à une classe X_0.
- La précondition précise le domaine de définition $X \subset X_0$, éventuellement restreint.
- La donnée de sortie est d'un certain type, elle appartient à une classe Y.
- La postcondition précise pour tout $x \in X$ la valeur cherchée $y = f(x)$.
- La méthode explicite les étapes du calcul en tout détail.

Dans l'exemple 1.1 les données d'entrée (n,k) appartiennent à l'ensemble $X_0 = \mathbb{Z}^2$, et on exige que la méthode s'applique à tout l'ensemble $X = X_0$. Le résultat est un entier et appartient donc à $Y = \mathbb{Z}$. La postcondition exige que $f(n,k)$ soit égal au coefficient binomial $\binom{n}{k}$.

Remarque 1.4 (première reformulation)**.** La spécification décrit le problème que l'on cherche à résoudre, en spécifiant les « conditions aux bords » : le point de départ est fixé par les données d'entrée, dont on suppose certaines préconditions ; le but est d'arriver à des données de sortie, pour lesquelles on doit garantir certaines postconditions. La méthode propose une solution à ce problème, c'est-à-dire un chemin allant du départ au but. (Le mot « méthode » est dérivé du grec ὁδός (*hodos*) signifiant « la voie ».) Bien sûr, pour un problème donné, plusieurs méthodes sont possibles, plusieurs chemins mènent au but.

Exemple 1.5. Il existe plusieurs solutions pour le problème de recherche d'un objet dans une liste (cf. chapitre V). Lorsque la précondition assure que les données sont ordonnées, la méthode peut en profiter. Ainsi l'algorithme suivant effectue correctement une recherche dichotomique sur une liste ordonnée. Travaille-t-il toujours correctement sur une liste non ordonnée ? (Donner un contre-exemple le cas échéant.)

Algorithme VIII.6 Recherche dichotomique

Entrée: un élément b et une liste ordonnée $A = (a_1, a_2, \dots, a_n)$.

Sortie: le premier indice k tel que $a_k = b$, ou bien *non trouvé* si b n'appartient pas à A.

$i \leftarrow 1, \quad j \leftarrow n$
tant que $i < j$ **faire** $m \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$: **si** $b \leq a_m$ **alors** $j \leftarrow m$ **sinon** $i \leftarrow m+1$
si $a_i = b$ **alors retourner** i **sinon retourner** *non trouvé*

Remarque 1.6 (deuxième reformulation)**.** Souvent on interprète la *spécification* comme un contrat : si l'instance appelante assure les préconditions, alors la méthode appelée se charge d'assurer les postconditions. Comment elle le fait est explicité dans la *méthode*. (La méthode ne fait pas partie du contrat ; dans le cas extrême, elle peut rester un secret professionnel de l'instance appelée.) Il ne faut pas s'étonner si la méthode échoue si les préconditions ne sont pas remplies. Ceci n'est pas la faute à la méthode : les données ne sont simplement pas dans son domaine d'application.

Exemple 1.7. Dans la pratique il est en général prudent de tester les préconditions dans les limites du raisonnable. Mais juridiquement parlant c'est la responsabilité de l'instance appelante et non de la méthode appelée. Souvent de tels tests sont coûteux (voire impossibles). Un exemple frappant est la recherche dichotomique : elle requiert une liste *ordonnée* pour effectuer une recherche efficace, mais il serait ridicule, en début de chaque recherche, de parcourir toute la liste pour vérifier l'ordre.

1.2. Preuve de correction. Parfois on voit un « algorithme » sans spécification. C'est une mauvaise habitude qui n'est justifiée que dans les rares cas où le contexte laisse sous-entendre la spécification sans équivoque. Voici la principale raison pour laquelle la spécification doit être explicitée :

Définition 1.8. On dit qu'un algorithme est *correct* si la méthode fait ce qu'exige la spécification. Plus précisément : un algorithme est correct si pour toute donnée d'entrée vérifiant la précondition, la méthode se termine et renvoie une donnée de sortie vérifiant la postcondition.

Attention. — La preuve qu'un algorithme est correct comporte toujours deux parties. *D'abord* il faut montrer que l'algorithme s'arrête pour toute donnée d'entrée valable ; la méthode aboutit alors à un résultat. *Ensuite* il faut montrer que ce résultat vérifie la postcondition. Autrement dit :

☞ *Avant de prouver que le résultat est correct, il faut montrer qu'il y en a un.* ✍

Question 1.9. Dans l'exemple 1.1 la méthode VIII.3 est fausse par rapport à la spécification VIII.1. Montrer que la même méthode est correcte dans l'algorithme VIII.7 grâce à une précondition plus restrictive. Obtient-on un résultat correct aussi pour des données d'entrée (n,k) avec $k<0$ ou $k>n$? Pourquoi cette question ne met pas en cause la correction de l'algorithme VIII.7 ?

Algorithme VIII.7 Calcul de coefficients binomiaux

Entrée: deux entiers n et k tels que $0 \leq k \leq n$

Sortie: l'entier b vérifiant $b = \binom{n}{k}$

si $k=0$ ou $k=n$ **alors retourner** 1 **sinon retourner** $\binom{n-1}{k-1}+\binom{n-1}{k}$

Exemple 1.10. Une preuve de correction est en général difficile et demande une analyse détaillée. Le projet IV, par exemple, avait pour but de prouver puis d'implémenter l'algorithme suivant :

Algorithme VIII.8 Calcul approché de π à une précision p

Entrée: deux nombres naturels $b \geq 2$ et $p \in [\![0, 10^6]\!]$

Sortie: une suite d'entiers $t_0, t_1, \ldots, t_p$ avec $0 \leq t_i < b$ pour tout $i = 1, \ldots, p$

Garanties: la valeur $t = \sum_{k=0}^{k=p} t_k b^{-k}$ vérifie $t < \pi < t + b^{-p}$.

$\tilde{p} \leftarrow p+50$, $n \leftarrow \tilde{p}+3$, $m \leftarrow 3+\lfloor \tilde{p} \log_2 b \rfloor$
pour i **de** 0 **à** m **faire** $s_i \leftarrow 2$ **fin pour**
pour j **de** 0 **à** n **faire**
  $t_j \leftarrow s_0$, $s_0 \leftarrow 0$, $s_m \leftarrow b s_m$
  pour i **de** m **à** 1 **faire** $q \leftarrow 2i+1$, $s_{i-1} \leftarrow b s_{i-1} + i\,(s_i \operatorname{div} q)$, $s_i \leftarrow s_i \bmod q$ **fin pour**
fin pour
pour j **de** n **à** 1 **faire** $t_{i-1} \leftarrow t_{i-1} + (t_i \operatorname{div} b)$, $t_i \leftarrow (t_i \bmod b)$ **fin pour**
retourner $(t_0, t_1, \ldots, t_p)$

Vous pouvez constater qu'un tel algorithme « tombé du ciel » est peu compréhensible. Heureusement l'approche du chapitre IV fut plus soigneuse — si vous voulez, vous pouvez résumer la preuve de correction. Sans être précédé par un tel développement détaillé, l'algorithme ci-dessus paraît sans doute assez cryptique. (Il en est de même pour le programme I.21.)

☞ *Idéalement on construit un algorithme parallèlement à sa preuve de correction.* ✍

Remarque 1.11. Une fonction en C++ est une méthode : elle reçoit des données d'entrée (ses paramètres) et en déduit des données de sortie (son résultat). Dans ce but le C++, comme tout langage de programmation, exige une description très détaillée de la méthode, formulée à l'aide des instructions primitives.

Soulignons cependant que le C++ ne prévoit pas le concept de spécification. C'est bien dommage, car sans spécification la question de correction reste vide : on ne sait même pas formuler l'énoncé à démontrer.

Le mieux que l'on puisse faire en C++ est de détailler la spécification sous forme de commentaire, ou dans la documentation annexe, destiné non au compilateur mais au lecteur humain. Ainsi on peut énoncé puis démontrer la correction d'une fonction en toute rigueur. Cette justification reste bien entendu à l'extérieur du C++, et malheureusement elle est trop souvent négligée.

Remarque 1.12. Il existe des langages de programmation, comme par exemple le langage *Coq* (voir le site web `coq.inria.fr`), qui intègrent spécifications et preuves dans le code source. Pour cela il faut évidemment un formalisme assez élaboré ; la programmation n'est plus seulement le développement d'une méthode, mais comprend aussi le développement d'une preuve formelle.

Le principal avantage est le suivant : muni du code source avec preuve intégrée, *le compilateur peut vérifier la correction du programme !* Soulignons que les deux tâches sont nettement séparées : le programmeur *développe* la preuve, le compilateur la *vérifie* : il serait irréaliste de croire que le compilateur puisse inventer, à lui tout seul, une preuve de correction. À nos jours cette approche, dite *vérification automatique*, ne reste plus théorique. L'investissement supplémentaire s'amortit dans des applications sensibles qui exigent un très haut niveau de sécurité, l'idéal étant des *logiciels certifiés zéro défaut*.

1.3. Le problème de terminaison. Comme nous avons souligné plus haut, pour tout algorithme la preuve de correction commence par une preuve de terminaison. Suivant la méthode en question ceci peut être plus ou moins difficile. Évidemment, si la méthode n'utilise ni de boucles ni d'appels récursifs, alors l'algorithme se termine toujours. (Le justifier.) Ce cas simpliste est rare ; en général il faut trouver un argument plus profond, typiquement une preuve par une récurrence bien choisie.

Exemple 1.13. Pour un exemple peu trivial, regardons la *fonction d'Ackermann*. Elle est chère aux informaticiens, principalement parce qu'elle croît à une vitesse hallucinante. On pourrait prendre l'algorithme VIII.9 ci-dessous comme sa définition. Seul problème : montrer qu'il se termine.

Algorithme VIII.9 Calcul de la fonction d'Ackermann $a\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$

Entrée: deux nombres naturels n et k

Sortie: la valeur $a(n,k)$ de la fonction d'Ackermann

si $n=0$ **alors retourner** $k+1$
si $k=0$ **alors retourner** $a(n-1,1)$
retourner $a(n-1,a(n,k-1))$

Exercice 1.14. La terminaison est une propriété qu'il faut démontrer, et que l'on ne peut pas déterminer de manière expérimentale. Pour vous en convaincre, vous êtes vivement invités à faire l'expérience suivante : écrire une fonction `ackermann` en C++ et faire calculer quelques petites valeurs (disons $0\leq n\leq 4$ et $0\leq k\leq 10$). C'est un bon exercice pratique, car le résultat est peu prévisible…

Pendant que l'ordinateur rame, vous pouvez déterminer « à la main » les valeurs suivantes : par définition on a $a(0,k)=k+1$. Montrer par récurrence que $a(1,k)=k+2$, et $a(2,k)=2k+3$, puis $a(3,k)=2^{k+3}-3$. Finalement, déterminer $a(4,k)$. Même sans hâte, vous finirez avant que l'ordinateur ne calcule la valeur $a(4,2)$. Félicitations, vous calculez plus vite que votre ordinateur !

Exemple 1.15. On reprend l'exercice I.8.4 sur le « problème $3x+1$ » qui donne lieu à l'algorithme VIII.10 ci-dessous. Le lecteur en quête de célébrité peut se lancer dans la preuve de terminaison.

Algorithme VIII.10 Le problème $3x+1$

Entrée: un nombre naturel $x\geq 1$

Sortie: un nombre naturel $l\geq 0$ qui compte les itérations

$l\leftarrow 0$
tant que $x>1$ **faire**
  $l\leftarrow l+1$
  si x est pair **alors** $x\leftarrow x/2$ **sinon** $x\leftarrow 3x+1$
fin tant que
retourner l

2. La notion de complexité et le fameux « grand O »

Pour la plupart des problèmes il existe un grand nombre d'algorithmes possibles. Comment en choisir le meilleur ? Quels sont les différents degrés de complexité que l'on peut rencontrer ?

> Throughout all modern logic, the only thing that is important is whether a result can be achieved in a finite number of elementary steps or not. (...) In the case of an automaton the thing which matters is not only whether it can reach a certain result in a finite number of steps at all, but also how many such steps are needed. (John von Neumann, Hixon Symposium lecture, 1948)

Ces questions sont souvent d'une grande importance pratique, et la théorie associée constitue un domaine d'étude très poussée de l'informatique. (Pour un développement plus détaillé voir Graham-Knuth-Patashnik [**17**], chapitre 9, et l'appendice de Gathen-Gerhard [**11**] pour un tour d'horizon.)

2.1. Le coût d'un algorithme. Étant donné un algorithme et une donnée d'entrée x, on peut mesurer le *coût* $c(x)$, aussi appelé la *complexité*, de la méthode en question. Ce qui nous intéresse le plus souvent est la *complexité temporelle* : l'exécution de chaque instruction primitive nécessite un certain temps, et le temps d'exécution de l'algorithme est le temps cumulatif de toutes les instructions exécutées l'une après l'autre. (Pour simplifier on suppose parfois que les instructions primitives ont toutes le même coût, ainsi défini comme *coût unitaire*.) Très souvent le coût cumulatif est proportionnel au nombre d'itérations, et on se contente de déterminer ce dernier. L'analyse de complexité consiste ainsi à étudier la fonction $c\colon x \to c(x)$ qui associe le coût $c(x)$ à chaque entrée possible x soumise à l'algorithme.

Exemple 2.1. L'addition de deux nombres x et y à n chiffres a un coût αn, avec une constante $\alpha > 0$. Leur multiplication avec la méthode scolaire a un coût βn^2, avec une constante $\beta > 0$. (Rappeler pourquoi.) Même sans connaître les détails de l'implémentation, ceci veut dire que la méthode de la multiplication est plus coûteuse, pour n grand, que la méthode de l'addition.

Exemple 2.2. Pour trier une liste $x = (x_1, \dots, x_n)$ le tri par sélection (algorithme V.3) effectue $\frac{1}{2}n(n-1)$ itérations et son coût est $c_1(x) = \frac{1}{2}\alpha_1 n(n-1) + \beta_1(n-1)$. Le tri fusion (algorithme V.6) effectue des appels récursifs ; son coût exact $c_2(x)$ est encadré par le théorème V.3.5 : $\alpha_2 n\lfloor \log_2 n\rfloor + \beta_2(n-1) \le c_2(x) \le \alpha_2 n\lceil \log_2 n\rceil + \beta_2(n-1)$. Après avoir mesuré les constantes α_1, β_1 et α_2, β_2 sur une implémentation bien testée et optimisée, vous pouvez choisir la meilleure des deux méthodes en fonction de la taille n. Même sans connaître ces détails de l'implémentation, on voit que le tri élémentaire est compétitif seulement pour n assez petit. (Le détailler.)

2.2. Le coût moyen, dans le pire et dans le meilleur des cas. L'analyse fine que l'on vient de décrire n'est souvent pas praticable. Dans un tel cas on préfère des informations plus globales, en regroupant les données d'une même « taille ». Typiquement les données d'entrée x admettent une notion de taille naturelle, que nous noterons $|x|$: pour une liste ou un vecteur c'est sa longueur, pour un nombre entier c'est la longueur de son développement binaire, pour un polynôme c'est son degré, etc.

Pour une taille n donnée, on peut alors regarder l'ensemble $X_n = \{x \mid |x| = n\}$ de toutes les données de taille n. On définit le coût moyen par

$$\overline{c}(n) := \frac{1}{|X_n|} \sum_{x \in X_n} c(x).$$

Ici on sous-entend que l'ensemble X_n est fini et que toutes les données sont équiprobables. (D'autres distributions de probabilités sont parfois mieux adaptées, selon le contexte.) Pour être prudent, on regarde également le coût dans le pire des cas :

$$\hat{c}(n) = \max_{x \in X_n} c(x).$$

Ce point de vue est indispensable si l'on veut *garantir* une certaine performance dans *tous* les cas, non seulement en moyenne. Il est parfois instructif de regarder le coût dans le meilleur des cas :

$$\check{c}(n) = \min_{x \in X_n} c(x).$$

Exemple 2.3. Le coût du tri rapide (algorithme V.7) dépend fortement de la liste à trier et non seulement de la taille n. Le coût dans le meilleur des cas est $\check{c}(n) = \alpha n \log_2 n$, le coût dans le pire des cas est $\hat{c}(n) = \frac{\alpha}{2}n(n-1)$, mais le coût en moyenne est $\overline{c}(n) = 2\alpha n \ln n$ seulement.

2.3. Complexité asymptotique. Souvent c'est le principe de l'algorithme que l'on veut juger, et non les détails de l'implémentation. On veut donc abstraire de tous les facteurs constants ; de toute façon, ils changeront d'une implémentation à une autre, et d'une machine à une autre. Pour ne retenir que l'essentiel, on regroupe les fonctions suivant leur « ordre de grandeur » :

Définition 2.4. Soit $g\colon \mathbb{N} \to \mathbb{R}_+$ une fonction positive. On définit alors les classes suivantes :

(1) $O(g) = \{\, f\colon \mathbb{N} \to \mathbb{R}_+ \mid \exists C > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq Cg(n) \,\}$
Ce sont les fonctions qui croissent au plus aussi vite que g (finalement *majorées* par Cg).

(2) $\Omega(g) = \{\, f\colon \mathbb{N} \to \mathbb{R}_+ \mid \exists c > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \geq cg(n) \,\}$
Ce sont les fonctions qui croissent au moins aussi vite que g (finalement *minorées* par cg).

(3) $\Theta(g) = O(g) \cap \Omega(g) = \{\, f\colon \mathbb{N} \to \mathbb{R}_+ \mid \exists c, C > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : cg(n) \leq f(n) \leq Cg(n) \,\}$
Ce sont les fonctions qui croissent aussi vite que g (qui ont *même ordre* de grandeur).

(4) $o(g) = \{\, f\colon \mathbb{N} \to \mathbb{R}_+ \mid f(n)/g(n) \to 0 \text{ pour } n \to \infty \,\}$
Ce sont les fonctions qui croissent moins vite que g (qui sont *négligeables* devant g).

(5) Finalement, si $f(n)/g(n) \to 1$ pour $n \to \infty$, on dit que f et g sont équivalentes, noté $f \sim g$.

Ces notations sont traditionnellement attribuées à Landau. La tradition veut aussi que l'on écrive $f = O(g)$ à la place de $f \in O(g)$, abus de langage, hélas, auquel il faut s'habituer.

Exemple 2.5. On voit par exemple que $O(1)$ est l'ensemble des fonctions bornées, et $o(1)$ est l'ensemble des fonctions qui tendent vers 0 pour $n \to \infty$.

Exemple 2.6. Dans le projet I on a utilisé k boucles imbriquées pour parcourir tous les k-uplets $x \in \mathbb{Z}^k$ vérifiant $1 \leq x_1 \leq \cdots \leq x_k \leq n$. Le nombre $f(n)$ de tels k-uplets mesure le coût temporel de cet algorithme. On voit facilement que $f \in O(n^k)$, même $f \in \Theta(n^k)$, plus précisément on a l'équivalence $f \sim \frac{1}{k!} n^k$. Bien sûr la description la plus précise est de dire $f(n) = \binom{n+k-1}{k} = \frac{1}{k!} n(n+1)\cdots(n+k-1)$.

Exemple 2.7. Le tri fusion a un coût $f(n) = \alpha n \ln n + \beta n$ avec $\alpha > 0$. Cette fonction appartient à $O(n \ln n)$, même $\Theta(n \ln n)$, plus précisément on a l'équivalence $f \sim \alpha n \ln n$.

Exemple 2.8. La fonction $f(n) = n!$ est dans $O(n^n)$, même dans $o(n^n)$. Plus précisément, selon la formule de Stirling, f est équivalente à $n^n \cdot e^{-n} \cdot \sqrt{2\pi n}$. Chercher dans un cours d'analyse l'énoncé précis sous forme d'encadrement ; en quoi est-il plus précis qu'une simple équivalence asymptotique ?

Exercice/M 2.9. Si $f \in \Theta(g)$ a-t-on $g \in \Theta(f)$? puis $O(f) = O(g)$? et $o(f) = o(g)$?

Exercice/M 2.10. Montrer que $\ln n \in o(n^\varepsilon)$ pour tout $\varepsilon > 0$. Par conséquent $n^\alpha \ln^\beta n \in O(n^{\alpha+\varepsilon})$ pour tout $\alpha \geq 0$ et $\varepsilon > 0$. En négligeant le terme logarithmique, on dit ainsi que $n^\alpha \ln^\beta n$ est *presque* d'ordre n^α. Plus généralement on peut définir la classe $O^+(g) = \cap_{\varepsilon>0} O(g(n)n^\varepsilon)$. Ce sont les fonction de croissance *presque* d'ordre de g. Si $f_1 \in O^+(g_1)$ et $f_2 \in O^+(g_2)$, a-t-on $f_1 f_2 \in O^+(g_1 g_2)$?

2.4. Les principales classes de complexité. Dans la pratique on a souvent affaire à des fonctions de complexité dont le comportement asymptotique est un des suivants :

Complexité constante, $O(1)$***:*** Les instructions primitives en C++ sont de coût constant, par exemple tous les calculs effectués sur les variables de type `int` (dont la taille est fixée). De même pour accéder à l'élément `v[i]` d'un vecteur `v`. *Exemple :* déterminer le reste modulo 2 d'un entier en numération décimale.

Complexité logarithmique, $O(\ln n)$***:*** Un programme de complexité logarithmique devient seulement très légèrement plus lent quand n croît. Chaque fois que n est doublé, le coût n'augmente que par addition d'une constante. *Exemple :* recherche dichotomique.

Complexité linéaire, $O(n)$***:*** C'est le mieux que l'on puisse espérer pour un algorithme qui doit traiter n données une par une. Chaque fois que n est doublé, le coût double lui aussi. *Exemples :* parcourir une liste de longueur n pour trouver le maximum ou le minimum ; addition de deux entiers de longueur n en numération décimale ; déterminer le reste modulo 3 d'un nombre naturel en numération décimale. (Le détailler.)

Complexité presque linéaire, $O(n \ln n)$***:*** C'est la complexité typique pour les algorithmes de type « diviser pour régner ». Chaque fois que n est doublé, le coût est un peu plus que doublé (mais guère plus). *Exemple :* le tri fusion ou le tri rapide (dans le cas générique).

Complexité sous-quadratique, $O(n^\alpha)$ ***avec*** $\alpha < 2$***:*** La multiplication de deux entiers de longueur n en numération décimale nécessite un temps $O(n^{1,585})$ avec la méthode de Karatsuba (voir chap. II, §3). Cette complexité se situe donc entre la complexité linéaire de l'addition et la complexité quadratique de la multiplication scolaire.

Complexité quadratique, $O(n^2)$***:*** Les complexités quadratiques sont typiques pour les algorithmes traitant tous les couples parmi n données (éventuellement par deux boucles imbriquées). *Exemples :* la multiplication scolaire de deux entiers de longueur n en numération décimale (la rappeler) ; le tri élémentaire (par sélection, transposition, ou insertion).

Complexité polynomiale, $O(n^k)$ ***avec*** $k > 1$***:*** Typiquement un algorithme qui traite tous les k-uplets parmi n données est de complexité $O(n^k)$. De tels algorithmes ne sont utilisables que pour des problèmes relativement petits. *Exemple :* La recherche exhaustive des solutions de $a^4 + b^4 + c^4 = d^4$ effectuée dans le projet I est de complexité $O(n^4)$, puis on l'a réduite à $O(n^3)$. Dans le projet V elle sera réduite à $O^+(n^2)$, ce qui permettra finalement de résoudre le problème.

Complexité exponentielle, $O(e^{\alpha n})$ ***avec*** $\alpha > 0$***, voire*** $O(e^{p(n)})$ ***avec un polynôme*** p***:*** Un algorithme de complexité exponentielle est pratiquement inutilisable, sauf peut-être pour les problèmes très petits. *Exemple :* parcourir tous les 2^n sous-ensembles $S \subset X$ d'un ensemble X de taille n, ou parcourir toutes les $n!$ permutations de X.

Complexité sur-exponentielle: Il existe aussi des fonctions de croissance sur-exponentielle, comme $\exp(\exp(n))$. Des algorithmes d'une telle complexité n'ont pas d'intérêt pratique ; ils peuvent néanmoins être très importants au niveau théorique, par exemple pour montrer *l'existence* d'une solution, aussi inefficace qu'elle soit.

2.5. À la recherche du temps perdu. Dans une application sérieuse il est impensable d'utiliser un algorithme sans aucune connaissance de sa complexité. Les notions décrites ci-dessus nous aideront à fournir des indications, en particulier l'idée de classer la complexité en quelques catégories est utile pour expliquer le comportement asymptotique. Ceci est souvent un bon guide pour le choix d'algorithme.

Mais soyons clairs : la complexité asymptotique ne dit absolument rien sur le temps d'exécution dans un cas concret. Autant doit-on réfléchir longuement avant d'utiliser un algorithme cubique au lieu d'un algorithme quadratique, autant doit-on se méfier de suivre aveuglement les résultats de complexité exprimés en notation grand O. L'analyse de complexité asymptotique n'est qu'une première approximation dans un développement de plus en plus fin.

Exemple 2.11. Comparons deux méthodes, de coût n^2 heures et n^3 secondes, respectivement. Certes, elles sont d'ordre $\Theta(n^2)$ et $\Theta(n^3)$, mais regardons les *constantes cachées* : la première méthode (de complexité quadratique) n'est avantageuse que pour $n \geq 3600$, elle s'amortit donc à partir d'un temps d'exécution de 1500 années environ. Pour des applications avec $n < 3600$ on choisira la deuxième méthode (de complexité cubique). Cet exemple banal sert d'avertissement : vous avez tout intérêt à choisir la meilleure méthode dans le cas concret, disons $n = 1000$, ce qui peut ou non correspondre au comportement asymptotique. Ainsi on choisira l'algorithme en fonction de la donnée, ce que l'on appelle un *algorithme hybride*. Le bon choix des intervalles d'application (*cross-over points* en anglais) est une technique importante d'optimisation.

Exercices supplémentaires

Exercice/M 2.12. L'énoncé $O(f+g) = O(\max(f,g))$ est-il justifié ? Plus précisément : soient $f, g\colon \mathbb{N} \to \mathbb{R}_+$ deux fonctions positives et soit $h\colon \mathbb{N} \to \mathbb{R}_+$ définie par $h(n) = \max\{f(n), g(n)\}$. Comparer l'ordre de grandeur de $f+g$ et de h : a-t-on $f+g \in \Theta(h)$? puis $O(f+g) \subset O(h)$? et $O(h) \subset O(f+g)$?

Exercice/M 2.13. Dans la pratique on a souvent affaire à des boucles imbriquées ou des appels de sous-algorithmes. Dans de tels cas, la majoration suivante peut être utile. Si $f_1 \in O(g_1)$ et $f_2 \in O(g_2)$, montrer que $f_1 f_2 \in O(g_1 g_2)$. Ceci peut se résumer comme $O(f) \cdot O(g) \subset O(fg)$. Est-ce que l'énoncé $O(f) \cdot O(g) = O(fg)$ est aussi correct ?

Exercice/M 2.14. Montrer que $f \sim g$ est une relation d'équivalence. Il en est de même pour la relation $f \approx g$ définie par $f \in \Theta(g)$. Montrer que $f \sim g$ implique $f \approx g$, mais la réciproque est fausse. Montrer que la relation $f \preccurlyeq g$ définie par $f \in O(g)$ est un ordre partiel. Vérifier que $f \preccurlyeq g$ et $g \preccurlyeq f$ implique $f \approx g$. Par contre, ce n'est pas un ordre total : trouver deux fonctions croissantes $f, g \colon \mathbb{N} \to \mathbb{N}$ de sorte que ni $f \in O(g)$ ni $g \in O(f)$.

Quelques réponses et indications

[Exemple 1.1, calcul du coefficient binomial] La méthode VIII.2 se termine et elle renvoie toujours 0, ce qui n'est pas la valeur cherchée. La méthode VIII.3, se termine pour toutes les entrées n, k vérifiant $0 \leq k \leq n$ en renvoyant la valeur correcte. Pour $k < 0$ ou $k > n$, par contre, elle ne se termine pas. La méthode VIII.4 se termine toujours : déjà pour $k < 0$ ou $k > n$ elle renvoie 0, comme il faut. Pour $0 \leq k \leq n$ on peut montrer la terminaison, puis la correction, par une récurrence sur n. Quant à la méthode VIII.5, sa correction et sa performance, voir la discussion du chapitre II, §1.3.

[Exercice 2.14, ordres inéquivalents] On pourrait regarder les deux fonctions $f, g \colon \mathbb{N} \to \mathbb{N}$ définies par $f(n) = n!$ et $g(n) = (n-1)!$ si n est pair, puis $f(n) = (n-1)!$ et $g(n) = n!$ si n est impair.

Passent les jours et passent les semaines
Ni temps passé ni les amours reviennent
Vienne la nuit sonne l'heure
Les jours s'en vont je demeure
Guillaume Apollinaire, *Le pont Mirabeau*

PROJET VIII

Puissance dichotomique

Objectifs

Ce projet discute le problème de calculer efficacement la *puissance modulaire* x^n modulo m pour $x, n, m \in \mathbb{N}$. Il s'agit d'un outil omniprésent dans l'algorithmique des entiers, et quand n et m sont grands il est indispensable de comprendre les pièges et les astuces.

Sommaire

1. Le critère de Pépin

Commençons par une application typique de la puissance dichotomique :

Exemple 1.1 (nombres de Fermat)**.** Pour $k \in \mathbb{N}$ on appelle $F_k := 2^{2^k} + 1$ le $k$ième *nombre de Fermat*. Les cinq premiers termes sont $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$, $F_4 = 65537$, et Fermat constata qu'il s'agit de cinq nombres premiers. Il conjectura même que F_k serait premier pour tout $k \in \mathbb{N}$, mais ses tests empiriques n'allaient pas plus loin que $k = 4$. Le problème est évidemment que les nombres F_k croissent rapidement : comme F_{k+1} est à peu près le carré de F_k, le nombre de chiffres double en augmentant le rang. Ainsi la nature de $F_5 = 4294967297$ est déjà moins évidente. Plus ces nombres sont grands, plus on a besoin de méthodes efficaces. Comment tester efficacement la primalité de ces nombres ?

On admettra ici le critère de Pépin, que l'on établira un peu plus tard dans le projet X :

Lemme 1.2. *Pour $k \geq 1$ le nombre F_k est premier si et seulement si* $3^{\frac{F_k-1}{2}} \equiv -1 \pmod{F_k}$.

On souhaite donc calculer la puissance modulaire pour $x = 3$ et $n = 2^{2^k-1}$ et $m = 2n+1$. On va soumettre chacune des méthodes ci-dessous à la question cruciale : jusqu'à quelle valeur de k pouvons-nous effectuer le test de Pépin ? Lesquels des nombres F_k sont premiers, lesquels composés ?

On verra que parmi quatre méthodes qui viennent à l'esprit, une seule arrivera au bout… Selon vos préférences, l'exploration de fausses pistes vous semblera ou pédagogiquement réaliste ou artificiellement pénible. En tout cas elle servira, je l'espère, à vous vacciner durablement contre la programmation naïve.

2. Puissance linéaire

2.1. L'algorithme. La puissance linéaire est la première méthode qui vient à l'esprit :

Algorithme VIII.11 Puissance linéaire

Entrée: la base x et l'exposant $n \in \mathbb{N}$
Sortie: la puissance $p = x^n$

$p \leftarrow 1$	// Après initialisation on a $px^n = x^n$.
tant que $n > 0$ **faire** $p \leftarrow p * x, \quad n \leftarrow n-1$	// Le produit px^n ne change pas de valeur.
retourner p	// Ici $n = 0$ et on renvoie $p = px^n$ comme souhaité.

2.2. L'implémentation. Regardons deux implémentations de la puissance linéaire :

Exercice/P 2.1. Écrire une fonction `puissance_lineaire(x,n)` qui calcule x^n dans $\mathbb{Z}$. Pour chacun des paramètres choisir le mode de passage qui convient le mieux. Jusqu'à quelle valeur de k pouvez-vous effectuer le test de Pépin ? Quel en est le résultat ?

Exercice/P 2.2. Ajouter une fonction `puissance_lineaire(x,n,m)` qui calcule x^n modulo m. (Choisir les modes de passage.) Pourquoi a-t-on intérêt à réduire systématiquement modulo m ? Jusqu'à quelle valeur de k pouvez-vous effectuer le test de Pépin ? Quel en est le résultat ?

Exercice/P 2.3 (ici « P » non comme « programmation » mais comme « poésie »)**.** Durant les longs mois d'hiver que vous attendez une réponse de votre ordinateur, la poésie pourra vous offrir de réconfort. Récitez le poème d'Appolinaire ci-dessus puis ajoutez quelques vers. Voici les contributions de Guenaëlle De Julis et Cédric Frambourg (Licence de Mathématiques à l'Institut Fourier en 2005) :

— ∗ —

Le temps se prélasse
Les minutes se tassent
Programme infinissable
Attente détestable
Devant mon ordinateur
Toujours je demeure

— ∗ —

Stoïque devant
mon écran blafard
De rares souvenirs
montent à ma mémoire
Fixant d'un œil hagard
un tas de pixels noires

— ∗ —

Contemplons aussi le poème de Steve Planchin, Licence de Mathématiques à l'Institut Fourier en 2007 :

Les nombres restent figés sur un sombre écran
Tels le pauvre reflet sur le miroir navrant
De notre esprit embrumé empli du néant
D'un langoureux calcul bloqué entre deux temps

Question 2.4. Les expressions `puissance_lineaire(x,n)%m` et `puissance_lineaire(x,n,m)` rendent-elles le même résultat ? Laquelle est préférable et pourquoi ? Êtes-vous satisfaits du gain de performance ?

3. Puissance dichotomique

3.1. L'algorithme. Il est particulièrement facile de calculer les puissances $x^1, x^2, x^4, x^8, x^{16}, \ldots$: on pose $x_0 := x$ et calcule $x_i := x_{i-1} * x_{i-1}$ par récurrence. On obtient ainsi les valeurs $(x_0, x_1, \ldots, x_k)$ avec $x_i = x^{2^i}$ en effectuant k multiplications seulement !

Quant à une puissance x^n avec $n \geq 1$ quelconque, on peut écrire l'exposant n en base 2, c'est-à-dire $n = \sum_{i=0}^{i=k} n_i 2^i$ avec $n_i \in \{0,1\}$. Nous pouvons supposer que $n_k = 1$, ce qui rend cette écriture unique ; en particulier on a $2^k \leq n < 2^{k+1}$, donc $k = \lfloor \log_2 n \rfloor$. En supprimant les termes nuls on obtient $n = \sum_{i \in I} 2^i$, où l'ensemble I consiste des indices i avec $n_i = 1$. Ceci permet d'écrire

$$x^n = x^{\sum_{i \in I} 2^i} = \prod_{i \in I} x^{2^i} = \prod_{i \in I} x_i.$$

Ainsi on calcule x^n avec $k + |I| - 1$ multiplications seulement ! L'algorithme VIII.12 en donne une variante prête-à-programmer, qui se passe de la liste $(x_0, x_1, \ldots, x_k)$ et n'utilise que les trois variables x, n, p :

Algorithme VIII.12 Puissance dichotomique

Entrée: la base x et l'exposant $n \in \mathbb{N}$
Sortie: la puissance $p = x^n$

$p \leftarrow 1$	// Après initialisation on a $px^n = x^n$.
tant que $n > 0$ **faire**	
tant que n est pair **faire** $x \leftarrow x * x, \quad n \leftarrow n/2$	// Le produit px^n ne change pas de valeur.
$p \leftarrow p * x, \quad n \leftarrow n - 1$	// Le produit px^n ne change pas de valeur.
fin tant que	
retourner p	// Ici $n = 0$ et on renvoie $p = px^n$ comme souhaité.

Exercice/M 3.1. Montrer que l'algorithme VIII.12 se termine, puis prouver sa correction en détaillant l'invariant indiqué dans les commentaires. *Indication.* — Il sera utile d'introduire un indice i pour indiquer les valeurs p_i, x_i, n_i après la $i$ème itération, puis établir une récurrence sur $i = 0, 1, 2, \ldots$.

Exercice/M 3.2. Déterminer le nombre exact de multiplications effectuées par l'algorithme VIII.12, noté $\mu(n)$. Quel est son ordre de grandeur ? a-t-on $\mu \in \Theta(\log_2 n)$? voire $\mu \sim \alpha \log_2 n$?

Remarque 3.3. À première vue la puissance dichotomique semble miraculeuse. Après s'y être habitué, on pourrait soupçonner que cette méthode soit toujours optimale. Pourtant, pour certains exposants n on peut faire mieux, le plus petit exemple étant $n = 15$: la méthode dichotomique permet de calculer x^{15} par les 6 étapes suivantes : $(x, x^2, x^4, x^8, x^{12}, x^{14}, x^{15})$. On peut faire avec 5 multiplications seulement : $(x, x^2, x^3, x^5, x^{10}, x^{15})$. Le gain est assez faible, mais une fonction optimisée peut être justifiée si vous utilisez des puissances x^{15} très fréquemment ou si les multiplications sont très coûteuses. Pour l'usage général la méthode dichotomique est largement suffisante. Pour une discussion détaillée, consultez Knuth [**8**], §4.6.3.

3.2. L'implémentation. Pour vous convaincre de l'utilité — ou plutôt de la *nécessité* — de la puissance dichotomique, implémentez-la puis comparez sa performance à la puissance linéaire :

Exercice/P 3.4. Implémenter une fonction `puissance(x,n)` qui profite de la puissance dichotomique. Jusqu'à quelle valeur de k pouvez-vous effectuer le test de Pépin ? Quel en est le résultat ?

Exercice/P 3.5. Ajouter une fonction `puissance(x,n,m)` qui calcule x^n modulo m. Pourquoi a-t-on intérêt à réduire systématiquement modulo m durant le calcul ? Jusqu'à quelle valeur de k pouvez-vous finalement effectuer le test de Pépin ? Quel en est le résultat ?

Question 3.6. Les deux expressions `puissance(x,n)%m` et `puissance(x,n,m)` rendent-elles le même résultat ? Laquelle est préférable et pourquoi ? Êtes-vous satisfaits du gain de performance ?

Exercice/P 3.7. Pour comparaison, compiler votre programme final en utilisant la classe `Naturel` pour les grands entiers au lieu de la bibliothèque GMP. (C'est notre classe faite maison discutée au chapitre II.) Les résultats coïncident-ils ? Qu'observez-vous quant à la performance ? Comment expliquer ce phénomène ?

4. Analyse de complexité

Les exercices suivants illustrent dans quelle généralité on peut appliquer les deux algorithmes de puissance. De plus, ils analysent la complexité afin d'expliquer la performance observée.

☞ Pour simplifier vous pouvez supposer que le coût de la multiplication et de la division euclidienne de deux entiers de longueur $\leq \ell$ est (presque) linéaire en ℓ, ce qui est réaliste avec une implémentation efficace. (Revoir le chapitre II pour les opérations arithmétiques élémentaires.)

4.1. Puissance linéaire. Nos implémentations de puissance placent le calcul dans l'anneau $\mathbb{Z}$ ou $\mathbb{Z}_m$. Comme seule la multiplication est utilisée, on peut regarder la situation générale d'un ensemble M muni d'une multiplication $*\colon M \times M \to M$. Pour simplifier la notation, nous supposons que la multiplication admet un élément neutre à gauche, noté 1. On définit les puissances x^n d'un élément $x \in M$ de manière récursive par $x^0 := 1$ et $x^{n+1} := x^n * x$. Ceci correspond au parenthésage $x^n = (\cdots((x * x) * x)\cdots * x)$, dit parenthésage à gauche. (Sans hypothèse sur l'associativité il faut faire un choix ou l'autre.)

Exercice/M 4.1. L'algorithme VIII.11 avec initialisation $p \leftarrow$ (élément neutre) calcule-t-il correctement la puissance x^n dans $(M, *)$? Le résultat reste-t-il le même quand on remplace $p \leftarrow p * x$ par $p \leftarrow x * p$?

Exercice/M 4.2. Afin d'estimer le coût on comptera le nombre de multiplications : dans l'algorithme VIII.11 il en faut n. On sous-entend que le coût d'une multiplication est constant, ce qui est parfaitement justifié quand M est fini, comme $\mathbb{Z}_m$ par exemple. Plus précisément, en supposant que m est de longueur ℓ, estimer le coût du calcul de x^n dans $\mathbb{Z}_m$ en fonction de n et ℓ.

Exercice/M 4.3. Quant aux calculs dans $\mathbb{Z}$, les nombres peuvent devenir arbitrairement grands. Une telle situation nécessite une analyse plus fine : étant donné $x \in \mathbb{Z}$ ayant ℓ chiffres on s'attend à un résultat x^n ayant $n\ell$ chiffres environ. Estimer le coût de ce calcul en fonction de n et ℓ. Justifier ainsi que les calculs de `puissance_lineaire(x,n)%m` et `puissance_lineaire(x,n,m)` ne sont pas de même complexité. Ce résultat théorique correspond-il à vos expériences pratiques ?

4.2. Puissance dichotomique. La puissance linéaire effectue n multiplications pour calculer x^n. On peut faire beaucoup mieux, sous condition que la multiplication soit *associative*.

Un *monoïde* $(M,*)$ est un ensemble M muni d'une multiplication $*: M\times M\to M$ qui est associative et admet un élément neutre. Par exemple $(\mathbb{N},+)$ est un monoïde, ainsi que $(\mathbb{N},\cdot)$ et $(\mathbb{Z},\cdot)$ et plus généralement la structure multiplicative $(A,\cdot)$ d'un anneau A quelconque, commutatif ou non, par exemple l'anneau des matrices $\mathrm{Mat}(d\times d;A)$. C'est cette hypothèse d'associativité qui fait marcher notre affaire :

Proposition 4.4. *Dans un monoïde M on a $x^{m+n}=x^m*x^n$ pour tout $x\in M$ et $m,n\in\mathbb{N}$. Autrement dit, l'application $\mathbb{N}\to M$, $n\mapsto x^n$ est un homomorphisme entre les monoïdes $(\mathbb{N},+)$ et $(M,*)$. En particulier les puissances x^n d'un élément x commutent entre elles.* □

Exercice/M 4.5. Montrer soigneusement la proposition précédente. En déduire que l'algorithme VIII.12 reste correct dans un monoïde quelconque. En particulier on conclut :

Corollaire 4.6. *Dans un monoïde on peut calculer x^n avec au plus $2\lfloor\log_2 n\rfloor$ multiplications.* □

Exercice/M 4.7. Afin d'estimer le coût d'une puissance dichotomique dans $\mathbb{Z}_m$, supposons que m est de longueur ℓ. Exprimer le coût du calcul de x^n dans $\mathbb{Z}_m$ en fonction de n et ℓ.

Exercice/M 4.8. Étant donné $x\in\mathbb{Z}$ ayant ℓ chiffres, estimer le coût du calcul de x^n en fonction de n et ℓ. Justifier ainsi que les calculs de `puissance(x,n)%m` et `puissance(x,n,m)` ne sont pas de même complexité. Ce résultat théorique correspond-il à vos expériences pratiques ?

Anecdotique

En s'inspirant du paradigme « diviser pour régner » ou plutôt « régner pour diviser », les étudiants Rosencrantz et Guildenstern, amis d'école de Hamlet, proposent les deux fonctions suivantes, `puiss` et `poiss`, pour implémenter la puissance dichotomique. Sont-elles correctes ? Sont-elles efficaces ? Comment dénouer cette tragicomédie ? (Cf. Tom Stoppard, *Rosencrantz et Guildenstern ont tort*, Éditions du Seuil, Paris 1967.)

Programme VIII.1 Puissance anecdotique `hamlet.cc`

```
1  //========================================================================
2  // Though this be madness, yet there is method in't. (Hamlet, 2.2)
3  //========================================================================
4
5  // Puissance dichotragique selon Rosencrantz
6  Integer puiss( const Integer& x, const Integer& n )
7  {
8    Integer y= puiss( x, n/2 );
9    if ( n%2 == 0 ) return y*y;
10   else return y*y*x;
11 }
12
13 // Puissance dichocomique selon Guildenstern
14 Integer poiss( const Integer& x, const Integer& n )
15 {
16   if ( n < 0 )  return 0;
17   if ( n == 0 ) return 1;
18   if ( n == 1 ) return x;
19   Integer m= n/2;
20   return poiss( x, m ) * poiss( x, n-m );
21 }
```

Unsere Allergrößten, wie Archimedes, Newton, Gauß,
haben stets Theorie und Anwendung gleichmäßig umfaßt.
Felix Klein, *Elementarmathematik vom höheren Standpunkt aus*, 1908

CHAPITRE IX

Pgcd et l'algorithme d'Euclide-Bézout

Objectifs

Ce chapitre reprend l'arithmétique des nombres entiers, notamment l'algorithme d'Euclide et ses nombreuses ramifications. C'est une relecture de l'arithmétique sous un aspect algorithmique : structures algébriques sous-jacentes, preuve de correction, analyse de complexité.

C'est aussi une étape charnière pour l'algèbre, dont les chapitres suivants traiteront différents aspects. On parlera plus en détail des anneaux quotients $\mathbb{Z}_n$ au chapitre X, de la primalité et de la factorisation d'entiers au chapitre XI, on implémentera le corps des fractions $\mathbb{Q}$ au chapitre XII, suivi de l'anneau $\mathbb{Z}[i]$ dans le projet XII et des anneaux des polynômes au chapitre XIII.

Implémentation. — Afin de réaliser des implémentations complexes, songez à distribuer le travail en équipe puis à mutualiser vos solutions. Le but sera de réunir les fonctions d'intérêt général dans le fichier `integer.cc` commencé en chapitre II. Vous obtenez ainsi une mini-bibliothèque portant sur l'arithmétique des entiers. Les implémentations continueront tout au long des chapitres suivants. Comme d'habitude il convient de bien tester et commenter vos implémentations, d'autant plus en vue d'une réutilisation.

Sommaire

Approfondissement. — L'annexe IX résume brièvement le vocabulaire des anneaux commutatifs qui sera essentiel pour la suite. On saisit l'occasion de souligner quelques aspects algorithmiques et de préparer ainsi nos futures implémentations d'anneaux plus généraux. Le projet IX présente l'algorithme de Gauss-Bézout pour la résolution de systèmes d'équations linéaires sur $\mathbb{Z}$. On en déduit le théorème des diviseurs élémentaires et la classification des groupes abéliens finiment engendrés.

1. Structure de l'anneau $\mathbb{Z}$

1.1. Structure d'anneau factoriel. Le théorème fondamental de l'arithmétique dit que tout entier positif a s'exprime de manière unique comme produit de nombres premiers positifs :

$$a = 2^{v_2} \cdot 3^{v_3} \cdot 5^{v_5} \cdot 7^{v_7} \cdots = \prod_{p \text{ premier}} p^{v_p}$$

avec des exposants $v_p = v_p(a) \in \mathbb{N}$ dont tous sauf un nombre fini sont nuls. Pour le plus grand commun diviseur (pgcd) et le plus petit commun multiple (ppcm) on obtient alors

$$\operatorname{pgcd}(a,b) = \prod_{p \text{ premier}} p^{\min(v_p(a),v_p(b))} \qquad \text{et} \qquad \operatorname{ppcm}(a,b) = \prod_{p \text{ premier}} p^{\max(v_p(a),v_p(b))}.$$

Bien que ces deux formules soient importantes d'un point de vue théorique, elles n'offrent pas de solution efficace pour le calcul du pgcd ou du ppcm : pour ce faire il faudrait d'abord factoriser a et b. Or, pour les grands entiers, la factorisation est un problème très dur, dont on ne connaît pas de méthode rapide (nous y reviendrons au chapitre XI). Heureusement pour le pgcd il existe un algorithme très efficace, l'algorithme d'Euclide, qui évite entièrement le problème de factorisation.

1.2. Structure d'anneau euclidien. Étant donnés $a \in \mathbb{Z}$ et $b \in \mathbb{Z}^*$ il existe $q, r \in \mathbb{Z}$ tels que $a = bq + r$ et $|r| < |b|$. Ceci est appelé une *division euclidienne* avec quotient q et reste r. Le couple (q,r) n'est en général pas unique : pour $a = 22$ et $b = 9$, par exemple, on a $a = 2b + 4 = 3b - 5$. (On a toujours deux solutions : l'une avec $q = \lfloor \frac{a}{b} \rfloor$, l'autre avec $q = \lceil \frac{a}{b} \rceil$; elles coïncident si et seulement si b divise a dans $\mathbb{Z}$ avec reste $r = 0$.) Cette ambiguïté n'est pas gênante d'un point de vue mathématique, mais pour une implémentation sur ordinateur il faut bien fixer un choix. Précisons d'abord les exigences générales.

Définition 1.1. Soit A un anneau commutatif. Une *division euclidienne* sur A est la donnée
– d'une fonction $\nu : A \to \mathbb{N}$ vérifiant $\nu(a) = 0$ si et seulement si $a = 0$, et
– d'une application $\delta : A \times A^* \to A \times A$, $(a,b) \mapsto (q,r)$ telle que $a = bq + r$ et $\nu(r) < \nu(b)$.
Dans ce cas on appelle ν un *stathme euclidien*, et δ une *division euclidienne* par rapport au stathme ν.

Définition 1.2. Un anneau A est dit *euclidien* s'il est intègre et admet une division euclidienne.

D'après ce qui précède, $\mathbb{Z}$ est euclidien par rapport au stathme $\nu(a) = |a|$. Quant à la division euclidienne δ, on a une infinité de choix possibles. Les conventions suivantes semblent les plus utiles : elles choisissent les quotients $q = \lceil \frac{a}{b} \rfloor$, $q = \lfloor \frac{a}{b} \rfloor$, $q = \lceil \frac{a}{b} \rceil$, et $q = \lfloor \frac{a}{b} \rceil$ respectivement.

Exercice/P 1.3. Pour les types entiers en C++ l'opération `a/b` donne $\lceil \frac{a}{b} \rfloor$, la partie entière du quotient, appelé quotient « tronqué », ou encore « arrondi vers zéro ». Par exemple `5/3` vaut `1` et `5%3` vaut `2`, ainsi que `(-5)/3` vaut `-1` et `(-5)%3` vaut `-2`. Par conséquent le reste `a%b` est ou zéro ou du même signe que `a`. Cette convention a été adoptée également pour la classe `Integer`. Le vérifier sur des exemples.

Optimisation. — Très souvent on veut calculer le quotient `q` et le reste `r` en même temps. Bien sûr on pourrait écrire `q=a/b` puis `r=a%b`, mais ceci effectue deux fois la même division (expliquer pourquoi). Dans ce cas il est plus efficace de n'effectuer qu'une seule division euclidienne $(a,b) \mapsto (q,r)$ comme suit :

```
void tdiv( const Integer& a, const Integer& b, Integer& q, Integer& r )
{ mpz_tdiv_qr( q.get_mpz_t(), r.get_mpz_t(), a.get_mpz_t(), b.get_mpz_t() ); }
Integer tdiv( const Integer& a, const Integer& b )
{ return a/b; }
Integer tmod( const Integer& a, const Integer& b )
{ if( b == 0 ) return a; else return a%b; }
```

Au lieu des opérateurs `/` et `%` on peut aussi utiliser deux fonctions `tdiv` et `tmod`. Ceci permet de rectifier un petit défaut de l'opérateur `%`, à savoir que `tmod(a,0)` est toujours bien défini, bien que `tdiv(a,0)` ne le soit pas. (Expliquer pourquoi c'est mathématiquement raisonnable.)

Exercice/P 1.4. On peut définir une deuxième division euclidienne en choisissant l'unique couple (q,r) avec $a = bq + r$ et $0 \leq r < |b|$. Dans ce cas nous écrivons $q = a \operatorname{div} b$ et $r = a \bmod b$; c'est la division euclidienne avec *reste positif*, usuelle en mathématique. L'implémenter sous la forme

```
void    pdiv( const Integer& a, const Integer& b, Integer& q, Integer& r );
Integer pdiv( const Integer& a, const Integer& b );
Integer pmod( const Integer& a, const Integer& b );
```

De la même manière on pourra définir et implémenter la division euclidienne avec *reste négatif* :

```
void    ndiv( const Integer& a, const Integer& b, Integer& q, Integer& r );
Integer ndiv( const Integer& a, const Integer& b );
Integer nmod( const Integer& a, const Integer& b );
```

Indication. — Dans le souci d'efficacité on pourra commencer par `tdiv(a,b,q,r)` puis corriger `q` et `r`. Vous pouvez aussi consulter la documentation via `info gmp` pour vous informer sur les fonctions `fdiv` et `cdiv` de la bibliothèque GMP.

Exercice/P 1.5. Une façon économique de choisir (q,r) est d'exiger $a = bq + r$ avec $|r| \leq \frac{1}{2}|b|$: c'est la division avec *reste symétrique*. Vérifier qu'elle *minimise* $|r|$. L'implémenter sous la forme

```
void    sdiv( const Integer& a, const Integer& b, Integer& q, Integer& r );
Integer sdiv( const Integer& a, const Integer& b );
Integer smod( const Integer& a, const Integer& b );
```

Remarque. — La définition laisse un choix seulement dans le cas où $b = 2n$ et $r = \pm n$. Afin de résoudre cette dernière ambiguïté, on pourra choisir l'unique couple (q,r) avec q pair.

2. Le pgcd et l'algorithme d'Euclide

2.1. Définition du pgcd. Rappelons la définition du pgcd en dû détail :

Définition 2.1. On dit que d *divise* a dans $\mathbb{Z}$, noté $d \mid a$, s'il existe $d' \in \mathbb{Z}$ de sorte que $dd' = a$. On dit que c est un diviseur commun de $a_1, \dots, a_n$ dans $\mathbb{Z}$ si $c \mid a_k$ pour tout k. On dit que d est un *plus grand commun diviseur* de $a_1, \dots, a_n$ s'il est un diviseur commun et que tout autre diviseur commun c divise aussi d.

Attention. — Le pgcd n'est pas unique : si d est un pgcd de $a_1, \dots, a_n$, alors $-d$ en est un autre. Cette ambiguïté fait qu'il faut dire correctement *un* pgcd et non *le* pgcd. Pour nos futures implémentations ceci pose un problème de spécification. Heureusement dans l'anneau $\mathbb{Z}$ l'ambiguïté se limite au signe. On peut s'en tirer en choisissant *le pgcd positif* pour pgcd préféré, ce qui rend la définition univoque.

Remarque 2.2. Le pgcd jouit des propriétés suivantes (les montrer) :
- Si $a = bq + r$ alors $\operatorname{pgcd}(a,b) = \operatorname{pgcd}(b,r)$ car $c|a \;\&\; c|b \iff c|b \;\&\; c|r$.
- Pour tout $a \in \mathbb{Z}$ on a $\operatorname{pgcd}(a,0) = \operatorname{pgcd}(a) = |a|$.
- On a $\operatorname{pgcd}(a_1, a_2, \dots, a_n) = \operatorname{pgcd}(a_1, \operatorname{pgcd}(a_2, \dots, a_n))$.
  Il suffit donc de savoir calculer le pgcd de deux entiers.

2.2. L'algorithme d'Euclide. Rappelons l'algorithme d'Euclide comme il est typiquement formulé dans un cours d'algèbre. Étant donnés deux entiers a, b on construit une suite finie (r_i) de la manière suivante : comme valeurs initiales on pose $r_0 = a$ et $r_1 = b$. Tant que $r_i \neq 0$ on définit r_{i+1} par une division euclidienne $r_{i-1} = r_i q_i + r_{i+1}$ avec $|r_{i+1}| < |r_i|$. Finalement $r_{n+1} = 0$, donc la dernière division $r_{n-1} = r_n q_n$ est exacte. On vérifie aisément que r_n est un pgcd de a et b, donc $|r_n|$ est le pgcd positif cherché.

Pour l'implémentation il est inutile de stocker toute la suite $r_0, r_1, \dots, r_n$; il suffit à chaque moment de travailler avec les deux derniers éléments r_{i-1} et r_i. Voici un tel algorithme « prêt à programmer » :

Algorithme IX.1 Calcul du pgcd de deux entiers selon Euclide

Entrée: deux entiers a et b

Sortie: le pgcd positif de a et b

tant que $b \neq 0$ **faire** division euclidienne $(q,r) \leftarrow \delta(a,b)$, puis affecter $a \leftarrow b$ et $b \leftarrow r$
si $a \geq 0$ **alors retourner** a **sinon retourner** $-a$

Proposition 2.3. *L'algorithme IX.1 est correct.*

DÉMONSTRATION. Notons $a_0 = a$ et $b_0 = b$ les valeurs initiales, puis a_k et b_k les valeurs des variables a et b après la $k$ième itération.

Terminaison : Soit $\nu \colon \mathbb{Z} \to \mathbb{N}$ un stathme pour la division euclidienne δ utilisée ici : par définition on a $\nu(b_0) > \nu(b_1) > \nu(b_2) > \dots$. Comme c'est une valeur dans $\mathbb{N}$, ceci ne peut durer éternellement. On arrive donc à $\nu(b_n) = 0$ après un certain nombre n d'itérations. À ce moment-là la boucle s'arrête avec $b_n = 0$.

Correction : Si $a = bq + r$ alors $\operatorname{pgcd}(a,b) = \operatorname{pgcd}(b,r)$. Autrement dit, le pgcd est préservé lors de chaque itération de la boucle. Ainsi on obtient $\operatorname{pgcd}(a_0,b_0) = \operatorname{pgcd}(a_1,b_1) = \cdots = \operatorname{pgcd}(a_n,b_n) = \operatorname{pgcd}(a_n,0) = |a_n|$. L'algorithme renvoie donc le pgcd cherché. □

Exercice/P 2.4. Implémenter une fonction `Integer pgcd( Integer a, Integer b )`. Motiver le mode de passage des paramètres. On souhaiterait normaliser le pgcd à la fin de sorte qu'il soit toujours positif ou nul. Pour visualiser le comportement de cet algorithme et pour compter le nombre d'itérations effectuées, vous pouvez faire afficher les calculs intermédiaires. Testez votre fonction sur des entiers de plus en plus grands ; combien d'itérations faut-il environ ?

Exemple 2.5. Calculer le pgcd de $a = 33! + 1$ et $b = 32! + 1$. Peut-on trouver aussi facilement le pgcd via la décomposition en facteurs premiers ? Pour information, les factorisations sont

$$33! + 1 = 101002716748738111 \cdot 143446529 \cdot 175433 \cdot 50989 \cdot 67 \qquad \text{et}$$
$$32! + 1 = 2889419049474073777 \cdot 61146083 \cdot 652931 \cdot 2281.$$

Justifier la supériorité de l'algorithme d'Euclide par rapport au calcul du pgcd via factorisation.

2.3. Analyse de complexité. Rappelons que pour a,b vérifiant $\text{len}(a),\text{len}(b) \leq \ell$ la division euclidienne nécessite un temps $O(\ell^2)$ avec la méthode scolaire, voire $O^+(\ell)$ avec des méthodes sophistiquées. Par construction la suite des restes successifs vérifie $|r_1| > |r_2| > \cdots > |r_n| > |r_{n+1}| = 0$. Le coût total de l'algorithme d'Euclide appliqué au couple (a,b) est donc d'ordre $O(n\ell^2)$, voire $O^+(n\ell)$. Que peut-on dire du nombre n d'itérations nécessaires ?

Exercice/M 2.6. En utilisant la division euclidienne avec reste minimal, on a $|r_{i+1}| \leq \frac{1}{2}|r_i|$. En déduire que $n \leq \ell$, donc cet algorithme d'Euclide est de complexité $O(\ell^3)$, voire $O^+(\ell^2)$.

Remarque 2.7. Pour une analyse plus fine voir Gathen-Gerhard [**11**], §3.3. Il se trouve que la complexité est $O(\ell^2)$ même avec la division scolaire. Dans [**11**], §11.1 vous trouverez un raffinement de complexité $O^+(\ell)$ seulement !

Exercice/M 2.8. Expliciter une division euclidienne $(a,b) \mapsto (q,r)$ qui *maximise* $|r|$. L'algorithme d'Euclide aboutit-il toujours à trouver le pgcd ? Quelle est sa complexité dans le pire des cas ? (Voir `euclide.cc` .)

Exercice/M 2.9. En utilisant `pmod` ou `tmod`, montrer que $|r_{i+2}| < \frac{1}{2}|r_i|$ pour tout $i \geq 1$. En déduire que $n \leq 2\,\text{len}(b) \leq 2\ell$, donc la complexité est au plus un facteur deux plus grande qu'avec `smod`.

Exercice/M 2.10. On peut expliciter le pire cas de l'algorithme d'Euclide utilisant la division euclidienne avec reste positif. La suite de Fibonacci $(f_k)_{k\in\mathbb{N}}$ est définie par $f_0 = 1$, $f_1 = 1$ puis $f_{k+2} = f_{k+1} + f_k$. Les premiers termes sont $1,1,2,3,5,8,13,21,34,\ldots$

(1) Montrer que pour $a = f_{n+1}$ et $b = f_n$ l'algorithme d'Euclide nécessite exactement n itérations. Réciproquement, si pour $a > b \geq 0$ l'algorithme d'Euclide nécessite n itérations, alors $a \geq f_{n+1}$ et $b \geq f_n$.

(2) En déduire que l'usage de `smod` est plus efficace que `pmod` dans l'algorithme d'Euclide.

(3) Montrer la formule close $f_n = (\lambda_+^{n+1} - \lambda_-^{n+1})/\sqrt{5}$ avec $\lambda_\pm = (1 \pm \sqrt{5})/2$, et en déduire le théorème de Lamé : pour la division euclidienne avec reste positif le nombre d'itérations dans l'algorithme d'Euclide est majoré par $5\,\text{len}_{10}(b)$.

Exercice/P 2.11. Un théorème de Dirichlet affirme que deux entiers a,b « aléatoires » sont premiers entre eux avec probabilité $6/\pi^2 \approx 60\%$. Pour vérification empirique vous pouvez écrire un programme qui parcourt $(a,b) \in [\![1,N]\!]^2$ et compte les couples vérifiant $\text{pgcd}(a,b) = 1$. Que trouvez-vous pour $N = 10$? $N = 100$? $N = 1000$? $N = 10000$?

Analogie. — Imaginez une « forêt mathématique » formée d'une infinité d'arbres très fins, avec un arbre planté à chaque position du réseau $\mathbb{Z}^2$ dans le plan $\mathbb{R}^2$. Vous êtes à l'origine. Quelle fraction d'arbres voyez-vous ?

2.4. Bézout ou Euclide étendu. Rappelons une propriété principale de l'anneau $\mathbb{Z}$:

Proposition 2.12. *Tout sous-groupe I de $(\mathbb{Z},+)$ est de la forme $I = a\mathbb{Z}$ pour un entier $a \in \mathbb{Z}$.*

DÉMONSTRATION. Si $I = \{0\}$ alors $I = 0\mathbb{Z}$ et on prend $a = 0$. Sinon on a $I \neq \{0\}$, il existe donc $a \in I$ avec $a \neq 0$. On choisit a avec $|a|$ minimal. Comme $-a \in I$, on peut supposer que $a > 0$. Comme I est un sous-groupe, on a déjà $I \supset a\mathbb{Z}$. Réciproquement, pour $x \in I$ quelconque on considère la division euclidienne par a : il existe $q,r \in \mathbb{Z}$ tels que $x = qa + r$ et $|r| < a$. Avec $a \in I$ on a aussi $qa \in I$ et donc $r = x - qa \in I$. Notre choix minimal de a veut dire que $|r| < a$ n'est possible que pour $r = 0$, donc x est un multiple de a, autrement dit $x \in a\mathbb{Z}$. On conclut que $I = a\mathbb{Z}$. □

Proposition 2.13. *Pour tout couple $a,b \in \mathbb{Z}$ on a $a\mathbb{Z} + b\mathbb{Z} = d\mathbb{Z}$ où d est un pgcd de a et b. Il existe donc $u,v \in \mathbb{Z}$, appelés* coefficients de Bézout, *tels que $au + bv = \text{pgcd}(a,b)$. Ces coefficients ne sont pas uniques : les solutions entières de l'équation $aU + bV = d$ sont données par $\{\ (u,v) + k(\frac{b}{d}, -\frac{a}{d}) \mid k \in \mathbb{Z}\ \}$.*

DÉMONSTRATION. Soit d un pgcd de a et b. Comme $d|a$ et $d|b$ on a $d|au+bv$ pour tout $u,v \in \mathbb{Z}$, donc $a\mathbb{Z} + b\mathbb{Z} \subset d\mathbb{Z}$. D'autre part le sous-groupe $a\mathbb{Z} + b\mathbb{Z}$ est de la forme $c\mathbb{Z}$ pour un $c \in \mathbb{Z}$. Comme $a,b \in c\mathbb{Z}$ on a $c|a$ et $c|b$, il s'agit donc d'un diviseur commun de a et b. Pour celui-ci on sait que $c|d$, autrement dit $c\mathbb{Z} \supset d\mathbb{Z}$. On conclut que $a\mathbb{Z} + b\mathbb{Z} = d\mathbb{Z}$ comme énoncé. (Le reste est laissé en exercice.) □

Après avoir établi leur existence, il se pose la question naturelle de savoir comment trouver efficacement des coefficients de Bézout. Dans un anneau euclidien, on dispose de l'algorithme suivant :

Comme avant on construit une suite finie (r_i) commençant par $r_0 = a$ et $r_1 = b$ puis $r_{i+1} = r_{i-1} - r_i q_i$ par une division euclidienne itérée. Parallèlement on pose $w_0 = (1,0)$ et $w_1 = (0,1)$ puis $w_{i+1} = w_{i-1} - q_i w_i$. On assure ainsi que $r_i = w_i \binom{a}{b}$ pour tout i. On arrive finalement à $r_n = \text{pgcd}(a,b)$ et $w_n = (u,v)$ avec $r_n = au + bv$ comme souhaité. Voici un tel algorithme « prêt à programmer » :

Algorithme IX.2 Algorithme d'Euclide-Bézout

Entrée: deux entiers a_0 et b_0

Sortie: trois entiers d,u,v tels que $d = a_0u + b_0v$ soit un pgcd de a_0 et b_0

$\begin{pmatrix} a & u & v \\ b & s & t \end{pmatrix} \leftarrow \begin{pmatrix} a_0 & 1 & 0 \\ b_0 & 0 & 1 \end{pmatrix}$ // Initialement $a = a_0u + b_0v$ et $b = a_0s + b_0t$

tant que $b \neq 0$ **faire**

  division euclidienne $(q,r) \leftarrow \delta(a,b)$

  $\begin{pmatrix} a & u & v \\ b & s & t \end{pmatrix} \leftarrow \begin{pmatrix} b & s & t \\ r = a - qb & u - qs & v - qt \end{pmatrix}$ // Préserve $a = a_0u + b_0v$ et $b = a_0s + b_0t$

fin tant que

si $a \geq 0$ **alors retourner** a,u,v **sinon retourner** $-a,-u,-v$ // On renvoie toujours le pgcd positif

Exercice/M 2.14. Montrer que l'algorithme IX.2 est correct. *Indication.* — Comme pour l'algorithme d'Euclide on voit que l'algorithme IX.2 s'arrête et trouve un pgcd de a_0 et b_0. Pour les coefficients de Bézout vérifier les égalités $a_k = a_0u_k + b_0v_k$ et $b_k = a_0s_k + b_0t_k$ avant et après chaque itération de la boucle.

Exercice/P 2.15. Afin d'optimiser on peut finalement remplacer le calcul itéré de v et t pendant la boucle par un seul calcul de v après la boucle. Prouver la correction de l'algorithme IX.3 ci-dessus et l'implémenter en une fonction `Integer pgcd( Integer a0, Integer b0, Integer& u, Integer& v )`.

Attention. — Les affectations « matricielles » sont une écriture commode qui ne se traduit pas littéralement : en C++ il faudra des variables auxiliaires pour ne pas écraser des valeurs dont on aura encore besoin.

Algorithme IX.3 Algorithme d'Euclide-Bézout (légèrement optimisé)

Entrée: deux entiers a_0, b_0

Sortie: trois entiers d,u,v tels que $d = a_0u + b_0v$ soit un pgcd de a_0 et b_0

$\begin{pmatrix} a & u \\ b & s \end{pmatrix} \leftarrow \begin{pmatrix} a_0 & 1 \\ b_0 & 0 \end{pmatrix}$ // Initialement $a \equiv a_0u$ et $b \equiv a_0s \pmod{b_0}$

tant que $b \neq 0$ **faire**

  division euclidienne $(q,r) \leftarrow \delta(a,b)$

  $\begin{pmatrix} a & u \\ b & s \end{pmatrix} \leftarrow \begin{pmatrix} b & s \\ r = a - qb & u - qs \end{pmatrix}$ // Préserve $a \equiv a_0u$ et $b \equiv a_0s \pmod{b_0}$

fin tant que

si $b_0 = 0$ **alors** $v \leftarrow 0$ **sinon** $v \leftarrow (a - a_0u)/b_0$ // On sait que b_0 divise $a - a_0u$ sans reste

si $a \geq 0$ **alors retourner** a,u,v **sinon retourner** $-a,-u,-v$ // On renvoie toujours le pgcd positif

3. Premières applications

3.1. Inversion dans l'anneau quotient $\mathbb{Z}_n$. Les anneaux quotients $\mathbb{Z}/n\mathbb{Z}$ interviennent dans nombre d'applications. On va utiliser l'écriture abrégée $\mathbb{Z}_n := \mathbb{Z}/n\mathbb{Z}$ qui est moins standard mais plus concise. (Si les algorithmiciens la trouvent bien commode, les algébristes la réservent pour un tout autre objet.)

Chaque entier a représente un élément dans $\mathbb{Z}_n$ via l'application quotient $\pi_n : \mathbb{Z} \to \mathbb{Z}_n$, $a \mapsto \pi_n(a)$. Pour les implémentations il est commode de prendre l'intervalle $[\![0,n[\![:= \{0,1,2,\dots,n-1\}$ comme système préféré de représentants. Ainsi π_n établit une bijection entre $\{0,1,2,\dots,n-1\}$ et $\mathbb{Z}_n$.

Exercice/M 3.1. Montrer qu'un entier a représente un élément inversible dans $\mathbb{Z}_n$ si et seulement si $\operatorname{pgcd}(a,n) = 1$. Conclure en particulier que $\mathbb{Z}_n$ est un corps si et seulement si n est premier.

Exercice/P 3.2. Écrire une fonction `Integer inverse( Integer a, Integer n )` qui renvoie l'inverse $u \in [\![1,n[\![$ de a modulo n lorsque c'est possible, et renvoie 0 sinon. Pour un traitement plus net du cas non inversible, vous pouvez implémenter, si vous préférez, deux fonctions

```
bool inversible( Integer a, Integer n )
bool inversible( Integer a, Integer n, Integer& u )
```

La première teste simplement si a est inversible modulo n, la deuxième calcule parallèlement l'inverse u de a lorsque c'est possible. *Indication.* — Dans chaque cas on pourra adapter l'algorithme IX.3 sur mesure.

3.2. Le théorème des restes chinois. Soient a et b deux entiers premiers entre eux, autrement dit $\operatorname{pgcd}(a,b)=1$. Le théorème des restes chinois affirme que pour tout $y,z\in\mathbb{Z}$ le système

$$\begin{cases} x\equiv y \pmod a \\ x\equiv z \pmod b \end{cases}$$

admet une solution $x\in\mathbb{Z}$, et que $x+\mathbb{Z}ab$ est l'ensemble de toutes les solutions. Par exemple le système $x\equiv 7 \pmod 8$ et $x\equiv 48 \pmod{125}$ admet une unique solution $x\in[\![0,1000[\![$, mais laquelle ? Évidemment il est facile de trouver y et z à partir de x, mais comment retrouver x à partir de y et z ?

Théorème 3.3 (Théorème chinois)**.** *Soit $m_1,\dots,m_k\geq 1$ une famille d'entiers et soit $m=m_1\cdots m_k$ leur produit. Alors il existe un unique homomorphisme d'anneaux $\Phi\colon \mathbb{Z}_m\to\mathbb{Z}_{m_1}\times\cdots\times\mathbb{Z}_{m_k}$, à savoir*

$$\Phi\big(\pi_m(x)\big)=\big(\pi_{m_1}(x),\dots,\pi_{m_k}(x)\big).$$

Si m_i et m_j sont premiers entre eux pour tout $i\neq j$, alors Φ est un isomorphisme. Plus explicitement, $m_i'=m/m_i=\prod_{j\neq i}m_j$ est inversible modulo m_i, il existe donc un représentant $u_i\in[\![0,m_i[\![$ de l'inverse de m_i' modulo m_i. L'application inverse de Φ est alors donnée par $\Psi\colon \mathbb{Z}_{m_1}\times\cdots\times\mathbb{Z}_{m_k}\to\mathbb{Z}_m$

$$\Psi\big(\pi_{m_1}(y_1),\dots,\pi_{m_1}(y_k)\big)=\pi_m\big(y_1u_1m_1'+\cdots+y_ku_km_k'\big).$$

Attention. — Si les entiers $m_1,\dots,m_k$ ne sont pas premiers entre eux, alors l'homomorphisme Φ existe toujours mais il n'est plus un isomorphisme. Le détailler pour $\Phi\colon\mathbb{Z}_4\to\mathbb{Z}_2\times\mathbb{Z}_2$ en explicitant image et noyau. Montrer plus généralement qu'il n'existe pas d'homomorphisme de groupes entre $\mathbb{Z}_4$ et $\mathbb{Z}_2\times\mathbb{Z}_2$.

Exercice/M 3.4. Prouver le théorème : montrer que Φ est bien définie (existence) et qu'ici c'est le seul homomorphisme d'anneaux possible (unicité). Les formules pour Φ et Ψ sont explicites ; on peut donc calculer directement $\Psi\circ\Phi$ et $\Phi\circ\Psi$ pour montrer que ces applications sont inverses l'une à l'autre.

Exemple 3.5. Appliquons le théorème pour résoudre le système suivant :

$$\begin{cases} x &\equiv& 18000 & \pmod{19687} \\ x &\equiv& 13 & \pmod{17} \end{cases}$$

Avec $a=19687$ et $b=17$ on trouve $\operatorname{pgcd}(a,b)=1$ avec des coefficients de Bézout $u=1$ et $v=-1158$. On a $ab=334679$ et $bv=-19686$ et $au=19687$, et ainsi l'application inverse cherchée est ici

$$\Psi(\pi_a(y),\pi_b(z))=\pi_{ab}(-19686y+19687z).$$

Pour $y=18000$ et $z=13$ on trouve la solution $x=-354092069$. On réduit ensuite ce nombre modulo ab, ce qui donne $x=332992$. Vous pouvez finalement vérifier que $x\equiv y \pmod a$ et $x\equiv z \pmod b$.

À noter que malgré la petitesse du nombre cherché $x\in[\![0,ab[\![$ le calcul provoque l'apparition d'une quantité mille fois plus grande. Ce phénomène est assez général et montre que la formule explicitée dans le théorème n'est pas optimale pour le calcul : une implémentation maladroite de l'application Ψ peut largement dépasser l'intervalle $[\![0,m[\![$. Soulignons donc que *l'application* Ψ est unique, mais la *formule explicite* pour son calcul ne l'est pas ! Il convient donc d'en développer une autre qui soit plus efficace.

3.3. Un développement plus efficace. Nous donnons ici une démonstration alternative du théorème chinois qui reprend l'idée de numération en base mixte. Cette approche a le mérite de produire une formule plus efficace. Rappelons que tout entier $x\in[\![0,m_1m_2\cdots m_k[\![$ s'écrit de manière unique comme

$$x=x_1+m_1x_2+m_1m_2x_3+\cdots+m_1m_2\cdots m_{k-1}x_k$$

avec des « chiffres » $x_i\in[\![0,m_i[\![$. Ceci n'est rien autre que la numération dans la base mixte donnée par les « poids » $m_1,m_2,\dots,m_k$. (Voir le chapitre II et l'annexe II.)

Comment trouver x tel que $x\equiv y_1 \pmod{m_1}$? Évidemment il faut poser $x_1=y_1 \bmod m_1$. Comment satisfaire en plus à $x\equiv y_2 \pmod{m_2}$? Ici il suffit de résoudre $x_1+m_1x_2\equiv y_2 \pmod{m_2}$, posons donc $x_2=[u_2(y_2-x_1)] \bmod m_2$, où l'entier $u_2\in[\![0,m_2[\![$ représente l'inverse de m_1 modulo m_2. On peut ainsi continuer à calculer un par un les coefficients $x_1,x_2,\dots,x_n$:

Théorème 3.6. *Soit $m_1,\dots,m_k \geq 1$ une famille d'entiers, premiers entre eux deux à deux. Soit $u_k \in [\![0,m_k[\![$ l'inverse de $m_1\dots m_{k-1}$ modulo m_k. Étant donné $y_1,\dots,y_k \in \mathbb{Z}$ l'algorithme suivant calcule l'unique entier $x = a_k \in [\![0,m_1\cdots m_k[\![$ de sorte que $x \equiv y_1 \pmod{m_1}$, …, $x \equiv y_k \pmod{m_k}$:*

$$\begin{aligned} x_1 &\leftarrow y_1 \bmod m_1 & a_1 &\leftarrow x_1 \\ x_2 &\leftarrow [u_2(y_2-a_1)] \bmod m_2 & a_2 &\leftarrow a_1 + m_1 x_2 \\ x_3 &\leftarrow [u_3(y_3-a_2)] \bmod m_3 & a_3 &\leftarrow a_2 + m_1 m_2 x_3 \\ &\vdots \\ x_k &\leftarrow [u_k(y_k-a_{k-1})] \bmod m_k & a_k &\leftarrow a_{k-1} + m_1\cdots m_{k-1} x_k \end{aligned}$$

De plus, cet algorithme est le plus économe possible dans le sens que tous les calculs intermédiaires se placent dans l'intervalle $[\![0,m_1\cdots m_k[\![$.

Exercice/M 3.7. Prouver ce théorème. Vérifier que $x_i \in [\![0,m_i[\![$ et que $a_i \in [\![0,m_1\cdots m_i[\![$ par construction. Il ne reste qu'à montrer les congruences souhaitées $a_i \equiv y_j \pmod{m_j}$ pour $j \leq i$.

Exemple 3.8. Reprenons l'exemple précédent avec $a = 19687$, $b = 17$, donc $u = 1$. Pour $y = 18000$ et $z = 13$ on calcule $r = u(z-y) \bmod b = 16$, puis $x = y + ar = 332992$, ce qui est bien le nombre cherché.

Exemple 3.9. Pour son examen oral un étudiant X doit réunir deux examinateurs : Le professeur A ne peut que tous les 12 jours à partir de lundi, 1er janvier. Le professeur B ne peut que les mercredis. Quelles sont les dates possibles ? (Vous pouvez trouver la solution sans aucune théorie, bien sûr. Il sera néanmoins instructif de comparer votre solution avec les formules ci-dessus en précisant les étapes du calcul.)

Remarque 3.10. Ce genre de calcul permettait aux généraux chinois de dénombrer leur troupe, sans trop d'efforts pour eux-mêmes, en ordonnant : « rangez-vous 7 par 7, puis 11 par 11, puis 13 par 13, puis 17 par 17 ». Si cette anecdote est vraie on peut en déduire une borne maximum du nombre des soldats dans une troupe, et une grande agitation pendant le dénombrement.

Exercice/P 3.11. Écrire un programme qui lit un par un les couples $(m_1,y_1),\dots,(m_k,y_k)$. Il s'arrête si l'utilisateur entre la valeur $m_k = 0$ ou bien si $m_1,m_2,\dots,m_k$ ne sont plus premiers entre eux. Autrement il affiche l'unique solution $x \in [\![0,m_1m_2\cdots m_k[\![$ vérifiant les congruences $x \equiv y_i \pmod{m_i}$ pour tout $i \leq k$, puis continue à demander (m_{k+1},y_{k+1}).

Exercice 3.12. Une fermière va au marché avec une charrette plein d'oeufs. Le ministre de l'agriculture (plus exactement son chauffeur) brûle un feu rouge et casse tous les oeufs. Bien sûr un fond de l'Union Européenne est prévu précisément pour de tels accidents. Malheureusement la fermière, traumatisée par le choc, se souvient seulement qu'elle avait essayé de ranger ses oeufs par $2,3,4,5,6$ et chaque fois il en restait un, et qu'elle avait pu finalement les ranger par 7. Le ministre suppose qu'elle avait moins de 600 oeufs, ce qui est le plafond exigé par l'administration (dont on ignore les raisons). Les intéressés arrivent-ils à remplir les formulaires nécessaires ? La fermière combien d'oeufs avait-elle ? Quel est l'âge du chauffeur ?

Une astuce qui sert plus qu'une fois est une théorie.
anonyme

COMPLÉMENT IX

Le vocabulaire des anneaux

Le bref résumé qui suit est une invitation à relire votre cours d'algèbre. Pour notre propos l'exemple phare est l'anneau $\mathbb{Z}$ des entiers et l'algorithme d'Euclide. Avant tout nous essayerons donc d'approfondir la notion d'anneau euclidien introduite en §1.2 au début de ce chapitre. En même temps il semble utile de rappeler les notions de base afin de fixer le vocabulaire des anneaux commutatifs, surtout de la trilogie des anneaux *euclidiens, principaux, factoriels*. Ceci servira à mieux situer le cas particulier $\mathbb{Z}$ et de préparer de futures implémentations d'anneaux plus généraux.

Sommaire

1. Anneaux et corps

1.1. Anneaux. Commençons par le tout début (ou presque) :

Définition 1.1. Un *anneau* $(A,+,\cdot)$ est un ensemble A muni de deux applications, *l'addition* $+\colon A\times A\to A$ et la *multiplication* $\cdot\colon A\times A\to A$, de sorte que $(A,+)$ soit un groupe abélien, que $(A,\cdot)$ soit un monoïde, et que la multiplication soit distributive sur l'addition. La distributivité veut dire que pour tout $x,y,z\in A$ on a

$$x\cdot(y+z)=(x\cdot y)+(x\cdot z),$$
$$(x+y)\cdot z=(x\cdot z)+(y\cdot z).$$

On note $0=0_A$ l'élément neutre pour l'addition et $1=1_A$ l'élément neutre pour la multiplication ; ils sont uniquement déterminés par la structure d'anneau. On exige que $1\neq 0$ pour exclure le cas dégénéré $A=\{0\}$.

Remarque 1.2. La distributivité implique pour tout $a\in A$ que $a\cdot 0=a\cdot(0+0)=a\cdot 0+a\cdot 0$ donc $a\cdot 0=0$. De la même manière $a\cdot(-1)+a=a\cdot(-1)+a\cdot 1=a(-1+1)=a\cdot 0=0$ donc $a\cdot(-1)=-a$. De même $0\cdot a=0$ et $(-1)\cdot a=-a$.

Définition 1.3. Un anneau $(A,+,\cdot)$ est dit *commutatif* si la multiplication $\cdot$ est commutative.

Sauf mention du contraire nous supposerons dans la suite que les anneaux considérés sont commutatifs.

Exemple 1.4. Voici quelques exemples d'anneaux : l'anneau des entiers, noté $\mathbb{Z}$; les entiers modulo n, noté $\mathbb{Z}_n$; les nombres rationnels $\mathbb{Q}$, réels $\mathbb{R}$, complexes $\mathbb{C}$, tous avec leurs opérations usuelles. Les nombres naturels $\mathbb{N}$ avec leur addition et leur multiplication usuelles ne forment pas un anneau car $(\mathbb{N},+)$ n'est pas un groupe.

Exemple 1.5. Si $(A_i)_{i\in I}$ est une famille d'anneaux, alors leur produit cartésien $A=\prod_{i\in I}A_i$ est un anneau pour l'addition $(a_i)_{i\in I}+(b_i)_{i\in I}=(a_i+b_i)_{i\in I}$ et la multiplication $(a_i)_{i\in I}\cdot(b_i)_{i\in I}=(a_i\cdot b_i)_{i\in I}$. Dans le cas d'une famille finie d'anneaux $A_1,\dots,A_n$ on écrit aussi $A=A_1\times\cdots\times A_n$ pour leur produit cartésien.

Exemple 1.6. Si A est un anneau, on peut construire l'anneau $A[X]$ des polynômes en une variable X à coefficients dans A. (Le chapitre XIII présentera un développement détaillé.) Cette construction peut être itérée : $A[X,Y]=A[X][Y]$ est l'anneau des polynômes en deux variables, $A[X,Y,Z]=A[X,Y][Z]$ est l'anneau des polynômes en trois variables, etc.

Définition 1.7. Un *sous-anneau* d'un anneau $(A,+,\cdot)$ est une partie $B\subset A$ telle que $(B,+)$ soit un sous-groupe de $(A,+)$ ainsi qui $(B,\cdot)$ soit un sous-monoïde de $(A,\cdot)$. Dans ce cas $(B,+,\cdot)$ est un anneau pour la restriction de l'addition et de la multiplication.

Remarque 1.8. Si $(B_i)_{i\in I}$ est une famille de sous-anneaux d'un anneau A, alors l'intersection $B=\bigcap_{i\in I} B_i$ est un sous-anneau de A. *Attention.* — La réunion de sous-anneaux n'est en général pas un sous-anneau.

Définition 1.9. Soient A un anneau, $B\subset A$ un sous-anneau et $S\subset A$ une partie quelconque. On note $B[S]$ le plus petit sous-anneau de A contenant B et S, appelé *anneau engendré* par S sur B.

Exemple 1.10. Dans $\mathbb{C}$ on note $\mathbb{Z}[i]$ l'anneau engendré par i sur $\mathbb{Z}$. Montrer que $\mathbb{Z}[i]=\{a+bi \mid a,b\in\mathbb{Z}\}$.

Exemple 1.11. Dans $\mathbb{R}$ on note $\mathbb{Q}[\sqrt{2}]$ l'anneau engendré par $\sqrt{2}$ sur $\mathbb{Q}$. Montrer que $\mathbb{Q}[\sqrt{2}]=\{a+b\sqrt{2} \mid a,b\in\mathbb{Q}\}$.

Très souvent on veut conclure que $ab=0$ implique $a=0$ ou $b=0$. Cette propriété ne fait pas partie de la définition d'anneau, et elle n'est pas vérifié dans $\mathbb{Z}_6$, par exemple.

Définition 1.12. On note $A^*=A\smallsetminus\{0\}$ l'ensemble des éléments non nuls. Un élément $a\in A^*$ est un *diviseur de zéro* s'il existe $b\in A^*$ tel que $ab=0$. L'anneau A est *intègre* s'il n'admet pas de diviseurs de zéro, c'est-à-dire si $ab=0$ implique $a=0$ ou $b=0$. Autrement dit, A est intègre si $(A^*,\cdot)$ est un monoïde.

Exemple 1.13. L'anneau $\mathbb{Z}$ est intègre. L'anneau $\mathbb{Z}_n$ est intègre si et seulement si n est premier.

Exemple 1.14. Dans un anneau intègre tout sous-anneau est intègre. En particulier $\mathbb{Z}[i]\subset\mathbb{C}$ est intègre.

Exemple 1.15. Un anneau produit $A=A_1\times\cdots\times A_n$ avec $n\geq 2$ n'est jamais intègre. (Pourquoi ?)

Définition 1.16. Un élément $u\in A$ est dit *inversible* s'il existe $v\in A$ de sorte que $uv=1$. On note $A^\times$ le groupe multiplicatif des éléments inversibles dans A. On dit que A est un *corps* si $A^\times=A^*$, c'est-à-dire si tout élément non nul admet un inverse. Autrement dit, A est un corps si $(A^*,\cdot)$ est un groupe.

Exemple 1.17. Bien sûr $\mathbb{Q}$, $\mathbb{R}$ et $\mathbb{C}$ sont des corps, et $\mathbb{Z}$ ne l'est pas : $\mathbb{Z}^\times=\{\pm1\}$ diffère de $\mathbb{Z}^*=\mathbb{Z}\smallsetminus\{0\}$. Les éléments de $\mathbb{Z}_n^\times$ correspondent aux entiers qui sont premiers avec n, et $\mathbb{Z}_n$ est un corps ssi n est premier.

Exercice 1.18. Un anneau intègre fini A est un corps. *Indication.* — Pour $a\in A^*$ la multiplication $x\mapsto ax$ définit une application $\gamma_a\colon A\to A$. Montrer qu'elle est injective, puis bijective. Conclure.

Définition 1.19. Le groupe $A^\times$ agit naturellement sur A via la restriction $A^\times\times A\to A$ de la multiplication. Deux éléments $a,b\in A$ sont *associés*, noté $a\sim b$, s'ils sont dans la même orbite sous cette action, c'est-à-dire s'il existe $u\in A^\times$ tel que $ua=b$. (Vérifier qu'il s'agit d'une relation d'équivalence.)

Exemple 1.20. Dans $\mathbb{Z}$ on a $\mathbb{Z}^\times=\{\pm1\}$, donc a et $-a$ sont associés. Dans chaque classe d'éléments associés on distingue ici un élément préféré : l'unique élément non négatif. Par exemple, pour calculer le pgcd de deux nombres entiers, on préfère le pgcd positif. Pour le moment ceci n'est qu'une convention pour rendre le pgcd unique.

Exercice 1.21. Si $a\sim b$ alors $a\mid b$ et $b\mid a$. Dans un anneau intègre on a aussi la conclusion réciproque.

1.2. Divisibilité. Bien connue des entiers, la notion de divisibilité se retrouve naturellement dans la théorie des anneaux :

Définition 1.22. On dit que *a divise b* dans A, noté $a\mid b$, s'il existe $c\in A$ de sorte que $ac=b$. On dit que c est un diviseur commun de $x_1,\dots,x_n$ dans A si $c\mid x_i$ pour tout i. On dit que d est un *plus grand commun diviseur* de $x_1,\dots,x_n$ s'il est un diviseur commun et que tout autre diviseur commun c divise aussi d.

Exercice 1.23. La divisibilité $a\mid b$ définit un ordre partiel sur A, dont 1 est un plus petit élément et 0 est le plus grand élément (le vérifier). Les éléments inversibles sont exactement les diviseurs de 1. Dans un anneau intègre $a\mid b$ et $b\mid a$ entraîne $a\sim b$ (le vérifier). Dans ce cas les pgcd de $x_1,\dots,x_n$ sont associés entre eux.

On connaît les notions *irréductible* et *premier* de l'anneau $\mathbb{Z}$, où elles coïncident. Pour un anneau général il faut les distinguer par une définition précise :

Définition 1.24. Un élément $a\in A$ est *irréductible* si $a=bc$ entraîne ou $b\in A^\times$ ou $c\in A^\times$.

Définition 1.25. Un élément $p\in A$ est *premier* s'il n'est pas inversible et si $p\mid ab$ entraîne $p\mid a$ ou $p\mid b$.

Exercice 1.26. Par définition un élément irréductible n'est ni nul ni inversible. (Relire la définition.) Par contre, l'élément 0 peut être premier (expliciter sous quelle condition exactement). L'irréductibilité de a veut dire que a n'admet pas de facteurs non triviaux : si $b\mid a$, alors $b\sim 1$ ou $b\sim a$. Montrer qu'un élément premier non nul est irréductible. (La réciproque est fausse en générale, voir l'exercice 4.6.)

1.3. Homomorphismes. Comme pour tout objet algébrique, la notion d'homomorphisme est primordiale pour la théorie des anneaux.

Définition 1.27. Un *homomorphisme* entre deux anneaux unitaires A et A' est une application $\varphi\colon A \to A'$ vérifiant $\varphi(a+b) = \varphi(a)+\varphi(b)$ pour tout $a,b \in A$, ainsi que $\varphi(a\cdot b) = \varphi(a)\cdot\varphi(b)$ et $\varphi(1) = 1$.

Exemple 1.28. Pour tout anneau A l'identité id_A est un homomorphisme. Si $f\colon A \to B$ et $g\colon B \to C$ sont des homomorphismes d'anneaux, alors leur composée $g\circ f\colon A \to C$ est un homomorphisme d'anneaux.

Remarque 1.29. Si $B \subset A$ est un sous-anneau, alors l'inclusion $B \hookrightarrow A$ est un homomorphisme d'anneaux.

Remarque 1.30. Soit $\varphi\colon A \to A'$ un homomorphisme d'anneaux. Alors pour tout sous-anneau B de A, l'image $\phi(B)$ est un sous-anneau de A'. Pour tout sous-anneau B' de A', l'image réciproque $\phi^{-1}(B')$ est un sous-anneau de A.

Définition 1.31. Un *isomorphisme* d'anneaux est un homomorphisme bijectif.

Remarque 1.32. Un homomorphisme d'anneaux $\varphi\colon A \to A'$ est un isomorphisme si et seulement s'il existe un homomorphisme d'anneaux $\psi\colon A' \to A$ tel que $\psi\circ\varphi = \mathrm{id}_A$ et $\varphi\circ\psi = \mathrm{id}_{A'}$. (Le montrer.)

Exemple 1.33. La projection canonique $\mathbb{Z} \to \mathbb{Z}_n$ est un homomorphisme surjectif mais non injectif (pourvu que *nne*0). L'inclusion $\mathbb{Z} \hookrightarrow \mathbb{Q}$ est un homomorphisme injectif mais non surjectif.

Exemple 1.34. Pour toute paire $m,n \in \mathbb{Z}$ il existe un unique homomorphisme d'anneaux $\varphi\colon \mathbb{Z}_{mn} \to \mathbb{Z}_m \times \mathbb{Z}_n$. C'est un isomorphisme si et seulement si m et n sont premiers entre eux. (Le montrer.) Dans ce cas on obtient un isomorphisme des groupes multiplicatifs $\varphi^\times\colon \mathbb{Z}_{mn}^\times \to \mathbb{Z}_m^\times \times \mathbb{Z}_n^\times$.

1.4. Idéaux. Après les homomorphismes il est naturel de regarder les idéaux :

Définition 1.35. Pour un sous-ensemble $I \subset A$ on définit $AI := \{ax \mid a \in A, x \in I\}$. Un *idéal* I est un sous-groupe additif de A tel que $AI = I$, c'est-à-dire pour tout $a \in A$ et $x \in I$ on a $ax \in I$.

Exemple 1.36. Pour $x \in A$ l'ensemble $(x) = Ax = \{ax \mid a \in A\}$ est un idéal, dit l'idéal *principal* engendré par x. Plus généralement toute famille $x_1,\dots,x_n \in A$ engendrent un idéal $(x_1,\dots,x_n) := \{a_1x_1 + \cdots + a_nx_n \mid a_i \in A\}$, et tout sous-ensemble $X \subset A$ engendre un idéal $(X) := \{\sum a_ix_i \mid a_i \in A, x_i \in X\}$. (Vérifier qu'il s'agit d'idéaux.)

Exemple 1.37. L'idéal (0) est réduit à l'élément 0. L'idéal (1) est l'anneau A tout entier. Il se peut que ce soient les seuls : A est un corps si et seulement si (0) et (1) sont les seuls idéaux dans A (le montrer).

Exercice 1.38. Toute question de divisibilité se reformule en terme d'idéaux. Vérifier par exemple que $a \mid b$ équivaut à $(b) \subset (a)$. Ensuite montrer que d est un pgcd de $x_1,\dots,x_n$ si et seulement si (d) est le plus petit idéal principal qui contienne $(x_1,\dots,x_n)$. Formuler puis montrer un énoncé analogue pour le ppcm.

Proposition 1.39. *Pour un homomorphisme $\varphi\colon A \to A'$ on note* $\ker(\varphi) := \{a \in A \mid \varphi(a) = 0\}$ *son* noyau. *Il s'agit d'un idéal de A. Réciproquement tout idéal I de A donne lieu à un anneau quotient A/I tel que la projection canonique $\pi\colon A \to A/I$ soit un homomorphisme d'anneaux avec* $\ker(\pi) = I$.

Exercice 1.40. Soit A un anneau et $I \subset A$ un idéal. Alors la projection canonique $\pi\colon A \to A/I$ établit une bijection entre les idéaux J contenant I et les idéaux de l'anneau quotient A/I, définie par $J \mapsto J/I$. Montrer ainsi que les idéaux de l'anneau $\mathbb{Z}_m = \mathbb{Z}/m\mathbb{Z}$ sont de la forme $n\mathbb{Z}/m\mathbb{Z}$ où $n \mod m$. Expliciter le treilli des idéaux de $\mathbb{Z}_{12}$.

Définition 1.41. Un idéal $I \subset A$ est dit *premier* si A/I est intègre, et *maximal* si A/I est un corps.

Remarque 1.42. Autrement dit, un idéal $I \subset A$ est premier si et seulement $I \neq A$ et $ab \in I$ implique $a \in I$ ou $b \in I$.

Si A/I est un corps, alors en particulier A/I est intègre, donc tout idéal maximal $I \subset A$ est premier.

Si A/I est un corps alors ses seuls idéaux sont 0 et A/I. Les seuls idéaux de A contenant I sont donc I et A. Ceci implique que I est maximal si et seulement si tout idéal J vérifiant $I \subset J \subset A$ est soit $J = I$ soit $J = A$.

Théorème 1.43. *Tout homomorphisme $\varphi\colon A \to A'$ factorise de manière unique comme $\varphi = \iota\bar{\varphi}\pi$ par la projection $\pi\colon A \to A/\ker(\varphi)$, un isomorphisme $\bar{\varphi}\colon A/\ker(\varphi) \to \mathrm{im}(\varphi)$, et l'inclusion $\iota\colon \mathrm{im}(\varphi) \to A'$.*

Exemple 1.44. Tout anneau A admet un unique homomorphisme $\varphi\colon \mathbb{Z} \to A$. (Le construire.) Son image est le plus petit sous-anneau de A. (Pourquoi ?) Son noyau $\ker(\varphi) = (n)$ est engendré par un certain entier $n \geq 0$. On appelle n la *caractéristique* de l'anneau A. Ainsi tout anneau de caractéristique n contient donc un sous-anneau isomorphe à $\mathbb{Z}_n$.

2. Anneaux euclidiens

2.1. Stathmes euclidiens. Les notions de division euclidienne et de stathme euclidien on été précisées dans la définition 1.1 du chapitre IX. Pour un stathme donné $\nu : A \to \mathbb{N}$ il existe en général plusieurs divisions euclidiennes possibles ; pour une implémentation il faut donc spécifier laquelle on choisit.

Exercice 2.1. Soit $\nu : A \to \mathbb{N}$ un stathme euclidien ; on fixe deux applications $\mathrm{div}, \mathrm{mod} : A \times A^* \to A$ de sorte que le quotient $q = a \operatorname{div} b$ et le reste $r = a \bmod b$ satisfassent $a = bq + r$ et $\nu(r) < \nu(b)$. Bien que commode, la donnée de deux applications div et mod est un peu redondante : expliquer comment définir mod à partir de div, et réciproquement comment définir div à partir de mod (dans un anneau intègre).

Exercice 2.2. Dans ce cours nous considérons des stathmes euclidiens à valeurs dans $\mathbb{N}$. Après réflexion, on n'utilise que l'ordre sur $\mathbb{N}$. Essayons donc de dégager les propriétés nécessaires pour l'algorithme d'Euclide :

Soit N un ensemble muni d'une relation d'ordre $\leq$ tel que toute partie non vide $S \subset N$ admette un plus petit élément. Dans ce cas on dit que $(N, \leq)$ est un *ensemble bien ordonné*. Montrer que toute suite décroissante $n_1 > n_2 > n_3 > \dots$ dans N est forcément de longueur finie. Définir ce qui est un anneau euclidien avec stathme $\nu : A \to N$. Vérifier que l'algorithme d'Euclide reste correct dans ce cadre généralisé. Pour un exemple non trivial, regarder l'anneau $A = \mathbb{Z} \times \mathbb{Z}$ avec le stathme $\nu : A \to \mathbb{N}^2$, $\nu(a,b) = (|a|, |b|)$. Ici il convient de munir $\mathbb{N}^2$ de l'ordre lexicographique. Vérifier que A admet une division euclidienne par rapport à ν. (Avouons que A est facile à construire mais il n'est pas intègre, il n'est donc pas euclidien dans le sens de la définition usuelle. Il existe aussi de tels exemples intègres.)

2.2. Le stathme minimal. Si A est euclidien, le stathme $\nu : A \to \mathbb{N}$ n'est pas unique : par exemple on peut le composer avec une application $\phi : \mathbb{N} \to \mathbb{N}$, $\phi(0) = 0$, ϕ strictement croissante. Par contre on peut s'intéresser au plus petit stathme euclidien sur A.

Exercice 2.3. Soit A un anneau euclidien. On définit $\mu : A \to \mathbb{N}$ par $\mu(x) = \min_\nu \nu(x)$ où ν parcourt tous les stathmes euclidiens sur A. Montrer que μ est un stathme euclidien.

Exercice 2.4. Un corps K est-il un anneau euclidien ? Est-ce que toute fonction $\nu : K \to \mathbb{N}$, avec $\nu(a) = 0$ ssi $a = 0$, est un stathme euclidien ? Quelle est donc le plus petit stathme euclidien sur K ? Réciproquement, un anneau avec un stathme euclidien qui ne prend que deux valeurs, est-il un corps ?

Exercice 2.5. Même pour $\mathbb{Z}$ la valeur absolue $a \mapsto |a|$ n'est pas le seul stathme intéressant. Rappelons que $\mathrm{len} : \mathbb{Z} \to \mathbb{N}$ associe à chaque entier a la longueur de son développement binaire, c'est-à-dire $\mathrm{len}\, a := \min\{\ell \in \mathbb{N} \mid |a| < 2^\ell\}$, donc $\mathrm{len}\, 0 = 0$ et $\mathrm{len}(a) = 1 + \lfloor \log_2 |a| \rfloor$ pour $a \neq 0$. Montrer que len est un stathme euclidien sur $\mathbb{Z}$: pour tout a et $b \neq 0$ dans $\mathbb{Z}$ il existe $q, r \in \mathbb{Z}$ tels que $a = bq + r$ et $\mathrm{len}(r) < \mathrm{len}(b)$. Prouver qu'il s'agit même du stathme minimal.

Exercice 2.6. Le stathme euclidien minimal est canonique dans le sens qu'il ne dépend que de la structure d'anneau. Soit A un anneau intègre. On définit une famille croissante $A_0 \subset A_1 \subset A_2 \subset \cdots \subset A$ comme suit. On pose $A_0 = \{0\}$ puis par récurrence $A_n = A_{n-1} \cup \{a \in A \mid aA + A_{n-1} = A\}$. On constate par exemple que $A_1 = \{0\} \cup A^\times$. Montrer que A est euclidien si et seulement si $A = \bigcup A_n$; dans ce cas la fonction $\mu : A \to \mathbb{N}$ définie par $\mu(a) = \min\{n \in \mathbb{N} \mid a \in A_n\}$ est le stathme euclidien minimal sur A.

Exercice 2.7. Outre son intérêt théorique, le stathme euclidien minimal assure un fonctionnement efficace de l'algorithme d'Euclide ; il évite en particulier la pathologie rencontrée dans l'exercice 2.8 du chapitre IX. Montrer que le stathme euclidien minimal μ a d'autres propriétés sympathiques :

(1) On a $\mu(a) = 1$ si et seulement si a est inversible. Si $\mu(a) = 2$ alors a est irréductible.

(2) Pour tout $a, b \in A^*$ on a $\mu(ab) \geq \mu(a)$, avec égalité si et seulement si b est inversible. (facile)

(3) Pour tout $a, b \in A^*$ on a même $\mu(ab) \geq \mu(a) + \mu(b) - 1$. (plus fort mais plus difficile)

(4) Soit δ une division euclidienne par rapport au stathme μ. Si $a = bq$ alors $\delta(a,b) = (q, 0)$.

Rappeler que ce sont des propriétés bien connues des stathmes euclidiens « raisonnables » sur $\mathbb{Z}$.

Exercice 2.8. On rappelle que $b \mapsto |b|$ est un stathme euclidien sur $\mathbb{Z}$. À titre d'avertissement regardons $\nu : \mathbb{Z} \to \mathbb{N}$ définie par $\nu(b) = b$ si $b \geq 0$, et $\nu(b) = -2b$ si $b < 0$. Montrer que c'est un stathme euclidien, mais aucune des propriétés sympathiques énoncées dans l'exercice précédent n'est vérifiée. Ceci souligne que nous avons tout intérêt d'utiliser le stathme minimal, ou au moins d'exiger certaines de ses propriétés.

Pour une discussion approfondie d'anneaux euclidiens, et des solutions aux exercices précédents, nous renvoyons à l'article de P. Samuel, *About Euclidean Rings*, Journal of Algebra 19 (1971), pages 282–301, dont le §4 traite du stathme euclidien minimal.

3. Anneaux principaux

3.1. Motivation. Étant donnée une famille d'éléments $a_1,\dots,a_n \in A$ et $b \in A$ on considère l'équation $a_1x_1+\cdots+a_nx_n = b$. On veut savoir si elle admet une solution $(x_1,\dots,x_n) \in A^n$ et on souhaite en trouver une le cas échéant. Plus généralement on voudrait décrire toutes les solutions en terme d'une solution particulière et une famille engendrant toutes les solutions du problème homogène $a_1x_1+\cdots+a_nx_n = 0$.

Remarque 3.1. Supposons que notre anneau A admet une solution algorithmique à ce problème. Dans ce cas on peut en particulier déterminer si b appartient à l'idéal $(a_1,\dots,a_n)$. On peut ainsi déterminer : si a est inversible dans A, ce qui équivaut à $1 \in (a)$; si a est divisible par b dans A, ce qui équivaut à $a \in (b)$; si a,b sont associés dans A, ce qui équivaut à $(a) = (b)$, c'est-à-dire $a \in (b)$ et $b \in (a)$. La liste des applications est longue…

Un cadre adéquat pour traiter les équations linéaires sont les anneaux principaux :

Définition 3.2. Un idéal $I \subset A$ est *principal* s'il est engendré par un seul élément, c'est-à-dire que $I = (a)$ pour un certain $a \in I$. Un anneau A est dit *principal* s'il est intègre et tout idéal I dans A est principal.

Exercice 3.3. On a vu que l'anneau $\mathbb{Z}$ est principal. L'anneau $\mathbb{Z}[X]$ des polynômes sur $\mathbb{Z}$, par contre, n'est pas principal : l'idéal $(2,X)$ par exemple n'est pas principal.

Exercice 3.4. Vérifier que dans un anneau principal d est un pgcd de $x_1,\dots,x_n$ si et seulement si $(d) = (x_1,\dots,x_n)$. En déduire une identité de Bézout. Si l'anneau n'est pas principal, la situation se complique : Dans $\mathbb{Z}[X]$, montrer que 1 est un pgcd de 2 et X, mais $(1) \subsetneq (2,X)$. Dans un tel cas il est inutile de chercher des coefficients de Bézout.

Exercice 3.5. Montrer que tout anneau euclidien est principal, en s'inspirant de la preuve que nous avons vue pour $\mathbb{Z}$.

Remarque 3.6. Légèrement plus généraux que les anneaux euclidiens, les anneaux principaux forment une classe plus grande mais encore bien maniable. Il existe des anneaux principaux non euclidiens, par exemple le fameux anneau $\mathbb{Z}[\sqrt{-19}]$, mais une analyse détaillée nous entraînerait trop loin.

3.2. Aspects algorithmiques. Comme dans le cas euclidien, une implémentation d'un anneau principal nécessitera une fonction concrète pour les coefficients de Bézout. Voici une formulation précise :

Définition 3.7. Soit A un anneau. Une *fonction de Bézout* est une application $\beta : A\times A \to A\times A$, $(a,b) \mapsto (u,v)$ telle que $au+bv$ soit un pgcd de a et b.

Remarque 3.8. Sur un anneau principal il existe toujours des fonctions de Bézout. Comme on a vu dans le cas $\mathbb{Z}$, il est illusoire d'espérer unicité ; le mieux que l'on puisse faire est d'en choisir une selon le contexte. Après l'existence, le vrai problème est le calcul efficace : étant donné (a,b) dans un anneau principal, comment trouver des coefficients de Bézout ? Heureusement cette difficulté disparaît dans le cas euclidien :

Exercice 3.9. Si A est euclidien, montrer que l'algorithme d'Euclide étendu définit une fonction de Bézout.

Exercice 3.10. Étant donné une fonction de Bézout sur A, montrer que tout idéal finiment engendré de A est principal. (A priori il peut y avoir de méchants idéaux qui ne sont pas finiment engendrés, et donc non principaux. On exclut cette pathologie en exigeant que A soit *noethérien*.)

Exercice 3.11. L'équation $a_1x_1+\cdots+a_nx_n = b$ admet une solution si et seulement si l'idéal $(a_1,\dots,a_n)$ contient l'élément b. Pour $n = 1$ il s'agit de tester la divisibilité de b par a_1 ; calculer la solution x_1 revient à diviser b par a_1. Pour $n = 2$, supposons que A est principal avec fonction de Bézout β. Expliciter un algorithme pour résoudre l'équation $a_1x_1+a_2x_2 = b$. Esquisser un solution du problème général $a_1x_1+\cdots+a_nx_n = b$ (par récurrence).

Remarque 3.12. Le traitement algorithmique d'idéaux dans $\mathbb{Z}[X]$ et $K[X,Y]$, voire dans $K[X_1,\dots,X_n]$, est un problème assez profond, et hors de la portée de ce cours. Il en existe une solution via les *bases de Gröbner* et les algorithmes associés. Si cela vous intéresse, lisez le premier chapitre de *Some Tapas of Computer Algebra* édité par A.M. Cohen, H. Cuypers et H. Sterk (Springer-Verlag, Berlin 1999).

Exercice 3.13. Dans tout anneau A on a les équivalences : a est premier $\Leftrightarrow$ l'idéal (a) est premier $\Leftrightarrow$ $A/(a)$ est intègre. Dans $\mathbb{Z}$ par exemple n est premier ssi $\mathbb{Z}/(n)$ est intègre. Dans ce cas $\mathbb{Z}/(n)$ est même un corps (rappeler pourquoi). On conclut que (n) est même un idéal maximal. Ce phénomène n'est pas un hasard :

Exercice 3.14. Pour un élément $a \neq 0$ d'un anneau principal A sont équivalents : l'élément a est irréductible $\Leftrightarrow$ l'idéal (a) est premier $\Leftrightarrow$ l'idéal (a) est maximal. Expliquer en rétrospective pourquoi dans $\mathbb{Z}$ on ne voit pas certaines subtilités, qui risquent de se produire dans des anneaux plus généraux.

4. Anneaux factoriels

4.1. Factorisation. Au début du chapitre on a mentionné la structure factorielle de l'anneau $\mathbb{Z}$. Dans un cours d'algèbre on en extrait l'abstraction suivante. Étant donné une partie $P \subset A$ on note $\mathbb{N}^{(P)}$ l'ensemble des fonctions $\nu\colon P \to \mathbb{N}$ à support fini, c'est-à-dire que $\nu_p = 0$ pour tout $p \in P$ sauf un nombre fini. Cette précaution permet de définir l'application $\Pi_P\colon A^\times \times \mathbb{N}^{(P)} \to A^*$ par $\Pi_P(u,\nu) = u \cdot \prod_{p\in P} p^{\nu_p}$. À noter qu'il s'agit bien d'un produit *fini*, bien que l'ensemble P puisse être infini.

Définition 4.1. Une *structure factorielle* sur un anneau A est une partie $P \subset A$ telle que l'application $\Pi_P\colon A^\times \times \mathbb{N}^{(P)} \to A^*$ soit une bijection. Dans ce cas l'application inverse Π_P^{-1} est appelée la *factorisation* par rapport à P. Un anneau est dit *factoriel* s'il admet une structure factorielle.

Un anneau factoriel et forcément intègre. En général il peut être difficile à déterminer si un anneau intègre donné est factoriel ou non. Pour les anneaux euclidiens ou principaux, par contre, la question devient facile. Si vous l'avez déjà vu dans votre cours d'algèbre, essayez de redémontrer le théorème suivant :

Théorème 4.2. *Tout anneau euclidien est principal. Tout anneau principal est factoriel.*

Exercice 4.3. En déduire en particulier que l'anneau $\mathbb{Z}$ est factoriel, comme énoncé au début de ce chapitre. Pour la structure factorielle on choisit typiquement l'ensemble des nombres premiers *positifs*. (Vérifier que l'on pourrait aussi choisir des signes différents.) *Attention.* — On pourrait être tenté de prendre la factorialité de $\mathbb{Z}$ comme une évidence. Ce n'est pas du tout le cas. On explicitera un anneau non factoriel plus bas.

Exercice 4.4. Soit A un anneau avec une structure factorielle P. Vérifier que $\Pi_P\colon A^\times \times \mathbb{N}^{(P)} \to A^*$ est un isomorphisme de monoïdes. (Rappeler d'abord les lois sur $A^\times \times \mathbb{N}^{(P)}$ et sur A^*.) Montrer qu'une structure factorielle $P \subset A$ consiste d'éléments irréductibles. Chaque élément irréductible de A est associé à exactement un élément de P. Ainsi P est un *système de représentants* des éléments irréductibles de A.

Exercice 4.5. Vérifier que pour toute application $\varepsilon\colon P \to A^\times$ l'ensemble $\varepsilon P = \{\varepsilon(p)\cdot p \mid p \in P\}$ donne également une structure factorielle. Réciproquement, si P et P' sont deux structures factorielles sur A, alors il existe $\varepsilon\colon P \to A^\times$ de sorte que $P' = \varepsilon P$. Conclusion : s'il existe une structure factorielle sur A elle est essentiellement unique (à multiplication par des inversibles près).

Exercice 4.6. Dans beaucoup d'anneaux qui apparaissent dans la nature c'est la *non-unicité* de la factorisation qui empêche la factorialité, c'est-à-dire que Π_P est surjectif mais non injectif. Vous rencontrerez de tels anneaux dans votre cours d'algèbre. En voici un exemple simple : les polynômes $p \in \mathbb{R}[X]$ vérifiant $p'(0) = 0$ forment un sous-anneau $A \subset \mathbb{R}[X]$. Les polynômes X^2 et X^3 sont irréductibles dans A. (Pourquoi ?) Par conséquent X^6 admet *deux factorisations distinctes* en facteurs irréductibles dans A, à savoir $X^6 = X^2X^2X^2$ et $X^6 = X^3X^3$.

Exercice 4.7. Montrer le lemme d'Euclide dans un anneau factoriel : si $\operatorname{pgcd}(a,b) = 1$ et $b \mid ac$, alors $b \mid c$. En déduire tout élément irréductible est premier. Sans factorialité, il n'en est rien : dans l'anneau A de l'exemple précédent, X^2 est irréductible mais non premier : on a $X^2 | X^3X^3$ sans que $X^2 | X^3$. De même $X^3 | X^2X^4$ sans que $X^3 | X^2$ ou $X^3 | X^4$.

Exemple 4.8. Remarquons finalement que $\mathbb{Z}[\sqrt{-5}]$ est intègre mais non factoriel : on peut vérifier que 3 et $2 \pm \sqrt{-5}$ sont irréductibles, et que $9 = 3\cdot 3 = (2+\sqrt{-5})\cdot(2-\sqrt{-5})$ sont deux décompositions distinctes. On voit dans cet exemple que les éléments irréductibles 3 et $2\pm\sqrt{-5}$ ne sont pas premiers (le détailler).

4.2. Aspects algorithmiques. Supposons que A admet une structure factorielle P. Il est en général facile d'évaluer l'application $\Pi_P\colon A^\times \times \mathbb{N}^{(P)} \to A^*$, car il ne s'agit que des multiplications. Par contre, l'application inverse $\Pi_P^{-1}\colon A^* \to A^\times \times \mathbb{N}^{(P)}$ peut être très difficile à calculer sur des exemples concrets ! On verra que c'est déjà très dur dans l'anneau $\mathbb{Z}$, problème qui nous occupera dans le chapitre XI.

Exercice 4.9. Vérifier que dans un anneau factoriel les formules données au début du chapitre,

$$\operatorname{pgcd}\bigl(u\textstyle\prod p^{\nu(p)}, u'\prod p^{\nu'(p)}\bigr) = \prod p^{\min(\nu(p),\nu'(p))} \quad \text{et}$$
$$\operatorname{ppcm}\bigl(u\textstyle\prod p^{\nu(p)}, u'\prod p^{\nu'(p)}\bigr) = \prod p^{\max(\nu(p),\nu'(p))},$$

définissent bien un pgcd et un ppcm. À noter en particulier que le choix de P spécifie un pgcd et un ppcm *préféré* et enlève ainsi l'ambiguïté notoire dans la définition d'un pgcd et d'un ppcm. Plus généralement, on peut définir $\operatorname{pref}\colon A^* \to A^*$ par $\operatorname{pref}(u\prod p^{\mu(p)}) = \prod p^{\mu(p)}$ ce qui choisit un représentant préféré dans chaque classe d'éléments associés. Cette construction a le bon goût d'être multiplicative, $\operatorname{pref}(xy) = \operatorname{pref}(x)\cdot\operatorname{pref}(y)$.

The source of all great mathematics is the special case, the concrete example.
It is frequent in mathematics that every instance of a concept of seemingly
great generality is in essence the same as a small and concrete special case.
Paul Halmos, *I Want to be a Mathematician*

PROJET IX

Algorithme de Gauss-Bézout et diviseurs élémentaires

Objectif. Nous souhaitons résoudre un *système d'équations linéaires* sur les entiers :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= y_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= y_2 \\ & \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= y_m \end{cases}$$

Ici les coefficients a_{ij} et y_i sont des entiers et l'on cherche les solutions x_i également dans les entiers. En algèbre linéaire on développe la théorie de ces systèmes sur un corps $\mathbb{K}$, et on apprend certaines méthodes pour leur résolution, notamment la méthode de Gauss rappelée plus bas. Dans notre situation nous devons adapter cet algorithme pour tenir compte du fait que l'on ne puisse pas toujours diviser par un pivot a_{ij}. On verra comment l'algorithme d'Euclide-Bézout résout ce problème. Cette observation aboutit au théorème des diviseurs élémentaires et sa version effective, l'algorithme de Gauss-Bézout. Ce résultat classique est déjà très intéressant en lui-même, et il résout en particulier notre système d'équations linéaires.

Remarquons en passant que le théorème des diviseurs élémentaires sera tout aussi intéressant à plus long terme car il se généralise à tout anneau euclidien, notamment l'anneau des polynômes $\mathbb{K}[X]$ sur un corps $\mathbb{K}$, ou plus généralement à tout anneau principal. Comme application importante on en déduira la classification des groupes abéliens finis, ou plus généralement des modules finiment engendrés sur un anneau principal. Ne vous inquiétez pas si ces termes ne vous parlent pas encore pour le moment : c'est juste le vocabulaire général pour les observations que nous dégagerons ici sur l'anneau des entiers $\mathbb{Z}$.

Aperçu de l'approche. Comme vous en avez l'habitude, la matrice $A = (a_{ij})_{j=1,\dots,n}^{i=1,\dots,m}$ et les vecteurs $x = (x_j)_{j=1,\dots,n}$ et $y = (y_i)_{i=1,\dots,m}$ permettent d'écrire notre système plus succinctement comme $Ax = y$. Ayant fixé la matrice $A \in \mathbb{Z}^{m\times n}$ et le vecteur $y \in \mathbb{Z}^m$, il s'agit de trouver les solutions $x \in \mathbb{Z}^n$ vérifiant $Ax = y$. C'est bien plus qu'une notation commode : cette structuration des données est le point de départ de tous les algorithmes pour la résolution de systèmes linéaires.

Remarquons d'abord que le problème devient trivial si la matrice A est *diagonale*, c'est-à-dire $a_{ij} = 0$ pour toute paire d'indices $i \neq j$. Dans ce cas nos équations $a_{ii}x_i = y_i$ sont découplées les unes des autres, ce qui ramène le calcul à l'anneau de base $\mathbb{Z}$: l'existence d'une solution $x_i \in \mathbb{Z}$ revient à une question de divisibilité $a_{ii} \mid y_i$, et dans le cas favorable l'unique solution est $x_i = y_i/a_{ii}$.

Signalons quelques cas exceptionnels évidents. Si $a_{ii} = y_i = 0$, alors on a bien $a_{ii} \mid y_i$ et tout $x_i \in \mathbb{Z}$ est solution de l'équation $a_{ii}x_i = y_i$. De manière analogue, si $m < n$, alors les variables $x_{m+1}, \dots, x_n \in \mathbb{Z}$ peuvent être choisies arbitrairement. Dans le cas contraire $m > n$ notre système diagonal admet une solution seulement si les coefficients $y_{n+1}, \dots, y_m \in \mathbb{Z}$ s'annulent. (Le détailler.)

Dans le cas général l'idée est de se ramener à un système diagonal, en passant de la matrice donnée A à une matrice diagonale $D = SAT$ où S et T sont des matrices inversibles, dites *matrices de passage*. L'algorithme de Gauss-Bézout nous permet de calculer de telles matrices D, S et T, et le théorème des diviseurs élémentaires affirme que la matrice diagonale D est unique dans un certain sens.

Sommaire

1. L'algorithme de Gauss-Bézout

1.1. Calcul matriciel. Fixons deux nombres naturels $m,n \in \mathbb{N}$ et posons $I = \{1,\dots,m\}$ ainsi que $J = \{1,\dots,n\}$. Une *matrice* de taille $m \times n$ à coefficient dans $\mathbb{K}$ est une famille $A = (a_{ij})$ d'éléments $a_{ij} \in \mathbb{K}$ indexés par $(i,j) \in I \times J$. Ce n'est rien autre qu'une application $a\colon I \times J \to \mathbb{K}$ notée $(i,j) \mapsto a_{ij}$. L'ensemble de ces matrices sera noté $\mathbb{K}^{m\times n}$ ou bien $\mathrm{Mat}(m \times n;\mathbb{K})$.

Notation. Dans la pratique une telle matrice A s'écrit comme un schéma rectangulaire, avec i indexant les lignes et j indexant les colonnes. Dans cette écriture les matrices $m \times 1$ sont les vecteurs colonnes, alors que les matrices $1 \times n$ sont les vecteurs lignes. À chaque matrice $A = (a_{ij})_{ij}$ de taille $m \times n$ on peut associer la matrice transposée $A^t = (a_{ij})_{ji}$ de taille $n \times m$.

Jusqu'ici $\mathbb{K}$ puis $\mathbb{K}^{m\times n}$ n'est qu'un ensemble sans structure spécifique. La théorie devient intéressante quand $\mathbb{K}$ est un anneau : dans ce cas on peut définir une addition et une multiplication :

$$+\colon \mathbb{K}^{m\times n} \times \mathbb{K}^{m\times n} \to \mathbb{K}^{m\times n}, \quad (A,B) \mapsto C = A+B \quad \text{avec } c_{ij} = a_{ij} + b_{ij}, \tag{1}$$

$$*\colon \mathbb{K}^{m\times n} \times \mathbb{K}^{n\times r} \to \mathbb{K}^{m\times r}, \quad (A,B) \mapsto C = A*B \quad \text{avec } c_{ik} = \textstyle\sum_{j=1}^{n} a_{ij}b_{jk}. \tag{2}$$

En particulier on obtient une action des matrices sur les vecteurs par l'application $*\colon \mathbb{K}^{m\times n} \times \mathbb{K}^n \to \mathbb{K}^m$ avec $(A,x) \mapsto y = Ax$ défini par $y_i = \sum_{j=1}^{n} a_{ij}x_j$. En plus on a la multiplication scalaire (à gauche)

$$\cdot\colon \mathbb{K} \times \mathbb{K}^{m\times n} \to \mathbb{K}^{m\times n}, \quad (\lambda,A) \mapsto B = \lambda A \quad \text{avec } b_{ij} = \lambda a_{ij}. \tag{3}$$

Toutes ces notions apparaissent naturellement en algèbre linéaire, où les matrices sont un outil formidable pour représenter les applications linéaires. On suppose connu ce contexte, et on se servira du langage associé sans rentrer dans les détails d'une révision plus complète.

Exercice/M 1.1 (structure additive). L'ensemble $\mathbb{K}^{m\times n}$ muni de l'addition (1) forme un groupe abélien. L'élément neutre est la matrice nulle, notée $0_{m\times n}$ ou 0 simplement. La multiplication scalaire fait de $\mathbb{K}^{m\times n}$ un espace vectoriel sur $\mathbb{K}$. Cette terminologie suppose que $\mathbb{K}$ est un corps. — Si $\mathbb{K}$ est un anneau on exprime le même constat par des mots différents : on dit plus prudemment que $\mathbb{K}^{m\times n}$ est un module sur l'anneau $\mathbb{K}$.

Exercice/M 1.2 (structure multiplicative). La multiplication (2) est associative et admet pour élément neutre à gauche la matrice identité $1_{m\times m}$, ainsi que pour élément neutre à droite la matrice identité $1_{n\times n}$. La multiplication est distributive sur l'addition. (La matrice identité $1_{n\times n}$ est aussi notée 1_n ou simplement 1 si la dimension n est claire par le contexte.)

Exercice/M 1.3 (structure d'anneau). L'ensemble $\mathbb{K}^{n\times n}$ des matrices carrées de taille $n \times n$ sur $\mathbb{K}$ muni de l'addition et de la multiplication définies ci-dessus forme un anneau. L'application $\mathbb{K} \to \mathbb{K}^{n\times n}$, $\lambda \mapsto \lambda 1_{n\times n}$ est un isomorphisme entre notre anneau de base $\mathbb{K}$ et le sous-anneau $\mathbb{K}1_{n\times n} = \{\lambda 1_{n\times n} \mid \lambda \in \mathbb{K}\}$. Dans le cas particulier $n = 1$ on retrouve l'isomorphisme évident $\mathbb{K} \cong \mathbb{K}^{1\times 1}$. Pour $n \geq 2$ l'anneau $\mathbb{K}^{n\times n}$ est non commutatif, même si l'anneau de base $\mathbb{K}$ l'est.

Puisque $\mathbb{K}^{n\times n}$ est un anneau (non commutatif), on peut appliquer le vocabulaire usuel. Rappelons en particulier la notion d'élément inversible, qui joue toujours un rôle très important :

Définition 1.4. On dit qu'une matrice $A \in \mathbb{K}^{n\times n}$ est *inversible* dans $\mathbb{K}^{n\times n}$ s'il existe une matrice $B \in \mathbb{K}^{n\times n}$ telle que $AB = BA = 1$. Dans ce cas l'élément B est unique, on l'appelle *l'inverse* de A, et on le note A^{-1}. Les éléments inversibles de $\mathbb{K}^{n\times n}$ forment un groupe, appelé *groupe linéaire* et noté $\mathrm{GL}(n;\mathbb{K})$ ou $\mathrm{GL}_n(\mathbb{K})$.

Tout ce que l'on vient de dire est vrai pour tout anneau $\mathbb{K}$, supposé associatif et unitaire, non forcément commutatif. Dans votre cours d'algèbre linéaire vous trouvez un développement beaucoup plus complet sous l'hypothèse que $\mathbb{K}$ est un corps, c'est-à-dire un anneau unitaire commutatif dans lequel tout élément non nul est inversible. Certains résultats s'étendent encore aux anneaux commutatifs :

Théorème 1.5 (existence et unicité du déterminant)**.** *Soit $\mathbb{K}$ un anneau commutatif unitaire. Pour tout $n \in \mathbb{N}$ il existe une et une seule application* $\det\colon \mathbb{K}^{n\times n} \to \mathbb{K}$, *appelée* déterminant, *qui soit alternée et multilinéaire par rapport aux colonnes et normée dans le sens que* $\det(1_{n\times n}) = 1_{\mathbb{K}}$. *Elle jouit des propriétés suivantes :*

(1) *Le déterminant se développe en la formule polynomiale* $\det A = \sum_{\sigma\in S_n} \mathrm{sign}(\sigma)\cdot a_{1,\sigma(1)}\cdot a_{2,\sigma(2)}\cdots a_{n,\sigma(n)}$ *où σ parcourt toutes les permutations dans le groupe symétrique S_n.*

(2) *Le déterminant est invariant par transposition, c'est-à-dire il satisfait* $\det(A^t) = \det(A)$. *Par conséquent il est également alterné et multilinéaire par rapport aux lignes.*

(3) *Le déterminant est multiplicatif dans le sens que* $\det(AB) = \det(A)\det(B)$. *Par restriction on obtient donc un homomorphisme de groupes* $\det\colon \mathrm{GL}_n(\mathbb{K}) \to \mathbb{K}^\times$, *manifestement surjectif.*

(4) *Une matrice A est inversible dans* $\mathbb{K}^{n\times n}$ *si et seulement si* $\det(A)$ *est inversible dans* $\mathbb{K}$. *(Ici « ⇒ » est claire ; pour « ⇐ » il existe une formule polynomiale qui exprime* A^{-1} *en fonction de A.)*

Si vous connaissez ce résultat pour les corps, vous êtes vivement invités à le redémontrer dans le cadre général des anneaux commutatifs. À noter que ce théorème n'est plus valable pour un anneau $\mathbb{K}$ non commutatif ; dégager donc bien les arguments de la preuve qui se servent de la commutativité.

Corollaire 1.6. *L'ensemble* $\mathrm{SL}_n(\mathbb{K}) = \{A \in \mathbb{K}^{n\times n} \mid \det(A) = 1\}$ *est un sous-groupe de* $\mathrm{GL}_n(K)$. □

Remarque 1.7. L'application $\det\colon \mathbb{K}^{n\times n} \to \mathbb{K}$ est multiplicative, mais pour $n \geq 2$ elle n'est pas additive : il ne s'agit pas d'un homomorphisme d'anneaux ! (Donner un contre-exemple de matrices 2×2.)

1.2. L'algorithme de Gauss-Bézout. Ce paragraphe présente l'algorithme de Gauss-Bézout pour transformer une matrice A en une matrice diagonale $D = SAT$. L'algorithme comme nous le décrivons est suffisamment efficace pour être intéressant dans la pratique ; on discutera l'implémentation plus bas.

Nous allons d'abord expliciter un sous-algorithme qui permet d'éliminer la première colonne de notre matrice A. Considérons la première ligne a_1 et la $i$ème ligne a_i : les éléments en tête sont $x = a_{11}$ et $y = a_{i1}$, respectivement. On calcule leur pgcd d avec des coefficients de Bézout $u, v \in \mathbb{Z}$ de sorte que

$$d = \mathrm{pgcd}(x,y) = ux + vy.$$

Ces données nous permettent de définir la matrice

$$M := \begin{pmatrix} u & v \\ -y/d & x/d \end{pmatrix}.$$

Elle est à coefficients entiers puisque d est un diviseur commun de x et y. Par sa construction elle vérifie $M\binom{x}{y} = \binom{d}{0}$ ainsi que $\det(M) = 1$. (Le vérifier.) Son inverse est d'ailleurs facile à expliciter :

$$M^{-1} = \begin{pmatrix} x/d & -v \\ y/d & u \end{pmatrix}.$$

On peut maintenant appliquer la matrice M aux lignes a_1 et a_i, ce qui revient à calculer $a'_1 \leftarrow ua_1 + va_i$ puis $a'_i \leftarrow -\frac{y}{d}a_1 + \frac{x}{d}a_i$. Dans la nouvelle matrice A' le coefficient a'_{i1} s'annule comme souhaité. En parcourant $i = 2, \ldots, m$ on peut ainsi éliminer la première colonne. Les mêmes arguments se transposent aux opérations sur les colonnes, ce qui permet d'éliminer la première ligne. En voici un exemple détaillé :

Exemple 1.8. Essayons de mettre sous forme diagonale la matrice

$$A_0 = \begin{pmatrix} 48 & 12 & 18 \\ 36 & 21 & 9 \end{pmatrix}.$$

Pour les coefficients $x = 48$ et $y = 36$ dans la première colonne on trouve $d = \mathrm{pgcd}(x,y) = 12$ avec des coefficients de Bézout $u = 1$ et $v = -1$ vérifiant $d = ux + vy$. Ceci nous mène à

$$M_0 := \begin{pmatrix} 1 & -1 \\ -3 & 4 \end{pmatrix} \quad \text{puis} \quad A_1 := M_0 A_0 = \begin{pmatrix} 12 & -9 & 9 \\ 0 & 48 & -18 \end{pmatrix}.$$

Nous éliminons ensuite la première ligne de la même façon. Pour les coefficients $x = 12$ et $y = -9$ dans la première ligne nous trouvons $d = 3$ ainsi que $u = 1$ et $v = 1$. Puisque nous effectuons maintenant les transformations sur les colonnes, ceci se traduit par multiplier A à droite :

$$M_1 := \begin{pmatrix} 1 & 3 & 0 \\ 1 & 4 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{donne} \quad A_2 := A_1 M_1 = \begin{pmatrix} 3 & 0 & 9 \\ 48 & 192 & -18 \end{pmatrix}.$$

Pour $x = 3$ et $y = 9$ on trouve $d = 3$ ainsi que $u = 1$ et $v = 0$, donc la transformation par

$$M_2 := \begin{pmatrix} 1 & 0 & -3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{donne} \quad A_3 := A_2 M_2 = \begin{pmatrix} 3 & 0 & 0 \\ 48 & 192 & -162 \end{pmatrix}.$$

On vient d'éliminer la première ligne, mais en contrepartie on a gâché un peu la première colonne. Restons optimiste et recommençons : pour $x = 3$ et $y = 48$ on trouve $d = 3$ ainsi que $u = 1$ et $v = 0$, donc

$$M_3 := \begin{pmatrix} 1 & 0 \\ -16 & 1 \end{pmatrix} \quad \text{donne} \quad A_4 := M_3 A_3 = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 192 & -162 \end{pmatrix}.$$

Nous avons finalement éliminé à la fois la première colonne *et* la première ligne. Nous pouvons passer à la sous-matrice qui reste : pour $x = 192$ et $y = -162$ nous trouvons $d = 6$ ainsi que $u = 11$ et $v = 13$:

$$M_4 := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 11 & 27 \\ 0 & 13 & 32 \end{pmatrix} \quad \text{donne} \quad A_5 := A_4 M_4 = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 6 & 0 \end{pmatrix}.$$

Ceci termine l'algorithme. Mettant tout ensemble, on obtient ainsi les matrices de passage

$$S := M_3 M_0 = \begin{pmatrix} 1 & -1 \\ -19 & 20 \end{pmatrix} \quad \text{et} \quad T := M_1 M_2 M_4 = \begin{pmatrix} 1 & -6 & -15 \\ 1 & 5 & 12 \\ 0 & 13 & 32 \end{pmatrix} \quad \text{vérifiant} \quad SAT = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 6 & 0 \end{pmatrix}.$$

Remarque 1.9 (matrices de passage)**.** Afin de construire les matrices de passage S et T lors du calcul, on commence par les matrices $S_0 = 1_{m\times m}$ et $T_0 = 1_{n\times n}$ vérifiant $A_0 = S_0 A T_0$.

- Une opération sur les lignes correspond à une multiplication à gauche : $A_{k+1} := M_k A_k$ avec une matrice inversible M_k comme ci-dessus. On pose $S_{k+1} := M_k S_k$ et $T_{k+1} := T_k$.
- Une opération sur les colonnes correspond à une multiplication à droite : $A_{k+1} := A_k M_k$avec une matrice inversible M_k comme ci-dessus. On pose $S_{k+1} := S_k$ et $T_{k+1} := T_k M_k$.

Dans les deux cas on part de $A_k = S_k A T_k$ et on assure que $A_{k+1} = S_{k+1} A T_{k+1}$. Chaque transformation correspond à une matrice M_k de déterminant $+1$. Ceci assure que les matrices de passages S_k et T_k sont également de déterminant $+1$. L'algorithme se termine avec une matrice diagonale $D = A_k$ pour un certain k. Avec $S = S_k$ et $T = T_K$ on obtient $D = SAT$ comme souhaité.

En guise de résumé, voici la version concise de l'algorithme de Gauss-Bézout. Pour le moment nous entendons par *forme normale* une forme diagonale quelconque. Ceci sera précisé plus loin par une condition supplémentaire.

Algorithme IX.4 Algorithme de Gauss-Bézout

Entrée: une matrice $A \in \mathbb{Z}^{m\times n}$

Sortie: trois matrices $D \in \mathbb{Z}^{m\times n}$, $S \in \mathbb{Z}^{m\times m}$, $T \in \mathbb{Z}^{n\times n}$ telles que $D = SAT$

Garanties: D est sous forme normale et S et T sont inversibles de déterminant 1.

Initialiser $D \leftarrow A$ et $S \leftarrow 1_{m\times m}$ et $T \leftarrow 1_{n\times n}$	// On assure $D_0 = S_0 A T_0$.
tant que D n'est pas encore sous forme normale **faire**	
Effectuer une transformation sur deux lignes ou deux colonnes	// On s'approche du résultat souhaité.
Mettre à jour les matrices de passages S et T	// $D_k = S_k A T_k \Rightarrow D_{k+1} = S_{k+1} A T_{k+1}$.
fin tant que	
retourner (D, S, T)	// $D_k = S_k A T_k$ est sous forme normale.

1.3. Preuve de correction. L'algorithme de Gauss-Bézout a bien marché sur l'exemple précédent. Montrons que l'approche réussit toujours :

Proposition 1.10. *Les opérations élémentaires sur les lignes et les colonnes permettent de transformer toute matrice $A \in \mathbb{K}^{m\times n}$ en une matrice A' telle que $a'_{i1} = a'_{1j} = 0$ pour tout $i, j \geq 2$.*

DÉMONSTRATION. On descend d'abord la première colonne ; les opérations sur les lignes décrites ci-dessus permettent d'obtenir $a_{i1} = 0$. Ensuite on traverse la première ligne pour obtenir $a_{1j} = 0$. Or, ces dernières opérations ajoutent des multiples des colonnes $j \geq 2$ à la première colonne. Par conséquent on ne préserve en général pas la condition $a_{i1} = 0$ et on est obligé de repasser la première colonne, puis la première ligne, etc.

Heureusement ce processus se termine après un nombre fini d'itérations : dans chaque opération le coefficient a_{11} est remplacé par un de ses diviseurs, a savoir $\operatorname{pgcd}(a_{11}, a_{ij})$ où a_{ij} est le coefficient que l'on

cherche à annuler. Ceci ne peut modifier la valeur de a_{11} qu'un nombre fini de fois. S'il ne change plus, ceci veut dire que a_{11} divise tous les coefficients a_{i1} de la première colonne ainsi que tous les coefficients a_{1j} de la première ligne. On arrive ainsi au cas simple de l'algorithme usuel sur un corps : avec $d = a_{11}$, $u = 1$, $v = 0$ la transformation revient à calculer $A'_i \leftarrow A_i - \frac{a_{i1}}{a_{11}} A_1$ sans changer la ligne A_1, et il en est de même pour les opérations sur les colonnes. Après ce dernier passage on obtient l'annulation souhaitée. □

Proposition 1.11. *Les opérations élémentaires sur les lignes et les colonnes permettent de transformer toute matrice $A \in \mathbb{K}^{m\times n}$ en une matrice diagonale D, c'est-à-dire $d_{ij} = 0$ pour toute paire d'indices $i \neq j$. En plus on peut assurer la divisibilité successive $d_{11} \mid d_{22} \mid d_{33} \mid \dots$ des termes diagonaux.*

DÉMONSTRATION. Supposons que $m \leq n$. Pour $k = 1, \dots, m$ on applique la proposition précédente à la sous-matrice indexée par des paires (i, j) avec $i, j \geq k$. Pour $k = 1$ on élimine ainsi la première ligne et la première colonne. Pour $k = 2$ on élimine la seconde ligne et la seconde colonne de la sous-matrice, et ainsi de suite. Le résultat final est une matrice D dont tous les coefficients hors de la diagonale s'annulent.

Étant donné deux termes diagonaux $x, y \in \mathbb{Z}$ on calcule à nouveau $d := \text{pgcd}(x, y) = ux + vy$ avec des coefficients de Bézout $u, v \in \mathbb{Z}$. On a $e := \text{ppcm}(x, y) = xy/d$, et on vérifie aisément que

$$\begin{pmatrix} u & v \\ -y/d & x/d \end{pmatrix} \cdot \begin{pmatrix} x & 0 \\ 0 & y \end{pmatrix} \cdot \begin{pmatrix} 1 & -vy/d \\ 1 & ux/d \end{pmatrix} = \begin{pmatrix} d & 0 \\ 0 & e \end{pmatrix}.$$

Les deux matrices de passage sont inversibles car de déterminant 1. En traversant ainsi toute la diagonale on peut assurer que d_{11} soit le pgcd de tous les termes diagonaux. Ensuite on réitère pour assurer que d_{22} soit le pgcd de tous les termes diagonaux suivants, et ainsi de suite. □

1.4. Implémentation. Dans le développement mathématique nous avons utilisé certaines matrices M qui représentent des opérations sur les lignes ($A' \leftarrow MA$) ou les colonnes ($A' \leftarrow AM$). On pourrait l'implémenter littéralement, c'est-à-dire, construire la matrice M puis faire appel à la multiplication des matrices. Or, la matrice M est très creuse : c'est presque la matrice identité, avec seulement quatre coefficients potentiellement non triviaux. Nous allons donc implémenter les transformations élémentaires par deux fonctions spécialisées comme suit :

```
void gauss_gauche( const Integer& a, const Integer& b,
                   const Integer& c, const Integer& d,
                   Matrix<Integer>& mat, int i, int j, int k=1 );

void gauss_droite( const Integer& a, const Integer& b,
                   const Integer& c, const Integer& d,
                   Matrix<Integer>& mat, int i, int j, int k=1 );
```

Ici la matrice $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ opère sur les lignes/colonnes i et j de la matrice A. Par souci d'efficacité on songe déjà au cas d'une réduction plus évoluée, où les $k-1$ premières lignes et colonnes sont déjà diagonalisées et ne jouent plus de rôle. Puisqu'il ne sert à rien de manipuler des zéros, les opérations ci- dessus ne s'appliquent qu'à la sous-matrice $i, j \geq k$. Ainsi les fonctions suivantes annulent la ligne ou colonne k :

```
bool gauss_colonne( Matrix<K>& mat, Matrix<K>& s, Dim k );
bool gauss_ligne( Matrix<K>& mat, Matrix<K>& t, Dim k );
```

Exercice/P 1.12. En suivant le modèle `gauss-bezout.cc`, implémenter efficacement l'algorithme de Gauss-Bézout en une fonction

```
void gauss_bezout( Matrix<K>& mat, Matrix<K>& s, Matrix<K>& t );
```

Ici `mat` contient la matrice initiale qui sera transformée à fur et à mesure en une matrice diagonale, en modifiant directement la matrice `mat`. Les matrices `s` et `t` sont initialisées par les matrices identités convenables. On fait agir les opérations à gauche sur `s` et les opérations à droite sur `t` comme ci-dessus.

Exercice/P 1.13. Testez votre implémentation sur des matrices variées. Comment peut-on vérifier efficacement les résultats ? Est-ce que votre implémentation est suffisamment efficace pour des matrices denses aléatoires de taille 10×10 ? 20×20 ? 50×50 ? 100×100 ? Quels phénomènes observez-vous ?

Remarque 1.14 (résolution d'un système linéaire)**.** L'algorithme de Gauss-Bézout résout notre problème initial d'un système $Ax = y$. On passe à la matrice diagonale $D = SAT$, puis le système diagonal $D\hat{x} = \hat{y}$ se résout aisément. Ici on pose $\hat{y} = Sy$, et la solution x s'obtient ensuite par $x = T\hat{x}$.

1.5. Calcul efficace du déterminant. Rappelons que dans le théorème du déterminant énoncé ci-dessus, l'unicité et l'existence du déterminant sont établies par la formule explicite

$$\det A = \sum_{\sigma\in S_n} \operatorname{sign}(\sigma)\cdot a_{1,\sigma(1)}\cdot a_{2,\sigma(2)}\cdots a_{n,\sigma(n)}.$$

Traduite littéralement, cette formule donne un algorithme de complexité $n!$ ce qui n'est pas du tout efficace pour n grand. (Calculer 10! ou 20! voire 50! pour vous en convaincre.) Le développement récursif par rapport à une ligne ou une colonne n'est qu'une reformulation de cette approche, et donc aussi inefficace. Le seul cas lucratif est le développement par rapport à une ligne ou une colonne *creuse*, c'est-à-dire contenant peu de coefficients non nuls.

Dans le cas général d'une grande matrice dense, l'algorithme de Gauss-Bézout se révèle plus avantageux, à savoir de complexité d'environ n^3 opérations dans $\mathbb{K}$. Un problème notoire est « l'explosion des coefficients » lors des calculs intermédiaires. Soulignons que le nombre d'opérations dans $\mathbb{K}$ n'est qu'une indication grossière : les opérations dans $\mathbb{Z}$ deviennent plus coûteuses en temps et en mémoire quand les coefficients grandissent. Quelques exemples sur des matrices aléatoires vous convaincront que ce phénomène est bien réel, même si la matrice initiale n'a que des coefficients de petite taille.

Bien sûr, une explosion des coefficients ne peut se produire que dans un anneau infini. Une astuce éprouvée est donc de réduire modulo un nombre premier p afin d'effectuer le calcul dans le corps fini $\mathbb{Z}_p$. Pour reconstituer le résultat dans $\mathbb{Z}$ on rassemble l'information modulo plusieurs nombres premiers $p_1, p_2, \ldots$. Pour un développement de cette idée voir Gathen-Gerhard [**11**], §5.5.

1.6. Le théorème des diviseurs élémentaires. D'après ce qui précède on sait maintenant transformer une matrice donnée A en une matrice diagonale D : l'algorithme de Gauss-Bézout ci-dessus en explicite une démarche. Il y a pourtant de nombreux choix : d'autres manières de procéder sont imaginables et leurs matrices de passages seront très différentes. Le résultat suivant est donc tout à fait remarquable : il dit que la matrice diagonale qui en résulte est toujours la même :

Théorème 1.15 (le théorème des diviseurs élémentaires)**.** *Pour toute matrice $A \in \mathrm{Mat}(m\times n;\mathbb{Z})$ il existe des matrices inversibles $S \in \mathrm{SL}_m(\mathbb{Z})$ et $T \in \mathrm{SL}_n(\mathbb{Z})$ telles que la matrice produit $D = SAT$ soit diagonale et vérifie la divisibilité successive $d_{11} \mid d_{22} \mid d_{33} \mid \ldots$ des termes diagonaux. Dans ce cas ces termes sont uniques aux signes près : pour toute autre diagonalisation $D' = S'AT'$ avec $S' \in \mathrm{SL}_m(\mathbb{Z})$ et $T' \in \mathrm{SL}_n(\mathbb{Z})$ satisfaisant la condition $d'_{11} \mid d'_{22} \mid d'_{33} \mid \ldots$ on a $d'_{ii} = \pm d_{ii}$ pour tout i.*

Définition 1.16. Une matrice $D \in \mathbb{Z}^{m\times n}$ est sous *forme normale* si elle est diagonale et ses termes diagonaux vérifient $d_{11} \mid d_{22} \mid d_{33} \mid \ldots$. Le théorème des diviseurs élémentaires dit que toute matrice $A \in \mathbb{Z}^{m\times n}$ peut être mise sous forme normale. On appelle *diviseurs élémentaires* de A la suite $d_{11} \mid d_{22} \mid d_{33} \mid \ldots$ dont l'existence et l'unicité (aux signes près) sont assurées par le théorème précédent.

Remarque 1.17. Pour la fonction `pgcd` nous avons tacitement fait usage de notre convention : *le* pgcd de deux entiers est entendu comme le pgcd *positif*. Ainsi la matrice D retournée par notre algorithme satisfait à la condition $d_{11} \mid d_{22} \mid d_{33} \mid \ldots$ avec des termes diagonaux *positifs*, à l'exception éventuelle du dernier. Ainsi le signe du déterminant est retenu dans le tout dernier terme diagonal.

1.7. Unicité du résultat. Dans l'algorithme de Gauss-Bézout on a plusieurs choix : déjà les coefficients de Bézout utilisés à chaque étape ne sont pas uniques, puis l'ordre par lequel on effectue les opérations n'est pas canonique. Les matrices de passages S et T obtenues à la fin dépendent de ces choix et ne sont pas du tout uniques. On pourrait même imaginer des approches totalement différentes pour mettre une matrice A sous une forme diagonale. Il n'y a donc a priori aucune raison de croire que le résultat soit canonique. Des exemples simples, comme le suivant, montrent que les termes diagonaux peuvent changer :

$$A = \begin{pmatrix} 4 & 0 \\ 0 & 6 \end{pmatrix} \quad \text{se transforme en} \quad SAT = \begin{pmatrix} 2 & 0 \\ 0 & 12 \end{pmatrix} \quad \text{avec} \quad S = \begin{pmatrix} 2 & -1 \\ -3 & 2 \end{pmatrix} \quad \text{et} \quad T = \begin{pmatrix} 1 & 3 \\ 1 & 4 \end{pmatrix}.$$

Il est donc tout à fait remarquable que les diviseurs élémentaires soient essentiellement uniques ! Pour la preuve nous allons employer ce merveilleux outil qu'est le déterminant. Supposons que A est une matrice de taille $m\times n$. Pour $I' \subset I$ et $J' \subset J$ de cardinal $|I| = |J| = k$ nous définissons la sous-matrice $A|_{I'\times J'} = (a_{ij})_{i\in I', j\in J'}$ par restriction des indices.

Définition 1.18. On note $\Delta_k(A)$ le pgcd des déterminants de toutes les sous-matrices de taille $k \times k$ de A.

Lemme 1.19. *$\Delta_k(A)$ ne change pas lors d'une transformation élémentaire sur les lignes ou les colonnes.*

DÉMONSTRATION. Il suffit de le prouver pour une transformation sur les lignes. Regardons une matrice $M = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) \in \mathrm{SL}_2(\mathbb{Z})$ agissant sur les lignes i et j. Si une sous-matrice B ne contient ni la ligne i ni la ligne j, alors la sous-matrice B et son déterminant $\det(B)$ ne changent pas. Si une sous-matrice B contient les deux lignes, alors la matrice B change mais non son déterminant.

Le cas intéressant est celui où une sous-matrice B contient la ligne i mais non la ligne j. Soit C la sous-matrice correspondante où l'on remplace la ligne i par la ligne j. Notons $x = \det(B)$ et $y = \det(C)$ leurs déterminants. Après transformation nous obtenons deux sous-matrices modifiées B' et C' avec déterminants $x' = \det(B')$ et $y' = \det(C')$. Par multilinéarité du déterminant on trouve $x' = ax + by$ et $y' = cx + dy$, donc les diviseurs communs de x et y sont aussi des diviseurs communs de x' et y'. La réciproque est également vraie puisque M est inversible sur $\mathbb{Z}$. Ceci veut dire que $\mathrm{pgcd}(x,y) = \mathrm{pgcd}(x',y')$. On conclut que $\Delta_k(A)$ ne change pas lors d'une transformation élémentaire, comme énoncé. □

Lemme 1.20. *Le groupe $\mathrm{SL}_n(\mathbb{Z})$ est engendré par les sous-groupes $\mathrm{SL}_2^{ij}(\mathbb{Z})$ avec $1 \leq i < j \leq n$.*

DÉMONSTRATION. Précisons d'abord la notation : une matrice $A \in \mathbb{Z}^{n\times n}$ appartient à $\mathrm{SL}_2^{ij}(\mathbb{Z})$ si la sous-matrice $\left(\begin{smallmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{smallmatrix}\right)$ appartient à $\mathrm{SL}_2(\mathbb{Z})$ alors que tous les autres coefficients sont ceux de la matrice identité $1_{n\times n}$. Ce sont précisément les matrices qui apparaissent dans l'algorithme de Gauss-Bézout lors des transformations élémentaires. L'énoncé découle de l'application de cet algorithme à une matrice $A \in \mathrm{SL}_n(\mathbb{Z})$ pour la transformer en une matrice diagonale $D = SAT$. Par construction les matrices S et T sont produits de matrices dans $\mathrm{SL}_2^{ij}(\mathbb{Z})$ avec $1 \leq i < j \leq n$. Puisque $D \in \mathrm{SL}_n(\mathbb{Z})$ on a $d_{ii} = \pm 1$, et avec notre convention de signes on a même $D = 1_{n\times n}$. □

PREUVE DU THÉORÈME. Supposons que A est une matrice diagonale de taille $m \times n$, disons avec $m \leq n$, vérifiant $a_{11} \mid a_{22} \mid \cdots \mid a_{mm}$. Cette propriété entraîne que $\Delta_1(A) = \pm a_{11}$, puis $\Delta_2(A) = \pm a_{11}a_{22}$, … jusqu'à $\Delta_m(A) = \pm a_{11}a_{22}\cdots a_{mm}$. Supposons que l'on transforme A en une matrice diagonale $A' = SAT$ avec $S \in \mathrm{SL}_m(\mathbb{Z})$ et $T \in \mathrm{SL}_n(\mathbb{Z})$. D'après le lemme 1.20 ceci revient à effectuer des transformations élémentaires sur les lignes et les colonnes. Ceci ne change pas les invariants $\Delta_1, \ldots, \Delta_m$, ce qui permet de conclure que $a_{11} = \pm a'_{11}$, puis $a_{22} = \pm a'_{22}$, … jusqu'à $a_{mm} = \pm a'_{mm}$, comme souhaité. □

2. Applications aux groupes abéliens

2.1. Groupes abéliens libres. Pour tout groupe $(G,+)$ on a une unique application $\sigma\colon \mathbb{Z} \times G \to G$, notée $(\lambda, a) \mapsto \lambda a$, vérifiant $0a = 0$ puis $(\lambda+1)a = \lambda a + a$ pour tout $\lambda \in \mathbb{Z}$. On en déduit que $1a = a$ ainsi que $(\lambda + \lambda')a = \lambda a + \lambda' a$ et $(\lambda\lambda')a = \lambda(\lambda' a)$. Par contre, l'application σ vérifie $\lambda(a+b) = \lambda a + \lambda b$ pour tout $\lambda \in \mathbb{Z}$ et $a, b \in G$ si et seulement si G est abélien. (Exercice.) En termes savants on dit qu'un groupe abélien est un *module* sur l'anneau $\mathbb{Z}$.

Définition 2.1. Soit $(G,+)$ un groupe abélien et soit $(g_i)_{i\in I}$ une famille d'éléments $g_i \in G$ indexés par $i \in I$. Une *combinaison linéaire* (sur $\mathbb{Z}$) est une somme $\sum_{i\in I} \lambda_i g_i$ avec des coefficients entiers $\lambda_i \in \mathbb{Z}$. Si l'ensemble I est infini nous ajoutons toujours la condition que seul un nombre fini de coefficients λ_i soient non nuls. (La sommation sur une infinité de termes non nuls n'a pas de sens.)

Définition 2.2. La famille $(g_i)_{i\in I}$ est *génératrice* pour le groupe G si tout élément $g \in G$ s'écrit comme une combinaison linéaire $g = \sum_{i\in I} \lambda_i g_i$ avec $\lambda_i \in \mathbb{Z}$. On dit que G est *finiment engendré* s'il admet une famille génératrice finie $(g_1, \ldots, g_n)$.

Définition 2.3. La famille $(g_i)_{i\in I}$ dans un groupe abélien G est *libre* si la seule combinaison linéaire nulle $\sum_{i\in I} \lambda_i g_i = 0$ est la somme triviale avec $\lambda_i = 0$ pour tout $i \in I$. La famille $(g_i)_{i\in I}$ est une *base* de G si elle est génératrice et libre. Ceci équivaut à dire que tout élément $g \in G$ s'écrit de manière unique comme $g = \sum_{i\in I} \lambda_i g_i$ avec $\lambda_i \in \mathbb{Z}$. Le groupe G est *libre* s'il admet une base.

Exemple 2.4. Le groupe $\mathbb{Z}$ est libre à base 1 (ou -1). Pour tout $m \in \mathbb{N}$ le groupe $\mathbb{Z}^m$ est libre : les éléments $e_i \in \mathbb{Z}^m$ avec $e_{ij} = 0$ pour $j \neq i$ et $e_{ii} = 1$ forment une base, dite *base canonique*.

Exemple 2.5. Pour $n \geq 2$ le groupe quotient $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$ n'est pas libre. Un élément $\bar{a} \in \mathbb{Z}_n$ est générateur si et seulement si $\text{pgcd}(a,n) = 1$. Il n'est pas libre car $n\bar{a} = 0\bar{a} = 0$. (Que dire des cas $n = 1$ et $n = 0$?)

Exemple 2.6. Soit I un ensemble et soit $\mathbb{Z}^{(I)}$ l'ensemble des applications $I \to \mathbb{Z}$ a support fini. C'est un groupe libre : comme base on prendra les applications $e_i \colon I \to \mathbb{Z}$ avec $e_i(j) = 0$ pour $j \neq i$ et $e_i(i) = 1$.

Proposition 2.7. *Un groupe abélien G est libre si et seulement s'il est isomorphe à un groupe $\mathbb{Z}^{(I)}$.*

DÉMONSTRATION. Si G est libre, alors il existe une base $(g_i)_{i\in I}$ et l'application $f \colon \mathbb{Z}^{(I)} \to G$ définie par $(\lambda_i)_{i\in I} \to \sum_{i\in I} \lambda_i g_i$ est un isomorphisme de groupes avec $e_i \mapsto g_i$. Réciproquement, s'il existe un isomorphisme de groupes $f \colon \mathbb{Z}^{(I)} \to G$, alors la famille $(g_i)_{i\in I}$ avec $g_i = f(e_i)$ est une base de G. □

2.2. Applications linéaires. Un homomorphisme $f \colon G \to H$ entre deux groupes abéliens est une application vérifiant $f(a+b) = f(a) + f(b)$ pour tout $a, b \in G$. Dans ce cas elle vérifie automatiquement $f(\lambda a) = \lambda f(a)$ pour tout $\lambda \in \mathbb{Z}$ et $a \in G$, et plus généralement $f(\sum_i \lambda_i a_i) = \sum_i \lambda_i f(a_i)$ pour $\lambda_i \in \mathbb{Z}$ et $a_i \in G$. (Exercice.) Au lieu d'homomorphismes de groupes abéliens on peut donc parler d'applications $\mathbb{Z}$-linéaires (ou encore d'homomorphismes de $\mathbb{Z}$-modules).

Proposition 2.8. *Soit G un groupe abélien libre à base $(g_i)_{i\in I}$. Étant donné un groupe abélien H est une famille $(h_i)_{i\in I}$ d'éléments $h_i \in H$, il existe un unique homomorphisme de groupes $f \colon G \to H$ vérifiant $f(g_i) = h_i$ pour tout $i \in I$.*

DÉMONSTRATION. *Unicité.* — Supposons que $f, f' \colon G \to H$ vérifient $f(g_i) = f'(g_i) = h_i$. Puisque $(g_i)_{i\in I}$ est une famille génératrice, tout élément $g \in G$ s'écrit comme $g = \sum_{i\in I} \lambda_i g_i$, donc $f(g) = f(\sum_i \lambda_i g_i) = \sum_i \lambda_i f(g_i) = \sum_i \lambda_i f'(g_i) = f'(\sum_i \lambda_i g_i) = f'(g)$.

Existence. — On définit $f \colon G \to H$ pour $g = \sum_{i\in I} \lambda_i g_i$ par $f(g) = \sum_{i\in I} \lambda_i h_i$. Puisque $(g_i)_{i\in I}$ est une base, tout élément $g \in G$ s'écrit ainsi de manière unique, ce qui assure que f est bien définie. L'application f est manifestement un homomorphisme de groupe qui vérifie $f(g_i) = h_i$, comme souhaité. □

Corollaire 2.9. *Soient G un groupe abélien libre à base $(g_1, \ldots, g_n)$ et H un groupe abélien libre à base $(h_1, \ldots, h_m)$. À tout homomorphisme de groupe $f \colon G \to H$ on peut associer une unique matrice $A \in \mathbb{Z}^{m\times n}$ de sorte que $f(g_j) = \sum_{i=0}^{m} a_{ij} h_i$ pour tout $j = 1, \ldots, n$. Réciproquement à toute matrice $A \in \mathbb{Z}^{m\times n}$ on peut associer un unique homomorphisme de groupe $f \colon G \to H$ vérifiant cette formule.* □

Nous regarderons dans la suite seulement des groupes abéliens finiment engendrés. C'est une restriction naturelle si l'on veut étudier des questions algorithmiques. Mais aussi mathématiquement c'est une classe beaucoup plus maniable et très importante.

Proposition 2.10. *Il existe un isomorphisme de groupes $\mathbb{Z}^n \cong \mathbb{Z}^m$ si et seulement si $n = m$.*

DÉMONSTRATION. Supposons par absurde qu'il existe un isomorphisme $f \colon \mathbb{Z}^n \xrightarrow{\sim} \mathbb{Z}^m$ pour $n > m$. Soit $A \in \mathbb{Z}^{m\times n}$ la matrice qui représente f dans les bases canoniques de $\mathbb{Z}^n$ et $\mathbb{Z}^m$. L'algorithme de Gauss-Bézout transforme A en une matrice diagonale $D = SAT$. Puisque $n > m$, la dernière colonne de D est nulle, et donc $De_n = 0$. Ainsi $A = S^{-1}DT^{-1}$ a aussi un noyau non trivial, car Te_n est envoyé sur 0. Ceci contredit l'hypothèse que f était un isomorphisme. □

Corollaire 2.11. *Si G est un groupe abélien libre avec deux bases $(g_1, \ldots, g_n)$ et $(h_1, \ldots, h_m)$ alors $n = m$. Ceci permet de définir le* rang *d'un groupe abélien libre G comme le cardinal d'une de ses bases.* □

Remarque 2.12. Vous connaissez le résultat analogue pour les espaces vectoriels sur un corps, ce qui permet de définir la *dimension*, notion puissante et omniprésente en algèbre linéaire. Pour les groupes abéliens (les modules sur $\mathbb{Z}$) il faut se restreindre aux groupes abéliens *libres* pour parler de bases. Puis la même question se pose : la notion de rang est-elle bien définie ? Nous avons choisi ici une preuve qui tire profit de l'algorithme de Gauss-Bézout.

L'énoncé se généralise à tout anneau commutatif unitaire A. Dans cette généralité la preuve ne s'applique pas telle quelle, parce que nous n'avons plus l'algorithme de Gauss-Bézout à notre disposition. Si A est intègre on peut passer à son corps des fractions. Si A n'est pas intègre on peut quotienter par un idéal maximal I : comme $F = A/I$ est un corps la preuve ci-dessus nous donne à nouveau le résultat souhaité. Si vous connaissez ces outils, vous pouvez tenter une preuve.

2.3. Sous-groupes de $\mathbb{Z}^m$. Dans ce paragraphe nous allons classifier les sous-groupes de $\mathbb{Z}^m$. Notre but sera d'abord de comprendre quels sous-groupes sont possibles, et ensuite de décrire comment ils sont plongés dans le groupe ambiant $\mathbb{Z}^m$.

Lemme 2.13. *Tout sous-groupe $H \subset \mathbb{Z}^m$ est libre et* $\operatorname{rang} H \leq m$.

DÉMONSTRATION. Nous allons établir le résultat par récurrence sur m. Pour $m = 0$ on n'a rien à montrer : le seul sous-groupe $H = \mathbb{Z}^0 = \{0\}$ est libre ayant la famille vide pour base. Si $m = 1$ nous avons un sous-groupe $H \subset \mathbb{Z}$: soit $H = \{0\}$ soit $H = \mathbb{Z}a$ pour un élément $a \in H \smallsetminus \{0\}$ de plus petite norme. (L'anneau $\mathbb{Z}$ est principal, voir la proposition 2.12.) Dans ce dernier cas la famille (a) forme une base.

Pour $m \geq 2$ soit $p\colon \mathbb{Z}^m \to \mathbb{Z}$, $(x_1, \dots, x_m) \mapsto x_m$, la projection sur la dernière coordonnée. Nous pouvons identifier $\mathbb{Z}^{m-1}$ avec le noyau $\ker(p)$ via l'application $(x_1, \dots, x_{m-1}) \mapsto (x_1, \dots, x_{m-1}, 0)$. La projection p nous permet de construire le sous-groupe $K = \ker(p|_H) = H \cap \mathbb{Z}^{m-1}$ de $\ker(p) \cong \mathbb{Z}^{m-1}$. Par hypothèse de récurrence K admet une base $v_1, \dots, v_{n-1} \in K$ de cardinal $n - 1 \leq m - 1$. L'image $p(H) \subset \mathbb{Z}$ est un sous-groupe de $\mathbb{Z}$. Si $p(H) = 0$ alors $H = K$ et il n'y a plus rien à montrer. Sinon $p(H) = \mathbb{Z}a$ pour un élément $a \in p(H) \smallsetminus \{0\}$ de plus petite norme. Soit $v_n \in H$ un élément tel que $p(v_n) = a$. Nous affirmons que $v_1, \dots, v_{n-1}, v_n$ est une base de H.

C'est une famille génératrice. — Pour tout $v \in H$ nous avons $p(v) = \lambda_n a$ avec $\lambda_n \in \mathbb{Z}$. Ainsi $v - \lambda_n v_n \in K$, et donc $v - \lambda_n v_n = \sum_{i=0}^{n-1} \lambda_i v_i$ puisque K est engendré par $v_1, \dots, v_{n-1}$ par hypothèse de récurrence.

C'est une famille libre. — Si $\sum_{i=0}^{n} \lambda_i v_i = 0$ alors $0 = p(\sum_{i=0}^{n} \lambda_i v_i) = \lambda_n a$ donc $\lambda_n = 0$. Ensuite $\sum_{i=0}^{n-1} \lambda_i v_i = 0$ entraîne $\lambda_1 = \cdots = \lambda_{n-1} = 0$ parce que la famille $v_1, \dots, v_{n-1}$ est libre par hypothèse de récurrence. □

Théorème 2.14. *Soit $H \subset \mathbb{Z}^m$ un sous-groupe. Alors il existe*

(1) *une base $b_1, \dots, b_m$ de $\mathbb{Z}^m$ et*

(2) *un entier r avec $0 \leq r \leq m$ et*

(3) *des entiers $e_1, \dots, e_r \geq 1$ vérifiant $e_1 \mid \cdots \mid e_r$*

tels que $e_1 b_1, \dots, e_r b_r$ soit une base de H. La suite des entiers $e_1, \dots, e_r$ est uniquement déterminée par H et on les appelle les diviseurs élémentaires *du sous-groupe $H \subset \mathbb{Z}^m$.*

DÉMONSTRATION. *Existence.* — Le lemme précédent nous assure l'existence d'une famille génératrice finie $v_1, \dots, v_n$. (Il n'est pas nécessaire de la supposer libre.) Ceci permet de définir une application $\mathbb{Z}$-linéaire $f\colon \mathbb{Z}^n \to \mathbb{Z}^m$ par $(\lambda_1, \dots, \lambda_n) \mapsto \lambda_1 v_1 + \cdots + \lambda_n v_n$. La matrice $A \in \mathbb{Z}^{m \times n}$ qui représente f est formée par les colonnes $v_1, \dots, v_n$. L'algorithme de Gauss-Bézout transforme A en une matrice diagonale $D = SAT$. Notons $e_1, \dots, e_r$ ses termes diagonaux non nuls et considérons $A = S^{-1} D T^{-1}$. Les colonnes $b_1, \dots, b_m$ de S^{-1} forment une base de $\mathbb{Z}^m$. Visiblement, les éléments $e_1 b_1, \dots, e_r b_r$ forment une base de l'image de A. On conclut que c'est une base de H, comme énoncé.

Unicité. — L'application $f\colon \mathbb{Z}^r \to \mathbb{Z}^m$ définie par $(\lambda_1, \dots, \lambda_r) \mapsto \lambda_1 e_1 b_1 + \cdots + \lambda_r e_r b_r$ est un isomorphisme entre $\mathbb{Z}^r$ et son image $H \subset \mathbb{Z}^m$. Soit $A \in \mathbb{Z}^{m \times n}$ la matrice associée par rapport aux bases canoniques de $\mathbb{Z}^n$ et $\mathbb{Z}^m$. Par construction ses diviseurs élémentaires sont $e_1, \dots, e_r, 0 \dots, 0$.

Supposons que $b'_1, \dots, b'_m$ est une autre base de $\mathbb{Z}^m$ telle que $e'_1 b'_1, \dots, e'_r b'_r$ soit une base de H avec $e'_1, \dots, e'_r \geq 1$ vérifiant $e'_1 \mid \cdots \mid e'_r$. Si $A' \in \mathbb{Z}^{m \times n}$ est la matrice associée, comme avant, alors ses diviseurs élémentaires sont $e'_1, \dots, e'_r, 0 \dots, 0$. Par construction on a $A' = SAT$ avec certaines matrices de passage $S \in \mathrm{SL}_m(\mathbb{Z})$ et $T \in \mathrm{SL}_r(\mathbb{Z})$. (Leur construction détaillée est laissée en exercice.) L'unicité énoncée dans le théorème 1.15 assure que $e'_i = \pm e_i$ pour tout $i = 1, \dots, r$. □

Corollaire 2.15. *Soit G un groupe abélien libre de rang fini. Alors tout sous-groupe H de G est libre et* $\operatorname{rang} H \leq \operatorname{rang} G$. *Il existe une base $g_1, \dots, g_m$ de G et un entier r avec $0 \leq r \leq m$ et des entiers $e_1, \dots, e_r \geq 1$ vérifiant $e_1 \mid \cdots \mid e_r$ tels que $e_1 g_1, \dots, e_r g_r$ soit une base de H. La suite des entiers $e_1, \dots, e_r$ est uniquement déterminée par H, aux signes près, et on les appelle* les diviseurs élémentaires *du sous-groupe $H \subset G$.* □

Corollaire 2.16. *Soient H et H' deux sous-groupes dans G ayant les familles $(e_1, \dots, e_r)$ et $(e'_1, \dots, e'_{r'})$, respectivement, pour diviseurs élémentaires. Il existe un automorphisme $\phi\colon G \to G$ vérifiant $\phi(H) = H'$ si et seulement si $(e_1, \dots, e_r) = (e'_1, \dots, e'_{r'})$.* □

2.4. Groupes abéliens finiment engendrés. Nous concluons avec un très beau théorème, la classification des groupes abéliens finiment engendrés :

Théorème 2.17. *Soit G un groupe abélien finiment engendré. Alors il existe un isomophisme de groupes*

$$G \cong \mathbb{Z}_{e_1} \times \mathbb{Z}_{e_2} \times \cdots \times \mathbb{Z}_{e_k} \times \mathbb{Z}^r$$

où $e_1, e_2, \ldots, e_k \geq 2$ sont des entiers satisfaisant $e_1 \mid e_2 \mid \cdots \mid e_k$. Ces nombres sont uniquement déterminés par G. On appelle r le rang *de la partie libre et $e_1, e_2, \ldots, e_k$ les* diviseurs élémentaires *de G.*

DÉMONSTRATION. *Existence.* — Par hypothèse G admet une famille génératrice finie $(g_1, \ldots, g_m)$. Soit $f\colon \mathbb{Z}^m \to G$ l'homomorphisme de groupes défini par $(\lambda_1, \ldots, \lambda_m) \mapsto \lambda_1 g_1 + \cdots + \lambda_m g_m$. Par hypothèse f est surjectif, le théorème d'isomorphisme nous assure donc $G \cong \mathbb{Z}^n/K$ où $K := \ker(f) \subset \mathbb{Z}^m$ est le noyau de f. D'après le théorème 2.14 il existe une base $b_1, \ldots, b_m$ de $\mathbb{Z}^m$ de $\mathbb{Z}^m$ et des entiers $e_1, e_2, \ldots, e_k \geq 1$ vérifiant $e_1 \mid e_2 \mid \cdots \mid e_k$ tels que $e_1 b_1, \ldots, e_k b_k$ soit une base de K. Par conséquent le groupe quotient $\mathbb{Z}^m/K$ est isomorphe au groupe $\mathbb{Z}_{e_1} \times \mathbb{Z}_{e_2} \times \cdots \times \mathbb{Z}_{e_k} \times \mathbb{Z}^r$ où $r = m - k$ est le rang de la partie libre. En supprimant d'éventuels facteurs triviaux $\mathbb{Z}_1 = \mathbb{Z}/\mathbb{Z} \cong \{0\}$, nous arrivons à la forme souhaitée avec $e_1, e_2, \ldots, e_k \geq 2$.

Unicité. — Si l'on ajoute un générateur g_{m+1} à la famille génératrice $(g_1, \ldots, g_m)$, ceci élargit $\mathbb{Z}^m$ à $\mathbb{Z}^{m+1}$ mais aussi le noyau par une relation $g_{m+1} = \sum_{i=1}^{m} \lambda_i g_i$. Les diviseurs élémentaires ne changent que par un facteur $e = 1$ supplémentaire, ce qui ne change pas le résultat. Cet argument prouve que les diviseurs élémentaires sont indépendants du choix de la famille génératrice : si $(g_1, \ldots, g_m)$ et $(g'_1, \ldots, g'_{m'})$ sont deux familles génératrices, alors $(g_1, \ldots, g_m, g'_1, \ldots, g'_{m'})$ est aussi une famille génératrice. D'après l'argument précédent, les diviseurs élémentaires calculés à partir de ces trois familles sont les mêmes. □

Rappelons qu'un groupe G est le produit direct de sous-groupes $G_1, \ldots, G_n$, noté $G = G_1 \times \cdots \times G_n$, si et seulement si l'application $G_1 \times \cdots \times G_n \to G$ donnée par le produit $(g_1, \ldots, g_n) \mapsto g_1 \cdots g_n$ est un isomorphisme de groupes. L'algorithme de Gauss-Bézout entraîne que tout groupe abélien finiment engendré est un produit direct de sous-groupes cycliques :

Corollaire 2.18. *Soit G un groupe abélien finiment engendré. Alors il existe des éléments non triviaux $g_1, \ldots, g_n \in G$ tels que $G = \langle g_1 \rangle \times \cdots \times \langle g_n \rangle$ et les ordres $e_i = \operatorname{ord}(g_i)$ vérifient $e_1 \mid \cdots \mid e_n$. Les nombres $(e_1, \ldots, e_m)$ sont uniquement déterminés par G et caractérisent le groupe G à isomorphisme près.*

Plus explicitement : supposons que H est un autre groupe abélien tels que $H = \langle h_1 \rangle \times \cdots \times \langle h_m \rangle$ pour certains éléments non triviaux $h_1, \ldots, h_m$ dont les ordres $f_j = \operatorname{ord}(h_j)$ vérifient $f_1 \mid \cdots \mid f_m$. Alors il existe un isomorphisme $G \cong H$ si et seulement si $(e_1, \ldots, e_n) = (f_1, \ldots, f_m)$. □

Exemple 2.19. Voici la liste des groupes abéliens d'ordre ≤ 12 à isomorphismes près. Ordre 1 : $\mathbb{Z}_1$; ordre 2 : $\mathbb{Z}_2$; ordre 3 : $\mathbb{Z}_3$; ordre 4 : $\mathbb{Z}_4$, $\mathbb{Z}_2 \times \mathbb{Z}_2$; ordre 5 : $\mathbb{Z}_5$; ordre 6 : $\mathbb{Z}_6$; ordre 7 : $\mathbb{Z}_7$; ordre 8 : $\mathbb{Z}_8$, $\mathbb{Z}_2 \times \mathbb{Z}_4$, $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_2$; ordre 9 : $\mathbb{Z}_9$; ordre 10 : $\mathbb{Z}_{10}$; ordre 11 : $\mathbb{Z}_{11}$; ordre 12 : $\mathbb{Z}_{12}$, $\mathbb{Z}_2 \times \mathbb{Z}_6$.

Le théorème de classification assure que cette liste est complète et ne contient pas de doublons : pour tout groupe abélien G d'ordre ≤ 12 il existe un et un seul groupe dans la liste qui soit isomorphe à G.

Exercice 2.20. Les groupes $\mathbb{Z}_5^\times$ et $\mathbb{Z}_8^\times$ et $\mathbb{Z}_{12}^\times$ sont tous d'ordre 4. Pour chacun entre eux trouver le groupe isomorphe dans la liste. Même question pour le groupe $\mathbb{Z}_{13}^\times$ d'ordre 12.

Exercice 2.21. Continuer la liste des groupes abéliens jusqu'à l'ordre 32 (ou plus loin si vous voulez). Comment énumérer les groupes d'ordre p^e où p est un nombre premier ? Comment le faire dans le cas général d'ordre $p_1^{e_1} \cdots p_k^{e_k}$? À titre d'illustration, énumérer les groupes d'ordre 8000 à isomorphisme près.

On two occasions I have been asked by members of Parliament,
'Pray, Mr. Babbage, if you put into the machine wrong figures,
will the right answers come out?' I am not able rightly to apprehend
the kind of confusion of ideas that could provoke such a question.
Charles Babbage (1792-1871)

CHAPITRE X

Arithmétique du groupe $\mathbb{Z}_n^\times$

Ce chapitre considère l'anneau quotient $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$ et plus particulièrement le groupe multiplicatif $\mathbb{Z}_n^\times$ des éléments inversibles. Ce groupe se révélera très important dans les applications des chapitres suivants, qui s'appuient sur la structure de $\mathbb{Z}_p^\times$ avec p premier. Dans le souci d'une implémentation efficace, ce paragraphe développe quelques algorithmiques spécifiques à cet objet. Le projet discutera les résidus quadratiques et le symbole de Jacobi, dont le calcul est similaire à l'algorithme d'Euclide.

Sommaire

1. Structure du groupe $\mathbb{Z}_n^\times$

On s'intéressera dans la suite à l'anneau $\mathbb{Z}_n$ et plus particulièrement à $\mathbb{Z}_n^\times$, le groupe multiplicatif des éléments inversibles dans $\mathbb{Z}_n$. Le cas d'un nombre premier se révèle le plus important :

Proposition 1.1. *Pour tout nombre naturel n les trois conditions suivantes sont équivalentes :*

(1) *Le nombre n est premier.*

(2) *L'anneau $\mathbb{Z}_n$ est un corps.*

(3) *Le groupe $\mathbb{Z}_n^\times$ est d'ordre $n-1$.*

Exercice/M 1.2. Montrer l'énoncé précédent. Rappeler le théorème de Lagrange sur l'ordre d'un élément (ou d'un sous-groupe) dans un groupe fini donné. En déduire le résultat suivant :

Corollaire 1.3 (Petit théorème de Fermat)**.** *Si p est premier, alors tout $x \in \mathbb{Z}_p^\times$ vérifie $x^{p-1} = 1$. En multipliant par x on obtient la formule $x^p = x$, qui est valable pour tout $x \in \mathbb{Z}_p$. Autrement dit, étant donné un nombre premier p, tout entier $x \in \mathbb{Z}$ vérifie $x^p \equiv x \pmod{p}$.*

Exercice/M 1.4. Vérifier que pour p premier et $x \in \mathbb{Z}_p^\times$ on pourrait calculer l'inverse x^{-1} par la puissance x^{p-2}. Estimer la complexité de ce calcul en utilisant la puissance dichotomique. Cette méthode est-elle plus rapide que l'inversion via Euclide-Bézout ? Est-elle aussi générale et facile à appliquer ?

Étant donné un nombre premier p il existe en général plusieurs groupes abéliens non isomorphes d'ordre $p-1$. (Voir la classification des groupes abéliens finis à la fin du projet IX.) Miraculeusement la structure du groupe $\mathbb{Z}_p^\times$ est toujours la plus simple qui soit :

Théorème 1.5. *Pour tout nombre premier p le groupe multiplicatif $\mathbb{Z}_p^\times$ est cyclique d'ordre $p-1$, c'est-à-dire qu'il existe $g \in \mathbb{Z}_p^\times$ tel que $\mathbb{Z}_p^\times = \langle g \rangle = \{g^1, g^2, \ldots, g^{p-1} = 1\}$. Un tel élément g est appelé un* générateur *de $\mathbb{Z}_p^\times$ ou une* racine primitive *modulo p.*

Exercice/M 1.6. Montrer ce théorème en détaillant l'esquisse suivante :

ESQUISSE DE PREUVE. Comme $\mathbb{Z}_p$ est un corps, le groupe $\mathbb{Z}_p^\times$ est d'ordre $n = p-1$. Soit $n = q_1^{e_1} \cdots q_k^{e_k}$ la décomposition en facteurs premiers. Le polynôme $X^{n/q_i} - 1$ possède au plus n/q_i racines dans le corps $\mathbb{Z}_p$. Il existe alors un élément $z_i \in \mathbb{Z}_p^\times$ tel que $z_i^{n/q_i} \neq 1$. Par conséquent $g_i = (z_i)^{n/q_i^{e_i}}$ est d'ordre $q_i^{e_i}$. Les ordres $q_1^{e_1}, \ldots, q_k^{e_k}$ étant premiers entre eux, on conclut que le produit $g = g_1 \cdots g_k$ est d'ordre $n = q_1^{e_1}, \ldots, q_k^{e_k}$, comme souhaité. □

Exemple 1.7. Vous pouvez vérifier à la main que $\mathbb{Z}_5^\times$ est engendré par 2 (et 3), et que $\mathbb{Z}_7^\times$ est engendré par 3 (et 5). Essayez de trouver des racines primitives pour des nombres premiers suivants.

Remarque 1.8. Soulignons que le théorème assure l'existence d'une racine primitive modulo p sans en expliciter aucune. Effectivement, on ne connaît pas de formule miracle pour trouver une racine primitive de $\mathbb{Z}_p^\times$. En particulier la valeur de la plus petite racine primitive modulo p reste mystérieuse ; il nous ne reste que le tâtonnement par essais successifs. Ceci dit, on traduira dans la suite la preuve d'existence en une méthode efficace pour chercher une racine primitive.

Remarque 1.9. Une fois on a trouvé *une* racine primitive de $\mathbb{Z}_p^\times$ on les connaît toutes : toute racine primitive $g \in \mathbb{Z}_p^\times$ induit un isomorphisme $\phi : \mathbb{Z}_{p-1} \xrightarrow{\sim} \mathbb{Z}_p^\times$, $k \mapsto g^k$, et réciproquement tout tel isomorphisme ϕ correspond au choix d'une racine primitive $g = \phi(1)$. D'un coté $x \in \mathbb{Z}_p^\times$ est une racine primitive ssi x est un générateur du groupe $\mathbb{Z}_p^\times$. De l'autre coté $k \in \mathbb{Z}_{p-1}$ est un générateur ssi k est inversible, c'est-à-dire $k \in \mathbb{Z}_{p-1}^\times$. Ainsi l'isomorphisme ϕ établit une bijection entre $\mathbb{Z}_{p-1}^\times$ et les racines primitives de $\mathbb{Z}_p^\times$.

1.1. Structure du groupe $\mathbb{Z}_n^\times$. Le théorème précédent donne la structure de $\mathbb{Z}_p^\times$ pour p premier. On peut ensuite s'interroger sur la structure de $\mathbb{Z}_n^\times$ pour un entier $n \geq 2$ quelconque. Ce problème se simplifie considérablement en appliquant le théorème des restes chinois : On décompose $n = p_1^{e_1} \cdots p_k^{e_k}$ avec $p_1 < \cdots < p_k$ premiers et $e_1, \ldots, e_k \geq 1$. Le théorème chinois fournit un isomorphisme d'anneaux $\mathbb{Z}_n \cong \mathbb{Z}_{p_1^{e_1}} \times \cdots \times \mathbb{Z}_{p_k^{e_k}}$. On en déduit un isomorphisme de groupes $\mathbb{Z}_n^\times \cong \mathbb{Z}_{p_1^{e_1}}^\times \times \cdots \times \mathbb{Z}_{p_k^{e_k}}^\times$. (Le détailler.)

Exercice/M 1.10. *L'indicateur d'Euler* est la fonction $\varphi : \mathbb{N} \to \mathbb{N}$ définie par $\varphi(n) := |\mathbb{Z}_n^\times|$.
- Montrer que $\varphi(p^e) = (p-1)p^{e-1}$ si p est premier et $e \geq 1$.
- Montrer que $\varphi(ab) = \varphi(a)\varphi(b)$ si a et b sont premiers entre eux.
- Pour $n = p_1^{e_1} \cdots p_k^{e_k}$ conclure que $\varphi(n) = \prod_{i=1}^k (p_i - 1) p_i^{e_i - 1} = n \prod_{i=1}^k (1 - \frac{1}{p_i})$

Comme application montrer le résultat suivant, qui généralise le petit théorème de Fermat :

Corollaire 1.11 (Euler-Lagrange)**.** *L'ordre de tout élément $x \in \mathbb{Z}_n^\times$ divise l'ordre du groupe $\mathbb{Z}_n^\times$, donc $x^{\varphi(n)} = 1$. Autrement dit, tout entier x premier avec n vérifie $x^{\varphi(n)} \equiv 1 \pmod{n}$.*

Exercice/M 1.12. Vérifier que a est un générateur de $(\mathbb{Z}_n, +)$ si et seulement si a est inversible dans $\mathbb{Z}_n$. En déduire que tout groupe cyclique d'ordre n admet exactement $\varphi(n)$ générateurs. En particulier, pour p premier, il existe exactement $\varphi(p-1)$ racines primitives dans $\mathbb{Z}_p^\times$. Si l'on choisit $x \in \mathbb{Z}_p^\times$ de manière aléatoire, quelle est la probabilité de tomber sur une racine primitive ?

Outre l'ordre $\varphi(n)$ on veut connaître la structure précise du groupe $\mathbb{Z}_n^\times$. À nouveau, par le théorème des restes chinois, il suffit de traiter le cas $n = p^e$. Pour le résultat suivant consultez votre cours d'algèbre :

Théorème 1.13. *Si $n = p^e$ est la puissance d'un nombre premier impair $p \geq 3$ à l'exposant $e \geq 2$, alors le groupe $\mathbb{Z}_n^\times$ est cyclique d'ordre $\varphi(p^e) = (p-1)p^{e-1}$. Si g est un générateur de $\mathbb{Z}_p^\times$, alors g ou $g+p$ est un générateur de $\mathbb{Z}_{p^e}^\times$ pour tout $e \geq 2$.*

Pour $p = 2$ la situation est différente : $\mathbb{Z}_2^\times = \{1\}$ est trivial, $\mathbb{Z}_4^\times = \{\pm 1\}$ est cyclique d'ordre 2, mais pour $e \geq 3$, le groupe $\mathbb{Z}_{2^e}^\times$ n'est plus cyclique. Il est le produit directe du sous-groupe $\langle -1 \rangle$ d'ordre 2 et du sous-groupe $\langle 5 \rangle$ d'ordre 2^{e-2}. □

Exercice/M 1.14. Déduire du théorème que $\mathbb{Z}_n^\times$ est cyclique si et seulement si $n = 2, 4, p^e, 2p^e$ avec un nombre premier $p \geq 3$ et $e \geq 1$. *Indication.* — Dans tout autre cas on peut construire un homomorphisme surjectif $\mathbb{Z}_n^\times \to \mathbb{Z}_2 \times \mathbb{Z}_2$. Comme le groupe image n'est pas cyclique, $\mathbb{Z}_n^\times$ ne l'est pas non plus.

1.2. Déterminer l'ordre d'un élément. Comment déterminer efficacement l'ordre de x dans $\mathbb{Z}_m^\times$? Évidemment la méthode naïve consiste à calculer successivement $x^1, x^2, x^3, \ldots$ pour ainsi trouver le plus petit exposant $n \geq 1$ tel que $x^n = 1$ dans $\mathbb{Z}_m$. Ceci est très inefficace lorsque n est grand.

Exemple 1.15. Regardons $m = 2^{32} \cdot 3^{32} \cdot 5^{32} + 1 > 10^{47}$. Il se trouve que m est premier, ce qui permet de déduire $\varphi(m) = m - 1 = 2^{32} \cdot 3^{32} \cdot 5^{32}$. Comment déterminer l'ordre de $\overline{3}$ dans $\mathbb{Z}_m^\times$? Il se trouve que $\mathrm{ord}(\overline{3}) = 2^{26} \cdot 3^{30} \cdot 5^{32}$. Il est donc hors de question d'attaquer cette question par le tâtonnement naïf !

Calculons intelligemment en exploitant notre connaissance du théorème de Lagrange : il nous garantit que l'ordre de x est un diviseur de $\varphi(m)$, ce qui limite considérablement les exposants n à tester ! Supposons connue la décomposition $\varphi(m) = p_1^{m_1} \cdots p_k^{m_k}$ avec $p_1 < \cdots < p_k$ premiers et $m_1, \dots, m_k \geq 1$. L'ordre de $x \in \mathbb{Z}_n^\times$ est donc de la forme $\operatorname{ord}(x) = p_1^{n_1} \cdots p_k^{n_k}$ avec $0 \leq n_i \leq m_i$. Posons $q = \varphi(m)/p_i^{m_i}$ pour un indice $i = 1, \dots, k$. Alors $y = x^q$ est d'ordre $\operatorname{ord}(y) = p_i^{n_i}$. Pour trouver n_i il suffit maintenant de regarder $y, y^{p_1}, y^{p_1^2}, \dots, y^{p_i^{m_i}}$ afin de déterminer le plus petit n_i tel que $y^{p_i^{n_i}} = 1$.

Algorithme X.1 Déterminer l'ordre d'un élément x dans le groupe $\mathbb{Z}_m^\times$

Entrée: un élément $x = \bar{a}$ dans $\mathbb{Z}_m^\times$ et la factorisation $\varphi(m) = p_1^{m_1} \cdots p_k^{m_k}$

Sortie: l'ordre de x dans $\mathbb{Z}_m^\times$, c'est-à-dire le plus petit entier $n \geq 1$ tel que $x^n = 1$

si $\operatorname{pgcd}(a,m) > 1$ **alors retourner** « erreur »
pour i **de** 1 **à** k **faire**
  $q \leftarrow \varphi(m)/p_i^{m_i} = p_1^{m_1} \cdots \widehat{p_i^{m_i}} \cdots p_k^{m_k}$, $y \leftarrow a^q \bmod m$, $n_i \leftarrow 0$
  tant que $y \neq 1$ **faire** $y \leftarrow y^{p_i} \bmod m$, $n_i \leftarrow n_i + 1$
fin pour
retourner $n = p_1^{n_1} \cdots p_k^{n_k}$

Exercice/M 1.16. Prouver que l'algorithme précédent est correct. Pourquoi s'arrête-t-il ? Comment être sûr que dans la $i$ème itération x^q est d'ordre $p_i^{n_i}$? À noter que la spécification exige que $x \in \mathbb{Z}_m^\times$; quel est l'intérêt du test redondant $\operatorname{pgcd}(a,m) > 1$? Est-il coûteux ? Expliquer pourquoi tous les calculs s'effectuent efficacement si l'on utilise la puissance dichotomique modulaire (voir le projet VIII). Montrer ainsi que la complexité est d'ordre $O(k \log(m)^3)$ utilisant la multiplication/division scolaire. Justifier ainsi l'intérêt de cet algorithme vis-à-vis le tâtonnement naïf.

Exercice/P 1.17. Vérifier l'exemple précédent. Puis pour le nombre premier $p = 41! + 1$ déterminer l'ordre de $2, 3, 4, \dots$ dans $\mathbb{Z}_p^\times$. Quelle est la plus petite racine primitive dans $\mathbb{Z}_p^\times$?

Remarque 1.18. L'algorithme précédent s'applique plus généralement à un groupe G quelconque pourvu que l'on sache préalablement assurer $x^m = 1$ et factoriser $m = p_1^{m_1} \cdots p_k^{m_k}$. Si G est fini alors $m = |G|$ convient. L'algorithme s'applique même à des groupes infinis, dans quel cas il faut assurer $x^m = 1$ par un autre moyen pour pouvoir satisfaire l'hypothèse de l'algorithme.

Remarque 1.19. Afin d'être efficace, l'algorithme précédent suppose que l'on sache factoriser l'exposant en question. Ceci peut être facile dans certains cas, mais en général la factorisation est une tâche très dure ! C'est pour cette raison que nous l'avons placée ici parmi les *hypothèses* de l'algorithme : il faut d'abord résoudre le problème de factorisation avant de l'appliquer. Ainsi la factorisation d'entiers reste un problème à part ; nous le discuterons dans le chapitre suivant.

2. Algorithmes probabilistes

2.1. Recherche d'une racine carrée de -1 modulo p. Étant donné un nombre premier p on se propose de trouver une racine carrée de -1 dans $\mathbb{Z}_p^\times$. Le développement qui suit est une application exemplaire de nos connaissances sur la structure de $\mathbb{Z}_p^\times$. L'algorithme efficace qui en découle nous servira plus tard, dans le projet XII, dans un tout autre contexte.

Exercice/M 2.1. Montrer que $\mathbb{Z}_p^\times$ contient une racine carrée de -1 si et seulement si $4 \mid p - 1$.

Désormais soit $p = 4k + 1$ un nombre premier. On cherche une racine $y \in \mathbb{Z}_p^\times$ du polynôme $X^2 + 1$: il en existe exactement deux, notons-les y et $-y$. Pensons à un nombre p gigantesque, comme $10^{100} + 949$. *Comment trouver deux aiguilles dans une telle botte de foin ?*

Exercice/P 2.2. Écrire une fonction qui prend comme paramètre un nombre premier $p = 4k + 1$ et cherche le plus petit entier $y = 2, 3, \dots$ tel que $y^2 \equiv -1 \pmod{p}$.

Remarque. — Il sera instructif de faire afficher chaque essai par `cout << '.'`. Comme la deuxième racine est $p - y$, il suffit d'en trouver la première. Si l'on n'en trouve pas dans $y \in [\![2, 2k]\!]$, la fonction peut

renvoyer 0 pour signaler l'erreur : dans ce cas p ne peut être premier. (La conclusion réciproque est fausse : 25 n'est pas premier, mais il existe bien des racines carrées de -1 modulo 25. Les expliciter.)

Exemple 2.3. Tester votre fonction sur 5, 13, 17, 29, 37, ... puis sur des nombres plus grands :

$$1009,\ 10^6+33,\ 10^9+9,\ 10^{15}+37,\ 10^{30}+57,\ 10^{50}+577,\ 10^{100}+949.$$

Jusqu'où peut-on aller ? Convainquez-vous que la valeur de y en fonction de p semble aléatoire. De manière heuristique, quel est le nombre moyen d'itérations nécessaires pour trouver y ?

Peut-on trouver une méthode plus efficace ? Bien sûr ! Rappelons que $\mathbb{Z}_p^\times$ est cyclique d'ordre $4k$. Pour tout $x \in \mathbb{Z}_p^\times$ la puissance $y = x^k$ vérifie alors $y^4 = 1$. En particulier $z = y^2$ vérifie $z^2 = 1$, et dans un corps ceci implique soit $z = 1$ soit $z = -1$. Dans le cas favorable $z = -1$ on a trouvé avec y une des deux racines carrées de -1 modulo p. Ceci motive l'algorithme suivant :

Algorithme X.2 Trouver une racine carrée de -1 modulo p

Entrée: un nombre premier p de la forme $p = 4k+1$
Sortie: un entier y tel que $y^2 \equiv -1$ modulo p.

répéter
  choisir un entier $x \in [\![2, p-2]\!]$ de manière aléatoire
  calculer $y \leftarrow x^k \bmod p$, puis $z \leftarrow y^2 \bmod p$
jusqu'à $z \neq 1$
si $z = p-1$ **alors retourner** y **sinon retourner** « erreur : p n'est pas premier »

Exercice/M 2.4. Justifier l'algorithme précédent ; en particulier expliquer pourquoi on peut espérer de trouver rapidement un élément x qui convient. *Indication.* — Soient $\pm y$ les deux racines carrées de -1 modulo $p = 4k+1$. Vérifier que l'application $h\colon x \mapsto x^k$ définie un homomorphisme surjectif $h\colon \mathbb{Z}_p^\times \to \{\pm 1, \pm y\}$, le noyau étant le sous-groupe des éléments dont l'ordre divise k. Montrer ainsi que pour la moitié des éléments $x \in \mathbb{Z}_p^\times$ on tombe sur une des racines $\pm y$ cherchée.

Remarque. — Quand on appelle l'algorithme pour un nombre premier p, comme exigé par la spécification, alors on a toujours $z = p-1$ au test final. Bien que redondante, quel pourrait être l'intérêt pratique d'une telle mesure de précaution ? Est-elle coûteuse ? Vaut-il mieux la supprimer ou la garder en place ?

Exercice/P 2.5. Écrire une fonction efficace qui prend comme paramètre un nombre premier $p = 4k+1$ et qui renvoie un entier $y \in [\![2, p-2]\!]$ vérifiant $y^2 \equiv -1$ modulo p.

Remarque. — Veiller à implémenter une puissance efficace. Comme avant il sera instructif de faire afficher chaque essai par `cout << '.'`. Justifier que votre fonction travaille correctement et motiver son intérêt. Vérifier empiriquement votre prévision sur les exemples ci-dessus.

Remarque. — Dans la pratique on remplace souvent le choix aléatoire de x par des essais successifs $x = 2, 3, \ldots$. Quels avantages (pragmatiques) et inconvénients (théoriques) voyez vous dans cette variante ?

Exercice/M 2.6. Essayons de déterminer la complexité asymptotique des deux méthodes :

(1) Supposons que la première méthode nécessite k itérations en moyenne. (Ce nombre correspond-il à vos expériences ? Vous pouvez le justifier sous l'hypothèse que y parcourt l'intervalle de manière aléatoire.) Chaque test calcule $y \mapsto y^2$ en effectuant une multiplication modulo p. La complexité moyenne est donc d'ordre $\Theta(p \ln(p)^2)$.

(2) Supposons que la deuxième méthode nécessite 2 itérations en moyenne. (Vous pouvez le justifier rigoureusement si vous voulez. Ce nombre correspond-il à vos expériences ?) Chaque test effectue une puissance dichotomique $y \mapsto y^k$, ce qui nécessite entre $\log_2 k$ et $2\log_2 k$ multiplications modulo p. La complexité moyenne est donc d'ordre $\Theta(\ln(p)^3)$.

Vérifier ces affirmations et comparer les prévisions aux expériences.

2.2. Recherche d'un élément d'ordre q^e modulo p. Le paragraphe précédent a présenté une méthode efficace pour trouver un élément d'ordre 4 dans $\mathbb{Z}_p^\times$. On étendra aisément cette approche aux éléments d'ordre q^e avec q premier.

Algorithme X.3 Trouver un élément $y \in \mathbb{Z}_p^\times$ d'ordre q^e

Entrée: deux nombres premiers p et q et un exposant $e \geq 1$ tel que $p-1 = q^e r$
Sortie: un entier $y \in [\![1, p-1]\!]$ représentant un élément d'ordre q^e dans $\mathbb{Z}_p^\times$

répéter
  choisir $x \in [\![1, p-1]\!]$ de manière aléatoire
  calculer $y \leftarrow x^r \bmod p$, puis $z \leftarrow y^{q^{e-1}} \bmod p$
jusqu'à $z \neq 1$
si $z^q \equiv 1 \pmod p$ **alors retourner** y **sinon retourner** « erreur : p n'est pas premier »

Exercice/M 2.7. La preuve de cet algorithme suit exactement la démonstration précédente : Soit p un nombre premier. Vérifier que l'application $h\colon x \mapsto x^r$ définit un homomorphisme $h\colon \mathbb{Z}_p^\times \to \mathbb{Z}_p^\times$. Le noyau $\ker(h)$ est le sous-groupe des éléments dont l'ordre divise r.

Supposons que g est un générateur de $\mathbb{Z}_p^\times$ et que $p-1 = q^e r$. Alors $\ker(h)$ est cyclique d'ordre r, engendré par g^{q^e}. L'image $\mathrm{im}(h)$ est le sous-groupe des éléments dont l'ordre divise q^k ; il est cyclique d'ordre q^e, engendré par g^r. En particulier $\mathrm{im}(h)$ contient exactement $(q-1)q^{e-1}$ éléments d'ordre q^k.

Montrer qu'avec probabilité $1 - \frac{1}{q}$ le choix de x mène à un élément y d'ordre q^k. En déduire que l'algorithme précédent est correct, et justifier l'intérêt de cette approche.

Exercice/P 2.8. Écrire une fonction efficace qui implémente l'algorithme ci-dessus. Comme toujours, il faut veiller à utiliser une puissance efficace.

Remarque. — Quand on appelle l'algorithme pour un nombre premier p, comme exigé par la spécification, alors on a toujours $z^q = 1$ au test final. Bien que redondante, quel pourrait être l'intérêt pratique d'une telle mesure de précaution ? Est-elle coûteuse ? Vaut-il mieux la supprimer ou la garder en place ?

Remarque. — Dans la pratique on remplace souvent le choix aléatoire de x par des essais successifs $x = 2, 3, \ldots$. Quels avantages (pragmatiques) et inconvénients (théoriques) voyez vous dans cette variante ?

Exemple 2.9. Trouver un élément d'ordre 41 modulo $p = 41! + 1$. Vous pouvez vérifier aisément la factorisation $41! = 2^{38} \cdot 3^{18} \cdot 5^9 \cdot 7^5 \cdot 11^3 \cdot 13^3 \cdot 17^2 \cdot 19^2 \cdot 23 \cdot 29 \cdot 31 \cdot 37 \cdot 41$. Si vous voulez, essayez de trouver un élément d'ordre 1024, puis un élément d'ordre 81. Comment en fabriquer un élément d'ordre $3400704 = 2^{10} \cdot 3^4 \cdot 41$? On discutera plus bas la généralisation évidente : produire un élément $g \in \mathbb{Z}_{41!+1}^\times$ d'ordre 41! c'est-à-dire une racine primitive.

2.3. Recherche d'une racine primitive modulo p. Reprenons le théorème 1.5 qui assure l'existence d'une racine primitive modulo p sans en expliciter aucune. Heureusement nous sommes en mesure de remédier à ce défaut, non par une formule close, mais par un algorithme efficace.

D'après le théorème, $\mathbb{Z}_p^\times$ est un groupe cyclique d'ordre $n = p-1$. On pourrait donc parcourir $g = 2, 3, \ldots$ en calculant chaque fois l'ordre de g dans $\mathbb{Z}_p^\times$. Soulignons que ceci est trop coûteux avec la méthode naïve, mais tout à fait faisable avec l'algorithme efficace X.1, qui découle du théorème de Lagrange. Voici une version peaufinée pour la question restreinte :

Algorithme X.4 Déterminer si g est un générateur de $\mathbb{Z}_p^\times$

Entrée: Un entier g, un nombre premier p, et la factorisation $p-1 = q_1^{e_1} \cdots q_k^{e_k}$
Sortie: le message « g est un générateur » si et seulement si g est un générateur de $\mathbb{Z}_p^\times$.

pour i **de** 1 **à** k **faire**
  calculer $y \leftarrow g^{(p-1)/q_i} \bmod p$.
  si $y = 1$ **alors retourner** « g n'est pas un générateur »
  si $(y^{q_i} \bmod p) \neq 1$ **alors retourner** « erreur : p n'est pas premier »
fin pour
retourner « g est un générateur »

Exercice/M 2.10. Montrer que l'algorithme ci-dessus est correct.

Remarque. — Selon la spécification p est premier ; en déduire que l'on a toujours $(y^{q_i} \bmod p) = 1$. Ce test est donc redondant quand on assure d'avance que p est premier. Quel pourrait être l'intérêt pratique d'une telle mesure de précaution ? Est-elle coûteuse ? Vaut-il mieux la supprimer ou la garder en place ?

Exercice/M 2.11. Si l'on choisit $g \in \mathbb{Z}_p^\times$ de manière aléatoire, quelle est la probabilité de tomber sur une racine primitive ? En déduire une méthode efficace pour trouver une racine primitive. Comment en trouver la plus petite ? Heuristiquement pourquoi peut-on espérer de la trouver rapidement ?

Exercice/M 2.12. On peut faire légèrement mieux si l'on veut trouver *une* racine primitive, n'importe laquelle. Étant donné la factorisation $p-1 = q_1^{e_1} \cdots q_k^{e_k}$ on sait déjà trouver des éléments $g_1, \ldots, g_k$ d'ordre $q_1^{e_1}, \ldots, q_k^{e_k}$ respectivement (voir l'algorithme X.3 plus haut). Montrer que $g = g_1 \cdots g_k$ est d'ordre $p-1$, c'est donc une racine primitive comme souhaité. Voyez-vous l'avantage par rapport à l'approche précédente ? Déterminer la probabilité de succès et le nombre moyen d'itérations.

☞ *Remarque.* — Cette méthode est la version algorithmique de notre preuve du théorème 1.5 ci-dessus. Ainsi se ferme le cercle d'idées autour de la structure de $\mathbb{Z}_p^\times$.

Exercice/P 2.13. Montrer que $\overline{2}$ est une racine primitive modulo $p = 2^{32} \cdot 3^{32} \cdot 5^{32} + 1$. Puis trouver une racine primitive g modulo $p = 41! + 1$. Est-ce que ces questions sont abordables par une recherche naïve ? Par quelle méthode pensez-vous y parvenir le plus facilement ? Après avoir trouvé un élément $g \in \mathbb{Z}_p$ d'ordre $p-1$ peut-on conclure que p est premier ? Félicitations, vous venez de découvrir une preuve de primalité ! Contemplez ce bel exploit. Nous y reviendrons au chapitre suivant.

Tout problème mathématique pourrait, en principe, être directement résolu par une énumération exhaustive. Mais il existe des problèmes d'énumération qui peuvent actuellement être résolus en quelques minutes, alors que sans méthode toute une vie humaine n'y suffirait pas.
Ernst Mach, *Populär-wissenschaftliche Vorlesungen*, 1896

PROJET X

Résidus quadratiques et symbole de Jacobi

Sommaire

1. Le symbole de Jacobi

Étant donné un entier impair $n \geq 3$, on dit que $a \in \mathbb{Z}$ est un *résidu quadratique* ou *carré modulo n* s'il existe $r \in \mathbb{Z}$ tel que $r^2 \equiv a \pmod{n}$. Dans ce cas on appelle r une *racine carrée* de a modulo n. Ce projet étudie les résidus quadratiques modulo n. Voici la première méthode qui vient à l'esprit :

Exercice/P 1.1. Écrire une fonction `bool est_carre( Integer a, Integer n )` qui teste par des essais successifs pour $r \in [\![0, \frac{n-1}{2}]\!]$ si $a \equiv r^2 \pmod{n}$. Elle renvoie `true` si a est un résidu quadratique modulo n et `false` sinon. Cette méthode est-elle praticable pour $a = 2$ et $n = 1009$? pour $n = 10^9 + 7$? pour $n = 10^{18} + 3$? pour $n = 10^{56} + 3$? (On développera des méthodes plus efficaces dans la suite.)

1.1. Le symbole de Legendre. Soit $p \geq 3$ un nombre premier. Pour $a \in \mathbb{Z}$ on note $\bar{a} \in \mathbb{Z}_p$ sa classe modulo p. On définit alors le *symbole de Legendre* de a modulo p par

$$\left(\frac{a}{p}\right) := \begin{cases} +1 & \text{si } \bar{a} \in \mathbb{Z}_p^\times \text{ est un carré,} \\ -1 & \text{si } \bar{a} \in \mathbb{Z}_p^\times \text{ n'est pas un carré,} \\ 0 & \text{si } \bar{a} = 0. \end{cases}$$

Exercice/M 1.2. Afin d'avoir des exemples concrets, déterminer $\left(\frac{5}{7}\right)$ et $\left(\frac{7}{5}\right)$, puis $\left(\frac{7}{11}\right)$ et $\left(\frac{11}{7}\right)$.

Exercice/M 1.3. Rappelons que $\mathbb{Z}_p$ est un corps et que $\mathbb{Z}_p^\times$ est un groupe cyclique d'ordre $p-1$. Il existe alors une racine primitive g d'ordre $p-1$, c'est-à-dire $\mathbb{Z}_p^\times = \{g, g^2, g^3, \ldots, g^{p-1} = 1\}$.

(1) En fonction de g caractériser les carrés et les non-carrés dans $\mathbb{Z}_p^\times$. Déterminer $g^{\frac{p-1}{2}} \in \mathbb{Z}_p$.

(2) En déduire le critère d'Euler, dont l'énoncé ne fait plus intervenir la racine primitive : Pour tout $a \in \mathbb{Z}$ on a $\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}$.

(3) En déduire la multiplicativité $\left(\frac{a_1 a_2}{p}\right) = \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right)$ ainsi que $\left(\frac{1}{p}\right) = 1$.

Exercice/P 1.4. Écrire une fonction efficace `int legendre( Integer a, Integer p )` qui calcule la puissance $e = a^{\frac{p-1}{2}} \bmod p$ puis renvoie ± 1 si $e \equiv \pm 1$, et renvoie 0 dans tout autre cas. (Pour effectuer ce calcul on suppose que $p \geq 3$ est impair. Si jamais $e \not\equiv \pm 1$ la fonction signale l'erreur en renvoyant 0.)

Exercice 1.5. Combien d'itérations faut-il pour évaluer `legendre(a,p)` ? Cette méthode est-elle praticable pour les exemples de l'exercice 1.1 ? (Attention aux hypothèses différentes.) Êtes-vous contents de la performance ? Justifier l'intérêt de cette méthode vis-à-vis la méthode naïve utilisée en exercice 1.1.

1.2. La loi de réciprocité quadratique. Dans la suite on aura besoin d'une formule importante, dont on admettra la preuve. Il s'agit de la célèbre *loi de réciprocité quadratique* de Gauss :

Théorème 1.6. *Soient p,q deux nombres premiers impairs distincts. Alors on a la formule de réciprocité*

$$\left(\frac{q}{p}\right) = \left(\frac{p}{q}\right)\cdot\varepsilon(p,q) \quad \textit{avec} \quad \varepsilon(p,q) := \begin{cases} +1 & \textit{si } p\equiv 1 \textit{ ou } q\equiv 1 \pmod 4, \\ -1 & \textit{si } p\equiv 3 \textit{ et } q\equiv 3 \pmod 4. \end{cases}$$

Pour le cas exceptionnel $q=2$ on a la formule complémentaire :

$$\left(\frac{2}{p}\right) = \delta(p) \quad \textit{avec} \quad \delta(p) := \begin{cases} +1 & \textit{si } p\equiv \pm 1 \pmod 8, \\ -1 & \textit{si } p\equiv \pm 3 \pmod 8. \end{cases}$$

Exercice/M 1.7. Déterminer $\left(\frac{5}{7}\right)$ et $\left(\frac{7}{5}\right)$ comme en exercice 1.2. Que vaut $\varepsilon(5,7)$?
Vérifier la réciprocité dans ce cas particulier. Même exercice avec $\left(\frac{7}{11}\right)$ et $\left(\frac{11}{7}\right)$ et $\varepsilon(7,11)$.

Exercice/M 1.8. Vérifier la formule complémentaire pour $\left(\frac{2}{3}\right)$ et $\left(\frac{2}{5}\right)$ puis $\left(\frac{2}{7}\right)$ et $\left(\frac{2}{17}\right)$.

1.3. Le symbole de Jacobi. Le symbole de Legendre $\left(\frac{a}{p}\right)$ n'est défini que pour les nombres premiers $p\geq 3$. On l'étend aux nombres composés par multiplicativité : pour $b\geq 1$ impair on définit le *symbole de Jacobi* par $\left(\frac{a}{b}\right) := \prod_i \left(\frac{a}{p_i}\right)$ où $b=\prod_i p_i$ est la décomposition de b en facteurs premiers p_i. À noter que

$$\left(\frac{a}{1}\right) = 1 \qquad \text{et} \qquad \left(\frac{a}{b_1 b_2}\right) = \left(\frac{a}{b_1}\right)\left(\frac{a}{b_2}\right).$$

Exercice/M 1.9. Dans l'objectif d'un calcul efficace, justifier les règles de calcul suivantes :

① $\left(\frac{a}{b}\right) = 0$ si et seulement si $\operatorname{pgcd}(a,b)>1$

② $\left(\frac{1}{b}\right) = 1$ et $\left(\frac{a_1 a_2}{b}\right) = \left(\frac{a_1}{b}\right)\left(\frac{a_2}{b}\right)$

③ $\left(\frac{a}{b}\right) = \left(\frac{a'}{b}\right)$ si $a\equiv a' \pmod b$

④ $\left(\frac{a}{b}\right) = \left(\frac{b}{a}\right)\cdot\varepsilon(a,b)$ si a est impair

⑤ $\left(\frac{2}{b}\right) = \delta(b)$

Ici on sous-entend que ε et δ sont définis par les formules ci-dessus pour des nombres impairs quelconques. Pour prouver ces règles il faut passer par la décomposition en facteurs premiers : pour ⑤ montrer d'abord que δ est multiplicatif, et pour ④ que ε est multiplicatif en chaque argument.

Exercice/M 1.10. Calculer à la main la valeur de $\left(\frac{71}{83}\right)$ en utilisant les règles ci-dessus. Vous pouvez ensuite comparer votre résultat avec celui de `legendre(71,83)`, car 83 est premier.

Exercice/M 1.11. Soit $a=13\,353\,839$ et admettons que $p=64a+3$ est premier. Existe-t-il une solution $x,y\in\mathbb{Z}$ à l'équation $x^2+yp=4a$? Même question pour l'équation $x^2+yp=8a$. *Remarque.* — Tout le calcul est faisable à la main ! Vous pouvez ensuite comparer avec `legendre`.

1.4. Une implémentation efficace. Après ces préparations, on se propose d'implémenter le calcul du symbole de Jacobi. À noter que sa définition utilise la décomposition en facteurs premiers, opération très coûteuse pour les grands entiers. Fort heureusement la loi de réciprocité permet un calcul efficace :

Exercice/P 1.12. Écrire une fonction `int jacobi( Integer a, Integer b )` qui calcule le symbole de Jacobi $\left(\frac{a}{b}\right)$ par une méthode similaire à l'algorithme d'Euclide, en utilisant les règles ①,②,③,④,⑤ ci-dessus. (Une version récursive sera plus facile à écrire, une version itérative sera légèrement plus efficace.) Une fois la fonction est construite, essayer de prouver la terminaison, puis la correction du calcul. La tester sur les exemples précédents (exercices 1.7, 1.8, 1.10, 1.11) afin de trouver d'éventuelles erreurs. Dans ces tests il sera instructif d'afficher les étapes intermédiaires.

2. Deux applications aux tests de primalité

Comme applications nous établissons le critère de primalité de Pépin, fait sur mesure pour les nombres de Fermat, puis le test probabiliste de Solovay-Strassen, qui s'applique à un entier quelconque.

2.1. Nombres de Fermat et le critère de Pépin. Fermat conjectura que $F_n = 2^{2^n}+1$ est premier pour tout n. Effectivement $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$, $F_4 = 65537$ sont tous premiers. Mais déjà Euler trouva la décomposition $F_5 = 4294967297 = 641 \cdot 6700417$. On constate que $641 = 5 \cdot 2^7 + 1$ et $6700417 = 52347 \cdot 2^7 + 1$. Ceci n'est pas un hasard :

Exercice/M 2.1. Si un nombre premier p divise F_n avec $n \geq 2$, alors $p = k \cdot 2^{n+2} + 1$ avec $k \in \mathbb{N}$. *Indication.* — En supposant $p \mid F_n$ calculer 2^{2^n} mod p, et en déduire l'ordre de 2 dans $\mathbb{Z}_p^\times$. Pour $n \geq 2$ on a $p \equiv 1 \pmod 8$, ce qui permet de déterminer $\left(\frac{2}{p}\right)$. En déduire qu'il existe $x \in \mathbb{Z}_p^\times$ d'ordre 2^{n+2}.

Exercice/P 2.2. Implémenter cette observation pour trouver un facteur $k \cdot 2^{n+2} + 1$ de F_n, au moins dans les cas $n = 5, 6, 9, 10, 11, 12, 15, 16, 18, 19, 23, 30$, puis tester d'autres indices n. *Indication.* — À noter que p est petit tandis que F_n est très grand. Pour tester si p divise F_n il n'est donc pas une bonne idée de calculer F_n dans $\mathbb{Z}$. Il est plus efficace de fixer d'abord p puis de calculer $F_n = 2^{2^n}+1$ modulo p via une puissance dichotomique modulaire. Ceci garantit que tous les calculs intermédiaires restent bornés par p.

Dans les cas restants $n = 7, 8, 13, 14, 17, 20, 21, 22, 24, \ldots$ on ne trouve pas de petits facteurs, et il est certainement trop coûteux de tester *tous* les candidats possibles. On développera dans la suite une méthode efficace pour déterminer néanmoins si F_n est premier ou composé. (On a déjà utilisé ce critère, dit de Pépin, dans le projet VIII.)

Exercice/M 2.3. Commençons par un critère suffisant. Soit $N = 2^k + 1$ et supposons que $a \in \mathbb{Z}_N$ vérifie $a^{2^{k-1}} = -1$. Montrer que $a \in \mathbb{Z}_N^\times$ et déterminer son ordre. Conclure que N est premier.

Exercice/M 2.4. Calculer $\left(\frac{F_n}{3}\right)$ puis $\left(\frac{3}{F_n}\right)$ par réciprocité. En déduire le critère de Pépin : Pour $n \geq 1$ le nombre F_n est premier si et seulement si $3^{\frac{F_n-1}{2}} \equiv -1 \pmod{F_n}$.

Exercice/P 2.5. En déduire une fonction `bool pepin(int n)` qui renvoie `true` si F_n est premier et `false` sinon. (Vous pouvez réutiliser la fonction `legendre` de l'exercice 1.4.) Jusqu'à quelle valeur de n pouvez-vous déterminer la nature de F_n ?

— ∗ —

Remarque historique. — Mis à part les cinq premiers cas, on ignore s'il existe d'autres nombres premiers parmi les nombres de Fermat. Bien que très particulière, cette question a attiré beaucoup d'attention ; elle apparaît d'ailleurs dans le problème classique de la construction des polygones réguliers à la règle et au compas.

Pour tout $5 \leq n \leq 30$ on sait que F_n est composé : pour certains F_n on connaît un ou plusieurs petits facteurs, pour d'autres on sait que F_n est composé grâce au critère de Pépin, sans pour autant connaître de facteurs. Les cas les plus durs et les plus récents sont F_{14} en 1964, F_{20} en 1988, F_{22} en 1995, et F_{24} en 2003, tous résolus par le critère de Pépin. Vous pouvez vous convaincre que le nombre F_{24} a 2^{24} bits, c'est-à-dire environ 5 million de décimales. En 2003 le test de Pépin pour F_{24} nécessitait environ 200 jours de calcul.

Actuellement, en 2006, le plus petit nombre de Fermat dont la nature reste inconnue est F_{33}. Malgré ces difficultés, on peut dire qu'il est raisonnablement facile de déterminer si F_n est premier ou composé, au moins pour $n \leq 32$. Pourtant il est extrêmement dur de trouver la décomposition de F_n en facteurs premiers, même pour n aussi petit que 9 ou 10 ou 11 : leurs factorisations ont été trouvées entre 1988 et 1995, nécessitant chaque fois quelques mois de calcul. On ignore actuellement la factorisation complète de F_n pour $n \geq 12$.

Ces exemples laissent déjà imaginer que la factorisation des grands entiers présente des difficultés considérables, ce qui en fait davantage un domaine intéressant. Pour savoir plus sur l'approche algorithmique aux nombres premiers, vous pouvez consulter le livre récent de R. Crandall et C. Pomerance [**15**] ou l'incontournable P. Ribenboim [**14**].

2.2. Test de primalité d'après Solovay et Strassen. L'étude précédente est très spécifique aux nombres de Fermat $F_k = 2^{2^k}+1$. Pour un entier impair n quelconque on ne sait pas prédire la forme de ses facteurs, et il n'existe pas de critère simple de primalité non plus. En particulier on ne sait pas prédire quels nombres $a \in [\![1,n[\![$ vont témoigner que n est composé ou certifier que n est premier. Voici l'observation clé :

Théorème 2.6 (R. Solovay et V. Strassen, 1977)**.** *Pour tout entier impair n l'ensemble*

$$G_n = \left\{ \bar{a} \in \mathbb{Z}_n \;\middle|\; 0 \neq \left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n} \right\}$$

est un sous-groupe de $\mathbb{Z}_n^\times$. On a $G_n = \mathbb{Z}_n^\times$ si et seulement si n est premier. Si n est composé, alors G_n est d'indice ≥ 2 dans $\mathbb{Z}_n^\times$, et on a donc la majoration $|G_n| \leq \frac{n-1}{2}$.

DÉMONSTRATION. L'observation que G_n est un sous-groupe se vérifie aisément. (Le détailler.) En plus on a déjà établi le critère d'Euler dans l'exercice 1.3 : si n est premier, alors $G_n = \mathbb{Z}_n^\times$. Il reste à montrer que $G_n \neq \mathbb{Z}_n^\times$ pour n composé. Supposons que n se décompose en $n = p^e q$ avec p premier, $e \geq 1$, et $p \nmid q$. Le théorème chinois nous donne l'isomorphisme d'anneaux $\Phi\colon \mathbb{Z}_n \xrightarrow{\sim} \mathbb{Z}_{p^e} \times \mathbb{Z}_q$. Il existe $g \in \mathbb{Z}$ tel que $\bar{g} \in \mathbb{Z}_{p^e}$ soit une racine primitive, c'est-à-dire un générateur du groupe cyclique $\mathbb{Z}_{p^e}^\times$ d'ordre $(p-1)p^{e-1}$. Soit $a \in \mathbb{Z}$ tel que $\Phi(\bar{a}) = (\bar{g}, \bar{1})$. Par construction on a $\bar{a} \in \mathbb{Z}_n^\times$ et nous affirmons que $\bar{a} \notin G_n$.

Supposons d'abord que $e = 1$; dans ce cas on a $n = pq$ avec p premier et un cofacteur $q \geq 3$. On trouve $\left(\frac{a}{n}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right) = \left(\frac{g}{p}\right)\left(\frac{1}{q}\right) = -1$, mais $\Phi(\bar{a}^{\frac{n-1}{2}}) = (\bar{g}^{\frac{n-1}{2}}, \bar{1}) \neq (-\bar{1}, -\bar{1})$, donc $\bar{a}^{\frac{n-1}{2}} \neq -\bar{1}$.

Supposons enfin que $n = p^e q$ avec $e \geq 2$. Si l'on avait $\bar{a}^{\frac{n-1}{2}} = \left(\frac{a}{n}\right) = \pm 1$, alors $\bar{a}^{n-1} = \bar{1}$ et $\Phi(\bar{a}^{n-1}) = (\bar{g}^{n-1}, \bar{1}) = (\bar{1}, \bar{1})$. Ceci voudrait dire que $\mathrm{ord}(\bar{g}) = (p-1)p^{e-1}$ divise $n-1$, on aurait donc que $p \mid n-1$, ce qui est impossible. Donc $\bar{a}^{\frac{n-1}{2}} \neq \pm\bar{1}$, et on conclut que $a \notin G_n$. □

On ne sait pas grand chose sur G_n outre la majoration de son cardinal. D'autre part, pour tout $a \in \mathbb{Z}_n$ donné, il est facile de tester si $a \in G_n$: il suffit de comparer $\left(\frac{a}{n}\right)$ et $a^{\frac{n-1}{2}}$. Heureusement nous disposons de deux méthodes efficaces : la réciprocité quadratique pour calculer $\left(\frac{a}{n}\right)$, et la puissance dichotomique modulaire pour calculer $a^{\frac{n-1}{2}}$ modulo n. Ceci donne lieu à un test de primalité : on choisit $a \in [\![1,n[\![$ de manière aléatoire. Si $\left(\frac{a}{n}\right) \not\equiv a^{\frac{n-1}{2}}$, alors n est composé. Si $\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}}$, alors n est possiblement premier, donc on répète le test.

Exercice/M 2.7. *Argument probabiliste.* — Si n est premier, alors il passe le test pour tout a. Si n est composé, alors un élément a choisit au hasard le témoignera avec probabilité $\geq \frac{1}{2}$; la probabilité d'échec est $\leq \frac{1}{2}$. Après t tests indépendants la probabilité d'échec tombe à $\leq 2^{-t}$. Ainsi l'itération de t tests trouve avec probabilité $\geq 1 - 2^{-t}$ un témoin a vérifiant $\left(\frac{a}{n}\right) \not\equiv a^{\frac{n-1}{2}}$. Vérifier ces affirmations.

Conclusion réciproque ? — Si après quelques tests de Solovay-Strassen la réponse est « composé » alors n est composé : on a effectivement trouvé une preuve, sous forme d'un témoin a. Si la réponse après 100 tests est toujours « possiblement premier » alors on voudrait conclure que n est premier, mais il reste une probabilité d'erreur majorée par $2^{-100} < 10^{-30}$. Quelle conclusion vous semble justifiée ?

Exercice/P 2.8. Écrire une fonction `Integer cherche_temoin( Integer n, int t=100 )` qui effectue au plus `t` tests de Solovay-Strassen. Elle renvoie le premier témoin trouvé, ou 0 si aucun témoin n'a été trouvé. Comme application, trouver le plus petit nombre premier de la forme $10^{100}+k$ avec $k \geq 0$. Est-il raisonnable de tester la primalité par la méthode naïve (c'est-à-dire via des divisions successives) ? Justifier ainsi l'intérêt du test probabiliste de Solovay-Strassen.

— ∗ —

Remarque historique. — Publié en 1977, le test de Solovay-Strassen fut le premier test probabiliste de primalité, et il est vite devenu un exemple phare de l'approche probabiliste. On peut même dire qu'il a déclenché l'étude approfondie des algorithmes probalistes, auparavant considérés comme heuristiques, non rigoureux et mathématiquement inintéressants. Dans le chapitre XI nous discuterons son successeur, le test de Miller-Rabin, qui est plus facile à implémenter et plus performant. Si le caractère probabiliste vous gêne, on discutera aussi une alternative déterministe, bien que moins rapide.

3. Une preuve de la réciprocité

Nul n'est censé ignorer la loi
(de la reciprocité quadratique).

3.1. Un théorème peu plausible ? La réciprocité quadratique établit un lien remarquable et plutôt inattendu : pourquoi la propriété de q d'être un carré modulo p serait-elle liée à la propriété de p d'être un carré modulo q ? A priori le « monde modulo p » et le « monde modulo q » n'ont rien en commun — n'est-ce pas l'affirmation du théorème chinois ?

Pourtant, après avoir calculé suffisamment d'exemples, on constate une certaine régularité. Avouons toutefois que la réciprocité est loin d'être évidente : même en rétrospective, en contemplant la formule ci-dessus, qui aurait deviné le facteur $\varepsilon(p,q) = (-1)^{(p-1)(q-1)/4}$?

Historiquement la réciprocité fut conjecturée par Euler, puis formulée explicitement par Legendre, mais sa preuve restait incomplète. La première preuve fut trouvée par Gauss en 1796, à l'âge de 19 ans, qui publierait six preuves différentes dans les années suivantes. On en compte plus de deux cents variantes publiées aujourd'hui, soit à peu près une par an. On trouve une liste chronologique assez complète sur le site de Franz Lemmermeyer, `www.rzuser.uni-heidelberg.de/~hb3/rchrono.html`

Les exercices suivants présentent une des preuves les plus élémentaires, découverte par G. Zolotarev en 1872, qui ne repose que sur les *permutations* et leur *signature*.

3.2. Quelques préparatifs. Commençons par une révision de quelques jolis résultats.

Exercice/M 3.1. Tout d'abord, rappelons les propriétés essentielles de la signature :

(1) Rappeler la définition, construction, et unicité de la signature $\operatorname{sign}_X : \operatorname{Sym}(X) \to \{\pm 1\}$. Quelle est la signature d'une transposition (i,j) ? D'un cycle $(i_1, i_2, \dots, i_\ell)$ de longueur ℓ ?

(2) Supposons X ordonné. Quel est le rapport entre la signature d'une permutation $\sigma : X \to X$ et les inversions, c'est-à-dire les paires $(i,j) \in X \times X$ telles que $i < j$ mais $\sigma(i) > \sigma(j)$?

(3) Si $X \subset Y$, expliquer comment on peut construire naturellement un homomorphisme de groupes injectif $\iota : \operatorname{Sym}(X) \hookrightarrow \operatorname{Sym}(Y)$. A-t-on $\operatorname{sign}_X = \operatorname{sign}_Y \circ \iota$?

(4) Supposons X décomposé en $X = A \cup B$ avec $A \cap B = \emptyset$. Si une permutation $\sigma : X \to X$ vérifie $\sigma(A) = A$ et $\sigma(B) = B$, a-t-on $\operatorname{sign}_X(\sigma) = \operatorname{sign}_A(\sigma|_A) \cdot \operatorname{sign}_B(\sigma|_B)$?

(5) Soit $\Phi : X \xrightarrow{\sim} Y$ une bijection. Expliquer comment construire de manière naturelle un isomorphisme de groupes $\Phi_* : \operatorname{Sym}(X) \xrightarrow{\sim} \operatorname{Sym}(Y)$. A-t-on $\operatorname{sign}_X = \operatorname{sign}_Y \circ \Phi_*$?

$$\begin{array}{ccc} \operatorname{Sym}(X) & \xrightarrow[\cong]{\Phi_*} & \operatorname{Sym}(Y) \\ & {\scriptstyle \operatorname{sign}_X} \searrow \quad \swarrow {\scriptstyle \operatorname{sign}_Y} & \\ & \{\pm 1\} & \end{array}$$

(6) Si $\sigma : X \xrightarrow{\sim} X$ et $\tau : Y \xrightarrow{\sim} Y$ sont deux permutations, déterminer le signe de la permutation produit $\sigma \times \tau : X \times Y \xrightarrow{\sim} X \times Y$ définie par $(x,y) \mapsto (\sigma(x), \tau(y))$.

Exercice/M 3.2. Considérons ensuite l'ensemble $X = \mathbb{Z}_p$ pour un nombre premier impair $p \geq 3$. On se propose de calculer la signature de l'application affine $\alpha : \mathbb{Z}_p \to \mathbb{Z}_p$, $x \mapsto qx + r$ avec $q \in \mathbb{Z}_p^\times$ et $r \in \mathbb{Z}_p$.

(1) Décomposer $\sigma : x \mapsto x+1$ en cycles et en déduire $\operatorname{sign}(\sigma)$.

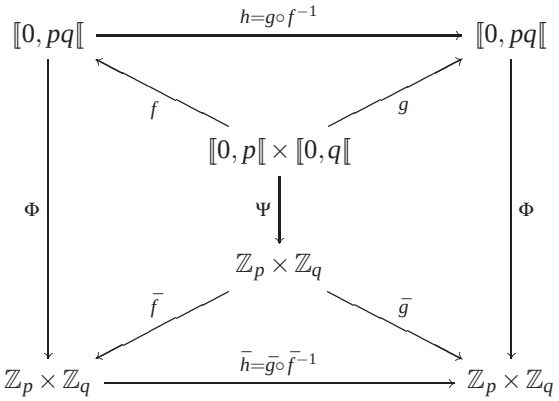
(2) Rappeler la structure du groupe multiplicatif $\mathbb{Z}_p^\times$.

(3) Pour une racine primitive g de $\mathbb{Z}_p^\times$, décomposer $\rho : x \mapsto gx$ en cycles et en déduire $\operatorname{sign}(\rho)$.

(4) Dans le cas général on a $q = g^k$, donc $\alpha = \sigma^r \rho^k$. En déduire $\operatorname{sign}(x \mapsto qx + r)$.

(5) En conclusion, exprimer $\operatorname{sign}(x \mapsto qx + r)$ par le symbole de Legendre.

3.3. La preuve de Zolotarev. Considérons deux nombres premiers impairs distincts $p, q \geq 3$. Pour l'intervalle $[\![0, pq[\![$ deux systèmes de numération mixte $f, g\colon [\![0, p[\![\times [\![0, q[\![\xrightarrow{\sim} [\![0, pq[\![$ viennent à l'esprit : d'une part $f(x,y) = x + py$, d'autre part $g(x,y) = qx + y$. Tous les deux sont des bijections, leur inverse étant donnée par la division euclidienne par p et par q respectivement. La composition $h = g \circ f^{-1}$ envoie $x + py$ sur $qx + y$. C'est la première ligne du diagramme suivant :



Par construction, h est une permutation. L'astuce de Zolotarev consiste à calculer la signature de h de deux manières différentes, pour en déduire la loi de réciprocité quadratique.

Exercice 3.3. On va d'abord exprimer $\operatorname{sign}(h)$ à l'aide des symboles de Legendre calculés ci-dessus. Pour cela on identifie $[\![0, p[\![\times [\![0, q[\![$ avec $\mathbb{Z}_p \times \mathbb{Z}_q$ via l'application $\Psi(x,y) = (\pi_p(x), \pi_q(y))$. L'application $\Phi\colon [\![0, pq[\![\xrightarrow{\sim} \mathbb{Z}_p \times \mathbb{Z}_q$ donnée par $\Phi(z) = (\pi_p(z), \pi_q(z))$ est une bijection par le théorème chinois. Ainsi nos fonctions f et g se traduisent en deux applications $\bar{f}\colon (\bar{x}, \bar{y}) \mapsto (\bar{x}, \overline{py + x})$ et $\bar{g}\colon (\bar{x}, \bar{y}) \mapsto (\overline{qx + y}, \bar{y})$ définies par les conditions que $\Phi \circ f = \bar{f} \circ \Psi$ et $\Phi \circ g = \bar{g} \circ \Psi$. Ceci se résume en disant que le diagramme précédent commute.

(1) Exprimer les signatures $\operatorname{sign}(\bar{f})$ et $\operatorname{sign}(\bar{g})$ par des symboles de Legendre.

(2) Expliquer pourquoi on a l'égalité $\operatorname{sign}(\bar{g} \circ \bar{f}^{-1}) = \operatorname{sign}(g \circ f^{-1})$.

Exercice 3.4. Ensuite on calcule $\operatorname{sign}(h)$ en comptant le nombre des inversions.

(1) Vérifier que

$$\begin{aligned} x + py < x' + py' \quad &\Longleftrightarrow \quad y < y' \text{ ou } (y = y' \text{ et } x < x'), \\ qx + y > qx' + y' \quad &\Longleftrightarrow \quad x > x' \text{ ou } (x = x' \text{ et } y > y'). \end{aligned}$$

(2) En déduire que $z = x + py$ et $z' = x' + py'$ vérifient $z < z'$ et $h(z) > h(z')$ si et seulement si $x > x'$ et $y < y'$. Compter le nombres des telles inversions, puis en déduire la signature de h.

(3) Établir la loi de réciprocité en mettant toutes les informations ensemble.

Exercice/M 3.5 (Question bonus)**.** Si vous voulez, vous pouvez réfléchir aux questions suivantes :

(1) Où utilise-t-on la primalité de p et q dans la preuve précédente ?

(2) A-t-on $\operatorname{sign}\left(\begin{smallmatrix} \mathbb{Z}_p \to \mathbb{Z}_p \\ x \mapsto qx + r \end{smallmatrix}\right) = \left(\frac{q}{p}\right)$ pour tout $p \geq 3$ impair ? (Symbole de Jacobi)

(3) Peut-on généraliser les arguments en supposant seulement que $\operatorname{pgcd}(p, q) = 1$?

(4) Comment calculer $\left(\frac{-1}{p}\right)$? Puis $\left(\frac{2}{p}\right)$? *Indication.* — On peut tenter la récurrence suivante :

$$\left(\frac{2}{m+2}\right) = \left(\frac{-m}{m+2}\right) = \pm\left(\frac{m}{m+2}\right) = \pm\left(\frac{m+2}{m}\right) = \pm\left(\frac{2}{m}\right).$$

(5) Comment généraliser $\left(\frac{q}{p}\right)$ et ce développement à p pair ? (Symbole de Kronecker)

Distinguer nombres premiers et nombres composés,
et décomposer ces derniers en facteurs premiers,
est un des problèmes les plus importants
et les plus utiles en arithmétique.
C.F. Gauss, *Disquisitiones Arithmeticae*, 1801

CHAPITRE XI

Primalité et factorisation d'entiers

Objectifs

Le théorème fondamental de l'arithmétique garantit que tout entier positif n s'écrit de manière unique comme produit $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ d'un certain nombre $k \geq 0$ de facteurs premiers $p_1 < p_2 < \cdots < p_k$ de multiplicités $e_1, e_2, \ldots, e_k \geq 1$. Étant donné un entier n, trois problèmes pratiques se posent :

(1) Déterminer rapidement si n est premier ou composé.

(2) Si n est premier, en trouver une preuve concise et facilement vérifiable.

(3) Si n est composé, trouver rapidement sa décomposition en facteurs premiers.

Pour les petits entiers ces problèmes sont faciles à résoudre. Par contre pour les grands entiers, déjà à partir de 30 décimales, les méthodes naïves échouent de manière catastrophique. Ce chapitre discutera quelques méthodes plus efficaces. Contrairement à ce que l'on pourrait penser, les trois problèmes sont bien distincts. Il se trouve que le premier admet de solutions efficaces, le deuxième aussi pourvu que l'on sache factoriser $n-1$, tandis que le troisième est en général très difficile.

Sommaire

Voici trois questions préliminaires qui nous serviront de fil conducteur :

Question 0.1. Comment prouver qu'un entier donné n est premier ? On pourrait tester tous les diviseurs possibles, … mais c'est hors de question si n est grand, comme 1 299 808 706 099 639 584 492 326 223 873. Existe-t-il une preuve de primalité qui soit concise et facile à vérifier ? Vous trouverez une réponse au §2.10. Le §4 esquisse une méthode récente et assez spectaculaire, qui résout le problème *théoriquement*.

Question 0.2. Comment prouver qu'un entier donné est composé ? C'est facile, dites-vous, il suffit d'en exhiber un facteur. Certes, de petits facteurs sont faciles à trouver par des essais successifs … mais ce n'est plus praticable pour des facteurs grands. Ainsi pour des cas comme $n = 24! - 1$ on constate que la recherche exhaustive est trop coûteuse. Nous regarderons des critères plus efficaces au §2.

Question 0.3. Après s'être convaincu qu'un entier donné, disons $n = 24! - 1$, est composé de grands facteurs, on revient à la question de factorisation : comment trouver efficacement la décomposition en facteurs premiers ? Ceci peut être une tâche très dure. Au §3 nous présentons une idée simple et amusante, qui permet de trouver des facteurs de taille moyenne, disons entre 10^6 et 10^{12}.

1. Méthodes exhaustives

1.1. Primalité. Commençons par la méthode évidente pour tester la primalité d'un entier :

Algorithme XI.1 Test de primalité par essais exhaustifs (non optimisé)

Entrée: un nombre naturel $n \geq 2$
Sortie: le plus petit facteur premier de n

$r \leftarrow \lfloor \sqrt{n} \rfloor$
pour d **de** 2 **à** r **faire si** $d \mid n$ **alors retourner** d
retourner n

Exercice/M 1.1. Justifier cet algorithme : bien que d parcoure nombres premiers et composés, pourquoi peut-on assurer que le facteur trouvé soit premier ? Que se passe-t-il pour n premier ?

Exercice/P 1.2. Vous pouvez déjà écrire une fonction `bool estPremier( const Integer& n )` qui teste si `n` est un nombre premier par des essais successifs. Veillez tout particulièrement aux cas $n = 0, \pm 1, \pm 2$.

Optimisation. — On peut économiser 50% du temps, en procédant, à partir de $d = 3$, par pas de $+2$. On peut encore économiser 33% du temps en procédant, à partir de $d = 5$, par pas de $+2, +4, +2, +4, \ldots$. D'autres optimisations analogues sont possibles mais de plus en plus complexes et de moins en moins efficaces (le détailler). On développera une optimisation plus systématique avec le crible d'Ératosthène plus bas (§1.4).

Exercice/M 1.3. Quel est le nombre d'itérations dans le meilleur des cas ? dans le pire des cas ? Jusqu'à quel n environ ce test est-il raisonnable ? Peut-on ainsi déterminer la nature de 1 219 326 331 002 895 961 ou de 1 219 326 331 002 895 901 ? (On les analysera dans l'exercice 2.32 et 3.17 plus bas.)

1.2. Factorisation. Regardons ensuite la méthode évidente de factorisation :

Algorithme XI.2 Factorisation par essais exhaustifs (non optimisé)

Entrée: un nombre naturel $n \geq 2$
Sortie: l'unique décomposition $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ en facteurs premiers
$p_1 < p_2 < \cdots < p_k$ avec multiplicités $e_1, e_2, \ldots, e_k \geq 1$

$r \leftarrow \lfloor \sqrt{n} \rfloor$, $p \leftarrow 2$, $e \leftarrow 0$
tant que $p \leq r$ **faire**
  tant que $p \mid n$ **faire** $n \leftarrow n/p$, $e \leftarrow e+1$
  si $e > 0$ **alors** rajouter p^e à la factorisation, puis recalculer $r \leftarrow \lfloor \sqrt{n} \rfloor$, $e \leftarrow 0$
  $p \leftarrow p+1$
fin tant que
si $n > 1$ **alors** rajouter le facteur premier n à la factorisation

Exercice/M 1.4. Justifier cet algorithme : pourquoi ne produit-il que des facteurs premiers ? Pourquoi un éventuel facteur restant $n > 1$ est-il premier ?

Exercice/P 1.5. Vous pouvez déjà écrire une fonction `vector<Integer> factorisation( Integer n )` qui calcule la factorisation $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ et renvoie le vecteur $(p_1, e_1; p_2, e_2; \ldots; p_k, e_k)$. Pour faire joli, vous pouvez ajouter une fonction `void affiche( vector<Integer>& dec )` qui permet d'afficher une décomposition comme `2^3·3^2·5·17`.

Optimisation. — Vous pouvez implémenter les optimisations indiquées en exercice 1.2. Y a-t-il une possibilité de n'effectuer qu'une seule division euclidienne pour tester $p \mid n$ et déduire le quotient n/p le cas échéant ? (Est-ce une optimisation importante ?) Plus bas on optimisera par le crible d'Ératosthène (§1.4), et plus tard on ajoutera le test de primalité de Miller-Rabin (§2.5).

Exercice/M 1.6. Quelle est la complexité en nombre d'itérations de la factorisation par divisions successives : dans le meilleur des cas ? dans le pire des cas ? Jusqu'à quel n environ cette méthode est-elle raisonnable ? Peut-on ainsi factoriser 24! – 1 par exemple ? (On factorisera cet exemple dans l'exercice 3.16.)

Exercice/P 1.7. On appelle *nombre de Mersenne* $M_k = 2^k - 1$ avec $k \geq 2$. Décomposer les nombres $M_2, \ldots, M_{60}$ en facteurs premiers. Que se passe-t-il pour M_{61} ? Expliquer le ralentissement observé. (On reprendra cet exemple dans l'exercice 2.34 plus bas).

1.3. Complexité. Essayons de préciser la complexité asymptotique des deux méthodes exhaustives ci-dessus. Soulignons d'abord qu'il est facile de *poser* un problème de primalité ou de factorisation : pour cela il suffit de présenter le nombre n à analyser, disons en système binaire. Son écriture est ainsi de longueur $\ell = \text{len}(n)$ avec $2^{\ell-1} \leq n < 2^\ell$. Cette longueur est une mesure adéquate pour la complexité du nombre n : c'est la mémoire nécessaire pour le stocker et aussi le temps nécessaire pour le transmettre.

Proposition 1.8. *Soit $c(\ell)$ la complexité en nombre d'itérations dans le pire cas quand on applique les méthodes exhaustives décrites ci-dessus aux entiers de longueur ℓ. Alors $c(\ell)$ est exponentielle en ℓ.*

Exercice/M 1.9. Détailler cette proposition. Après réflexion, il reste tout de même une question sur la répartition des nombres premiers : est-ce que le pire des cas se produit réellement pour une longueur ℓ donnée ? C'est effectivement le cas : le « postulat de Bertrand », démontré par Tchebycheff en 1850, affirme qu'il existe au moins un nombre premier dans l'intervalle $[\![a, 2a]\!]$ quel que soit $a \geq 1$. (Il en existe même beaucoup plus, comme vous pouvez déduire du théorème 1.17 ci-dessous.)

1.4. Le crible d'Ératosthène. Rappelons un des plus anciens théorèmes des mathématiques :

Théorème 1.10. *Il existe une infinité de nombres premiers.*

Vous êtes cordialement invités à redémontrer ce joli résultat. Après ce constat fondamental, considérons la question pratique : comment construire la liste de *tous* les nombres premiers $\leq m$? Ératosthène de Kyrène (environ 275–194 avant notre ère) développa une méthode qui est connue sous le nom de *crible d'Ératosthène*. Dans la version originale on écrit $2, 3, 4, 5, \ldots, n$ puis on raye les multiples de $2, 3, 5, \ldots$. Sur ordinateur ceci occupe trop d'espace. L'algorithme XI.3 ci-dessous en est une variante plus économe :

Algorithme XI.3 Une variante du crible d'Ératosthène

Entrée: la liste $(p_0 = 2, p_1 = 3, \ldots, p_{k-1})$ des k plus petits nombres premiers, avec $k \geq 2$
Sortie: la liste $(p_0 = 2, p_1 = 3, \ldots, p_{k-1}, p_k)$ des $k+1$ plus petits nombres premiers

$p \leftarrow p_{k-1} + 2$, $i \leftarrow 1$
tant que $p_i^2 \leq p$ **faire**
  si $p_i \nmid p$ **alors** $i \leftarrow i + 1$ **sinon** $p \leftarrow p + 2$, $i \leftarrow 1$
fin tant que
retourner la liste prolongée $(p_0 = 2, p_1 = 3, \ldots, p_{k-1}, p)$

Exercice/P 1.11. Prouver la correction de l'algorithme XI.3 puis l'implémenter en une fonction `void ajoutePremier( vector<int>& liste )`. Peut-on ainsi construire, dans un temps raisonnable, la liste des nombres premiers jusqu'à 10^6 ? jusqu'à 10^7 ? jusqu'à 10^8 ? jusqu'à 10^9 ?

Exercice 1.12. Pour $n \in \mathbb{N}$ on note $\pi(n)$ le nombre des entiers premiers dans l'intervalle $[\![1, n]\!]$. Écrire un programme qui lit n au clavier et qui affiche $\pi(n)$. Dans la mesure du possible vérifier les valeurs du tableau suivant (dans lequel au moins une erreur s'est glissée).

k	$\pi(10^k)$	k	$\pi(10^k)$	k	$\pi(10^k)$	k	$\pi(10^k)$
1	4	6	78 498	11	4 118 054 813	16	279 238 341 033 925
2	25	7	664 579	12	37 607 912 018	17	2 623 557 157 654 233
3	168	8	5 761 455	13	346 065 536 839	18	24 739 954 287 740 860
4	1 129	9	50 847 534	14	3 204 941 750 802	19	234 057 667 276 344 607
5	9 592	10	455 052 511	15	29 844 570 422 669	20	2 220 819 602 560 918 840

1.5. Factorisation limitée. Dans un programme on procède typiquement comme suit : on fixe une borne m et construit, une fois pour toute, la liste des petits nombres premiers $p_0, \ldots, p_k \leq m$. Selon le tableau ci-dessus, un choix entre 10^7 et 10^8 semble raisonnable, donc le type `int` suffira.

Exercice/M 1.13. En reprenant les algorithmes XI.1 et XI.2, expliquer pourquoi il est avantageux de parcourir seulement la liste $p_0, \ldots, p_k$. Vérifier que le test de primalité reste correct pour tout $n \leq m^2$. De même, la factorisation est complète si le facteur restant satisfait $n \leq m^2$. Si $n > m^2$ on peut au moins garantir que n n'a pas de petits facteurs, mais on ne peut pas conclure que n soit premier.

Exercice/P 1.14. On se contente, dans un premier temps, d'une factorisation limitée $n = p_{i_1}^{e_1} p_{i_2}^{e_2} \cdots p_{i_\ell}^{e_\ell} \cdot \hat{n}$ de sorte que le facteur restant $\hat{n}$ n'ait pas de diviseurs $\leq m$. L'implémenter en une fonction

```
vector<Integer> factorisation( Integer& n, const vector<int>& premiers )
```

qui extrait les petits premiers stockés dans la liste `premiers`. Ici la fonction remplace n par le facteur restant $\hat{n}$: celui-ci vaut 1 si la factorisation est complète, et > 1 si la factorisation laisse un facteur de nature indéterminée. (Justifier le mode de passage des deux paramètres.)

Optimisation. — Plus bas, dans l'exercice 2.33, on ajoutera le test de primalité selon Miller-Rabin. Ceci sert à compléter la factorisation dans les cas ou le facteur restant $\hat{n}$ est premier.

Exercice/P 1.15. Donner les factorisations limitées de $n!+1$, pour $n \in [\![1, 100]\!]$ disons, en indiquant les cas qui laissent un facteur de nature indéterminée. Si vous voulez vous pouvez regarder d'autres familles d'exemples comme $b \cdot n! + c$ ou $b^n + c$ avec $b \geq 2$ et $c \in \mathbb{Z}$ fixés. La deuxième famille inclut en particulier les nombres de Mersenne $2^n - 1$ et les nombres de Fermat $2^{2^n} + 1$.

1.6. Le théorème des nombres premiers. Pour trouver la valeur exacte de $\pi(n)$ il n'y a que le comptage fastidieux, plus ou moins optimisé. Deux questions évidentes se posent : peut-on approcher $\pi(n)$ par une fonction simple ? Quel est son comportement asymptotique ?

En regardant les tableaux des nombres premiers jusqu'à 10^6, qui venaient d'être publiés entre 1770 et 1811, Gauss conjectura que $\pi(n) \sim \frac{n}{\ln n}$. À la même époque Legendre proposa l'approximation $\frac{n}{\ln n - 1}$, ce qui correspond mieux aux valeurs $\pi(n)$ pour $10^4 \leq n \leq 10^{20}$ (les comparer). Bien entendu, le comportement asymptotique de ces deux approximations est le même. Basé sur les idées fondamentales de Riemann (1859), la conjecture de Gauss et Legendre fut finalement démontrée, indépendamment, par Hadamard et de la Vallée Poussin (1896) :

Théorème 1.16 (théorème des nombres premiers, version qualitative)**.** *On a l'équivalence asymptotique* $\pi(n) \sim \frac{n}{\ln n}$*, c'est-à-dire le quotient* $\frac{\pi(n)}{n/\ln n}$ *converge vers* 1 *pour* $n \to \infty$*. Autrement dit, la proportion des nombres premiers parmi les nombres dans l'intervalle* $[\![1, n]\!]$ *est à peu près* $\frac{1}{\ln n}$*.* □

La version suivante donne un encadrement étonnamment précis, dû à J. Rosser et L. Schoenfeld, *Approximate formulas for some functions of prime numbers*, Illinois Journal of Mathematics 6 (1962) 64–94.

Théorème 1.17 (théorème des nombres premiers, version quantitative)**.** *Pour* $n \geq 59$ *on a l'encadrement*

$$\frac{n}{\ln n}\left(1 + \frac{1}{2\ln n}\right) \quad < \quad \pi(n) \quad < \quad \frac{n}{\ln n}\left(1 + \frac{3}{2\ln n}\right). \tag{1}$$

Pour mieux apprécier ce résultat, on comparera avec profit le comptage exact avec $\frac{n}{\ln n}$ et l'encadrement. Tandis que tout comptage s'arrête forcément assez tôt, l'encadrement est valable éternellement !

Afin d'illustrer la fascination que provoque le théorème des nombres premiers, citons le résumé donné par Don Zagier dans son article "The first 50 million primes", Mathematical Intelligencer (1977) 1-19 :

> There are two facts about the distribution of prime numbers of which I hope to convince you so overwhelmingly that they will be permanently engraved in your hearts. The first is that, despite their simple definition and role as the building blocks of the natural numbers, the prime numbers (...) grow like weeds among the natural numbers, seeming to obey no other law than that of chance, and nobody can predict where the next one will sprout. The second fact is even more astonishing, for it states just the opposite : that the prime numbers exhibit stunning regularity, that there are laws governing their behaviour, and that they obey these laws with almost military precision.

Exercice/M 1.18. Expliquer pourquoi l'intervalle $[\![n, n+k[\![$ contient à peu près $k/\ln n$ nombres premiers. Quand on choisit un grand entier de manière aléatoire, quelle est la probabilité que l'on tombe sur un premier ? La formulation est volontairement provocatrice ; lui donner un sens mathématique précis.

Exercice/M 1.19. L'encadrement (1) du théorème 1.17 est-il suffisamment précis pour détecter la faute de frappe dans le tableau de l'exercice 1.12 ?

Exercice/M 1.20. L'encadrement (1) permet de déduire le postulat de Bertrand. Voyez-vous comment ? Peut-on obtenir ce résultat à partir de la version qualitative $\pi(n) \sim \frac{n}{\ln n}$ seulement ?

2. Le critère probabiliste selon Miller-Rabin

La situation. — Étant donné un nombre naturel n, la factorisation limitée (§1.5) permet d'extraire efficacement les petits facteurs premiers (s'il y en a). Dans certains cas favorables, on arrive ainsi à une factorisation complète. Sinon, il faut tôt ou tard abandonner la recherche exhaustive. Au moins on assure ainsi que le facteur restant n'a plus de petits facteurs.

Le problème. — Il nous reste à déterminer si un entier n donné, sans petits facteurs, est premier ou composé. À première vue la situation semble désespérée. Fort heureusement le critère de Miller-Rabin, développé dans la suite, permet un test efficace, couramment utilisé dans la pratique. Il s'agit d'ailleurs d'une belle application de la structure du groupe $\mathbb{Z}_n^\times$ (chapitre IX, §1).

Comment s'y prendre ? — En principe on pourrait tout de suite énoncer le critère de Miller-Rabin (le lemme 2.16 et le théorème 2.20 plus bas) et passer directement à l'implémentation (§2.8). Mais une telle démarche ne serait ni motivée ni motivante, et l'origine de l'énoncé resterait obscur. Il est plus naturel de développer ce critère en cinq petites étapes, amplement illustrées d'exemples. Cette démarche retrace le développement historique et logique de ce critère fort utile.

2.1. Éléments non inversibles dans $\mathbb{Z}_n^*$. Rappelons qu'un entier $n \geq 2$ est premier si et seulement si $\mathbb{Z}_n^* = \mathbb{Z}_n \smallsetminus \{0\}$ est un groupe (d'ordre $n-1$) pour la multiplication. Par contraposé : étant donné n, il suffit d'exhiber un élément nun nul $x \in \mathbb{Z}_n^*$ mais non inversible pour prouver que n est composé. Est-il possible d'en trouver un assez rapidement ? Malheureusement cette idée se révèle peu praticable : il y a simplement trop peu de tels éléments !

Exercice/M 2.1. Supposons que n est composé, avec factorisation $n = p_1^{e_1} \cdots p_k^{e_k}$. Vérifier que la probabilité qu'un élément $x \in \mathbb{Z}_n$ choisi au hasard soit inversible vaut $\frac{\varphi(n)}{n} = \prod_{i=1}^{i=k}(1 - \frac{1}{p_i})$. Si n n'a pas de petits facteurs, alors la probabilité de tomber sur un élément non inversible est proche de 0. Expliquer, en choisissant n, pourquoi elle peut même être arbitrairement petite.

Remarque 2.2. Trouver un élément $x \in \mathbb{Z}_n^*$ non inversible revient à trouver un facteur de n : en représentant x par $\tilde{x} \in \mathbb{Z}$, on peut rapidement calculer $d = \operatorname{pgcd}(n, \tilde{x})$, qui est un facteur de n vérifiant $1 < d < n$. Ainsi tomber sur un élément non inversible de $\mathbb{Z}_n^*$ est aussi probable que deviner un facteur de n.

2.2. Le test de Fermat. Le petit théorème de Fermat (voir le chapitre X, §1) affirme que pour n premier tout élément $x \in \mathbb{Z}_n^\times$ vérifie $x^{n-1} = 1$. Ceci peut servir à tester la primalité d'un nombre n donné :

Proposition 2.3. *Si $x \in \mathbb{Z}_n^*$ vérifie $x^{n-1} \neq 1$, alors n est composé. Dans ce cas on dit que x est un* témoin de décomposabilité *de n, ou aussi que x* témoigne contre la primalité *de n. À noter qu'un tel témoin prouve que n est composé sans pour autant donner une quelconque information sur les facteurs de n.* □

Remarque 2.4. Le critère que $x^{n-1} = 1$ est nécessaire pour la primalité de n mais non suffisante. Soulignons donc que $x^{n-1} = 1$ ne prouve en rien que n soit premier. Par exemple, pour $n = 15$ on trouve que $2^{14} \equiv 4 \pmod{15}$, donc 2 est un témoin. Par contre $4^{14} \equiv 1 \pmod{15}$, donc 4 n'est pas un témoin. Il s'agit d'une asymétrie fondamentale : ce genre de test peut prouver que n est composé, mais il ne peut pas prouver que n est premier.

La question cruciale pour toute application pratique est la suivante : étant donné un nombre composé n, peut-on rapidement trouver un témoin $x \in \mathbb{Z}_n^*$?

Remarque 2.5 (le principe probabiliste)**.** On ne sait pas prédire quels éléments vont témoigner. On choisit donc $x_1, x_2, \ldots, x_k \in \mathbb{Z}_n^*$ au hasard et les teste un par un. Si l'on trouve un témoin, ceci prouve que n est composé. Sinon, n est possiblement premier mais on ne peut être sûr.

Pour que cette approche soit intéressante, il faut assurer un taux de réussite assez grand. Notre question devient donc : si l'on choisit a au hasard, quelle est la probabilité que a témoigne contre la primalité de n ? Déjà les éléments non inversibles ne satisfont pas $x^{n-1} = 1$, mais ce cas est trop peu probable si n n'a pas de petit facteur (voir plus haut). Heureusement, le test de Fermat augmente considérablement nos chances ! Expérimentez avec le programme `temoins.cc` pour vous convaincre que ce test peut être assez efficace dans certains cas.

2.3. Nombres de Carmichael. Après la première euphorie, voici la mauvaise nouvelle :

Remarque 2.6 (Carmichael)**.** Il existe des nombres composés n tel que $a^n \equiv a \pmod{n}$ pour tout $a \in \mathbb{Z}$. En particulier, tout élément inversible $x \in \mathbb{Z}_n^\times$ vérifie $x^{n-1} = 1$. On appelle un tel n un *nombre de Carmichael*. Les plus petits exemples sont 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, … On sait depuis 1994 qu'il en existe une infinité.

Exercice/M 2.7. Établissons que les nombres de Carmichael sont la seule obstruction pour le test de Fermat. Si n est composé mais non un nombre de Carmichael, alors l'ensemble $\{x \in \mathbb{Z}_n^\times \mid x^{n-1} = 1\}$ forme un sous-groupe d'indice ≥ 2 dans $\mathbb{Z}_n^\times$. Par conséquent, le test de Fermat trouve un témoin avec probabilité d'échec $< \frac{1}{2}$. En itérant ce test k fois, la probabilité d'échec devient $< 2^{-k}$, donc arbitrairement petite.

Exercice/M 2.8. Vérifier que $n = 561 = 3 \cdot 11 \cdot 17$ est effectivement un nombre de Carmichael : montrer que le groupe $\mathbb{Z}_{561}^\times \cong \mathbb{Z}_3^\times \times \mathbb{Z}_{11}^\times \times \mathbb{Z}_{17}^\times$ est d'ordre 320, et que tout $x \in \mathbb{Z}_n^\times$ vérifie $x^{80} = 1$. Par conséquent $x^{560} = 1$ et $x^{561} = x$ pour tout $x \in \mathbb{Z}_{561}^\times$, et cette dernière égalité tient même pour tout $x \in \mathbb{Z}_{561}$. Même calcul pour les nombres suivants : $1105 = 5 \cdot 13 \cdot 17$, puis $1729 = 7 \cdot 13 \cdot 19$, puis $2821 = 7 \cdot 13 \cdot 31$, …

Exercice/M 2.9. Soit $n = pqr$ avec $p = 6k+1$ et $q = 12k+1$ et $r = 18k+1$ premiers. (Pour $k = 1$ par exemple on obtient $n = 1729$.) Montrer que n est un nombre de Carmichael : tout $x \in \mathbb{Z}_n^\times$ vérifie $x^{36k} = 1$ et $36k$ divise $n-1$. En supposant qu'il existe une infinité de tels nombres, déduire que la probabilité pour trouver un témoin avec le test de Fermat peut être arbitrairement petite.

Exercice/M 2.10. Supposons que $n = p_1 \cdots p_k$ avec des facteurs premiers $p_1 < \cdots < p_k$ de sorte que $p_i - 1$ divise $n-1$ pour tout i. Vérifier que n est un nombre de Carmichael. Réciproquement, tout nombre de Carmichael est forcément de cette forme-ci, impair, et composé d'au moins trois facteurs premiers. *Indication.* — On appliquera le théorème chinois et le fait que le groupe $\mathbb{Z}_{p^e}^\times$ est cyclique d'ordre $(p-1)p^{e-1}$.

2.4. L'astuce de la racine carrée. Le test de Fermat a ses défauts, mais il est trop beau pour être abandonné. Essayons donc de l'optimiser. Le lemme suivant explicite l'observation clé :

Lemme 2.11. *Soit p un nombre premier impair. Alors -1 est le seul élément d'ordre 2 dans $\mathbb{Z}_p^\times$.*

DÉMONSTRATION. Effectivement, -1 est d'ordre 2. Réciproquement, si x est d'ordre 2, alors $x^2 = 1$. Ceci implique $0 = x^2 - 1 = (x-1)(x+1)$, ce qui n'est possible que pour $x - 1 = 0$ ou $x + 1 = 0$, donc $x = \pm 1$. Comme $+1$ est d'ordre 1, on conclut que -1 est le seul élément d'ordre 2. □

Exercice/M 2.12. Si n est composé le groupe $\mathbb{Z}_n^\times$ peut contenir plusieurs éléments d'ordre 2. Dans $\mathbb{Z}_8$, par exemple, les éléments $3, 5, 7$ sont tous d'ordre 2.

Regardons $n = pq$ avec deux premiers $p > q > 2$. Par le théorème chinois nous avons un isomorphisme d'anneaux $\psi\colon \mathbb{Z}_p \times \mathbb{Z}_q \xrightarrow{\sim} \mathbb{Z}_n$. Dans $\mathbb{Z}_p^\times \times \mathbb{Z}_q^\times$ les éléments d'ordre 2 sont $(-1,-1)$, $(+1,-1)$ et $(-1,+1)$. Dans $\mathbb{Z}_n^\times$ nous avons donc également trois éléments d'ordre 2 : outre la solution standard $\psi(-1,-1) = -1$ nous trouvons deux autres éléments, $\psi(+1,-1)$ et $\psi(-1,+1)$. Les expliciter pour $n = 15$.

Exercice/M 2.13. D'après le théorème 1.13 énoncé au chapitre IX, le groupe $\mathbb{Z}_{p^e}^\times$ est cyclique d'ordre $(p-1)p^{e-1}$ pour tout premier $p \geq 3$ et exposant $e \geq 1$. En déduire que -1 est le seul élément d'ordre 2 dans $\mathbb{Z}_{p^e}^\times$. Pour $n = p_1^{e_1} \cdots p_k^{e_k}$ montrer qu'il existe exactement $2^k - 1$ éléments d'ordre 2 dans $\mathbb{Z}_n^\times$. Comment les expliciter ?

Le lemme précédent mène directement à un test de primalité un peu plus raffiné. Afin de l'illustrer vous pouvez analyser des exemples avec le programme `temoins.cc`.

Proposition 2.14. *Si n est un nombre premier impair, alors tout $x \in \mathbb{Z}_n^*$ vérifie $x^{\frac{n-1}{2}} = \pm 1$.* □

Exemple 2.15 (Carmichael, suite). Comme remarqué plus haut, $n = 561$ est un nombre de Carmichael : les 320 éléments inversibles $x \in \mathbb{Z}_n^\times$ satisfont tous au test de Fermat $x^{n-1} = 1$. Par contre, seuls 160 satisfont au critère $x^{\frac{n-1}{2}} = \pm 1$. Le test $x^{\frac{n-1}{2}} = \pm 1$ est donc plus fin que le test $x^{n-1} = 1$. (Voir le programme `temoins.cc`.)

Hélas, ce test raffiné n'est pas infaillible : dans le cas $n = 1729$ tous les éléments inversibles $x \in \mathbb{Z}_n^\times$ satisfont non seulement au test $x^{n-1} = 1$, mais aussi au test $x^{\frac{n-1}{2}} = \pm 1$. Comment sauver la situation ? En l'occurrence, remarquons que l'exposant $\frac{n-1}{2} = 864$ est un nombre pair. Si $x^{864} = +1$, on peut donc tester de plus si $x^{432} = \pm 1$: ce n'est rien d'autre que notre lemme, appliqué une deuxième fois. Si l'on trouve

$x^{432} = +1$, on peut tester si $x^{216} = \pm 1$, et ainsi de suite. On vient de découvrir un test encore plus fin : c'est la méthode de Miller-Rabin !

2.5. Le test de Miller-Rabin. Vers la fin des années 1970, G. Miller puis M. Rabin ont proposé un test de primalité très puissant. Il découle directement de notre développement précédent :

Lemme 2.16. *Soit $n \geq 3$ un entier impair. On décompose $n-1 = 2^e q$ avec $e \geq 1$ et q impair. Si n est premier, alors tout $x \in \mathbb{Z}_n^*$ satisfait ou $x^q = 1$ ou bien il existe $k \in [\![0, e[\![$ tel que $x^{2^k q} = -1$.*

Définition 2.17. On dit que $x \in \mathbb{Z}_n^*$ passe le test de Miller-Rabin si $x^q = 1$ ou $x^{2^k q} = -1$ pour un $k \in [\![0, e[\![$. Dans ce cas on dit aussi que n est *pseudo-premier* à base x. (À noter que ceci ne veut pas dire que n soit premier.) Réciproquement, si $x \in \mathbb{Z}_n^*$ ne passe pas le test de Miller-Rabin, alors on dit que x est un *témoin de décomposabilité* de n au sens de Miller-Rabin, ou aussi que *x témoigne contre la primalité* de n.

Exercice/M 2.18. Montrer le lemme et expliquer en quoi il est un raffinement du test de Fermat. Vérifier que l'algorithme suivant est une traduction fidèle du critère de Miller-Rabin :

Algorithme XI.4 Test de pseudo-primalité selon Miller-Rabin

Entrée: un entier impair $n \geq 5$ et un entier $x \in [\![2, n-2]\!]$.
Sortie: le message « pseudo-premier » ou « composé »

décomposer $n-1 = 2^e q$ avec $e \geq 1$ et q impair	// divisions itérées
calculer $y \leftarrow x^q \bmod n$	// puissance dichotomique
si $y = 1$ **alors retourner** « pseudo-premier »	// x passe tous les tests car $x^q \equiv 1$.
pour k **de** 1 **à** e **faire**	
si $y = n-1$ **alors retourner** « pseudo-premier »	// premier cas : $x^{2^{k-1}q} \equiv -1$, ça passe
$y \leftarrow y^2 \bmod n$	// après ce calcul on a $y = x^{2^k q} \bmod n$
si $y = 1$ **alors retourner** « composé »	// second cas : $x^{2^{k-1}q} \not\equiv \pm 1$ mais $x^{2^k q} \equiv 1$
fin pour	
retourner « composé »	// x ne passe même pas le test de Fermat.

Exercice/M 2.19. Rassurons-nous que la complexité algorithmique du critère de Miller-Rabin est tout à fait raisonnable. Soit n un entier de longueur $\ell = \mathrm{len}(n)$ en base 2. Rappelons d'abord qu'une multiplication modulo n est de complexité $O(\ell^2)$ avec la méthode scolaire, voire $O^+(\ell)$ avec une méthode plus sophistiquée. Vérifier que le test de Miller-Rabin nécessite moins de 2ℓ multiplications pour calculer x^q et ses carrés successifs. Au total, tester si $x \in [\![2, n-2]\!]$ satisfait au critère de Miller-Rabin est de complexité $O(\ell^3)$ voire $O^+(\ell^2)$.

2.6. Probabilité d'erreur. A priori le critère de Miller-Rabin est plus fin mais aussi un peu plus complexe que les critères précédents. Son intérêt réside dans le beau résultat suivant, montré en 1980 indépendamment par M. Rabin et L. Monier. Il garantit une grande probabilité de réussite :

Théorème 2.20. *Soit $n \geq 3$ impair. Si n est premier, alors tout $x \in \mathbb{Z}_n^*$ satisfait au critère de Miller-Rabin. Si n est composé, alors il y a au moins $\frac{3}{4}(n-1)$ témoins $x \in \mathbb{Z}_n^*$ au sens de Miller-Rabin.* □

Exemple 2.21 (Carmichael, suite et fin). Pour $n = 1729 = 7 \cdot 13 \cdot 19$ le groupe $\mathbb{Z}_n^\times$ est d'ordre $6 \cdot 12 \cdot 18 = 1296$. Tous les éléments $x \in \mathbb{Z}_n^\times$ satisfont aux tests $x^{n-1} = 1$ et $x^{\frac{n-1}{2}} = \pm 1$. Par contre, seuls 162 passent le test de Miller-Rabin. Autrement dit, il existe 1566 témoins, soit plus de 90%, conformément au théorème. (Voir `temoins.cc` .)

Corollaire 2.22. *Pour tester si un nombre n est composé on choisit $x_1, x_2, \ldots, x_k \in \mathbb{Z}_n^*$ au hasard et effectue le test de Miller-Rabin pour chacune de ces bases. Si n est premier, l'algorithme répondra toujours que n est pseudo-premier à base $x_1, x_2, \ldots, x_k$. Si par contre n est composé, alors l'algorithme répondra pseudo-premier à base $x_1, x_2, \ldots, x_k$ avec une probabilité $\leq 4^{-k}$; autrement dit, avec une probabilité $\geq 1 - 4^{-k}$ on trouve un témoin x_i qui prouve que n est composé.* □

Remarque 2.23. Supposons que nous soumettons un nombre n donné à k tests aléatoires de Miller-Rabin, qui répondent k fois « pseudo-premier ». Que peut-on en déduire ? On lit souvent la formulation suivante : « Le nombre n est premier avec probabilité $\geq 1 - 4^{-k}$. » Strictement parlant, cette phrase n'a aucun sens :

le nombre n est ou premier ou composé, c'est une propriété de n et non une question de probabilité. L'affirmation prend un sens quand on la reformule plus précisément, comme dans le corollaire précédent : c'est l'algorithme qui est probabiliste, et il reste une certaine probabilité d'erreur.

Exercice/M 2.24. Dans le test de Miller-Rabin la seule erreur possible est une conclusion erronée « probablement premier » alors que n est composé. Cette erreur se produit avec une probabilité inférieure à 4^{-k}. Feriez-vous confiance en un résultat qui est correct avec une probabilité d'erreur $\leq 4^{-k}$ pour $k = 10$? pour $k = 30$? pour $k = 100$? Comparer la probabilité 4^{-100} avec la probabilité de gagner plusieurs fois consécutives au loto. Contempler la conclusion d'Émile Borel (*Les probabilités et la vie*, 1943) : « Un phénomène dont la probabilité est 10^{-50} ne se produira donc jamais, ou de moins ne sera jamais observé. » Qu'en pensez-vous ?

Exercice/M 2.25. Tous nos calculs sont effectués sur des machines réelles, donc imparfaites : la probabilité ε que les circuits électriques produisent une erreur est très petite mais elle est certainement non nulle (ne mentionnons que mouvement thermique, radiation extérieure, effets quantiques). Par conséquent, même le résultat d'un algorithme parfaitement déterministe peut être erroné avec une probabilité d'erreur $\varepsilon > 0$ quand on l'exécute sur une machine imparfaite. Qu'estimez-vous, pour quel k a-t-on $4^{-k} \approx \varepsilon$? Sous cet angle rediscuter la sûreté du test de Miller-Rabin.

2.7. Un test optimisé. Le test de Miller-Rabin remédie à tous les défauts du test de Fermat, en particulier il reconnaît aisément les nombres de Carmichael comme composés. On verra dans ce paragraphe qu'il arrive même à les factoriser !

Comme remarqué au §2.1, tout élément non inversible $\bar{x} \in \mathbb{Z}_n^*$ nous fait cadeau d'un facteur de n, simplement en calculant $\operatorname{pgcd}(n,x)$. Dans ce sens l'observation suivante peut être utile :

Lemme 2.26. *Si $y \in \mathbb{Z}_n$ vérifie $y^2 = 1$ mais $y \neq \pm 1$, alors les éléments $y \pm 1 \in \mathbb{Z}_n^*$ sont non inversibles.*

DÉMONSTRATION. Comme on a vu dans le lemme 2.11, trouver un tel y prouve que n est composé. Supposons $n = pq$ pour simplifier. D'après le théorème chinois nous disposons d'un isomorphisme d'anneaux $\phi\colon \mathbb{Z}_n \xrightarrow{\sim} \mathbb{Z}_p \times \mathbb{Z}_q$. Notre élément y s'envoie donc sur $\phi(y) = (+1,-1)$ ou $\phi(y) = (-1,+1)$. C'est une information précieuse : $y+1$ correspond à $(2,0)$ ou $(0,2)$, et $y-1$ correspond à $(0,-2)$ ou $(-2,0)$. Ce sont donc des éléments non nuls mais non inversibles. □

Exercice/M 2.27. Compléter la démonstration précédente et en déduire l'algorithme XI.5 ci-dessous. Le principe est le même que le test de Miller-Rabin (algorithme XI.4). L'aspect nouveau est qu'on essaie d'exploiter d'éventuels éléments non inversibles qui peuvent apparaître au cours des calculs.

Algorithme XI.5 Test de primalité selon Miller-Rabin, version étendue

Entrée: un entier impair $n = 2^e q + 1$ et un entier $x \in [\![2, n-2]\!]$.

Sortie: le message « pseudo-premier » ou « composé » ou un facteur non trivial de n

$d \leftarrow \operatorname{pgcd}(n,x)$	// rapide avec l'algorithme d'Euclide.
si $d > 1$ **alors retourner** d	// x non inversible donne un facteur de n.
$y \leftarrow x^q \bmod n$	// rapide avec la puissance dichotomique.
si $y = 1$ **alors retourner** « pseudo-premier »	// x passe tous les tests car $x^q \equiv 1$.
pour k **de** 1 **à** e **faire**	
si $y = n-1$ **alors retourner** « pseudo-premier »	// Premier cas : x passe le test de Miller-Rabin.
$z \leftarrow y$, $y \leftarrow y^2 \bmod n$	// On remplace y par y^2 mais on stocke la racine.
si $y = 1$ **alors retourner** $\operatorname{pgcd}(n, z+1)$	// Second cas : ici $z \not\equiv \pm 1$ mais $z^2 \equiv 1$.
fin pour	
retourner « composé »	// x ne passe même pas le test de Fermat.

L'algorithme ci-dessus trouve un facteur de n chaque fois que x passe le test de Fermat modulo n mais non le test de Miller-Rabin. Ainsi les nombres de Carmichael, considérés plus haut comme le pire cas pour le test de Fermat, sont particulièrement faciles à factoriser. (Expliquer pourquoi.)

Deux autres cas sont possibles : Si x ne passe pas le test de Fermat, alors n est composé, mais le témoin x ne nous indique malheureusement aucun facteur de n. (C'est le cas le plus fréquent.) Si x passe le test de Miller-Rabin (y compris Fermat) on soupçonne que x est premier. On peut itérer le test pour être plus sûr.

2.8. Implémentation du test. La « longue marche » précédente avait pour but de motiver et d'élucider le critère de Miller-Rabin. Nous pouvons enfin passer à son implémentation.

Exercice/P 2.28. Écrire une fonction `bool estPseudoPremier( Integer n, Integer x )` qui teste si n est pseudo-premier à base x au sens de Miller-Rabin. Adapter le mode de passage de n et x aux besoins de votre fonction.

Exercice/P 2.29. Implémenter la version étendue du test de Miller-Rabin en une fonction
`bool estPseudoPremier( Integer n, Integer x, Integer& facteur )`
qui garde un éventuel facteur de n trouvé lors des calculs. Adapter le mode de passage de n et x aux besoins de votre fonction. Comme premier test et application, factoriser les nombres de Carmichael donnés dans la remarque 2.6. Ces nombres se révèlent plutôt sympathiques, après tout...

Exercice/P 2.30. Implémenter une fonction `bool estProbPremier( Integer n, int k=50 )` qui choisit successivement $x_1, x_2, \ldots, x_k \in [\![2, n-2]\!]$ de manière aléatoire et effectue le test de Miller-Rabin pour chacune de ces bases. *Indication.* — Quant à la primalité attraper d'abord le cas $n < 0$, puis $n = 0, 1, 2, 3$, puis les nombres pairs. Adapter le passage du paramètre `n`, le cas échéant, et expliquez l'utilisation du paramètre `k`, initialisé par défaut.

Remarque 2.31. On ne peut pas implémenter littéralement un algorithme probabiliste, car on ne dispose pas de source de nombres vraiment aléatoires. Dans la pratique tous les générateurs produisent de nombres « pseudo-aléatoires », c'est-à-dire une suite déterministe qui a l'aire d'être « suffisamment aléatoire ». Pour les sceptiques, l'usage des nombres aléatoires est un sujet délicat, voir Knuth [**8**], tome 3. Dans la pratique vous pouvez utiliser la fonction suivante, qui fait appel à un générateur de nombres pseudo-aléatoires, fourni par la bibliothèque `GMP`, auquel on fera confiance :

```
Integer random( const Integer& min, const Integer& max )
{ // construire un nombre aléatoire entre min et max inclus
  static gmp_randclass generateur(gmp_randinit_default);
  return generateur.get_z_range(max-min+1) + min; }
```

Exercice/P 2.32. Reprenez les deux entiers de l'exercice 1.3 et déterminez leur nature. Expliquer l'intérêt de la méthode de Miller-Rabin vis-à-vis la méthode naïve de l'exercice 1.3.

Exercice/P 2.33. Dans votre fonction qui réalise la factorisation limitée (exercice 1.14), ajoutez le test de primalité dans le cas d'un facteur restant $\hat{n} > 1$. Ceci permet de compléter la factorisation si $\hat{n}$ est (probablement) premier. Sinon, le facteur restant est laissé comme tel.

Exercice 2.34. Montrer que le nombre de Mersenne $M_{61} = 2^{61} - 1$ est premier, au moins très probablement. Factorisez quelques nombres M_k plus grands en indiquant lesquels laissent un facteur composé sans petits facteurs.
Remarque. — Pour les nombres de Mersenne $M_n = 2^n - 1$ il existe des tests plus spécifiques, analogues au test de Pépin pour les nombres de Fermat. Par exemple, si $2^n - 1$ est premier, alors n est premier. La réciproque est fausse : déjà $2^{11} - 1$ n'est pas premier. Si p est premier, alors tout facteur premier $q \mid M_p$ est de la forme $q = kp + 1$. Vous pouvez le vérifier empiriquement ou essayez de le montrer.

2.9. Comment trouver un nombre premier ? On se propose ici de produire de très grands nombres premiers, on dirait de manière industrielle. Voici une méthode possible :

Exercice/P 2.35. Écrire une fonction `Integer prochainPremier( Integer n )` qui trouve le plus petit premier $p > n$. Trouver ainsi le prochain nombre premier après 10^k pour $k = 2, \ldots, 200$.

Exercice/M 2.36. On pourrait également choisir un nombre premier de manière aléatoire dans l'intervalle $[\![a, b]\!]$. Évidemment il faut éviter que l'intervalle soit trop petit : s'il ne contient pas de nombre premier, inutile d'insister. Pour cela vous pouvez déduire une minoration de $\pi(b) - \pi(a-1)$ du théorème 1.17. En particulier le postulat de Bertrand affirme que l'intervalle $[\![a, 2a]\!]$ contient toujours de nombres premiers.

Exercice/M 2.37. Expliquer pourquoi l'algorithme XI.6 finira par trouver un nombre pseudo-premier comme promis. En s'inspirant de l'exercice 1.18, donner une estimation du nombre d'essais nécessaire.

Exercice/M 2.38. Supposons que pour chaque p on effectue des tests itérés de Miller-Rabin, de sorte que la probabilité d'erreur Prob(p pseudo-premier | p composé) soit ε. Supposons que l'algorithme XI.6 renvoie un pseudo-premier p. Quelle est la probabilité d'erreur de cet algorithme ? *Indication.* — La réponse n'est pas ε mais $\varepsilon \ln p$ environ ! Si vous

Algorithme XI.6 Engendrer un nombre pseudo-premier aléatoire entre a et b

Entrée: deux nombres naturels $a < b$ tels que $\pi(a-1) < \pi(b)$
Sortie: un nombre aléatoire $p \in [\![a,b]\!]$ qui soit pseudo-premier

$a' \leftarrow \lceil \frac{a-1}{2} \rceil$, $b' \leftarrow \lfloor \frac{b-1}{2} \rfloor$
boucle
  choisir $p' \in [\![a',b']\!]$ de manière aléatoire et poser $p \leftarrow 2p'+1$
  tester la pseudo-primalité de p par des tests itérés de Miller-Rabin
  si p est pseudo-premier **alors retourner** p
fin boucle

vous y connaissez en théorie des probabilités, on peut calculer Prob(p composé | p pseudo-premier) par les probabilités conditionnelles et la formule de Bayes. C'est une jolie application de la théorie élémentaire des probabilités. Comme toujours l'interprétation soigneuse des probabilités fait partie de l'honnêteté scientifique.

2.10. Comment prouver un nombre premier ? Rappelons que les deux observations clé de la méthode de Miller-Rabin sont les suivantes :

(1) Il est facile de déterminer si $x \in \mathbb{Z}_n^*$ est un témoin de décomposabilité.

(2) Si n est composé il est très probable de trouver un témoin $x \in \mathbb{Z}_n^*$.

Ceci correspond à deux types de problèmes bien distincts : *vérifier* une preuve et *trouver* une preuve. (Vous savez de votre propre expérience mathématique que trouver est en général plus difficile que vérifier.) Dans l'approche de Miller-Rabin le premier problème est résolu de manière déterministe, le deuxième de manière probabiliste.

Soulignons à nouveau que cette méthode permet de *prouver* qu'un nombre n est composé, mais en cas d'échec seulement de *soupçonner* que n est premier. L'asymétrie provient du fait qu'un seul témoin suffit pour montrer que n est composé, mais sans aucun témoin on ne peut pas conclure avec certitude.

Pour être sûr que n est premier nous voudrions également disposer d'une preuve. C'est tout à fait possible et l'idée est même très simple : il suffit de produire un élément d'ordre $n-1$ dans $\mathbb{Z}_n^\times$. Cette question a été résolue au §2.3 du chapitre IX. En voici le critère efficace :

Lemme 2.39. *On considère un entier n pour lequel on suppose connue la décomposition en facteurs premiers $n-1 = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$. Si un élément $g \in \mathbb{Z}_n^\times$ vérifie $g^{n-1} = 1$ alors son ordre divise $n-1$. Si de plus $g^{(n-1)/p_i} \neq 1$ pour tout $i = 1,2,\ldots,k$ alors l'ordre de g est exactement $n-1$. Dans ce cas $\mathbb{Z}_n^\times = \mathbb{Z}_n \smallsetminus \{0\}$, donc $\mathbb{Z}_n$ est un corps et n est un nombre premier.* □

La connaissance de la décomposition $n-1 = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ est une hypothèse délicate, car la factorisation peut être difficile en général. Heureusement elle est faisable dans beaucoup d'exemples.

Exercice/P 2.40. Comme avant on stocke la décomposition $n-1 = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ sous forme d'un vecteur $(p_1,e_1;p_2,e_2;\ldots;p_k,e_k)$. Écrire une fonction `bool estRacinePrimitive(g,n,decomp)` qui teste si g est d'ordre $n-1$ dans $\mathbb{Z}_n^\times$. Écrire une fonction `Integer racinePrimitive(n,decomp)` qui trouve la plus petite racine primitive modulo n, supposé premier, par des essais successifs. Choisir les modes de passage adéquats. Expliquer pourquoi on passe la décomposition de $n-1$ comme un paramètre. Comme alternative, serait-il une bonne idée de factoriser $n-1$ dans la fonction `estRacinePrimitive` ?

Exercice/P 2.41. Montrer que $n = 27!+1$ est premier en exhibant une racine primitive. Même exercice avec $37!+1$ et $73!+1$ puis $77!+1$. Plus généralement vous pouvez analyser $n!+1$ à condition d'effectuer au préalable un test de primalité.

Définition 2.42. En guise de résumé, précisons ce qu'il faut pour prouver la primalité d'un entier n. On appelle $C = (n,g;p_1,e_1;\ldots;p_k,e_k) \in \mathbb{N}^{2k+2}$ un *certificat de primalité* si

– on a $n-1 = p_1^{e_1} \cdots p_k^{e_k}$ avec $p_1 < \cdots < p_k$ premiers et $e_1,\ldots,e_k \geq 1$,

– le nombre g vérifie $g^{n-1} \equiv 1$ ainsi que $g^{(n-1)/p_i} \not\equiv 1$ modulo n pour $i = 1,\ldots,k$.

Dans ce cas on dit aussi que C *certifie* la primalité de n.

Exemple 2.43. Le plus petit certificat est $(2,1)$: la factorisation de $2-1=1$ est le produit vide, et $g=1$ est un élément d'ordre 1, comme exigé. Voici quelques certificats plus intéressants :

– $(3,2;2,1)$ certifie la primalité de 3. (Le vérifier. Y a-t-il d'autres certificats ?)
– $(17,3;2,4)$ certifie la primalité de 17. (Le vérifier. Y a-t-il d'autres certificats ?)
– $(103,5;2,1;3,1;17,1)$ certifie la primalité de 103 (le vérifier).
– $(n,5;2,21;3,14;17,9;103,3)$ certifie la primalité de
$n = 1\,299\,808\,706\,099\,639\,584\,492\,326\,223\,873$ (le vérifier).

Le dernier exemple soulève la question pratique : comment vérifier efficacement un certificat donné $(n,x;p_1,e_1;\ldots;p_k,e_k)$? D'abord il est facile à vérifier que $p_1^{e_1}p_2^{e_2}\cdots p_k^{e_k} = n-1$. Avec la puissance dichotomique on peut ensuite vérifier que $g^{n-1} \equiv 1$ et $g^{(n-1)/p_i} \not\equiv 1$ modulo n. Pour terminer la preuve de la primalité de n il ne reste qu'à prouver la primalité de $p_1,\ldots,p_k$. C'est simple : on exige des certificats !

Définition 2.44. Une *preuve de primalité* pour $n \in \mathbb{N}$ est une liste de certificats $C_1, C_2, \ldots, C_l$ pour des nombres premiers $n_1 < n_2 < \cdots < n_l = n$ de sorte que tout nombre premier utilisé dans un certificat C_j soit lui-même certifié par un certificat antérieur C_i ($i < j$).

Exemple 2.45. Les certificats de l'exemple précédent fournissent une preuve de primalité pour l'entier $n = 1\,299\,808\,706\,099\,639\,584\,492\,326\,223\,873$. Vérifier puis contempler ce bel exploit.

Exercice 2.46. Expliquer comment implémenter une fonction qui *vérifie* une preuve de primalité, puis une fonction qui *construit* une preuve de primalité. Expliquer pourquoi la vérification est facile, alors que la construction d'une preuve peut être difficile. À noter en particulier que la construction d'une preuve de primalité de n nécessite la factorisation de $n-1$, tâche qui peut être très coûteuse. Dans la pratique on ne lancera une telle recherche qu'après s'être convaincu par un test de Miller-Rabin. Si vous ne craignez pas de longs projets de programmation, vous pouvez implémenter toutes ces fonctions en C++.

3. Factorisation par la méthode ρ de Pollard

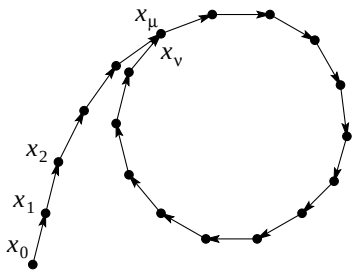
Dans ce qui précède nous avons discuté deux méthodes de grande importance pratique : la factorisation limitée (§1.5) et le critère de primalité selon Miller-Rabin (§2.5). Leur combinaison permet de factoriser tout nombre entier qui soit composé de petits facteurs premiers et au plus un grand facteur premier. (Rappeler pourquoi.) Reste à traiter le cas où n n'a pas de petits facteurs mais le test de Miller-Rabin prouve que n est composé. Avouons qu'en général cette tâche peut être extrêmement difficile !

La méthode ρ de Pollard décrite dans la suite est un algorithme probabiliste pour trouver des facteurs de taille moyenne, disons entre 10^6 et 10^{12}. Il existe des méthodes plus performantes, mais la méthode ρ a le mérite d'être très facile à implémenter. Si vous êtes impatients, vous pouvez commencer par l'implémentation et tester sa performance pratique (§3). Pour comprendre son succès, par contre, il nous faut quelques préparatifs (§3.1-3.3).

3.1. Détection de cycles selon Floyd. L'idée de base est d'analyser des suites récurrentes. Soit E un ensemble fini et $f\colon E \to E$ une application quelconque. À partir d'une valeur initiale $x_0 \in E$ on définit la suite $(x_k)_{k\in\mathbb{N}}$ par $x_{k+1} = f(x_k)$ pour tout $k \in \mathbb{N}$.

Exercice/M 3.1. Montrer qu'il existe $\mu < \nu$ tels que $x_\mu = x_\nu$. On pose $\lambda = \nu - \mu$. En déduire que la suite (x_k) devient ultimement périodique, c'est-à-dire $x_k = x_{k+\lambda}$ pour tout $k \geq \mu$.

Si l'on choisit μ et ν minimaux on dit que μ est *l'indice d'entrée* dans la période et λ est la *longueur* de la période. Cette situation est représentée par le dessin à droite. Il explique pourquoi la méthode de Pollard est aussi nommée la méthode ρ.



Comment déterminer μ et ν algorithmiquement ? La manière évidente est de stocker toute la suite $x_0,\ldots,x_{\nu-1}$ puis de chercher x_ν dans cette liste. Si x_ν n'y appartient pas encore, on le rajoute et continue avec $\nu \leftarrow \nu+1$. L'inconvénient de cette approche est, évidemment, qu'il faut stocker puis parcourir une liste éventuellement très longue. Afin de faciliter la recherche d'indices $k < k'$ avec $x_k = x_{k'}$, on se contentera d'une solution non minimale :

Exercice/M 3.2. Il existe $\ell > 0$ tel que $x_\ell = x_{2\ell}$. Si l'on choisit ℓ minimal alors $\mu \leq \ell \leq \nu \leq 2\ell$. En particulier ℓ et ν sont de même ordre de grandeur, à un facteur 2 près.

Il est facile de déterminer la valeur minimale ℓ : en commençant par $x_0 = y_0$ on parcourt la suite $x_{k+1} = f(x_k)$ et en même temps la la suite « à double vitesse » $y_{k+1} = f(f(y_k))$, et on teste successivement pour $\ell = 1,2,3,\dots$ si $x_\ell = y_\ell$. Ainsi on n'a plus besoin de stocker toute la liste, les deux valeurs courantes x_ℓ et y_ℓ suffisent !

3.2. Le paradoxe des anniversaires. En général il est difficile de prédire la longueur ℓ en fonction de f et de la valeur initiale x_0, mais elle est assez facile à mesurer sur des exemples donnés. Quand la suite passera-t-elle « en boucle » ? Il est clair que plus E est grand, plus ν et ℓ ont tendance à être grands. Essayons de donner un sens quantitatif à cette heuristique.

Exercice/M 3.3. Combien y a-t-il de fonctions $f : E \to E$? Quelle est la proportion des fonctions f avec $\nu(f) \geq 1$? avec $\nu(f) \geq 2$? avec $\nu(f) \geq 3$? En général, déterminer la proportion $p_{n,k} = \frac{\#\{f \mid \nu(f) \geq k\}}{\#\{f : E \to E\}}$.

Exercice/P 3.4 (le paradoxe des anniversaires)**.** En supposant que les anniversaires sont équidistribués sur les 365 jours de l'année, combien de personnes faut-il pour avoir deux de même jour d'anniversaire avec une probabilité $\geq \frac{1}{2}$? Écrire un programme qui calcule ce nombre, étonnamment petit. (Sans effort supplémentaire vous pouvez généraliser les paramètres $n = 365$ et $q = \frac{1}{2}$ dans ce calcul.)

Voici la version asymptotique de ce phénomène :

Proposition 3.5. *Soit E un ensemble fini de cardinal n. Quand on choisit $f : E \to E$ au hasard, la valeur moyenne de $\nu(f)$ est donnée par $\overline{\nu}_n = \sum_{k=1}^{n} p_{n,k}$. Pour $n \to \infty$ on a $\overline{\nu}_n \sim \sqrt{\frac{\pi}{2} n}$.* □

Autrement dit, pour un choix aléatoire de f on s'attend à ce que la suite récurrente $x_{k+1} = f(x_k)$ boucle à un indice ν d'ordre $\sqrt{n}$ environ. Il en est de même pour l'indice ℓ, de même ordre que ν mais plus facile à déterminer dans la pratique. Pour une preuve simple que $\overline{\nu}_n \in O(\sqrt{n})$ voir Gathen&Gerhard [**11**], théorème 19.5. Pour un développement détaillé, voir Knuth [**8**], vol. 2, §3.1, exercices 11 et 12 avec leur solution à la fin du tome, qui renvoient au vol. 1, §1.2.11.3 pour le calcul asymptotique.

3.3. Polynômes et fonctions polynomiales. Pour l'ensemble E on prendra dans la suite l'anneau $\mathbb{Z}_n$ et pour applications $f : \mathbb{Z}_n \to \mathbb{Z}_n$ nous regardons des fonctions polynomiales, comme $f(x) = x^2 + 1$.

Afin d'éviter toute confusion, il sera utile de distinguer *polynômes* $P \in \mathbb{Z}_n[X]$ et *fonctions polynomiales* $f : \mathbb{Z}_n \to \mathbb{Z}_n$. Rappelons que tout élément de $\mathbb{Z}_n[X]$ s'écrit comme $\sum_k a_k X^k$ avec coefficients $a_k \in \mathbb{Z}_n$ et qu'on a l'égalité $\sum_k a_k X^k = \sum_k b_k X^k$ si et seulement si $a_k = b_k$ pour tout k. Tout polynôme $P = \sum_k a_k X^k$ définit une fonction $\Phi_P : \mathbb{Z}_n \to \mathbb{Z}_n$ par $\Phi_P(x) = P(x) = \sum_k a_k x^k$, appelée la *fonction polynomiale* associée à P. Évidemment $P = Q$ entraîne $\Phi_P = \Phi_Q$, mais la réciproque est fausse : on peut avoir $\sum_k a_k x^k = \sum_k b_k x^k$ pour tout $x \in \mathbb{Z}_n$ sans que $a_k = b_k$ pour tout k.

Proposition 3.6. *Si p est premier, alors toute fonction $f : \mathbb{Z}_p \to \mathbb{Z}_p$ est polynomiale : il existe un polynôme $P \in \mathbb{Z}_p[X]$ et un seul de degré $< p$ tel que $f = \Phi_P$, à savoir l'interpolant de Lagrange $P = \sum_{a \in \mathbb{Z}_p} f(a) \prod_{b \in \mathbb{Z}_p \smallsetminus \{a\}} \frac{X-b}{a-b}$. L'application $\Phi : \mathbb{Z}_p[X] \to \mathrm{Fonc}(\mathbb{Z}_p, \mathbb{Z}_n)$, $P \mapsto \Phi_P$ est donc surjective. Il s'agit d'un homomorphisme d'anneaux qui a pour noyau l'idéal $(X^p - X)$.*

Exercice/M 3.7. Prouver cette proposition.

Supposons désormais que $n = p_1 p_2 \cdots p_m$ avec des nombres premiers $p_1, p_2, \dots, p_m$. Pour simplifier nous supposons que $p_1 < p_2 < \cdots < p_m$. (Nous négligeons ici le cas de facteurs multiples.) D'après le théorème des restes chinois, l'anneau $\mathbb{Z}_n$ se décompose en $\mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_m}$. Étant donné des fonctions $f_i : \mathbb{Z}_{p_i} \to \mathbb{Z}_{p_i}$ on peut les assembler en une fonction $f : \mathbb{Z}_n \to \mathbb{Z}_n$ d'après le schéma suivant :

$$\begin{array}{ccc} \mathbb{Z}_n & \xrightarrow{\quad f \quad} & \mathbb{Z}_n \\ \cong\downarrow & & \downarrow\cong \\ \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_m} & \xrightarrow{f_1 \times \cdots \times f_m} & \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_m} \end{array}$$

Proposition 3.8. *On dit qu'une fonction $f : \mathbb{Z}_n \to \mathbb{Z}_n$ est* décomposable *(en $f_1, \dots, f_m$) si elle peut être construite comme dans le diagramme précédent. Toute fonction polynomiale $f = \Phi_P$ est décomposable. Réciproquement toute fonction décomposable $f : \mathbb{Z}_n \to \mathbb{Z}_n$ est polynomiale.*

Exercice/M 3.9. Prouver cette proposition. Étant donné une fonction $f = \Phi_P$ comment la décomposer ? Réciproquement, étant donné f décomposable, comment construire un polynôme $P \in \mathbb{Z}_n[X]$ tel que $f = \Phi_P$? Expliquer pourquoi les fonctions décomposables ne forment en général qu'une petite fraction des applications $f : \mathbb{Z}_n \to \mathbb{Z}_n$.

3.4. L'heuristique de Pollard. Après ces préparations, nous allons les appliquer à la factorisation. Les arguments précédents se résument dans l'interprétation probabiliste suivante :

Corollaire 3.10. *Pour p premier fixons un degré $d \geq p$ et choisissons un polynôme $P \in \mathbb{Z}_p[X]$ de degré $< d$ de manière équiprobable, chacun avec probabilité p^{-d}. Alors la fonction polynomiale associée Φ_P tombe sur n'importe quelle fonction $\mathbb{Z}_p \to \mathbb{Z}_p$ avec la même probabilité p^{-p}. On s'attend donc au paradoxe des anniversaires : une suite récurrente $x_{k+1} = P(x_k)$ va boucler après $\sqrt{p}$ itérations environ.* □

Corollaire 3.11. *Supposons que $n = p_1 p_2 \cdots p_m$ est composé de premiers $p_1 < p_2 < \cdots < p_m$ et que l'on choisit aléatoirement un polynôme $P \in \mathbb{Z}_n[X]$ de degré $< n$. Ensuite on décompose la fonction polynomiale associée $f = \Phi_P$ en $f_i : \mathbb{Z}_{p_i} \to \mathbb{Z}_{p_i}$. Alors f_i est équidistribuée sur les $p_i^{p_i}$ fonctions possibles. On s'attend donc à nouveau au paradoxe des anniversaires : une suite récurrente $x_{k+1} = P(x_k)$ va boucler modulo p_1 après $\sqrt{p_1}$ itérations environ, modulo p_2 après $\sqrt{p_2}$ itérations environ, … et finalement modulo p_m après $\sqrt{p_m}$ itérations environ.* □

Exemple 3.12. Voici un exemple qui est trop petit pour être réaliste, mais suffisamment grand pour illustrer le principe. Regardons l'itération de $f : \mathbb{Z}_{221} \to \mathbb{Z}_{221}$ avec $f(x) = x^2 + 1$ et $x_0 = 1$:

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
x_k	1	2	5	26	14	197	135	104	209	145	31	78	118	2	…

Ici $\mu = 1$ et $\lambda = 12$. On constate que $\ell = 12$ est le plus petit indice positif tel que $x_\ell = x_{2\ell}$. Comme $221 = 13 \cdot 17$, regardons les suites dans les quotients. Modulo 17 la suite correspond à l'itération de $\mathbb{Z}_{17} \to \mathbb{Z}_{17}$, $x \mapsto x^2 + 1$ avec valeur initiale $x_0 = 1$:

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
x_k	1	2	5	9	14	10	16	2	5	9	14	10	16	2	…

Ici $\mu = 1$ et $\lambda = 6$ ainsi que $\ell = 6$. Modulo 13 nous obtenons :

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
x_k	1	2	5	0	1	2	5	0	1	2	5	0	1	2	…

Ici $\mu = 0$ et $\lambda = 4$ ainsi que $\ell = 4$. (Est-ce un hasard que $12 = \mathrm{ppcm}(4,6)$?)

Remarque 3.13. Avec beaucoup de bienveillance, on peut considérer l'exemple comme une illustration de la proposition 3.5. Bien sûr l'application $x \mapsto x^2 + 1$ n'a rien d'aléatoire, et les coïncidences $12 \approx \sqrt{221}$ et $6 \approx \sqrt{17}$ et $4 \approx \sqrt{13}$ peuvent sembler peu convaincantes. Soulignons que l'argument statistique ne décrit que le comportement *en moyenne*, et c'est ce comportement qui est souvent observé dans les exemples. (Des exceptions seront inévitables.) Pour vous en convaincre, vous pouvez tester d'autres fonctions polynomiales et d'autres entiers composés n avec votre implémentation plus bas.

3.5. L'algorithme de Pollard. Après l'heuristique, formulons l'algorithme probabiliste qui en découle :

Algorithme XI.7 Factorisation selon la méthode ρ de Pollard

Entrée: un entier $n > 1$, une valeur initiale $x \in \mathbb{Z}_n$ et un polynôme $P \in \mathbb{Z}_n[X]$
Sortie: un diviseur $d > 1$ de n (possiblement $d = n$)

```
y ← x                          // x et y représentent x_k et x_{2k} de la suite récurrente
répéter
   x ← P(x) mod n              // On passe de x_k à x_{k+1}
   y ← P(P(y)) mod n           // On passe de x_{2k} à x_{2k+2}
   d ← pgcd(x − y, n)          // On espère que x ≡ y modulo p_1 mais non modulo n
jusqu'à d > 1
retourner d
```

Exercice/M 3.14. Montrer que l'algorithme XI.7 s'arrête puis qu'il est correct. Expliquer son comportement en moyenne : quel résultat obtient-on et avec combien d'itérations ? En quoi cette méthode est-elle intéressante vis-à-vis la factorisation par essais successifs ?

3.6. Implémentation et tests empiriques. Jusqu'ici notre développement repose sur un solide argument statistique qui suppose que nous choisissons la fonction $f = \Phi_P$ aléatoirement. Or, il est très coûteux, voire impossible, de stocker puis d'évaluer des polynômes de très grand degré (d'ordre 10^9 disons). On choisira donc des polynômes de petit degré, typiquement quadratique, en espérant que cet échantillon sera suffisamment représentatif pour reproduire le paradoxe des anniversaires et son asymptotique décrit dans la proposition 3.5. (D'ailleurs, les fonctions linéaires ne conviennent pas. Pourquoi ?)

Exercice/P 3.15. Implémenter l'algorithme de Pollard pour $f(x) = x^2 + a$ en une fonction

```
Integer pollard( const Integer& n, Integer x, int a, int max=500000 )
```

Justifier l'usage d'un paramètre `max` pour majorer le nombre d'itérations dans le pire cas.

- Tester votre fonction sur $n = 221$, $x = 1$, $a = 1$ pour vérifier qu'elle produit bien les résultats de l'exemple 3.12 plus haut : on trouve le facteur $\text{pgcd}(x_4 - x_8, 221) = 13$.
- Tester votre fonction sur l'exemple $n = 143$ et $x = 1$. Vérifier que $a = 1$ donne le diviseur trivial $d = 143$. Que donnent les paramètres $a = 2$ et $a = 3$?

Ces expériences indiquent que l'on ne peut pas prédire le bon choix du polynôme f. Si un premier essai échoue, alors on recommence avec d'autres paramètres.

Exercice/P 3.16. Pour résumer le développement de ce chapitre, écrire un programme qui permet d'entrer un nombre entier puis de le factoriser (au moins partiellement) :

(1) Mettez au point une factorisation limitée pour trouver les petits facteurs (§1.5).

(2) Déterminer la nature d'un éventuel facteur restant par le test de Miller-Rabin (§2.8).

(3) Le cas échéant, appliquer la méthode ρ de Pollard. En cas d'échec il faut éventuellement réessayer. En cas de succès assurer que votre factorisation est complète, et réitérer si nécessaire.

Essayez ainsi de décomposer $2\cdot 19!+1$, $4\cdot 19!-1$, $6\cdot 19!+1$, $20!+1$, $24!-1$ en facteurs premiers. Indiquez les méthodes utilisées et les facteurs ainsi trouvés. Si la méthode en question dépend de paramètres à choisir, comme la factorisation limitée ou la méthode de Pollard, explicitez-les.

Exercice 3.17. Décomposer les deux exemples de l'exercice 1.3, donnés au début du chapitre.

Exercice 3.18. Pour $k = 20, \ldots, 40$ essayer de décomposer $k! + 1$ en facteurs premiers.

3.7. Conclusion. Le principal argument en faveur de l'approche heuristique de Pollard est, bien sûr, le succès pratique amplement illustré. Notre heuristique explique aussi pourquoi la méthode de Pollard échouera probablement dans le cas où n est composé de grands facteurs premiers : le nombre d'itérations nécessaire devient trop grand. C'est le cas par exemple pour

$$38! + 1 = 14\,029\,308\,060\,317\,546\,154\,181 \cdot 37\,280\,713\,718\,589\,679\,646\,221.$$

Pour savoir d'avantage sur des méthodes plus performantes, qui sont capables d'exhiber, par exemple, la factorisation précédente, on consultera avec profit l'excellent livre de R. Crandall et C. Pomerance [**15**]. Il existe plusieurs sites web consacrés à la chasse aux factorisations. La page de R. Brent est un bon endroit pour commencer, `web.comlab.ox.ac.uk/oucl/work/richard.brent/factors.html`

4. Le critère déterministe selon Agrawal-Kayal-Saxena

Le problème de primalité. — On veut analyser un entier n, donné sous forme d'un développement binaire de longueur $\ell = \text{len}(n)$. Il s'agit de déterminer si n est premier ou composé. On a déjà vu que le test exhaustif est d'une complexité exponentielle (cf. la proposition 1.8). Il est prohibitif pour des grands entiers, disons à partir d'une longueur $\ell \approx 60$, soit 20 décimales environ.

Approche probabiliste. — Le test de Miller-Rabin est d'une complexité polynomiale $O(\ell^3)$ voire $O^+(\ell^2)$, comme esquissé en exercice 2.19. De nombreux exemples témoignent de son efficacité, et d'un point de vue pratique on pourrait considérer le problème comme résolu. Cependant, sa nature probabiliste

laisse une certaine probabilité d'erreur (arbitrairement petite, mais non nulle). Il en est de même d'ailleurs pour le test de Solovay-Strassen, que nous avons discuté dans le projet X.

Approche déterministe. — Existe-t-il une méthode qui résolve le problème de primalité de manière *universelle*, c'est-à-dire pour n'importe quel $n \in \mathbb{N}$, et qui soit à la fois *déterministe* et de complexité *polynomiale* ? Cette question est restée longtemps ouverte et hantait les théoriciens. Assez récemment, en juillet 2002, la réponse spectaculaire fut établie par M. Agrawal, N. Kayal et N. Saxena :

Théorème 4.1. *Il existe un algorithme qui, pour tout entier n donné, détermine si n est premier ou composé, et dont le temps d'exécution est polynomial en la longueur* $\mathrm{len}(n)$ *du nombre n.*

Comme l'algorithme AKS est assez élémentaire, nous nous proposons ici de l'implémenter. Si ce sujet vous donne envie d'expérimenter, les considérations pratiques qui suivent vous permettront d'écrire votre propre programme AKS et d'entreprendre une exploration empirique. Quant aux mathématiques sous-jacentes, pour la plupart également élémentaires, nous nous contentons de présenter les idées essentielles. L'algorithme n'a sans doute pas encore trouvé sa forme optimale, et de nombreuses améliorations sont actuellement développées. Vous trouverez de plus amples explications dans la littérature :

(1) L'article de Agrawal-Kayal-Saxena est paru dans *Annals of Mathematics*, 160 (2004) 781–793, `www.math.princeton.edu/~annals/issues/2004/Sept2004/Agrawal.pdf`

(2) Un développement élémentaire a été rédigé par Michiel Smid, détaillant toutes les preuves que nous admettons, `www.scs.carleton.ca/~michiel/primes.ps.gz`.

4.1. Une belle caractérisation des nombres premiers. Nous reprenons à nouveau le petit théorème de Fermat : si n est premier, alors $a^n = a$ modulo (n) pour tout $a \in \mathbb{Z}$. Cette observation se généralise à un anneau quelconque, pourvu que l'on adapte convenablement l'énoncé :

Lemme 4.2. *Soient x, y deux éléments d'un anneau commutatif A. Alors pour tout $n \in \mathbb{N}$ on a la formule binomiale $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$. Si de plus n est premier, alors n divise le coefficient binomial $\binom{n}{k}$ pour tout $k = 1, \dots, n-1$, et ainsi $(x+y)^n \equiv x^n + y^n$ modulo (n).* □

Exercice/M 4.3. Montrer ce lemme et en déduire le petit théorème de Fermat $a^p \equiv a \pmod{p}$ par récurrence sur a.

On peut appliquer ce lemme comme un critère de primalité. Le test AKS est basé sur la jolie observation qu'en prenant l'anneau $\mathbb{Z}[X]$ le critère nécessaire est aussi suffisant :

Proposition 4.4. *Soit $n \geq 2$ et $a \in \mathbb{Z}$. Si n est premier alors $(X+a)^n \equiv X^n + a$ dans $\mathbb{Z}_n[X]$. Réciproquement, si* $\mathrm{pgcd}(a, n) = 1$ *et $(X+a)^n \equiv X^n + a$ dans $\mathbb{Z}_n[X]$, alors n est premier.* □

Exercice/M 4.5. Essayez de montrer cette proposition. *Indication.* — La première partie est claire. Quant à la deuxième, supposons que $n = p^e q$ avec p premier et $p \nmid q$. Déduire de $(X+a)^n \equiv X^n + a$ modulo (p) que $q = 1$. Déduire de $(X+a)^{p^e} \equiv X^{p^e} + a$ modulo (p^e) que $e = 1$. (Pour une solution voir Smid.)

C'est une très belle caractérisation de la primalité de n. Quant au calcul pratique, ce critère reste malheureusement impraticable pour n grand, à cause de l'immense longueur du polynôme $(X+a)^n$. Déjà pour $n \approx 10^{20}$, ce polynôme dépasse la capacité de tout ordinateur.

4.2. Polynômes cycliques. Afin de rendre le critère précédent calculable, on réduit modulo $(X^r - 1)$, ce qui veut dire que nous posons $X^r = 1$ dans tous nos calculs. Grâce à cette relation on peut immédiatement réduire tout monôme X^s au monôme $X^{s \bmod r}$. On calcule ainsi avec des exposants modulo r, d'où le nom polynômes « cycliques ». L'avantage de cette approche est que les calculs deviennent faisables, même assez efficaces (ce qui sera à discuter après implémentation). L'inconvénient est que la caractérisation précédente se transforme en un critère nécessaire qui n'est plus forcément suffisant :

Corollaire 4.6. *Si n est premier alors $(X+a)^n = X^n + a$ modulo $(n, X^r - 1)$. Par contraposé : si $(X+a)^n \not\equiv X^n + a$ modulo $(n, X^r - 1)$ alors n est composé.*

Exemple 4.7. Comme remarqué plus haut, $n = 1729$ est un nombre de Carmichael, c'est-à-dire tout $a \in \mathbb{Z}_n$ vérifie $a^n = a$. Ici le test AKS se révèle plus fin : on a $(X+2)^n \equiv 819X^2 + X + 912$ modulo $(n, X^4 - 1)$, ce qui ne coïncide pas avec $X^n + 2 \equiv X + 2$. À noter aussi que pour $X \mapsto 1$ on obtient exactement le test

de Fermat ; ici $(1+2)^n \equiv 819+1+912 \equiv 3$, comme il faut. (Cet exemple sera facile à calculer avec l'implémentation développée dans la suite.)

Convention. — On supposera dans la suite que $\text{pgcd}(n,r)=1$. Ceci est tout à fait légitime : on envisage de tester la primalité de n, et un $\text{pgcd}(n,r)>1$ donnerait immédiatement un facteur de n.

4.3. Une implémentation basique. Commençons par fixer les types des données. Pour les calculs dans $\mathbb{Z}$ nous auront besoin, comme d'habitude, d'un type `Integer` modélisant les grands entiers. Pour les exposants des monômes, les indices et les longueurs des vecteurs, par contre, les petits entiers suffiront. (Pourquoi ?) De toute façon, notre compilateur exige que les vecteurs soient indexés par un type primitif, comme `int` ou `unsigned int` . Pour fixer notre choix, nous appelons ce type `SmInt` pour *small integer* :

```
#include <vector>                 // définir la classe générique vector
#include <gmpxx.h>                // déclarer les grands entiers de GMP
typedef mpz_class       Integer;  // grands entiers (coefficients)
typedef unsigned int    SmInt;    // petits entiers (exposants)
typedef vector<Integer> Poly;     // polynôme stocké comme vecteur
```

Convention. — Tout polynôme P modulo (n,X^r-1) peut être représenté, de manière unique, par $a_0+a_1X+\cdots+a_{r-1}X^{r-1}$ avec coefficients $a_i \in [\![0,n[\![$. Sous cette forme-là il est stocké comme un vecteur $(a_0,a_1,\ldots,a_{r-1})$ de taille r exactement.

Exercice/P 4.8. Écrire une fonction qui effectue la multiplication « cyclique » :

`Poly mult_cyclique( const Poly& p, const Poly& q, const Integer& n )`

Indication. — La multiplication peut se réaliser, comme d'habitude, par deux boucles imbriquées. Veillez toutefois à ne calculer qu'avec des polynômes modulo (X^r-1) et de ne jamais dépasser la longueur r. Pensez aussi à réduire les coefficients modulo n à la fin.

Exercice/P 4.9. Comme toujours, la puissance dichotomique s'impose pour effectuer le calcul de $(X+a)^n$ modulo (n,X^r-1). Écrire une telle fonction

`Poly puissance_cyclique( Poly base, Integer exp, const Integer& n )`

qui emploie la multiplication cyclique implémentée dans l'exercice précédent.

Exercice/M 4.10. Soulignons à quel point les choix faits pour cette implémentation sont judicieux. Que se passerait-il si l'on ne réduisait pas systématiquement modulo n ? Que se passerait-il si l'on ne réduisait pas systématiquement modulo X^r-1 ? Expliquer heuristiquement pourquoi les réductions modulo n dans chaque coefficient *et* modulo X^r-1 pour tout monôme garantissent des données de taille bornée. (On analysera la complexité du calcul plus bas.)

Exercice/P 4.11. Écrire finalement une fonction qui teste si $P(X)^n = P(X^n)$ modulo (n,X^r-1) :

`bool testFermat( const Poly& p, const Integer& n )`

Indication. — Pour calculer $Q(X) := P(X)^n$ modulo (n,X^r-1) on se servira de la puissance dichotomique ci-dessus. Pour $P(X^n)$ il y a une astuce : comme r et n sont premiers entre eux, $P(X^n)$ mod (X^r-1) n'est qu'une permutation des coefficients (l'expliciter). On compare $Q(X)$ et $P(X^n)$ via cette permutation d'indices. Déterminer r et $t = n \bmod r$ peut se réaliser ainsi :

```
SmInt r= p.size();               // la taille du polynôme en question
SmInt t= Integer(n%r).get_ui();  // transformer Integer en unsigned int
```

Exercice/P 4.12. Implémenter une fonction qui teste plus spécifiquement si $(X-a)^n \equiv X^n-a$ modulo (n,X^r-1). C'est le test d'AKS proprement dit, avec paramètres $r \in \mathbb{N}$ et $a \in \mathbb{Z}$:

`bool temoinAKS( const Integer& n, SmInt r, Integer a )`

Tester vos fonctions sur beaucoup d'exemples. Si n est premier, votre fonction répond-elle correctement ? Si n est composé, trouve-t-on rapidement un témoin ? Que se passe-t-il pour les nombres de Carmichael ? Donner plus d'exemples illustrant que le test AKS est plus fin que le test de Fermat. Statistiquement, est-ce un critère intéressant dans la pratique ?

Remarque 4.13. Pour r grand, le facteur limitant est la multiplication cyclique des polynômes modulo (n,X^r-1). Notre implémentation nécessite r^2 opérations dans $\mathbb{Z}_n$. Afin de profiter de la multiplication rapide d'entiers, on peut transformer les polynômes $P(X)$ et $Q(X)$ en deux entiers $\hat{P}=P(B)$ et $\hat{Q}=Q(B)$ avec $B=2^k$ suffisamment grand. On

calcule $\hat{R} = \hat{P}\hat{Q}$ avec une multiplication rapide, et en retrouve le polynôme $R(X)$ tel que $R(B) = \hat{R}$. Ce procédé permet de calculer le produit $R = PQ$ avec une complexité presque linéaire en r.

4.4. Analyse de complexité. Essayons de déterminer la complexité du test AKS, c'est-à-dire du test de Fermat étendu à l'anneau $\mathbb{Z}_n[X]/(X^r-1)$, implémenté comme ci-dessus :

Proposition 4.14. *Soit n un entier de longueur $\ell = \mathrm{len}(n)$ en base 2. Le coût pour effectuer un test de Fermat dans $\mathbb{Z}_n[X]/(X^r-1)$ avec l'implémentation ci-dessus est d'ordre $r^2\ell^3$. Avec une multiplication optimisée on peut descendre jusqu'à une complexité presque d'ordre $r\ell^2$.*

ESQUISSE DE PREUVE. La multiplication de deux éléments dans $\mathbb{Z}_n$ est de complexité $O(\ell^2)$ avec la méthode scolaire. Optimal serait $O^+(\ell)$ avec des méthodes plus sophistiquées.

La multiplication de deux polynômes cycliques dans $\mathbb{Z}_n[X]/(X^r-1)$ nécessite r^2 multiplications dans $\mathbb{Z}_n$ avec la méthode scolaire, utilisée ci-dessus. Optimal serait $O^+(r)$. On arrive donc a une complexité de $O(r^2)O(\ell^2)$, voire $O^+(r)O^+(\ell)$ avec une implémentation optimisée.

La puissance dichotomique nécessite au plus 2ℓ multiplications pour calculer $P(X)^n$ modulo (n, X^r-1). Au total on arrive donc à une complexité de $O(r^2)O(\ell^3)$, voire $O^+(r)O^+(\ell^2)$. □

4.5. La découverte d'Agrawal-Kayal-Saxena. Avec notre implémentation on obtient un coût d'ordre $sr^2\ell^3$ pour effectuer s tests d'AKS dans $\mathbb{Z}_n[X]/(X^r-1)$. À première vue cela semble raisonnable ; le problème est de majorer r et s. La découverte surprenante d'Agrawal-Kayal-Saxena est le résultat suivant :

Théorème 4.15 (Agrawal-Kayal-Saxena, 2002)**.** *Il existe une constante $C > 0$ telle que pour tout nombre naturel n on puisse trouver un nombre premier $r \leq C\,\mathrm{len}(n)^6$ tel que le critère suivant soit vrai :*

Si $\mathrm{pgcd}(n,a) = 1$ et $(X+a)^n = X^n + a$ modulo (n, X^r-1) pour tout $a = 1, \ldots, s$ avec $s = 2\lfloor\sqrt{r}\rfloor \mathrm{len}\, n$, alors n est premier ou une puissance d'un nombre premier. □

On admettra ce théorème. Il peut être démontré par des méthodes élémentaires (voir Smid). Ce résultat implique, comme promis, la conclusion énoncée dans l'introduction :

Corollaire 4.16. *Étant donné un nombre naturel n on peut déterminer s'il est premier ou composé par un algorithme de complexité $O(\mathrm{len}(n)^{19})$. En utilisant une multiplication optimisée on peut descendre jusqu'à une complexité $O^+(\mathrm{len}(n)^{12})$. En particulier, le problème de primalité admet une solution algorithmique de complexité polynomiale dans la longueur* $\mathrm{len}(n)$.

Exercice/M 4.17. Déduire le corollaire du théorème, en (ré)analysant l'implémentation ci-dessus : traduire d'abord la phrase « un nombre premier r d'ordre $O(\mathrm{len}(n)^6)$ » en une formulation précise, puis appliquer la proposition 4.14. Expliquer ensuite pourquoi il est peu coûteux de déterminer si n est une puissance parfaite : rappeler que l'on peut rapidement calculer $a = \lfloor\sqrt[k]{n}\rfloor$ puis comparer a^k avec n. Il suffit de ce faire pour $k = 2, 3, \ldots, \mathrm{len}(n)$. (Pourquoi ?) Quelle en est la complexité totale ?

4.6. Vers un test pratique ? Pour rendre le test AKS praticable, il faut expliciter *comment* choisir r et s. En voici une version simplifiée, qui se prête bien à l'implémentation :

Théorème 4.18 (Agrawal-Kayal-Saxena, 2002)**.** *Soit n un nombre naturel. Soit $q \geq 32\,\mathrm{len}(n)^2$ un premier tel que $r := 2q+1$ soit également premier et $n \not\equiv 0, \pm 1$ modulo r. Si $\mathrm{pgcd}(n,a) = 1$ et $(X+a)^n = X^n + a$ modulo (n, X^r-1) pour tout $a = 1, \ldots, s$ avec $s = 2\lfloor\sqrt{r}\rfloor \mathrm{len}(n)$, alors n est premier ou une puissance d'un nombre premier.* □

On en déduit l'algorithme XI.8 pour déterminer si un entier donné est premier ou composé.

Remarque 4.19. La correction puis l'efficacité de l'algorithme précédent repose sur l'existence de certains nombres premiers : on appelle q un *nombre premier de Sophie Germain* si q et $r = 2q+1$ sont tous les deux premiers. Ces nombres ne sont pas si rares que cela. Voici les exemples jusqu'à 1000 :

$$2, 3, 5, 11, 23, 29, 41, 53, 83, 89, 113, 131, 173, 179, 191, 233, 239, 251, 281, 293,$$
$$359, 419, 431, 443, 491, 509, 593, 641, 653, 659, 683, 719, 743, 761, 809, 911, 953$$

Le théorème des nombres premiers nous dit que la probabilité qu'un nombre q soit premier est environ $\frac{1}{\ln q}$. Si l'on est prêt à croire que la primalité de q et la primalité de $2q+1$ sont en quelque sorte

Algorithme XI.8 Une variante du test de primalité selon Agrawal-Kayal-Saxena

Entrée: un nombre naturel n

Sortie: le message « premier » ou « composé »

pour k **de** 2 **à** $\text{len}(n)$ **faire** $a \leftarrow \lfloor \sqrt[k]{n} \rfloor$: **si** $a^k = n$ **alors retourner** « composé »
initialiser $q \leftarrow 32\,\text{len}(n)^2$, $r \leftarrow 2q+1$
tant que q ou r est composé ou $n \equiv 0, \pm 1$ modulo r **faire** $q \leftarrow q+1$, $r \leftarrow r+2$
$s \leftarrow 2\lfloor \sqrt{r} \rfloor \text{len}(n)$
pour a **de** 1 **à** s **faire**
  si $\text{pgcd}(n,a) > 1$ **alors retourner** « composé »
  si $(X-a)^n \not\equiv X^n - a$ modulo (n, X^r-1) **alors retourner** « composé »
fin pour
retourner « premier »

indépendantes, alors il semble plausible que la probabilité de tomber sur un nombre premier de Sophie Germain soit environ $\frac{1}{\ln(q)^2}$.

Conjecture 4.20. *La quantité des nombres premiers de Sophie Germain suit l'asymptotique*

$$\#\{q \leq x \mid q \text{ et } 2q+1 \text{ sont premiers } \} \sim \text{const} \frac{x}{\ln(x)^2}.$$

Contrairement au théorème des nombres premiers, cette conjecture reste toujours ouverte. Elle correspond assez bien aux données empiriques : quand on cherche un nombre premier de Sophie Germain $q \geq 32\,\text{len}(n)^2$, il semble toujours en exister un de cet ordre de grandeur-là.

Proposition 4.21. *Supposons qu'il existe une constante C de sorte qu'il soit toujours possible de trouver un nombre premier de Sophie Germain $q \in [\![32\,\text{len}(n)^2, C\,\text{len}(n)^2]\!]$. Dans ce cas r et s sont de l'ordre de grandeur $O(\text{len}(n)^2)$, et l'algorithme ci-dessous est de complexité $O(\text{len}(n)^9)$, voire $O^+(\text{len}(n)^6)$ en utilisant des multiplications sophistiquées.*

Même sous cette hypothèse optimiste, l'efficacité du test AKS laisse encore à désirer :

Exemple 4.22. Si l'on prend $q = 809$ et $r = 1619$, on peut tester la primalité des nombres n jusqu'à 5 bits, car $32 \cdot 5^2 = 800$. Ceci toujours sous réserve, bien entendu, que $n \not\equiv 0, \pm 1$ modulo r ; si jamais cette condition pose problème, il faut passer à $q = 911$, $r = 1823$ ou $q = 953$, $r = 1907$ etc … En tout état de cause, pour les petits nombres n cette démarche n'est pas efficace.

Exemple 4.23. Prenons un exemple plus réaliste : disons que l'on veut tester des entiers à 100 bits, soit 30 décimales environ. Ici le test AKS est moins ridicule. Pour $\text{len}(n) = 100$ les nombres premiers $q = 320009$, $r = 640019$ conviennent, ainsi que $q = 320063$, $r = 640127$, puis $q = 320081$, $r = 640163$ etc. On voit, d'une part, qu'il existe beaucoup de nombres premiers de Sophie Germain, et d'autre part que les valeurs de r dans l'algorithme sont déjà assez élevées.

Exemple 4.24. Regardons finalement les entiers à 1000 bits, soit 300 décimales environ. Pour $\text{len}(n) = 1000$ les nombres premiers $q = 32000651$, $r = 64001303$ conviennent, ainsi que $q = 32000669$, $r = 64001339$, puis $q = 32000753$, $r = 64001507$, etc. Bien que théoriquement intéressant, le test AKS commence à occuper trop de mémoire et de consommer trop de temps pour être praticable dans cet ordre de grandeur.

Ces exemples indiquent que le beau résultat d'AKS ne se prête pas encore à une implémentation efficace. En particulier la version actuelle ne peut pas concurrencer avec le test probabiliste selon Miller-Rabin, qui traite des entiers à 1000 bits assez rapidement (le tester). Pour l'instant c'est donc l'aspect théorique qui fait le succès du résultat AKS. Toutefois, après cette innovation inattendue, on peut espérer que d'autres bonnes surprises nous attendent encore. À suivre …

Exercice/P 4.25. Expérimenter avec votre implémentation du test AKS pour étudier son comportement dans des exemples concrets. Par mesure de prudence, vérifier la correction des réponses fournies par votre programme, disons en recoupant avec un test de Miller-Rabin. Pour un nombre composé donné n peut-on trouver r et a petits de sorte que $(X+a)^n \not\equiv X^n + a$ modulo (n, X^r-1) ? Est-ce que la performance

empirique est meilleure que la prévision théorique ? Chercher dans la littérature une variante optimisée qui utilise des paramètres r et s plus petits, puis l'implémenter. Déterminer empiriquement le temps et la mémoire nécessaires ; jusqu'où ce test vous semble-t-il praticable ?

5. Résumé et perspectives

En guise de résumé, voici l'approche aux problèmes évoqués au début de ce chapitre.

(1) Tout d'abord on établit une fois pour toute un tableau des petits nombres premiers (jusqu'à 10^7 disons) en utilisant le crible d'Ératosthène.

(2) Pour tester si un nombre n est premier on teste d'abord la divisibilité par des petits nombres premiers. Si l'on trouve un facteur, alors n est composé.

(3) Si pour un grand entier n on n'a pas trouvé de petits facteurs, on passe au test de Miller-Rabin. Si l'on trouve un témoin, alors n est composé, sinon n est probablement premier.

(4) Si n est probablement premier et on exige une preuve, alors on essaiera de factoriser $n-1$ pour trouver un élément d'ordre $n-1$ dans $\mathbb{Z}_n^\times$. Celui-ci prouve que n est premier.

(5) Si n n'est pas premier, alors on essaiera de le factoriser. On extrait d'abord les petits facteurs premiers. Si le facteur restant est trivial ou premier, on a terminé.

(6) Dans le pire des cas, n est composé de grands facteurs. Pour le factoriser on applique la méthode de Pollard ou une méthode plus puissante.

Le principal problème réside dans l'étape 6 : la factorisation d'un nombre composé de grands facteurs. Plusieurs méthodes de factorisation ont été développées, dont chacune a poussé la limite du possible un peu plus loin. Pour en savoir plus on consultera avec profit l'excellent livre de R. Crandall et C. Pomerance [**15**]. Malgré tout progrès dans ce domaine, la factorisation de grands entiers reste un problème difficile, dont on ne connaît pas à ce jour de solution efficace.

Suppose moreover that the numbers p and q are thrown away by mistake,
but that their product pq is saved. How to recover p and q ?
It must be felt as a defeat for mathematics that, in these circumstances, the most promising
approaches are searching the waste paper basket and applying mnemo-hypnotic techniques.
H.W. Lenstra, *Elliptic curves and number-theoretic algorithms*, 1986

PROJET XI

Application à la cryptographie

Objectifs

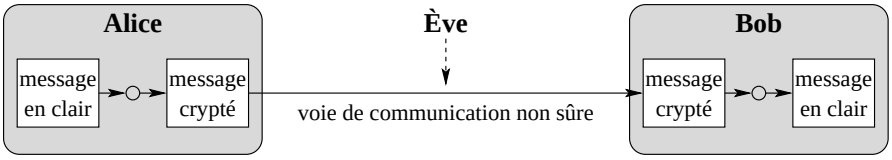
- Comprendre les idées essentielles de la cryptographie à clef publique.
- Implémenter une méthode répandue, la cryptographie selon RSA.

Sommaire

1. Qu'est-ce que la cryptographie ?

Des codes secrets ont été utilisés depuis fort longtemps dans le monde militaire et diplomatique pour assurer la confidentialité des messages. À nos jours l'usage civil s'est répandu avec l'avènement des ordinateurs et surtout de l'internet. Pour une introduction à cette fascinante histoire, je recommande vivement le livre de Simon Singh, *Histoire des codes secrets*, LGF, 2001.

1.1. Le problème de base. Le problème de base peut se résumer comme suit : Alice et Bob souhaitent établir une communication sécurisée afin d'échanger des informations confidentielles. Malheureusement le support (lettre, téléphone, internet) n'est pas sûr et peut être intercepté par la méchante Ève.



L'idée est de crypter des messages avant de les envoyer, de sorte que seule la personne autorisée puisse les décrypter. Alors que l'idée est évidente, la mise en œuvre l'est beaucoup moins !

1.2. Cryptage selon César. D'après la légende, Jules César utilisa un cryptage « alphabétique » suivant le schéma suivant. Les messages sont écrits dans un certain alphabet $\mathscr{A}$, disons l'alphabet latin usuel. Pour le cryptage on fixe une permutation $c\colon \mathscr{A} \to \mathscr{A}$ que l'on l'applique lettre par lettre. Le décryptage est ainsi l'application de la permutation inverse $d = c^{-1}$. Par exemple, avec

$$c = \begin{bmatrix} \texttt{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \\ \texttt{XYZABCDEFGHIJKLMNOPQRSTUVW} \end{bmatrix} \quad \text{et} \quad d = \begin{bmatrix} \texttt{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \\ \texttt{DEFGHIJKLMNOPQRSTUVWXYZABC} \end{bmatrix}$$

le message $m = \texttt{CECI EST UN MESSAGE}$ est crypté en $\bar{m} = m^c = \texttt{ZBZF BPQ RK JBPPXDB}$, puis décrypté en $m = \bar{m}^d = \texttt{CECI EST UN MESSAGE}$. Le programme `cesar.cc` vous fournira d'autres exemples.

Remarque 1.1. Le cryptage alphabétique n'est pas du tout sûr ! Le message crypté dévoile beaucoup trop d'information, et celle-ci peut être utilisée pour casser le code. Dans notre exemple on retrouve la longueur des mots, ce qui laisse deviner au moins les mots très courts. De la même veine, la *fréquence* des lettres (des paires, des triplets, ...) donne des indications assez précises. À noter que dans un texte français typique la lettre 'E' est la plus fréquente. Ainsi une simple analyse statistique du texte crypté dévoilera l'image de 'E', au moins très probablement. (Contempler aussi Georges Perrec, *La disparition*.)

1.3. Attaques statistiques. Pour juger de la sûreté d'un système cryptographique, il faut surtout connaître ses faiblesses : Quelles sont les attaques possibles ? Peut-on décrypter par « force brute » ? Ou par des analyses statistiques ? Ou par d'autres astuces, jusqu'ici inaperçues ? La principale faiblesse du cryptage selon César est qu'il succombe à des analyses statistiques :

Exercice 1.2. Décryptez le message « `VX VHWX XLM MKHI YTVBEX T VTLLXK` ». Explicitez votre démarche, notamment vos hypothèses tacites, puis le système de cryptage/décryptage dévoilé. Un logiciel adapté pourrait-il faire le même travail ? Peut-on être sûr que vous avez retrouvé le message initial ?

Exercice 1.3. Si ce genre de casse-tête vous amuse et que vous cherchez un défi, vous pouvez essayer de décrypter le message suivant : « `ERL BLCNEOGOJQR GFLGOQJCL LTO ER BLE BFET TECL NGJT OQEHQECT BGT OCLT IECL G ILACXBOLC. PQET PLRLV IL DQECRJC FG BCLEPL. GPLA ER BLE IL TOGOJTOJUEL LO ER IJAOJQRRGJCL DCGRAGJT ER BCQSCGNNL BLEO FL DGJCL JRTOGROGRLNLRO.` »

Remarque 1.4 (Bruit aléatoire). Afin de rendre l'analyse statistique plus difficile, on peut ajouter un « bruit aléatoire ». Plus explicitement, on peut identifier l'alphabet $\mathscr{A} = \{\mathtt{A}, \dots, \mathtt{Z}\}$ avec le groupe $\mathbb{Z}_{26}$. À chaque lettre $a \in \mathbb{Z}_{26}$ on peut ainsi ajouter un nombre aléatoire $b \in \mathbb{Z}_{26}$ afin d'obtenir la lettre bruitée $c = a + b$. À la place de la lettre initiale a on note ensuite les deux lettres c et b, afin de retrouver la lettre initiale $a = c - b$. Par exemple, la lettre `E` est maintenant codé par une des paires `EA` ou `FB` ou `GC` etc... Le programme `cesar.cc` implémente cette idée comme option. Qu'en pensez-vous ? Cette astuce rend-elle le code plus sûr ?

Remarque 1.5 (Camouflage). Alice et Bob ont tout intérêt à dévoiler le moins d'information possible :

Éviter des répétitions: Envoyer deux fois le même message m est potentiellement dangereux, car Ève voit passer deux fois le même message crypté $\overline{m}$. Si par exemple le message `JSRYSD-QZMC YSBUVR` déclenche toujours la même action, observable par Ève, ceci permet une interprétation sans explicitement décrypter le message.

Éviter des provocations: Réciproquement, Ève pourrait tenter de provoquer un certain message, par exemple `EVE VIENT DE ME TELEPHONER`. Bien sûr Ève s'attend à ce que le mot `TELEPHONE` apparaisse quelque part dans le message, ce qui facilitera le décryptage.

Éviter le silence: Le fait d'envoyer un message ou d'en envoyer aucun contient de l'information, facilement observable pour Ève. Il vaut mieux crypter puis envoyer des messages comme `CECI EST UN MESSAGE VIDE POUR RAISON DE CAMOUFLAGE`, sans pour autant livrer des informations statistiques gratuites à Ève.

Dans tous ces cas, un bruit aléatoire judicieux permet à Alice et Bob de diminuer des corrélations trop évidentes : entre les parties d'un message, entre messages successifs, ainsi qu'entre messages et informations extérieures. Réciproquement, Ève a intérêt a collectionner toute information accessible, sur une longue durée, afin d'effectuer des analyses statistiques les plus fines possibles.

1.4. Cryptage et décryptage. Le cryptage selon César n'est pas sûr, certes, mais il nous indique une démarche plus générale qui mérite d'être analysée. Notons M l'ensemble des messages, disons des textes utilisant l'alphabet usuel $\mathscr{A} = \{\_, \mathtt{A}, \mathtt{B}, \mathtt{C}, \dots, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$. Par commodité nous supposerons, comme avant, que les messages cryptés sont exprimés dans le même alphabet.

Choix du cryptage et du décryptage: Alice et Bob conviennent au préalable d'utiliser un cryptage $\mathrm{crypt}\colon M \to M$ et un décryptage $\mathrm{décrypt}\colon M \to M$ qu'ils estiment sûr.

Quel niveau de sécurité ? Ici « sûr » veut dire qu'il est difficile pour Ève de retrouver le message en clair m à partir du message crypté $\overline{m}$ dont elle dispose seulement. Par exemple, si l'on crypte lettre par lettre on retombe sur le cryptage à la César. Pour arriver à un bon niveau de sécurité il faut donc un système plus élaboré... Heureusement pour Alice et Bob, il y a beaucoup d'applications $\mathrm{crypt}\colon M \to M$ intéressantes, et possiblement plus « sûres ».

Quels que soient les détails, l'approche naïve souffre de deux problèmes fondamentaux :

Il faut absolument que la méthode reste secrète ! Dès que les fonctions crypt et décrypt sont dévoilées, elles n'offrent plus aucune sécurité. Pour ne pas se fier à une sécurité illusoire, il est donc très important de changer régulièrement le système cryptographique pour assurer un bon niveau de confidentialité. (Il faut prévoir un stock assez large...)

Comment se mettre d'accord sur une méthode de cryptage/décryptage ? Évidemment Alice ne peut pas la détailler à Bob en utilisant le support non sécurisé. L'accord préalable nécessite donc une deuxième voie de communication plus sûre (rendez-vous, lettre recommandée, ligne téléphonique sécurisée, etc.) Ainsi le vrai problème n'est que transféré et reste ouvert.

1.5. Cryptographie à clef. À cause du premier problème ci-dessus il s'avère beaucoup plus utile d'effectuer le cryptage/décryptage en fonction d'une clef, c'est-à-dire d'un paramètre supplémentaire.

Définition 1.6. Un *système cryptographique* à clef consiste en une méthode de cryptage $\mathrm{crypt}\colon C\times M\to M$ et une méthode de décryptage $\mathrm{décrypt}\colon D\times M\to M$, ainsi qu'une méthode pour produire des paires de clefs $(c,d)\in C\times D$ mutuellement inverses. Ceci veut dire que la clef $c\in C$ permet de crypter le message m avec pour résultat le message crypté $\overline{m}=\mathrm{crypt}_c(m)$, tandis que la clef correspondante $d\in D$ permet de décrypter le message $\overline{m}$ avec pour résultat le message initial $m=\mathrm{décrypt}_d(\overline{m})$.

La cryptographie à clef offre au moins deux grands avantages :

(1) On peut désormais publier le système cryptographique (les fonctions crypt et décrypt), par exemple pour l'analyser et le discuter publiquement, ou bien pour le breveter. Pour les utilisateurs Alice et Bob il suffira de garder secrète leur clef respective, c et d.

(2) Comme bénéfice supplémentaire, les utilisateurs peuvent régulièrement changer de clefs, sans pour autant changer entièrement de système cryptographique.

Exemple 1.7. Dans le cryptage alphabétique la clef pour le cryptage est la permutation $c\colon \mathscr{A}\to\mathscr{A}$, et la clef pour le décryptage est la permutation inverse $d=c^{-1}\colon \mathscr{A}\to\mathscr{A}$. Ici $C=D=\mathrm{Sym}(\mathscr{A})$ est le groupe symétrique, et le cryptage/décryptage est l'opération lettre par lettre, comme discuté ci-dessus.

À noter qu'a priori les ensembles C et D n'ont rien à voir : le cryptage et le décryptage sont assez indépendants, on exige seulement qu'une paire de clefs (c,d) fournisse deux fonctions $\mathrm{crypt}_c\colon M\to M$ et $\mathrm{décrypt}_d\colon M\to M$ mutuellement inverses. Plus précisément, pour le décryptage il suffit d'exiger $\mathrm{décrypt}_d\circ\mathrm{crypt}_c=\mathrm{id}_M$. Si l'ensemble M des messages est fini, ceci entraîne la bijectivité, donc automatiquement $\mathrm{crypt}_c\circ\mathrm{décrypt}_d=\mathrm{id}_M$. Ce n'est en général plus vrai pour M infini.

Exemple 1.8 (Camouflage)**.** Pour camoufler les messages on peut ajouter un bruit aléatoire, comme expliqué dans la remarque 1.4, puis crypter le texte ainsi obtenu. Inversement, après le décryptage on supprime la partie aléatoire afin d'obtenir le texte initial. Ce procédé garantit toujours que $\mathrm{décrypt}_d\circ\mathrm{crypt}_c=\mathrm{id}_M$, mais on n'a plus $\mathrm{crypt}_c\circ\mathrm{décrypt}_d=\mathrm{id}_M$. (Le détailler.) Pourquoi est-ce important que M soit infini ?

2. Cryptographie à clef publique

Un système cryptographique à clef permet d'utiliser une méthode publique et de changer fréquemment les clefs. Ceci résout le premier des deux problèmes fondamentaux, mais non le deuxième : comment échanger les clefs via une ligne non sûre ?

Vers 1975 Diffie et Hellman introduisirent le concept de cryptographie à clef publique, qui est l'objet de ce paragraphe. Ce principe permet de résoudre le problème d'échange de clefs d'une manière aussi élégante que radicale : on publie la clef du cryptage ! Il restait à implémenter ce concept…

Soulignons d'abord que le cryptage à la César est incompatible avec le concept de clef publique : la clef pour le cryptage est la permutation $c\colon \mathscr{A}\to\mathscr{A}$. Quand on la publie, il est très facile d'en déduire la clef pour le décryptage, $d=c^{-1}\colon \mathscr{A}\to\mathscr{A}$. Cette symétrie rend impossible une publication *partielle* de clefs : si l'on en connaît une, on en connaît l'autre.

☞ *Pour la cryptographie à clef publique il est primordial que la clef c pour le cryptage ne dévoile pas la clef d pour le décryptage.* ✍

Une telle asymétrie est-elle possible ? En avril 1977 R. Rivest, A. Shamir et L. Adleman proposèrent une implémentation basée sur la difficulté de factoriser des grands nombres entiers. Ce système cryptographique à clef publique est maintenant connu sous le nom de RSA et très largement répandu.

2.1. Le protocole RSA. Rappelons qu'il est relativement facile de trouver deux grands nombres premiers « aléatoires » p et q, puis de calculer leur produit $n=pq$. Par contre, sans aucune connaissance de sa genèse il est en général très difficile de factoriser n. Cette difficulté peut sembler étonnante, elle pourrait même être considérée comme un grave défaut de notre théorie dans l'état actuel. Ironiquement, *l'absence* d'une solution efficace peut aussi avoir des applications intéressantes !

Une telle application fut proposée par Rivest, Shamir et Adleman. Il s'agit d'établir une communication sécurisée entre Bob et Alice par une voie de communication publique. C'est un problème très classique, qui à nos jours s'applique notamment à l'internet.

Choix des clefs: Alice choisit deux grands nombres premiers distincts p et q et calcule $n = pq$ ainsi que $\varphi(n) = (p-1)(q-1)$. Elle choisit également un nombre c premier avec $\varphi(n)$ et calcule un inverse d tel que $cd \equiv 1$ modulo $\varphi(n)$. Tous ces calculs sont faciles à effectuer (le rappeler).

Clef sécrète: Alice connaît le triplet (n,c,d) vérifiant $cd \equiv 1$ modulo $\varphi(n)$. La clef sécrète est la valeur de d, qu'elle garde précieusement pour elle-même.

Clef publique: La clef publique d'Alice est la paire (n,c), qu'elle affiche publiquement. Pour recevoir des messages secrets elle a intérêt que tout le monde connaisse (n,c).

Cryptage: Pour envoyer un texte à Alice, Bob représente son message comme un nombre $m \in [\![0,n[\![$, disons en codant les lettres par $1,\dots,26$, l'espace par 0, et un texte par un développement en base 27. À l'aide de la clef publique d'Alice, il calcule alors le message crypté $\bar{m} = m^c \bmod n$.

Décryptage: Alice reçoit le message crypté $\bar{m} \in [\![0,n[\![$. À l'aide de sa clef sécrète d elle en déduit le message en clair $m = \bar{m}^d \bmod n$. Ceci est possible parce que $cd \equiv 1$ modulo $\varphi(n)$, ce qui assure $\bar{m}^d = m^{cd} = m$ dans $\mathbb{Z}_n^\times$.

Exercice/M 2.1 (Inversibilité)**.** Vérifier que $\mathrm{crypt}_c \colon \mathbb{Z}_n \to \mathbb{Z}_n$, $m \mapsto m^c$ et $\mathrm{décrypt}_d \colon \mathbb{Z}_n \to \mathbb{Z}_n$, $\bar{m} \mapsto \bar{m}^d$ définissent deux applications mutuellement inverses.

Remarque 2.2. La sécurité du système RSA se base sur les hypothèses suivantes :
- Sans connaître p et q, il est difficile de trouver $\varphi(n)$ à partir de n.
- Sans connaître $\varphi(n)$, il est difficile de trouver d à partir de c.
- Sans connaître d, il est difficile de trouver m à partir de $\bar{m}$.

D'un point de vue pratique ces hypothèses sont vérifiées dans le sens qu'à l'heure actuelle il n'existe pas d'algorithme efficace publiquement connu pour factoriser des grands entiers n, ni pour calculer $\varphi(n)$ à partir de n, ni pour calculer d à partir de c, ni pour calculer m à partir de $\bar{m}$. Toutefois il est important à souligner que la sécurité du système RSA se base sur l'expérience et non sur une preuve mathématique.

Remarque 2.3. Soit n composé de deux nombres premiers distincts. Alors trouver $\varphi(n)$ équivaut à factoriser n. Effectivement, la factorisation $n = pq$ permet de calculer $\varphi(n) = (p-1)(q-1)$. Réciproquement les racines du polynôme $X^2 + (\varphi(n) - n - 1)X + n$ sont les facteurs p et q cherchés.

2.2. Implémentation du protocole RSA.

Exercice 2.4 (Puissance modulaire)**.** Implémenter une fonction

```
Integer puissance( Integer base, Integer exp, const Integer& mod )
```

qui calcule efficacement $b^e \bmod n$ pour des entiers $b \in \mathbb{Z}$, $e \geq 0$, $n \geq 1$.

Exercice 2.5 (Inverse modulaire)**.** Implémenter une fonction

```
Integer inverse( Integer a, const Integer& mod )
```

qui calcule efficacement l'inverse de a modulo n, si a est inversible, et qui renvoie 0 sinon. S'il existe, l'inverse de a est représenté par l'unique entier $u \in \{1,\dots,n-1\}$ tel que $au \equiv 1 \pmod{n}$.

Exercice 2.6 (Développement d'un entier en une base donnée)**.** Rappeler la méthode de Horner et l'utiliser pour écrire deux fonctions

```
Integer convert( const vector<Integer>& chiffres, const Integer& b )
vector<Integer> convert( Integer n, const Integer& b )
```

qui effectuent le développement en base `b` d'un entier de type `Integer`. Pour une spécification détaillée voir le fichier `rsa.cc`. Vous y trouvez également la fonction `renverser(vec)` qui pourra vous être utile.

On se propose de coder un *texte* selon la méthode RSA. Pour cela il faut d'abord transformer le texte en entier, puis retransformer un entier en texte. Vous trouverez dans `rsa.cc` une petite fonction `majuscules( string& texte )` qui transforme un texte en lettres majuscules et underscore '`_`'.

Exercice 2.7 (Transformations entre textes et entiers)**.** On interprète les lettres `A,...,Z` comme nombres $1,\dots,26$ et '`_`' comme 0. Ce sont donc les « chiffres » en base $b = 27$. On peut ainsi interpréter un texte

$t = (c_k, c_{k-1}, \ldots, c_1, c_0)$ comme l'entier $m = c_k b^k + c_{k-1} b^{k-1} + \cdots + c_1 b^1 + c_0$, et réciproquement tout entier m comme un texte t. Écrire ainsi deux fonctions

```
Integer convert( const string& texte )
string convert( Integer nombre )
```

qui effectuent la transformation entre texte et entier. Testez vos fonctions sur quelques exemples (à joindre en commentaire au fichier source). Les transformations devraient être inverses l'une à l'autre.
Indication. — Après avoir calculé `Integer c= n%27` la conversion `int(c)` n'est malheureusement pas acceptée ; il faut appeler la fonction `c.get_ui()`, notation un peu lourde pour *get unsigned integer.*

Exercice 2.8 (Cryptage RSA)**.** Écrire une fonction qui crypte un texte selon la méthode RSA :

```
cryptageRSA( const Integer& mod, const Integer& exp, string& texte )
```

On se servira des fonctions précédentes, avec les notations $n =$ `mod`, $c =$ `exp`, $t =$ `texte`. Le texte t est transformé en un entier m, puis m est développé en $m_k, m_{k-1}, \ldots, m_1, m_0 \in \{0, \ldots, n-1\}$ en base n. Ensuite on peut calculer $m'_i \leftarrow m_i^c \bmod n$ et en déduire l'entier crypté m', puis le texte crypté t'.

Test de correction. — La fonction `cryptage( istream& in, ostream& out )` lit les données n, c, t du flot d'entrée et écrit n, c, t' dans le flot de sortie. Si votre programme compilé s'appelle `cryptage`, alors la commande `./cryptage < test1.rsa` produira le résultat `15 3 FELICITATIONS`. Vérifier aussi le (dé)cryptage dans `test2.rsa`. Quel message se cache dans `test3.rsa` ?

Exercice 2.9 (Question bonus)**.** Si cela vous inspire, essayez de décrypter le message dans `test4.rsa`. Pour la décomposition en facteurs premiers vous pouvez employer tout logiciel à votre disposition. Quels consignes formuleriez-vous pour le choix des nombres premiers p et q afin d'assurer la sûreté du système RSA ? Sur quelles informations ou hypothèses vos consignes se basent-elles ?

Exercice 2.10 (Question bonus)**.** En quoi se ressemblent les cryptages selon César et selon RSA ? Quelle est la principale différence ? Pourquoi pour César la publication de la clef c serait catastrophique tandis que pour RSA elle n'est pas considérée comme inquiétante ?

2.3. Production de clefs. Jusqu'ici on sait appliquer le cryptage/décryptage à une clef (n, c) et un texte donnés. Ce dernier paragraphe implémente finalement la production de clefs (n, c) convenables.

Exercice 2.11 (Test de primalité)**.** Implémenter une fonction

```
bool estPseudoPremier( const Integer& n, const Integer& x )
```

qui teste si n est pseudo-premier par rapport à la base x, puis

```
bool estProbablementPremier( Integer n, int essais=100 )
```

qui effectue le test de Miller-Rabin (jusqu'à 100 fois). En déduire une fonction

```
Integer produirePremierAleatoire( Integer pmin, Integer pmax )
```

qui produit un nombre premier aléatoire entre $p_{\min}$ et $p_{\max}$.

Exercice 2.12 (Production de clefs)**.** Écrire une fonction

```
produireClefRSA( int bits, Integer& n, Integer& c, Integer& d )
```

qui produit une clef aléatoire, de la taille souhaitée en bits, pour le cryptage RSA. Avec ces clefs aléatoires, tester le cryptage/décryptage sur quelques exemples (à joindre en commentaire au fichier source).

2.4. Signature et authentification.

Exercice 2.13. Expliquer comment Alice peut *signer* un document électronique, de sorte que tout le monde puisse *vérifier* la signature, mais personne ne puisse la *falsifier*. *Indication.* — La signature d'un document D en clair peut être un message m qui répète le document D et ajoute « lu et approuvé, Alice ». Évidemment ce message est trivial à falsifier. Pour cette raison Alice donne comme signature $\bar{m} = m^d \bmod n$. Discuter la question à savoir si cette méthode résout le problème de signature.

Exercice 2.14. Produire une clef publique (n, c) avec clef secrète d (que vous ne publiez pas, évidemment). Écrire un petit message en guise de conclusion de ce projet, puis signez-le avec votre clef secrète d. Je vérifierai votre signature avec votre clef publique (à joindre avec le message signé).

Partie D

Anneaux effectifs

The mathematician is entirely free,
within the limits of his imagination,
to construct what worlds he pleases.
John W. N. Sullivan

CHAPITRE XII

Exemples d'anneaux effectifs

Objectifs

- Expliciter ce qu'il faut pour rendre un anneau effectif.
- Implémenter les nombres rationnels comme un exemple de base.
- Plus généralement, implémenter le corps des fractions ou un anneau quotient.

Un anneau est *effectif* s'il existe une manière de *représenter* chacun de ses éléments sur ordinateur et des algorithmes pour *effectuer* chacune des opérations de l'anneau dans la représentation choisie. L'approche effective est une vraie restriction quant aux anneaux en question et un important défi quant à l'ingéniosité algorithmique. Ce chapitre précise ce que l'on exige d'un anneau effectif, discute quelques exemples typiques, et présente des implémentations possibles.

Notre exemple phare est, bien sûr, l'anneau $\mathbb{Z}$. La représentation des entiers et les algorithmes de base ont été discutés aux chapitres II, IX, et XI. On discute dans le présent chapitre son corps des fractions $\mathbb{Q}$ et on reconsidère les quotients $\mathbb{Z}_n$, constructions que l'on généralise à d'autres anneaux. Le projet à la fin traite de l'anneau $\mathbb{Z}[i]$ des entiers de Gauss.

Voici trois exemples importants et assez représentatifs pour illustrer l'étendue du concept :

Exemple 0.1. L'anneau $\mathbb{Q}[\sqrt{2}]$ est effectif : c'est un $\mathbb{Q}$-espace vectoriel à base $(1,\sqrt{2})$. On peut donc travailler avec des couples $(a,b)\in\mathbb{Q}^2$ pour représenter tout élément $a+b\sqrt{2}$ de manière unique. *Exercice.* — Exprimer les opérations de l'anneau $\mathbb{Q}[\sqrt{2}]$ sous cette forme.

Exemple 0.2. Si A est effectif, alors l'anneau $A[X]$ des polynômes sur A est effectif. *Exercice.* — L'idée évidente marchera : on représente tout polynôme $P\in A[X]$ par la suite *finie* de ses coefficients dans A. Détailler cette représentation et les opérations de l'anneau $A[X]$.

Exemple 0.3. Le corps $\mathbb{R}$ des nombres réels, par contre, n'est pas effectif. Ceci surprend beaucoup les débutants, mais la vie est ainsi faite. Évidemment le corps $\mathbb{R}$ est de grande importance ; toute l'analyse s'appuie sur lui, et dans des applications il est souvent indispensable d'effectuer des calculs « réels » sur ordinateur. Les difficultés *d'approcher* des calculs dans $\mathbb{R}$ sur ordinateur font l'objet du calcul numérique.

Attention. — Les types `float` et `double` *ne modélisent pas* le corps des nombres réels ! Loin de cela, ils ne représentent qu'un *sous-ensemble fini* de nombres rationnels, ce qui n'a rien à voir avec $\mathbb{R}$.

Remarque 0.4. Voici un argument intuitif mais *faux* que $\mathbb{R}$ n'est pas effectif : « il est impossible de stocker une suite infinie de décimales pour représenter un nombre irrationnel de manière exacte. » Certes, mais nul ne nous oblige de représenter nos nombres par un développement décimal ! Les sous-anneaux $\mathbb{Q}[\sqrt{2}]$ et $\mathbb{Q}[\pi]$ de $\mathbb{R}$, par exemple, contiennent des nombres irrationnels, mais on peut très bien les représenter sur ordinateur ! (Comment ?)

Si l'argument intuitif est faux, il nous met tout de même sur la bonne piste. Après réflexion, on en extrait une réponse plus solide : la représentation d'un nombre x sur ordinateur doit se faire par un objet fini (arbitrairement grand, mais toujours fini pour chaque x individuellement). On peut ainsi représenter seulement un nombre *dénombrable* d'éléments, mais le corps $\mathbb{R}$ est *non dénombrable*.

Sommaire

1. De l'anneau $\mathbb{Z}$ au corps $\mathbb{Q}$

On commence par un exemple concret et pratique, que l'on généralisera dans la suite : le passage de l'anneau $\mathbb{Z}$ des nombres entiers au corps $\mathbb{Q}$ des nombres rationnels. On révisera d'abord la construction du corps des fractions, puis on passera à l'implémentation en C++.

1.1. Qu'est-ce qu'un corps de fractions ? Certaines équations du type $a = bx$ avec $a, b \in \mathbb{Z}$ n'ont pas de solution x dans $\mathbb{Z}$. C'est le cas, par exemple, pour l'équation $2 = 3x$. Il serait pourtant très utile de disposer d'une solution ! Pour l'anneau $\mathbb{Z}$ vous connaissez bien sûr la solution de ce problème : on construit le *corps des fractions* $\mathbb{Q}$. On établit ainsi une théorie plus large mais toujours cohérente, dans laquelle le problème initial se retrouve et se résolve. Essayons d'en dégager une réponse générale, qui englobe tous les anneaux commutatif unitaires.

Exercice/M 1.1. Soit A un anneau commutatif unitaire. Optimiste que nous sommes, supposons dans un premier temps qu'il existe un homomorphisme injectif $\iota : A \to L$ dans un corps L. Afin de simplifier la notation nous pouvons identifier A avec son image $\iota(A)$, et ainsi supposer que $A \subset L$. (Pourquoi ?) Dans ce cas on peut regarder l'ensemble $\mathrm{Frac}(A) \subset L$ formé des éléments $\frac{a}{b} = ab^{-1}$ tels que $a \in A$ et $b \in A^*$. Vérifier d'abord que $\frac{a}{b} = \frac{c}{d}$ si et seulement si $ad = bc$. Établir ensuite les règles suivantes : $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$ et $\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}$. En déduire que $\mathrm{Frac}(A)$ est un sous-corps de L, contenant A. Plus précisément c'est le plus petit sous-corps de L qui contienne A. On l'appelle *corps des fractions de A dans L*. Forts de cette vérification rassurante, formulons-en la définition générale :

Définition 1.2. Soit A un anneau commutatif unitaire. Un *corps de fractions* de A est la donnée d'un couple (K, ι) où K est un corps et $\iota : A \to K$ est un homomorphisme d'anneaux injectif, de sorte que tout élément $x \in K$ s'écrive comme $x = \iota(a)\iota(b)^{-1}$ avec $a \in A$ et $b \in A^*$.

Exemple 1.3. $\mathbb{Q}$ est un corps de fractions de $\mathbb{Z}$ via l'inclusion $\iota : \mathbb{Z} \subset \mathbb{Q}$. Par contre, $\mathbb{R}$ n'est pas un corps de fractions de $\mathbb{Z}$: certains éléments $x \in \mathbb{R}$ ne s'écrivent pas comme $\frac{a}{b}$ avec $a \in A$ et $b \in A^*$. Soit p premier et $\phi : \mathbb{Z} \to \mathbb{Z}_p$ l'homomorphisme canonique. Tout élément $x \in \mathbb{Z}_p$ s'écrit comme $\phi(a)\phi(1)^{-1}$ avec $a \in \mathbb{Z}$. Le couple $(\mathbb{Z}_p, \phi)$ n'est cependant pas un corps de fractions, car l'homomorphisme ϕ n'est pas injectif.

Exercice/M 1.4. Un corps de fractions (K, ι) de A se réjouit de la propriété universelle suivante : si $\phi : A \to L$ est un homomorphisme injectif dans un corps L, alors il existe un et un seul homomorphisme $\Phi : K \to L$ tel que $\Phi \circ \iota = \phi$. Il en découle que le corps des fractions (K, ι) de A est unique à isomorphisme près. Formulez précisément ce que cela veut dire, puis prouvez votre énoncé.

Exercice/M 1.5. Contrairement à l'unicité, l'existence n'est pas toujours donnée : montrer qu'un anneau non intègre n'admet pas de corps de fractions. Heureusement pour nous, c'est le seul obstacle :

Théorème 1.6. *Tout anneau intègre admet un corps de fractions.*

Exercice/M 1.7. Où chercher ce corps ? Comment prouver son existence ? Il n'y a qu'une seule possibilité : il faut le construire ! Démontrer le théorème en détaillant la construction suivante.

ESQUISSE DE PREUVE. Supposons que A est un anneau intègre. Notre but est de donner un sens aux quotients $\frac{a}{b}$ dans un corps encore à construire. En s'inspirant des propriétés souhaitées, on considère l'ensemble $F = A \times A^*$ avec les opérations $(a,b) + (c,d) := (ad + bc, bd)$ et $(a,b) \cdot (c,d) := (ac, bd)$. On constate que $(F, +, \cdot)$ *n'est pas* un anneau, encore moins un corps. (Pourquoi ? Il y a une exception : laquelle ?) Afin de sortir de cette impasse, on définit la relation $(a,b) \sim (c,d)$ si $ad = bc$. On vérifie d'abord qu'il s'agit d'une relation d'équivalence, ensuite que les opérations $+$ et $\cdot$ sont bien définies sur le quotient $K = F/\sim$, et finalement que $(K, +, \cdot)$ est un corps. (Le détailler ! Où utilise-t-on l'intégrité de l'anneau A dans ce raisonnement ?) La classe d'équivalence de (a,b) est notée $\frac{a}{b}$. L'application $\iota : A \to K$ donnée par $a \mapsto \frac{a}{1}$ est un homomorphisme d'anneaux injectif. On peut ainsi identifier l'anneau A avec son image $\iota(A)$. Ceci fait, on constate que tout élément $x \in K$ s'écrit comme $x = ab^{-1}$ avec $a \in A$ et $b \in A^*$. □

Exercice/M 1.8. Vérifier que cette construction appliquée à $\mathbb{Z}$ donne le corps $\mathbb{Q}$ comme vous le connaissez. Tout corps de caractéristique 0 contient donc $\mathbb{Q}$ comme un sous-corps.

Exercice/M 1.9. Qu'obtient-on si l'on l'applique à $\mathbb{R}[X]$? Plus généralement à $K[X]$ sur un corps K ? Que se passe-t-il lorsqu'on l'applique à un anneau non intègre, comme $\mathbb{Z}_6$ par exemple ?

1.2. Implémentation du corps $\mathbb{Q}$. Une *classe* en C++ est un type défini par l'utilisateur. La construction de classes pour modéliser une situation donnée est le paradigme caractéristique de tout langage orienté objet. Ne citons que l'exemple qui nous a occupé dans les derniers chapitres : le type `int` de C++ ne modélise pas $\mathbb{Z}$ mais un anneau fini $\mathbb{Z}_n$, typiquement avec $n = 2^{32}$ ou $n = 2^{64}$. On a donc employé la classe `Integer`, issue de la bibliothèque GMP, qui modélise l'anneau $\mathbb{Z}$.

Jusqu'à présent nous ne disposons pas d'outils pour calculer avec des nombres rationnels. Ayant acquis suffisamment d'expérience, nous allons maintenant construire une telle classe nous-mêmes. À titre d'exemple, nous souhaitons une utilisation comme dans le programme suivant :

Programme XII.1 Exemple d'utilisation de la classe `Rationnel`

```
int main()
{
  cout << "Bienvenue au corps des nombres rationnels !" << endl
       << "Entrez deux éléments a,b sous la forme  num/den  svp : ";
  Rationnel a,b;
  cin >> a >> b;
  cout << "a   = " << a   << endl << "b   = " << b   << endl
       << "a+b = " << a+b << endl << "a-b = " << a-b << endl
       << "a*b = " << a*b << endl << "a/b = " << a/b << endl;
}
```

Nombres rationnels, version 0.1. Après s'être assuré de la construction du corps des fraction, nous allons « simplement » la traduire en C++ et ainsi construire notre classe `Rationnel`. Il faut d'abord choisir une représentation convenable, c'est-à-dire une structure des données adéquate. Dans notre cas c'est facile : tout élément $\frac{r}{s} \in \mathbb{Q}$ est caractérisé par deux entiers, un numérateur $r \in \mathbb{Z}$ et un dénominateur $s \in \mathbb{Z}^*$. En C++ ceci peut être modélisé comme suit :

Programme XII.2 Implémentation de la classe `Rationnel` – version 0.1

```
class Rationnel
{
public:
  Integer numer;
  Integer denom;
};
```

Ici le mot réservé `class` est suivi du nom de la classe, en l'occurrence `Rationnel`. Le corps de la classe est délimité par des accolades suivies d'un point-virgule. Il contient la définition des *éléments* ou *membres* de la classe : ici les deux variables `numer` et `denom` de type `Integer`.

Exemple 1.10. Avec cette définition de la classe `Rationnel` on pourrait écrire le code suivant :

```
Rationnel a;  a.numer= 4; a.denom= 6;
Rationnel b;  b.numer= 1; b.denom= 5;
```

La première ligne définit la variable `a` du type `Rationnel` ; on dit aussi que `a` est un *objet* de la classe `Rationnel`. Cet objet possède deux éléments : les variables `a.numer` et `a.denom`, auxquelles on affecte les valeurs `4` et `6` respectivement. Notre objet regroupe ces deux éléments en une seule entité. Il en est de même pour l'objet `b`. Lors de sa définition (ligne 2) sont créés ses deux éléments `b.numer` et `b.denom`, différents des éléments de `a` puisqu'il s'agit d'un *autre* objet. On dit que `a` et `b` sont deux *instances*, deux objets distincts de la classe `Rationnel`.

Remarque 1.11. Chaque objet `a`, `b`, ... mène sa vie individuelle : en particulier *l'état* d'un objet (la valeur prise par chacune des variables) est une *propriété individuelle*. Par contre, la classe décrit les *propriétés communes* : elle spécifie en l'occurrence que les deux éléments `numer` et `denom` sont toujours de type `Integer`, quelque soit l'objet en question. Comme ces deux éléments sont déclarés *publics*, on peut les utiliser comme des variables usuelles (voir l'exemple ci-dessus).

Nombres rationnels, version 0.2. Nous allons maintenant affiner notre implémentation en ajoutant des *méthodes* à la classe `Rationnel`. Commençons par la *normalisation*, qui repose sur une particularité des nombres rationnels (bien connue mais remarquable — la prouver) :

Proposition 1.12. *Tout nombre rationnel s'écrit comme une fraction rs^{-1} avec $r \in \mathbb{Z}$, $s \in \mathbb{Z}^*$, et on peut normaliser cette fraction de sorte que* $\operatorname{pgcd}(r,s) = 1$ *et* $s > 0$. *L'écriture normalisée est unique : si $rs^{-1} = uv^{-1}$ et chacune des fractions est normalisée, alors $r = u$ et $s = v$.*

La fonction `normaliser()` ci-dessous réduit toute fraction à cette forme normale.

Programme XII.3 Implémentation de la classe `Rationnel` – version 0.2

```
class Rationnel
{
public:
  Integer numer, denom;   // numérateur et dénominateur

  void normaliser(void)   // normaliser la fraction numer/denom
    { if ( denom==0 ) { cerr << "Division par zéro !\n"; exit(1); };
      Integer d= pgcd( numer, denom ); numer/= d; denom/= d;
      if ( denom<0 ) { numer= -numer; denom= -denom; }; };
};
```

Exemple 1.13 (suite)**.** On peut maintenant écrire

```
Rationnel a;  a.numer= 4; a.denom= 6;  a.normaliser();
```

Ceci réduit la représentation $4/6$ à la forme normale $2/3$, moyennant une fonction `pgcd` pour les entiers. La fonction `normaliser()` appartient à la classe `Rationnel`, ce qui est plus explicitement noté par `Rationnel::normaliser()`. Afin d'y accéder de l'extérieur de la classe, on appelle la fonction obligatoirement basée sur un objet, comme `a.normaliser()` dans l'exemple. Cet appel entraîne, naturellement, que la fonction soit appliquée à l'objet `a`. Il n'y a pas de sens d'appeler `normaliser()` sans aucun objet.

Remarque 1.14. Il serait également possible d'écrire une fonction

```
void normal( Rationnel& a )
{ if ( a.denom==0 ) { cerr << "Division par zéro !\n"; exit(1); };
  Integer d= pgcd( a.numer, a.denom ); a.numer/= d; a.denom/= d;
  if ( a.denom<0 ) { a.numer= -a.numer; a.denom= -a.denom; }; }
```

En utilisant cette fonction, notre exemple prendrait la forme

```
Rationnel a;  a.numer= 4; a.denom= 6;  normal(a);
```

Le résultat est le même, mais le point de vue est différent : la fonction `normal()` ne fait pas partie de la classe `Rationnel`, c'est une fonction extérieure. Par contre `Rationnel::normaliser()` est une méthode de la classe. Ceci peut entraîner différents droits d'accès, comme on verra plus bas.

Notez aussi la différence dans les noms des éléments : la fonction `normal()` doit adresser les éléments par `a.numer` et `a.denom`, tandis que la fonction `Rationnel::normaliser()` les appelle simplement `numer` et `denom` : évoquée par `a.normaliser()`, elle modifiera les éléments `a.numer` et `a.denom`.

Nombres rationnels, version 0.3. Afin d'*encapsuler* données et méthodes, une classe regroupe toutes les méthodes « essentielles » ou « caractéristiques » : elles font partie des propriétés communes des objets. Après avoir implémenté la normalisation nous voudrions maintenant ajouter des méthodes d'*affectation*. À titre d'exemple, on voudrait écrire

```
Rationnel a;  a.affecter(4,6);
```

ce qui devrait correspondre à `a.numer=4; a.denom=6;` suivi d'une normalisation. C'est une écriture naturelle, et la normalisation systématique évite que l'utilisateur oublie éventuellement d'appeler la fonction `normaliser()`. Voici une implémentation possible :

Programme XII.4 Implémentation de la classe `Rationnel` – version 0.3

```
class Rationnel
{
public:
  Integer numer, denom;   // numérateur et dénominateur

  void normaliser(void)   // normaliser la fraction numer/denom
    { if ( denom==0 ) { cerr << "Division par zéro !\n"; exit(1); };
      Integer d= pgcd(numer,denom); numer/= d; denom/= d;
      if ( denom<0 ) { numer= -numer; denom= -denom; }; };

  void affecter( const Integer& num, const Integer& den=1 )
    { numer= num; denom= den; normaliser(); };

  Rationnel& operator= ( const Rationnel& a )
    { numer= a.numer; denom= a.denom; return *this; };
};
```

Remarque 1.15. On voit dans la fonction `affecter()` que l'appel de la fonction `normaliser()` n'est pas précédé du nom de l'objet. Le compilateur comprend qu'il s'agit de l'objet courant sur lequel travaille la fonction `affecter()`.

Remarque 1.16. Notre implémentation se sert d'un paramètre initialisé par défaut. On peut ainsi appeler la fonction `affecter()` avec un seul argument, `num`, dans quel cas le compilateur pose `den=1` pour le deuxième argument. Ceci correspond à la conversion d'un entier de type `Integer` en `Rationnel` : on peut par exemple écrire `Rationnel a; a.affecter(5);` ce qui semble une écriture assez naturelle.

Quant à l'affectation par copie, on préfère évidemment une écriture courte et naturelle comme `a=b` à l'écriture longue est fastidieuse `a.numer=b.numer; a.denom=b.denom;` Dans ce but nous avons introduit l'opérateur `=` qui réalise cette affectation par copie. À noter que `a=b;` correspond plus explicitement à l'appel `a.operator=(b);` (le tester).

Remarque 1.17. En C++ un opérateur d'affectation renvoie une référence sur l'objet auquel on vient d'affecter la valeur. L'implémentation de la classe `Rationnel` se conforme à cette convention : Mais comment une fonction sait-elle sur quel objet elle opère ? Afin de résoudre cette crise d'identité, le mot réservé `this` fournit un pointeur sur l'objet courant, et `*this` est synonyme avec cet objet.

Nombres rationnels, version 0.4. Après avoir implémenté normalisation et affectation, nous voudrions ajouter un *constructeur*. Celui-ci est appelé exactement une fois lors de la création de l'objet, et sert à l'initialisation. En l'occurrence on souhaite que la définition `Rationnel a;` crée un objet `a` dont les deux éléments `a.numer` et `a.denom` soient initialisés à `0` et `1` respectivement. Il sera également commode de disposer des variantes à paramètres :

```
Rationnel a(4,6);                 // création de a avec initialisation simultanée
Rationnel b= Rationnel(4,6);      // création de b puis d'un objet auxiliaire anonyme
Rationnel c; c= Rationnel(4,6);   // création de c puis d'un objet auxiliaire anonyme
Rationnel d; d.affecter(4,6);     // création de d suivie d'une affectation séparée
Rationnel e; e.numer=4; e.denom=6; e.normaliser();
```

Ces définitions aboutissent toutes aux même résultat. La première est pourtant la plus naturelle et la plus efficace. L'implémentation suivante réalise un tel constructeur.

Remarque 1.18. Ne pas confondre construction et affectation : le constructeur n'est appelé qu'une seule fois pour l'objet `a`, à savoir lors de sa création. L'affectation par contre, s'effectue sur un objet déjà construit, et peut s'effectuer plusieurs fois pendant sa vie. Il existe aussi la notion de destructeur, qui est appelé pour détruire un objet à la fin de sa vie. Ici les variables `numer` et `denom` sont déjà munies du destructeur de la classe `Integer`, qui fait le nécessaire.

Programme XII.5 Implémentation de la classe `Rationnel` – version 0.4

```
class Rationnel
{
public:
  Integer numer, denom;   // numérateur et dénominateur

  void normaliser(void)   // normaliser la fraction numer/denom
    { if ( denom==0 ) { cerr << "Division par zéro !\n"; exit(1); };
      Integer d= pgcd(numer,denom); numer/= d; denom/= d;
      if ( denom<0 ) { numer= -numer; denom= -denom; }; };

  Rationnel( const Integer& num=0, const Integer& den=1 )
    : numer(num), denom(den) { normaliser(); };

  void affecter( const Integer& num, const Integer& den=1 )
    { numer= num; denom= den; normaliser(); };

  Rationnel& operator= ( const Rationnel& a )
    { numer= a.numer; denom= a.denom; return *this; };
};
```

Remarque 1.19. Un constructeur porte toujours le nom de la classe, `Rationnel` en l'occurrence. Comme il ne sert qu'à la création d'objets, il est impossible de spécifier un type de retour ; il ne renvoie jamais de valeur. Le constructeur peut prendre une liste de paramètres, ici deux paramètres de type `Integer` qui seront interprétés comme numérateur et dénominateur. L'implémentation initialise la variable `numer` à la valeur `num` et la variable `denom` à la valeur `den`, puis appelle la normalisation. Notre implémentation se sert à nouveau des paramètres initialisés par défaut. On peut donc appeler le constructeur sans aucun argument, dans quel cas le compilateur pose `num=0` et den=1. On peut l'appeler aussi avec un seul argument `num` : le compilateur pose alors `den=1`, ce qui correspond à la conversion de `Integer` en `Rationnel`.

Nombres rationnels, version 0.5. Jusqu'à présent la classe `Rationnel` n'est pas très utile : elle regroupe deux éléments `numer` et `denom` de type `Integer` mais elle ne permet pas encore de modéliser le corps $\mathbb{Q}$. Pour cela il faut encore implémenter les *opérations* nécessaires ; le programme XII.6 présente une implémentation plus complète.

Exercice/P 1.20. Tester l'implémentation de la classe `Rationnel` par un programme similaire à XII.1. Les opérations de base fonctionnent-elles comme souhaité ? Essayer d'expliquer leur fonctionnement en détail. Tester aussi les limites de notre implémentation actuelle : élargir votre programme en ajoutant de code de plus un plus exigeant vis à vis les opérations sur les nombres rationnels… Trouvez-vous des erreurs ? des incohérences ? des comportements contre-intuitifs ? Si oui, comment remédier à ces défauts ?

Exercice 1.21. Tester les définitions/affectations suivantes puis détailler leur fonctionnement.

```
Rationnel a(20,6);                      // ok, écriture fortement conseillée
Rationnel b= Rationnel(20,6);           // acceptable, mais non optimale
Rationnel c; c.affecter(20,6);          // acceptable, mais non optimale
Rationnel d= Rationnel(20)/Rationnel(6); // acceptable, encore moins optimale
Rationnel e(20); e.denom= 6;            // écriture désormais interdite
Rationnel f(20/6);                      // écriture trompeuse, à éviter
Rationnel g= 20/6;                      // écriture trompeuse, à éviter
```

Remarque 1.22. Dans la version 0.5 on a opté pour une restriction d'accès. Précédemment l'utilisateur pouvait créer des fractions non normalisées, par exemple en affectant `a.numer=4; a.denom=6;` car ces éléments étaient publics. Nul ne l'obligeait d'appeler la normalisation.

Désormais la classe contrôle toute écriture et toute lecture de ses éléments. Pour ce faire les éléments concernés sont déclarés *privés* : ils ne sont plus accessibles de l'extérieur de la classe. Pour lire le numérateur et le dénominateur on se sert de deux fonctions `numerateur()` et `denominateur()` qui réalisent la lecture (et la lecture seule). L'affectation via la fonction `affecter` reste en place comme avant. Comme elle appelle systématiquement la normalisation, on garantit ainsi que toute fraction soit stockée sous forme normale.

Programme XII.6 Implémentation de la classe `Rationnel` – version 0.5 `rationnel0.cc`

```
#include <iostream>
#include "integer.cc"
using namespace std;

class Rationnel
{
private:
  Integer numer, denom;   // numérateur et dénominateur
  void normaliser(void)   // normaliser la fraction numer/denom
    { if ( zero(denom) ) { cerr << "Division par zéro !\n"; exit(1); };
      Integer d= pgcd(numer,denom); numer/= d; denom/= d;
      if ( denom<0 ) { numer= -numer; denom= -denom; }; };

public:
  // Constructeurs divers
  Rationnel( int num=0, int den=1 )
    : numer(num), denom(den) { normaliser(); };
  Rationnel( const Integer& num, const Integer& den=1 )
    : numer(num), denom(den) { normaliser(); };
  Rationnel( const Rationnel& a )
    : numer(a.numer), denom(a.denom) {};

  // Lire le numérateur et le dénominateur
  const Integer& numerateur() const { return numer; };
  const Integer& denominateur() const { return denom; };

  // Affectation
  void affecter( const Integer& num=0, const Integer& den=1 )
    { numer= num; denom= den; normaliser(); };
  Rationnel& operator= ( const Rationnel& a )
    { numer= a.numer; denom= a.denom; return *this; };

  // Addition
  Rationnel operator+ ( const Rationnel& a ) const
    { return Rationnel( numer*a.denom + denom*a.numer, denom*a.denom ); };
  Rationnel& operator+= ( const Rationnel& a )
    { affecter( numer*a.denom + denom*a.numer, denom*a.denom ); return *this; };

  // Opposé et soustraction
  Rationnel operator- () const
    { return Rationnel( -numer, denom ); };
  Rationnel operator- ( const Rationnel& a ) const
    { return Rationnel( numer*a.denom - denom*a.numer, denom*a.denom ); };
  Rationnel& operator-= ( const Rationnel& a )
    { affecter( numer*a.denom - denom*a.numer, denom*a.denom ); return *this; };

  // Multiplication (correcte mais non optimisée)
  Rationnel operator* ( const Rationnel& a ) const
    { return Rationnel( numer*a.numer, denom*a.denom ); };
  Rationnel& operator*= ( const Rationnel& a )
    { affecter( numer*a.numer, denom*a.denom ); return *this; };

  // Division (correcte mais non optimisée)
  Rationnel operator/ ( const Rationnel& a ) const
    { return Rationnel( numer*a.denom, denom*a.numer ); };
  Rationnel& operator/= ( const Rationnel& a )
    { affecter( numer*a.denom, denom*a.numer ); return *this; };

  // Entrée et sortie
  friend istream& operator>> ( istream& in, Rationnel& a );
  friend ostream& operator<< ( ostream& out, const Rationnel& a );
};
```

1.3. Le principe de base : encapsuler données et méthodes ! À l'instar de la classe `Integer` modélisant les entiers, nous disposons désormais d'une classe `Rationnel` modélisant les nombres rationnels. Même dans ce petit exemple on reconnaît le germe de la programmation orientée objet : la définition d'une *classe* consiste en une structure de données ainsi que des méthodes opérant sur ces données. En reprenant le paradigme de N. Wirth on pourrait dire :

☞ *« classe = données + méthodes »* ✍

L'avantage d'une telle démarche est de regrouper, d'une manière logique et naturelle, tous ce qu'il faut pour modéliser les objets en question : en l'occurrence les nombres rationnels avec leurs opérations naturelles. Ce point de vue est tellement utile, voire nécessaire, pour organiser de projets importants, que l'on en fait un principe de programmation :

☞ *Tout objet gère ses données de manière autonome, sans intervention extérieure directe.* ✍

La communication avec le monde extérieur se fait uniquement via certaines interfaces bien choisies. Elles constituent la face visible des objets, alors que la représentation interne des données reste une affaire privée. Ce principe *soulage* l'utilisateur de la responsabilité de connaître les détails de l'implémentation, et il *protège* les objets de toute manipulation nuisible.

En suivant ce principe, une classe bien construite sera facile à maintenir, à élargir, et à réutiliser dans d'autres contextes. Ces avantages réduisent considérablement le coût du développement et évitent la réécriture inutile. C'est là le succès de la programmation orientée objet. Nous en avons déjà pleinement profité en utilisant la classe `Integer`.

Exercice/P 1.23. Comme vous pouvez le constater, les opérations d'entrée-sortie ne sont pas encore implémentées pour la classe `Rationnel`, version 0.5. La sortie est plutôt évidente, par exemple $\frac{2}{3}$ s'écrit `2/3`, alors que $\frac{2}{1}$ s'écrit `2`. L'entrée est un peu plus délicate, car la syntaxe doit être analysée afin de tenir compte des deux variantes précédentes. Vous trouvez l'implémentation complète dans le fichier `rationnel.cc`. Essayez d'en comprendre les détails.

Remarque 1.24. Les opérateurs d'entrée-sortie ne sont pas des méthodes de la classe, mais des fonctions extérieures. Néanmoins la classe `Rationnel` peut leur accorder un accès direct aux éléments privés en les déclarant `friend`. C'est souvent utile, mais ici on aurait facilement pu l'éviter. Voyez-vous comment ?

Exercice/P 1.25. Notre implémentation modélise la structure de corps mais ne tient pas encore compte de la structure d'ordre. Si vous voulez, vous pouvez ajouter des opérateurs de comparaison. *Indication.* — Détailler d'abord comment comparer deux fractions $\frac{r}{s}$ et $\frac{u}{v}$. En quoi notre convention $s > 0$, $v > 0$ est-elle importante ? Vous pouvez en profiter car la classe garantit, par une gestion intelligente et des restrictions d'accès, que cette convention soit toujours respectée.

Remarque 1.26. Pour l'implémentation de la comparaison et de toute autre fonction, plusieurs variantes sont imaginables : comme méthode de la classe, comme fonction extérieure, ou bien comme fonction déclarée `friend`. Laquelle vous semble la mieux adaptée ici ?

Conversion implicite vs explicite. Il est souvent utile d'avoir une conversion implicite entre différents types — nous devrions dire maintenant : entre différentes classes. En l'occurrence, la conversion correspond à une convention bien connue en mathématiques : on interprète tout nombre entier $a \in \mathbb{Z}$ aussi comme l'élément $\frac{a}{1}$ dans le corps de factions $\mathbb{Q}$.

En C++ il est donc naturel de convertir `int` ou `Integer` en `Rationnel` quand le contexte le demande. A priori ce sont des types bien différents, comment le compilateur saura-t-il effectuer la conversion ? Bien sûr il ne connaît rien des corps de fractions, et pourtant... un seul indice lui suffit : le constructeur à partir d'un `int` ou d'un `Integer` servira aussi bien pour la conversion.

Exemple 1.27. Les instructions suivantes devraient avoir un sens :

```
Rationnel r(2);    // conversion explicite de int en Rationnel
Rationnel s= r*3;  // acceptable car conversion implicite
```

La manière dont nous avons implémenter l'opérateur `*` exige que le premier argument soit l'objet courant, de type `Rationnel`. Si tel est le cas, le deuxième argument peut y être convertit, et le compilateur le fait tacitement. Il interprète donc le calcul précédent comme

```
Rationnel s= r * Rationnel(3);  // ok, conversion explicite
```

Exemple 1.28. À notre grande surprise, la conversion *implicite* ne s'applique plus au calcul suivant :

```
Rationnel t= 3*r;  // impossible sans conversion explicite
```

Voyez-vous pourquoi ? Bien sûr, la conversion *explicite* fonctionne toujours :

```
Rationnel s= Rationnel(3) * r;  // ok, conversion explicite
```

Le comportement serait différent si l'on implémentait l'opérateur `*` non comme une méthode de la classe, mais comme une fonction extérieure :

```
Rationnel operator* ( const Rationnel& a, const Rationnel& b )
{ return Rationnel( a.numerateur() * b.numerateur(),
                    a.denominateur() * b.denominateur() ); };
```

Cette écriture est plus longue, mais maintenant la situation est entièrement symétrique : et `3*r` et `r*3` déclenchent implicitement une conversion via le constructeur `Rationnel(3)`.

Remarque 1.29. Souvent commode, la conversion implicite est parfois indésirable voire dangereuse, car le compilateur pourrait effectuer des conversions non souhaitées par le programmeur. Pour l'éviter, on peut restreindre un constructeur en faisant précéder le mot clé `explicit`. Ainsi il ne sera appliqué qu'à la suite d'un appel explicite, comme par exemple `Rationnel(3)`. Il est en général prudent de déclarer les constructeurs `explicit`. Afin d'améliorer le confort, en peut ensuite écrire des opérateurs mixtes entre `Rationnel` et `int` ou `Integer`, là où ils sont explicitement souhaités.

2. Anneaux effectifs

Nous essayerons d'étendre nos considérations de $\mathbb{Z}$ et $\mathbb{Q}$ à des anneaux plus généraux.

2.1. Que faut-il pour implémenter un anneau ? Soit A un anneau (commutatif unitaire). En C++ on souhaiterait disposer d'un type `A` qui modélise l'anneau A dans le sens suivant :

Représentation et entrée-sortie: Tout d'abord on veut que tout élément de l'anneau A puisse être représenté comme une valeur de type `A`, et que les opérateurs d'entrée `>>` et de sortie `<<` permettent de lire et d'écrire ces valeurs dans une écriture convenable ; ils traduisent alors entre représentation externe et représentation interne.

Constructeurs: On doit être capable de construire au moins les éléments 0 et 1 de notre anneau. Ceci peut se faire par des constructeurs `A(0)` et `A(1)`. Plus généralement, pour un entier `n` on veut que `A(n)` construise l'élément $n \cdot 1_A$ dans A. Afin d'atteindre tous les éléments de A, d'autres constructeurs seront parfois souhaitables.

Comparaison: On voudra utiliser les opérateurs de comparaison `==` et `!=` avec la syntaxe usuelle : ils prennent deux arguments de type `A` et renvoient un résultat de type `bool`. Bien sûr on exige qu'ils correspondent à la comparaison dans l'anneau A. (Les comparaisons `<`, `<=`, `>`, `>=` n'ont un sens naturel que dans un anneau ordonné.)

Affectation: On voudra utiliser l'opérateur d'affectation `=` avec la syntaxe et la sémantique usuelle : il prend deux arguments, une variable de type `A` à gauche et une valeur de type `A` à droite, puis il affecte la valeur à la variable. (C'est une opération assez triviale mais omniprésente car elle gère la copie d'objets pendant l'exécution d'un programme.)

Opérateurs arithmétiques: Les opérateurs arithmétiques `+`, `-`, `*` prennent deux arguments de type `A` et renvoient un résultat de type `A`. Quant à leur sémantique, on exige que le résultat corresponde au résultat de l'opération respective dans A.

Opérateurs mixtes: Les opérateurs `+=`, `-=`, `*=` devraient s'utiliser d'une manière naturelle, de sorte que `a+=b` équivaille à `a=a+b` etc. L'intérêt est une meilleure lisibilité, et dans certains cas une légère optimisation de performance.

Inversibilité et divisibilité: Étant donnés deux éléments $a, b \in A$ on doit souvent répondre aux questions suivantes : Déterminer si a est inversible, et le cas échéant trouver son inverse. Déterminer si a et divisible par b, et le cas échéant trouver c tel que $a = bc$. Déterminer si a et b sont associés, et le cas échéant trouver $u \in A^\times$ tel que $ua = b$.

Exercice/P 2.1. Le code en C++ ci-dessous explicite les méthodes et fonctions requises pour un anneau effectif. Essayez de vous convaincre que nous avons trouvé un modèle convenable sur lequel on pourra baser toutes les constructions ultérieures : lesquelles pouvez-vous envisager ? Ceci est un point important : bien que l'implémentation concrète puisse changer, les interfaces, elles, devraient rester les mêmes. Tout changement ultérieur d'interfaces sera coûteux, car il nécessitera une révision entière des applications déjà écrites.

Programme XII.7 Modèle minimal pour implémenter un anneau `anneau0.cc`

```
1   class Anneau
2   {
3   public :
4     Anneau( int n=0 );                              // constructeur/conversion
5     Anneau( const Anneau& source );                 // constructeur par copie
6     Anneau& operator=  ( int n );                   // affectation/conversion
7     Anneau& operator=  ( const Anneau& a );         // affectation par copie
8     bool    operator== ( const Anneau& a ) const;   // test d'égalité
9     bool    operator!= ( const Anneau& a ) const;   // test de différence
10    Anneau  operator+  ( const Anneau& a ) const;   // addition
11    Anneau& operator+= ( const Anneau& a );         // addition en place
12    Anneau  operator-  () const;                    // opposé
13    Anneau  operator-  ( const Anneau& a ) const;   // soustraction
14    Anneau& operator-= ( const Anneau& a );         // soustraction en place
15    Anneau  operator*  ( const Anneau& a ) const;   // multiplication
16    Anneau& operator*= ( const Anneau& a );         // multiplication en place
17  };
18
19  // Opérateurs d'entrée-sortie
20  ostream& operator<< ( ostream& out, const Anneau& a );
21  istream& operator>> ( istream& in, Anneau& a );
22
23  // Reconnaître les éléments 0 et 1
24  bool    zero       ( const Anneau& a );
25  bool    one        ( const Anneau& a );
26
27  // Déterminer si a est inversible, si oui calculer son inverse
28  bool    inversible ( const Anneau& a );
29  bool    inversible ( const Anneau& a, Anneau& inverse );
30
31  // Déterminer si a est divisible par b, si oui calculer le quotient q
32  bool    divisible  ( const Anneau& a, const Anneau& b );
33  bool    divisible  ( const Anneau& a, const Anneau& b, Anneau& q );
34
35  // Déterminer si a et b sont associés, si oui calculer u tel que ua=b
36  bool    associes   ( const Anneau& a, const Anneau& b );
37  bool    associes   ( const Anneau& a, const Anneau& b, Anneau& u );
38
39  // Choisir un représentant préféré r=ua parmi les éléments associés à a
40  Anneau  assopref   ( const Anneau& a );
41  Anneau  assopref   ( const Anneau& a, Anneau& u );
```

Remarque 2.2. Dans ce modèle on a inclus certaines fonctions non strictement nécessaires mais pratiques, par exemple la reconnaissance des éléments 0 et 1. On pourrait, bien sûr, écrire `a == A(0)` ou `a == A(1)`, mais il y a souvent des méthodes plus efficaces qui évitent la création d'objets auxiliaires `A(0)` et `A(1)`.

Aucune implémentation n'est fournie dans ce modèle. Pour une implémentation concrète il faut remplacer le nom `Anneau` par le nom de la classe à implémenter, puis définir chacune des fonctions déclarées ci-dessus. Ceci est fait pour la classe `Integer` dans le fichier `integer.cc`, puis pour la classe `Rationnel` dans le fichier `rationnel.cc`. Vérifier que ce sont des traductions fidèles des anneaux en question.

2.2. Vers une formulation mathématique. On dit que A est un *anneau effectif* s'il satisfait aux exigences détaillées ci-dessus : on peut représenter chacun de ses éléments dans une structure de données convenable, puis effectuer les opérations de l'anneau dans la représentation choisie. Après réflexion, le fait qu'un anneau soit effectif ne dépend pas du langage de programmation, ni de la machine qui exécutera le programme : c'est l'existence des algorithmes qui compte.

Remarque 2.3. Bien sûr, on sous-entend ici que l'on travaille sur un ordinateur « usuel ». La seule idéalisation que l'on fait habituellement est de supposer que la mémoire vive de notre ordinateur soit toujours suffisamment grande. Il est clair que notre approche pragmatique laisse tous les détails dans le flou.

Afin d'être rigoureux, on devrait à ce point expliciter le modèle de la machine qui stocke les données et exécute les algorithmes. Une approche éprouvée est de définir un ordinateur abstrait, la machine de Turing ou une de ses variantes équivalentes. On explicite ainsi la structure essentielle d'un ordinateur : tout ce que peut calculer un ordinateur « usuel » peut être calculé par une machine de Turing.

L'avantage d'une approche rigoureuse est la possibilité de prouver des énoncés négatifs : on peut montrer que certaines fonctions ne sont pas calculables. Le développement d'une telle *théorie de la calculabilité* est un important et vaste domaine, que nous n'aborderons pas ici.

Exercice/M 2.4. L'anneau $\mathbb{Z}$ des nombres entiers est effectif. Nous en avons déduit que le corps $\mathbb{Q}$ des nombres rationnels est aussi effectif. Le corps $\mathbb{R}$ des nombres réels, par contre, n'est pas effectif. (Le détailler.) Pour un point de vue plus optimiste (certes idéalisé), on consultera avec profit le livre récent de L. Blum, F. Cucker, M. Shub, S. Smale, *Complexity and real computation*, Springer Verlag, New York 1998.

Exercice/M 2.5. En reprenant notre modèle pragmatique, on veut interpréter les valeurs prises par le type `A` comme des éléments de l'anneau A. Ce n'est rien d'autre qu'une application $I\colon \{\text{valeurs du type } \texttt{A}\} \to A$. Tout d'abord on souhaite que le type `A` puisse représenter n'importe quel élément de A, autrement dit, on veut que I soit surjective. Essayez de reformuler précisément toutes nos exigences ci-dessus à l'aide de l'application I. Pourquoi exige-t-on la surjectivité de I mais on n'insiste pas sur l'injectivité ? L'opérateur `==` induit-il une relation d'équivalence ? Dans quel sens peut-on dire que I induit un isomorphisme entre le modèle informatique `A` et l'objet mathématique A ?

Exercice/M 2.6. Comme un cas simple mais déjà très intéressant regardons $d \in \mathbb{Z}$ sans facteur carré et $\mathbb{Z}[\sqrt{d}] = \{x+y\sqrt{d} \mid x,y \in \mathbb{Z}\}$. Montrer que $\mathbb{Z}[\sqrt{d}]$ est un anneau et que $(1,\sqrt{d})$ en est une $\mathbb{Z}$-base. Expliquer pourquoi $\mathbb{Z}[\sqrt{d}]$ est un anneau effectif et esquisser une implémentation. On implémentera l'anneau $\mathbb{Z}[i]$ dans le projet à la fin de ce chapitre, ce qui correspond au cas particulier $d=-1$.

Exercice/M 2.7. En généralisant l'exemple précédent on peut considérer l'anneau $\mathbb{Z}[\theta]$ où $\theta \in \mathbb{C}$ est racine d'un polynôme unitaire $P = X^n + c_{n-1}X^{n-1} + \cdots + c_1X + c_0$ à coefficients entiers. On peut supposer que P est le polynôme minimal de θ. Montrer que $\mathbb{Z}[\theta] = \{\sum_{i=0}^{n-1} z_i\theta^i | z_i \in \mathbb{Z}\}$ est un anneau et que $(1,\theta,\ldots,\theta^{n-1})$ en est une $\mathbb{Z}$-base de $\mathbb{Z}[\theta]$. Est-ce un anneau effectif ?

2.3. Anneaux euclidiens. Rappelons qu'un anneau intègre A est euclidien s'il existe un stathme $\nu\colon A \to \mathbb{N}$ et une division $\delta\colon A \times A^* \to A \times A$, $(a,b) \mapsto (q,r)$ telle que $a = bq + r$ et $\nu(r) < \nu(b)$. Au delà de l'existence nous souhaitons désormais un calcul effectif.

On dit que A est un *anneau euclidien effectif* s'il est effectif dans le sens précédent et que la division δ est effectivement calculable. En C++ nous exigeons l'implémentation suivante :

```
void   eudiv( const Anneau& a, const Anneau& b, Anneau& q, Anneau& r );
Anneau eudiv( const Anneau& a, const Anneau& b );
Anneau eumod( const Anneau& a, const Anneau& b );
```

Ici la fonction `eudiv(a,b,q,r)` implémente la division euclidienne $\delta\colon (a,b) \mapsto (q,r)$. Les deux autres fonctions renvoient q et r séparément, dans le but d'un usage plus commode. Remarquons toutefois que la fonction `eumod` devrait être définie aussi pour $b=0$, dans quel cas elle renvoie a. Par contre `eudiv` n'est définie que pour $b \neq 0$, sinon elle provoque une erreur.

Exemple 2.8. L'anneau $\mathbb{Z}$ est un anneau euclidien effectif. Il en est de même pour $\mathbb{Q}[X]$: comme on verra dans le prochain chapitre, si K est un corps effectif, alors l'anneau $K[X]$ des polynômes sur K est un anneau euclidien effectif. (Esquisser pourquoi.)

Exercice 2.9. Étant donné un anneau euclidien effectif, montrer que les algorithmes d'Euclide et de Bézout sont applicables. En particulier on sait ainsi calculer le pgcd et des coefficients de Bézout pour tout $a,b \in A$. Le programme ci-dessous présente des fonctions génériques. (Pour l'usage des fonctions génériques, ou « patrons de fonctions », voir le chapitre V, §4.)

Programme XII.8 Modèle générique de l'algorithme d'Euclide `euclide0.hh`

```
1   // L'algorithme d'Euclide pour calculer le pgcd de a et b
2   template <typename Anneau>
3   Anneau pgcd( Anneau a, Anneau b )
4   {
5     Anneau r;
6     while( !zero(b) ) { r= eumod(a,b); a= b; b= r; };
7     return assopref(a);
8   }
9
10  // L'algorithme d'Euclide étendu pour calculer pgcd(a,b) = au + bv
11  template <typename Anneau>
12  Anneau pgcd( Anneau a, Anneau b, Anneau& u, Anneau& v )
13  {
14    u= Anneau(1); v= Anneau(0);
15    Anneau x(0), y(1), q, r;
16    while( !zero(b) )
17      {
18        eudiv(a,b,q,r); a= b; b= r;
19        r= u - q * x;   u= x; x= r;
20        r= v - q * y;   v= y; y= r;
21      };
22    a= assopref(a,q); u*= q; v*= q;
23    return a;
24  }
```

2.4. Anneaux principaux. Dans un anneau principal A tout couple d'éléments $a,b \in A$ admet un pgcd et il existe $u,v \in A$ vérifiant l'identité de Bézout : $\text{pgcd}(a,b) = au + bv$.

Au delà de l'existence nous souhaitons un calcul effectif. On dit que A est un *anneau principal effectif* s'il est effectif et admet un algorithme qui calcule pour tout $a,b \in A$ leur pgcd avec des coefficients de Bézout. En C++ nous exigeons donc l'implémentation suivante :

```
Anneau pgcd( const Anneau& a, const Anneau& b, Anneau& u, Anneau& v );
Anneau pgcd( const Anneau& a, const Anneau& b, Anneau& u );
Anneau pgcd( const Anneau& a, const Anneau& b );
```

À nouveau on inclut deux fonctions spécialisées pour un usage commode. La première sert notamment à calculer l'inverse de a modulo b : si $\text{pgcd}(a,b) = 1 = au + bv$ il suffit de connaître u. La deuxième sert à calculer le pgcd sans coefficients de Bézout, ce qui arrive assez souvent. Notez qu'une fonction spécialisée est en général plus efficace, car un calcul restreint peut être optimisé.

Exemple 2.10. Tout anneau euclidien effectif est un anneau principal effectif. En particulier, ceci est le cas pour l'anneau $\mathbb{Z}$ des nombres entiers et l'anneau $K[X]$ des polynômes sur un corps K.

Exercice/M 2.11. Une application importante des anneaux principaux réside dans la résolution des équations linéaires. Expliquez comment résoudre l'équation $a_1x_1 + a_2x_2 = b$ dans un anneau principal effectif : comment déterminer si elle admet de solution ? comment en trouver une ? comment en trouver toutes ? Essayez de formuler un algorithme, puis généralisez-le à l'équation linéaire $a_1x_1 + \cdots + a_nx_n = b$.

Si vous êtes courageux, vous pouvez ensuite regarder un système d'équations linéaires, disons sous forme matricielle, et formuler l'élimination de Gauss sur un anneau principal effectif. Ce n'est pas immédiat, mais tout fonctionnera comme vous l'espérez !

Remarque 2.12. Il existe des anneaux qui sont principaux mais non euclidiens. Il n'est pas évident d'exhiber un tel exemple, mais $\mathbb{Z}[\xi]$ avec $\xi = \frac{1}{2}(1+i\sqrt{19})$ en est un. On vérifie d'abord que ξ a pour polynôme minimal $X^2 - X + 5$; le couple $(1,\xi)$ constitue donc une $\mathbb{Z}$-base de $\mathbb{Z}[\xi]$. Vous pouvez ainsi vous convaincre qu'il s'agit d'un anneau effectif ; son caractère principal mais non euclidien est moins facile à prouver.

2.5. Anneaux factoriels. Étant donné un anneau factoriel A on peut définir le pgcd et le ppcm (rappeler comment). Au delà de la définition abstraite nous souhaitons un calcul effectif. On dit que A est un *anneau à pgcd effectif* s'il est effectif et admet un algorithme pour calculer le pgcd. Pour l'implémentation en C++ on exige une fonction

```
Anneau pgcd( const Anneau& a, const Anneau& b );
```

Exemple 2.13. Tout anneau principal effectif est aussi un anneau à pgcd effectif.

Exemple 2.14. L'anneau $\mathbb{Z}[X]$ n'est pas principal : il suffit de se convaincre que l'idéal $(2,X)$, par exemple, n'est pas principal. Néanmoins, $\mathbb{Z}[X]$ est factoriel et permet un calcul effectif du pgcd.

Remarque 2.15. La factorialité de $\mathbb{Z}[X]$ est un célèbre théorème de Gauss ; le temps venu votre cours d'algèbre vous le présentera. Il affirme plus généralement que $A[X]$ est factoriel si et seulement si A est factoriel. Si vous regarder la preuve sous l'angle algorithmique, vous prouverez en même temps : si A est un anneau intègre à pgcd effectif, alors $A[X]$ est un anneau intègre à pgcd effectif. On trouve ainsi maints exemples d'anneaux factoriels qui ne sont pas euclidiens, ni principaux, mais qui permettent néanmoins de calculer effectivement le pgcd. Pour en savoir plus, on consultera avec profit le développement dans [**9**], § II-8.

Remarque 2.16. L'anneau A étant factoriel, on voudrait certes disposer des fonctions permettant de tester l'irréductibilité d'un élément $a \in A$, et le cas échéant de décomposer a en un produit d'éléments irréductibles. Malheureusement ces deux questions peuvent être assez difficiles. Pour nos besoins modestes on se contentera de calculer le pgcd, ce qui est souvent plus facile.

Exemple 2.17. On a déjà rencontré les difficultés de la factorisation dans l'anneau $\mathbb{Z}$ au chapitre XI. Dans certains anneaux la situation est encore plus compliquée, dans d'autres encore, par exemple $\mathbb{F}_q[X]$ sur un corps fini $\mathbb{F}_q$, la question d'irréductibilité et de factorisation sont beaucoup plus faciles que dans $\mathbb{Z}$. Nous ne poursuivons pas cette approche ici. Pour en savoir plus, vous pouvez consulter [**11**].

2.6. Corps des fractions. Reprenons la construction du corps des fractions. À partir de la classe `Integer` modélisant l'anneau $\mathbb{Z}$ nous avons déjà implémenté la classe `Rationnel` modélisant le corps $\mathbb{Q}$. Nous avons aussi prouvé que cette construction se généralise à n'importe quel anneau intègre.

Précisons pourtant que, pour une implémentation efficace, nous avons besoin du pgcd : on a tout intérêt à réduire systématiquement toute fraction $\frac{r}{s}$ afin d'assurer $\operatorname{pgcd}(r,s) = 1$. (Pourquoi ?) Le programme XII.9 en donne une implémentation générique.

Exercice/P 2.18. Comme vous le constatez, le programme XII.9 n'implémente pas encore d'entrée-sortie ; vous trouverez une implémentation plus complète dans le fichier `fraction.hh`. Vérifier soigneusement cette implémentation et effectuer quelques tests sur la classe `Fraction<Integer>` pour vous convaincre qu'elle modélise bien le corps des fraction $\operatorname{Frac}(\mathbb{Z}) = \mathbb{Q}$. Le programme `fraction.cc` en donne un exemple d'utilisation. Expliquez l'intérêt d'une classe générique `Fraction`. Quel pourrait être l'intérêt d'une classe spécialisée comme `Rationnel` ?

2.7. Anneaux quotients. Étant donné un anneau A implémenté par une classe `A` en C++, nous souhaitons implémenter l'anneau quotient A/I où I est un idéal de A. Ce problème général est trop dur, mais il devient facile quand nous exigeons que A soit un anneau principal effectif : dans ce cas $I = (m)$ pour un certain $m \in A$.

Dans un souci d'efficacité nous allons même supposer que A est un anneau euclidien effectif. Dans ce cas on représentera la classe $a + (m)$ par le reste $r = a \bmod m$ de la division euclidienne dans A. Le principal intérêt du passage au reste et de réduire la taille des données.

Exercice/M 2.19. Expliquer comment représenter les éléments de $A/(m)$ et comment effectuer les opérations d'anneau. Analysez ensuite l'implémentation proposée dans le fichier `quotient.hh` et vérifier soigneusement sa correction. Compléter les fonctions manquantes :

- Déterminer si a est inversible, et le cas échéant trouver son inverse.
- Déterminer si a et divisible par b, et le cas échéant trouver c tel que $a = bc$.
- Déterminer si a et b sont associés, et le cas échéant trouver $u \in A^\times$ tel que $ua = b$.
- Étant donné un élément a, choisir un élément associé préféré $r = ua$ avec $u \in A^\times$.
  (En absence de préférences on prendra $r := a$ et $u := 1$.)

Programme XII.9 Corps des fractions d'un anneau à pgcd effectif `fraction0.hh`

```
template <typename Anneau>
class Fraction
{
private:
  Anneau numer, denom;   // numérateur et dénominateur
  void normaliser(void)  // normaliser numer/denom
    { if ( zero(denom) ) { cerr << "Division par zéro !\n"; exit(1); };
      Anneau d= pgcd(numer,denom);
      divisible(numer,d,numer); divisible(denom,d,denom);
      Anneau u; denom= assopref(denom,u); numer*= u; };

public:
  // Constructeurs divers
  Fraction( int num=0, int den=1 )
    : numer(num), denom(den) { normaliser(); }
  Fraction( const Anneau& num, const Anneau& den )
    : numer(num), denom(den) { normaliser(); };
  Fraction( const Fraction& a )
    : numer(a.numer), denom(a.denom) {};

  // Affectation
  void affecter( int num=0, int den=1 )
    { numer= num; denom= den; normaliser(); };
  void affecter( const Anneau& num, const Anneau& den )
    { numer= num; denom= den; normaliser(); };
  Fraction& operator= ( const Fraction& a )
    { numer= a.numer; denom= a.denom; return *this; };

  // Lire le numérateur et le dénominateur
  const Anneau& numerateur() const { return numer; };
  const Anneau& denominateur() const { return denom; };

  // Comparaison
  bool operator== ( const Fraction& a ) const
    { return ( numer*a.denom == denom*a.numer ); };
  bool operator!= ( const Fraction& a ) const
    { return ( numer*a.denom != denom*a.numer ); };

  // Addition
  Fraction operator+ ( const Fraction& a ) const
    { return Fraction( numer*a.denom + denom*a.numer, denom*a.denom ); };
  Fraction& operator+= ( const Fraction& a )
    { affecter( numer*a.denom + denom*a.numer, denom*a.denom ); return *this; };

  // Opposé et soustraction
  Fraction operator- () const
    { return Fraction( -numer, denom ); };
  Fraction operator- ( const Fraction& a ) const
    { return Fraction( numer*a.denom - denom*a.numer, denom*a.denom ); };
  Fraction& operator-= ( const Fraction& a )
    { affecter( numer*a.denom - denom*a.numer, denom*a.denom ); return *this; };

  // Multiplication (correcte mais non optimisée)
  Fraction operator* ( const Fraction& a ) const
    { return Fraction( numer*a.numer, denom*a.denom ); };
  Fraction& operator*= ( const Fraction& a )
    { affecter( numer*a.numer, denom*a.denom ); return *this; };

  // Division (correcte mais non optimisée)
  Fraction operator/ ( const Fraction& a ) const
    { return Fraction( numer*a.denom, denom*a.numer ); };
  Fraction& operator/= ( const Fraction& a )
    { affecter( numer*a.denom, denom*a.numer ); return *this; };
};
```

Programme XII.10 Quotient d'un anneau euclidien effectif `quotient0.hh`

```
template <typename Anneau>
class Quotient
{
private:
  Anneau mod, rep; // le module et le représentant

public:
  // Constructeurs
  Quotient( const Anneau& r=Anneau(0), const Anneau& m=Anneau(0) )
    : mod( assopref(m) ), rep( eumod( r, mod ) ) {};
  Quotient( const Quotient& a )
    : mod(a.mod), rep(a.rep) {};

  // Affectation
  void affecter( const Anneau& r=0, const Anneau& m=0 )
    { mod= assopref(m); rep= eumod( r, mod ); };
  Quotient& operator= ( const Quotient& a )
    { mod= a.mod; rep= a.rep; return *this; };

  // Lire le module et le représentant
  const Anneau& module() const { return mod; };
  const Anneau& representant() const { return rep; };

  // Comparaison
  bool operator== ( const Quotient& a ) const
    { return associes( mod, a.mod ) && divisible( rep-a.rep, mod ); };
  bool operator!= ( const Quotient& a ) const
    { return !associes( mod, a.mod ) || !divisible( rep-a.rep, mod ); };

  // Addition
  Quotient operator+ ( const Quotient& a ) const
    { return Quotient( rep + a.rep, pgcd( mod, a.mod ) ); };
  Quotient& operator+= ( const Quotient& a )
    { affecter( rep + a.rep, pgcd( mod, a.mod ) ); return *this; };

  // Opposé et soustraction
  Quotient operator- () const
    { return Quotient( -rep, mod ); };
  Quotient operator- ( const Quotient& a ) const
    { return Quotient( rep - a.rep, pgcd( mod, a.mod ) ); };
  Quotient& operator-= ( const Quotient& a )
    { affecter( rep - a.rep, pgcd( mod, a.mod ) ); return *this; };

  // Multiplication
  Quotient operator* ( const Quotient& a ) const
    { return Quotient( rep * a.rep, pgcd( mod, a.mod ) ); };
  Quotient& operator*= ( const Quotient& a )
    { affecter( rep * a.rep, pgcd( mod, a.mod ) ); return *this; };
};
```

Remarque 2.20. La classe `Quotient` stocke tout élément $a+(m)$ du quotient $A/(m)$ par le représentant $r = a \bmod m$. Soulignons à titre d'avertissement que le passage au reste $r = a \bmod m$ n'est en général pas équivalent à la projection $\pi\colon A \to A/(m)$, car on ne peut pas garantir l'unicité du reste de la division euclidienne : il peut y avoir $a \equiv a' \pmod m$ avec $r = a \bmod m$ différent de $r' = a' \bmod m$. Il est donc prudent de tester l'égalité entre $r+(m)$ et $r'+(m)$ non par $r = r'$, mais par la divisibilité $m \mid r - r'$. Le programme XII.10 tient compte de cette subtilité.

Exercice/M 2.21. Montrer pour l'anneau $A = K[X]$ des polynômes sur un corps K, que la projection $\pi\colon A \to A/(m)$ est équivalente au passage au reste $a \mapsto a \bmod m$: ceci est possible parce qu'il n'existe qu'un seul représentant r avec $\deg(r) < \deg(m)$. Détailler un contre-exemple dans $\mathbb{Z}$ en explicitant une division euclidienne convenable, $a = bq + r$ telle que $|r| < |b|$. Discuter en particulier l'opérateur `%` du C++, qui est souvent une source d'erreurs. (Comment peut-on rendre le reste unique dans ce cas concret ?)

Remarque 2.22. Dans l'implémentation précédente, tout objet stocke son propre module `mod` ainsi qu'un représentant `rep` de la classe modulo `mod`. Très souvent on ne calcule que modulo un seul idéal (m) de A, il est donc inutile de stocker la même valeur m pour chacun des objets — quel gaspillage de ressources !

Le programme XII.11 ci-dessous montre comment implémenter un module *commun* pour tous les objets de la classe `Quot<Anneau>`. En C++ un tel élément se dit `static` et se comporte comme une variable globale, seul le nom `Quot<Anneau>::mod` rappelle qu'il appartient à la classe `Quot<Anneau>`. Alors que l'élément `rep` est une variable individuelle de chaque objet, l'élément `mod` n'existe qu'en un seul exemplaire, ce qui est le comportement souhaité. Pour le manipuler on implémente deux fonctions, également déclarées `static`, ce qui permet un accès via la classe, indépendemment de tout objet.

Question 2.23. En quoi est-ce périlleux de changer le module en cours du calcul ? Sous quelle condition les valeurs gardent-elles un sens après changement de module ? (Quand est-ce une opération « bien définie » ?)

Programme XII.11 Quotient d'un anneau (à compléter en exercice) `quot.hh`

```
#include <iostream>
using namespace std;

// Définition d'une classe générique modélisant un anneau quotient
template <typename Anneau>
class Quot
{
private:
  static Anneau mod;  // le module commun de tous les objets de la classe
  Anneau rep;         // le représentant particulier de l'objet courant

public:
  // Lire et redéfinir le module commun
  static const Anneau& module() { return mod; };
  static void module( const Anneau& m ) { mod= assopref(m); };

  // Lire et réduire le représentant modulo mod
  void reduire() { rep= eumod( rep, mod ); };
  const Anneau& representant() const { return rep; };

  // Constructeurs
  Quot( const Quot& a ) : rep( a.rep ) {};
  Quot( const Anneau& r=Anneau(0) ) : rep( eumod( r, mod ) ) {};

  // S'ajoutent ici d'autres méthodes encore à implémenter ...
};

// On définit ensuite la variable statique de la classe Quot<Anneau>
template <typename Anneau>
Anneau Quot<Anneau>::mod(0);

// Exemple d'utilisation
#include "integer.cc"
int main()
{
  Quot<Integer> a(11), b(12);
  cout << "a = " << a.representant() << " mod " << a.module() << endl;
  cout << "b = " << b.representant() << " mod " << b.module() << endl;
  Quot<Integer>::module(5);
  cout << "a = " << a.representant() << " mod " << a.module() << endl;
  cout << "b = " << b.representant() << " mod " << b.module() << endl;
  Quot<Integer> c= a+b;
  cout << "c = " << c.representant() << " mod " << c.module() << endl;
}
```

Exercice/P 2.24. Compléter la classe `Quot<Anneau>` conformément au modèle XII.7.

One Ring to rule them all,
One Ring to find them.
J.R.R. Tolkien, *The Lord of the Rings*

PROJET XII

Entiers de Gauss et sommes de deux carrés

Objectifs

- Étudier l'anneau $\mathbb{Z}[i]$ et expliciter son caractère euclidien.
- En déduire une application classique aux sommes de deux carrés.

Une question classique de la théorie des nombres est la suivante : quels entiers peuvent être écrits comme somme de deux carrés ? Voici un premier constat :

$0 = 0^2 + 0^2$	$1 = 1^2 + 0^2$	$2 = 1^2 + 1^2$	$3 = ?$
$4 = 2^2 + 0^2$	$5 = 2^2 + 1^2$	$6 = ?$	$7 = ?$
$8 = 2^2 + 2^2$	$9 = 3^2 + 0^2$	$10 = 3^2 + 1^2$	$11 = ?$
$12 = ?$	$13 = 3^2 + 2^2$	$14 = ?$	$15 = ?$
$16 = 4^2 + 0^2$	$17 = 4^2 + 1^2$	$18 = 3^2 + 3^2$	$19 = ?$
$20 = 4^2 + 2^2$	$21 = ?$	$22 = ?$	$23 = ?$

Ce projet a pour but de résoudre ce problème. On considère d'abord le point de vue théorique : ici l'anneau $\mathbb{Z}[i] \subset \mathbb{C}$ des entiers de Gauss nous rendra d'excellents services. Ensuite vient l'aspect algorithmique : en sachant que $10^{100} + 949$ admet une unique décomposition en somme de deux carrés, comment la trouver de manière efficace ? C'est le caractère euclidien de $\mathbb{Z}[i]$ qui se révélera un merveilleux outil.

Sommaire

1. Analyse mathématique du problème

Exercice/P 1.1. Écrire une fonction qui prend comme argument un entier n et renvoie tous les couples $(x,y) \in \mathbb{Z}^2$ tels que $x^2 + y^2 = n$ et $x \geq y \geq 0$. Pour l'instant une méthode exhaustive suffira, néanmoins on effectuera l'optimisation évidente moyennant la fonction `sqrt`. Combien d'itérations sont nécessaires ? Est-ce raisonnable pour $10^6 + 33$? pour $10^{12} + 61$? pour $10^{50} + 577$? pour $10^{100} + 949$? Quel est le plus petit n admettant deux telles sommes ? trois ? quatre ? cinq ?

Exercice/M 1.2. L'expérience montre que beaucoup d'entiers s'écrivent comme sommes de deux carrés. On constate aussi que $4k+3$ n'est jamais une somme de deux carrés. Donner une preuve.

Pour simplifier on ne regardera dans la suite que les nombres premiers. Le but de ce projet et de montrer le théorème suivant et en même temps de développer un algorithme efficace.

Théorème 1.3. *Tout nombre premier $p = 4k+1$ s'écrit de manière unique comme somme de deux carrés, $p = x^2 + y^2$ avec $x > y > 0$ dans $\mathbb{Z}$.*

Le développement qui suit consiste en deux parties. La première donne une preuve du théorème en étudiant l'anneau $\mathbb{Z}[i]$ des entiers de Gauss. La deuxième est consacrée à l'implémentation d'une classe `Gauss` et d'une méthode efficace pour trouver la décomposition $p = x^2 + y^2$.

1.1. Unicité. Nous commençons notre étude de la décomposition $p = x^2 + y^2$ par l'unicité : si p est premier, alors $p = x^2 + y^2 = u^2 + v^2$ entraîne $\{x^2, y^2\} = \{u^2, v^2\}$.

Exercice/M 1.4. La norme $N\colon \mathbb{Z}[i] \to \mathbb{N}$ est définie par $N(z) = z\bar{z}$, ou encore $N(x+yi) = x^2 + y^2$ pour $x, y \in \mathbb{Z}$. Montrer que la norme est multiplicative : $N(ab) = N(a)N(b)$ pour tout $a, b \in \mathbb{Z}[i]$. En déduire que $a \in \mathbb{Z}[i]$ est inversible si et seulement si $N(a) = 1$, ce qui équivaut à $a \in \{\pm 1, \pm i\}$. Montrer que a est irréductible dans $\mathbb{Z}[i]$ si $N(a)$ est irréductible dans $\mathbb{Z}$. *Attention.* — la réciproque est fausse.

Exercice/M 1.5. Montrer que $\mathbb{Z}[i]$ est euclidien par rapport à la norme N, donc principal, donc factoriel. *Indication.* — Regarder l'anneau $\mathbb{Z}[i]$ et son corps des fractions $\mathbb{Q}[i]$ dans le plan complexe $\mathbb{C}$. Pour a et $b \neq 0$ dans $\mathbb{Z}[i]$ approcher leur quotient $\frac{a}{b} \in \mathbb{Q}[i]$ de manière optimale par $q \in \mathbb{Z}[i]$. (Faites un dessin !) Conclure que $a = bq + r$ avec un reste r vérifiant $N(r) \leq \frac{1}{2}N(b)$.

Exercice/M 1.6. Pour $p \in \mathbb{N}$ premier montrer que $p = x^2 + y^2 = u^2 + v^2$ avec $x, y, u, v \in \mathbb{Z}$ implique $\{x^2, y^2\} = \{u^2, v^2\}$. *Indication.* — Regarder les décompositions $p = (x+yi)(x-yi) = (u+vi)(u-vi)$.

1.2. Existence. Après l'unicité montrons l'existence d'une décomposition $p = x^2 + y^2$.

Exercice/M 1.7. Soit $p \in \mathbb{N}$ premier. Montrer que -1 admet une racine carrée modulo p si et seulement si p est de la forme $p = 4k+1$ ou $p = 2$. Dans ce cas il existe donc $q \in \mathbb{Z}$ tel que p divise $q^2 + 1$. En déduire que p n'est pas premier dans $\mathbb{Z}[i]$. *Indication.* — regarder le produit $q^2 + 1 = (q+i)(q-i)$ dans $\mathbb{Z}[i]$.

Exercice/M 1.8. En supposant que p est premier dans $\mathbb{Z}$ mais non dans $\mathbb{Z}[i]$, on sait qu'il s'écrit comme $p = ab$ avec deux facteurs $a, b \in \mathbb{Z}[i]$ non inversibles. En déduire que $N(a) = N(b) = p$, puis $\bar{a} = b$, et terminer la preuve du théorème 1.3.

Exercice/M 1.9. Reste la question pratique : comment calculer la décomposition $p = ab$ dans $\mathbb{Z}[i]$? Montrer que, quitte à échanger a et b, on a $a \sim \operatorname{pgcd}(p, q+i)$ et $b \sim \operatorname{pgcd}(p, q-i)$. Expliquer l'intérêt pratique.

1.3. Cas général. On vient de prouver qu'un nombre $p = 4k+1$ qui est premier dans $\mathbb{Z}$ devient réductible dans $\mathbb{Z}[i]$. On peut se poser la question de savoir ce qui se passe avec les autres nombres premiers, ceux de la forme $p = 4k+3$. Plus généralement, quels sont les éléments irréductibles dans $\mathbb{Z}[i]$?

Exercice/M 1.10. Commencez par montrer que les éléments suivants sont irréductibles dans $\mathbb{Z}[i]$:

- $x+yi$ avec $p = x^2 + y^2 \in \mathbb{N}$ un nombre premier de la forme $p = 4k+1$ ou $p = 2$.
- p avec $p \in \mathbb{N}$ un nombre premier de la forme $p = 4k+3$.

À noter que 2 se décompose dans $\mathbb{Z}[i]$ en deux facteurs irréductibles associés, $(1+i)$ et $(1-i)$. Un nombre premier $p = 4k+1$ se décompose dans $\mathbb{Z}[i]$ en deux facteurs irréductibles conjugués, $x+yi$ et $x-yi$, qui ne sont pas associés dans $\mathbb{Z}[i]$. Un nombre premier $p = 4k+3$ reste irréductible dans $\mathbb{Z}[i]$.

Exercice/M 1.11. Montrer que tout élément irréductible dans $\mathbb{Z}[i]$ est associé à un des précédents. *Indication.* — Étant donné $z \in \mathbb{Z}[i]$ décomposer $N(z) = z\bar{z}$ en éléments irréductibles, d'abord dans $\mathbb{Z}$, puis dans $\mathbb{Z}[i]$.

Exercice/M 1.12. Après avoir résolu la question pour les nombres premiers, caractériser les nombres naturels s'écrivant comme somme de deux carrés. Que peut-on dire du nombre de telles décompositions ? Votre résultat correspond-il aux observations empiriques de l'exercice 1.1 ?

2. Implémentation de la classe `Gauss`

Afin de calculer commodément dans $\mathbb{Z}[i]$ nous allons implémenter une classe `Gauss` modélisant cet anneau. Pour faciliter cette tâche, le programme XII.12 ci-dessous déclare les méthodes de base nécessaires ainsi que quelques fonctions supplémentaires.

Exercice/P 2.1. Relire notre modèle d'un anneau effectif (§2) puis implémenter la classe `Gauss` comme déclarée dans le programme XII.12. Vous pouvez prendre les fichiers `integer.cc` et `rationnel.cc` comme modèle, mais veillez à implémenter correctement les opérations de $\mathbb{Z}[i]$. *Remarque.* — La fonction `assopref` sert à rendre le pgcd unique. Pour les entiers de Gauss il n'y a pas de choix canonique. Pour un élément non nul on pourrait choisir l'associé $x+yi$ avec $x > 0$ et $y \geq 0$.

Conseil. — Comme toujours, testez puis relisez soigneusement votre implémentation. Pour les opérations de base choisissez des méthodes simples mais efficaces. Par exemple, tester si $x+yi$ est inversible dans $\mathbb{Z}[i]$ en calculant $x^2 + y^2$ est inefficace pour x, y grands ; essayez de faire mieux.

Programme XII.12 Déclaration de la classe `Gauss` (à compléter en exercice) `gauss.cc`

```
#include <iostream>
#include "integer.cc"
using namespace std;

// La classe Gauss modélisant l'anneau Z[i]
class Gauss
{
public:
  Integer re, im;
  Gauss( const Integer& x=0, const Integer& y=0 );
  Gauss( const Gauss& a );
  Gauss& operator=  ( const Gauss& a );
  bool   operator== ( const Gauss& a ) const;
  bool   operator!= ( const Gauss& a ) const;
  Gauss  operator+  ( const Gauss& a ) const;
  Gauss& operator+= ( const Gauss& a );
  Gauss  operator-  () const;
  Gauss  operator-  ( const Gauss& a ) const;
  Gauss& operator-= ( const Gauss& a );
  Gauss  operator*  ( const Gauss& a ) const;
  Gauss& operator*= ( const Gauss& a );
};

// Opérateurs d'entrée-sortie
ostream& operator<< ( ostream& out, const Gauss& a );
istream& operator>> ( istream& in, Gauss& a );

// Quelques fonctions supplémentaires
bool    zero ( const Gauss& a );
bool    one  ( const Gauss& a );
Gauss   conj ( const Gauss& a );
Integer norme( const Gauss& a );

// Inversibilité, divisibilité, éléments associés
bool  inversible( const Gauss& a );
bool  inversible( const Gauss& a, Gauss& b );
bool  divisible ( const Gauss& a, const Gauss& b );
bool  divisible ( const Gauss& a, const Gauss& b, Gauss& q );
bool  associes  ( const Gauss& a, const Gauss& b );
bool  associes  ( const Gauss& a, const Gauss& b, Gauss& u );
Gauss assopref  ( const Gauss& a );
Gauss assopref  ( const Gauss& a, Gauss& u );

// Une division euclidienne adaptée à la norme
void  eudiv( const Gauss& a, const Gauss& b, Gauss& q, Gauss& r );
Gauss eudiv( const Gauss& a, const Gauss& b );
Gauss eumod( const Gauss& a, const Gauss& b );

// L'algorithme d'Euclide-Bézout
Gauss pgcd( Gauss a, Gauss b, Gauss& u, Gauss& v );
Gauss pgcd( Gauss a, Gauss b, Gauss& u );
Gauss pgcd( Gauss a, Gauss b );
```

Exercice/P 2.2. Votre classe `Gauss` devrait se prêter aisément à la construction du corps des fractions $\mathrm{Frac}(\mathbb{Z}[i]) \cong \mathbb{Q}[i]$. Écrire un petit programme qui teste la classe `Fraction<Gauss>`. Votre classe `Gauss` est-elle conforme aux exigences ? La classe `Fraction<Gauss>` fonctionne-t-elle comme prévu ?

Exercice/P 2.3. Écrire un petit programme pour tester la classe `Quotient<Gauss>` ou `Quot<Gauss>` qui modélise les anneaux quotients de $\mathbb{Z}[i]$. Le quotient $Q = \mathbb{Z}[i]/(3)$ est-il un corps ? Quel est son cardinal ? Trouver un élément d'ordre 8 dans $Q^\times$. Généraliser cette approche à un nombre premier $p = 4k+3$ quelconque, puis écrire un programme qui trouve un élément d'ordre p^2-1 dans $(\mathbb{Z}[i]/(p))^\times$.

3. Décomposition en somme de deux carrés

Pour mener ce projet à bonne fin, il nous faut encore une méthode pour trouver $q \in \mathbb{Z}$ tel que p divise q^2+1. Ce problème a été résolu au chapitre X, §2.1.

Exercice 3.1. Écrire un programme qui lit un premier $p \equiv 1 \pmod 4$ au clavier, puis trouve $q \in \mathbb{Z}$ vérifiant $p \mid q^2+1$ et calcule $a = \operatorname{pgcd}(p, q+i)$ dans $\mathbb{Z}[i]$. En déduire la décomposition cherchée $p = x^2+y^2$. Ajouter une vérification du résultat et le tester sur suffisamment de petits exemples.

Exemple. — Trouver la décomposition en somme de deux carrés de $p = 1009$, puis 10^6+33 et 10^9+9 et $10^{12}+61$. Comparez les résultats et la vitesse de ce programme avec celui de l'exercice 1.1. Ajuster le programme pour que l'on puisse entrer p sous la forme $p = 10^e+k$. Le tester sur les nombres $10^{50}+577$ et $10^{100}+949$ et $10^{200}+357$ et $10^{500}+961$. Que peut-on dire de la performance ? Ces résultats auraient-ils été accessibles avec une recherche exhaustive ?

Exercice 3.2. On sait maintenant traiter efficacement les nombres premiers $p \in \mathbb{Z}$. Esquisser comment implémenter le cas général afin de décomposer un entier quelconque $n \in \mathbb{Z}$ en sommes de deux carrés.

4. Une preuve d'existence non constructive

Nous redémontrons ici le théorème 1.3 de manière *élémentaire*, en développant une preuve élégante due à Don Zagier. Sa note *A one-sentence proof that every prime* $p \equiv 1 \pmod 4$ *is a sum of two squares* peut être admirée dans American Mathematical Monthly 77 (1990), page 144. Dans un premier temps le but sera de comprendre la preuve ; ensuite on comprendra peut-être mieux la différence entre une preuve d'existence et une preuve constructive.

L'idée. On considère l'ensemble $S = \{(x,y,z) \in \mathbb{N}^3 \mid x^2+4yz = p\}$ avec l'involution $\tau\colon S \to S$ donnée par $\tau(x,y,z) = (x,z,y)$. L'idée est simple et géniale : tout point fixe de τ est de la forme (x,y,y) et nous fournit une solution $x^2+(2y)^2 = p$ comme souhaité. Pour montrer qu'un tel point existe il suffit de montrer que le cardinal de S est impair. (Pourquoi ?)

La preuve. On vérifie d'abord que S est un ensemble fini. On le partitionne en trois parties S_1, S_2, S_3 sur lesquelles on définit des applications affines $\sigma_i\colon S_i \to \mathbb{N}^3$ comme suit :

$$\begin{aligned}
S_1 &= \{(x,y,z) \in S \mid x < y-z\}, & \sigma_1(x,y,z) &= (x+2z, z, y-x-z)\\
S_2 &= \{(x,y,z) \in S \mid y-z < x < 2y\}, & \sigma_2(x,y,z) &= (2y-x, y, x-y+z)\\
S_3 &= \{(x,y,z) \in S \mid 2y < x\}, & \sigma_3(x,y,z) &= (x-2y, x-y+z, y)
\end{aligned}$$

Ensuite on vérifie que $\sigma_1(S_1) \subset S_3$ et $\sigma_3(S_3) \subset S_1$, ainsi que $\sigma_2(S_2) \subset S_2$. Les applications $\sigma_1\colon S_1 \to S_3$ et $\sigma_3\colon S_3 \to S_1$ sont inverses l'une à l'autre, et l'application $\sigma_2\colon S_2 \to S_2$ vérifie $\sigma_2^2 = \mathrm{id}$. Finalement, on vérifie que $S = S_1 \cup S_2 \cup S_3$ est une réunion disjointe, comme souhaité. En mettant tout ensemble, on obtient ainsi une involution $\sigma\colon S \to S$ définie par morceaux : $\sigma(x,y,z) = \sigma_i(x,y,z)$ pour $(x,y,z) \in S_i$.

La conclusion. Forcément tout point fixe $(x,y,z) = \sigma(x,y,z)$ est dans S_2. Dans ce cas $(x,y,z) = (2y-x, y, x-y+z)$ implique $x = y$. Par définition de S, un tel point (x,x,z) vérifie $x(x+4z) = p$, ce qui implique $x = 1$ et $z = \frac{p-1}{4}$. Autrement dit $(1,1,\frac{p-1}{4})$ est l'unique point fixe de σ. Comme σ est une involution, le cardinal $|S|$ doit être impair. On conclut que l'involution τ, elle aussi, admet un point fixe. Il existe donc $x,y \in \mathbb{N}$ tels que $x^2+(2y)^2 = p$.

Question 4.1. Nous avons vu deux démonstrations différentes du théorème $p = x^2+y^2$. En quoi la preuve de Zagier est-elle plus élémentaire ou plus élégante que la preuve développée dans le projet ci-dessus ? La preuve de Zagier nous indique-t-elle comment trouver (x,y) effectivement ? Résumer les avantages et les inconvénients des deux approches.

Remarque 4.2. La technique de prouver l'existence d'une solution par un argument de point fixe est fréquemment utilisée en mathématique, voir par exemple le théorème de point fixe de Banch et ses nombreuses applications. (Voir par exemple le chapitre XVII pour le calcul numérique.) Souvent on se contente de l'existence, et ainsi la construction ou la recherche effective ne font pas toujours partie du théorème. Par exemple le théorème de point fixe de Brouwer dit que toute application continue $f\colon [0,1]^n \to [0,1]^n$ admet au moins un point fixe, sans indiquer sa localisation. Quels sont les avantages et les inconvénients de ces deux théorèmes ?

Partie E

Méthodes numériques élémentaires

Give a digital computer a problem in arithmetic, and it will grind away methodically, tirelessly, at gigahertz speed, until ultimately it produces the wrong answer. (...) The problem is simply that computers are discrete and finite machines, and they cannot cope with some of the continuous and infinite aspects of mathematics.
Brian Hayes, *A Lucid Interval*

CHAPITRE XV

Calcul arrondi

Objectif. Dans ce chapitre notre but est de nous familiariser avec le calcul arrondi. Notamment nous voulons comprendre et illustrer ses avantages et ses pièges :

- Le calcul exact n'est pas toujours possible ; même quand il est possible, il n'est pas toujours efficace. Ainsi l'arrondi permet de rendre certains calculs faisables, ou bien plus efficaces sur ordinateur.
- En général le résultat d'un calcul arrondi sera erroné. Il est donc indispensable de connaître l'erreur commise, au moins de la majorer convenablement. La prudence s'impose !
- Sans aucun contrôle de la marge d'erreur la valeur numérique calculée n'apporte *aucune* information sur la valeur exacte cherchée. (J'insiste : *aucune*.)

D'où vient le problème ? Dans toute l'analyse mathématique le corps $\mathbb{R}$ des nombres réels joue un rôle primordial. Malheureusement l'implémentation de tels calculs sur ordinateur pose de sérieux problèmes. Tandis que les calculs avec les nombres entiers $\mathbb{Z}$ ou rationnels $\mathbb{Q}$ peuvent être effectués de manière exacte sur ordinateur, ceci est impossible pour les nombres réels. C'est cette difficulté qui distingue le calcul algébrique (exact) du calcul numérique (approché). La raison est simple :

Limitation dans l'espace: Comme tout objet doit être représenté par une suite *finie* de bits 0 et 1, il est impossible de représenter tout nombre réel de manière exacte sur ordinateur.

Limitation dans le temps: Les constructions en analyse utilisent la notion de limite « $u_n \to u$ pour $n \to \infty$ », alors que sur ordinateur on ne peut effectuer qu'un nombre *fini* d'opérations.

Si ces restrictions sont assez évidentes, les solutions possibles le sont beaucoup moins.

Que peut-on faire ? Dans les applications scientifiques ou technologiques on est souvent obligé de modéliser des calculs dans $\mathbb{R}$ sur ordinateur, malgré tout. Pour cela on utilise typiquement les *nombres à virgule flottante*, qui ne forment qu'un certain sous-ensemble fini $R \subset \mathbb{R}$ des réels. Ces nombres sont bien adaptés à l'ordinateur, mais se comportent assez différemment de ce que vous connaissez des mathématiques. Pour cette raison on les appelle aussi *nombres machine*. Le choix de R n'est pas du tout naturel, il n'a aucune signification mathématique, et il ne suit que des considérations pragmatiques.

Peut-on néanmoins en déduire des informations sur le résultat réel cherché ? On verra qu'au moins dans des circonstances favorables la réponse est « oui », heureusement. Cet espoir explique l'usage fréquent du calcul numérique, mais il existe aussi de mauvais usages — qu'il faut éviter à tout prix.

Pourquoi de petites erreurs sont-elles embêtantes ? Parce qu'elles ne restent pas toujours petites ! Approcher les nombres réels par des nombres machine introduit des *erreurs d'arrondi*. C'est un problème inhérent et omniprésent du calcul numérique. Ces erreurs peuvent se propager dans les calculs, elles peuvent s'accumuler, voire exploser, ce qui peut rendre le résultat calculé inutilisable. C'est d'autant plus dangereux que l'utilisateur n'aura souvent aucune indication sur l'erreur commise et se fiera aveuglement à un résultat grossièrement faux ! Ceci arrive plus fréquemment que l'on ne pense. L'expérience montre qu'un utilisateur typique surestime systématiquement la précision des calculs numériques.

Comment s'y prendre ? Le bon sens et une certaine méfiance éclairée sont indispensables pour tout utilisateur, et ne serait-ce que pour interpréter avec prudence les résultats crachés par une application toute faite. Afin d'expliquer les quelques règles de bon sens, nous consacrons ce chapitre entier aux nombres à virgule flottante et leur calcul arrondi. Pour les raisons évoquées, le calcul arrondi est peu intuitif et l'usage des nombres flottants est assez délicat. Utilisés imprudemment, mêmes les calculs numériques les plus simples peuvent être grossièrement erronés. Pour vous en convaincre, ce chapitre vous présente de nombreux exemples, certes simplifiés mais assez réalistes. Il ne faut pas en conclure que le calcul

numérique soit inutile et se détourner avec mépris. Au contraire, il faut comprendre les fondements afin de calculer intelligemment avec ces nombres, puis interpréter correctement des résultats numériques.

Que serait donc un usage réaliste ? On rencontre souvent deux positions extrêmes :
– Le débutant imagine, à tort, que tout calcul numérique effectué sur ordinateur est correct, et qu'il peut s'y fier aveuglement. On va voir que ce point de vue naïf est *trop optimiste*.
– Après avoir vécu un certain nombre de mauvaises surprises, notre débutant résigne. Il pense maintenant, également à tort, que tout calcul numérique est grossièrement erroné, et qu'on ne peut jamais rien en conclure avec certitude. Ce point de vue est *trop pessimiste*.

Soyez assurés : le calcul numérique peut mener à des conclusions transparentes et prouvables. Mais comme tout outil informatique il nécessite un certain apprentissage. Notre objectif sera donc de développer un usage *réaliste*. Au prochain chapitre nous discuterons les éléments d'un *calcul fiable*, thème qui fait actuellement l'objet d'intenses recherches et qui sert déjà bien dans la pratique.

Peut-on augmenter la précision ? En C++ on dispose des types `float`, `double` et `long double` pour les nombres à virgule flottante. Leur précision est fixée à 24, 53 et 64 bits, respectivement, ce qui correspond à 8, 16 et 20 décimales environ. C'est suffisant pour beaucoup d'applications. Si jamais il vous faut plus de précision vous pouvez faire appel à une bibliothèque spécialisée, comme la *GNU multiple precision library* (GMP), qui implémente des nombres flottants en précision arbitraire (mais toujours finie).

Augmenter la précision n'est pourtant pas le remède à tous les maux numériques : d'une part cela alourdit les calculs, d'autre part il ne vous protège pas de possibles explosions d'erreurs. Le principe du calcul arrondi restera le même : bien que les erreurs d'arrondi soient plus petites, elles sont toujours non nulles, elles se propagent et parfois explosent, et toutes les difficultés mentionnées persisteront. On ne peut donc pas échapper aux problèmes fondamentaux de l'arithmétique arrondie.

Pour en savoir plus. On n'expliquera dans ce chapitre que l'idée essentielle des nombres flottants. Pour ne laisser rien à l'interprétation, le comportement des nombres machine a été standardisé en 1985, sous la norme « IEEE-754 : Standard for Binary Floating-Point Arithmetic ». Cet effort fut fondamental pour abolir l'anarchie qui régnait dans les divers implémentations et pour que le calcul numérique se comporte toujours de manière identique sur deux machines différentes. Littérature :

(1) Pour la petite histoire, on lira avec profit les mémoires de W. Kahan, un des initiateurs du standard IEEE-754 : `www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`.

(2) Pour avoir une idée du standard actuel et futur (en cours de développement) vous pouvez consulter le site `grouper.ieee.org/groups/754/` et les liens y indiqués.

(3) Vous trouvez une excellente introduction à l'arithmétique flottante dans le cours de V. Lefèvre et P. Zimmermann, `www.vinc17.org/research/papers/arithflottante.pdf`.

(4) Vous pouvez également consulter les ouvrages de votre bibliothèque, par exemple le livre de J.-C. Bajard et J.-M. Muller, *Calcul et arithmétique des ordinateurs*, Lavoisier, Paris, 2004.

Sommaire

1. Motivation — quelques applications exemplaires du calcul numérique

Nous commençons par esquisser quelques applications assez représentatives du calcul numérique. Vous pouvez ainsi directement vous lancez dans la programmation, si cela vous intéresse et que vous en avez le loisir. Les chapitres suivants vous expliqueront quelques techniques de base pour analyser et améliorer de telles implémentations. Par conséquent, même si vous ne les programmez pas maintenant, gardez ces problèmes exemplaires en tête afin d'y revenir plus tard.

1.1. Fonctions usuelles. En mathématiques et dans de nombreuses applications vous utilisez fréquemment des fonctions « usuelles ». Si vous avez eu de la chance, quelques unes ont été construites et développées pour vous dans un cours d'analyse. Ensuite, pour les utiliser dans des calculs numériques, il faut encore les implémenter sur ordinateur. Comme toujours dans la programmation, le principal problème sera de le faire *correctement* et *efficacement*. On pourrait, bien sûr, se servir d'une bibliothèque de telles fonctions toute faite, mais essayons de voir si nous y arrivons nous-mêmes…

Exemple 1.1 (repris au §3)**.** Écrire une fonction `sqrt(x)` qui calcule $\sqrt{x}$ à partir d'un nombre réel $x > 0$, en n'utilisant que les quatre opérations $+$, $-$, $*$, $/$. Documentez, vérifiez, puis testez soigneusement votre implémentation. Si vous voulez vous pouvez ensuite généraliser à une fonction `racine(x,n)` qui calcule $\sqrt[n]{x}$ pour $x > 0$ et $n \in \mathbb{N}$. Pouvez-vous majorer la marge d'erreur de vos résultats numériques ?

Exemple 1.2 (repris au §5.5)**.** Écrire une fonction `exp(x)` qui calcule e^x à partir d'un nombre réel $x \in \mathbb{R}$, en n'utilisant que les quatre opérations $+$, $-$, $*$, $/$. Documentez, vérifiez, puis testez soigneusement votre implémentation. Si vous voulez vous pouvez ensuite réfléchir sur `log(x)`, `sin(x)`, `cos(x)`, … ou vos fonctions préférées. Pouvez-vous majorer la marge d'erreur de vos résultats numériques ?

☞ *Ces notes ne peuvent être qu'une modeste invitation à la programmation numérique, en privilégiant des expériences pratiques. Elles partent du principe que vous disposez des bases requises en analyse et que vous les révisez en parallèle. On fera des rappels étape par étape, certes, mais en aucun cas ceux-ci ne peuvent remplacer un solide cours d'analyse.* ✍

1.2. Résolution d'équations. Les problèmes suivants sont classiques et seront discutés dans la suite.

Exemple 1.3 (repris au §5.8)**.** Écrire un programme pour résoudre une équation quadratique $ax^2+bx+c = 0$. Le résultat calculé est-il raisonnablement proche de la solution exacte ? Comment le vérifier ? Votre programme sait-il traiter tous les cas ? Dans quels cas l'erreur devient-elle inacceptable ?

Exemple 1.4 (repris au chapitre XVII)**.** Écrire un programme pour résoudre une équation polynomiale de la forme $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = 0$. Alors qu'en degré $n \leq 4$ il existe des « formules closes », ceci n'est plus le cas en degré $n \geq 5$. Y-a-t il toujours des solutions ? Combien ? Comment les trouver ? Une fois trouvées, comment vérifier la qualité des solutions approchées ?

Exemple 1.5 (repris au chapitre XVIII)**.** Comment résoudre une équation linéaire $Ax = b$ où A est une matrice, b est un vecteur donné, et le vecteur x est inconnu ? Y a-t-il toujours une solution ? Est-elle unique ? Si oui, comment la trouver ? Une fois trouvée, comment vérifier la qualité d'une solution approchée ?

1.3. Intégration numérique. Assez souvent, quand le calcul exact est impossible ou trop dur, on veut numériquement calculer une intégrale de la forme $I = \int_a^b f(x)dx$, par exemple $\int_0^1 e^{-x^2/2}dx$. Dans ce cas, au lieu de l'intégrale exacte, on pourra regarder l'approximation par la $n$ième somme de Riemann

$$I_n := I_n(f,a,b) = \sum_{k=1}^{n} f(x_k)\Delta x \qquad \text{avec} \quad \Delta x = \frac{b-a}{n} \quad \text{et} \quad x_k = a + k\Delta x$$

Théorème 1.6. *Si $f\colon [a,b] \to \mathbb{R}$ est continue, alors $I_n \to I = \int_a^b f(x)dx$ pour $n \to \infty$.*

Exercice/M 1.7. Faire un dessin pour motiver l'approximation de l'intégrale I par la somme I_n. En passant vous êtes invités à réviser le résultat précédent dans votre cours d'analyse. Dans notre exemple, f est monotone ; donner un encadrement explicite de I en fonction de I_n qui prouve que $I_n \to I$.

Exemple 1.8. Écrire une fonction `riemann(a,b,n)` qui calcule I_n pour une fonction $f\colon [a,b]\to\mathbb{R}$ fixée. Documentez, vérifiez, puis testez soigneusement votre implémentation. Pouvez-vous majorer la marge d'erreur de vos résultats numériques ? Qu'obtenez vous dans notre exemple $\int_0^1 e^{-x^2/2}dx$? Quand vous faites augmenter $n=2,4,8,16,32,\dots$ est-ce que les approximations successives se comportent comme prévues ? Quels problèmes constatez-vous ? Mettent-ils en question le théorème ? Ou plutôt nos méthodes de calculs ? Essayez de décrire les problèmes aussi précisément que possible, puis esquisser quelques solutions ou pistes qui vous semblent prometteuses.

1.4. Systèmes dynamiques. Comme vous savez peut-être, le mouvement de deux corps sous la force gravitationelle peut être décrit par une jolie formule close. Pour un nombre $n\geq 3$ de corps, par contre, un tel traitement est en général impossible et il faut faire appel à des méthodes numériques. Si cela vous intéresse, nous esquissons ici les ingrédients d'une telle implémentation assez simple.

Beaucoup de systèmes en physique, chimie, biologie, climatologie, etc. sont modélisés d'une manière très similaire. Historiquement, le mouvement planétaire fut un des premiers exemples à être étudié, et reste à ce jour un sujet classique que tout étudiant en sciences devrait connaître. Bien sûr vous pouvez le remplacer par un autre qui vous est plus proche.

Un modèle de la mécanique céleste. — On considère n corps, numérotés $i=1,\dots,n$, chacun d'une certaine masse constante $m_i>0$. On s'intéresse à la position $x_i=x_i(t)\in\mathbb{R}^3$ et la vitesse $v_i=v_i(t)\in\mathbb{R}^3$ au cours du temps $t\geq 0$. Au temps $t=0$ les positions initiales $x_i(0)=x_i^0$ et les vitesses $v_i(0)=v_i^0$ sont données, puis elles évoluent suivant la mécanique newtonienne : on a $x_i'(t)=v_i(t)$, puis la vitesse $v_i(t)$ change sous l'influence gravitationelle des autres corps. Plus précisément, la force gravitationelle sur le corps i est donnée par $F_i=\sum_{j\neq i}\gamma m_i m_j\frac{(x_j-x_i)}{|x_j-x_i|^3}$, avec une certaine constante universelle γ, et on a $m_i v_i'=F_i$.

Le système à résoudre. — Calculer les trajectoires de n planètes, c'est donc résoudre le système d'équations différentielles $x_i'=v_i$ et $v_i'=F_i/m_i$ avec les conditions initiales $x_i(0)=x_i^0$ et $v_i(0)=v_i^0$. Évidemment on va supposer $x_i\neq x_j$ pour $i\neq j$ et tout $t\geq 0$, sinon notre modèle va tout droit dans la catastrophe. Si jamais un tel problème se produira lors des calculs, on abandonnera en expliquant brièvement ce qui s'est passé.

Discrétisation du temps. — Le modèle continu ne peut pas s'implémenter directement sur ordinateur. Nous allons donc discrétiser le temps t en intervalles d'une longueur fixée Δt. Autrement dit, au lieu d'un temps continue $t\in\mathbb{R}_+$ nous regardons un temps discret $t_k=k\Delta t$ pour $k=0,1,2,3,\dots$.

Approximation numérique. — Une fois discrétisé, le système ci-dessus se transforme en une simple formule de récurrence : nous avons $x_i'\approx\frac{x_i(t_{k+1})-x_i(t_k)}{\Delta t}$, ce qui nous mène à $x_i(t_{k+1})\approx x_i(t_k)+v_i(t_k)\Delta t$. De même $v_i'\approx\frac{\Delta v_i}{\Delta t}$ nous mène à $v_i(t_{k+1})\approx v_i(t_k)+\Delta t F_i(t_k)/m_i$. Nous obtenons ainsi à nouveau l'approximation d'une intégrale :

$$x_i(t_{k+1})=x_i(t_k)+v_i(t_k)\Delta t\quad\text{et}\quad v_i(t_{k+1})=v_i(t_k)+\sum_{j\neq i}\gamma m_j\frac{x_j-x_i}{|x_j-x_i|^3}\Delta t.$$

L'espoir tacite de stabilité. — Bien entendu notre reformulation numérique s'est éloignée du système exact, mais nous pouvons espérer que les trajectoires calculées et les trajectoires exactes se ressemblent. Pour le moment nul ne nous garantit que cet espoir soit bien fondé. Les résultats numériques seront donc à vérifier et à interpréter avec la plus grande prudence !

Exemple 1.9. Si cela vous intéresse vous pouvez implémenter l'approximation numérique esquissée ci-dessus. (Pour faire joli il serait souhaitable d'ajouter un affichage graphique…) Documentez, vérifiez, puis testez soigneusement votre implémentation. Expérimenter avec le choix du paramètre Δt. Est-ce que la trajectoire calculée en dépend ? Quels sont les avantages et les inconvénients de choisir Δt plus petit ou plus grand ? Voyez-vous des situations où le modèle numérique se comporte raisonnablement ? Pouvez-vous concevoir des situations où le calcul numérique échoue ? Quelles en sont les causes possibles ?

☞ *Invariants :* Pour vérification automatique par votre logiciel, vous pouvez faire calculer l'énergie et la quantité du mouvement totales. Dans le modèle exacte ces quantités restent constantes au cours du temps, et une bonne approximation doit faire pareil. Il peut y avoir d'autres invariants. À noter aussi que ce n'est qu'une exigence minimale, et non une garantie de correction.

2. Le problème de la stabilité numérique

2.1. Un exemple simple : les suites de Fibonacci. Comme premier exemple, simple et concret, nous allons regarder des suites de Fibonacci : on se donne deux valeurs initiales $x_0, x_1 \in \mathbb{R}$ puis on procède par la récurrence $x_{n+2} = x_{n+1} + x_n$. Pour un système dynamique on ne peut plus simple !

Exemple 2.1. Pour $x_0 = 1$ et $x_1 = 1$ on obtient la célèbre suite de Fibonacci $x_2 = 2$, $x_3 = 3$, $x_4 = 5$, $x_5 = 8$, etc. Regardons ce qui se passe si l'on varie légèrement les conditions initiales, disons $x'_1 = 1.01$, ce qui est une déviation de 1%. Suivant la récurrence on trouve :

n	0	1	2	3	4	5	6	7	8	9	10
x_n	1.00	1.00	2.00	3.00	5.00	8.00	13.00	21.00	34.00	55.00	89.00
x'_n	1.00	1.01	2.01	3.02	5.03	8.05	13.08	21.13	34.21	55.34	89.55

On constate qu'ici l'erreur initiale se propage d'une manière bien contrôlée : dans toute la suite les valeurs x_n et x'_n ne diffèrent que de 1%. Vous pouvez tester cette observation sur d'autres exemples à l'aide du programme `fibonacci.cc`.

Définition 2.2 (stabilité, formulation heuristique)**.** On dit qu'un calcul numérique est *stable* lors qu'un petit changement des données initiales n'entraîne qu'un petit changement des résultats.

Exemple 2.3. Malheureusement pas tous les calculs se réjouissent de cette bonne propriété de stabilité. Même la récurrence de Fibonacci peut être numériquement méchante. Par exemple, on constate un phénomène étrange pour $x_0 = 1$ et $x_1 \approx -0.618$:

n	0	1	2	3	4	5	6	7	8	9	10
x_n	1.000	−0.618	0.382	−0.236	0.146	−0.090	0.056	−0.034	0.022	−0.012	0.010
x'_n	1.000	−0.619	0.381	−0.238	0.143	−0.095	0.048	−0.047	0.001	−0.046	−0.045

Dans la suite on voit que l'erreur croît de plus en plus rapidement : on trouve $x_{20} = 0.230$ et $x'_{20} = -6.535$, puis $x_{30} = 28.280$ et $x'_{30} = -803.760$. Ceci veut dire qu'une petite erreur initiale (moins de 1%) peut se propager et s'amplifier, et finalement entraîner une erreur considérable au cours de quelques itérations.

Remarque 2.4. Souvent les valeurs avec lesquelles on calcule ne sont pas exactes :

- Si les valeurs initiales sont issues d'un calcul arrondi, elles sont en général déjà contaminées d'erreurs d'arrondi. Le mieux que l'on puisse espérer est que le calcul précédent permet explicitement de garantir une certaine précision.
- Si les données initiales sont des quantités réelles mesurées (par une expérience physique ou autre), elles ne peuvent non plus être exactes. Dans ce cas il est nécessaire de spécifier la précision, disons sous forme d'un « intervalle de confiance ».
- Même dans les rares cas où les données initiales sont exactes, les calculs suivants introduiront des erreurs d'arrondi. On n'échappera donc pas à des perturbations des données, qui font que les calculs peuvent s'éloigner des résultats exacts.

Dans une telle situation *numériquement instable* un calcul ne sert à rien : la moindre erreur peut exploser au cours du calcul, de sorte que le résultat calculé n'apporte plus aucune information.

☞ *Des calculs numériques sans contrôle d'erreur sont toujours périlleux. Dans une situation instable ils tournent à la catastrophe car la moindre perturbation peut s'amplifier au cours du calcul. Des résultats ainsi calculés sont aussi fiables qu'un tirage au sort.* ✍

2.2. Analyse mathématique du phénomène. À titre d'illustration, analysons une question naturelle :

Exemple 2.5. Que peut-on dire de la convergence ou divergence d'une suite de Fibonacci en fonction des valeurs initiales x_0 et x_1 ? En fixant $x_0 = 1$, est-il possible de choisir x_1 de sorte que l'on obtienne une suite convergente ? Si oui, quelle sera la limite ?

- Pour $x_0 = 1$ et $x_1 = 1$ on trouve $x_n \to +\infty$. (Pourquoi ?)
- Pour $x_0 = 1$ et $x_1 = -1$ on trouve $x_n \to -\infty$. (Le vérifier.)

Vue les valeurs numériques ci-dessus, on peut soupçonner que pour $x_1 \geq -0.618$ la suite s'échappe vers $+\infty$, alors que pour $x_1 \leq -0.619$ elle s'échappe vers $-\infty$. Peut-être trouve-t-on une valeur intermédiaire qui donne lieu à une suite convergente ?

Certes, l'approche par tâtonnement devient vite fastidieuse. Heureusement il est possible de répondre à cette question dès que nous disposons d'une formule close pour le terme général x_n :

Proposition 2.6. *Toute suite vérifiant $x_{n+2} = x_{n+1} + x_n$ pour tout $n \geq 0$ est de la forme $x_n = a\alpha^n + b\beta^n$ avec des coefficients $a, b \in \mathbb{R}$ et des constantes $\alpha = \frac{1+\sqrt{5}}{2}$ et $\beta = \frac{1-\sqrt{5}}{2}$. En voici deux conséquences :*

(1) *Pour les deux premiers termes on obtient $x_0 = a + b$ et $x_1 = a\alpha + b\beta$, ce qui permet de trouver a, b en fonction de x_0, x_1. Concrètement, en fixant $x_0 = 1$, on obtient $a = \frac{x_1 - \beta}{\alpha - \beta}$ et $b = 1 - a$.*

(2) *Pour $a \neq 0$ la suite $(x_n)_{n\in\mathbb{N}}$ diverge, plus précisément on voit que $x_n \to +\infty$ pour $a > 0$, et $x_n \to -\infty$ pour $a < 0$. Le cas critique $a = 0$ se produit si et seulement si $x_1 = \beta x_0$: dans ce cas on a convergence vers zéro, $x_n = b\beta^n \to 0$ pour $n \to \infty$.*

Exercice/M 2.7. Montrer la proposition précédente.

Exercice 2.8. Le programme `fibonacci.cc` permet de calculer une suite de Fibonacci à partir de x_0 et x_1. Essayez de comprendre son fonctionnement, puis expérimentez avec ce programme.

(1) Quand vous vérifiez les calculs pour les tableaux ci-dessus, vous allez constater de petites différences étranges : alors que les valeurs initiales sont connues de manière exacte, le calcul des termes suivants est contaminé d'erreurs d'arrondi. A priori on ne s'y attend pas dans un calcul aussi simple, avec des « chiffres ronds ». Ceci s'expliquera plus loin : le programme repose sur les nombres à virgule flottante, qui ont un comportement particulier et qui font l'objet de ce chapitre.

(2) Qu'observez-vous autour du cas critique, $x_0 = 1$ et $x_1 \approx \beta \approx -0.6180339887498949$? Les premières valeurs calculées $x_2, x_3, x_4, \ldots$ semblent-elles plausibles, autant que l'on puisse dire des décimales affichées ? Les valeurs absolues $|x_n|$ décroissent-elles, comme prévu ? À partir de x_{50} (voire x_{100}) le calcul numérique déraille ; comment expliquer cette explosion d'erreurs ?

(3) Est-il possible de construire sur ordinateur une suite de Fibonacci convergente ? Quelle est la difficulté ? Dans quel sens le calcul numérique (utilisant les nombres machine) peut-il correspondre au raisonnement mathématique (concernant les nombres réels) ? Dans quelle mesure peut-on faire confiance à notre calcul numérique ?

2.3. Conclusion. Ce joli exemple est trop simpliste dans le sens qu'il est linéaire et vous disposez d'une formuler close. Ainsi vous pouvez analysez la situation dans le moindre détail, et en particulier la propagation des erreurs se comprend parfaitement. Dans un exemple réaliste la situation sera plus compliquée, mais qualitativement les mêmes phénomènes peuvent se produire, quoique de façon moins transparente. Ainsi pour tout calcul numérique qui se respecte il sera question de stabilité et de propagation d'erreurs.

3. Le problème de l'efficacité : calcul exact vs calcul arrondi

En analyse en construit certaines fonctions $f : \mathbb{R} \supset U \to \mathbb{R}$, comme $\sqrt[n]{x}$, $\sin(x)$, $\cos(x)$, $\exp(x)$, $\log(x)$, etc. Pour le calcul numérique sur ordinateur, nous aimerions calculer explicitement la valeur $f(x)$ pour un x donné. Typiquement cela ne sera pas possible de manière exacte, mais on peut espérer d'implémenter un calcul approché, si possible efficace.

3.1. La méthode de Newton-Héron. Pour entrer dans le vif du sujet, nous allons illustrer notre propos par une méthode importante et peu triviale : le calcul approché d'une racine $\sqrt[n]{a}$ d'un nombre réel $a > 0$. La méthode de Newton nous donne un formidable outil pour de telles approximations, car elle fournit une suite $(u_k)_{k\in\mathbb{N}}$ facilement calculable qui converge rapidement vers la racine cherchée.

Proposition 3.1 (Newton-Héron, version qualitative)**.** *Soient $n \geq 2$ un entier et $a \in \mathbb{R}_+$ un nombre réel positif. Pour toute valeur initiale $u_0 > 0$ la suite récurrente $u_{k+1} = \frac{1}{n}\left((n-1)u_k + a/u_k^{n-1}\right)$ converge vers la racine $r = \sqrt[n]{a}$, c'est-à-dire vers l'unique nombre réel $r \in \mathbb{R}_+$ vérifiant $r^n = a$.*

Vous connaissez ce résultat sans doute de votre cours d'analyse. À noter qu'il s'agit d'une affirmation topologique, à caractère purement qualitatif. Pour le calcul numérique il est indispensable de l'affiner par une majoration d'erreur (résultat métrique, à caractère quantitatif). Un peu mieux encore, l'énoncé suivant donne un encadrement (ce qui repose sur une structure encore plus fine : la relation d'ordre sur $\mathbb{R}$).

Théorème 3.2 (Newton-Héron, version quantitative)**.** *Soient $n \geq 2$ un entier et $a \in \mathbb{R}_+$ un nombre réel positif. Pour toute valeur initiale $u_0 > 0$ la suite récurrente $u_{k+1} = \frac{1}{n}\big((n-1)u_k + a/u_k^{n-1}\big)$ converge vers la racine $r = \sqrt[n]{a}$. Pour $k \geq 1$ on a convergence monotone $u_k \searrow r$. Symétriquement pour $v_k = a/u_k^{n-1}$ on a $v_k \nearrow r$. On obtient ainsi des encadrements explicites $v_k \leq r \leq u_k$ de plus en plus fins :*

$$v_1 \leq v_2 \leq v_3 \leq \cdots \leq r \leq \cdots \leq u_3 \leq u_2 \leq u_1$$

Quant à la vitesse de convergence, l'écart relatif $\varepsilon_k = \frac{u_k - r}{r}$ vérifie

$$\varepsilon_{k+1} \leq \min\big(\, \tfrac{n-1}{2}\varepsilon_k^2 \,,\, \tfrac{n-1}{n}\varepsilon_k \,\big)$$

Pour une valeur u_k loin de la racine r, la convergence est donc au moins linéaire, avec un rapport de contraction $\frac{n-1}{n} < 1$. Finalement, pour u_k proche de la racine r (à savoir pour $r \leq u_k \leq \frac{n+2}{n}r$) la convergence est quadratique : à chaque itération le nombre de décimales valables double à peu près.

Remarque 3.3. Pas tous les problèmes numériques admettent de solutions aussi élégantes et efficaces. Pour la mettre en relief, soulignons donc trois points forts de la méthode de Newton-Héron :

- C'est la convergence quadratique qui fait de ce théorème un outil très puissant : si vous avez calculé un encadrement $[v_k, u_k]$ à $\approx 10^{-2}$ près, disons, l'itération suivante ne laissera qu'un écart $\approx 10^{-4}$, celle d'après $\approx 10^{-8}$, puis $\approx 10^{-16}$ etc.
- Non seulement le théorème précédent vous garantit une convergence rapide, mais vous pouvez à tout moment, par un simple calcul, contrôler l'écart restant $|u_k - v_k|$. Ceci vous permet de terminer l'itération lorsque la précision est suffisante pour vos besoins.
- De plus la méthode est numériquement stable : si la valeur approchée u_k est perturbée par une erreur d'arrondi, les itérations suivantes convergeront tout de même.

Exercice/M 3.4. La dynamique de l'itération de Newton est une jolie application de l'étude de fonctions. Comme exercice, vous pouvez prouver le théorème en détaillant l'esquisse suivante.

ESQUISSE DE PREUVE. La fonction $p\colon \mathbb{R}_+ \to \mathbb{R}_+$, $x \mapsto x^n$ est strictement croissante, donc injective. Elle est continue avec $p(0) = 0$ et $p(x) \to \infty$ pour $x \to \infty$, donc surjective par le théorème des valeurs intermédiaires. Autrement dit, il s'agit d'une bijection de $\mathbb{R}_+$ sur $\mathbb{R}_+$: pour tout $a \in \mathbb{R}_+$ il existe un et un seul réel positif $r \in \mathbb{R}_+$ vérifiant $r^n = a$. Pour approcher la valeur r cherchée, nous itérons la fonction

$$f\colon \mathbb{R}_+ \to \mathbb{R}_+ \quad \text{donnée par} \quad f(x) = \frac{1}{n}\big((n-1)x + a/x^{n-1}\big).$$

L'unique point fixe de f est r. Elle est dérivable de classe C^∞, avec $f'(x) = \frac{n-1}{n}(1 - a/x^n)$. Sur $]0, r[$ on a $f' < 0$ et la fonction f est strictement décroissante et vérifie donc $f(x) > f(r) = r$. Sur $]r, +\infty[$ on a $0 < f'(x) < \frac{n-1}{n}$ et la fonction f est strictement croissante. Par le théorème des accroissements finis on a $f(x) - r = f(x) - f(r) = f'(\xi)(x - r) \leq \frac{n-1}{n}(x - r)$ donc $r < f(x) < x$. On conclut que le point fixe r est attractif, avec tout $\mathbb{R}_+$ pour bassin d'attraction. Les suites v_k et u_k donnent ainsi des encadrements

$$v_1 \leq v_2 \leq v_3 \leq \cdots \leq r \leq \cdots \leq u_3 \leq u_2 \leq u_1.$$

Par conséquent, les limites $v = \lim v_k$ et $u = \lim u_k$ existent et vérifient $v \leq r \leq u$. D'autre part, la continuité de f et l'équation de récurrence $u_{k+1} = f(u_k)$ donnent, par passage à la limite,

$$u = \lim u_{k+1} = \lim f(u_k) = f(\lim u_k) = f(u).$$

On conclut que $u = r$ par unicité du point fixe. De manière analogue $v = r$.

Étudions finalement la vitesse de convergence. Pour tout $k \geq 1$ nous avons $u_k = r(1 + \varepsilon_k)$ avec une erreur relative $\varepsilon_k \geq 0$. Après un petit calcul on trouve que $\varepsilon_{k+1} = g(\varepsilon_k)$ est obtenue en itérant la fonction

$$g(\varepsilon) = \tfrac{n-1}{n}\varepsilon + \tfrac{1}{n}\big[(1+\varepsilon)^{1-n} - 1\big].$$

Évidemment g est majorée par $\frac{n-1}{n}\varepsilon$, ce qui donne la convergence linéaire. Mais g est aussi majorée par $h(\varepsilon) = \frac{n-1}{2}\varepsilon^2$, ce qui assure la convergence quadratique. Effectivement, on a $g(0) = h(0) = 0$ et $g'(0) = h'(0) = 0$ ainsi que $g''(\varepsilon) \leq h''(\varepsilon)$ pour tout $\varepsilon \geq 0$, ce qui implique $g' \leq h'$ puis $g \leq h$. □

3.2. Exemples numériques.

Exemple 3.5. Calculons par exemple des valeurs approchées de $r = \sqrt{2}$ à 10^{-10} près. Avec la valeur initiale $u_0 = 1$ l'itération de Newton donne les encadrements suivants :

$$\begin{aligned} \tfrac{4}{3} \le r \le \tfrac{3}{2} \qquad & 1.3333333333 \le r \le 1.5000000000 \\ \tfrac{24}{17} \le r \le \tfrac{17}{12} \qquad & 1.4117647058 \le r \le 1.4166666667 \\ \tfrac{816}{577} \le r \le \tfrac{577}{408} \qquad & 1.4142114384 \le r \le 1.4142156863 \\ \tfrac{941664}{665857} \le r \le \tfrac{665857}{470832} \qquad & 1.4142135623 \le r \le 1.4142135624 \end{aligned}$$

Dans la dernière ligne on a $v_4 - u_4 \le 10^{-10}$. À noter que les valeurs dans $\mathbb{Q}$ à gauche sont exactes, alors que les développements décimaux à droite sont arrondis à 10 décimales : vers le bas pour v_k et vers le haut pour u_k, afin de garantir la correction de l'encadrement affiché.

Exemple 3.6. Dans l'exemple précédent, le calcul dans $\mathbb{Q}$ semble satisfaisant. En général il n'en est rien ! Regardons le calcul de $r = \sqrt[3]{10}$ à 10^{-10} près. Avec la valeur initiale $u_0 = 1$ l'itération de Newton donne :

$$\begin{aligned} \tfrac{5}{8} &\le r \le \tfrac{4}{1} \\ \tfrac{640}{529} &\le r \le \tfrac{23}{8} \\ \tfrac{44774560}{24098281} &\le r \le \tfrac{4909}{2116} \\ \tfrac{6500450623479657648040}{3049614553054553981809} &\le r \le \tfrac{55223315303}{25495981298} \\ \tfrac{151137897149456207062734071576976875584120695784505967636052968 10}{701559854671230732015021361552743850496833886768021475309468 6841} &\le r \le \tfrac{837591699261179839454692621670 29}{388764578053937685469668481040 41} \end{aligned}$$

Après cinq itérations la précision $v_5 - u_5 \approx 0,0001837$ est encore loin d'être satisfaisante, mais les fractions produites dans le calcul commencent déjà à exploser. Il faut encore deux itérations pour atteindre la précision souhaitée, et les numérateurs et dénominateurs ont quelques centaines de chiffres. (Inutile de les reproduire ici, mais vous êtes invités à le vérifier sur ordinateur.)

Exemple 3.7. On peut calculer directement avec des développements décimaux sous forme de nombres à virgule flottante, en utilisant l'arithmétique arrondie expliquée plus bas. Nous obtenons ainsi les encadrements suivants de la racine cherchée $r = \sqrt[3]{10}$:

$$\begin{aligned} 0.6250000000000000 \le r \le 4.0000000000000000 \\ 1.2098298676748582 \le r \le 2.8750000000000000 \\ 1.8579980870834728 \le r \le 2.3199432892249528 \\ 2.1315646651045386 \le r \le 2.1659615551777928 \\ 2.1543122250101293 \le r \le 2.1544959251533748 \\ 2.1544346865510652 \le r \le 2.1544346917722930 \\ 2.1544346900318837 \le r \le 2.1544346900318838 \end{aligned}$$

Après sept itérations nous arrivons ainsi à une précision $< 10^{-16}$, comme souhaité.

3.3. Conclusion. Soulignons à nouveau que les nombres rationnels $\mathbb{Q}$ forment un corps, ce qui veut dire que vous pouvez effectuer les quatre opérations de base $+, -, \cdot, /$ comme vous les connaissez. Ajoutons que l'on peut représenter tout nombre rationnel de manière exacte sur ordinateur, typiquement sous forme de numérateur et dénominateur, et ainsi les calculs dans $\mathbb{Q}$ s'effectuent de manière exacte.

En analyse on s'intéresse surtout au corps $\mathbb{R}$ des nombres réels. Ici une représentation exacte sur ordinateur est en général impossible, mais on peut approcher n'importe quel nombre réel par des rationnels. Cette propriété est fondamentale pour la théorie et aussi pour des calculs numériques.

Jusqu'ici tout marche bien, mais malheureusement les nombres rationnels entraînent assez souvent des calculs inutilement lourds. La catastrophe de l'exemple 3.6 illustre que le calcul exact dans $\mathbb{Q}$, bien que théoriquement préférable, peut être mal adapté et inefficace pour certaines tâches. Ce phénomène est assez fréquent dans les calculs itératifs : lors d'un calcul dans $\mathbb{Q}$ les fractions peuvent exploser avant que l'on n'arrive à un résultat satisfaisant (suffisamment proche de la limite cherchée). Dans ce cas la manipulation des nombres rationnels devient trop coûteuse en temps et mémoire.

4. Qu'est-ce que les nombres à virgule flottante ?

Par souci d'efficacité il semble avantageux de calculer avec un développement décimal convenablement arrondi à chaque étape. De toute manière, c'est souvent ce format qui est souhaité pour le résultat final. Dans ces circonstances on abandonne le calcul exact au profit des *nombres à virgule flottante*. C'est une astucieuse invention de l'informatique qui permet des calculs efficaces. Hélas, le prix à payer sont les erreurs d'arrondi. Par conséquent il faut comprendre leur fonctionnement et quelques règles de bon sens afin d'utiliser intelligemment cet outil informatique et d'interpréter correctement ses résultats.

4.1. Développement binaire. Regardons le développement binaire d'un nombre réel $x \in [1,2]$:

$$x = \langle 1.a_1a_2a_3a_4\ldots\rangle_{\text{bin}} := 1 + \sum_{k=1}^{\infty} a_k 2^{-k} \quad \text{avec des chiffres binaires } a_k \in \{0,1\}. \tag{1}$$

Toute telle série converge vers un nombre réel $x \in [1,2]$, et réciproquement tout $x \in [1,2]$ peut être représenté par un tel développement. (Certains nombres en admettent deux — lesquels ?) Pour recouvrir tout $\mathbb{R}_+$ on multiplie par un facteur 2^e avec exposant $e \in \mathbb{Z}$, puis on ajoute un signe pour atteindre $\mathbb{R}_-$:

$$x = \pm\langle 1.a_1a_2a_3a_4\ldots\rangle_{\text{bin}} \cdot 2^e. \tag{2}$$

La représentation (1) est *à virgule fixe*, alors que la représentation (2) est *à virgule flottante*, parce que le facteur 2^e permet de décaler la virgule, c'est-à-dire de la rendre « flottante ». En base 10 par exemple on a $1234,567 = 123,4567 \cdot 10^1 = 12,34567 \cdot 10^2 = 1,234567 \cdot 10^3$. On peut ainsi supposer que la virgule se trouve immédiatement après le premier chiffre non nul, ce qui rend cette représentation unique.

4.2. Nombres à virgule flottante. Sur ordinateur on ne peut pas stocker une suite *infinie* de chiffres. On fixe donc une longueur ℓ et on ne considère que les nombres réels qui s'écrivent

$$\pm\,\langle 1.a_1a_2a_3\ldots a_\ell\rangle_{\text{bin}} \cdot 2^e. \tag{3}$$

La suite des chiffres $m = (1, a_1, a_2, a_3, \ldots, a_\ell)$ est appelée la *mantisse*, et $\ell+1$ est donc la *longueur* de la mantisse. Très souvent on restreint aussi l'exposant e à un intervalle $[\![e_{\min}, e_{\max}]\!]$ fixé d'avance. La définition des nombres à virgule flottante dépend donc du choix des trois paramètres $\ell, e_{\min}, e_{\max}$.

Définition 4.1. L'ensemble R formé de 0 et des nombres de la forme (3) est *l'ensemble des nombres exactement représentables* avec une mantisse de longueur $\ell+1$ et un exposant $e \in [\![e_{\min}, e_{\max}]\!]$.

$$R = \{0\} \cup \left\{ \pm\left(1 + \frac{m}{2^\ell}\right) \cdot 2^e \mid m \in [\![0, 2^\ell[\![,\ e \in [\![e_{\min}, e_{\max}]\!] \right\}$$

On les appelle aussi *nombres à virgule flottante*, ou un peu plus court *nombres flottants*, ou encore *nombres machine*. Il s'agit d'un sous-ensemble fini de $\mathbb{R}$. À noter que R ne forme pas un sous-corps de $\mathbb{R}$ et que R n'a aucune propriété mathématique intéressante — mise à part la simplicité de ses éléments en développement binaire.

Exemple 4.2. Avec une mantisse de longueur $\ell+1 = 3$ et un exposant $e \in [\![-3, 0]\!]$, les nombres exactement représentables sont $R = \{0\} \cup \{\pm 1\} \cdot \{\frac{4}{4}, \frac{5}{4}, \frac{6}{4}, \frac{7}{4}\} \cdot \{2^{-3}, 2^{-2}, 2^{-1}, 2^0\}$. Graphiquement, ceci donne la discrétisation suivante de l'intervalle $]-2,2[$, qui ressemble vaguement à une *échelle logarithmique* :

On voit par exemple que $x = 9/8$ ne peut être représenté par un tel nombre machine : il faut l'approcher par un de ses voisins $\lfloor x \rfloor_R = 1,0$ ou $\lceil x \rceil_R = 1,25$. Il en est de même pour le nombre $\pi/2$ qui peut être approché par son voisin le plus proche $\lfloor \pi/2 \rceil_R = 1,5$, par exemple. Ainsi le choix des nombres machine introduit des erreurs d'arrondi inévitables. Typiquement il faut s'attendre à une erreur relative $\frac{|\hat{x}-x|}{|x|} \approx 2^{-\ell}$.

4.3. Les types primitifs du C++. Augmenter la longueur ℓ de la mantisse rend la discrétisation plus fine. Élargir l'intervalle de l'exposant $[\![e_{\min}, e_{\max}]\!]$ étend l'intervalle de la discrétisation. (Le détailler sur des exemples.) En C++ on dispose des types `float`, `double` et `long double` pour les nombres à virgule flottante, correspondant aux paramètres suivants :

type	mantisse	erreur relative	exposant	minimum	maximum
`float`	24 bits	$2^{-24} \approx 6 \cdot 10^{-8}$	8 bits	$2^{-128} \approx 10^{-38}$	$2^{127} \approx 10^{38}$
`double`	53 bits	$2^{-53} \approx 1 \cdot 10^{-16}$	11 bits	$2^{-1024} \approx 10^{-308}$	$2^{1023} \approx 10^{308}$
`long double`	64 bits	$2^{-64} \approx 5 \cdot 10^{-20}$	15 bits	$2^{-16384} \approx 10^{-4932}$	$2^{16383} \approx 10^{4932}$

Exercice 4.3. On peut empiriquement tester ces valeurs à l'aide du programme `precision.cc`. Il provoque délibérément une erreur d'arrondi pour déterminer la longueur de la mantisse, ou un dépassement de capacité pour déterminer la plage des exposants. L'idée est de trouver par tâtonnement le plus petit exposant $k > 0$ tel que pour $\varepsilon = 2^{-k}$ on ait `1.0 + eps == 1.0`. Essayer d'expliquer le fonctionnement de ce test, puis l'utiliser pour vérifier le tableau ci-dessus. Vous constaterez de petites différences entre les valeurs du tableau et vos résultats empiriques. (Si cela vous intéresse vous pouvez vous renseigner sur Internet sur la norme IEEE 754 qui définit les nombres flottants dans le moindre détail. Voir aussi `www.vinc17.org/research/papers/arithflottante.pdf`.)

4.4. Comment calculer avec les flottants ? À l'instar des opérations réelles $+, -, *, /: \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ on veut définir les opérations élémentaires `+`, `-`, `*`, `/` pour les nombres flottants. On peut bien-sûr regarder la restriction $+, -, *, /: R \times R \to \mathbb{R}$, mais dans la majorité des cas les résultats ne retomberont pas dans le sous-ensemble $R \subset \mathbb{R}$ des nombres machine. Il faut donc *arrondir* pour représenter le résultat.

☞ *Due à la précision limitée, les opérations élémentaires ne peuvent rendre qu'une valeur approchée lorsque le résultat exact n'est pas représentable dans la numération choisie.*
*On parle ainsi de l'*arithmétique arrondie *ou aussi de l'*arithmétique flottante. ✍

Exercice/M 4.4. Essayez de définir une addition $+: R \times R \to R$ pour notre exemple minuscule ($\ell + 1 = 3$, $e \in [\![-3, 0]\!]$). Quelles additions sont exactes, lesquelles faut-il arrondir ? Dans votre définition vous pouvez vous servir des valeurs exceptionnelles $\pm\infty$ si vous voulez. (Elles sont tellement utiles qu'elles sont prévues dans toutes les implémentations standards des nombres à virgules flottantes.)

☞ On explicitera une implémentation complète au §XVI, quand on parlera du « calcul arrondi fiable ».

Tous les calculs ultérieurs se baseront sur ces opérations élémentaires : évaluation des polynômes ou des fractions rationnels, approximation des fonctions usuelles comme $\sqrt{}, \exp, \log, \sin, \cos, \ldots$. Comme les opérations élémentaires sont déjà des approximations, on doit s'attendre à une propagation d'erreurs (parfois non négligeable). On en verra quelques exemples dans la suite.

4.5. Quand vaut-il mieux calculer de manière exacte ? Le principal problème du calcul approché est la propagation des erreurs d'arrondi. Ainsi le résultat final calculé peut être assez éloigné du résultat exact cherché. Un exemple flagrant de ce genre est le « polynôme de Rump » (voir `rump.cc`) :

$$f(x,y) = \frac{1335}{4} y^6 + \frac{11}{2} y^8 + \frac{x}{2y} + x^2 \left(11x^2y^2 - y^6 - 121y^4 - 2\right)$$

On trouve par un calcul exact que $f(77617, 33096) = -54767/66192 \approx -0.8274$. Or, l'évaluation en $x = 77617$, $y = 33096$ provoque de graves problèmes quand on utilise des nombres à virgule flottante :

- Calcul de $f(77617, 33096)$ utilisant le type `float` : `-1.1056291e+30`
- Calcul de $f(77617, 33096)$ utilisant le type `double` : `1.787028332140613e+20`
- Calcul de $f(77617, 33096)$ utilisant le type `long double` : `0`

On observe ici une perte totale de chiffres significatifs autrement dit une explosion de l'erreur relative. On n'arrive même pas à déterminer le bon signe ! Si vous voulez vérifier ces résultats, tout à fait imprévisibles, vous pouvez vous servir du programme `rump.cc`. Par exemple, vous pouvez tester des paramètres voisins afin de comprendre un peu mieux ce qui se passe. (Voir §5.4 pour un phénomène similaire d'annulation.) Ici le calcul exact est clairement préférable : la question admet une formulation qui ne fait intervenir que de nombres rationnels, et les calculs exacts à effectuer sur ordinateur ne sont pas trop coûteux.

5. Des pièges à éviter

Le but principal des exercices suivants est de vous convaincre que les nombres machine modélisent assez mal le corps $\mathbb{R}$ des nombres réels ! Ainsi l'usage du calcul numérique puis l'interprétation des résultats présentent des difficultés particulières. Une certaine expérience et un esprit critique sont donc importants pour tout utilisateur, et d'autant plus pour tout programmeur qui se respecte.

Même sans ambitions approfondies, il faut au moins connaître *de bons et de mauvais exemples*. Nous allons donc faire quelques expériences numériques sur ordinateur. Vous trouverez dans la suite des exemples suffisamment drastiques, j'espère, pour vous vacciner durablement contre le calcul naïf. Notre but sera ensuite de comprendre sous quelles conditions un calcul arrondi peut tout de même aboutir à un résultat significatif.

5.1. Nombres non représentables. Les erreurs d'arrondi peuvent se produire dans les calculs les plus simples, même avec des nombres rationnels. Voici un exemple typique. Le calcul suivant ne donne pas 0, comme il serait mathématiquement correct. Le tester puis expliquer son résultat :

```
float a= 10.0 / 3.0;        // calcul exact : a vaut 10/3
float b=    a - 3.0;        //                b vaut 1/3
float c=    b * 3.0;        //                c vaut 1
float d=    c - 1.0;        //                d vaut 0
cout << d << endl;          // Que vaut le résultat approximatif ?
```

Effectuer aussi le calcul similaire `a=10.0/4.0; b=a-2.0; c=b*2.0; d=c-1.0;` Cette fois-ci le résultat est-il exact ? Comment expliquer ce phénomène chanceux ?

☞ *Même les calculs les plus innocents peuvent produire des erreurs d'arrondi.* ✍

5.2. Quand deux nombres flottants sont-ils « égaux » ? Les erreurs d'arrondi sont aussi inévitables qu'imprévisibles. Pour en comprendre les effets catastrophiques possibles, souvent inattendus, regardons un logiciel de gestion comme le suivant (légèrement simplifié).

Programme XV.1 L'agent comptable est innocent ! `compte.cc`

```
#include <iostream>
using namespace std;

int main()
{
  float somme= 0.0;
  for ( int i=1; i<=10; ++i ) somme+= 0.1;
  cout << "La somme vaut " << somme << "." << endl;
  if ( somme == 1.0 ) cout << "Le compte est bon." << endl;
  else cout << "Vous êtes accusé de détournement de fonds." << endl;
}
```

Exercice/P 5.1. Quel résultat ce programme donne-t-il ? Le tester puis l'expliquer. *Indication.* — On pourra faire afficher `somme-1` pour tester si les deux flottants sont égaux ou seulement très proches. Par curiosité on pourrait augmenter la précision : est-ce que le type `float` ou `double` ou `long double` joue un rôle ? A priori on ne s'attend pas aux erreurs quand on calcule avec des nombres « ronds » comme 0.1. Pour comprendre l'occurrence des erreurs d'arrondi, déterminer le développement binaire de $0,1_{\text{dec}}$.

☞ *Afin de tester l'égalité de deux nombres flottants, il faut remplacer `a==b` par `abs(a-b)<eps` et `a!=b` par `abs(a-b)>=eps`.* ✍

5.3. Combien de chiffres sont significatifs ? Le fichier en-tête `<cmath>` fournit quelques fonctions usuelles, comme `exp`, `log`, `pow`, `sqrt`, etc. Quelle précision peut-on attendre de ces fonctions ?

Exercice/P 5.2. Pour plus de précision on pourrait être tenté d'afficher plus de chiffres, par exemple :

```
float a= sqrt(2.0); cout.precision(50); cout << a << endl;
```

Expliquer l'erreur logique dans cette approche. Combien de chiffres sont significatifs ? Pour résoudre ce mystère et pour bien rigoler, remplacer `sqrt(2.0)` par `10.0/3.0`, puis `float` par `double`, ...

☞ *Il faut se méfier de l'affichage inutile de chiffres non significatifs.* ✍

Psychologiquement, un nombre affiché avec beaucoup de décimales suggère une grande précision, le problème étant que les chiffres terminaux n'ont souvent aucune signification ! Vous pouvez vérifier cette observation au quotidien. Penser par exemple à un sondage qui parle de « 57,14% des personnes interrogées » au lieu de dire « quatre des sept personnes interrogées ».

☞ *L'affichage des chiffres non significatifs est soit maladroite soit malhonnête.*
Afficher n décimales seulement si l'erreur relative est plus petite que $5 \cdot 10^{-n}$. ✍

5.4. Perte de chiffres significatifs. D'après ce qui précède, il faut préserver précieusement un maximum de chiffres significatifs, ce qui est souvent difficile. Par contre, les expériences suivantes montrent qu'il est très facile de détruire la signification d'un résultat.

Exercice/P 5.3. Incroyable mais vrai : l'addition des flottants n'est même pas associative ! Pour `a=-1e30, b=1e30, c=1.0` comparer `(a+b)+c` et `a+(b+c)`. Expliquer la différence.

☞ *L'addition de deux nombres, l'un « grand » l'autre « petit »,*
entraîne la perte de chiffres significatifs contribués par le petit. ✍

Exercice/P 5.4. La définition $f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$ pourrait inspirer la tentative d'une dérivation numérique comme suit. Essayez d'en prédire le résultat, puis vérifiez-le.

Programme XV.2 Dérivation numérique naïve `derivation.cc`

```
1  #include <iostream>
2  using namespace std;
3
4  double f( double x )
5  { return 3.14159265358979323846 * x * x; }
6
7  int main()
8  {
9    cout.precision(20);                 // demander 20 décimales à l'affichage
10   double a= 1.0, eps= 1.0;            // on calcule avec 16 décimales
11   for ( int i=0; i<60; ++i, eps/=2 )
12     cout << ( f(a+eps)-f(a) ) / eps << endl;
13 }
```

☞ *La soustraction de deux nombres proches, ou l'addition de deux nombres presque opposés,*
produit une perte (parfois considérable voire totale) de chiffres significatifs. À éviter ! ✍

En voici un exemple :

$x = 1,23456789047321$	avec quinze décimales significatives
$y = 1,23456789021588$	avec quinze décimales significatives
$x - y = 0,00000000025733$	seulement cinq décimales significatives

5.5. Comment éviter une perte de chiffres significatifs ? Regardons un exemple important : comment implémenter une approximation pas trop mauvaise de l'exponentielle $\exp\colon \mathbb{R}\to\mathbb{R}_+$? Le programme `exp.cc` le fait via la série $\exp(x)=\sum_{k=0}^{\infty}\frac{1}{k!}x^k$. Évidemment on ne peut pas sommer une infinité de termes, on doit donc se contenter d'aller jusqu'à un certain rang n. Celui-ci doit être choisi

– suffisamment grand pour garantir une bonne majoration du reste négligé,
– mais pas inutilement grand afin d'obtenir un calcul efficace.

Dans notre exemple naïf on laisse le choix de x et n à l'utilisateur. Ensuite le programme évalue la série tronquée après le $n$ième terme, $s_n(x)=\sum_{k=0}^{n}\frac{x^k}{k!}$, par la méthode de Horner :

$$s_n(x)=\sum_{k=0}^{n}\frac{x^k}{k!}=\left(\left(\dots\left(\left(\left(\frac{x}{n}+1\right)\frac{x}{n-1}+1\right)\frac{x}{n-2}+1\right)\dots\right)\frac{x}{2}+1\right)\frac{x}{1}+1$$

Soulignons qu'il est toujours une bonne idée de considérer Horner quand il s'agit d'évaluer un polynôme : c'est simple et efficace. Pourtant, dans le calcul numérique deux problèmes se posent :

– Si $|x|$ est très grand, les premiers termes $\frac{|x^k|}{k!}$ croissent avant que la factorielle $k!$ ne l'emporte. Ceci veut dire qu'il faut aller assez loin dans la série afin d'obtenir un reste suffisamment petit. Le tester empiriquement pour $x=10,20,30,40,50,\dots$.
– Pour $x<0$ les termes $\frac{x^k}{k!}$ sont de signes alternés. Quand $|x|$ est grand, ceci entraîne une perte dramatique de chiffres significatifs. Le tester empiriquement pour $x=-10,-20,-30,-40,-50,\dots$. Pourquoi ne sert-il plus à rien d'augmenter le rang n dans ce cas ?

☞ *Souvent la perte de précision peut être évitée en reformulant le calcul.* ✍

Dans ce cas concret la solution est très simple : on évalue la série seulement quand $|x|$ est petit, disons pour x dans l'intervalle $[-1,1]$ où le comportement numérique est excellent. Pour $|x|>1$ on se ramène au cas précédent via $\exp(x)=\exp(x/2)^2$, en profitant de notre précieuse connaissance de la fonction exp.

Exercice/M 5.5. Montrer l'égalité de $\sqrt{1+x}-1$ et $\frac{x}{1+\sqrt{1+x}}$. Prédire puis comparer les résultats d'un calcul numérique pour x proche de zéro. Quelle formule est préférable et pourquoi ?

Exercice/M 5.6. Discuter la formule $\sinh(x)=\frac{1}{2}(e^x-e^{-x})$ sous l'aspect d'une éventuelle perte de précision. Esquisser une implémentation sérieuse de la fonction $\sinh(x)$ pour x proche de 0. Est-ce que le même problème se pose pour $\cosh(x)=\frac{1}{2}(e^x+e^{-x})$?

5.6. Phénomènes de bruit. Rappelons que la méthode dichotomique pour résoudre une équation réelle $f(x)=0$ se base sur le théorème des valeurs intermédiaires :

Théorème 5.7. *Soit $f\colon[a,b]\to\mathbb{R}$ une fonction continue et y une valeur comprise entre $f(a)$ et $f(b)$. Alors il existe $x\in[a,b]$ de sorte que $f(x)=y$. En particulier, si $f(a)$ et $f(b)$ n'ont pas le même signe, alors il existe un nombre réel $x\in[a,b]$ avec $f(x)=0$.*

Exercice/M 5.8. La méthode dichotomique pour résoudre $f(x)=0$ procède comme suit : supposons que $a_k<b_k$ et $f(a_k)<0<f(b_k)$. On prend $x_k=\frac{1}{2}(a_k+b_k)$ puis on compare : si $f(x_k)>0$ alors on continue avec l'intervalle $[a_k,x_k]$, si $f(x_k)<0$ alors on continue avec l'intervalle $[x_k,b_k]$. Montrer que cette algorithme produit une suite (x_k) qui converge vers une limite $x\in[a,b]$ vérifiant $f(x)=0$. Montrer de plus la majoration $|x-x_k|\le\frac{1}{2}|b_k-a_k|=2^{-k}|b-a|$, ce qui prouve une convergence linéaire.

Selon vos expériences numériques, quels problèmes prédiriez-vous pour une implémentation de cette méthode sur ordinateur ? Quelle précision peut-on espérer à réaliser ? Quels sont les facteurs limitant ?

Le programme suivant illustre un phénomène de « bruit » très embêtant dans l'évaluation de fonctions, aussi gentilles qu'elles soient. On évalue le polynôme $f(x)=x^6-9x^5+30x^4-40x^3+48x-32$ par la méthode de Horner, ce qui en soi est une bonne idée. Puis on affiche les valeurs $f(x)$ pour x proche de 2.

Exercice 5.9. Vérifier que le polynôme f s'annule en $x=2$ (exactement). Numériquement, en utilisant le type `double`, il se trouve que pour $x\in[1,9997;2,0003]$ la valeur $f(x)$ *oscille* aléatoirement autour de 0. Vérifiez-le et essayez d'expliquer ce phénomène. Est-ce une propriété de la fonction f ou bien un artefact de notre implémentation ? Bien sûr, ce phénomène de bruit est désespérant quand on cherche à trouver une solution de $f(x)=0$ par la méthode dichotomique !

Programme XV.3 Bruit « aléatoire » dans l'évaluation d'une fonction `bruit.cc`

```
#include <iostream>
using namespace std;

double f( const double& x )
{ return ((((x-9)*x+30)*x-40)*x*x+48)*x-32; }

int main()
{
  cout.precision(10);                              // Demande d'afficher 10 décimales
  cout.setf( ios::scientific | ios::showpos );     // pour bien aligner les chiffres.
  for( double x=1.9997; x<=2.0003; x+=0.00001 )    // Cette boucle produit un tableau.
    cout << "f(" << x << ") = " << f(x) << endl; // Il y aura des surprises...!
}
```

5.7. Conditions d'arrêt. Un problème particulier se pose pour la condition d'arrêt dans une itération $x_{k+1} = f(x_k)$. Supposons que la suite converge vers une limite $x = \lim x_k$, dont on cherche une valeur approchée à une certaine précision $\varepsilon > 0$ près. Mathématiquement il faut itérer jusqu'à ce que $|x - x_k| \leq \varepsilon$. (Dans la pratique on ne connaît en général pas la valeur x, donc on remplace cette condition d'arrêt par $|x_k - x_{k-1}| \leq \varepsilon$, et on majore la vraie distance $|x - x_k|$ par une autre méthode.) Numériquement cette condition peut parfois être impossible à atteindre. Reprenons la méthode de Newton :

Programme XV.4 Première tentative d'implémenter Newton-Héron

```
float inf= 1, sup= a, ecart;
do {
  sup= ( (n-1)*sup + inf ) / n;
  inf= a / puissance( sup, n-1 );
  ecart= abs( sup-inf );
} while( ecart >= eps );  // Cette condition d'arrêt est-elle raisonnable ?
```

Exercice/P 5.10. À cause de la précision limitée, la condition d'arrêt risque de causer de sérieux problèmes. Tout marche, par chance, pour $a = 2$, $n = 2$, $\varepsilon = 1e-20$, mais tourne à la catastrophe pour $a = 3$. Le tester empiriquement ; il faut donc diminuer la précision exigée. Que se passe-t-il quand on remplace `float` par `double` puis par `long double` ? Quelle précision est raisonnable ? Comment le savoir d'avance ?

☞ *Même avec le calcul le plus sophistiqué on ne peut pas surpasser la précision (souvent très limitée) des nombres flottants utilisés.* ✍

Malgré ces difficultés, il faut modifier la condition d'arrêt de sorte que le calcul se termine toujours : soit l'écart souhaité est atteint, soit l'écart ne diminue plus. Tester ainsi le programme `racine.cc`, convenablement modifié. Peut-on ainsi calculer $\sqrt[2]{3}$ à $\varepsilon = 10^{-10}$ près ? à $\varepsilon = 10^{-20}$ près ? Peut-on être sûr de la précision atteinte ? (On développera une réponse satisfaisante avec le calcul fiable plus bas.)

Programme XV.5 Seconde tentative d'implémenter Newton-Héron

```
float inf= 1, sup= a, ecart= abs( sup-inf ), ecart_precedent;
do {
  sup= ( (n-1)*sup + inf ) / n;
  inf= a / puissance( sup, n-1 );
  ecart_precedent= ecart;
  ecart= abs( sup-inf );
} while( ecart >= eps && ecart < ecart_precedent );
```

5.8. Équations quadratiques. L'équation $ax^2+bx+c=0$ admet deux solutions, données par la formule bien connue $x_\pm = \frac{1}{2a}(-b \pm \sqrt{b^2-4ac})$. Outre les opérations arithmétiques elle n'utilise que la fonction $x \mapsto \sqrt{x}$, dont on vient de discuter une méthode d'approximation numérique. Le programme `quadratique.cc` en déduit une implémentation hâtive et insoucieuse pour les équations quadratiques. Il résout correctement $x^2-x-2=0$, mais il traite beaucoup d'autres cas de manière maladroite :

(1) Notre programme ne fait aucun effort pour éviter une perte de chiffres significatifs : Pour $x^2-12345x+1=0$ il trouve $x_+ = 12345$ et $x_- = 0$ au lieu de $x_+ \approx 12344,99992$ et $x_- \approx 0,00008$. L'erreur relative de x_+ est petite, mais pour x_- elle est de 100%. Dans un tel cas il sera avantageux de calculer d'abord x_+ puis $x_- = \frac{c}{ay}$. (Le cas inverse est également possible.)

(2) Dans le cas $a=0$ il s'agit d'une équation linéaire, ce qui est plus facile mais nécessite un traitement à part. Lancer notre programme pour résoudre $x-2=0$, il y aura des surprises...

(3) Dans le cas $d<0$ les solutions sont complexes ; tester le comportent du programme sur l'exemple x^2+2x+3. Pour un programme qui se respecte il sera certes une bonne idée de prévoir des solutions complexes, ou mieux encore d'autoriser même des coefficients complexes.

☞ Un nombre complexe peut être modélisé par une paire de nombres flottants, ce qui entraîne les avantages et inconvénients discutés plus haut : calcul efficace mais erreurs d'arrondi inévitables. Après inclusion du fichier en-tête `<complex>` vous pouvez utiliser les types `complex<float>`, `complex<double>`, etc. Si vous voulez, modifier ainsi le programme `quadratique.cc` et optimisez-le.

6. Sommation de séries

Dans les exercices suivants vous pouvez expérimenter avec les différents types de nombres flottants comme `float`, `double`, `long double`, ou bien `mpf_class` de la bibliothèque GMP (tapez `info gmp C++` dans une ligne de commande).

Exercice/P 6.1. La série $\sum \frac{1}{k}$ diverge, c'est-à-dire les sommes partielles $s_n = \sum_{k=1}^{n} \frac{1}{k}$ croissent sans borne. Pourtant elles deviennent stationnaires quand on les calcule naïvement sur ordinateur ! Essayez de prédire la valeur stationnaire pour le type `float`. Le vérifier avec le programme `divergence.cc`. Quelle valeur trouvez-vous ? Que se passe-t-il avec le type `double` ?

☞ *Se méfier d'une apparente « convergence numérique » sans contrôle d'erreur.* ✍

Exercice/P 6.2. La série $\sum_{k=1}^{\infty} \frac{1}{k^2}$ converge. Pour approcher la limite, en utilisant le type `float`, calculer $\sum_1^n \frac{1}{k^2}$ dans le sens des indices croissants, puis $\sum_n^1 \frac{1}{k^2}$ dans le sens des indices décroissants. Vu la nature des erreurs d'arrondi, quelle approche vous semble plus exacte ? Comparer avec les résultats obtenus avec le type `double` et `long double`.

☞ *Lors d'une sommation numérique l'ordre des termes peut influencer le résultat.* ✍

Exercice/M 6.3. Afin d'encadrer la valeur de $\zeta(2) = \sum_{k=1}^{\infty} \frac{1}{k^2}$ il ne suffit évidemment pas de calculer la somme partielle s_n, il faut aussi majorer le reste $r_n = \sum_{k=n+1}^{\infty} \frac{1}{k^2}$. Montrer que $\frac{1}{n+1} < r_n < \frac{1}{n}$:

(1) via l'encadrement $\frac{1}{k} - \frac{1}{k+1} < \frac{1}{k^2} < \frac{1}{k-1} - \frac{1}{k}$ puis une somme télescopique,

(2) via l'encadrement $\int_k^{k+1} \frac{1}{x^2}dx < \frac{1}{k^2} < \int_{k-1}^{k} \frac{1}{x^2}dx$ puis la somme des intégrales.

En déduire un programme qui calcule un encadrement de $\zeta(2)$ en fonction de n. (On sait d'ailleurs que $\zeta(2) = \pi^2/6$, mais ce beau résultat ne joue pas de rôle ici.)

☞ *C'est la majoration du reste qui fait d'une série $\sum_{k=1}^{\infty} a_k$ une méthode praticable pour calculer une valeur approchée de la somme.* ✍

Exercice/M 6.4. En généralisant l'exercice précédent, écrire un programme qui calcule un encadrement de $\zeta(3) = \sum_{k=1}^{\infty} \frac{1}{k^3}$ en fonction de n. Si vous voulez, vous pouvez étendre cette approche afin d'encadrer $\zeta(s)$ pour $s = 2,3,4,5,\ldots$. Est-ce envisageable pour tout $s>1$?

☞ *Essayez toujours de justifier vos résultats numériques en précisant leur marge d'erreur.* ✍

Exercice/P 6.5. Afin de calculer l'exponentielle $\exp(x)$ on pourrait théoriquement se servir de la limite $\exp(x) = \lim_{n\to\infty} \left(1+\frac{x}{n}\right)^n$. Quels problèmes prédiriez-vous ? Pour le tester empiriquement, implémentez efficacement ce calcul et essayez d'ainsi calculer $\exp(\pm 1)$ ou $\exp(\pm 100)$ à 10 décimales près.
Indication. — Il est inutile de parcourir tous les cas $n = 1,2,3,\ldots$, vous pouvez en choisir une suite extraite judicieuse. À noter que pour $n = 2^k$ la puissance a^n est particulièrement facile à calculer : évitez une boucle $a^1, a^2, a^3, \ldots, a^n$ de longueur n, une boucle $a^2, a^4, a^8, \ldots, a^n$ de longueur k suffit !

Comparer avec l'approche du §5.5. Si vous voulez, vous pouvez complétez le programme `exp.cc` en une implémentation plus robuste qui calcule l'exponentielle à au moins 10 décimales près. Le tester sur beaucoup de valeurs de x, petites et grandes, positives et négatives.

Exercice/P 6.6. Afin de calculer $\ln 2$ on pourrait théoriquement se servir de la série $\ln 2 = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{1}{k}$. Écrire un programme qui calcule $s_n = \sum_{k=1}^{n} (-1)^k \frac{1}{k}$. Majorer le reste r_n puis donner un encadrement de $\ln 2$. Peut-on ainsi calculer $\ln 2$ à 10^{-6} près ? à 10^{-12} près ?

☞ *On a tout intérêt à choisir, si possible, une série qui converge rapidement.* ✍

Exercice/P 6.7. Dans l'exemple précédent, rien ne nous empêche de calculer $\ln 2$ par une série mieux adaptée. Pour $|x| < 1$ on a les développements en série

$$\begin{aligned} \ln(1+x) &= +x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots \\ \ln(1-x) &= -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \ldots \\ \Longrightarrow \quad \ln\left(\frac{1+x}{1-x}\right) &= 2\left(x + \frac{x^3}{3} + \frac{x^5}{5} + \ldots\right) \end{aligned}$$

Pour $x = \frac{1}{3}$ on obtient ainsi $\ln 2 = \sum_{k=1}^{\infty} \frac{2}{2k-1} 3^{1-2k}$. Écrire un programme qui calcule la somme partielle $s_n = \sum_{k=1}^{n} \frac{2}{2k-1} 3^{1-2k}$. Majorer le reste r_n puis donner des encadrements de $\ln 2$. Peut-on ainsi calculer $\ln 2$ à 10^{-10} près ? à 10^{-100} près ?

Numerical data piles up and numerical programs grow ever more ambitious and complicated while their users become, on average, far less knowlegeable about numerical error-analysis, though no less clever than their predecessors about subjects they care to learn. Consequently numerical anomalies go mostly unobserved or, if observed, routinely misdiagnosed. Fortunately most of them don't matter. Most computations don't matter.
W. Kahan, *How futile are mindless assessments of roundoff in floating-point computation?*

PROJET XV

Dérivation numérique et extrapolation de Richardson

Avant de traiter l'intégration numérique, le présent projet discute la dérivation numérique, souvent plus facile. Le développement qui suit introduit une technique fondamentale : l'extrapolation de Richardson. Elle nous servira plus tard pour l'intégration dans la méthode de Romberg.

Les expériences numériques du chapitre XV (exercice 5.4) ont déjà montré que le calcul numérique d'une dérivée $\lim_{h\to 0} \frac{f(a+h)-f(a)}{h}$ est loin d'être trivial : si les valeurs $f(a)$ et $f(a+h)$ sont calculées avec n chiffres significatifs, il est assez difficile d'en déduire une valeur approchée de $f'(a)$ avec n chiffres significatifs. Dans de telles situations, l'extrapolation de Richardson peut remédier à la perte de précision.

Sommaire

1. Convergence linéaire vs quadratique

Nous supposons que $f :]a-\varepsilon, a+\varepsilon[\to \mathbb{R}$ est de classe C^{n+1}, avec n aussi grand que nécessaire. Nous avons en particulier la formule de Taylor (avec reste de Lagrange) :

$$f(a+h) = \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!} h^k + \frac{f^{(n+1)}(\xi)}{(n+1)!} h^{n+1} \quad \text{avec} \quad \xi = a+\theta h \text{ et } \theta \in]0,1[.$$

Du coté numérique, nous disposons d'une implémentation `Flo f( Flo x )` qui calcule f avec une précision satisfaisante, disons de 15 décimales, en utilisant un type flottant que l'on nommera `Flo`. À noter toutefois qu'un tel calcul peut être coûteux ; on essaiera de s'en servir avec modération.

Par contre, nous n'avons aucune connaissance des dérivées $f', f'', \ldots$, on suppose seulement leur existence. Notre but est de développer une méthode efficace pour calculer une valeur approchée de $f'(a)$ à partir de très peu de valeurs de f dans un voisinage de a.

Exercice/M 1.1. Dans un premier temps, on pourrait prendre $\phi_1(h) := \frac{f(a+h)-f(a)}{h}$ comme valeur approchée de la dérivée exacte $f'(a)$. Vérifier l'estimation d'erreur $\phi_1(h) = f'(a) + \frac{1}{2} f''(\xi) h$. Pour $h \to 0$ la convergence $\phi_1(h) \to f'(a)$ est donc au moins *linéaire*, souvent abrégé $\phi_1(h) = f'(a) + O(h)$.

Un peu plus raffinée, considérons la valeur approchée $\phi_2(h) := \frac{f(a+h)-f(a-h)}{2h}$. Vérifier l'estimation d'erreur $\phi_2(h) = f'(a) + \frac{1}{3!} f'''(\xi) h^2$. Pour $h \to 0$ la convergence $\phi_2(h) \to f'(a)$ est donc au moins *quadratique*, souvent abrégé $\phi_1(h) = f'(a) + O(h^2)$. Ceci explique l'intérêt de cette deuxième approche.

Exercice/P 1.2. Comparer les deux approches sur un exemple numérique, disons $f : \mathbb{R} \to \mathbb{R}$ donnée par $f(x) = \sin(x)$. Calculer des valeurs approchées de f' en $a = 1$ par les deux méthodes ci-dessus pour $h = 2^{-k}$ avec $k = 1, 2, 3, \ldots, 50$. On pourra commencer l'implémentation par

```
typedef double Flo;
const Flo a= 1.0;
```

Bien sûr on connaît la valeur $f'(a) = \cos(1) \approx 0.54030230586$; pour information faites afficher `cos(a)` de la bibliothèque `cmath`. Quelle est la précision maximale que l'on puisse atteindre avec la méthode linéaire ? avec la méthode quadratique ? Quelles sont les valeurs optimales pour k, environ ? Expliquer pourquoi il faut choisir k ni trop petit ni trop grand.

2. Convergence d'ordre 4

Exercice/M 2.1. Nous nous proposons d'obtenir une convergence encore plus rapide. Reprenons la dérivée approchée $\phi_2(h) = \frac{f(a+h)-f(a-h)}{2h}$. En supposant f de classe C^{2n+3}, montrer que

$$\phi_2(h) = f'(a) + a_2h^2 + a_4h^4 + \dots a_{2n}h^{2n} + \frac{f^{(2n+3)}(\xi)}{(2n+3)!}h^{2n+2}.$$

Vérifier que pour tout $\alpha \in \mathbb{R}$ la fonction $\phi_4(h) := \phi_2(h) + \alpha\big[\phi_2(h) - \phi_2(2h)\big]$ converge vers $f'(a)$ pour $h \to 0$. Déterminer α de sorte que la convergence soit d'ordre 4.

Exercice/P 2.2. Implémenter la méthode ci-dessus pour notre exemple $f(x) = \sin(x)$ et calculer ainsi des valeurs approchées de f' en $a = 1$ en calculant $\phi_2(h)$ puis $\phi_4(h)$ pour pour $h = 2^{-k}$ avec $k = 1,2,3,\dots,50$. Veillez à ne pas évaluer la fonction f inutilement, en réutilisant les valeurs de ϕ_2 déjà calculées. Quelle est la précision maximale que l'on puisse atteindre ? Quelle est la valeur optimale pour k, environ ? En quoi la méthode est-elle intéressante ?

3. Extrapolation de Richardson

En généralisant ce qui précède, supposons que $\phi\colon]0,\varepsilon] \to \mathbb{R}$ est une fonction que l'on sait calculer pour $\varepsilon 2^{-k}$ avec $k = 0,1,2,\dots$. Afin d'en déduire une valeur approchée de $\lim_{h\to 0}\phi(h)$, on supposera que ϕ admet un développement limité

$$\phi(h) = a_0 + a_2h^2 + a_4h^4 + \dots + a_{2n}h^{2n} + O(h^{2n+2})$$

avec des constantes $a_0, a_2, a_4, \dots, a_{2n} \in \mathbb{R}$ que nous ignorons. Pour $k = 0,1,2,\dots$ on pose $d_k^0 := \phi(\varepsilon 2^{-k})$; c'est la première ligne du schéma suivant. La convergence $d_k^0 \to \phi(0)$ est quadratique. Afin d'obtenir une convergence plus rapide, on calcule la ligne $i = 1,2,\dots,n$ par $d_k^i := d_{k+1}^{i-1} + \frac{1}{4^i-1}\big[d_{k+1}^{i-1} - d_k^{i-1}\big]$.

$$\begin{array}{cccccc}
d_0^0 & d_1^0 & d_2^0 & \dots & d_n^0 & \to \ \phi(0) \\
d_0^1 & d_1^1 & \dots & d_{n-1}^1 & & \to \ \phi(0) \\
d_0^2 & \dots & d_{n-2}^2 & & & \to \ \phi(0) \\
\vdots & & & & & \\
d_0^n & & & & & \to \ \phi(0)
\end{array}$$

Exercice/M 3.1. Vérifier que $d_k^i \to \phi(0)$ pour $k \to \infty$ et déterminer l'ordre de la convergence.

Exercice/P 3.2. Nous supposons de disposer d'une implémentation `Flo phi( Flo h )`. Écrire une fonction `Flo richardson( Flo eps, int n )` qui met en œuvre l'algorithme développé ci-dessus pour calculer $\lim_{h\to 0}\phi(h)$.
Indication. — Une méthode éprouvée est de ne stocker que la diagonale $d_0^k, \dots, d_{k-2}^2, d_{k-1}^1, d_k^0$. Explicitement : pour $k = 0$ on calcule d_0^0 et le stocke dans un vecteur ; pour $k = 1$ on ajoute d_1^0 et remplace d_0^0 par d_0^1 ; pour $k = 2$ on ajoute d_2^0, remplace d_1^0 par d_1^1, puis d_0^1 par d_0^2. Ainsi le vecteur s'agrandit chaque fois d'un élément, les améliorations se propagent de la fin vers le début, et la « meilleur approximation » d_0^k est toujours stockée en position 0. C'est la valeur d_0^n qui est finalement renvoyée par la fonction. (Pour des tests vous pouvez faire afficher d_0^k dans chaque itération.)

Exercice/P 3.3. Appliquer votre fonction `richardson` à l'exemple $f(x) = \sin(x)$ et calculer ainsi des valeurs approchées de f' en $a = 1$ pour $\varepsilon = 1$ et $n = 1,\dots,20$. Peut-on choisir n trop petit ? trop grand ? Quelles sont les valeurs acceptables pour n, environ ? Cette méthode vous semble-t-elle assez robuste ?

Exercice 3.4. Vérifier que $\pi = \lim_{k\to\infty} 2^k \sin(2^{-k}\pi)$. Interpréter géométriquement ces valeurs en comparant avec la circonférence d'un 2^k-gone régulier inscrit dans un cercle de rayon 1. Pour calculer $a_k = \sin(2^{-k}\pi)$ numériquement on n'utilisera pas de valeur approchée de π, ni la fonction sin, mais la formule de récurrence $1 - 2a_{k+1}^2 = \sqrt{1-a_k^2}$ en commençant par $a_1 = 1$. (La vérifier.) Calculer ainsi des valeurs approchées de $2^k \sin(2^{-k}\pi)$ pour $k = 1,2,3,\dots$ et juger si la convergence est satisfaisante. Pourquoi cette formule de récurrence est-elle mal adaptée au calcul numérique ? À quelle précision pouvez-vous ainsi approcher π ? Appliquer la méthode de Richardson pour calculer une meilleur approximation de π.

If you think it's simple, then
you have misunderstood the problem.
Bjarne Stroustrup

CHAPITRE XVI

Calcul arrondi fiable et arithmétique d'intervalles

Objectif. Dans ce chapitre notre but est de comprendre les éléments du calcul arrondi fiable.
► L'arrondi permet de rendre certains calculs faisables (ou bien plus efficaces) sur ordinateur.
► En général le résultat ainsi calculé sera erroné ; il est donc nécessaire de majorer l'erreur commise.
Cette erreur provient de la méthode (approximation) et de l'arithmétique utilisée (erreurs d'arrondis). Dans cette problématique les arrondis dirigés peuvent donner des informations précieuses :
► L'arrondi vers $-\infty$ donne un résultat approché a qui fournit une minoration fiable.
► L'arrondi vers $+\infty$ donne un résultat approché b qui fournit une majoration fiable.
Les deux bornes ensemble fournissent un encadrement $[a,b]$ du résultat exact $r \in \mathbb{R}$. L'objectif d'une bonne implémentation est d'abord de garantir l'encadrement $a \leq r \leq b$, puis de minimiser l'écart $|b-a|$, et finalement d'économiser les ressources (temps et mémoire).

Le problème fondamental du calcul arrondi. Lors d'un calcul arrondi les valeurs approchées calculées peuvent s'éloigner de la valeur exacte cherchée. Comme à la fin nous ne disposons que de la valeur calculée (erronée) et non de la valeur cherchée (exacte), il nous faut un moyen de contrôler la marge d'erreur. Dans le pire des cas une « explosion d'erreur » peut rendre le calcul inutilisable, car la valeur calculée n'a plus aucun rapport avec la valeur cherchée. Comment savoir si un résultat est acceptable ou non ? Plusieurs stratégies sont imaginables pour assurer (ou au moins tester) la fiabilité des résultats calculés :

(1) Dans certains cas on peut éviter les arrondis en faisant un calcul *exact* dans un anneau effectif comme $\mathbb{Z}$ ou $\mathbb{Q}$ ou $\mathbb{Q}[\sqrt{2}]$ etc. On se ramène ainsi à une situation entièrement algébrique, où tout élément peut être représenté de manière exacte et toute opération peut être exécutée sans arrondi. Si le problème en question s'y prête et que le calcul n'est pas trop coûteux, c'est la solution la plus élégante et la plus sûre.

(2) En implémentant un calcul numérique, on est tenté de calculer avec le type `double`, disons, comme si c'était un calcul exact. Cette stratégie naïve est la plus répandue, mais aussi la plus périlleuse. Un calcul maladroit peut provoquer la perte totale de chiffres significatifs. Même un calcul habile ne fournit aucune indication quant à la marge d'erreur.

(3) Par mesure de sécurité on peut calculer avec beaucoup plus de chiffres que nécessaire, disons 40 décimales pour ne retenir que 20 chiffres significatifs à la fin du calcul. C'est un peu moins naïf, mais ne protège pas contre les catastrophes. Bien que les chances soient meilleures, on n'a toujours aucune garantie de correction

(4) On peut calculer avec ℓ chiffres, puis refaire le calcul avec 2ℓ chiffres de précision. On n'acceptera que les chiffres qui coïncident, en espérant que ce sont en quelque sorte des chiffres « stables », donc « corrects ». `(` Un programmeur craintif recalculera avec 3ℓ chiffres. `;-)` Cette recette marche souvent, mais on peut bien sûr tomber sur de méchants contre-exemples.

(5) Pour avoir des encadrements prouvables, il ne reste que le contrôle minutieux des arrondis. Typiquement on fait deux calculs séparés afin d'établir une minoration puis une majoration. Ceci demande une implémentation plus soigneuse, mais en contrepartie le résultat calculé est toujours honnête. Quand le calcul tourne mal, au moins on le verra ouvertement.

Suivant la structure du problème et l'importance du résultat on choisira une de ces stratégies, ou une des nombreuses variantes. Il est clair que l'on programmerait différemment les exercices de ce chapitre s'ils servaient à piloter une fusée. (Espérons, au moins.) Ce sont la méthode choisie et le soin apporté à l'implémentation qui détermineront la fiabilité du résultat.

Quels son les avantages d'un calcul fiable ? Soulignons que même dans un calcul arrondi fiable il y aura toujours des erreurs d'arrondi. C'est un phénomène caractéristique et inévitable : ce calcul fut inventé pour rendre les calculs plus efficaces, au prix des arrondis. Le but n'est donc pas de supprimer les arrondis, mais de contrôler la marge d'erreur dans le résultat final.

Un calcul fiable fournit un encadrement $[a,b]$ du résultat exact : cet encadrement est prouvable et met en évidence la marge d'erreur. Quoi qu'il arrive, nous obtiendrons toujours un résultat avec garantie de correction ! Dans le pire des cas l'intervalle de l'encadrement peut être trop grossier, mais même ce résultat décevant contient une information importante : il signale que le calcul a mal tourné, et qu'il faut traiter le résultat avec prudence. Si la précision atteinte est jugée insuffisante, il faudra refaire un calcul plus raffiné. En tout cas il sera malhonnête de prétendre une précision supérieure à l'intervalle établi par le calcul.

Si la précision est suffisante, on extrait une valeur approchée avec une précision garantie. Pour cette raison le calcul fiable est aussi appelé *calcul auto-certifiant* (terme publicitaire).

Comment s'y prendre ? On peut bien sûr faire appel à une bibliothèque toute faite, telle que la GMP (*GNU multiple precision library*) et ses extensions MPFR (*multiple-precision floating-point computations with correct rounding*) et MPFI (multiple-precision floating-point interval arithmetic). Vous êtes vivement invités à vous renseigner sur les sites respectifs `www.swox.com/gmp` et `www.mpfr.org` pour avoir une idée de ce qu'elles offrent. Ces informations sont également disponible en local, en tapant `info mpfr` et `info gmp C++` dans une ligne de commande.

De manière plus pédestre, nous nous proposons ici d'illustrer le principe par une implémentation « faite maison ». Comme bénéfice collatéral ceci nous donne l'occasion d'expliquer les nombres flottants et l'arithmétique arrondie en tout détail. On complète ainsi notre introduction aux nombres flottants esquissée au chapitre précédent.

Ayant implémentée *l'arithmétique arrondie fiable*, on peut pousser ce concept un peu plus loin. Afin de rendre les objets plus intuitifs et plus agréables à manipuler, nous implémentons *l'arithmétique d'intervalles* qui calcule majoration et minoration parallèlement. Nous arrivons ainsi à un calcul numérique, sous une forme naturelle et une écriture commode, qui fournit non seulement une valeur approchée mais en même temps la marge d'erreur sous forme d'encadrement.

Pour en savoir plus. Le calcul arrondi fiable et l'arithmétique d'intervalles ne datent pas d'hier, mais ils attirent de plus en plus d'attention ces dernières années, surtout dans les domaines sensibles qui exigent un très haut niveau de sécurité et une garantie de précision. L'arithmétique d'intervalles n'est pas le remède à tous les maux numériques, mais elle offre souvent une démarche sûre et simple.

Pour en savoir d'avantage, consultez le site `www.cs.utep.edu/interval-comp`, qui rassemble de nombreux liens utiles concernant l'arithmétique d'intervalles :

(1) Pour le coté vulgarisation scientifique, lire l'excellent survol de B. Hayes, *A lucid interval*, `www.cs.utep.edu/interval-comp/hayes.pdf`.

(2) Si cela vous chante, vous pouvez aussi regarder — puis analyser — un film destiné aux étudiants, `www-sop.inria.fr/coprin/logiciels/ALIAS/Movie/movie_undergraduate.mpg`.

(3) Pour une discussion provocatrice lire W. Kahan, *How futile are mindless assessments of roundoff in floating-point computation ?* `www.cs.berkeley.edu/~wkahan/Mindless.pdf`.

Sommaire

1. Deux types d'erreurs inévitables

1.1. Erreurs de discrétisation. Afin de modéliser le calcul numérique dans $\mathbb{R}$ nous sommes obligés de nous restreindre à un sous-ensemble $R \subset \mathbb{R}$ de nombres exactement représentables sur machine. À titre d'exemple, R peut être constitué des nombres flottants de longueur ℓ. Pour simplifier nous autorisons des exposants $e \in \mathbb{Z}$. (Nous sous-entendons que l'exposant e ne sera pas trop grand, mais nous n'imposons pas de limites a priori.) Ainsi l'ensemble des nombres machine est donné par

$$R = \{0\} \cup \left\{ m \cdot 2^e \;\middle|\; m, e \in \mathbb{Z} \text{ avec } -2^\ell < m < 2^\ell \right\}. \tag{1}$$

Rappelons que dans l'écriture $m \cdot 2^e$ on appelle m la *mantisse* et e *l'exposant*. En fixant la longueur de la mantisse à ℓ bits on restreint la précision des valeurs ainsi représentables, mais en contrepartie on diminue aussi le coût des calculs (en temps et en mémoire).

Contrairement aux nombres réels $\mathbb{R}$ qui modélisent des phénomènes continus, le modèle informatique des nombres flottants R est forcément discrétisé. Néanmoins on est obligé d'approcher tout nombre réel $x \in \mathbb{R}$ par un nombre flottant $\hat{x} \in R$. Pour presque tout nombre réel x nous avons $x \neq \hat{x}$, et l'application $x \mapsto \hat{x}$ introduit donc une *erreur d'arrondi* $|\hat{x} - x|$, aussi appelé *erreur de discrétisation*.

La différence $|\hat{x} - x|$ est *l'erreur absolue*. Il est souvent plus informatif de considérer *l'erreur relative* $\varepsilon = |\frac{\hat{x}-x}{x}|$. En arrondissant au plus proche on obtient $\varepsilon \leq 2^{-\ell}$, autrement dit la différence entre la valeur réelle $x \in \mathbb{R}$ et son approximation discrète $\hat{x} \in R$ s'exprime comme $\hat{x} = x(1+\delta)$ avec un facteur de perturbation δ vérifiant $|\delta| \leq 2^{-\ell}$. (Voir le chap. XV, §4.)

☞ *La longueur ℓ de la mantisse détermine la granularité de la discrétisation.* ✍

1.2. Erreurs de calcul. Nous résumons la différence entre un calcul exact dans $\mathbb{R}$ et un calcul approché dans $R \subset \mathbb{R}$ par le diagramme suivant. L'approximation $\mathbb{R}^n \to R^n$ envoie $(x_1, \ldots, x_n)$ sur $(\hat{x}_1, \ldots, \hat{x}_n)$ où chaque coefficient $x_k \in \mathbb{R}$ est représenté par une valeur approchée $\hat{x}_k \in R$. De même, $\hat{f} \colon R^n \to R$ est une fonction calculable sur machine qui ne fournit qu'une approximation de la fonction réelle $f \colon \mathbb{R}^n \to \mathbb{R}$.

$$\begin{array}{lccl} \text{nombres réels} & x \in \mathbb{R}^n & \xrightarrow{\;f\;} & \mathbb{R} \ni f(x) \qquad \text{résultat exact mais inconnu} \\ & \text{arrondi}\downarrow & & \uparrow\text{inclusion} \\ \text{nombres machine} & \hat{x} \in R^n & \xrightarrow{\;\hat{f}\;} & R \ni \hat{f}(\hat{x}) \qquad \text{résultat calculé mais erroné} \end{array}$$

☞ *En général ce diagramme ne commute pas, c'est-à-dire $f(x) \neq \hat{f}(\hat{x})$.*
Seule la connaissance de la marge d'erreur donne autorité au résultat calculé $\hat{f}(\hat{x})$. ✍

Plus explicitement, l'écart entre le résultat exact et la valeur approchée calculée vaut

$$f(x) - \hat{f}(\hat{x}) = \big[f(x) - f(\hat{x})\big] + \big[f(\hat{x}) - \hat{f}(\hat{x})\big].$$

On voit apparaître la somme de deux erreurs :

(1) La première erreur est due à la discrétisation de x et dépend des nombres machine utilisés : un nombre réel n'est stocké qu'avec un nombre limité de chiffres significatifs (disons ℓ bits).

(2) La seconde erreur est due à la méthode utilisée pour approcher f. On a tout intérêt à utiliser une approximation $\hat{f}$ qui soit aussi proche de f que possible, mais il restera souvent un écart.

Soulignons à nouveau qu'il est indispensable de majorer l'erreur totale. Sinon il est inutile, voire nuisible, de se lancer dans le calcul. Pour certains calculs très simples il est relativement facile de majorer l'erreur. Pour des calculs réalistes ceci devient de plus en plus dur : une majoration fine est souvent inextricable, et des majorations grossières ne donnent souvent pas de résultat satisfaisant. Pour cette raison nous développons dans la suite les éléments d'un calcul arrondi fiable, aussi appelé *calcul auto-certifiant* :

☞ *On ne peut espérer $\hat{f}(\hat{x}) = f(x)$, mais on peut garantir $\hat{f}(\hat{x}) \leq f(x)$ ou $f(x) \leq \hat{f}(\hat{x})$,*
respectivement, et ainsi encadrer la valeur exacte par deux valeurs approchées. ✍

2. Calcul arrondi fiable

Ce qui gâche les calculs avec le type `double` n'est souvent pas la précision insuffisante : la longueur de la mantisse est de 53 bits, soit 16 décimales environ, ce qui est suffisant pour beaucoup d'applications. Le problème est surtout la difficulté de contrôler les erreurs d'arrondi. Ce paragraphe essaie d'apporter quelques réponses à ce problème et de développer les éléments d'un calcul arrondi fiable.

2.1. Arrondis dirigés. Supposons que durant un long calcul numérique chaque résultat intermédiaire est arrondi vers le nombre machine le plus proche. A priori c'est une bonne idée car on minimise localement l'erreur de chaque opération élémentaire. Mais ce n'est vrai que *localement*. L'écart accumulé peut être assez grand, sans que l'utilisateur soit averti. (Revoir les pièges discutés au chapitre XV.) Ce qu'il faut assurer est une relation fiable entre le résultat calculé (mais erroné) et le résultat exact (mais inconnu).

☞ *À première vue l'arrondi vers le plus proche semble toujours la meilleure option. C'est le mode le plus courant, mais il offre le moins de contrôle sur le résultat.* ✍

Pour les nombres réels $\mathbb{R}$, la relation fiable à garantir sera l'ordre : tout nombre réel x peut être encadré par deux nombres machine $\underline{x},\overline{x} \in R$ de sorte que $\underline{x} \le x \le \overline{x}$. Si jamais $x \in R$, on pourra tomber sur $\underline{x} = x = \overline{x}$, mais cela restera une exception rare. En général on a une inégalité stricte $\underline{x} < x < \overline{x}$ et il faut décider avec laquelle des deux approximations on continuera le calcul.

☞ *En choisissant judicieusement les modes d'arrondi on peut garantir un encadrement fiable du résultat final.* ✍

Définition 2.1 (modes d'arrondi)**.** Nous supposons que les nombres machine $R \subset \mathbb{R}$ forment un sous-ensemble fermé dans la topologie de $\mathbb{R}$. Nous exigeons aussi que R contienne au moins $0, \pm 1, \pm 2$, qu'il soit symétrique ($R = -R$) et qu'il « recouvre » tout $\mathbb{R}$ dans le sens que $\sup R = +\infty$ et $\inf R = -\infty$. Toutes ces exigences sont satisfaites pour notre ensemble (1).

Étant donné un nombre réel $x \in \mathbb{R}$, on ne peut en général pas le représenter de manière exacte par un nombre machine. On prendra donc une valeur approchée $\hat{x} \in R$, le meilleur choix étant un de ses deux « voisins » dans R. Au moins cinq modes d'arrondi $\mathbb{R} \to R$ s'offrent à nous :

$\lfloor x \rfloor_R := \sup\{\underline{x} \in R \mid \underline{x} \le x\}$	arrondi vers $-\infty$, vers le bas	(RNDD = *round down*)
$\lceil x \rceil_R := \inf\{\overline{x} \in R \mid x \le \overline{x}\}$	arrondi vers $+\infty$, vers le haut	(RNDU = *round up*)
$[x]_R := \begin{cases} \lfloor x \rfloor_R & \text{si } x \ge 0 \\ \lceil x \rceil_R & \text{si } x \le 0 \end{cases}$	arrondi vers zéro	(RNDZ = *round to zero*)
$]x[_R := \begin{cases} \lceil x \rceil_R & \text{si } x \ge 0 \\ \lfloor x \rfloor_R & \text{si } x \le 0 \end{cases}$	arrondi vers $\pm\infty$	(RNDI = *round to infinity*)
$\lfloor x \rceil_R := \begin{cases} \lfloor x \rfloor_R & \text{si } x < m \\ \lceil x \rceil_R & \text{si } x > m \end{cases}$	arrondi vers le plus proche	(RNDN = *round to nearest*)

Pour l'arrondi vers le plus proche on pose $m = \frac{1}{2}(\lfloor x \rfloor_R + \lceil x \rceil_R)$. En cas d'égalité $x = m$ il faut fixer une règle d'arbitrage. Pour les nombres flottants (1) on convient de choisir celui dont la mantisse est paire, i.e. finit par un zéro dans son développement binaire (« arrondi pair »).

Exercice/M 2.2. Vérifier que pour $R = \mathbb{Z}$ on obtient les définitions usuelles de $\lfloor x \rfloor$, $\lceil x \rceil$, $[x]$, $\lfloor x \rceil$.

(1) Plus généralement, si $R \subset \mathbb{R}$ est un sous-ensemble *fermé* dans la topologie de $\mathbb{R}$, montrer que $\lfloor x \rfloor_R$ et $\lceil x \rceil_R$ sont toujours des éléments de R, quelque soit $x \in \mathbb{R}$. Les valeurs arrondies sont donc exactement représentables, ce qui est la propriété essentielle.

(2) Discuter l'ensemble $R = \mathbb{Q}$ qui n'est pas fermé dans $\mathbb{R}$. Que valent $\lfloor x \rfloor_{\mathbb{Q}}$ et $\lceil x \rceil_{\mathbb{Q}}$? Les valeurs ainsi « arrondies », sont-elles représentables sur machine ? Expliquer pourquoi cette approche, tellement utile en analyse et algèbre, ne convient pas pour le calcul numérique.

2.2. Arithmétique arrondie. Pour les nombres réels nous disposons des opérations élémentaires $+,-,*,/$ dont nous pouvons restreindre le domaine d'application à l'ensemble R :

$$+\colon R\times R\to\mathbb{R} \qquad *\colon R\times R\to\mathbb{R}$$
$$-\colon R\times R\to\mathbb{R} \qquad /\colon R\times R^*\to\mathbb{R}$$

Remarque. — Nous ne pouvons pas espérer que les images soient incluses dans R. Si jamais on avait un modèle de nombres machine R tel que $+,-,*,/$ prenaient toutes leurs valeurs dans R, alors R contiendrait le corps $\mathbb{Q}$ des nombres rationnels. Ce serait donc un sous-ensemble dense de $\mathbb{R}$, aussi inconvénient que $\mathbb{Q}$ pour le calcul arrondi. (Voir l'exercice précédent.) Ceci montre que le problème des opérations $+,-,*,/$ est inhérent au calcul arrondi, et non seulement un artefact d'une implémentation maladroite.

On fait donc appel aux arrondis introduits dans la définition précédente. On implémente l'addition, par exemple, comme l'addition exacte $+\colon R\times R\to\mathbb{R}$ suivie de l'arrondi spécifié $\mathbb{R}\to R$. Ainsi l'addition $+\colon\mathbb{R}\times\mathbb{R}\to\mathbb{R}$ est modélisée non par *une* mais par *cinq* opérations pour le calcul approché :

$$\lfloor+\rfloor,\ \lceil+\rceil,\ [+],\ ]+[,\ \lfloor+\rceil\colon R\times R\to R$$

Regardons plus généralement le calcul d'une fonction $f\colon\mathbb{R}\to\mathbb{R}$, comme $\exp\colon\mathbb{R}\to\mathbb{R}$. À nouveau on ne peut pas espérer que $f(R)\subset R$, il faut donc arrondir. La façon la plus honnête sera d'implémenter deux approximations $\lfloor f\rfloor,\lceil f\rceil\colon R\to R$ qui garantissent pour tout $x\in R$ un encadrement

$$\lfloor f\rfloor(x) \quad\leq\quad f(x) \quad\leq\quad \lceil f\rceil(x).$$

Étant donnée f, il nous reste évidemment le problème d'implémenter correctement $\lfloor f\rfloor$ et $\lceil f\rceil$ de manière efficace et avec un écart $\lceil f\rceil-\lfloor f\rfloor$ aussi petit que possible. C'est souvent faisable et permettra de *garantir* que l'encadrement final sera correct et satisfaisant.

2.3. Une implémentation « faite maison ». Après la discussion des modes d'arrondi, passons à une implémentation concrète ! Le programme XVI.1 donne une esquisse de la classe `RReal` modélisant des *nombres flottants fiables*, ou *reliably rounded real number* en anglais. Vous trouvez le code source complet dans le fichier `rreal.cc`, ainsi qu'un exemple d'utilisation dans `rreal-test.cc`.

L'idée est d'implémenter des nombres flottants en précision ℓ arbitraire. La présentation (1) a l'avantage de se baser uniquement sur les nombres entiers : les flottants sont stockés sous la forme $m\cdot 2^e$ avec deux entiers m (la mantisse) et e (l'exposant). Afin de pouvoir choisir la longueur de la mantisse (éventuellement très grande) il nous faut une implémentation de grands entiers. Nous nous servirons ici de la classe `mpz_class` de la bibliothèque GMP, qui fournit toutes les opérations nécessaires.

2.4. Comment arrondir ? On aura besoin d'une division arrondie : pour deux entiers a et $b\neq 0$ la fonction `eudiv(a,b,mode)` effectue une division euclidienne $a=qb+r$ avec $|r|<|b|$. On a donc $q=\lfloor\frac{a}{b}\rfloor_{\mathbb{Z}}$ ou $q=\lceil\frac{a}{b}\rceil_{\mathbb{Z}}$, et le quotient renvoyé est choisi suivant le mode d'arrondi spécifié.

Comme les entiers sont stockés par leur développement binaire, la division euclidienne par 2^e est particulièrement efficace. (Rappeler pourquoi.) La fonction `eudiv2(a,e,mode)` effectue cette division de a par 2^e telle que le quotient soit arrondi comme spécifié. Ceci permet de réduire une mantisse `m` à une longueur ℓ, simplement en posant `m= eudiv2(m,exces,mode)`. C'est justement l'objectif de la fonction `arrondir(mode,len)`.

Le constructeur `RReal(man,exp)` et la fonction `set(man,exp)` permettent de définir aisément un nombre de la forme $m\cdot 2^e$. Si en plus les paramètres `mode` et `len` sont spécifiés, on réduit la longueur de la mantisse à `len` bits en appliquant le mode d'arrondi spécifié. Si ces paramètres ne sont pas spécifiés, on prend les valeurs par défaut `mode= stdMode` et `len= stdLen`.

Finalement, la fonction `rallonger(len)` rajoute des bits zéro à la mantisse, alors que la fonction `remplir(len)` assure que la mantisse soit de longueur $\geq$ `len`.

2.5. Comment multiplier ? L'arithmétique arrondie implémente un principe net et simple : on effectue toute opération élémentaire d'abord de manière exacte, puis on arrondit la mantisse à la longueur prescrite. La multiplication de deux nombres à virgule flottante est particulièrement simple, voir le programme XVI.1. C'est presque trivial à un détail important près : la fonction `set` employée ici effectue automatiquement l'arrondi nécessaire. (Pourquoi est-ce important ?)

Programme XVI.1 La classe `RReal` — nombres flottants avec arrondi dirigé

```
// Quelques types et fonctions auxiliaires
enum Mode { RNDD, RNDU, RNDZ, RNDI, RNDN }; // modes d'arrondi
typedef mpz_class Man;      // type pour la mantisse (grand entier)
typedef mpz_class Exp;      // type pour l'exposant  (grand entier)
typedef int Len;            // type pour la longueur (petit entier)
Len length( const Man& a ); // longueur d'un entier = nombre de bits

// Division euclidienne de a par b, puis de a par 2^e, suivant le mode d'arrondi
Man eudiv( const Man& a, const Man& b, Mode mode= stdMode );
Man eudiv2( const Man& a, Exp exp, Mode mode= stdMode );

// La classe RReal implémente des nombres flottants fiables
class RReal
{
public:
  Man mantisse; // la mantisse
  Exp exposant; // l'exposant

  // Précision par défaut (variable statique, càd globale pour RReal)
  static Len stdLen;
  static Len len() { return stdLen; };
  static Len len( Len l ) { Len old= stdLen; stdLen= max(l,2l); return old; }

  // Mode d'arrondi par défaut (variable statique, càd globale pour RReal)
  static Mode stdMode;
  static Mode mode() { return stdMode; }
  static Mode mode( Mode m ) { Mode old= stdMode; stdMode= m; return old; }

  // La fonction suivante effectue l'arrondi comme spécifié par mode et len
  void arrondir( Mode mode= stdMode, Len len= stdLen )
    {
      if ( mantisse == 0 ) { exposant= 0; return; }; // cas particulier
      Len exces= length() - max( len, 2l );          // bits de trop
      if ( exces <= 0 ) return;                      // rien à raccourcir
      mantisse= eudiv2( mantisse, exces, mode );     // supprimer l'excès
      exposant+= exces;                              // compenser l'exposant
    }

  // Constructeur à partir d'une pair (mantisse, exposant) suivi d'arrondi
  RReal( Man man=0, Exp exp=0, Mode mode= stdMode, Len len= stdLen )
    : mantisse(man), exposant(exp) { arrondir( mode, len ); }

  // Affectation d'une pair (mantisse, exposant) suivi d'arrondi
  RReal& set( Man man=0, Exp exp=0, Mode mode= stdMode, Len len= stdLen )
    { mantisse= man; exposant= exp; arrondir( mode, len ); return *this; }

  // La fonction  rallonger  rajoute des bits zéro à la mantisse
  void rallonger( Len len )
    { if ( len > 0 ) { mantisse <<= len; exposant -= len; }; }

  // La fonction  remplir  assure que la mantisse soit de longueur >= len.
  void remplir( Len len= stdLen )
    { rallonger( len - length() ); }

  // Multiplication : multiplier les mantisses et additionner les exposants
  RReal& mul( const RReal& a, const RReal& b, Mode mode= stdMode, Len len= stdLen )
    { return set( a.mantisse * b.mantisse, a.exposant + b.exposant, mode, len ); }
};

// Multiplication sous forme d'opérateur, créant un objet temporaire
RReal operator* ( const RReal& a, const RReal& b )
{ RReal c; c.mul(a,b); return c; }
```

2.6. Comment diviser ? *Attention.* — La tentative suivante est vouée à l'échec :

```
RReal operator/ ( const RReal& a, const RReal& b )
{ return RReal( a.mantisse / b.mantisse, a.exposant - b.exposant ); }
```

Il faut d'abord rallonger la mantisse de `a` pour obtenir un quotient de précision suffisante ; ensuite `eudiv(a,b,mode)` calcule le quotient entier suivant le mode d'arrondi spécifié :

```
RReal& RReal::div( RReal a, const RReal& b, Mode mode= stdMode, Len len= stdLen )
{ a.remplir( len + b.length() );
  return set( eudiv(a.mantisse,b.mantisse,mode), a.exposant-b.exposant, mode, len ); }

RReal operator/ ( const RReal& a, const RReal& b )
{ RReal c; c.div(a,b); return c; }
```

2.7. Comment additionner ? L'addition est un peu moins évidente : elle nécessite d'abord d'égaliser les exposants. Plus explicitement, pour additionner $a = m_1 \cdot 2^{e_1}$ et $b = m_2 \cdot 2^{e_2}$ on détermine $e = \min\{e_1, e_2\}$ pour écrire $a = m'_1 \cdot 2^e$ et $b = m'_2 \cdot 2^e$ avec deux entiers $m'_1 = m_1 \cdot 2^{e_1 - e}$ et $m'_2 = m_2 \cdot 2^{e_2 - e}$. Sous cette forme on calcule ensuite $a + b = (m'_1 + m'_2) \cdot 2^e$, puis on arrondit le résultat le cas échéant.

```
RReal& RReal::add( RReal a, RReal b, Mode mode= stdMode, Len len= stdLen )
{ Exp minexp= min( a.exposant, b.exposant );
  a.rallonger( a.exposant - minexp );
  b.rallonger( b.exposant - minexp );
  return set( a.mantisse + b.mantisse, minexp, mode, len ); }

RReal operator+ ( const RReal& a, const RReal& b )
{ RReal c; c.add(a,b); return c; }
```

Exercice/P 2.3. Bien évidemment nos algorithmes pour le calcul arrondi sont susceptibles d'optimisation. Surtout l'égalisation des exposants est inutilement coûteuse si les deux exposants diffèrent de plus de `len` : de toute façon l'arrondi final ne gardera que les `len` premiers bits. Si vous voulez, vous pouvez optimiser l'addition de sorte que le décalage nécessaire n'excède jamais la longueur `len` souhaitée.

☞ Soustraction et comparaison sont similaires à l'addition. Voir le fichier `rreal.cc`, qui implémente aussi des opérateurs d'entrée-sortie en base 10. Munie de ces opérations de base, notre classe `RReal` est déjà opérationnelle, et les exemples suivants montrent quelques applications.

☞ Soulignons que l'implémentation de la classe `RReal` proposée ici a pour seul but d'illustrer le principe. Pour une application professionnelle il faudra encore la peaufiner ; typiquement on privilégiera une implémentation plus complète et plus soigneusement optimisée, comme MPFR et MPFI qui sont actuellement en cours de développement (voir `www.mpfr.org` ou `info mpfr` en local).

2.8. Newton-Héron revisité. Le programme `rreal-test.cc` calcule la racine d'un nombre réel par la méthode de Newton-Héron en profitant des arrondis dirigés (voir le programme XVI.2 ci-dessous). En l'occurrence on utilise le mode d'arrondi vers le haut globalement. On choisit une valeur initiale plus grande que la racine cherchée puis on applique l'itération de Newton tant que le résultat s'améliore. On peut ainsi garantir une majoration de la racine exacte cherchée.

Exercice/P 2.4. Implémenter la méthode de Newton-Héron pour calculer $\sqrt[n]{a}$ avec n quelconque :

```
RReal power( RReal a, int n, Mode mode= stdMode, Len len= stdLen );
RReal root( const RReal& a, int n, Mode mode= stdMode, Len len= stdLen );
```

Indication. — Dans la fonction `power` on devrait attraper le cas $n < 0$. Ensuite vous pouvez écrire une simple boucle ou bien une puissance dichotomique si vous envisagez appeler la puissance pour n assez grand. Les arrondis individuels sont tous donnés par le mode spécifié si $a \geq 0$; le cas $a < 0$ par contre est particulier et devrait être traité à part. Dans la fonction `root` on devrait attraper le cas $a < 0$, puis $n < 0$ ou $n = 0$ ou $n = 1$. Ensuite on peut itérer Newton-Héron comme dans le cas $n = 2$ ci-dessus. Pour chaque calcul intermédiaire veillez particulièrement à choisir le bon mode d'arrondi.

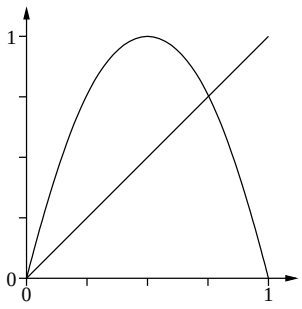
Programme XVI.2 Calcul fiable de la racine d'après Newton-Héron

```
RReal racine( const RReal& a, Mode mode= stdMode, Len len= stdLen )
{
  RReal u0, u1= (1+a)/2;          // Choix d'une valeur initiale.
  do {                            // L'itération de  u -> (u+a/u)/2 :
    u1.swap( u0 );                // échanger u0 et u1 (opération exacte),
    u1.div( a, u0, RNDU, len );   // u1 =  a / u0 , arrondi vers le haut,
    u1.add( u0, RNDU, len );      // u1 = u1 + u0 , arrondi vers le haut,
    u1.div( 2, RNDU, len );       // u1 = u1 / 2  , arrondi vers le haut.
  } while ( u1 < u0 );            // On continue tant que le résultat s'améliore.

  // On adapte la valeur finale selon le mode d'arrondi spécifié
  if( mode==RNDD || mode==RNDZ || mode==RNDN ) u1.div( a, u1, mode, len );
  return u1;
}
```

2.9. Instabilité numérique revisitée. L'itération de Newton-Héron pour calculer $\sqrt[n]{a}$ est, fort heureusement, numériquement stable : une petite perturbation de u_k (par une erreur d'arrondi, disons) restera petite, elle diminue même au cours des itérations suivantes. D'autres systèmes se comportent de manière contraire : des petites perturbations s'amplifient et « explosent » au cours des itérations suivantes. Ceci pose évidemment d'énormes problèmes pour le calcul numérique. On a déjà vu de tels exemples au chapitre XV, et nous allons regarder ici un exemple sur l'intervalle $[0,1]$.

Regardons une des fonctions les plus simples exhibant un comportement instable, $f\colon [0,1] \to [0,1]$, $f(x) = 4x(1-x)$. On peut interpréter l'itération $x \mapsto f(x)$ comme un *modèle de population*, quoique très simplifié. Penser à une population de bactéries contrainte à un environnement fixé. Toutes les heures on mesure $x_k \in [0,1]$, la quantité de bactéries par rapport à la population maximale. Pour x petit ($x \approx 0$) on a $f(x) \approx 4x$, donc la population quadruple à peu près. Quand elle devient trop grande, la nourriture commence à manquer et la croissance ralentit. Pour $x \geq \frac{3}{4}$ on a une situation de surpopulation (famine) et la population décroît. C'est un cas particulier de la *fonction logistique*, cf. `fr.wikipedia.org/wiki/Fonction_logistique`.



Exemple numérique. — Commençons l'itération par la valeur initiale $x_0 = 0.1$ disons. Avec notre petit programme `instable-naif.cc`, utilisant le type `double`, on trouve $x_{24} \approx 0.44165$ (un jour) puis $x_{48} \approx 0.03768$ (deux jours). Cette valeur serait donc notre prévision d'ici deux jours.

Phénomène d'instabilité. — Évidemment il faut s'attendre à certaines erreurs, déjà provenant de la valeur initiale x_0 (le « comptage » des bactéries), puis des arrondis lors du calcul itéré de $x_{k+1} = f(x_k)$. Est-ce que ces petites erreurs jouent un rôle ici ? Pour le tester, commençons l'itération par une valeur initiale légèrement perturbée, disons $x'_0 = 0,1000000001$. Le résultat est assez désillusionnant :

$$
\begin{array}{ll}
x_0 = 0,1000000000 & x'_0 = 0,1000000001 \\
x_1 = 0,3600000000 & x'_1 = 0,3600000003 \\
x_2 = 0,9216000000 & x'_2 = 0,9216000004 \\
\quad\cdots & \quad\cdots \\
x_{24} = 0,4416454191 & x'_{24} = 0,4388692524 \\
x_{25} = 0,9863789715 & x'_{25} = 0,9850521268 \\
x_{26} = 0,0537419842 & x'_{26} = 0,0588977372 \\
\quad\cdots & \quad\cdots \\
x_{48} = 0,0376796921 & x'_{48} = 0,7981824730 \\
x_{49} = 0,1450397316 & x'_{49} = 0,6443488512 \\
x_{50} = 0,4960128314 & x'_{50} = 0,9166536366
\end{array}
$$

Exercice/M 2.5. Expliquer pourquoi $|f'(x)| > 1$ implique qu'une petite perturbation de x est amplifiée. Expliquer qualitativement le comportement observé dans l'exemple précédent. La fonction f a deux points fixes : sont-ils stables ou instables ? Peut-on espérer une convergence vers un de ces points fixes ?

Exercice/P 2.6. Peut-on faire confiance en les valeurs x_{24} et x_{48} calculées par notre programme ? Comme chaque itération de la fonction f est susceptible d'introduire une petite erreur d'arrondi, c'est au moins douteux. Le type `double` a une mantisse de 53 bits ; refaire le calcul avec le type `long double` qui a une mantisse de 64 bits. Qu'observez-vous ? Peut-on raisonnablement trouver la valeur exacte de x_{48} ?

On peut obtenir des encadrements fiables avec la classe `RReal` en contrôlant judicieusement le mode d'arrondi. Le programme `instable-rreal.cc` met en œuvre cette idée. (Le lire puis tester.) Certes, le calcul fiable ne peut pas enlever l'instabilité qui est une caractéristique de la fonction f. En l'occurrence les encadrements successifs deviennent de moins en moins précis, mais on peut toujours garantir leur correction. Ainsi tout s'affiche honnêtement sur les encadrements calculés : quand le calcul tourne mal, au moins on en est averti.

Remarque 2.7. Le phénomène d'instabilité est souvent appelé *comportement chaotique*, terme largement popularisé. Tous les systèmes ne sont pas chaotiques, heureusement, mais certains modèles (biologiques, physiques, météorologiques, etc.) le sont. Dans un tel cas on retrouve toutes les difficultés numériques de notre exemple minimaliste, et d'autres encore. Pensez-y quand vous consultez la météo pour le week-end.

3. Arithmétique d'intervalles

Les arrondis dirigés permettent d'encadrer tout nombre réel $a \in \mathbb{R}$ par une minoration $\underline{a}$ et une majoration $\overline{a}$ dans l'ensemble $R \subset \mathbb{R}$ des nombres machine. Ceci revient à remplacer la valeur exacte a par un intervalle $\mathbf{a} = [\underline{a}, \overline{a}]$ contenant a. C'est nécessaire quand a n'est pas exactement représentable ($a \notin R$) ou notre calcul n'aboutit pas à déterminer a plus précisément.

Le calcul séparé d'une minoration puis d'une majoration est souvent assez répétitif et la notation en C++ devient vite assez lourde. Il est plus commode d'encapsuler les deux bornes en un seul objet. L'idée est simple : au lieu des valeurs réelles $a \in \mathbb{R}$ on stocke et travaille les intervalles $\mathbf{a} = [\underline{a}, \overline{a}]$ avec $\underline{a}, \overline{a} \in R$.

3.1. Arithmétique d'intervalles idéalisée. Dans la suite nous allons regarder des intervalles compacts $[a,b] = \{x \in \mathbb{R} \mid a \le x \le b\}$ avec $a \le b$ dans $\mathbb{R}$. Nous n'autorisons pas l'intervalle vide ici, ni des intervalles infinis comme $[a, +\infty[$ ou $]-\infty, b]$ voire $]-\infty, +\infty[$. Ces intervalles peuvent être bien utiles, mais pour simplifier nous ne regardons que des intervalles compacts non vides. Par un léger abus de langage on confondera l'intervalle $[a,a] = \{a\}$ avec le point $a \in \mathbb{R}$.

Notation. Dans la suite une lettre en gras comme $\mathbf{a}$ dénote un intervalle, $\mathbf{a} = [\underline{a}, \overline{a}]$, alors que la lettre soulignée $\underline{a} = \min \mathbf{a}$ et la lettre surlignée $\overline{a} = \max \mathbf{a}$ dénotent les extrémités de l'intervalle. Cette écriture est assez intuitive et évite tout conflit avec d'éventuels indices. Nous écrivons $\mathbf{a} \in \mathbb{R}$ si $\underline{a} = \overline{a}$.

Proposition 3.1. *Si $\mathbf{a} = [\underline{a}, \overline{a}]$ est un intervalle compact et que $f : \mathbb{R} \to \mathbb{R}$ est une fonction continue, alors l'image $f(\mathbf{a}) = \{f(a) \mid a \in \mathbf{a}\}$ est à nouveau un intervalle compact.* □

Exercice/M 3.2. Montrer la proposition, puis vérifier les formules suivantes :
- Si f est croissante, alors $f([\underline{a}, \overline{a}]) = [f(\underline{a}), f(\overline{a})]$.
- Si f est décroissante, alors $f([\underline{a}, \overline{a}]) = [f(\overline{a}), f(\underline{a})]$.

Plus généralement, si f est monotone par morceaux, on peut calculer l'image $f([\underline{a}, \overline{a}])$ en mettant bout à bout les intervalles sur lesquels f est monotone. Considérons par exemple $f : \mathbb{R} \to \mathbb{R}$, $f(x) = |x|$:
- Si $0 \le \underline{a} \le \overline{a}$ alors $f([\underline{a}, \overline{a}]) = [f(\underline{a}), f(\overline{a})]$.
- Si $\underline{a} \le \overline{a} \le 0$ alors $f([\underline{a}, \overline{a}]) = [f(\overline{a}), f(\underline{a})]$.
- Si $\underline{a} \le 0 \le \overline{a}$ alors $f([\underline{a}, \overline{a}]) = [0, c]$ avec $c = \max(f(\underline{a}), f(\overline{a}))$.

Les mêmes formules sont valables pour toute fonction f qui est décroissante sur $\mathbb{R}_-$ et croissante sur $\mathbb{R}_+$, comme par exemple $f(x) = x^n$ pour $n = 2, 4, 6, \ldots$.

Définition 3.3. Pour deux intervalles $\mathbf{a}$ et $\mathbf{b}$ on définit les opérations élémentaires $+, -, *, /$ comme suit : si $\circ$ dénote une de ces opérations, alors on pose $\mathbf{a} \circ \mathbf{b} := \{a \circ b \mid a \in \mathbf{a}, b \in \mathbf{b}\}$.

Proposition 3.4. *L'arithmétique d'intervalles obéit aux règles suivantes :*

$$\mathbf{a}+\mathbf{b} = [\underline{a}+\underline{b}, \overline{a}+\overline{b}] \qquad \mathbf{a}*\mathbf{b} = [\,\min\{\underline{ab},\underline{a}\overline{b},\overline{a}\underline{b},\overline{a}\overline{b}\}\,,\ \max\{\underline{ab},\underline{a}\overline{b},\overline{a}\underline{b},\overline{a}\overline{b}\}\,]$$
$$-\mathbf{b} = [-\overline{b}, -\underline{b}] \qquad \mathbf{b}^{-1} = [\overline{b}^{-1}, \underline{b}^{-1}] \quad si \quad 0 \notin \mathbf{b}$$
$$\mathbf{a}-\mathbf{b} = \mathbf{a}+(-\mathbf{b}) = [\underline{a}-\overline{b}, \overline{a}-\underline{b}] \qquad \mathbf{a}/\mathbf{b} = \mathbf{a}*\mathbf{b}^{-1}$$

Exercice/M 3.5. Vérifier ces règles de calculs, dont seule la multiplication est délicate. Justifier d'abord que $\mathbf{c} = \mathbf{a}*\mathbf{b}$ est un intervalle compact, et que $\min \mathbf{c}$ et $\max \mathbf{c}$ sont toujours atteints dans un des quatre coins du rectangle $\mathbf{a}\times\mathbf{b} \subset \mathbb{R}^2$, donc $\underline{c} = \min\{\underline{ab},\underline{a}\overline{b},\overline{a}\underline{b},\overline{a}\overline{b}\}$ et $\overline{c} = \max\{\underline{ab},\underline{a}\overline{b},\overline{a}\underline{b},\overline{a}\overline{b}\}$.
Optimisation. — Pour une implémentation efficace on peut prédire quel(s) produit(s) réalisent le minimum ou maximum : l'intervalle $\mathbf{a}$ peut être ou strictement positif ($0 < \underline{a} \leq \overline{a}$, noté $0 < \mathbf{a}$), ou strictement négatif ($\underline{a} \leq \overline{a} < 0$, noté $\mathbf{a} < 0$), ou bien il contient zéro ($0 \in \mathbf{a}$ équivaut à $\underline{a} \leq 0 \leq \overline{a}$). De même pour l'intervalle $\mathbf{b}$; il y a donc au total 9 cas à distinguer. Dans 8 cas, le calcul de $\underline{c}$ et $\overline{c}$, respectivement, ne nécessite qu'une seule multiplication. Le dernier cas $\underline{a} < 0 < \overline{a}$ et $\underline{b} < 0 < \overline{b}$ est plus délicat et nécessite quatre multiplications.

Proposition 3.6. *L'arithmétique d'intervalles se réjouit des propriétés suivantes :*

Associativité :	$(\mathbf{a}+\mathbf{b})+\mathbf{c} = \mathbf{a}+(\mathbf{b}+\mathbf{c})$	$(\mathbf{a}*\mathbf{b})*\mathbf{c} = \mathbf{a}*(\mathbf{b}*\mathbf{c})$
Commutativité :	$\mathbf{a}+\mathbf{b} = \mathbf{b}+\mathbf{a}$	$\mathbf{a}*\mathbf{b} = \mathbf{b}*\mathbf{a}$
Élément neutre :	$\mathbf{0}+\mathbf{a} = \mathbf{a}+\mathbf{0} = \mathbf{a}$	$\mathbf{1}*\mathbf{a} = \mathbf{a}*\mathbf{1} = \mathbf{a}$
Sous-distributivité :	$\mathbf{a}*(\mathbf{b}+\mathbf{c}) \subseteq (\mathbf{a}*\mathbf{b})+(\mathbf{a}*\mathbf{c})$	

En général la dernière inclusion peut être stricte, mais on a égalité $\mathbf{a}*(\mathbf{b}+\mathbf{c}) = (\mathbf{a}*\mathbf{b})+(\mathbf{a}*\mathbf{c})$ *si* $\mathbf{a} \in \mathbb{R}$ *ou* $\mathbf{b},\mathbf{c} \geq 0$ *ou* $\mathbf{b},\mathbf{c} \leq 0$. *Pour un intervalle* $\mathbf{a}$ *qui n'est pas réduit à un point il n'existe pas d'intervalle inverse, ni pour l'addition ni pour la multiplication : on a* $\mathbf{a}+(-\mathbf{a}) \supsetneqq \mathbf{0}$ *et de même* $\mathbf{a}*\mathbf{a}^{-1} \supsetneqq \mathbf{1}$. □

Exercice/M 3.7. Les propositions précédentes montrent que l'arithmétique d'intervalles possède de bonnes propriétés, mais aussi qu'il faut s'habituer aux exceptions. Essayez de construire un exemple où $\mathbf{a}*(\mathbf{b}+\mathbf{c}) \neq (\mathbf{a}*\mathbf{b})+(\mathbf{a}*\mathbf{c})$.

Exercice/M 3.8. A-t-on toujours $\mathbf{a}+\mathbf{a} = 2\mathbf{a}$? Pour $n \in \mathbb{N}$ on définit $\mathbf{a}^n = \{a^n \mid a \in \mathbf{a}\}$. Montrer que $\mathbf{a}*\mathbf{a} = \mathbf{a}^2$, ou plus généralement $\mathbf{a}^m * \mathbf{a}^n = \mathbf{a}^{m+n}$, si $\mathbf{a} \geq 0$ ou $\mathbf{a} \leq 0$. A-t-on toujours $\mathbf{a}*\mathbf{a} \supseteq \mathbf{a}^2$, ou plus généralement $\mathbf{a}^m * \mathbf{a}^n \supseteq \mathbf{a}^{m+n}$? Construire un exemple où $\mathbf{a}*\mathbf{a} \neq \mathbf{a}^2$. Ceci montre que deux expressions algébriques peuvent définir la même application $\mathbb{R} \to \mathbb{R}$, mais différentes applications sur l'ensemble des intervalles.

3.2. Arithmétique d'intervalles arrondie. Jusqu'à présent l'arithmétique de $\mathbb{R}$ a été utilisée pour décrire l'arithmétique d'intervalles. Une implémentation doit prendre en compte l'arithmétique arrondie disponible sur ordinateur : quand on implémente l'arithmétique d'intervalles sur ordinateur, il est naturel de stocker tout intervalle $\mathbf{a} = [\underline{a},\overline{a}]$ par ses extrémités $\underline{a}$ et $\overline{a}$. Il s'agit de deux nombres à virgule flottante, et il faut passer des intervalles idéalisés aux intervalles arrondis :

- Pour un intervalle idéalisé $\mathbf{a} = [\underline{a},\overline{a}]$ les extrémités $\underline{a},\overline{a} \in \mathbb{R}$ ne sont en général pas exactement représentables par des nombres machines, c'est-à-dire $\underline{a} \notin R$ ou $\overline{a} \notin R$. On passe donc à un intervalle (légèrement) plus grand $\mathbf{b} = [\underline{b},\overline{b}]$ dont les extrémités $\underline{b} = \lfloor \underline{a} \rfloor_R$ et $\overline{b} = \lceil \overline{a} \rceil_R$ sont des nombres machines. À noter que les arrondis dirigés assurent l'inclusion $\mathbf{a} \subseteq \mathbf{b}$: pour cela il faut arrondir $\underline{a}$ vers $-\infty$ et $\overline{a}$ vers $+\infty$. On parle de *l'arrondi vers l'extérieur*.
- Même si $\mathbf{a} = [\underline{a},\overline{a}]$ est exactement représentable par des nombres machines $\underline{a},\overline{a} \in R$, l'application d'une fonction f produit en général un intervalle $f(\mathbf{a})$ que ne l'est plus. Considérons une fonction croissante, comme $f(x) = \exp(x)$: au lieu de l'intervalle $f([\underline{a},\overline{a}]) = [f(\underline{a}), f(\overline{a})]$ nous allons nous contenter d'un intervalle (légèrement) plus grand $\mathbf{b} = [\underline{b},\overline{b}]$ dont les extrémités $\underline{b} \leq f(\underline{a})$ et $\overline{b} \geq f(\overline{a})$ sont des valeurs approchées, convenablement arrondies. À nouveau les arrondis dirigés vers l'extérieur assurent que $f(\mathbf{a}) \subseteq \mathbf{b}$.

☞ *L'arithmétique d'intervalles arrondie repose sur le principe que tout calcul retourne un encadrement garanti du résultat exact.* ✍

L'implémentation nécessite donc l'arrondi dirigé : en arrondissant vers l'extérieur on garantit toujours que l'intervalle calculé contient tous les résultats possibles à partir des données d'entrée.

3.3. Une implémentation « faite maison ». Afin d'avoir une écriture commode nous souhaitons implémenter une classe `Interval` qui permette d'écrire le code suivant :

```
Interval rayon, pi( 3.14, 3.15 );  // encadrement grossier mais correct de pi
cin >> rayon;                      // donnée initiale provenant d'une mesure
Interval aire= pi * r * r;         // calcul selon l'arithmétique d'intervalles
cout << "aire = " << aire << endl; // affichage du résultat sous forme d'intervalle
```

Le programme XVI.3 ci-dessous montre le début d'une telle classe, encore rudimentaire, qui met au point l'arithmétique d'intervalles selon les règles développées ci-dessus. Pour les arrondis dirigés nous utilisons notre classe `RReal` expliquée plus haut.

Remarque 3.9. Un objet de la classe `Interval` est donné par une paire (`mini`,`maxi`) où `mini` et `maxi` sont des nombres réels représentables par le type de base, en l'occurrence notre classe `RReal`. Vous constaterez une particularité dans le design de la classe `Interval` : nous exigeons toujours que `mini` $\leq$ `maxi`. Afin de maintenir cet invariant nous sommes obligés de déclarer les éléments `mini` et `maxi` comme étant privés à la classe. Ceci empêche d'accéder directement à ces éléments – prudence oblige. Ensuite toutes nos opérations s'engagent à produire des résultats avec `mini` $\leq$ `maxi`.

Exercice/P 3.10. Le fichier `interval.cc` contient une implémentation plus complète ; essayez de comprendre sa structure et de vous familiariser avec les diverses fonctions qu'elle offre. (Lire aussi puis tester `interval-test.cc`.) En particulier vous y trouverez une implémentation de la multiplication, qui minimise le nombre de multiplications à effectuer sur le type `RReal`. Essayez de comprendre puis de vérifier son fonctionnement (cf. l'exercice 3.5).

Remarque 3.11. Afin d'utiliser l'écriture usuelle des opérations élémentaires `a+b`, `a-b`, `a*b`, `a/b` il faut encore définir ces opérateurs en C++. Ceci n'est qu'une simple reformulation des fonctions déjà implémentées :

```
Interval add( const Interval& a, const Interval& b, Len len= stdLen )
{ Interval c; c.add( a, b, len ); return c; }

Interval operator+ ( const Interval& a, const Interval& b )
{ Interval c; c.add( a, b ); return c; }
```

À noter que sous cette forme on crée un objet temporaire, ici nommé `c`, puis on lui affecte la valeur calculée. Sous la première forme `add(a,b,len)` on peut optionnellement spécifier la longueur de la mantisse, c'est-à-dire la précision souhaitée. Sous la seconde forme `a+b` cette information n'apparaît plus explicitement : il est sous-entendu que l'on utilise la valeur spécifiée dans la variable globale `stdLen`.

Exercice/P 3.12. Après les opérations élémentaires, les fonctions usuels peuvent être implémentées pour les intervalles. Le programme `interval.cc` contient deux exemples faciles :

```
Interval sqr( const Interval& a );
Interval sqrt( const Interval& a );
```

La fonction `sqr` applique la fonction $x \mapsto x^2$ à un intervalle, alors que `sqrt` applique $x \mapsto \sqrt{x}$ pour $x \geq 0$. À noter que $x \mapsto x^2$ n'est que monotone par morceaux, donc il faut distinguer plusieurs cas. Rappelons aussi qu'en général $\mathbf{a}^2 \neq \mathbf{a} * \mathbf{a}$, il est donc fort utile de disposer des implémentations faites sur mesure pour les intervalles. Plus généralement, vous y trouverez une implémentation des fonctions $x \mapsto x^n$ et $x \mapsto \sqrt[n]{x}$:

```
Interval power( const Interval& a, long n );
Interval root( const Interval& a, long n );
```

3.4. Exemples d'utilisation.

Exercice/P 3.13. Les fichiers `somme-rreal.cc` et `somme-interval.cc` implémentent le calcul de $s_n = \sum_{k=1}^{n} \frac{1}{k^2}$ en utilisant les types `RReal` et `Interval`, respectivement. Lisez attentivement ces petits exemples pour vous familiariser avec l'usage de ces types. Vérifiez que c'est strictement la même démarche et le même résultat. La seule différence est que l'écriture sous forme d'intervalle est plus commode.

Exercice/P 3.14. Vérifier, tester, puis comparer `instable-rreal.cc` et `instable-interval.cc`.
Attention. — Appliquer la fonction $f : \mathbb{R} \to \mathbb{R}$, $f(x) = 4x(1-x)$ à un intervalle $\mathbf{a}$ n'est pas du tout équivalent à calculer $\mathbf{4} * \mathbf{a} * (\mathbf{1} - \mathbf{a})$. Discuter cette différence et l'illustrer par des exemples.

Programme XVI.3 La classe `Interval` — arithmétique d'intervalles

```
class Interval
{
private:
  // Données privées: le minimum et le maximum de l'intervalle en question
  RReal mini, maxi;

  // Affectation sans contrôle (à usage privé uniquement)
  Interval& assign( const RReal& a, const RReal& b )
    { mini= a; maxi= b; return *this; }

public:
  // Constructeur à partir d'un intervalle (contrôle inutile)
  Interval( const Interval& interval )
    : mini( interval.mini ), maxi( interval.maxi ) {}

  // Constructeur à partir de deux bornes (avec contrôle)
  Interval( const RReal& a, const RReal& b )
    : mini(a), maxi(b) { if( mini > maxi ) mini.swap( maxi ); }

  // Constructeur à partir de deux bornes (avec contrôle)
  Interval( double a, double b )
    { if(a>b) std::swap(a,b); mini= RReal(a,RNDD); maxi= RReal(b,RNDU); }

  // Accès contrôlé aux informations -- en lecture seulement !
  const RReal& min() const { return mini; }   // lire le minimum
  const RReal& max() const { return maxi; }   // lire le maximum

  // Addition de deux intervalles
  Interval& add( const Interval& a, const Interval& b, Len len= stdLen )
    { return assign( Numeric::add( a.mini, b.mini, RNDD, len ),
                     Numeric::add( a.maxi, b.maxi, RNDU, len ) ); }

  // Soustraction de deux intervalles
  Interval& sub( const Interval& a, const Interval& b, Len len= stdLen )
    { return assign( Numeric::sub( a.mini, b.maxi, RNDD, len ),
                     Numeric::sub( a.maxi, b.mini, RNDU, len ) ); }

  // La multiplication de deux intervalles
  Interval& mul( const Interval& a, const Interval& b, Len len= stdLen );

  // Inversion d'un intervalle ne contenant pas zéro
  Interval& inv( const Interval& a, Len len= stdLen )
    {
      if( sign(a.mini) <= 0 && sign(a.maxi) >= 0 )
        { cerr << "Interval division by zero!" << endl; exit(1); }
      return assign( Numeric::div( 1, a.maxi, RNDD, len ),
                     Numeric::div( 1, a.mini, RNDU, len ) );
    }

  // La division a/b est ramenée à une multiplication  a * (1/b).
  Interval& div( const Interval& a, const Interval& b, Len len= stdLen )
    { return mul( a, inv( b, len ), len ); }

  // L'entrée-sortie
  void write( ostream& out ) const;
  void read( istream& in );
};
```

4. Applications

Ce chapitre met à votre disposition les classes `RReal` et `Interval` pour le calcul arrondi en précision arbitraire avec des arrondis dirigés. Ceci permet d'établir des résultats numériques rigoureux, c'est-à-dire mathématiquement prouvables. Ces outils sont particulièrement bien adaptés pour évaluer des séries, dont ce paragraphe vous propose quelques exemples classiques. Dans les exercices suivants vous pouvez donc expérimenter avec nos classes `RReal` et `Interval` faites maison.

4.1. Retour sur les problèmes du chapitre XV. Avec nos nouveaux outils, vous pouvez reprendre quelques calculs du chapitre XV et les réécrire plus savamment comme calculs fiables : l'instabilité de Fibonacci (§2), l'évaluation du polynôme de Rump (§4.5), puis les pièges à éviter (§5).

Question 4.1. Analysez en particulier le comportement de l'arithmétique d'intervalles lors d'une perte de chiffres significatifs (phénomène d'annulation, voir chapitre XV,§5.4). Le problème persiste-t-il ? Quel est alors l'avantage de l'arithmétique d'intervalles dans ce cas ?

Exercice/P 4.2. Les équations quadratiques sont reprises dans `chap14/quadratique.cc`. Testez puis discutez quels problèmes sont ainsi résolus, et lesquels persistent ou s'ajoutent... Optimisez ce programme pour qu'il devienne robuste et produise des résultats satisfaisants.

4.2. La fonction zéta de Riemann.

Exercice/P 4.3. La série $\sum \frac{1}{k}$ diverge, c'est-à-dire les sommes partielles $s_n = \sum_{k=1}^{n} \frac{1}{k}$ croissent sans borne. Pourtant elles deviennent stationnaires quand on les calcule naïvement sur ordinateur ! Essayez de prédire la valeur stationnaire pour le type `float`. Le vérifier avec le programme `chap13/divergence.cc`.

Ce phénomène bizarre persistera-t-il quand on encadre s_n avec la classe `RReal` , ou plus commodément avec `Interval` ? Essayez de prédire le résultat, puis testez-le empiriquement si cela vous intéresse. Pour imiter le type `float` on prendra des mantisses de 24 bits, mais vous pouvez varier ce choix.

☞ *Se méfier d'une apparente « convergence numérique » sans contrôle d'erreur.* ✍

Exercice/P 4.4. La série $\sum_{k=1}^{\infty} \frac{1}{k^2}$ converge. Pour approcher la limite, en utilisant le type `float`, calculer $\sum_{1}^{n} \frac{1}{k^2}$ dans le sens des indices croissants, puis $\sum_{n}^{1} \frac{1}{k^2}$ dans le sens des indices décroissants (voir `somme-naive.cc`). Vu la nature des erreurs d'arrondi, quelle approche vous semble plus exacte ?

Comparer avec les encadrements fiables : lire puis tester les deux programmes `somme-rreal.cc` et `somme-interval.cc`. Les deux encadrements ainsi obtenus sont corrects, mais l'un est plus fin que l'autre ! Le tester puis expliquer le résultat.

☞ *Lors d'une sommation numérique l'ordre des termes peut influencer le résultat.* ✍

Exercice/M 4.5. Afin d'encadrer la valeur de $\zeta(2) = \sum_{k=1}^{\infty} \frac{1}{k^2}$ il ne suffit évidemment pas de calculer la somme partielle $s_n = \sum_{k=1}^{\infty} \frac{1}{k^2}$, il faut aussi majorer le reste $r_n = \sum_{k=n+1}^{\infty} \frac{1}{k^2}$. Montrer que $\frac{1}{n+1} < r_n < \frac{1}{n}$:

(1) via l'encadrement $\frac{1}{k} - \frac{1}{k+1} < \frac{1}{k^2} < \frac{1}{k-1} - \frac{1}{k}$ puis une somme télescopique,

(2) via l'encadrement $\int_k^{k+1} \frac{1}{x^2} dx < \frac{1}{k^2} < \int_{k-1}^{k} \frac{1}{x^2} dx$ puis la somme des intégrales.

En déduire un programme qui calcule un encadrement fiable de $\zeta(2)$ en fonction de n. (On sait d'ailleurs que $\zeta(2) = \pi^2/6$, mais ce beau résultat ne joue pas de rôle ici.)

☞ *C'est la majoration du reste qui fait d'une série $\sum_{k=1}^{\infty} a_k$ une méthode praticable pour calculer une valeur approchée de la somme.* ✍

Exercice/M 4.6. En généralisant l'exercice précédent, écrire un programme qui calcule un encadrement de $\zeta(3) = \sum_{k=1}^{\infty} \frac{1}{k^3}$ en fonction de n. Si vous voulez, vous pouvez étendre cette approche afin d'encadrer $\zeta(s)$ pour $s = 2,3,4,5,\ldots$. Est-ce praticable pour tout $s > 1$ et $\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s}$?

4.3. Séries alternées : sin, cos, arctan, etc. Pour une suite $(u_k)_{k\in\mathbb{N}}$ de nombres réels $u_k \in \mathbb{R}$ on écrit $u_k \searrow u$ si la suite est décroissante, $u_0 \geq u_1 \geq u_2 \geq \ldots$, avec pour limite $\lim u_k = \inf u_k = u$. De manière analogue on écrit $u_k \nearrow u$ si la suite est croissante, $u_0 \leq u_1 \leq u_2 \leq \ldots$, avec $\lim u_k = \sup u_k = u$.

Théorème 4.7. *Si $u_k \searrow 0$ alors la série alternée $\sum_{k=0}^{\infty}(-1)^k u_k$ converge, c'est-à-dire les sommes partielles $s_n = \sum_{k=0}^{n}(-1)^k u_k$ convergent vers un nombre réel $s \in \mathbb{R}$. Plus précisément on a $s_{2n} \searrow s$ et $s_{2n+1} \nearrow s$. Ainsi on obtient des encadrements $s_{2n-1} \leq s \leq s_{2n}$ de plus en plus fins de la valeur $s = \sum_{k=0}^{\infty}(-1)^k u_k$.*

Exercice/M 4.8. Montrer ce théorème.

Exercice/P 4.9. Pour appliquer ce théorème, par exemple à la série $\sum_{k=1}^{\infty}(-1)^{k-1}\frac{1}{\sqrt{k}}$, on pourra calculer le terme général par la fonction

```
Interval terme_general( long n ) { return 1/sqrt(Interval(n)); }
```

Comme les séries alternées sont assez fréquentes, il sera utile de disposer d'une fonction générique

```
Interval somme_alternee( long debut, long fin );
```

qui donne un encadrement prouvable de $s = \sum_{k=k_0}^{\infty}(-1)^{k-k_0}u_k$. Le choix de k_0 est commode si vous voulez sommer à partir d'un indice quelconque, en l'occurrence $k_0 = 1$ et non $k_0 = 0$. Pour simplifier on pourra sommer un nombre pair de terme, puis ajouter la marge d'erreur obtenue par le terme suivant.

Plus souvent il est préférable de prescrire une borne $\varepsilon > 0$ et de sommer jusqu'à ce que $u_{n+1} \leq \varepsilon$. Implémenter une telle fonction

```
Interval somme_alternee( long debut, const RReal& eps );
```

Si possible, veillez à ne pas évaluer inutilement la fonction `terme_general`.

Exercice/P 4.10. Implémenter les fonctions

$$\cos(x) = \sum_{k=0}^{\infty}(-1)^k\frac{x^{2k}}{(2k)!} \qquad \text{et} \qquad \sin(x) = \sum_{k=0}^{\infty}(-1)^k\frac{x^{2k+1}}{(2k+1)!}$$

pour $x \in [0,1]$ à une précision $\varepsilon = 2^{-\texttt{len}}$ près. En supposant que nous disposons d'une bonne approximation de π, comment implémenter $\sin(x)$ et $\cos(x)$ efficacement pour tout $x \in \mathbb{R}$?

Exercice/P 4.11. On se propose de calculer $g(x) = \int_0^x e^{-t^2}dt$. Développer $f(t) = e^{-t^2}$ en une série entière, intégrer terme par terme pour obtenir la série entière pour la fonction primitive g avec $g' = f$ et $g(0) = 0$. Justifier ce résultat. Implémenter ainsi le calcul de $g(x)$ pour $x \in [0,1]$ à une précision $\varepsilon = 2^{-\texttt{len}}$ près.

Exercice/P 4.12. Pour $x \in [-1,1]$ montrer que la série $\sum_{k=1}^{\infty}(-1)^{k+1}\frac{x^{2k-1}}{2k-1}$ converge vers $\arctan(x)$: développer $\frac{d}{dx}\arctan(x) = \frac{1}{1+x^2}$ en une série entière, puis intégrer terme par terme pour obtenir la série entière de la fonction primitive $\arctan(x)$. Justifier ce résultat pour $-1 < x < 1$, puis aux extrémités $x = \pm 1$.

Est-ce une méthode praticable pour calculer $\arctan(x)$ avec $x \in [0, \frac{1}{2}]$? Implémenter ce calcul à une précision $\varepsilon = 2^{-\texttt{len}}$ près. Combien de termes sont nécessaires ? Y a-t-il un problème d'annulation de termes consécutifs et de perte de chiffres significatifs ? Est-ce encore praticable proche de l'extrémité $x = 1$? Comment implémenter $\arctan(x)$ efficacement pour tout $x \in \mathbb{R}$?

☞ *La sommation numérique est efficace seulement si la série converge rapidement.* ✍

4.4. Calcul de π.

Exercice/P 4.13. Dans le projet IV on a développé un calcul de $\pi = 2\sum_{k=0}^{\infty}\frac{1\cdot 2\cdots k}{3\cdot 5\cdots(2k+1)}$ à 10000 décimales exactes — avec preuve de correction ! Vous pouvez refaire, si vous voulez, un tel calcul avec un type de nombres à virgule flottante convenable. *Indication.* — La série peut être sommée dans les deux sens. Elle peut aussi être évaluée par la méthode de Horner. Choisir une de ces méthodes, ou bien tester les toutes.

Exercice/P 4.14. Vérifier que $x := e^{\pi\sqrt{163}} \approx 262\,537\,412\,640\,768\,743,999\,999\,999\,999\ldots$ est invraisemblablement proche de l'entier $y := 262\,537\,412\,640\,768\,744$. On soupçonne néanmoins que $x \neq y$: peut-on implémenter un calcul qui permet de prouver que $x \neq y$? Si l'on avait $x = y$, serait-il possible de prouver cette égalité par un calcul numérique sur ordinateur ?

PROJET XVI

Calcul fiable de exp et log

Objectif

- Implémenter correctement et efficacement les fonctions exp et log.
- Majorer l'erreur de l'approximation afin de garantir un encadrement.
- Garantir un calcul numérique fiable grâce aux arrondis dirigés.

On appelle *usuelles* les fonctions exp et log, les fonctions trigonométriques sin, cos, tan avec leurs inverses arcsin, arccos, arctan, ainsi que les fonctions hyperboliques sinh, cosh, tanh avec leurs inverses asinh, acosh, atanh (et d'autres encore selon le contexte). Toute bibliothèque numérique qui se respecte offre ces fonctions, comme par exemple `<cmath>` pour le type `double`. Dans ce projet on implémentera les deux premières fonctions, exp et log, certes les plus simples mais suffisamment représentatives pour toute la famille. En choisissant judicieusement les modes d'arrondi on peut garantir la correction du résultat sous forme d'un encadrement fiable, à n'importe quelle précision souhaitée.

1. Calcul de l'exponentielle

On implémente d'abord la fonction $\exp\colon \mathbb{R} \to \mathbb{R}_+$ définie par $\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$. Comme on ne peut calculer que des sommes finies, on évaluera le polynôme $s_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!}$ pour un certain rang n, encore à déterminer. Pour le calcul numérique deux problèmes se posent. Si $|x|$ est grand, les premiers termes croissent avant que la factorielle $k!$ ne l'emporte. Pire encore, pour $x < 0$ les termes sont de signes alternés, ce qui entraîne de sérieuses difficultés d'annulation (perte de chiffres significatifs, voir chapitre XV, §5.5).

Exercice/M 1.1 (réduction de l'argument)**.** Pour les raisons évoquées on évaluera la série seulement pour $x \in [0,1]$ où le comportement numérique est excellent. Notre première tâche sera de majorer le reste $r_n(x) = \sum_{k=n+1}^{\infty} \frac{x^k}{k!}$. Évidemment on a $r_n(x) \geq \frac{x^{n+1}}{(n+1)!}$; montrer la majoration $r_n(x) \leq \frac{2x^{n+1}}{(n+1)!} =: \varepsilon_n$.

Exercice/P 1.2 (choix du rang n)**.** Pour $x \in [0,1]$ on sait que $\exp(x) \in [1,3]$. Afin de calculer $\exp(x)$ avec une mantisse de longueur `len`, l'écart toléré sera donc $\varepsilon = 2^{-\texttt{len}}$. (*Astuce.* — Comment construire *sans calcul* ce nombre ε en utilisant le type `RReal` ?) Le choix $n = \max\{5, \texttt{len}\}$ suffira largement mais sera inutilement grand. Écrire une boucle qui calcule successivement ε_n et qui détermine le plus petit n tel que $\varepsilon_n \leq \varepsilon$. (*Astuce.* — Le passage de ε_{n-1} à ε_n ne nécessite que deux opérations arithmétiques.)

Exercice/P 1.3 (implémentation, cas restreint)**.** En utilisant les exercices précédents, écrire une fonction

```
RReal exp1( const RReal& x, Mode mode= stdMode, Len len= stdLen )
```

qui calcule une approximation de $\exp(x)$ pour $x \in [0,1]$ en choisissant judicieusement les arrondis.
Indication. — Afin d'évaluer $s_n(x)$ on utilisera la méthode de Horner :

$$s_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!} = \left(\left(\dots\left(\left(\left(\frac{x}{n}+1\right)\frac{x}{n-1}+1\right)\frac{x}{n-2}+1\right)\dots\right)\frac{x}{2}+1\right)\frac{x}{1}+1$$

Tous les facteurs $\frac{x}{k}$ sont positifs, donc pour obtenir un certain arrondi `RNDD`, `RNDU`, `RNDZ`, ou `RNDI` dans le résultat final, il suffit d'appliquer ce même mode d'arrondi à chaque opération élémentaire intermédiaire.
Attention. — Pour `RNDU` et `RNDI` n'oubliez pas d'ajouter la marge d'erreur ε_n ou ε à la fin de la fonction.

Exercice/P 1.4 (implémentation, cas général)**.** Afin de calculer $\exp(x)$ pour $x \in \mathbb{R}$ quelconque on se ramène au cas restreint où x est dans l'intervalle $[0,1]$:

(1) Pour $x < 0$ on utilise $\exp(x) = 1/\exp(-x)$. *Astuce.* — Pour un mode d'arrondi `mode` l'inverse est donné par `!mode` : ceci échange `RNDD` et `RNDU`, ainsi que `RNDZ` et `RNDI`.

(2) Pour $x \geq 1$ on applique $\exp(x) = \exp(x/2^k)^{(2^k)}$. *Astuce.* — La fonction `x.magnitude()` aidera à déterminer k. Elle est définie et documentée dans le fichier `rreal.cc`. La division par 2^k n'est qu'une simple soustraction des exposants. La puissance $y^{(2^k)}$ ne nécessite que k opérations.

En tenant compte du mode d'arrondi choisi, écrire ainsi deux fonctions

```
RReal exp( RReal x, Mode mode= stdMode, Len len= stdLen );  // entre 15 et 20 lignes
Interval exp( const Interval& a, Len len= stdLen );         // une seule ligne suffit
```

Vérification. — Encadrer $e = \exp(1)$ à 100 décimales, soit 330 bits environ. Pour vérification comparer avec les résultats du chapitre IV, §2, ou une autre source suffisamment sérieuse. De même, par souci de tester votre implémentation, encadrer aussi d'autres valeurs, comme e^{100} et e^{-100} par exemple.

Exercice/P 1.5. Calculer un encadrement fiable de $\sqrt{163}$, de π, puis de $x := \exp(\pi\sqrt{163})$. Adapter la précision afin de prouver que x n'est pas un nombre entier. (Voir l'exercice 4.14 du chapitre XVI,)

2. Calcul du logarithme

Comme inverse de l'exponentielle on se propose d'implémenter la fonction $\log\colon \mathbb{R}_+ \to \mathbb{R}$. Pour tout $t \in]-1,+1[$ on a $\log(1+t) = \sum_{k=1}^{+\infty} (-1)^{k+1} \frac{t^k}{k}$, mais cette série converge trop lentement proche des bords ± 1 (voir le chapitre XV, exercices 6.6 et 6.7). Afin d'accélérer la convergence on passe donc à la série

$$\log\left(\frac{1+t}{1-t}\right) = 2\left(t + \frac{t^3}{3} + \frac{t^5}{5} + \dots\right) = 2t \sum_{k=0}^{\infty} \frac{t^{2k}}{2k+1}$$

Exercice/M 2.1 (transformation entre x et t)**.** Pour $x \in \mathbb{R}_+$ expliciter l'unique valeur $t \in]-1,+1[$ telle que $x = \frac{1+t}{1-t}$. Implémenter ce calcul $x \mapsto t$ en tenant compte du mode d'arrondi choisi.

Dans la suite on considérera seulement $x \in [1,2]$, ce qui se traduit en $t \in [0, \frac{1}{3}]$ (le vérifier). Sur cette intervalle notre série converge très rapidement : un peu mieux que la série géométrique à base $t^2 \leq \frac{1}{9}$.

Exercice/M 2.2 (majoration de l'erreur)**.** Pour $t \geq 0$ notre série donne la valeur $\log\left(\frac{1+t}{1-t}\right) \geq 2t$. Pour une mantisse de longueur `len` on tolère donc une erreur absolue de $2t \cdot 2^{-\texttt{len}}$. D'autre part la somme partielle $s_n(t) = 2t \sum_{k=0}^{n} \frac{t^{2k}}{2k+1}$ laisse une erreur $r_n(t) = 2t \sum_{k=n+1}^{+\infty} \frac{t^{2k}}{2k+1}$. Donner une majoration commode ε_n de $\sum_{k=n+1}^{+\infty} \frac{t^{2k}}{2k+1}$. Bien que le choix $n = \texttt{len}/3 - 1$ suffise toujours, on peut faire mieux si t est petit. Comment déterminer efficacement le plus petit indice n tel que $\varepsilon_n < 2^{-\texttt{len}}$?

Exercice/P 2.3 (implémentation, cas restreint)**.** En utilisant les exercices précédents, écrire une fonction

```
RReal log1( const RReal& x, Mode mode= stdMode, Len len= stdLen )
```

qui calcule une approximation de $\log(x)$ pour $x \in [1,2]$ en tenant compte du mode d'arrondi choisi.

Indication. — Comme toujours il vaut mieux évaluer $s_n(t) = 2t \sum_{k=0}^{n} \frac{t^{2k}}{2k+1}$ par la méthode de Horner. Pour `RNDU` et `RNDI` il faut ajouter la majoration du reste à la fin.

Vérification. — Encadrer ainsi $\log(2)$ à 100 décimales. Comparer avec la valeur fournie par la bibliothèque `cmath` ; combien de décimales sont exactes ? Si possible comparer avec une source plus sérieuse.

Exercice/P 2.4 (implémentation, cas général)**.** Afin de calculer $\log(x)$ pour $x \in \mathbb{R}_+$ quelconque on se ramène au cas restreint où x est dans l'intervalle $[1,2]$:

(1) Pour $x < 1$ on utilise $\log(x) = -\log(1/x)$.

(2) Pour $x \geq 2$ on applique $\log(x) = \log(x/2^k) + k\log(2)$.

En déduire des fonctions

```
RReal log( RReal x, Mode mode= stdMode, Len len= stdLen );  // entre 15 et 20 lignes
Interval log( const Interval& a, Len len= stdLen );         // une seule ligne suffit
```

Vérification. — Tester si les implémentations de exp et log donnent des encadrements cohérents : pour $x \in \mathbb{R}$ calculer un encadrement $\underline{y} \lessapprox \exp(x) \lessapprox \overline{y}$, puis calculer $\underline{z} \lessapprox \log(\underline{y})$ arrondi vers le bas et $\overline{z} \gtrapprox \log(\overline{y})$ arrondi vers le haut. Trouve-t-on $\underline{z} \leq x \leq \overline{z}$ comme il faut ? L'écart $\overline{z} - \underline{z}$ est-il acceptable ?

Exercice/P 2.5. Encadrer $\sqrt{2}$ à 100 décimales en calculant $2^{\frac{1}{2}} = \exp(\log(2)/2)$. Comparer avec la méthode de Newton-Héron ; quelle méthode vous semble préférable quant à la précision et l'efficacité ?

Exercice/P 2.6. À partir des fonctions `exp` et `log` écrire une fonction

```
RReal power( const RReal& x, const RReal& y, Mode mode= stdMode, Len len= stdLen );
Interval power( const Interval& x, const Interval& y, Len len= stdLen );
```

qui calcule x^y en tenant compte du mode d'arrondi choisi. (Attraper d'abord les exceptions.)

To explain all nature is too difficult a task for any one man
or even for any one age. 'Tis much better to do a little with certainty,
and leave the rest for others that come after you, than to explain all things.
Isaac Newton (1642-1727)

CHAPITRE XVII

Méthodes itératives pour la résolution d'équations

Objectif. Les méthodes itératives figurent parmi les méthodes numériques les plus courantes et le plus puissantes. L'idée est de partir d'une valeur approchée (souvent grossière) de la solution, puis d'augmenter la précision par l'application itérée d'un algorithme bien choisi.

Dans ce chapitre nous discutons deux méthodes itératives classiques : la méthode du point fixe pour résoudre une équation du type $f(x) = x$ où f est une fonction contractante, puis son raffinement, la méthode de Newton pour résoudre $f(x) = 0$ où f est une fonction dérivable. Pour des compléments voir le livre de J.-P. Demailly, *Analyse numérique et équations différentielles*, EDP Sciences, 1996.

Sommaire

Retour sur la méthode dichotomique

Commençons par la méthode « par tâtonnement » que l'on appelle plus savamment « la méthode dichotomique ». Cette méthode a le mérite d'être élémentaire, mais elle a deux inconvénients : d'abord elle ne converge que lentement, puis implémentée naïvement (avec des arrondis aléatoires) elle peut être inutilisable à cause des phénomènes de bruit, déjà discutés au chapitre XV, §5.6.

Exemple 0.1. Le programme `dichotomie.cc` résout le problème de bruit par un calcul fiable basé sur l'arithmétique d'intervalles. (Le lire puis le tester.) La difficulté principale est d'implémenter soigneusement la fonction donnée $f : \mathbb{R} \to \mathbb{R}$ selon l'arithmétique d'intervalles arrondie en une fonction

```
Interval f( const Interval& x ) .
```

D'une part il faut garantir l'encadrement de l'image exacte, et d'autre part cet encadrement doit être assez précis, c'est-à-dire on veut éviter des sur-encadrements grossiers. Discuter l'importance de ces prérequis pour la correction et l'efficacité de la méthode dichotomique. Puis analyser brièvement comment satisfaire ces exigences pour un polynôme comme $f(x) = x^6 - 9x^5 + 30x^4 - 40x^3 + 48x - 32$.

Exemple 0.2. La méthode dichotomique se généralise du cas unidimensionnel $f : \mathbb{R} \to \mathbb{R}$ aux fonctions $f : \mathbb{R}^m \to \mathbb{R}^n$, disons avec $m = n = 2$ pour fixer les idées. Si cela vous intéresse, vous pouvez regarder le film `www-sop.inria.fr/coprin/logiciels/ALIAS/Movie/movie_undergraduate.mpg` puis analyser la méthode proposée. Essayez d'abord de décrire l'algorithme plus en détail : comme avant la principale difficulté est de bien implémenter f selon l'arithmétique d'intervalles arrondie. Étant donnée une telle implémentation de f, formulez l'algorithme dichotomique puis prouver sa correction. (Si vous êtes courageux, vous pouvez l'implémenter en partant du programme `dichotomie.cc`.)

Dans ce chapitre nous chercherons à obtenir des méthodes itératives qui convergent plus rapidement et qui seront plus faciles à implémenter que la méthode dichotomique. Nous développons l'essentiel de la théorie, le théorème du point fixe et la méthode de Newton, sous une forme prête à programmer. Par contre, la problématique des calculs fiables sera (temporairement) négligé afin de ne pas encombrer la première présentation. Ceci est partiellement justifié par le fait qu'une fonction contractante soit numériquement stable, et lors des itérations les erreurs accumulées restent bornées (et on espère même négligeables).

1. La méthode du point fixe

1.1. Dynamique autour d'un point fixe. Nous allons nous intéresser aux itérations d'une fonction $f : E \to E$, où (E,d) est un espace métrique. Typiquement E est une partie de $\mathbb{R}^m$ et $d : E \times E \to \mathbb{R}$ est la distance euclidienne $d(x,y) = \sqrt{\sum_i (x_i - y_i)^2}$. La fonction f peut être compliquée, d'autant plus ses itérées $f^2 = f \circ f$, $f^3 = f \circ f \circ f$, $f^4 = f \circ f \circ f \circ f$, … Nous ne calculons jamais ces itérées toutes entières. Nous supposons seulement qu'il est facile à évaluer f en *un point* donné, et nous allons suivre sa trajectoire :

Définition 1.1. Étant donnée $f : E \to E$ et une valeur initiale x_0 nous construisons la *suite itérative* $(x_n)_{n\in\mathbb{N}}$ partant de x_0 par l'application itérée de la fonction f, de sorte que $x_{n+1} = f(x_n)$ pour tout $n \in \mathbb{N}$.

Le cas trivial est celui d'un *point fixe* de f, c'est-à-dire d'un point $a \in E$ tel que $f(a) = a$. Par contre, le comportement de f dans un voisinage d'un point fixe a peut nous donner des renseignements utiles :

Exemple 1.2. L'exemple le plus simple est une fonction linéaire $f : \mathbb{R} \to \mathbb{R}$, $f(x) = kx$ avec une constante $k \in \mathbb{R}$. Elle admet $a = 0$ pour point fixe. Pour $x_0 \neq 0$ deux phénomènes peuvent se produire :

(1) Si $|k| < 1$, par exemple $k = \frac{1}{2}$, alors les images successives $x_n = f^n(x_0)$ s'approchent de a, car $|f^n(x_0) - a| = |k|^n \cdot |x_0 - a| \to 0$. On dit que a est un point fixe *attractif* (ou *stable*).

(2) Si $|k| > 1$, par exemple $k = 2$, alors les images successives $x_n = f^n(x_0)$ s'éloignent de a, car $|f^n(x_0) - a| = |k|^n \cdot |x_0 - a| \to \infty$. On dit que a est un point fixe *répulsif* (ou *instable*).

Exemple 1.3. Plus généralement, étudions une fonction $f : \mathbb{R} \to \mathbb{R}$ que l'on suppose continûment dérivable. Comme avant nous supposons qu'elle admet un point fixe $a = f(a)$.

(1) Supposons que $|f'(a)| < 1$. Dans ce cas on peut choisir n'importe quelle constante k telle que $|f'(a)| < k < 1$, et la continuité de f' nous assure l'existence d'un voisinage $V = [a-\varepsilon, a+\varepsilon]$ tel que $|f'(\xi)| \leq k$ pour tout $\xi \in V$. Pour tout $x \in V$ nous pouvons ainsi majorer les accroissements : il existe ξ entre a et x tel que $f(x) - f(a) = f'(\xi)(x-a)$, et par conséquent

$$|f(x) - a| = |f(x) - f(a)| = |f'(\xi)(x-a)| \leq k|x-a|.$$

Ainsi les images itérées de $x \in V$ sont de plus en plus proches de a, plus précisément :

$$|f^n(x) - a| \leq k^n |x-a| \qquad \text{pour tout } n \in \mathbb{N}.$$

Autrement dit, a est un point fixe *attractif*.

(2) Réciproquement l'inégalité $|f'(a)| > 1$ implique que $|f'| \geq k > 1$ dans un voisinage V de a : toute valeur initiale $x \in V \smallsetminus \{a\}$ s'éloigne de a, dans le sens que $|f(x) - a| \geq k|x-a|$. Autrement dit, a est un point fixe *répulsif*.

☞ *Astuce.* — Dans ce cas on peut inverser f, au moins localement : la restriction $f|_V : V \to U$ est une bijection de V sur $U = f(V)$; son inverse $g = f|_V^{-1}$ est une fonction continûment dérivable avec $g(a) = a$ et $g'(a) = \frac{1}{f'(a)}$. (Le prouver.) On peut ainsi se ramener au premier cas.

(3) Dans le cas douteux $|f'(a)| = 1$, une analyse plus fine s'impose. Pour $x \mapsto x - x^3$ le point fixe $a = 0$ est attractif, pour $x \mapsto x + x^3$ il est répulsif. (Le détailler.) Pour $x \mapsto x + x^2$ il est attractif à gauche mais répulsif à droite, pour $x \mapsto x - x^2$ c'est l'inverse. (Le détailler.)

Remarque 1.4. En dimension ≥ 2 la situation peut être plus complexe. Considérons une application linéaire $\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) \mapsto \left(\begin{smallmatrix} \lambda & 0 \\ 0 & \mu \end{smallmatrix}\right) \left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$. Si $|\lambda|, |\mu| < 1$, le point fixe $a = 0$ est attractif. Si $|\lambda|, |\mu| > 1$, il est répulsif. Si $|\lambda| < 1 < |\mu|$, il existe une direction stable et une direction instable.

1.2. Espaces métriques. Étant donné un ensemble E et une suite $(x_n)_{n\in\mathbb{N}}$ dans E, comment définir la notion de convergence ? La façon la plus commode serait de disposer d'une notion de distance $d(x,y)$ entre deux points $x, y \in E$, c'est-à-dire une application $d : E \times E \to \mathbb{R}$. Pour que la distance se comporte comme on le souhaite, nous exigeons les trois propriétés suivantes :

Positivité: $d(x,y) \geq 0$ pour tout $x, y \in E$, avec $d(x,y) = 0$ si et seulement si $x = y$.

Symétrie: $d(x,y) = d(y,x)$ pour tout $x, y \in E$.

Inégalité triangulaire: $d(x,z) \leq d(x,y) + d(y,z)$ pour tout $x, y, z \in E$.

Si $d : E \times E \to \mathbb{R}$ satisfait à ces axiomes, on appelle d une *métrique* sur E, et la paire (E,d) est appelée un *espace métrique*. (Souvent on appelle E un espace métrique si d est sous-entendue sans équivoque.)

Exemple 1.5. Tout ensemble E peut être muni de la *métrique discrète* $d : E \times E \to \mathbb{R}$ définie par $d(x,y) = 0$ si $x = y$, et $d(x,y) = 1$ si $x \neq y$. (Exercice.) C'est gratuit et peu intéressant.

Exemple 1.6. Sur $\mathbb{R}$ on a bien sûr la métrique usuelle $d(x,y) = |x-y|$. L'ensemble $\mathbb{R}^m$ peut être muni de la métrique euclidienne $d_2(x,y) = \sqrt{\sum_k (x_k - y_k)^2}$. Plus généralement, pour tout $p \geq 1$, on obtient une métrique $d_p(x,y) = (\sum_k |x_k - y_k|^p)^{1/p}$. Pour $p = \infty$ on pose $d_\infty(x,y) = \sup_k |x_k - y_k|$.

(Pour ces métriques la positivité et la symétrie sont immédiates, mais l'inégalité triangulaire est délicate. C'est un bon exercice de révision si vous l'avez déjà vu.)

Exemple 1.7. Si (E,d) est un espace métrique et $F \subset E$ est une partie, alors (F, d_F) est un espace métrique muni de la métrique *induite* $d_F := d|_{F\times F} : F \times F \to \mathbb{R}$ obtenue par restriction. Ainsi toute partie de $\mathbb{R}^m$ est munie de la métrique d_p héritée de $\mathbb{R}^m$.

Définition 1.8. Soit (E,d) un espace métrique. On dit qu'une suite $(x_n)_{n\in\mathbb{N}}$ dans E *converge* vers $a \in E$ (pour la métrique d) si la distance $d(x_n,a)$ converge vers 0. Plus explicitement : pour tout $\varepsilon > 0$ il existe $N \in \mathbb{N}$ tel que $d(x_n,a) < \varepsilon$ pour tout $n \geq N$.

Exercice 1.9. Vérifier que sur $\mathbb{R}^1$ toutes les métriques d_p coïncident. Dans le plan $\mathbb{R}^2$ dessiner la « boule » $B(0,1) = \{x \in \mathbb{R}^2 \mid d(x,0) \leq 1\}$ de rayon 1 autour de 0 pour les métriques d_1, d_2, d_∞. Bien que ces métriques soient distinctes, la notion de convergence est la même : une suite x_n dans $\mathbb{R}^2$ converge vers a pour la métrique d_p si et seulement si x_n converge vers a pour la métrique d_q. Il existe des constantes $\alpha, \beta > 0$ telles que $\alpha d_p(x,y) \leq d_q(x,y) \leq \beta d_p(x,y)$. On dit que ces métriques sont *équivalentes*.

1.3. Fonctions contractantes. Souvent l'équation à résoudre se présente (ou peut se reformuler) comme un problème de point fixe $f(x) = x$. Cette approche se prête à une vaste généralité qui s'avère très utile. Soulignons d'abord le caractère purement *métrique* :

Définition 1.10. Soit (E,d) un espace métrique, soit $f : E \to E$ une application, et soit $k < 1$ une constante. On dit que f est *contractante* de rapport k si $d(f(x), f(y)) \leq k d(x,y)$ pour tout $x,y \in E$. Autrement dit, l'application f rapproche les points au moins de rapport k (toujours avec $k < 1$ fixé).

Remarque 1.11. Soit $I \subset \mathbb{R}$ un intervalle et soit $f : I \to \mathbb{R}$ une fonction dérivable. On pose $k := \sup_I |f'|$. Alors f est contractante si $k < 1$. Effectivement, pour tout $x,y \in I$ le théorème des accroissements finis nous assure qu'il existe ξ entre x et y tel que $f(x) - f(y) = f'(\xi)(x-y)$, donc $|f(x) - f(y)| \leq k|x-y|$.

Exercice 1.12. Prouver un critère analogue pour $f : I \to \mathbb{R}^m$ sur une partie convexe $I \subset \mathbb{R}^m$.
Attention. — Il ne suffit pas de supposer $I \subset \mathbb{R}^m$ connexe. (Esquisser un contre-exemple.)

Exercice 1.13. Comme on a vu, la convergence d'une suite x_n vers un point x est une propriété topologique : elle ne change pas lorsqu'on remplace la métrique d par une métrique équivalente. Par contre la propriété d'une fonction d'être contractante dépend de la métrique. En s'inspirant de la figure ci-dessus, regardons la fonction affine $f : \mathbb{R}^2 \to \mathbb{R}^2$ donnée par $f\binom{x}{y} = k\left(\begin{smallmatrix}\cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha\end{smallmatrix}\right) \cdot \binom{x}{y} + \binom{a}{b}$. Vérifier que pour la métrique d_2 cette application est contractante si et seulement si $k < 1$. Pour $\alpha = \frac{\pi}{4}$ et k proche de 1, l'application f n'est pas contractante par rapport à la métrique d_1, ni par rapport à la métrique d_∞.

1.4. Le théorème du point fixe. Intuitivement, une fonction contractante $f : E \to E$ se contracte sur un point, l'unique point fixe de f. Le but de ce paragraphe est justement d'établir ce théorème, dit de point fixe. Malheureusement l'intuition trompe facilement : le point crucial est l'*existence* d'un point fixe.

Exemple 1.14. Comme on a vu au chapitre XV, l'itération de la fonction $f(x) = \frac{1}{2}(x + \frac{2}{x})$ permet d'approcher la valeur de $\sqrt{2}$, l'unique point fixe de f. Restreint à l'intervalle $I = [1, +\infty[$ il s'agit d'une fonction contractante car la dérivée $f'(x) = \frac{1}{2} - \frac{1}{x^2}$ est à valeur dans $[-\frac{1}{2}, \frac{1}{2}[$.

Dans la pratique on ne calcule qu'avec des nombres rationnels, et sur $E = [1, +\infty[\cap \mathbb{Q}$ la restriction $h = f|_E : E \to E$ est toujours contractante. Mais, malheureusement, elle n'a plus de point fixe dans E : le problème est que l'espace E contienne des « trous », il n'est pas ce que l'on appelle *complet*.

Définition 1.15. Une suite $(x_n)_{n\in\mathbb{N}}$ dans un espace métrique (E,d) est dite *de Cauchy* si pour tout $\varepsilon>0$ il existe $N\in\mathbb{N}$ tel que $d(x_n,x_m)<\varepsilon$ pour tout $n,m\geq N$.

Toute suite convergente est de Cauchy mais la réciproque est fausse en général : regarder une suite dans $\mathbb{Q}$ qui converge vers $\sqrt{2}\in\mathbb{R}$. On arrive ainsi à la définition suivante :

Définition 1.16. Un espace métrique (E,d) est *complet* si toute suite de Cauchy converge.

Remarque 1.17. L'espace $\mathbb{R}$ avec la métrique usuelle est complet. L'espace $\mathbb{R}^m$ avec la métrique d_p est complet, quelque soit $p\in[1,\infty[$. Il en est de même pour tout sous-ensemble *fermé*. (Le montrer.) Par exemple, tout intervalle fermé $[a,b]\subset\mathbb{R}$ avec la métrique induite est un espace métrique complet. Par contre $\mathbb{Q}\subset\mathbb{R}$ n'est pas complet.

Les espaces métriques complets sont très importants en analyse parce qu'ils permettent de *construire* certains objets comme limites de suites de Cauchy, l'existence étant assurée par l'hypothèse de complétude. En voici un exemple fondamental aussi bien pour la théorie que pour le calcul numérique :

Théorème 1.18 (le théorème du point fixe)**.** *Soit (E,d) un espace métrique complet et soit $f\colon E\to E$ une application contractante de rapport $k<1$. Alors :*

(1) *Il existe un et un seul point $a\in E$ vérifiant $f(a)=a$.*

(2) *Pour tout $x_0\in E$ la suite itérative $x_{n+1}=f(x_n)$ converge vers a, vérifiant $d(x_n,a)\leq k^n d(x_0,a)$. La convergence est donc au moins aussi rapide que celle de la suite géométrique $k^n\to 0$.*

(3) *Pour le calcul concret on a la majoration d'erreur $d(x_n,a)\leq\frac{k}{1-k}d(x_n,x_{n-1})$.*

Remarque. — Pour le calcul concret on suppose, comme dans les exemples précédents, que l'on sache calculer $x_{n+1}=f(x_n)$ à partir de x_n. Par contre on ignore typiquement la valeur limite a. Dans cette situation l'inégalité $d(x_n,a)\leq\frac{k}{1-k}d(x_n,x_{n-1})$ nous est très utile car elle permet de majorer la distance de la valeur approchée x_n à la valeur inconnue a en fonction de x_n et x_{n-1} seulement. Contrairement à $d(x_n,a)$, la quantité $\frac{k}{1-k}d(x_n,x_{n-1})$ est en général très facile à calculer.

DÉMONSTRATION. *Unicité.* — Si $a,b\in E$ sont deux points fixes d'une fonction contractante de rapport $k<1$, alors $d(a,b)=d(f(a),f(b))\leq kd(a,b)$ entraîne $d(a,b)=0$ donc $a=b$.

Existence. — Une récurrence facile montre $d(x_{n+1},x_n)\leq k^n d(x_1,x_0)$ pour tout $n\in\mathbb{N}$, puis

$$\begin{aligned} d(x_{n+p},x_n) &\leq d(x_{n+p},x_{n+p-1})+\cdots+d(x_{n+2},x_{n+1})+d(x_{n+1},x_n)\\ &\leq (k^{p-1}+\cdots+k^1+k^0)d(x_{n+1},x_n)\leq\tfrac{k^n}{1-k}d(x_1,x_0)\end{aligned}$$

pour tout $n,p\in\mathbb{N}$. La suite $(x_n)_{n\in\mathbb{N}}$ est donc de Cauchy et converge puisque E est complet. Notons a sa limite, et vérifions qu'il s'agit d'un point fixe : l'application f, étant contractante, est continue. L'équation de récurrence $x_{n+1}=f(x_n)$ donne donc $a=\lim x_{n+1}=\lim f(x_n)=f(\lim x_n)=f(a)$.

Vitesse de convergence. — On a $d(x_n,a)\leq k^n d(x_0,a)$, donc pour toute valeur initiale $x_0\in E$ la suite itérative $x_{n+1}=f(x_n)$ converge vers le point fixe a. Pour estimer la distance de x_n à a, on a la majoration $d(x_{n+p},x_n)\leq\frac{k}{1-k}d(x_n,x_{n-1})$. Le passage à la limite $p\to\infty$ donne l'inégalité cherchée. □

Exemple 1.19. Dans notre exemple préféré la fonction $f\colon[1,2]\to[1,2]$, $f(x)=\frac{1}{2}(x+\frac{2}{x})$ est contractante de rapport $k=\frac{1}{2}$. Le théorème affirme donc que pour toute valeur initiale $x_0\in[1,2]$ la suite itérative $x_{n+1}=f(x_n)$ converge vers l'unique point fixe $a=\sqrt{2}$, et que $|x_n-a|\leq(\frac{1}{2})^n|x_0-a|$. (La convergence est en fait beaucoup plus rapide que celle de la suite géométrique $\left(\frac{1}{2}\right)^n\to 0$, voir plus bas.)

Remarque 1.20. La majoration $d(x_n,a)\leq k^n d(x_0,a)$, donne $-\log d(x_n,a)\geq-\log d(x_0,a)-n\log(k)$. Dans le cas $E=\mathbb{R}$ le nombre de décimales exactes de x_n comme approximation de a est donc minoré par une fonction affine de n. On parle d'une *convergence linéaire*. On remarquera que la constante de contraction $k<1$ influence sensiblement la vitesse de convergence.

Exemple 1.21. Soulignons que la majoration $d(x_n,x_{n-1})<\varepsilon$ n'implique pas forcément que $d(x_n,a)<\varepsilon$. Il faut tenir compte du facteur $\frac{k}{1-k}$: appliquons la condition d'arrêt $|x_n-x_{n-1}|<10^{-4}$ à l'itération de $f\colon\mathbb{R}\to\mathbb{R}$, $f(x)=0,999999\cdot x$ et la valeur initiale $x_0=100$. Cette fonction est contractante, mais seulement très faiblement : $x_0=100$ et $x_1=99,9999$ sont proches, mais encore loin du point fixe $a=0$.

Exemple 1.22. Pour appliquer le théorème il faut soigneusement vérifier les hypothèses. À titre d'avertissement, voici quelques exemples voisins où le théorème ne s'applique pas :

(1) $f\colon [0,1] \to \mathbb{R}$, $f(x) = 1 + \frac{1}{2}x$ est contractante mais n'admet pas de point fixe. Pourquoi ?

(2) $g\colon]0,1] \to]0,1]$, $g(x) = \frac{1}{2}x$ est contractante mais n'admet pas de point fixe. Pourquoi ?

(3) Sur $E = [1,2] \cap \mathbb{Q}$ la fonction $h(x) = \frac{1}{2}(x + \frac{2}{x})$ n'admet pas de point fixe. Pourquoi ?

(4) La fonction $f\colon \mathbb{R} \to \mathbb{R}$, $f(x) = x + \frac{1}{1+e^x}$ vérifie $0 < f'(x) < 1$. Est-elle contractante ? sur un compact $[a,b]$? Admet-elle un point fixe ? Pourquoi le théorème ne s'applique-t-il pas ?

1.5. Quelques applications. Pour $f(a) = a$ et $|f'(a)| < 1$ nous avons vu qu'il existe un voisinage $V = [a-\varepsilon, a+\varepsilon]$ sur lequel la fonction f est contractante de rapport $k = \sup_V |f'| < 1$. Pour tout $x \in V$ nous avons $|f(x) - a| \leq k|x-a|$, donc $f(V) \subset V$, et nous pouvons appliquer le théorème : pour tout $x_0 \in V$ on a convergence $x_n \to a$ de vitesse (au moins) linéaire $|x_n - a| \leq k^n |x_0 - a|$.

Remarque 1.23. Dans la pratique, on se donne souvent une fonction $f\colon \mathbb{R} \to \mathbb{R}$. Il faut d'abord *expliciter* un fermé $V \subset \mathbb{R}$ tel que $f(V) \subset V$ et une constante k telle que $\sup_V |f'| \leq k < 1$. Ceci correspond à *localiser* le point fixe a en explicitant un voisinage V qui soit suffisamment précis pour appliquer le théorème.

☞ Plus la constante k est petite, plus la convergence $k^n \to 0$ est rapide. Comme k est minorée par $|f'(a)|$, on a intérêt à choisir plutôt un *petit* voisinage V de a, si possible, afin de minimiser k.

☞ Si $\sup_V |f'| \geq 1$, alors on a choisi V trop grand et il faut recommencer avec un fermé mieux adapté. Si l'on sait d'avance que V ne contient pas de point fixe, ou si le point fixe a vérifie $|f'(a)| \geq 1$, il est inutile d'insister : le théorème ne pourra pas s'appliquer.

Exemple 1.24. On se propose d'approcher l'unique solution $x \in \mathbb{R}$ de l'équation $x = \cos x$.

Exercice préparatoire. — Tracer sommairement le graphe de $\cos x$ afin de localiser la solution. (Il n'y a qu'une seule.) Déterminer un intervalle $[a,b]$ sur lequel le théorème du point fixe s'applique. Quelle constante de contraction obtenez-vous ?

Calcul numérique. — Le programme `iter1.cc` calcule la suite itérative $x_{n+1} = \cos x_n$ à partir de $x_0 = 0$. La suite converge numériquement, mais cette observation empirique ne remplace pas l'analyse mathématique précédente. Étant donnée la constante de contraction k, le programme peut déterminer la marge d'erreur $\frac{k}{1-k}|x_n - x_{n-1}|$. Ainsi on itère jusqu'à la précision ε souhaitée, et on peut conclure que $|x_n - x| \leq \varepsilon$.

Exercice 1.25. Encadrer les solutions réelles de l'équation $\exp(x) = x^3$ à 10^{-8} près.

Indication. — Reformuler le problème sous forme de point fixe, $f(x) = x$. Tracer sommairement le graphe de f ; il y a deux solutions. Les points fixes sont ils attractifs ou répulsifs ? Dans le cas attractif, appliquer le théorème du point fixe. Dans le cas répulsif passer à la fonction inverse f^{-1}.

Exercice/M 1.26. On se propose d'analyser l'application $f\colon \mathbb{R}^2 \to \mathbb{R}^2$, $f\left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right) = \left(\begin{smallmatrix} -x_1 + \frac{3}{2}x_2 + \frac{5}{4} \\ -\frac{1}{2}x_1 + x_2^2 + \frac{3}{4} \end{smallmatrix}\right)$.

Déterminer les points fixes de f, puis calculer la dérivée $(\frac{\partial f_i}{\partial x_j})$ dans ces points. Quelles sont les valeurs propres ? Les points fixes sont-ils attractifs ? Soit x le point fixe non attractif de f. Montrer que f est localement inversible autour de x, c'est-à-dire il existe un voisinage V de x tel que $f|_V\colon V \to U$ soit une bijection sur $U := f(V)$, et que l'inverse $g = f|_V^{-1}\colon U \to V$ soit une application continûment dérivable. Est-ce que le point fixe x est attractif pour g ?

Exercice/M 1.27. Quels sont les points fixes de l'application $f\colon \mathbb{R}^2_+ \to \mathbb{R}^2_+$, $(x,y) \mapsto (\frac{x+y}{2}, \sqrt{xy})$? Écrire un programme qui calcule une valeur approchée de la limite $\lim_{n\to\infty} f^n(x,y)$ et admirer la vitesse.

Remarque. — La limite $AGM(x,y) := \lim f^n(x,y)$ est un compromis astucieux, inventé par Gauss, entre la moyenne arithmétique $\frac{x+y}{2}$ et la moyenne géométrique $\sqrt{xy}$. Pour cette raison on appelle la valeur $AGM(x,y)$ la *moyenne arithmético-géométrique*, ou *arithmetic-geometric mean* en anglais.

Quelques pistes de réflexion. — Comment expliquer la convergence super-rapide ? La fonction f est-elle contractante ? Est-ce que la suite itérative $f^n(x,y)$ converge pour tout $(x,y) \in \mathbb{R}^2_+$? Dans quel sens est-ce que la fonction f « contracte vers la diagonale » ? Pour un point (x,x) de la diagonale calculer la dérivée dans la direction orthogonale $(1,-1)$. Dans quel sens la diagonale est-elle « super-attractive » ? (Voir la suite.)

2. La méthode de Newton

2.1. Vitesse de convergence. Regardons à nouveau la dynamique autour d'un point fixe attractif : soit $\phi\colon \mathbb{R}\to\mathbb{R}$ une fonction continûment dérivable, et soit $a=\phi(a)$ un point fixe vérifiant $|\phi'(a)|<1$. On a vu que dans un voisinage $V=[a-\varepsilon,a+\varepsilon]$, la fonction ϕ est contractante de rapport $k=\sup_V|\phi'|<1$, et pour tout $x_0\in V$ on a convergence linéaire : $|\phi^n(x_0)-a|\le k^n|x_0-a|$. Le cas $\phi'(a)=0$ est particulier : quand $\phi^n(x_0)$ approche a, la fonction ϕ contracte de plus en plus fortement, ce qui accélère la convergence :

Remarque 2.1. Soit $\phi\colon\mathbb{R}\to\mathbb{R}$ de classe C^2 telle que $\phi(a)=a$ et $\phi'(a)=0$. Soit $V=[a-\varepsilon,a+\varepsilon]$ et $M:=\max_V|\phi''|$. D'après le développement de Taylor, pour tout x il existe ξ entre a et x tel que

$$\phi(x)=\phi(a)+\phi'(a)(x-a)+\frac{1}{2}\phi''(\xi)(x-a)^2.$$

Pour tout $x_0\in V$ ceci implique $|\phi(x_0)-a|\le\frac{M}{2}|x_0-a|^2$. On conclut que pour toute valeur initiale $x_0\in V$ vérifiant $|x_0-a|<\frac{2}{M}$ on a une convergence particulièrement rapide :

$$\tfrac{M}{2}|\phi^n(x_0)-a|\le\left(\tfrac{M}{2}|x_0-a|\right)^{2^n}.$$

Ceci veut dire que le nombre de décimales exactes de $x_n=\phi^n(x_0)$ comme approximation de a double à chaque itération ! C'est cette convergence super-rapide qui fait de cette observation un outil très puissant : si vous avez calculé une approximation x_n avec un écart $\frac{M}{2}|x_n-a|\le 2^{-1}$ près, disons, l'itération suivante ne laissera qu'un écart $\le 2^{-2}$, celle d'après $\le 2^{-4}$, puis $\le 2^{-8}$, puis $\le 2^{-16}$, puis $\le 2^{-32}$ etc.

Définition 2.2. Dans un espace métrique (E,d) soit $(x_n)_{n\in\mathbb{N}}$ une suite convergente de limite a. On note $e_n=d(x_n,a)$ l'écart entre x_n et la limite cherchée a, et on suppose $e_n>0$ pour tout n.

(1) On dit que la convergence est asymptotiquement (au moins) *linéaire* si $\limsup\frac{e_{n+1}}{e_n}<1$.
Ceci équivaut à dire qu'il existe $k<1$ et $n_0\in\mathbb{N}$ tels que $e_{n+1}\le ke_n$ pour tout $n\ge n_0$.
Dans le cas $E=\mathbb{R}$, le nombre de décimales exactes est minoré par une fonction affine de n.

(2) On dit que la convergence est asymptotiquement (au moins) *quadratique* si $\limsup\frac{e_{n+1}}{e_n^2}<+\infty$.
Ceci équivaut à dire qu'il existe $K\in\mathbb{R}_+$ tel que $e_{n+1}\le Ke_n^2$ pour tout $n\in\mathbb{N}$.
Dans le cas $E=\mathbb{R}$, le nombre de décimales exactes double à peu près à chaque itération.

Exemple 2.3. Le théorème du point fixe promet une convergence linéaire. Nous avons vu la convergence quadratique de la méthode de Newton-Héron au chapitre XV, §3.1.

Proposition 2.4. *Soit $U\subset\mathbb{R}^m$ un ouvert et $\phi\colon U\to\mathbb{R}^m$ une application de classe C^2. Si $\phi(a)=a$ et $\phi'(a)=0$ alors existe un voisinage V de a tel que $\phi|_V$ soit contractante de rapport $\frac{1}{2}$. Par conséquent :*

(1) *On a $\phi(V)\subset V$ et pour tout $x_0\in V$ la suite itérative $x_n=\phi^n(x_0)$ converge vers a.*

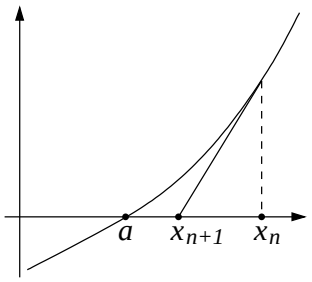
(2) *La vitesse de convergence est au moins linéaire, $|x_n-a|\le 2^{-n}|x_0-a|$.*

(3) *Finalement la convergence devient quadratique, $|x_{n+1}-a|\le\frac{M}{2}|x_n-a|^2$.*

Exercice/M 2.5. Prouver cette proposition dans le cas $\mathbb{R}^m$ en généralisant les arguments donnés pour $\mathbb{R}$. (Cf. Demailly, §IV.2.3.) Ce résultat motive la définition suivante :

Définition 2.6. Soit $\phi\colon\mathbb{R}^m\supset U\to\mathbb{R}^m$ de classe C^1. Un point fixe $a=\phi(a)$ est *super-attractif* si $\phi'(a)=0$. Un voisinage V de a est dit *newtonien* pour ϕ s'il satisfait aux conditions (1), (2), (3) ci-dessus.

2.2. Itération de Newton. On cherche à résoudre une équation $f(x)=0$ dans $\mathbb{R}^m$ dont on connaît une solution approchée x_n. La méthode de Newton permet de transformer l'équation $f(x)=0$ en un problème de point fixe $\phi(x)=x$. Son intérêt réside dans le fait que les zéros de f deviennent des points fixes *super-attractifs* pour ϕ, ce qui nous permettra des calculs extrêmement efficaces.



Au lieu d'une approche purement métrique à la méthode du point fixe, on veut tirer profit du calcul différentiel ! On approche donc la fonction f par la tangente en x_n, c'est-à-dire $t(x):=f(x_n)+f'(x_n)(x-x_n)$. C'est l'approximation de Taylor d'ordre 1.

Pour x_{n+1} on prendra l'unique solution de l'équation affine $t(x)=0$, à savoir $x_{n+1}=x_n-f'(x_n)^{-1}f(x_n)$. Autrement dit, pour approcher les zéros de f on itère l'application $\phi(x)=x-f'(x)^{-1}f(x)$.

Théorème 2.7 (méthode de Newton, version qualitative)**.** *Soit $U\subset\mathbb{R}^m$ un ouvert et soit $f\colon U\to\mathbb{R}^m$ une fonction de classe C^2 telle que $f'(x)$ soit inversible pour tout $x\in U$. Alors la fonction $\phi\colon U\to\mathbb{R}^m$ définie par $\phi(x)=x-f'(x)^{-1}f(x)$ est de classe C^1, les points fixes de ϕ sont exactement les zéros de f, et en tout point fixe $a=\phi(a)$ la dérivée s'annule : $\phi'(a)=0$.*

Les zéros de f deviennent ainsi des points fixes super-attractifs de ϕ, comme souhaité, et dans un voisinage convenable on a convergence quadratique :

Corollaire 2.8. *Soit $f\colon\mathbb{R}^m\supset U\to\mathbb{R}^m$ une fonction de classe C^3 et soit $a\in U$ une racine simple de f, c'est-à-dire $f(a)=0$ à dérivée $f'(a)$ inversible. Alors il existe un rayon $\delta>0$ de sorte que la boule $B(a,\delta)=\{x\in\mathbb{R}^m\mid|x-a|\leq\delta\}$ soit contenue dans U et soit newtonienne pour ϕ.*

Exercice/M 2.9. Prouver ce théorème et son corollaire (pour $m=1$ si vous voulez simplifier). Qu'obtient-on quand on l'applique à $f\colon\mathbb{R}_+\to\mathbb{R}$, $f(x)=x^n-a$? Pour quelles valeurs initiales x_0 l'itération de Newton converge-t-elle ? Comparer avec le résultat du chapitre XV, §3.1 : en quoi notre formulation du théorème de Newton-Héron est-elle plus précise ou plus générale que le résultat qualitatif précédent ?

☞ *Heuristique : Si $f(a)=0$ et l'on dispose d'une valeur initiale $x_0\approx a$ suffisamment proche de a, alors l'itération de Newton $x_n=\phi^n(x_0)$ converge très rapidement vers a.* ✍

2.3. Exemples. En vue d'une implémentation sur ordinateur nous sous-entendons ici qu'il est raisonnablement facile d'évaluer f et f' en x_n, c'est-à-dire on sait implémenter des approximations de $f(x_n)$ et de $f'(x_n)$ avec une précision et une efficacité satisfaisantes. C'est le cas pour tous nos exemples, mais pour un problème réaliste cette étape peut en elle-même nécessiter une analyse approfondie. Nous supposerons ce problème résolu par une bibliothèque convenable.

Exemple 2.10. On reprend l'équation $\cos(x)=x$ qui a été résolue par le programme `iter1.cc` en utilisant la méthode du point fixe. Testez le programme `iter2.cc` qui résout cette équation par la méthode de Newton, et admirez la vitesse de convergence.

Exercice/P 2.11. Si vous voulez, vous pouvez reprendre l'équation $\exp(x)=x^3$, résolue plus haut par la méthode du point fixe, et approcher ses deux solutions par la méthode de Newton.

Remarque 2.12. Avant d'appliquer la méthode de Newton il faut bien vérifier que l'on est en présence d'un *zéro simple*, c'est-à-dire $f(a)=0$ mais $f'(a)$ inversible. Dans ce cas il existe un voisinage U tel que $f'(x)$ reste inversible pour tout $x\in U$, et on peut appliquer le théorème précédent.

2.4. Bassin d'attraction. La convergence de la méthode de Newton n'est assurée que pour une valeur initiale proche d'un zéro. Nos résultats précédents motivent la définition suivante :

Définition 2.13 (Rayon de Newton)**.** Soit $f\colon\mathbb{R}^m\supset U\to\mathbb{R}^m$ une fonction de classe C^2 et soit $a\in U$ une racine simple de f, c'est-à-dire $f(a)=0$ à dérivée $f'(a)$ inversible. Dans ce cas on appelle

$$\rho_a:=\sup\{\delta\in\mathbb{R}_+\mid\text{la boule } B(a,\delta)\subset U\text{ est newtonienne pour }\phi\}$$

le *rayon de Newton* de la racine a.

Remarque 2.14. Le corollaire 2.8 assure que $\rho_a>0$ (pourvu que f est de classe C^3, mais C^2 suffit d'après le théorème 2.20 plus bas). Une fois on dispose d'une valeur approchée x_0 de la racine a à une marge $\leq\rho_a$ près, on peut approcher a aussi précisément que l'on le souhaite. Il suffit d'itérer la méthode de Newton : la convergence est assurée et on peut même garantir une bonne vitesse de convergence !

Remarque 2.15. Comment majorer la marge d'erreur entre l'approximation calculée x_n (mais erronée) et la valeur exacte cherchée a (mais inconnue) ? La formulation ci-dessus fait semblant de connaître a, mais dans la pratique on ignore la valeur exacte et on ne dispose que de la valeur approchée. Une fois on est sûr d'être dans un voisinage newtonien, le théorème du point fixe nous dit que $|x_n-a|\leq|x_n-x_{n-1}|$.

Remarque 2.16. Dans un cours d'analyse on se contente souvent de la formulation *qualitative* du théorème 2.7 donnée ci-dessus. Quant à l'application *pratique* avouons qu'elle laisse encore à désirer :

(1) Comment déterminer (ou au moins minorer) le rayon de Newton ρ_a d'une racine a ? La formulation qualitative ne donne pas de réponse explicite, elle assure seulement que $\rho_a > 0$. Une version quantitative sera donnée par le théorème 2.20 plus bas.

(2) Comment savoir pour une valeur initiale x_0 si l'on est dans le rayon de Newton d'une racine a ? Normalement on ne connaît pas les racines, donc on veut un critère en fonction de x_0 seulement. Un tel critère sera donné par le théorème 2.25.

(3) Comment trouver une approximation initiale x_0 telle que l'on puisse garantir la convergence rapide $x_n = \phi^n(x_0) \to a$? Certes, il suffit de localiser la racine a à ρ_a près, mais comment faire ? En général, partant juste de la fonction f sans aucune connaissance au préalable de ses racines, c'est un problème délicat qu'il faut analyser au cas par cas.

☞ *L'algorithme de Newton est une méthode* locale *et non* globale *: il faut de bonnes valeurs initiales pour assurer sa convergence et son efficacité.* ✍

Exercice 2.17. À titre d'exemple, analyser $f : \mathbb{R} \to \mathbb{R}$, $f(x) = \arctan(x)$ afin d'illustrer la convergence ou divergence de la méthode de Newton. L'unique zéro est $a = 0$. Tracer le graphe de f et déterminer pour quelles valeurs initiales x_0 l'itération de Newton $\phi^n(x_0)$ converge, puis pour lesquelles elle diverge. Que se passe-t-il pour les cas critiques intermédiaires ? (L'équation $\arctan(x) = 2x/(1+x^2)$ peut elle-même être résolue par la méthode de Newton.) Construire ainsi une suite itérative de Newton qui converge, mais dont la convergence est initialement aussi lente que l'on veut. Déterminer finalement le rayon de Newton ρ_a.

Exercice 2.18. Pour les fonctions unidimensionnelles $f : \mathbb{R} \to \mathbb{R}$ il est très commode de donner des critères « géométriques » de convergence pour de l'itération Newton $x_n = \phi^n(x_0)$:

(1) Supposons que $f(a) = 0$ ainsi que $f' > 0$ et $f'' \geq 0$ sur $I = [a, a+\varepsilon]$. Alors $\phi(I) \subset I$ et pour toute valeur initiale $x_0 \in I$ on obtient une suite décroissante $x_0 \geq x_1 \geq x_2 \geq \dots$ qui converge vers a.

(2) Supposons que $f(a) = 0$ ainsi que $f' > 0$ et $f'' \leq 0$ sur $I = [a-\varepsilon, a]$. Alors $\phi(I) \subset I$ et pour toute valeur initiale $x_0 \in I$ on obtient une suite croissante $x_0 \leq x_1 \leq x_2 \leq \dots$ qui converge vers a.

Dans les deux cas on obtient la majoration d'erreur

$$|x_n - a| \leq \left| \frac{f(x_n)}{f'(a)} \right|.$$

En particulier, pour $f'(a) > 0$ et $f''(a) > 0$ il suffit de commencer l'itération légèrement à droite de a afin de garantir $x_n \to a$. Pour $f'(a) > 0$ et $f''(a) < 0$ il suffit de commencer légèrement à gauche de a. La taille exacte de l'intervalle peut être déterminé en étudiant f''. Que faire dans le cas $f'(a) < 0$? Dessin !

Exercice 2.19. On considère la fonction polynomiale

$$f : \mathbb{R} \to \mathbb{R}, \qquad f(x) = 3x^3 - 10x^2 - 4x + 8.$$

(1) Montrer que f admet trois racines réelles distinctes, que l'on notera $a, b, c \in \mathbb{R}$ avec $a < b < c$. Encadrer ces racines par trois intervalles de la forme $]k, k+1[$ avec $k \in \mathbb{Z}$. Dans la suite on veut appliquer la méthode de Newton pour approcher les racines a, b, c avec plus de précision.

(2) Sur quels intervalles la fonction f est-elle croissante/décroissante ? Sur quels intervalles est-elle concave/convexe ? En déduire une valeur initiale $c_0 \in \mathbb{Z}$ pour laquelle vous pouvez garantir que la méthode de Newton donne une suite récurrente $c_n = \phi^n(c_0)$ qui converge vers la racine c. (Justifiez votre choix.) Même question pour a_0 tel que $a_n \to a$, puis b_0 tel que $b_n \to b$.

(3) Calculer les valeurs approchées a_3, b_3, c_3 des trois racines a, b, c par la méthode de Newton, comme préparée ci-dessus, en effectuant 3 itérations pour chacune. Majorer l'écart $|a_3 - a|$, $|b_3 - b|$, $|c_3 - c|$. Vérifier les approximations en développant $f_3(x) = 3(x-a_3)(x-b_3)(x-c_3)$. Obtient-on $f(x)$ exactement ? Que se passe-t-il pour cinq puis dix itérations ?

(4) Exhiber une valeur initiale u_0 pour laquelle la méthode de Newton donne une suite $(u_n)_{n\in\mathbb{N}}$ qui ne converge pas. Pourquoi ceci ne met pas en cause le théorème 2.7 ?

2.5. Version quantitative. Le théorème 2.7 reste seulement qualitatif dans le sens qu'il n'explicite pas de voisinage sur lequel il garantit une convergence. Les exercices précédents montrent des critères géométriques pour les fonctions convexes/concaves, ce qui fait intervenir f''. Le théorème suivant donne une version quantitative qui explicite un intervalle sur lequel on peut garantir la convergence :

Théorème 2.20 (méthode de Newton, version quantitative)**.** *Soit $f\colon \mathbb{R}\to\mathbb{R}$ de classe C^2. Supposons que $f(a)=0$ ainsi que $|f'|\geq m>0$ et $|f''|\leq M$ sur un intervalle $I=[a-\varepsilon,a+\varepsilon]$ avec $\varepsilon>0$. Nous posons $\delta:=\min(\varepsilon,m/M)$ et $V:=[a-\delta,a+\delta]$. Alors pour toute valeur initiale $x_0\in V$ la suite $x_n=\phi^n(x_0)$ vérifie*

$$|x_n-a|\leq\left(\frac{1}{2}\right)^{2^n-1}|x_0-a|.$$

En particulier V est un voisinage newtonien de a dans le sens de la définition 2.6.

DÉMONSTRATION. Par translation nous pouvons supposer que $a=0$. Nous avons

$$x_{n+1}=\phi(x_n)=x_n-f'(x_n)^{-1}f(x_n)=f'(x_n)^{-1}\left(\,f'(x_n)x_n-f(x_n)\,\right).$$

D'autre part le développement de Taylor en x_n nous donne

$$0=f(0)=f(x_n)-f'(x_n)x_n+\frac{1}{2}f''(\xi_n)x_n^2$$

pour un $\xi_n\in[0,x_n]$ convenable. De ces deux équations on déduit

$$x_{n+1}=\frac{1}{2}\,f'(x_n)^{-1}\,f''(\xi_n)\,x_n^2.$$

Avec $|f'|\geq m>0$ et $|f''|\leq M$ ceci donne

$$|x_{n+1}|\leq\frac{M}{2m}|x_n|^2$$

ou encore $\frac{M}{2m}|x_{n+1}|\leq(\frac{M}{2m}|x_n|)^2$. Par récurrence on obtient

$$\frac{M}{2m}|x_n|\leq\left(\frac{M}{2m}|x_0|\right)^{2^n}.$$

Pour $x_0\in V$ on a $|x_0|\leq\delta\leq\frac{m}{M}$, donc $\frac{M}{2m}|x_0|\leq\frac{1}{2}$. On conclut que $|x_n|\leq\left(\frac{1}{2}\right)^{2^n-1}|x_0|$. □

Exercice/M 2.21. Vérifier soigneusement le théorème précédent, puis essayer de formuler et de prouver une version pour les fonctions complexes $f\colon\mathbb{C}\to\mathbb{C}$, ou plus généralement pour $f\colon\mathbb{R}^n\to\mathbb{R}^n$.

Exercice/M 2.22. La méthode de Newton est assez universelle et sa convergence foudroyante en fait un outil omniprésent. En voici un exemple plus poussé, d'après Demailly, §IV, qui sert pour l'inversion des matrices :

Soit A une algèbre unitaire normée sur $\mathbb{R}$, par exemple les matrices carrées $m\times m$.

(a) Soit $u\in A$ un élément inversible. Déterminer deux constantes $\alpha,\beta\in\mathbb{R}$ de sorte que l'application $\phi\colon A\to A$, $\phi(x)=\alpha x+\beta xux$, admette u^{-1} comme point fixe super-attractif. Montrer que $|\phi'(x)|\leq 2|u|\cdot|x-u^{-1}|$. En déduire que la suite $x_n=\phi^n(x_0)$ converge vers u^{-1} pour tout $x_0\in B(u^{-1},r)$ avec $r<\frac{1}{2|u|}$.

(b) On suppose désormais que A est complète. (C'est automatique si A est de dimension finie.) En supposant $|v|<1$ montrer que $u=1-v$ est inversible et $u^{-1}=\sum_{k=0}^{\infty}v^k$. En quoi la méthode (a) est-elle plus efficace ? Déterminer un entier $n\in\mathbb{N}$ de sorte que la méthode (a) converge pour $x_0=1+v+\cdots+v^n$.

Pour résumer : quels sont les avantages et inconvénients des deux méthodes (a) et (b) utilisées séparément ? Quel est l'intérêt de les combiner ?

2.6. Critères pratiques. La problématique de choisir de bonnes valeurs initiales pour la méthode de Newton reste d'actualité. On peut se poser une question plus facile : étant donné z_0, comment savoir si l'itération de Newton $z_n=\phi^n(z_0)$ convergera ? Le théorème 2.20 part de la racine a et minore son rayon de Newton. Ceci n'est souvent pas pratique car on ignore les racines encore à trouver. Le critère suivant, par contre, part d'une valeur initiale z_0 donnée :

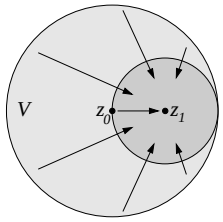
Théorème 2.23. *Considérons une fonction analytique $f\colon \mathbb{C} \supset U \to \mathbb{C}$ avec son application de Newton $\phi(z) = z - \frac{f(z)}{f'(z)}$. Soit $z_0 \in U$ une valeur initiale telle que $f(z_0) \neq 0$ et $f'(z_0) \neq 0$, et soit $r := \frac{|f(z_0)|}{|f'(z_0)|}$ le pas initial dans l'itération de Newton. Supposons que la boule $B(z_0, 2r)$ soit incluse dans U, et que*

$$\frac{|f''(z)|}{|f'(z)|} \leq \frac{1}{8}\frac{|f'(z_0)|}{|f(z_0)|} \quad \text{pour tout } z \in B(z_0, 2r) =: V.$$

Alors $\phi|_V$ est contractante de rapport $\frac{1}{2}$ et vérifie $\phi(V) \subset V$. Par conséquent $f|_V$ admet une unique racine $a \in V$, et la suite itérative $z_n = \phi^n(z_0)$ converge vers a. La convergence est quadratique comme expliqué dans la proposition 2.4.

Remarque. — Par hypothèse on a $|\phi(z_0) - z_0| = r$. La première itération reste donc bien dans $V = B(z_0, 2r)$, mais il faut encore contrôler les suivantes. L'idée est de montrer que $\phi|_V$ est contractante de rapport $\frac{1}{2}$, c'est-à-dire $|\phi(x) - \phi(y)| \leq \frac{1}{2}|x - y|$ pour tout $x, y \in V$. Ceci entraîne $\phi(V) \subset V$, car pour tout z avec $|z - z_0| \leq 2r$ on trouve $|\phi(z) - \phi(z_0)| \leq r$ et donc

$$|\phi(z) - z_0| \leq |\phi(z) - \phi(z_0)| + |\phi(z_0) - z_0| \leq 2r.$$



DÉMONSTRATION. Après translation on peut supposer que $z_0 = 0$. Par hypothèse f' ne s'annule pas dans $V = B(0, 2r)$, donc ϕ est définie sur V. Pour montrer que $\phi|_V$ est contractante de rapport $\frac{1}{2}$ on montrera que sa dérivée $\phi'(z) = f(z)f''(z)/f'(z)^2$ vérifie $|\phi'| \leq \frac{1}{2}$ sur V. Étudions le quotient $u(z) = f(z)/f'(z)$ et sa dérivée $u'(z) = 1 - u(z)f''(z)/f'(z)$. On a $|u(0)| = r$ et $|u'(z)| \leq 1 + M|u(z)|$ avec $M := \sup_V \frac{|f''|}{|f'|}$. Pour tout $R > r$ la fonction auxiliaire $v\colon \mathbb{R} \to \mathbb{R}$ définie par $v(t) = (R + M^{-1})\exp(Mt) - M^{-1}$ est croissante et vérifie $v(0) > |u(0)|$ et $v'(t) = 1 + Mv(t)$. Ceci entraîne que $|u(z)| < v(|z|)$ pour tout $z \in B(0, 2r)$. Sinon il existerait z tel que $|u(z)| = v(|z|)$ avec $|z|$ minimal, et on arriverait à la contradiction suivante :

$$\begin{aligned} |u(z)| - |u(0)| \leq |u(z) - u(0)| = \left|\int_0^z u'(\xi)\,d\xi\right| &\leq \int_0^z |u'(\xi)|\,d\xi \leq \int_0^z 1 + M|u(\xi)|\,d\xi \\ &< \int_0^{|z|} 1 + Mv(t)\,dt = \int_0^{|z|} v'(t)\,dt = v(|z|) - v(0) = |u(z)| - v(0) < |u(z)| - |u(0)|. \end{aligned}$$

On en déduit que $|\phi'(z)| \leq M|u(z)| < Mv(2r)$ pour tout $z \in B(0, 2r)$. Puisque c'est valable pour tout $R > r$ on obtient $|\phi'(z)| \leq Mv(2r)$ dans le cas limite $R = r$. Il ne reste qu'à déterminer M de sorte que $Mv(2r) = (Mr + 1)\exp(2Mr) - 1 \leq \frac{1}{2}$. Le choix $Mr = \frac{1}{8}$ convient, ce qui prouve le théorème. □

Exercice 2.24. Vérifier la preuve précédente, en particulier la majoration $|u(z)| < v(|z|)$. Montrer que l'équation $(x + 1)\exp(2x) = 3/2$ admet une unique solution x_0 et vérifier que $x_0 \approx 0.1381 > \frac{1}{8} = 0.125$.

Le théorème précédent est un premier pas vers un critère pratique, mais il nous demande encore d'étudier f dans une boule $B(z_0, 2r)$ afin de majorer la proportion $|f'(z)^{-1}f''(z)|$. Le théorème suivant, développé par S. Smale, remplace cette étude par des informations locales en z_0 seulement, à savoir les dérivées $f^{(k)}(z_0)$ pour $k \in \mathbb{N}$. Ces informations suffisent à contrôler le comportement de f si f est un polynôme, ou plus généralement une fonction analytique :

Théorème 2.25 (S. Smale, 1986)**.** *Soit $f\colon \mathbb{C} \to \mathbb{C}$ une fonction analytique. Pour que l'itération de la fonction $\phi(z) = z - f'(z)^{-1}f(z)$ converge pour une valeur initiale $z_0 \in \mathbb{C}$ il suffit que*

$$\sup_{k \geq 2} \sqrt[k-1]{\left|\frac{f^{(k)}(z_0)}{k!\,f'(z_0)}\right|} \leq \frac{1}{8}\left|\frac{f'(z_0)}{f(z_0)}\right|.$$

Dans ce cas f admet une unique racine a dans la boule $B(z_0, 2r)$ où $r := |f(z_0)/f'(z_0)|$ est le pas initial dans l'itération de Newton, et la suite $z_n = \phi^n(z_0)$ vérifie $|z_n - a| \leq (\frac{1}{2})^{2^n - 1} \cdot |z_0 - a|$. □

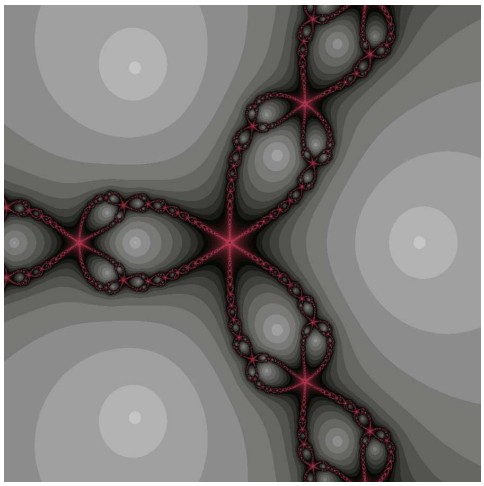
A noter que le théorème de Smale s'appuie uniquement sur des estimations au point z_0 donné, et que ce critère s'applique très facilement aux polynômes f : il suffit de vérifier la condition pour $k = 2, \ldots, n$.

Pour une discussion détaillée voir le chapitre 8 intitulé "Newton's method" du livre de L. Blum, F. Cucker, M. Shub, S. Smale : *Complexity and real computation*, Springer, New York 1997.

3. Application aux polynômes complexes

En 1685 Wallis publia son livre nommé *Algèbre* dans lequel il décrit une méthode inventée par Newton pour la résolution d'équations. Une version modifiée fut publiée par Raphson en 1690 ; c'est la méthode qui est maintenant connue sous le nom de Newton ou de Newton-Raphson. Newton lui-même avait discuté sa méthode en 1669 et l'avait illustrée par l'exemple $X^3 - 2X - 5 = 0$. Continuons cette tradition vielle de plus de 300 ans qui veut que tout étudiant de méthodes numériques résolve cette vénérable équation :

Exercice/P 3.1. Appliquez la méthode de Newton au polynôme $X^3 - 2X - 5$. Expérimentez avec le programme `newton1669.cc` pour trouver les trois racines de ce polynôme. Comme de nos jours ces calculs ne coûtent pas cher, vous pouvez facilement faire varier la valeur initiale z_0. Voyez-vous une corrélation entre le choix de z_0 et le résultat obtenu ? et le nombre d'itérations ? Tester plusieurs cas où z_0 est « proche » d'une racine, puis des cas où z_0 en est « loin ». Après plusieurs expériences réussies, regarder par exemple $z_0 = -60, -61, -62, \ldots, -70$ afin de tester délibérément les limites de la méthode.



Étant donné un polynôme $P \in \mathbb{C}[X]$ on peut appliquer l'itération de Newton $\phi(z) = z - P'(z)^{-1}P(z)$ à une valeur initiale $z_0 \in \mathbb{C}$ choisie au hasard, en espérant que $\phi^n(z_0)$ convergera vers une racine de P. De manière empirique, on constate que « la plupart » des choix initiaux mènent à une racine. Une fois on sait localiser une racine, la méthode de Newton permet de calculer de très bonnes approximations au bout de quelques itérations seulement. Sans connaissance préalable des racines, par contre, il est difficile de deviner de bonnes valeurs initiales. Pour une valeur initiale loin des racines il semble impossible de prédire vers quelle racine convergera l'itération. L'inconvénient pratique est surtout que la convergence peut être très lente. Dans le pire des cas, certains choix n'aboutiront jamais : pour $X^2 + 1$ une valeur initiale $z_0 \in \mathbb{R}$ ne permet pas de trouver une des deux racines. (Pourquoi ?)

On peut s'amuser avec des graphiques suivantes : Le polynôme $P(z) = z^3 - 1$ possède trois racines complexes : 1, j et j^2. On souhaite visualiser la dynamique de l'application $\phi(z) = z - P'(z)^{-1}P(z)$. Pour $a \in \{1, j, j^2\}$ on veut représenter graphiquement l'ensemble $A(a) = \{z_0 \in \mathbb{C} \mid \phi^n(z_0) \to a\}$, dit « bassin d'attraction ». On peut écrire un programme représentant les ensembles $A(a)$ par trois couleurs distinctes et utilisant une quatrième couleur pour l'ensemble des $z_0 \in \mathbb{C}$ tels que la suite (z_n) « ne semble pas » converger. Il existe des logiciels spécialisés pour ces graphiques dites « fractales », comme `fractint`.

Exemple 3.2. À titre d'avertissement, considérons le polynôme $p(z) = z^3 - 2z + 2$. La méthode de Newton itère $\phi(z) = z - \frac{z^3-2z+2}{3z^2-2} = \frac{2z^3-2}{3z^2-2}$, dont la dérivée est $\phi'(z) = \frac{6z^4-12z^2+12z}{9z^4-12z^2+4}$. Comme $\phi(0) = 1$ et $\phi(1) = 0$, on a un cycle de longueur 2. Ce cycle est super-attractif car $\phi'(0) = 0$: tous les points dans un voisinage de $\{0, 1\}$ sont attirés vers le cycle $0 \to 1 \to 0 \to 1 \to \ldots$, et ne convergerons pas vers une racine de p. Ce phénomène montre que l'on ne peut pas espérer la convergence pour « presque tout » point initial.

3.1. Le théorème de Gauss-d'Alembert. Soit K un corps et $P \in K[X]$ un polynôme de degré n. Pour $a \in K$ la division euclidienne donne $P(X) = (X - a)Q(X) + r$ avec $Q \in K[X]$ et $r \in K$. Ceci veut dire que a est une racine de P si et seulement si le facteur linéaire $(X - a)$ divise le polynôme P.

Soit $a_1 \in K$ une racine de P. Par récurrence on trouve $m_1 \geq 1$ tel que $P(X) = (X - a_1)^{m_1}Q(X)$ et $Q(a_1) \neq 0$. On appelle m_1 la *multiplicité* de la racine a_1. Si $a_1, \ldots, a_k$ sont des racines distinctes de P dans K alors $P = (X - a_1)^{m_1} \cdots (X - a_k)^{m_k}Q(X)$ avec des multiplicités $m_1, \ldots, m_k \geq 1$ et le polynôme restant $Q \in K[X]$ ne s'annule pas en $a_1, \ldots, a_k$. Puisque $\deg(P) = m_1 + \cdots + m_k + \deg(Q)$, on conclut que $m_1 + \cdots + m_k \leq n$, et ce procédé doit s'arrêter quand Q n'a plus aucune racine dans K :

Proposition 3.3. *Sur un corps K tout polynôme $P \in K[X]$ de degré n admet* au plus *n racines.* □

Si P admet n racines $a_1, a_2, \ldots, a_n \in K$ (éventuellement avec répétitions) alors il se factorise comme

$$P = c \cdot (X - a_1)(X - a_2) \cdots (X - a_n).$$

Une telle factorisation en polynômes de degré 1 n'est pas toujours possible : le polynôme $X^2 + 1 \in \mathbb{R}[X]$ par exemple n'admet aucune racine dans $\mathbb{R}$. Sur $\mathbb{C}$ la situation est beaucoup meilleure :

Théorème 3.4 (de Gauss-d'Alembert)**.** *Pour tout polynôme $P \in \mathbb{C}[X]$ unitaire de degré n il existe des nombres complexes $a_1, a_2, \ldots, a_n \in \mathbb{C}$ tels que $P = (X - a_1)(X - a_2) \cdots (X - a_n)$.*

On dit aussi que sur $\mathbb{C}$ tout polynôme est *scindé*, ou qu'il se *factorise* en facteurs linéaires sur $\mathbb{C}$. Cette propriété est tellement importante en algèbre qu'on lui donne un nom : on dit que le corps $\mathbb{C}$ est *algébriquement clos*. Ce superbe théorème est parfois appelé le « théorème fondamental de l'algèbre » (bien que le corps $\mathbb{C}$ appartienne plutôt à l'analyse). Remarquons toutefois qu'il s'agit d'un pur résultat d'existence. Il ne nous indique nullement comment *trouver* les racines. Reste le problème pratique évident : étant donné P, comment « calculer » ses racines ? Au moins deux approches se présentent :

L'approche algébrique. — En degré 2 on peut calculer les racines de $P = X^2 + aX + b$ par la formule $\frac{1}{2}(-a \pm \sqrt{a^2 - 4b})$. De formules similaires existent en degré 3 (formule de Cardan) et degré 4 (formule de Ferrari). Elles sont malheureusement assez complexes, et par conséquent rarement utilisées.

Jusqu'au XIXe siècle des généralisations aux degrés ≥ 5 furent cherchées en vain. C'est un des succès spectaculaires de la théorie de Galois d'en montrer l'impossibilité :

Théorème 3.5 (N.H. Abel, 1824)**.** *Il n'existe pas de formule construite à partir des opérations arithmétiques du corps $\mathbb{C}$ et des racines nièmes qui résolve une équation polynomiale générale de degré ≥ 5.* □

Sont résolubles par radicaux seulement certaines équations particulières, comme $X^n - a = 0$ pour un exemple évident. L'équation $X^5 - X + 1 = 0$ par contre n'est pas résoluble par radicaux. La théorie de Galois établit le critère général : une équation est résoluble par radicaux si et seulement si le groupe de Galois associé est résoluble.

L'approche numérique. — Étant donné un polynôme $P \in \mathbb{C}[X]$ il ne reste souvent que le recours aux méthodes numériques pour localiser les racines. Évidemment on ne peut en général pas espérer de tomber sur les racines exactes, mais de bonnes valeurs approchées donnent déjà des informations précieuses. Le projet mettra en œuvre une telle démarche.

3.2. Relation entre racines et coefficients. Intuitivement, les racines $a_1, \ldots, a_n \in \mathbb{C}$ d'un polynôme $p(z) = z^n + p_1 z^{n-1} + \cdots + p_{n-1} z + p_n$ varient continûment en fonction des coefficients $p_1, \ldots, p_n \in \mathbb{C}$. C'est vrai mais pas si facile à montrer. La proposition suivante donne une preuve dans le cas « générique » où toutes les racines sont simples ; la situation est plus lisse et le résultat est plus fort :

Proposition 3.6. *On considère l'application $\Phi\colon \mathbb{C}^n \to \mathbb{C}^n$ qui associe à n racines $(a_1, \ldots, a_n) \in \mathbb{C}^n$ les coefficients $(s_1, \ldots, s_n) \in \mathbb{C}^n$ du polynôme*

$$p(z) = \prod_{k=1}^{n} (z - a_k) = \sum_{k=0}^{n} (-1)^k s_k z^{n-k}.$$

On a $s_0 = 1$ et les fonctions $s_1, \ldots, s_n$ sont les fonctions symétriques élémentaires :

$$s_k = \sum_{i_1 < i_2 < \cdots < i_k} a_{i_1} a_{i_2} \cdots a_{i_k}.$$

L'application Φ est polynomiale, et $\Phi' = \left(\frac{\partial s_i}{\partial a_j}\right)$ a pour déterminant $\det \Phi' = \prod_{i<j} (a_i - a_j)$. Par conséquent la dérivée $\Phi'(a_1, \ldots, a_n)$ est inversible si et seulement si $a_1, \ldots, a_n$ sont distincts deux à deux. Le théorème d'inversion locale implique que dans un voisinage d'un polynôme séparable p (avec discriminant non nul) les racines dépendent de manière C^∞ (même analytiquement) des coefficients. □

Remarque 3.7. Le carré $(\det \Phi')^2 = \prod_{i<j} (a_i - a_j)^2$ est appelé le *discriminant* du polynôme p. C'est un polynôme symétrique dans les a_i et peut donc être exprimé comme un polynôme dans les coefficients s_j. Il existe des méthodes efficaces, similaires à l'algorithme d'Euclide, qui permettent de calculer le discriminant d'un polynôme donné.

Exercice/M 3.8. Si vous vous intéressez au calcul différentiel et/ou l'algèbre, vous êtes vivement invité à effectuer les calculs énoncés ci-dessus et de profiter des outils utilisés dans votre cours de calcul différentiel ou d'algèbre.

3.3. Instabilité des racines mal conditionnées. Même si les racines dépendent continûment des coefficients, on constate malheureusement un problème d'amplification d'erreur : des petites perturbations dans les coefficients peuvent entraîner des grandes perturbations dans les racines. Dans la pratique ceci se traduit à nouveau en un phénomène d'instabilité numérique.

Exemple 3.9. Les racines de $P(X) = X^n - 2^{-bn}$ sont faciles à expliciter : $a_k = 2^{-b}\exp(k \cdot 2\pi i/n)$ avec $k = 1, \dots, n$. Si l'on considère le polynôme voisin $Q(X) = X^n$ on trouve la racine 0 de multiplicité n. Un changement d'ordre 2^{-bn} dans les coefficients entraîne ici un changement d'ordre 2^{-b} dans les racines. (Pensez par exemple à $b = 50$ et $n = 10$.) Pour la pratique on retient la règle suivante :

☞ *Afin de calculer les racines d'un polynôme de degré n à une précision de b bits, il faut en général connaître les coefficients de p à une précision de nb bits.* ✍

Exercice 3.10. Dans des implémentations il est important de *vérifier* la qualité des racines approchées $\hat{a}_1, \dots, \hat{a}_n$ en comparant $\hat{P} = (X - \hat{a}_1) \cdots (X - \hat{a}_n)$ au polynôme P donné au départ. Nous sommes contents de notre factorisation si P et $\hat{P}$ sont proches. Est-ce un critère valable ? Discuter cette vérification.

Précisons nos observations de manière qualitative : Soit $P \in \mathbb{C}[X]$ un polynôme unitaire de degré n à racines distinctes $a_1, \dots, a_n$. En particulier $P'(a_k) \neq 0$ en toute racine a_k. On regarde le polynôme perturbé $P_t = P + tQ$, où $t \in \mathbb{R}$ et Q est un polynôme de degré $< n$.

Supposons que a est une racine de P, c'est-à-dire $P(a) = 0$, et que $a_t = a + \delta(t)$ est la racine correspondante du polynôme perturbé P_t, donc $0 = P_t(a_t) = P(a_t) + tQ(a_t)$. La dérivation par rapport à t donne

$$0 = \left[\frac{d}{dt}P_t(a_t)\right]_{t=0} = P'(a)\delta'(0) + Q(a) \qquad \text{donc} \qquad \delta'(0) = -\frac{Q(a)}{P'(a)}.$$

Pour $Q = X^k$ on obtient ainsi $\delta(0) = -a^k/P'(a)$; c'est le taux de changement de la racine a quand on perturbe le $k$ième coefficient du polynôme P.

- Pour une racine a avec $|a| > 1$ le maximum est atteint pour $k = n - 1$; ce sont donc les plus hauts coefficients qui influencent le plus les grandes racines.
- Pour une racine a avec $|a| < 1$ le maximum est atteint pour $k = 0$; ce sont donc les plus bas coefficients qui influencent le plus les petites racines.

☞ *La sensibilité d'une racine a aux perturbations des coefficients est amplifiée par le facteur $P'(a)^{-1}$. Plus la dérivée $P'(a)$ est proche de zéro, plus la racine a est instable.* ✍

Le pire des cas arrive si m racines $a_1, \dots, a_m$ sont très proches, car dans ce cas $P'(a_k) \approx 0$. Dans la limite on peut supposer qu'elles coïncident en une seule racine a de multiplicité m, et dans ce cas $P'(a) = 0$. L'instabilité d'une telle situation a déjà été illustrée dans l'exemple 3.9.

Exercice 3.11. Vérifiez ces constats de manière empirique sur des exemples de votre choix. Pour les calculs vous pouvez utiliser un logiciel de votre choix, ou bien votre propre implémentation issue du projet suivant. Est-ce que des petits changements dans les coefficients peuvent entraîner des grands changement des racines ? Qu'est-ce que cela veut dire pour notre vérification $\operatorname{dist}(\hat{P}, P) < \varepsilon$?

Exercice/M 3.12. Si le polynôme P risque d'avoir de racines multiples, on calcule d'abord $Q = \operatorname{pgcd}(P, P')$: les racines de Q sont exactement les racines multiples de P, et P/Q n'admet que de racines simples. Cette astuce peut considérablement faciliter le calcul des racines. Détailler cette approche algorithmiquement pour $P \in \mathbb{K}[X]$ avec $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{K} = \mathbb{Q}[i]$. Quels problèmes apparaissent pour $\mathbb{K} = \mathbb{R}$ et $\mathbb{K} = \mathbb{C}$?

Vous avez trouvé par de long ennuis
ce que Newton trouva sans sortir de chez lui.
Voltaire (1694-1778)

PROJET XVII

Factorisation de polynômes complexes

1. Approche probabiliste

Nous nous proposons de développer un programme pour factoriser des polynômes complexes. Soit $P \in \mathbb{C}[X]$ un polynôme unitaire de degré n, donné sous la forme $P(X) = X^n + p_1 X^{n-1} + \cdots + p_{n-1}X + p_n$ par ses coefficients $p_1, \ldots, p_n \in \mathbb{C}$. Nous cherchons les racines $a_1, a_2, \ldots, a_n \in \mathbb{C}$ telles que

$$P = (X - a_1)(X - a_2)\cdots(X - a_n)$$

Le théorème de Gauss-d'Alembert nous assure que ces nombres existent : sur $\mathbb{C}$ tout polynôme se factorise en facteurs linéaires. Il ne nous dit pas, par contre, comment les trouver.

Après nos expériences on peut espérer que pour « beaucoup » de valeurs initiales z_0, la suite de Newton $\phi^n(z_0)$ convergera vers une racine a. Ceci donne un procédé probabiliste pour trouver une valeur approchée d'une des racines de P. Comment les trouver toutes ? On pourrait essayer différentes valeurs initiales z_0 et espérer de tomber « par chance » sur les autres racines…

L'idée suivante se révèle plus efficace : on met la première racine trouvée a_1 en facteur, par une division euclidienne $P = (X - a_1)Q_1$, puis on recommence avec Q_1. De manière itérée on calculera ainsi toutes les racines de P. Vous trouverez les rudiments d'une telle implémentation dans le fichier `newton.cc`. Les exercices suivants vous demandent de compléter les fonctions qui manquent.

Remarque 1.1 (Propagation des erreurs d'arrondi)**.** Afin d'approcher les racines de P une par une, la fonction `factoriser` met en facteur les racines déjà trouvées, $P = (X - a_1)\cdots(X - a_k)Q_k$. L'itération de Newton produira (au moins probablement) une racine a_{k+1} de Q_k ; sans erreurs d'arrondi, ce serait aussi une racine de P. Malheureusement, nous ne disposons que de *racines approchées* $\hat{a}_1, \ldots, \hat{a}_k$, qui ne sont en général pas de racines exactes ! Déjà la première factorisation $P = (X - \hat{a}_1)\hat{Q}_1 + \hat{r}_1$ laissera donc un petit reste $\hat{r}_1 = P(\hat{a}_1) \neq 0$. Les factorisations suivantes accumulent forcément de telles erreurs.

Par conséquent il faut s'attendre à un polynôme erroné $\hat{Q}_k$, et une racine $\hat{a}_{k+1}$ de $\hat{Q}_k$ peut s'éloigner considérablement de la racine correspondante a_{k+1} de P. Néanmoins elle peut nous servir pour *localiser* grossièrement la vraie racine a_{k+1} de P. Pour augmenter la précision de $\hat{a}_{k+1}$ on réapplique donc l'itération de Newton par rapport à P (au lieu de $\hat{Q}_k$). Cette deuxième itération convergera vite vers a_{k+1} pourvu que $\hat{a}_{k+1}$ soit suffisamment proche. Cette approche en deux phases motive l'introduction des paramètres `localisation` et `affinage` : le premier spécifie le nombre d'itérations pour localiser une racine approchée de $\hat{Q}_k$, le deuxième spécifie le nombre d'itérations pour l'affinage par rapport à P.

2. Implémentation

Exercice/P 2.1. Lisez attentivement le code déjà implémenté dans `newton.cc` et survoler le fichier auxiliaire `polynome.cc`. Écrire une fonction `diff` qui implémente la dérivation $P \mapsto P'$.

Exercice/P 2.2. Pour mettre en facteur une racine approchée $\hat{a}$, implémenter une fonction `fact`, qui effectue la division euclidienne $P = (X - \hat{a})Q + \hat{r}$, puis renvoie Q.

Exercice/P 2.3. Afin de vérifier les racines approchées $\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_n$, on comparera P au produit $\hat{P} = (X - \hat{a}_1)(X - \hat{a}_2)\cdots(X - \hat{a}_n)$. Écrire une fonction `prod` qui implémente ce calcul.

3. Tests empiriques

Exercice 3.1. Pour tester votre programme, appliquez-le à quelques polynômes de votre choix, comme le polynôme $X^3 - 2X - 5$ regardé ci-dessus. S'il vous manque d'idées, vous pouvez tester $X^n - 1$ dont on connaît toutes les racines $e^{2i\pi k/n}$, ou bien $P_n = 1 + 2X + 3X^2 + \cdots + nX^{n-1} + X^n$, pour n de plus en plus grand. Quelles difficultés rencontrez-vous ? Peut-on tout de même parvenir à une factorisation en augmentant les paramètres `localisation` et `affinage` ?

Expérimenter avec ces paramètres pour mieux comprendre la problématique des arrondis :

(1) Sans affinage, jusqu'à quel degré les résultats sont-ils satisfaisants ?

(2) Pour P_{10}, P_{20}, P_{30}, disons, quels paramètres donnent de résultats satisfaisants ?

(3) Quand le degré croît, peut-on formuler une règle pour adapter les paramètres ?

(4) Arrive-t-on à factoriser P_{50} ? puis P_{100} ? Quel est l'obstacle ?

Exercice 3.2. Les types primitifs `double` et `long double` du C++ sont très efficaces quant au temps d'exécution, mais ils imposent aussi de sévères restrictions (rappeler leur précision). Pour plus de précision on calculera avec des nombres flottants en précision arbitraire, par exemple notre classe `RReal` faite maison. Pour ceci il suffit de remplacer la première ligne par

```
#include "rreal.cc"      // nombres flottants faits maison
using namespace Numeric; // accès à l'espace de noms Numeric
typedef RReal Flo;       // synonyme utilisé dans la suite
```

Pour rassurer le compilateur les constantes littérales comme `2.0` sont à écrire comme `Flo(2.0)`, et de la même manière d'éventuelles conversions implicites ambiguës sont à rendre explicites. Avec cette précaution, votre programme devrait compiler avec la classe `RReal`. N'oublier pas l'option `-lgmpxx` : on utilise la bibliothèque GMP. Mettez au point et testez cette nouvelle version de votre programme.

Exercice 3.3. Expliquer pourquoi on doit calculer les premières racines avec une très grande précision si le degré n est grand. Supposons que l'on veut calculer toutes les racines à 20 décimales, soit 60 bits environs, ou bien une précision à demander à l'utilisateur. Comment adapter la précision durant les calculs intermédiaires afin d'assurer la précision souhaitée dans les résultats finaux ? Explicitez une règle pratique, implémentez-là, et testez-là sur des polynômes de plus en plus grands. Arrive-t-on à la précision souhaitée ? Comment la vérifier d'une manière convainquante ? Que dire du temps d'exécution ?

4. La danse des racines

On peut joindre deux polynômes unitaires P_0 et P_1 de degré n par le chemin $P_t = (1-t)P_0 + tP_1$ paramétré par $t \in [0,1]$. Si l'on connaît une racine a_0 de P_0 on peut la suivre : puisque P_t change continûment, il existe un chemin $[0,1] \to \mathbb{C}$, $t \mapsto a_t$ tel que $P_t(a_t) = 0$ pour tout $t \in [0,1]$. Nous allons tacitement supposer que tout polynôme intermédiaire P_t est séparable : dans ce cas la proposition 3.6 nous assure que la fonction $t \mapsto a_t$ est analytique en t.

Dans la pratique on pourra commencer par le polynôme $P_0 = X^n - 1$ dont on connaît toutes les racines $a_0^{(k)} = e^{2i\pi k/n}$. On subdivise l'intervalle $[0,1]$ par $0 = t_0 < t_1 < \cdots < t_N = 1$. Les racines initiales $a_{t_0}^{(k)}$ sont connues. Ayant calculé les racines $a_{t_j}^{(k)}$ du polynôme P_{t_j}, on peut appliquer la méthode de Newton pour le polynôme $P_{t_{j+1}}$: avec une subdivision suffisamment fine, la valeur initiale $a_{t_j}^{(k)}$ convergera rapidement vers la racine $a_{t_{j+1}}^{(k)}$ de $P_{t_{j+1}}$. Après N étapes on arrive finalement au polynôme P_1 et toutes ses racines $a_1^{(k)}$.

Exercice/P 4.1. Si cette approche par « homotopie » vous intéresse, vous pouvez l'implémenter en étendant votre programme. Expérimenter avec la subdivision $t_j = \frac{j}{N}$ et la précision avec laquelle vous calculer. On peut combiner cette approche avec le critère du théorème 2.25 pour assurer la convergence. Ceci permet d'adapter la subdivision localement comme nécessaire. (La seule difficulté est d'assurer $\mathrm{disc}(P_t)^2 > 0$.)

CHAPITRE XVIII

Systèmes linéaires et matrices

Systèmes linéaires. Considérons un *système d'équations linéaires* :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= y_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= y_2 \\ & \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= y_m \end{cases}$$

En algèbre linéaire on développe la théorie de ces systèmes sur un corps ou un anneau $\mathbb{K}$, et on apprend certaines méthodes pour leur résolution, notamment la méthode de Gauss rappelée plus bas. Ici les données a_{ij} et y_i sont des coefficients dans $\mathbb{K}$ et les x_j sont les inconnues, à déterminer dans la suite. Pour $\mathbb{K} = \mathbb{Z}$ ou $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{K}$ un corps fini, on peut effectuer ces calculs de manière exacte sur ordinateur. Lorsque $\mathbb{K} = \mathbb{R}$ ou $\mathbb{K} = \mathbb{C}$, par contre, on fait recours au calcul numérique arrondi : les données initiales, les calculs intermédiaires et les résultats finaux ne sont que des valeurs approchées.

Structuration du problème. Comme vous en avez l'habitude, la matrice $A = (a_{ij})_{j=1,\dots,n}^{i=1,\dots,m}$ et les vecteurs $x = (x_j)_{j=1,\dots,n}$ et $y = (y_i)_{i=1,\dots,m}$ permettent d'écrire ce système plus succinctement comme

$$Ax = y.$$

C'est bien plus qu'une notation commode. Cette structuration des données est le point de départ de tous les algorithmes pour la résolution de systèmes linéaires. Ayant fixé les données A et y, il s'agit de trouver les solutions x vérifiant $Ax = y$. Or, les problèmes liés à la résolution des systèmes linéaires sont divers. Il y a d'abord la question de la représentations des données et du choix des algorithmes adaptés (petites matrices denses, grandes matrices creuses, matrices trigonales, en blocs, en bandes, etc.). Il y a d'une part des questions habituelles de la complexité algorithmique. Il y a d'autre part, lors du calcul *numérique*, les problèmes liés au conditionnement du système et à la propagation d'erreurs.

Résolution d'un système linéaire. Si A est une matrice inversible, de taille $n \times n$, le système $Ax = y$ est équivalent à $x = A^{-1}y$. C'est cette méthode qui est souvent utilisée dans les petits exemples, disons $n = 3$ ou $n = 4$. On développera cette démarche au §1 pour les matrices de petite taille, disons $n \leq 10$, par la méthode de Faddeev. Soulignons deux avertissements :

- ☞ Le calcul de la matrice inverse A^{-1} est équivalent à la résolution de n systèmes linéaires $Av_i = e_i$. Ceci peut être assez coûteux, à savoir d'ordre $O(n^3)$ opérations si A est dense, c'est-à-dire, ne comporte que peu de coefficients nuls. Ce n'est pas toujours la meilleure solution.
- ☞ La formule de Cramer qui nécessite le calcul de $n+1$ déterminants, soit $(n+1)!$ opérations, est, quant à elle totalement inexploitable. (Calculer 10! ou 20! voire 50! pour vous en convaincre.) Aussi importante qu'elle soit pour la théorie, ne songez pas à l'utiliser sur ordinateur.

Ce chapitre rappelle et implémente quelques méthodes élémentaires, notamment l'élimination de Gauss, permettant de résoudre des systèmes de taille moyenne, disons $n \leq 100$. Nous n'indiquerons qu'en passant des problèmes possibles et des variantes qui remédient aux inconvénients les plus fréquents.

Sommaire

1. Implémentation de matrices en C++

1.1. Matrices denses vs creuses. Commençons par les éléments de base de tout traitement algorithmique : la représentation des données et les opérations élémentaires. Cette première étape est de grande importance, car la modélisation dépend fortement du champs d'application envisagé :

- Veut-on modéliser des matrices *denses*, c'est-à-dire comportant peu de coefficients nuls ? S'agit-il des matrices génériques ? ou symétriques ? Ont-elles des propriétés particulières ? ...
- Veut-on modéliser des matrices *creuses*, c'est-à-dire comportant beaucoup de zéros ? S'agit-il des matrices concentrées autour de la diagonale ? Ou des matrices triangulaires ? Ou en blocs ? ...

Question 1.1 (à titre d'exemple)**.** Une matrice dense de taille $m \times n$ nécessite le stockage de mn coefficients, ce qui peut facilement déborder la mémoire disponible. Discuter si l'on peut stocker puis additionner et multiplier des matrices denses de taille 100×100, 1000×1000, ou $10^4 \times 10^4$, ou $10^5 \times 10^5$, etc.

Jusqu'où est-ce possible pour des matrices creuses ? Précisez quel genre de matrices creuses vous considérez puis esquissez comment les stocker et comment les additionner et multiplier.

1.2. Une implémentation faite maison. Dans ce chapitre on considérera des matrices denses, alors que le projet annexe sur Google traitera un cas de matrices creuses.

Exercice/P 1.2. Le fichier `matrix.cc` implémente une classe générique `Matrix<T>` pour stocker des matrices denses de taille $m \times n$ dont les coefficients sont d'un type `T` et indexés par (i, j) avec $i \in \{1, \dots, m\}$ et $j \in \{1, \dots, n\}$. Essayez de comprendre le code déjà implémenté. Ajouter les opérations usuelles :

```
Matrix<T> operator + ( const Matrix<T>& a, const Matrix<T>& b )
Matrix<T> operator - ( const Matrix<T>& a, const Matrix<T>& b )
Matrix<T> operator * ( const Matrix<T>& a, const Matrix<T>& b )
```

Conseil. — Quand vous utilisez des constantes, il vaut mieux écrire `T(0)` et `T(1)` au lieu de `0` et `1` de type `int` : la conversion explicite assurera le bon résultat, quelque soit le type `T` utilisé. Sans cette précaution, vous obligez le compilateur à deviner lui-même la conversion, ce qui n'est pas toujours possible. Même si c'est possible, la conversion implicite choisie n'est pas forcément celle que vous souhaitez.

Gestion d'exceptions. — Quand les dimensions de `a` et `b` correspondent, l'implémentation ne pose pas de problème. Sinon, il faut décider comment gérer cette exception. On peut renvoyer la matrice vide, de dimension 0×0, pour signaler l'erreur ou bien afficher un message d'erreur et abandonner le calcul. On pourrait aussi « tronquer » convenablement les matrices, ou bien les « stabiliser » c'est-à-dire les élargir convenablement en ajoutant des coefficients zéros.

Remarque 1.3 (complexité)**.** Pour estimer le coût des calculs ultérieurs il est important de connaître d'abord la complexité des opérations élémentaires. On considère les matrices $n \times n$ et on compte le nombre d'opérations effectuées sur les coefficients.

(1) Pour l'addition $C = A + B$ on parcourt toutes les paires (i, j) par deux boucles imbriquées pour $i = 1, \dots, n$ et $j = 1, \dots, n$. Pour chaque (i, j) on calcule $c_{ij} \leftarrow a_{ij} + b_{ij}$. Ceci fait n^2 additions.

(2) Pour la multiplication scalaire $C = \lambda A$ on utilise deux boucles imbriquées et calcule $c_{ij} \leftarrow \lambda a_{ij}$ pour chaque (i, j). Ceci nécessite n^2 multiplications, et on ne peut pas espérer de faire mieux.

(3) La multiplication $C = A * B$, par contre, est plus complexe. Le calcul directe de $c_{ij} \leftarrow \sum_{k=1}^{n} a_{ik} b_{kj}$ nécessite *trois* boucles imbriquées, pour i et j et k. Cet algorithme de multiplication nécessite donc n^3 multiplication et $n^2(n-1)$ additions, au total donc presque $2n^3$ opérations.

1.3. Multiplication d'après Strassen. La définition du produit $C = A * B$ par la formule $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ donne immédiatement lieu à un algorithme de multiplication de matrices. Si cet algorithme est satisfaisant pour les matrices de petite taille, le coût cubique se fait sentir pour les grandes matrices. En 1969 V. Strassen découvrit une multiplication plus rapide.

Puisque les multiplications sont plus coûteuse que les additions, on essaiera d'économiser les multiplications — même au prix de quelques additions supplémentaires. Concrètement, pour calculer $C = A * B$ on suppose A, B, C de taille $n \times n$ avec $n = 2m$. On les décompose en blocs de taille $m \times m$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

On veut implémenter les formules qui définissent la multiplication des matrices :

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22},$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Comme telles, ces formules utilisent 8 multiplications et 4 additions. On peut faire avec 7 multiplications et 18 additions. D'abord on calcule les sept produits auxiliaires suivants :

$$P_1 := (A_{11} + A_{22})(B_{11} + B_{22}), \qquad P_2 := (A_{21} + A_{22})B_{11},$$
$$P_3 := A_{11}(B_{12} - B_{22}), \qquad P_4 := A_{22}(B_{21} - B_{11}),$$
$$P_5 := (A_{11} + A_{12})B_{22}, \qquad P_6 := (A_{21} - A_{11})(B_{11} + B_{12}),$$
$$P_7 := (A_{12} - A_{22})(B_{21} + B_{22}).$$

Le calcul de $P_1, \ldots, P_7$ utilise 7 multiplications et 10 additions / soustractions. On peut en déduire les coefficients de C avec les 8 additions / soustractions suivantes :

$$C_{11} = P_1 + P_4 - P_5 + P_7, \qquad C_{12} = P_3 + P_5,$$
$$C_{21} = P_2 + P_4, \qquad C_{22} = P_1 - P_2 + P_3 + P_6.$$

Quel est donc l'intérêt de remplacer 8 multiplications et 4 additions par 7 multiplications et 18 additions ? N'est-ce pas une perte d'efficacité ? C'est sans doute vrai pour les matrices 2×2, mais pour les matrices plus grandes il faut regarder de plus prêt :

	Formules de la définition :		Formules de Strassen :	
	multiplications	additions	multiplications	additions
taille n	n^3	$n^2(n+1)$	$7(\frac{n}{2})^3$	$7(\frac{n}{2})^2(\frac{n}{2}-1)+18(\frac{n}{2})^2$
2	8	4	7	18
4	64	48	56	100
6	216	180	189	288
8	512	448	448	624
10	1000	900	875	1150
12	1728	1584	1512	1908
14	2744	2548	2401	2940
16	4096	3840	3584	4288
18	5832	5508	5103	5994
20	8000	7600	7000	8100
22	10648	10164	9317	10648
24	13824	13248	12096	13680
26	17576	16900	15379	17238
28	21952	21168	19208	21364
30	27000	26100	23625	26100

On voit que la méthode de Strassen commence à s'amortir à partir d'une certaine taille n_0 entre 16 et 30. Le point exact dépend du coût respectif de la multiplication et de l'addition des coefficients. Mais quelque soit cette pondération, au plus tard pour $n = 30$ la méthode de Strassen devient plus efficace.

Application récursive. — Pour les grandes matrices on peut appliquer la méthode de Strassen de manière récursive. Pour les additions / soustractions on utilise l'algorithme évident qui ne laisse rien à désirer. Pour les 7 multiplications, par contre, on applique à nouveau la méthode de Strassen aux matrices de tailles $\frac{n}{2} \times \frac{n}{2}$. Ainsi si $c(n)$ dénote le côut de la multiplication de deux matrices de taille $n \times n$, on obtient la formule de récurrence $c(n) = 7c(\lceil \frac{n}{2} \rceil) + 18\alpha(\frac{n}{2})^2$ où α est le coût d'une addition de coefficients. On en déduit que $c(n)$ est d'ordre $O(n^\sigma)$ avec un exposant $\sigma = \log_2(7) \approx 2.808$.

Exercice 1.4. Le fichier `strassen.cc` implémente la multiplication de matrices d'après Strassen comme expliquée ci-dessus. Aux additions et multiplications s'ajoute le coût pour *copier* les sous-matrices. Pour la complexité asymptotique c'est négligeable, mais dans une implémentation conrète ce problème décale le point d'amortissement. Vous pouvez empiriquement déterminer les champs d'applications des deux méthodes : dans notre implémentation non-optimisée Strassen commence à s'amortir à partir de $n \approx 100$. Le gain est de 10% pour $n \approx 200$, de 20% pour $n \approx 300$, et de 40% pour $n \approx 500$.

Remarque 1.5. Malheureusement, l'algorithme de Strassen est numériquement instable : à cause des additions / soustractions supplémentaires il introduit typiquement plus d'erreurs d'arrondi que l'algorithme classique. Pour cette raison il n'est en général pas utilisé pour le calcul numérique. Lors d'un calcul exact, par contre, il est nettement plus efficace pour les grandes matrices que l'algorithme cubique.

Remarque 1.6. En 1987 D. Coppersmith et S. Winograd ont publié un algorithme de complexité $O(n^{2.376})$, mais jusqu'ici c'est resté un résultat purement théorique. Rien n'indique que l'exposant est optimal, et on pourrait soupçonner que l'on puisse arriver à $O(n^{2+\varepsilon})$ pour tout $\varepsilon > 0$. Il est claire, par contre, que l'exposant ne peut être inférieur à 2 puisque l'algorithme doit au moins écrire les n^2 coefficients du résultat.

1.4. Inversion d'après Faddeev. À partir des opérations élémentaires, que nous venons de discuter, on peut déjà calculer l'inverse d'une matrice par une méthode simple mais astucieuse découverte par L. Faddeev. On la présente ici parce qu'elle s'implémente très facilement — et que sa preuve est amusante.

Soit A une matrice $n \times n$. Son *polynôme caractéristique* est $P(X) := \det(A - XI) \in \mathbb{K}[t]$. La méthode de Faddeev permet de calculer P et, lorsque A est inversible, l'inverse de A.

Proposition 1.7. *Soit $A \in \mathrm{Mat}(n \times n; \mathbb{K})$. On pose $p_0 = 1$ et $B_0 = I$, puis pour $k = 1, \dots, n$ on définit*

$$C_k := A \cdot B_{k-1} = A^k - p_1 A^{k-1} - \cdots - p_{k-2} A^2 - p_{k-1} A$$

ainsi que $p_k = \frac{1}{k} \operatorname{tr} C_k$ et $B_k = C_k - p_k I$. L'algorithme s'arrête avec $B_n = 0$, et le polynôme caractéristique de A est $P = (-1)^n (X^n - \sum_{i=1}^n p_i X^{n-i})$. De plus, si $p_n \neq 0$, l'inverse de A n'est autre que $B = \frac{1}{p_n} B_{n-1}$.

On se propose d'implémenter cette méthode afin de la *tester* empiriquement. De manière complémentaire vous pouvez la *prouver* afin de justifier la correction du programme.

Exercice/M 1.8. Développer une preuve de la proposition par les étapes suivantes :

(1) Vérifier la proposition pour une matrice diagonale $A = \mathrm{diag}(\lambda_1, \lambda_2)$, puis $A = \mathrm{diag}(\lambda_1, \lambda_2, \lambda_3)$. Si vous voulez, vous pouvez tenter une preuve pour $A = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$ quelconque.

(2) Supposons que l'énoncé est vrai pour les matrices diagonales. Reste-t-il vrai pour toute matrice triangulaire A ? Puis pour TAT^{-1}, la matrice conjuguée par $T \in \mathrm{GL}_n \mathbb{K}$? Conclure.

Exercice/P 1.9. Implémenter la méthode de Faddeev en une fonction

```
void faddeev( const Matrix<T>& a, Matrix<T>& b, vector<T>& poly )
```

et l'appliquer aux matrices définies dans les fichiers d'exemples. Vérifier la correction de vos calculs en multipliant A par l'inverse calculé. Quel est le nombre d'opérations aritmétiques nécessaire pour calculer l'inverse d'une matrice $n \times n$ par la méthode de Faddeev ?

Jusqu'ici, au moins au niveau théorique, tout va bien. Pour le calcul numérique il existe pourtant une difficulté supplémentaire : les erreurs d'arrondi ! Voici un exemple classique et assez frappant :

Exercice/P 1.10 (matrices de Vandermonde)**.** Pour $n \in \mathbb{N}$ on définit la matrice V_n de taille $n \times n$ comme ayant i^{j-1} pour coefficient (i, j). Écrire une fonction `Matrix<T> vandermonde( Dim n )` qui construit la matrice V_n, puis calculer l'inverse de $V_6, V_7, V_8, \dots$ (avec vérification bien sûr). Qu'observer vous avec le type `double` ? Pour quels rangs n le résultat est-il acceptable ? Pour quels n est-il grossièrement faux ? En quoi est-ce surprenant ? Comparer avec des calculs exacts utilisant les types `Rationnel` et `RReal`.

Ce test exhibe une difficulté typique : pour n grand, les coefficients de V_n varient violemment, et la matrice se révèle *mal conditionnée*, c'est-à-dire difficile à résoudre par des méthodes numériques (§2.2).

Exercice/P 1.11 (matrices de Hilbert)**.** Voici un deuxième exemple de matrices mal conditionnées. Pour $n \in \mathbb{N}$ on définit la matrice de Hilbert H_n de taille $n \times n$ par les coefficients $\frac{1}{i+j-1}$ pour $i, j = 1, \dots, n$. Écrire une fonction `Matrix<T> hilbert( Dim n )` qui construit la matrice H_n, puis calculer l'inverse de H_n pour n de plus en plus grand. (Ne pas oublier la vérification du résultat.) Pour quels rangs n le résultat est-il acceptable ? Pour quels n est-il grossièrement faux ? En quoi est-ce surprenant ?

A priori ces difficultés numériques pourraient être des artefacts de la méthode de Faddeev. On essaiera plus bas d'invertir ces matrices par la méthode de Gauss, pour comparer laquelle des deux méthodes gère mieux les erreurs d'arrondis. Puis on discutera au §2.2 les propriétés mathématiques qui font que l'inversion d'une matrice peut être numériquement méchante.

2. La méthode de Gauss

Motivation. Comme expliqué au début du chapitre on veut résoudre un système $Ax = y$. Si A est triangulaire supérieure, la solution est immédiate : il suffit de remonter. Plus explicitement, on résout $a_{nn}x_n = y_n$, puis on remonte pour résoudre $a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = y_{n-1}$, et ainsi de suite :

$$Ax = y \qquad \text{avec} \qquad A = \begin{pmatrix} a_{11} & \dots & \dots & a_{1n} \\ 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_{nn} \end{pmatrix}.$$

Dans ce cas on suppose que tous les coefficients diagonaux a_{kk} sont non nuls, et donc inversibles dans le corps $\mathbb{K}$. Plus généralement la solution est facile si A est échelonnée : ici la matrice A n'est plus supposée inversible, ni même carrée. Rappelons qu'une matrice est dite *échelonnée* si le nombre de zéros précédant le premier coefficients non nul d'une ligne augmente ligne par ligne. C'est le cas dans l'exemple ci-dessous, avec des *pivots* a_{11}, a_{23}, a_{34}, a_{47} non nuls et des coefficients $*$ quelconques :

$$Ax = y \qquad \text{avec} \qquad A = \begin{pmatrix} a_{11} & * & * & * & * & * & * \\ 0 & 0 & a_{23} & * & * & * & * \\ 0 & 0 & 0 & a_{34} & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{47} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

À noter que l'application $f_A \colon \mathbb{K}^n \to \mathbb{K}^m$, $x \mapsto Ax$ n'est en général ni surjective ni injective. Dans l'exemple précédent l'équation $Ax = y$ n'a pas de solution si $y_5 \neq 0$. Si $y_5 = 0$, par contre, les solutions x telles que $Ax = y$ forment un sous-espace affine de $\mathbb{K}^n$ de dimension 3. (Le détailler.)

2.1. L'algorithme de Gauss. La méthode de Gauss pour un système $Ax = y$ consiste à déterminer une matrice inversible M telle que la matrice $U = MA$ soit trigonale supérieure, ou bien échelonnée si A n'est pas nénessairement inversible. Ensuite le système $Ax = y$ est équivalent à $Ux = My$, ce qui se résout par la méthode des remontées.

Plus explicitement, on effectue des opérations sur les lignes afin d'obtenir une matrice échelonnée. Trois opérations sont à notre disposition :

$A_i \leftrightarrow A_j$ **:** échanger la ligne i et la ligne j,

$A_i \leftarrow aA_i$ **:** multiplier la ligne i par un facteur inversible a,

$A_i \leftarrow A_i + aA_j$ **:** ajouter un multiple de la ligne j à la ligne i.

L'invariant. On pose $U_0 = A$ et $M_0 = I$, la matrice identité de taille $m \times m$ afin d'assurer la condition initiale $U_0 = M_0 A$. Chacune des trois opérations de base correspond à multiplier à gauche par une certaine matrice inversible T_k. (Expliciter T_k et T_k^{-1}.) Dans une implémentation on ne créera pas T_k explicitement, mais on l'applique à U_{k-1} pour obtenir $U_k := T_k U_{k-1}$, et simultanément à M_{k-1} afin d'obtenir $M_k := T_k M_{k-1}$. Ainsi M_k est à nouveau inversible et vérifie toujours $U_k = M_k A$. Si finalement on arrive à une matrice $U = U_k$ échelonnée, alors on a trouvé $M = M_k$ inversible de sorte que $U = MA$ comme souhaité.

L'algorithme. Pour la première colonne, on choisit un élément $a_{i,1} \neq 0$, que l'on appelle alors le *pivot*, puis on échange la ligne i et la première ligne : c'est l'opération $L_1 \leftrightarrow L_i$. Maintenant $a_{1,1} \neq 0$ et on peut diviser la première ligne par $a_{1,1}$, c'est-à-dire $L_1 \leftarrow \frac{1}{a_{1,1}} L_1$, afin d'obtenir $a_{1,1} = 1$. Finalement, pour tout $i = 2, \dots, n$, on effectue $L_i \leftarrow L_i - a_{i,1} L_1$ de façon à obtenir, dans la première colonne, des termes nuls. On itère l'algorithme en l'appliquant à la matrice $(n-1) \times (n-1)$ obtenue en ignorant la première ligne et première colonne de A. Si jamais une colonne est entièrement zéro, on procède à la suivante.

Exercice 2.1. Vérifier que dans la méthode de Gauss le nombre d'opérations arithmétiques est d'ordre $O(n^3)$ pour une matrice $n \times n$. Si vous voulez, vous pouvez préciser le coût exact en comptant les différentes opérations sur les coefficients : copie, addition, soustraction, multiplication, division. Une fois les matrices U et M calculées, quel est le coût de résoudre $Ax = y$? À noter qu'il est souvent utile de garder U et M afin de résoudre $Ax = y$ pour plusieurs y.

Exercice 2.2. Cette méthode, appliquée à une matrice carrée, permet aussi de calculer efficacement le déterminant. Vérifier que les trois opérations élémentaires ci-dessus changent le déterminant respectivement par un facteur de -1, de a ou pas du tout. Le déterminant de la matrice finale U est le produit de ses éléments diagonaux, ce qui permet de calculer $\det(A)$. Quelle est la complexité de cette méthode ? Comparer avec le calcul via les permutations : $\det(A) = \sum_{\sigma\in\mathfrak{S}_n} \operatorname{sign}(\sigma)\cdot a_{1,\sigma(1)}\cdot a_{2,\sigma(2)}\cdots a_{n,\sigma(n)}$. Cette formule *polynomiale* est très importante pour la théorie des déterminants, mais elle s'avère catastrophique pour des calculs concrets ! Expliciter pourquoi.

L'algorithme de Gauss-Jordan. Si A est inversible on ne peut tomber, au cours de l'algorithme de Gauss, sur une colonne entièrement zéro. Supposons de plus que dans l'algorithme on élimine les coefficients en dessous *et* au dessus du pivot. Dans ce cas la matrice finale sera $U = I$, donc $M = A^{-1}$. Ceci donne une méthode pratique pour inverser des matrices.

Exercice 2.3. Détailler cette variante de l'algorithme. Montrer ainsi qu'il est possible d'inverser une matrice $n\times n$ avec $O(n^3)$ opérations arithmétiques seulement. Comparer à la méthode de Faddeev (théoriquement puis empiriquement après implémentation, voir plus bas).

Préparation numérique. Afin de minimiser les erreurs d'arrondi, on peut préparer les coefficients de A en divisant la ligne L_i par $\sup_j |a_{i,j}|$. On assure ainsi que $\sup_j |a_{i,j}| = 1$ dans le souci de minimiser les variations dans les coefficients et les pertes de précision qui peuvent en résulter. Il faut aussi remarquer que le choix des pivots *n'est pas anodin* : on choisira en général le pivot de module maximum.

Exercice 2.4. On considère le système $\varepsilon x_1 + x_2 = 1$ et $x_1 + x_2 = 2$ avec $\varepsilon = 10^{-9}$. Résoudre ce système en prenant ε pour pivot, puis en prenant 1 comme pivot. Dans ces calculs on ne travaillera qu'avec 8 chiffres significatifs (type `float`). Comparer puis discuter les résultats.

Implémentation. Après ces préparations, passons à l'implémentation de l'algorithme de Gauss ou, si vous préférez, de la variante de Gauss-Jordan.

Exercice/P 2.5. Écrire une fonction mettant en œuvre la méthode de Gauss :

```
template <typename T>
T gauss( Matrix<T>& a, Matrix<T>& m )
```

Ici `a` est la matrice initiale que sera transformée au cours de l'algorithme, et `m` est la matrice accompagnatrice de sorte que `m*a` reste constant. La valeur renvoyée est le déterminant de la matrice initiale.

Attention. — L'implémentation proposée ici opère directement sur les deux matrices `a` et `m` passées par référence. (Justifier le choix de ce mode de passage.) Aucune matrice auxiliaire, notée T_k plus haut, n'est construite explicitement : ce serait très coûteux et inutilement compliqué. (Expliquer pourquoi.)

Vérification. — Pour une application numérique, veillez en particulier à préparer la matrice `a` comme indiqué ci-dessus, et à choisir à chaque étape un pivot de module maximal. Tester votre fonction sur quelques-uns des exemples précédents, en utilisant le type `double` puis le type `Rationnel`.

Exercice/P 2.6. Écrire une fonction qui permet d'inverser une matrice par la méthode de Gauss. Tester les matrices de Vandermonde V_n et de Hilbert H_n vues plus haut. (Ne pas oublier la vérification du résultat.) Pour quels rangs n le résultat est-il acceptable ? Pour quels n est-il grossièrement faux ? Que dire du temps du calcul ? (Vous pouvez utiliser le fichier `timer.hh` pour chronométrer.)

Exercice/P 2.7. Parfois on ne veut que calculer le déterminant $\det(A)$ sans pour autant calculer M de sorte que MA soit échelonnée. Pour ce cas particulier on pourrait implémenter une fonction optimisée : `template <typename T> T det( const Matrix<T>& a )`. Discuter la complexité de ce calcul. À votre avis, est-ce que le gain de performance justifie une implémentation séparée ?

Exercice/P 2.8 (optionnel)**.** Écrire un programme qui permet de lire une matrice A et un vecteur y d'un fichier, puis résout le système $Ax = y$. Améliorez votre implémentation afin de déterminer noyau et image d'une matrice, par exemple

$$A = \begin{pmatrix} 1 & -8 & -9 & -18 \\ 2 & -11 & -12 & -29 \\ 1 & -8 & -9 & -10 \\ 0 & 5 & 6 & -1 \end{pmatrix}.$$

2.2. Conditionnement. Pour toute méthode numérique il est important de comprendre la propagation d'erreurs afin d'éviter un usage inapproprié. Ainsi tout algorithme a ses limites inhérentes, parfois dues à la méthode, parfois dictées par les données elles-mêmes.

Exemple 2.9. On considère l'équation $Ax = y$ avec

$$A = \begin{pmatrix} 0{,}780 & 0{,}563 \\ 0{,}913 & 0{,}659 \end{pmatrix} \quad \text{et} \quad y = \begin{pmatrix} 0{,}217 \\ 0{,}254 \end{pmatrix}.$$

Lequel des deux résultats approchés suivants est meilleur,

$$\tilde{x} = \begin{pmatrix} +0{,}999 \\ -1{,}001 \end{pmatrix} \quad \text{ou} \quad \hat{x} = \begin{pmatrix} +0{,}341 \\ -0{,}087 \end{pmatrix} ?$$

Comme la solution exacte est souvent inconnue, on pourrait simplement calculer les erreurs $|A\tilde{x} - y|$ et $|A\hat{x} - y|$ et choisir la solution qui minimise cette erreur. Vérifier qu'ici c'est $\hat{x}$. En sachant que la solution exacte est $x = \binom{+1}{-1}$, c'est pourtant $\tilde{x}$ qui est nettement plus proche.

Comment expliquer puis quantifier ce phénomène étrange ? Supposons, comme dans l'exemple, que l'on travaille sur $E = \mathbb{R}^n$ ou $E = \mathbb{C}^n$. On munit cet espace d'une norme $E \to \mathbb{R}_+$, $x \mapsto |x|$, habituellement la norme euclidienne $|x| = \sqrt{\sum_{k=1}^n |x_k|^2}$. On regarde une matrice A de taille $n \times n$ comme application linéaire $E \to E$, et on définit sa norme par $|A| := \sup\{|Ax|; |x| \leq 1\}$.

Remarque 2.10. La norme $|A|$ mesure l'effet de « distorsion » : on a $|Ax| \leq |A| \cdot |x|$ pour tout $x \in E$, et $|A|$ est la plus petite valeur possible pour cette inégalité. Si A est diagonalisable, la norme $|A|$ est simplement la plus grande valeur propre en valeur absolue. (Exercice !)

Dans la suite on considère une matrice A *inversible*, et on s'intéresse à la stabilité numérique du système $Ax = y$. Comment varie la solution x si l'on perturbe le vecteur y ? Soit $y' = y + \delta y$ et x' solution de $Ax' = y'$; on a donc $x' = x + \delta x$ avec $A\delta x = \delta y$. Que dire de l'erreur relative $\frac{|\delta x|}{|x|}$ par rapport à l'erreur relative $\frac{|\delta y|}{|y|}$? D'une part on a $|\delta y| \leq |A| \cdot |\delta x|$ et $|\delta x| \leq |A^{-1}| \cdot |\delta y|$, donc

$$\frac{1}{|A|} \frac{|\delta y|}{|x|} \leq \frac{|\delta x|}{|x|} \leq |A^{-1}| \frac{|\delta y|}{|x|}.$$

D'autre part on a $|y| \leq |A| \cdot |x|$ et $|x| \leq |A^{-1}| \cdot |y|$, donc

$$\frac{1}{\mathrm{cond}(A)} \frac{|\delta y|}{|y|} \leq \frac{|\delta x|}{|x|} \leq \mathrm{cond}(A) \frac{|\delta y|}{|y|}.$$

Ici on a introduit $\mathrm{cond}(A) := |A| \cdot |A^{-1}|$, appelé le *conditionnement* de la matrice A.

Remarque 2.11. Chacune des inégalités précédentes devient une égalité pour un choix convenable de y et δy. Ainsi le conditionnement décrit comment une perturbation de y s'amplifie en une perturbation de x.

- On a toujours $\mathrm{cond}(A) = |A| \cdot |A^{-1}| \geq |AA^{-1}| = |I| = 1$.
- Si $\mathrm{cond}(A)$ est proche de 1, alors une petite perturbation de y entraîne une petite perturbation de x.
- Si $\mathrm{cond}(A)$ est grand, une petite perturbation de y peut entraîner une grande perturbation de x.

Ainsi un mauvais conditionnement de la matrice A implique en général une perte de précision :

☞ *Typiquement* $\mathrm{cond}(A) \approx 10^c$ *veut dire que la donnée de y avec une précision de* ℓ *décimales mène à une solution x avec une précision de* $\ell - c$ *décimales.* ✍

Exemple 2.12. On considère $A = \binom{7\ 10}{5\ 7}$ avec $A^{-1} = \binom{-7\ 10}{5\ -7}$. Les valeurs propres de A sont $\lambda_1 = 7 - 5\sqrt{2} \approx -0{,}071068$ et $\lambda_2 = 7 + 5\sqrt{2} \approx 14{,}071$, donc $|A| \approx 14{,}071$. Les valeurs propres de A^{-1} sont $-7 \pm 5\sqrt{2}$, donc $|A^{-1}| \approx 14{,}071$ et $\mathrm{cond}(A) \approx 198$. Ceci indique que $Ax = y$ peut être sensible aux perturbations de y. Effectivement $Ax = \begin{pmatrix} 1{,}00 \\ 0{,}70 \end{pmatrix}$ donne $x = \begin{pmatrix} 0{,}00 \\ 0{,}10 \end{pmatrix}$, alors que $Ax' = \begin{pmatrix} 1{,}01 \\ 0{,}69 \end{pmatrix}$ donne $x' = \begin{pmatrix} -0{,}17 \\ 0{,}22 \end{pmatrix}$. On constate que le changement relatif en x est plus grand que le changement relatif en y.

Exercice 2.13. En utilisant un logiciel de calcul formel, calculer le conditionnement des matrices de Vandermonde V_n et de Hilbert H_n de taille $n \times n$. Expliquer ainsi les difficultés numériques rencontrées lors de l'inversion de ces matrices. Est-ce que les mêmes problèmes se font sentir lors d'un calcul exact utilisant le type `Rationnel` ? En revanche, quels problèmes peuvent se présenter dans le calcul exact ?

2.3. Factorisation *LU*. La méthode de Gauss admet de nombreuses variantes et spécialisations à des situations particulières. Nous n'en mentionnons que deux : la factorisation LU puis la méthode de Cholesky.

La factorisation dite *LU* est un cas particulier de la méthode de Gauss où l'on choisit toujours $a_{i,i}$ comme pivot, sans jamais échanger de lignes. En général ce coefficient peut s'annuler, il faut donc une hypothèse supplémentaire :

Proposition 2.14. *Si $A = (a_{i,j})$ est une matrice $n \times n$ telle que les n sous-matrices diagonales $\Delta_k = \begin{pmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & & \vdots \\ a_{k,1} & \cdots & a_{k,k} \end{pmatrix}$ soient inversibles, alors il existe une matrice triangulaire inférieure L et une matrice triangulaire supérieure U telles que $A = LU$. Si l'on impose en outre que tous les éléments diagonaux de L soient égaux à 1, alors il y a unicité.*

Exercice/M 2.15. Montrer la proposition en suivant la méthode de Gauss : étant donnée l'hypothèse, on peut toujours choisir $a_{i,i}$ comme pivot. Vérifier que l'on construit ainsi L et U comme souhaité. Pour l'unicité remarquer que le produit de deux matrices triangulaires inférieures (resp. supérieures) est à nouveau triangulaire inférieure (resp. supérieure).

Remarque 2.16. L'hypothèse de la proposition est vérifiée pour les matrices *à diagonale dominante*, c'est-à-dire vérifiant $|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$. De telles matrices sont inversibles. Les hypothèses sont encore vérifiées pour les matrices *symétriques définies positives*, c'est-à-dire $A^t = A$ et $v^t A v > 0$ pour tout $v \neq 0$.

Remarque 2.17. L'intérêt de la factorisation *LU* réside dans le fait suivant : le système $LUx = y$ est équivalent à $Lw = y$ et $Ux = w$; on est donc ramené à résoudre deux systèmes triangulaires.

Exercice/P 2.18. Écrire un programme qui réalise la décomposition *LU* d'une matrice carrée A et qui utilise cette décomposition pour résoudre des systèmes linéaires $Ax = y$.

2.4. Méthode de Cholesky. La méthode de Cholesky est un cas particulier de la factorisation *LU* appliquée aux matrices symétriques définies positives :

Proposition 2.19. *Si A est une matrice symétrique définie positive, alors il existe une matrice triangulaire inférieure B telle que $A = BB^t$. Cette matrice est unique si l'on impose la condition $b_{i,i} > 0$ pour tout i.*

Exemple 2.20. Regardons $A = \left(\begin{smallmatrix} 4 & -2 \\ -2 & 10 \end{smallmatrix}\right)$. S'il existe $B = \begin{pmatrix} b_{11} & 0 \\ b_{21} & b_{22} \end{pmatrix}$ de sorte que $A = BB^t$, alors on peut déterminer ses coefficients un par un. D'abord $b_{11}^2 = 4$, on pose donc $b_{11} = 2$. Ensuite $b_{11} b_{21} = -2$, donc $b_{21} = -1$. Finalement $b_{21}^2 + b_{22}^2 = 10$, donc $b_{22} = 3$. On vérifie aisément que $BB^t = A$, comme souhaité.

Exercice/M 2.21. Montrer que la méthode esquissée se généralise à toute matrice symétrique définie positive de taille $n \times n$, ce qui démontre la proposition. Vérifier aussi que toute matrice $A = BB^t$, avec B inversible, est symétrique définie positive. La construction précédente de B, dans le cas de réussite, constitue donc une preuve que A est définie positive.

Exercice/P 2.22. Écrire un programme qui vérifie si une matrice A est symétrique définie positive et qui, dans l'affirmative, donne une matrice triangulaire inférieure B à coefficients diagonaux positifs telle que $A = BB^t$. À cet effet on cherchera B par la méthode des coefficients indéterminés.

Exercice/P 2.23. Adapter la méthode de Cholesky pour décomposer, quand cela est possible, une matrice symétrique sous la forme BDB^t où D est diagonale et B est triangulaire inférieure avec $b_{i,i} = 1$. Comme application analyser la matrice suivante. Est-elle définie positive ?

$$A = \begin{pmatrix} 20 & 5 & -1 & 20 \\ 5 & 1 & 0 & 5 \\ -1 & 0 & -2 & -1 \\ 20 & 5 & -1 & 20 \end{pmatrix}.$$

Exercice/P 2.24. Calculer l'inverse de la matrice $X = \left(\begin{smallmatrix} A & B \\ B & A \end{smallmatrix}\right)$ où $A = \begin{pmatrix} 1 & 0 & -2 & -2 \\ 0 & 1 & -2 & -2 \\ -2 & -2 & 1 & 0 \\ -2 & -2 & 0 & 1 \end{pmatrix}$ et $B = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$.

Expliquer la méthode suivie.

PROJET XVIII

Classement de pages web à la Google

Objectifs

- Comprendre le fonctionnement de Google, un outil de recherche omniprésent.
- Résoudre un système linéaire creux par une méthode itérative bien adaptée.

Ce projet implémente la technique utilisée par Google, un moteur de recherche généraliste qui a vu un succès fulgurant depuis sa naissance en 1998. Le point fort de Google est qu'il trie *par ordre d'importance* les résultats d'une requête, c'est-à-dire les pages web associées aux mots-clés donnés. On s'intéresse ici de plus près à l'algorithme de classement, qui est à la fois simple et ingénieux. Il s'agit essentiellement de résoudre un immense système d'équations linéaires.

[1] Vous trouvez un développement détaillé dans l'article *Comment fonctionne Google ?* dont ce projet ne donne qu'un résumé. Cet article discute les motivations et la modélisation générale, alors que ce projet se concentrera sur l'implémentation.

[2] Si vous préférez aller aux sources, et avoir une présentation plus informatique, vous pouvez consulter l'article fondateur : S. Brin et L. Page, *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, Stanford University 1998. (Pour le trouver, utiliser Google. ;-)

[3] Si vous voulez savoir plus sur la foudroyante histoire de l'entreprise Google, ses légendes et anecdotes, vous lirez avec profit le récent livre de David Vise et Mark Malseed, *Google story*, Dunod, Paris 2006.

Sommaire

1. Marche aléatoire sur la toile

1.1. Que fait un moteur de recherche ? À première vue, le principe d'un moteur de recherche est simple : on copie les pages web concernées en mémoire locale, puis on trie le contenu (les mots-clés) par ordre alphabétique afin d'effectuer des recherches lexiques. Une *requête* est la donnée d'un ou plusieurs mots-clés ; la *réponse* est une liste des pages contenant les mots-clés recherchés. C'est en gros ce que faisaient les moteurs de recherche, dits de première génération, dans les années 1990.

L'énorme quantité des données entraîne de sérieux problèmes car le nombre des documents à gérer est énorme et rien que le stockage et la gestion efficaces posent des défis considérables. Plus délicat encore : les pages trouvées sont souvent trop nombreuses, il faut donc en choisir les plus pertinentes. La grande innovation apportée par Google en 1998 est le tri des pages par ordre d'importance. Ce qui est frappant est que cet ordre correspond assez précisément aux attentes des utilisateurs.

Exemple 1.1. Par exemple, si vous vous intéressez à la programmation et vous faites chercher les mots-clés « C++ compiler », vous trouverez quelques millions de pages. Des pages importantes comme `gcc.gnu.org` se trouvent quelque part en tête du classement, ce qui est très raisonnable. Par contre, une petite page personnelle, où l'auteur mentionne qu'il ne connaît rien du C++ et n'arrive pas à compiler, ne figurera que vers la fin de la liste, ce qui est également raisonnable. Comment Google distingue-t-il les deux ?

Selon les informations fournies par l'entreprise elle-même (voir `www.google.com/corporate`), l'index de Google porte sur plus de 8 milliards d'adresses web (en avril 2007). Une bonne partie des informations répertoriées, pages web et documents annexes, changent fréquemment. Il est donc hors de question de les classer manuellement, par des êtres humains : ce serait trop coûteux, trop lent et jamais à jour. L'importance d'une page doit donc être déterminée de manière automatisée, par un algorithme. Comment est-ce possible ?

On ne va pas chercher à définir exactement ce qui est l'importance d'une page web. (Peut-il y en avoir une définition objective précise ?) Notre approche sera plus modeste : le mieux que l'on puisse espérer est que notre modèle dégage un résultat qui *approche* bien l'importance *ressentie* par les utilisateurs. L'idée est plutôt de considérer la popularité des pages web, c'est-à-dire leur fréquentation moyenne.

1.2. Matrice de transition. Dans la suite nous modélisons un surfeur aléatoire qui ne lit jamais rien mais qui clique au hasard. Ainsi ce n'est pas le contenu des pages web qui soit pris en compte, mais uniquement la structure du graphe formé par les pages et les liens entre elles. On renvoie à l'article [1] pour des arguments en faveur de ce modèle.

On considère des pages numérotées par $1,\dots,n$. Chaque page j émet un certain nombre ℓ_j de liens. On peut supposer $\ell_j \geq 1$; si jamais une page n'émet pas de liens on peut la faire pointer vers elle-même. Pour tout couple d'indices $i, j \in [\![1,n]\!]$ on définit un coefficient a_{ij} par

$$a_{ij} := \begin{cases} \frac{1}{\ell_j} & \text{si la page } j \text{ émet un lien vers la page } i, \\ 0 & \text{sinon.} \end{cases}$$

On interprète a_{ij} comme la probabilité d'aller de la page j à la page i, en suivant un des ℓ_j liens au hasard. La *marche aléatoire* associée consiste à se balader sur le graphe suivant les probabilités a_{ij}.

Selon sa définition, notre matrice $A = (a_{ij})$ vérifie

$$\begin{aligned} a_{ij} &\geq 0 \qquad \text{pour tout } i, j \text{ et} \\ \sum_i a_{ij} &= 1 \qquad \text{pour tout } j, \end{aligned}$$

ce que l'on appelle une *matrice stochastique*. À noter que la somme de chaque colonne vaut 1, mais on ne peut en général rien dire sur la somme dans une ligne.

1.3. Mesures invariantes. Supposons qu'un vecteur $x \in \mathbb{R}^n$ vérifie

$$x_j \geq 0 \quad \text{pour tout } j \text{ et} \quad \sum_j x_j = 1,$$

ce que l'on appelle un *vecteur stochastique* ou une *mesure de probabilité* sur les pages $1,\dots,n$: on interprète x_j comme la probabilité de se trouver sur la page j.

Effectuons un pas dans la marche aléatoire : avec probabilité x_j on démarre sur la page j, puis on suit le lien $j \to i$ avec probabilité a_{ij}. Ce chemin nous fait tomber sur la page i avec une probabilité $a_{ij}x_j$. Au total, la probabilité d'arriver sur la page i, par n'importe quel chemin, est la somme

$$y_i = \sum_j a_{ij} x_j.$$

Autrement dit, un pas dans la marche aléatoire correspond à l'application linéaire

$$T : \mathbb{R}^n \to \mathbb{R}^n, \quad x \mapsto y = Ax.$$

Exercice/M 1.2. Soit A une matrice stochastique. Majorer la norme de $y = Ax$ en fonction de la norme de x. Que peut-on en déduire pour les valeurs propres de A, ou plus précisément pour le rayon spectral de A ? Si x est un vecteur stochastique ($x_i \geq 0$, $|x| = 1$), vérifier que l'image $y = Ax$ est à nouveau stochastique.

Définition 1.3. Une mesure de probabilité μ vérifiant $\mu = T(\mu)$ est appelée une *mesure invariante* ou une *mesure d'équilibre*. En termes d'algèbre linéaire c'est un vecteur propre associé à la valeur propre 1. En termes d'analyse, μ est un point fixe de l'application T.

1.4. Le modèle utilisé par Google. Pour des raisons expliquées dans [1], Google utilise un modèle plus raffiné, dépendant d'un paramètre $c \in [0,1]$:

– Avec probabilité c, le surfeur abandonne la page actuelle et recommence sur une des n pages du web, choisie de manière équiprobable.
– Avec probabilité $1-c$, le surfeur suit un des liens de la page actuelle j, choisi de manière équiprobable parmi tous les ℓ_j liens émis. (C'est la marche aléatoire discutée ci-dessus.)

Ce modèle se formalise comme l'application

$$T : \mathbb{R}^n \to \mathbb{R}^n, \qquad T(\mu) = c\varepsilon + (1-c)A\mu$$

où A est la matrice stochastique définie en §1.2, et le vecteur stochastique $\varepsilon = (\frac{1}{n}, \dots, \frac{1}{n})$ correspond à l'équiprobabilité sur toutes les pages. Pour $c = 0$ on obtient donc exactement le modèle initial.

Proposition 1.4. *Soit A une matrice stochastique et $T : \mu \mapsto c\varepsilon + (1-c)A\mu$ avec une constante $c \in]0,1]$. Alors l'application T admet une unique mesure invariante $\mu = T(\mu)$. De plus, pour toute mesure initiale μ^0 la suite itérée $\mu^{n+1} = T(\mu^n)$ converge vers l'unique point fixe $\mu = T(\mu)$.*

C'est cette mesure invariante μ qui nous intéressera dans la suite et que l'on interprétera comme mesure d'importance. On la calculera d'ailleurs par la méthode itérative de la proposition.

Exercice/M 1.5. Prouver cette proposition en montrant que T est contractante pour la norme $|\mu| = \sum_i |\mu_i|$. (L'intérêt n'est pas de recopier la preuve de quelqu'un d'autre ; essayez plutôt de refaire la démonstration vous-mêmes et d'ainsi vérifier votre compréhension des arguments.) Quelle est la constante de contraction ? Majorer l'erreur $|\mu - \mu^n|$ en fonction de μ^n et μ^{n-1}. Que peut-on dire de la vitesse de convergence ?

Exercice/M 1.6. Est-ce une bonne idée de prendre $c = 1$ pour calculer le classement des pages web ? D'un autre coté, montrer par un exemple que la proposition est fausse pour $c = 0$.

Un bon choix de c se situe donc quelque part entre 0 et 1. En termes probabilistes, $\frac{1}{c}$ est le *nombre moyen* de liens suivis avant de recommencer sur une page aléatoire. En général on choisira la constante c positive mais proche de zéro. Par exemple, $c = 0,15$ correspond à suivre 7 liens en moyenne.

2. Implémentation en C++

Passons à l'implémentation de l'algorithme discuté ci-dessus. Le logiciel qui en résulte sera plutôt court (environ 40 lignes pour le calcul de μ plus quelques fonctions auxiliaires d'entrée-sortie, voir le fichier `graphe.cc`). Néanmoins il est important de préméditer la façon comment nous nous y prendrons.

2.1. Matrices creuses provenant de graphes. Rappelons qu'en réalité la matrice A est très grande : en 2004 Google affirmait que « le classement est effectué grâce à la résolution d'une équation de 500 millions de variables et de plus de 3 milliards de termes. » Comment est-ce possible ?

Certes, il est envisageable de stocker une matrice 1000×1000 sous le format usuel, c'est-à-dire dans un grand tableau de 10^6 coefficients indexés par $(i,j) \in [\![1,1000]\!]^2$. Ceci est hors de question pour une matrice $n \times n$ avec $n \approx 10^6$, voire $n \approx 10^8$.

Heureusement dans notre cas la plupart des coefficients vaut zéro, car une page n'émet que quelques dizaines de liens typiquement. Dans ce cas il suffit de stocker les coefficients non nuls, dont le nombre est d'ordre n et non n^2. Une telle matrice est appelée *creuse* (ou *sparse* en anglais).

☞ *Pour des applications réalistes il est donc nécessaire d'implémenter des structures et des méthodes spécialisées aux matrices creuses.* ✍

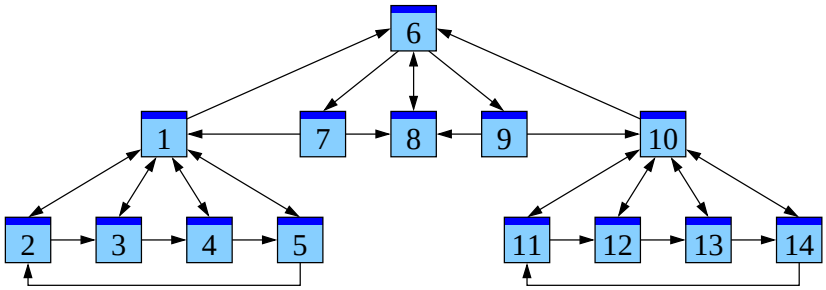
Exercice/P 2.1. Pour simplifier, nous allons spécialiser notre implémentation aux matrices creuses provenant de graphes. On peut implémenter la structure de graphe comme suit :

```
typedef unsigned int   Page;   // les pages forment les sommets du graphe
typedef vector<Page>   Liste;  // les liens forment les arrêtes (orientées)
typedef vector<Liste>  Graphe; // les pages et les liens forment le graphe
typedef vector<double> Mesure; // mesure de probabilité sur les sommets
```

Pour l'entrée-sortie il faut fixer une notation convenable. Dans le graphe ci-dessous la page `1` émet des liens vers les pages `2,3,4,5,6`. Ceci est noté simplement par `1:2,3,4,5,6;` comme dans le fichier `graphe1.mat`. Le graphe tout entier s'écrit comme

```
Graphe( 1:2,3,4,5,6; 2:1,3; 3:1,4; 4:1,5; 5:1,2; 6:7,8,9; 7:8,1; 8:6;
9:8,10; 10:6,11,12,13,14; 11:10,12; 12:10,13; 13:10,14; 14:10,11; )
```

L'entrée sous ce format-ci est déjà implémentée dans le fichier `graphe.cc`. Essayez de comprendre son fonctionnement, puis ajouter l'opérateur de sortie.



2.2. La méthode itérative.

Exercice/P 2.2. Implémenter une fonction `mesure_invariante` qui calcule la mesure invariante μ d'un graphe G. Comme motivé plus haut, on utilise un paramètre c, qui prend la valeur $0,15$ par défaut.

```
typedef vector<double> Mesure;
void mesure_invariante( const Graph& g, Mesure& m, const double c=0.15 );
```

Essayer de n'utiliser que les deux objets `g` et `m` ainsi qu'une copie de `m` durant sa mise à jour ; ni la matrice A ni le vecteur ε ne figureront explicitement dans l'implémentation. Nous rappelons que la construction explicite de la matrice dense T sera catastrophique pour toute application réaliste.

Exercice/P 2.3 (tests en taille minuscule)**.** Testez votre implémentation sur les deux exemples donnés dans les fichiers `graphe1.mat` et `graphe2.mat`. Avant les calculs, quels résultats prédirez-vous. Quels résultats obtient-on pour $c = 0$ et pour $c = 1$? Pour $c = 0,05,\ldots,0.95$? Ces résultats sont-ils plausibles ?

Exercice/P 2.4 (tests en taille moyenne)**.** Pour des exemples un peu plus grands et donc un peu plus réalistes, regardez les fichiers `graphe100.mat` et `graphe1000.mat`, qui donnent deux graphes à 100 et 1000 sommets respectivement. Peut-on deviner sans calcul quelles pages se dégageront comme les plus populaires ? Est-il envisageable de résoudre l'équation $T\mu = \mu$ par la méthode de Gauss ? En utilisant votre implémentation, trouvez les cinq pages les plus fréquentées (= populaires = importantes ?) ainsi que leur mesure calculée. Est-ce que le calcul s'effectue dans un temps raisonnable ?

Exercice/P 2.5 (extrapolation à une échelle réaliste)**.** Votre implémentation marchera-t-elle pour des graphes encore plus grands ? Quel temps d'exécution faudra-t-il environ et de quels paramètres dépend-il ? (Vous pouvez extrapoler ou, pour être plus précis, créer de graphes aléatoires via la fonction `random` implémentée dans `graphe.cc` puis mesurer le temps d'exécution.) Est-ce une méthode praticable à l'échelle « grandeur nature » de l'internet ?

Exercice/P 2.6 (expérience de manipulation)**.** Créez une copie du fichier `graphe1000.mat` nommée `graphe1000bis.mat`. Essayez de manipuler ce graphe pour que votre page préférée (disons la page 1) arrive en tête du classement. Dans une situation réaliste, vous ne pouvez évidemment pas changer les pages des autres, mais vous pouvez adapter les vôtres. La technique d'ajouter des pages et des liens à des fins stratégiques s'appelle « link farming » en anglais. Comment le faire de manière bien camouflée ? Comment l'entreprise Google peut-elle réagir à ces tentatives de manipulations ? De manière générale, quelles autres stratégies voyez-vous pour améliorer le classement de votre page préférée ?