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 $0 \leq k \leq n$, 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* $0 = f(0) \leq f(1) \leq f(2) \leq \dots$ *avec* $\sup f = +\infty$. *Alors pour tout* $y \in \mathbb{N}$ *il existe un unique* $x \in \mathbb{N}$ *de sorte que* $f(x) \leq y < f(x+1)$. □

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 $f(x) \leq y < f(x+1)$ 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 $f(x) \leq y < f(x+1)$.

```
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 $x \geq 1$ 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 $f(x) \leq y < f(x+1)$.

$r \leftarrow 0,\ \ s \leftarrow 1$	// On commence par $r = 0$ donc $f(r) \leq y$.
tant que $f(s) \leq y$ **faire** $r \leftarrow s,\ \ s \leftarrow 2s$	// On assure ainsi que $f(r) \leq y < f(s)$.
tant que $s - r > 1$ **faire**	// Tant que l'intervalle n'est pas réduit à un point …
$m \leftarrow \left\lfloor \frac{r+s}{2} \right\rfloor$	// … on divise l'intervalle au milieu et …
si $f(m) \leq y$ **alors** $r \leftarrow m$ **sinon** $s \leftarrow m$	// … on assure à nouveau que $f(r) \leq y < f(s)$.
fin tant que	// finalement $s = r+1$ et $f(r) \leq y < f(s)$
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.)