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** $a_k \leftarrow a_k + \mathit{retenue}$, $\mathit{retenue} \leftarrow a_k \operatorname{div} b$, $a_k \leftarrow a_k \bmod b$
tant que $\mathit{retenue} > 0$ **faire** $n \leftarrow n+1$, $a_n \leftarrow \mathit{retenue} \bmod b$, $\mathit{retenue} \leftarrow \mathit{retenue} \operatorname{div} b$
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 $(a_n,\ldots,a_1,a_0)$ 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 $0,\ldots,9$. 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 $a_n, \dots, a_1, a_0$, 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* I, II, III, IIII, IIIII, … 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 :

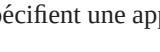
(1) Pour tout n dans $\mathbb{N}$ on a $\mathbf{S}n \neq 0$, autrement dit 0 n'appartient pas à l'image de $\mathbf{S}$.

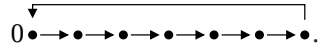
(2) Pour tout $n \neq m$ dans $\mathbb{N}$ on a $\mathbf{S}n \neq \mathbf{S}m$, autrement dit $\mathbf{S}$ est injectif.

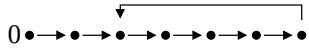
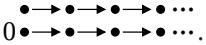
(3) Si $E \subset \mathbb{N}$ est un sous-ensemble contenant 0 et vérifiant $\mathbf{S}(E) \subset E$, alors $E = \mathbb{N}$.

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 **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;
}
```