Cours

Algorithmique pour l'agrégation interne

Mathieu Mansuy - Préparation à l'Agrégation Interne de Mathématiques, Université de Franche Comté
`mansuy.mathieu@hotmail.fr`

1 Algorithmique et informatique

Dans cette section, on passe en revue les notions de base d'algorithmique au programme de l'Agrégation Interne.

1.1 Types

En informatique, on classe les données par *type* en les regroupant en fonction de :

- la syntaxe des manipulations possibles les concernant,
- leur représentation en mémoire.

Présentons ici les types de données les plus communément rencontrées, à commencer par les types de nombres :

- le type *entier* muni des opérations (ou fonctions) élémentaires +, *, / (quotient de la division euclidienne), ... Pour les implémenter en mémoire, on peut s'appuyer sur la représentation binaire d'un entier[1] sur 32 bits (c'est à dire comme élément de $\{0,1\}^{32}$) où
  - le premier bit correspond au signe : 1 pour les négatifs, 0 pour les positifs ;
  - les 31 autres bits correspondent à l'écriture binaire de la valeur absolue.

  Remarquons tout de suite que les entiers sont donc bornés en machine par 2^{30} (pour le codage sur 32 bits).
- le type *flottant*, qui correspond aux réels en machine, muni des opérations élémentaires (pointées afin de les distinguer des opérations sur les entiers) +., *., /., $\sqrt{}$, ... On peut ici utiliser la représentation binaire sur 64 bits (norme universelle IEEE double précision instaurée en 1985) où
  - le premier bit correspond au signe : 1 pour les négatifs, 0 pour les positifs ;
  - les 11 bits suivants servent pour l'écriture binaire de l'exposant de la puissance de 2 par laquelle il faut multiplier pour avoir le réel (auquel on soustrait un biais de 1025) ;
  - les 52 derniers bits correspondent à l'écriture binaire de la (troncature de la) mantisse.

  Notons là aussi qu'il n'existe qu'un nombre fini de réels flottants et qu'ils sont strictement compris en valeur absolue entre $2^{-1022} \approx 2,23 \times 10^{-308}$ et $1,1\ldots1_2 \times 2^{1023} \approx 1,80 \times 10^{308}$. Ainsi pour représenter un réel x non flottant, on utilisera :
  - si $2^{-1022} < |x| < 1,1\ldots1_2 \times 2^{1023}$, le flottant le plus proche de x (par exemple) ;
  - si $x = 0$, un signe quelconque, un exposant (biaisé) nul et une mantisse nulle ;
  - si $0 < |x| < 2^{-1022}$, on dit que x est dans l'*underflow* et on le représentera en machine par un signe quelconque, un exposant biaisé nul et une mantisse non nulle quelconque ;
  - si $|x| > 1,1\ldots1_2 \times 2^{1023}$, on dit que x est dans l'*overflow* et on le représentera en machine par un signe quelconque, un exposant biaisé de 1023 et une mantisse nulle.

[1] Je vous renvoie à l'annexe pour des rappels sur la représentation d'un réel en base b.

Exemple.

Le flottant représenté en machine par :

$$\underbrace{0}_{\text{signe}}\ \underbrace{10000010010}_{\text{exposant biaisé}}\ \underbrace{0110101100010\ldots0}_{\text{mantisse}}$$

correspond au réel

$$x = (-1)^s \times 1,011010110001_2 \times 2^{E-1025}$$

avec $s = 0$ et $E = 10000010010_2 = 2^{10} + 2^4 + 2^1 = 1024 + 16 + 2 = 1042$.

- le type *booléen* comporte deux valeurs `vrai` et `faux`. Il est muni des opérations logiques `et`, `ou` et `non` pour la négation.

Citons d'autres types construits à partir de structures de données linéaires (c'est à dire représentables par des suites finies ordonnées de variables de même type) :

- le type *tableau* dont les principales caractéristiques sont :
  - la longueur est fixée initialement : une fois un tableau créé, la taille de ce dernier ne peut plus être modifiée. On parle de structure de donnée *statique* ;
  - on accède en temps constant à une case ;
  - on peut modifier le contenu de chaque case. On parle de structure de donnée *mutable*.
- le type (*chaîne de*) *caractères* dont les éléments sont intuitivement des tableaux de caractères. Il est muni des opérations d'accès à une coordonnée d'une chaine, de concaténation, ...
- le type *liste* (*chainée*) dont les principales caractéristiques sont :
  - l'accès au k-ième élément de la liste nécessite de parcourir les $k-1$ précédents : le coût de l'accès à une case requiert donc k opérations élémentaires ;
  - cette structure de donnée est *dynamique* : une fois la liste créée, il est toujours possible d'insérer une case supplémentaire.

Citons aussi :

- le type *matrice* ou *tableau multidimensionnel* ;
- les *structures arborescentes* ;
- ...

Ces types de données peuvent ne pas être implémentés selon le langage de programmation utilisé, ce qui posera donc le choix d'un tel langage lors de la programmation en machine. Par exemple, le type entier n'existe pas sur le logiciel `Scilab` (on peut représenter un entier par un flottant sans erreur de représentation s'il est inférieur en valeur absolue à 2^{53}), et il ne sera donc pas très adapté pour tout ce qui touche à l'arithmétique.

1.2 Algorithmes

Pour stocker une donnée en mémoire, on utilisera une *variable*. Une variable possède un nom, qui est composé d'au moins un caractère alphabétique. Elle peut contenir n'importe quel type de données.

Pour donner une valeur à une variable, on écrit un nom de variable suivi d'une instruction d'affectation symbolisée par une flèche et suivie par la valeur. Par exemple, l'instruction

```
a <- 2
```

affecte la valeur entière 2 à la variable `a`. On peut aussi utiliser la commande `saisir(a)` pour que l'utilisateur entre la valeur pour la variable `a`, et `afficher(a)` pour renvoyer la valeur contenue dans la variable `a`. Dans l'esprit des programmes actuels, on préfèrera cependant la création d'une fonction plutôt que d'un programme interactif utilisant ces commandes.

Des instructions plus complexes sont bien évidement possibles, pouvant faire appel à plusieurs noms de variables, des opérateurs, des appels à des fonctions, ou tester l'égalité (à l'aide du symbole =) ou l'ordre entre deux expressions. On peut utiliser la valeur de ce test pour exécuter une instruction conditionnelle comme l'instruction `si` :

```
Si condition Alors bloc_vrai Sinon bloc_faux Fin Si
```

On peut exécuter des instructions plusieurs fois de suite en utilisant une *boucle* définie (nombre d'exécutions fixé au début)

```
Pour ... de ... à ... Faire ... Fin Pour
```

ou indéfinie (le nombre d'exécutions n'est pas connu)

```
Tant que ... Faire ... Fin Tant que
```

On peut alors concevoir un algorithme.

Définition.

Un algorithme est une suite finie d'instructions non ambigües correspondant à la résolution d'un problème dont les entrées et les sorties sont précisées.

Pour réaliser des taches un peu plus complexes, il devient intéressant de les subdiviser en plusieurs entités indépendantes appelées fonctions, qui permettent d'étendre les possibilités des instructions du logiciel. Une fonction peut avoir 0, 1 ou plusieurs arguments (comme la fonction racine carrée `sqrt()` qui en a un). Elle calcule une valeur, appelée valeur de retour. Cette valeur de retour peut être utilisée dans une autre fonction ou expression algébrique, exactement comme le résultat de la fonction racine carrée.

La définition de fonctions peut parfois se faire directement avec une formule (fonction définie algébriquement) mais nécessite parfois un algorithme plus complexe. Elle peut calculer des résultats intermédiaires et les stocker dans des variables. Pour éviter toute interférence avec les variables (dîtes *globales*) définies en-dehors de la fonction, on utilise un type spécial de variables, les *variables locales* : leur valeur ne peut être modifiée ou accédée qu'à l'intérieur de la fonction.

Exemple.

Définissons une fonction `factorielle` prenant en entrée un entier n et renvoyant $n!$. On rédigera cette fonction, et toutes les suivantes en français en pseudo-code, sans référence à un langage de programmation particulier.

```
Fonction factorielle (n)
    r <- 1
    Pour i de 1 jusqu'à n faire
        r <- r*i
    Fin Pour
    Retourner r
Fin Fonction

a <- factorielle(5)
```

Les variables `r` et `i` sont locales, de type entier. La variable a est globale.

On peut au cours du déroulement d'une fonction faire appel à cette même fonction avec un ou des arguments « plus simples », de sorte qu'après un nombre fini d'appels récursifs, l'argument ou les arguments sont devenus triviaux, et on peut alors renvoyer le résultat. On parle alors de *fonction récursive*.

Exemple.

La fonction `factorielle` peut se réécrire sous forme récursive comme suit :

```
Fonction factorielle (n)
    Si n=0 alors
        r <- 1
    Sinon
        r <- n*factorielle(n-1)
    Retourner r
Fin Fonction
```

1.3 Terminaison, correction

Dans l'étude d'un algorithme ou d'une fonction se pose les questions de

- la terminaison : la fonction va-t-elle rendre un résultat (en un temps fini) ;
- la correction : le résultat rendu est-il celui attendu.

Terminaison

Pour s'assurer de la terminaison d'une fonction **itérative**, il suffit de vérifier la sortie des structures itératives conditionnelles (boucles `Tant que`).

Exemples.

La fonction `factorielle` (version itérative) termine puisqu'elle ne fait pas intervenir de boucle `tant que`.

Considérons les deux fonctions suivantes :

```
Fonction f(n)
    a <- n
    Tant que a>0 faire
        a <- a-1
    Fin Tant que
Fin Fonction

Fonction g(n)
    a <- n
    Tant que a>0 faire
        a <- a
    Fin Tant que
```

```
Fin Fonction
```

La fonction `f` termine pour un argument positif alors que la fonction `g` ne termine jamais pour un tel argument.

La terminaison d'une fonction **récursive** repose sur le principe d'induction et se vérifie ainsi :

- la fonction n'admet qu'un nombre fini d'appels récursifs et toujours avec des arguments strictement inférieurs,
- on aboutit toujours à un cas minimal, appelé cas de base, pour lequel la fonction termine.

Exemple.

La fonction `factorielle` (version récursive) termine puisque `factorielle(0)` termine, et que pour tout $n \in \mathbb{N}$, `factorielle(n)` n'admet qu'un nombre fini d'appels (un seul) qui s'effectue sur `factorielle(n-1)` où $n - 1 < n$.

Correction

Pour s'assurer de la correction d'une fonction **itérative**, il suffit d'exhiber un invariant de boucle, c'est-à-dire une quantité conservée depuis son premier calcul avec les arguments jusqu'à son expression à la sortie de la boucle.

Exemple.

Un invariant de boucle pour la fonction `factorielle` (version itérative) est : « à l'étape i, `r` est égal à $i!$ ».

Pour s'assurer de la correction d'une fonction **récursive**, on utilise le principe d'induction.

Exemple.

Prenons le cas encore une fois de la fonction `factorielle` (version récursive). On montre que le prédicat $\mathscr{P}(n)$: « `factorielle(n)` donne la valeur demandée » est vrai pour tout n par récurrence.

1.4 Complexité

En présence d'un algorithme, on souhaite savoir s'il s'exécute :

- en un temps « raisonnable » : c'est le problème de la *complexité temporelle* ;
- en utilisant une capacité mémoire « raisonnable » : il s'agit du problème de la *complexité spatiale*.

On traitera uniquement la complexité dans le pire des cas, c'est-à-dire lorsqu'à une taille n fixée, la donnée d'entrée de taille n est la plus défavorable (en nombre d'opérations). Voici les différentes classes de complexité selon les ordres de grandeurs :

$O(\log(n))$	logarithmique
$O(n)$	linéaire
$O(n\log(n))$	semi-linéaire
$O(n^2)$	quadratique
$O(n^k)$ $(k \geq 2)$	polynomiale
$O(k^n)$ $(k > 1)$	exponentielle

On se focalisera dans la suite sur le calcul de la complexité temporelle (les algorithmes que nous verrons n'étant pas très gourmands en espace mémoire). Pour la calculer, on commence par définir la ou les opérations à dénombrer (opérations arithmétiques, affectation, comparaisons, ... on pourra négliger les opérations les moins coûteuses) et la taille de la donnée en argument. Une fois ces critères fixés, la complexité s'obtient :

- comme une somme pour les structures itératives : on additionne le nombre d'opérations à chaque étape de la boucle ;
- comme le terme général d'une suite récurrente pour une fonction récursive.

Le calcul de la complexité d'un algorithme ne permet pas d'en déduire le temps d'exécution, mais seulement de comparer entre eux deux algorithmes résolvant le même problème. Cependant, il importe de prendre conscience des différences d'échelles considérables qui existent entre les classes de complexité. En s'appuyant sur une base de 10^9 opérations par secondes, on obtient :

	$\log(n)$	n	$n\log(n)$	n^2	n^3	2^n
10^2	7 ns	100 ns	0,7 μs	10 μs	1 ms	$4 \cdot 10^{13}$ années
10^3	10 ns	1 μs	10 μs	1 ms	1 s	10^{292} années
10^4	13 ns	10 μs	133 μs	100 ms	17 s	
10^5	17 ns	100 μs	2 ms	10 s	11,6 jours	
10^6	20 ns	1 ms	20 ms	17 min	32 années	

Une complexité temporelle supérieure à un coup quadratique est donc rédhibitoire pour l'exécution d'un algorithme en un temps raisonnable lorsque la taille des données est grande.

Exemple.
Dans l'algorithme `factorielle`, on effectue `n` multiplications, d'où un coût linéaire en $O(n)$.

2 Arithmétique dans $\mathbb{Z}$

2.1 Division euclidienne et algorithme d'Euclide

Division euclidienne

Rappelons que l'anneau $(\mathbb{Z}, +, \times)$ est euclidien avec pour stathme euclidien $N : n \in \mathbb{Z}^* \mapsto |n|$. Ainsi pour tout $a \in \mathbb{Z}$ et pour tout $b \in \mathbb{Z}^*$, il existe un unique couple $(q, r) \in \mathbb{Z}$ tel que :

$$a = b \times q + r \quad \text{avec} \quad 0 \leq r \leq |b| - 1.$$

De plus on a l'algorithme de division euclidienne suivant (écrit dans le cas où a et b sont positifs) :

```
Fonction div_eucl(a,b) //a et b des entiers, a>=0 et b>0
r <- a
q <- 0
Tant que r>=b faire
    r <- r-b
    q <- q+1
Fin Tant que
Retourner [q,r]
Fin Fonction
```

Justifions la terminaison de cet algorithme (itératif). Notons pour cela respectivement q_n et r_n les valeurs de q et r à la fin de la n-ième itération de la boucle `Tant que` et q_0 et r_0 leurs valeurs initiales. La suite des r_n est une suite d'entiers positifs et strictement décroissante, elle est donc finie. Puisque la condition d'arrêt de la boucle `Tant que` est que r_n soit $\leq b$, elle est atteinte en un temps fini.

Pour s'assurer de la correction de l'algorithme, on peut utiliser l'invariant de boucle suivant : à l'étape n, on a $bq_n + r_n = a$. On le montre aisément par récurrence sur n. De plus si $N \in \mathbb{N}$ est le nombre de boucles parcourues avant arrêt, on a :

- si $N = 0$, $r_0 = a \geq 0$ et $r_0 < b$ et on a $a = bq_0 + r_0$;
- si $N > 0$, on a $r_{N-1} > b$ et $r_N = r_{N-1} - b \in [\![0, b-1]\!]$ avec $a = bq_N + r_N$.

D'où la correction de l'algorithme.

Avant d'étudier la complexité de cet algorithme, discutons d'abord du type de données d'entrées. On va être amené à considérer de grands entiers pour certaines applications (cryptographie notamment), de l'ordre de $2^{1000} \simeq 10^{301}$ ou plus. On ne peut plus coder ces entiers sur 32 bits, et une approximation de ces nombres rendrait impossible toute manipulation arithmétique. On va donc stocker l'écriture binaire d'un grand entier n dans un tableau. Ce tableau contiendra $k+1$ termes avec k tel que $2^k \leq n < 2^{k+1}$, soit $k = \lfloor \log_2(n) \rfloor = O(\log_2(n)) = O(\ln(n))$. Pour deux entiers n et m de grande taille :

- l'addition se fait bit par bit, et son coût est donc de l'ordre de la taille du plus grand nombre, c'est à dire $O(\max(\ln(n), \ln(m))$.
- la multiplication correspond à $O(\ln(n))$ additions sur des nombres de longueur au plus $O(\ln(m))$, d'où un coût de $O(\ln(n)\ln(m))$.

L'algorithme de division euclidienne nécessite $2q$ additions sur des entiers de longueur au plus $\ln(a)$. Comme q est de l'ordre de a/b, on a donc environ a/b opérations à effectuer, soit au final une complexité en $O(\frac{a}{b}\ln(a))$.

Remarque. On peut écrire un algorithme de division euclidienne plus performant en $O(\ln(a)\ln(a/b))$ (voir [1]).

> **Exercice.**
> Écrire un algorithme prenant un entrée un entier naturel $n \in \mathbb{N}^*$ et un entier $b \geq 2$, et renvoyant la décomposition de n en base b. Justifier sa correction et sa terminaison. Calculer sa complexité.

Algorithme d'Euclide

On donne l'algorithme d'Euclide suivant :

```
Fonction euclide(a,b) //0<b<a
c <- a
d <- b
Tant que d <> 0 faire
    r <- div_eucl(c,d)(1)
    c <- d
    d <- r
Fin Tant que
Retourner c
Fin Fonction
```

Notons $r_{-1} = c_0 = a$, $r_0 = d_0 = b$ et r_k, c_k et d_k les valeurs respectives de r, c et d à la fin de la k-ième itération de la boucle `tant que`.

Pour s'assurer de la terminaison de cet algorithme, on notera que (d_k) est une suite d'entiers naturels strictement décroissante (ce sont les restes successifs des divisions euclidiennes). Elle est donc finie et la condition $d = 0$ est donc atteinte en un temps fini.

Pour établir la correction de l'algorithme, on peut utiliser l'invariant de boucle : « à l'étape k, on a $c \wedge d = a \wedge b$ ». On le montre aisément par récurrence sur k (exercice classique d'arithmétique). De plus pour N le nombre de boucles parcourues avant l'arrêt, on a :

$$a \wedge b = r_{-1} \wedge r_0 = r_{N-1} \wedge r_N = r_{N-1} \wedge 0 = r_{N-1}.$$

On obtient donc le résultat suivant :

Propriété 1

L'algorithme d'Euclide s'arrête au bout d'un nombre fini d'étapes. De plus, $a \wedge b$ est le dernier reste non nul dans l'algorithme d'Euclide.

Calculons la complexité de l'algorithme d'Euclide. Avec les notations introduites précédemment (avec $a > b > 0$), la complexité est majorée (à une constante près) par :

$$\sum_{k=0}^{N} \ln(\underbrace{r_{k-1}}_{\leq a}) \ln(\frac{r_{k-1}}{r_k}) \leq \ln(a) \sum_{k=0}^{N} \ln(\frac{r_{k-1}}{r_k}) = \ln(a) \ln(\frac{r_{-1}}{r_N}) = \ln(a) \ln(\frac{a}{a \wedge b}) \leq \ln(a)^2.$$

Ainsi la complexité de l'algorithme d'Euclide est en $O(\ln(a)^2)$

Outre la complexité de l'algorithme d'Euclide, on peut obtenir une majoration du nombre d'itérations en fonction de b : puisque la suite (r_k) est strictement décroissante et que $r_0 = b$, $r_N = 0$, on a déjà que $N \leq b$. Mais on peut faire mieux. Notons pour cela q_k le quotient de la division euclidienne à la k-ième itération, de sorte que :

$$c_{k-1} = q_k d_{k-1} + r_k.$$

Notons (F_k) la suite de Fibonacci, définie par :

$$F_0 = 0,\ F_1 = 1 \quad \text{et} \quad \forall k \geq 3,\ F_{k+2} = F_{k+1} + F_k.$$

On montre par récurrence que pour tout $0 \leq k \leq N-1$, on a $r_{N-1-k} \geq F_{k+2}$.

Init. On a $r_{N-2} > r_{N-1} > r_N = 0$ puisque la suite (r_k) est strictement décroissante, ce qui donne :

$$r_{N-1} \geq 1 = F_2 \quad \text{et } r_{N-2} \geq 2 = F_3.$$

Hér. Supposons la propriété vraie aux rang k et $k+1$ pour $0 \leq k \leq N-3$. Montrons la propriété au rang $k+2$. On a :

$$\begin{aligned} r_{N-1-(k+2)} = r_{N-k-3} &= q_{N-k-2} r_{N-k-2} + r_{N-k-1} \\ &\geq q_{N-k-2} F_{k+3} + F_{k+2} \quad \text{par hyp. de réc.} \\ &\geq F_{k+3} + F_{k+2} = F_{k+4} \end{aligned}$$

car $q_{N-k-2} \geq 1$ puisque la suite (r_k) est strictement décroissante.

Comme conséquence de cette inégalité, on a la

Propriété 2

Soit N un entier naturel non nul, et soient a et b deux entiers tels que $a > b$ et tels que l'algorithme d'Euclide appliqué à a et b nécessite exactement N divisions, et tels que a soit minimal pour cette propriété. Alors $a = F_{N+2}$ et $b = F_{N+1}$.

Preuve. Si on applique l'algorithme d'Euclide à F_{N+1} et F_N, on obtient :

$$F_{N+1} \wedge F_N = F_N \wedge F_{N-1} = \cdots = F_2 \wedge F_1 = F_1 \wedge F_0 = 1$$

On a donc exactement N divisions.

Réciproquement, on suppose que a et b satisfont les conditions de l'énoncé. On a d'après l'inégalité établie que :

$$\forall 0 \leq k \leq N-1, \quad r_{N-1-k} \geq F_{k+2}.$$

On en déduit que $b = r_0 \geq F_{N+1}$ et que :

$$a = r_0 q_1 + r_1 \geq r_0 + r_1 \geq F_{N+1} + F_N = F_{N+2}$$

□

Avec ces mêmes inégalités, on en déduit donc que :

$$b = r_0 \geq F_{N+1} \geq \phi^{n-1}$$

où $\phi = \dfrac{1+\sqrt{5}}{2}$ désigne le nombre d'or. La deuxième inégalité se démontre par récurrence immédiate. En prenant le logarithme, on obtient :

$$\log_{10}(b) \geq (n-1) \underbrace{\log_{10}(\phi)}_{> \frac{1}{5}} > \frac{n-1}{5}.$$

Si on note $p = \lfloor \log_{10}(b) \rfloor + 1$ le nombre de décimales dans l'écriture de b, on a donc :

$$p > \frac{n-1}{5} \quad \text{soit} \quad 5p > n-1 \quad \text{et donc} \quad 5p \geq n.$$

On a donc le résultat suivant.

Propriété 3 (Théorème de Lamé)

Soit $a > b > 0$ des entiers. On suppose que l'algorithme d'Euclide comporte n d'étapes avec $N \geq 2$. Alors N est majoré par 5 fois le nombre de décimales de b.

Algorithme d'Euclide étendu

On peut améliorer l'algorithme pour obtenir également une relation de Bezout. En effet si on écrit à chaque itération de l'algorithme $r_k = u_k a + v_k b$, avec $u_0 = 1$ et $v_0 = -q_0$, on aura :

$$r_{k+1} = r_{k-1} - q_k r_k = (u_{k-1} - q_k u_k)a + (v_{k-1} - q_k v_k)b.$$

On obtient par exemple l'algorithme suivant :

```
Fonction euclide_etendu(a,b) //0<b<a
c <- a ; d <- b ;
u1 <- 1 ; u2 <- 0 ;
Tant que d <> 0 faire
    [q,r] <- div_eucl(c,d)
    c <- d ; d <- r ;
    aux <- u2 ;
    u2 <- u1 - q * u2 ;
    u1 <- aux
Fin Tant que
u <- u1 ; v <- (r-u*a)/b ;
Retourner [c, u, v]
Fin Fonction
```

On notera que la complexité de l'algorithme d'Euclide étendu est inchangé.

2.2 Calcul modulaire

Coût du calcul modulaire

On se pose la question du coût du calcul modulaire. Soit pour cela $N > 0$ et k deux grands entiers, n et m deux entiers tels que $0 \leq n, m < N$.

- Le coût de calcul de $k[N]$ est en $O(\ln(k)\ln(k/N))$, puisque cela revient à faire une division euclidienne sur deux grands entiers k et N.
- Le coût de la somme $n + m[N]$ est celui de l'addition d'au plus trois grands entiers de taille N, donc en $O(\ln(N))$. En effet, il suffit d'appliquer l'algorithme de somme suivant :

  ```
  Si n+m-N<0 Alors
      Retourner n+m
  Sinon
      Retourner n+m-N
  Fin Si
  ```

- Le coût du produit $nm[N]$ est en $O(\ln(N)^2)$. En effet, il s'agit d'un produit de deux grands entiers suivit d'un calcul de modulo.
- Le coût de calcul d'une inversion modulo N, quand c'est possible, est en $O(\ln(N)^2)$. C'est le coût de l'algorithme d'Euclide.

Exponentiation rapide

Penchons nous à présent sur le calcul de puissance de grand nombres modulo N. Un premier algorithme qu'on pourrait proposé est le suivant :

```
Fonction Puissance(N,b,a)
c <- a [N] ;
Pour j de 2 à b faire
    c <- c * a [N]
Fin Pour
Retourner c
Fin Fonction
```

Regardons le coût de l'algorithme. On effectue $b-1$ multiplications modulo N avec des grands nombres, d'où un coût en $O(b(\ln N)^2) = O(e^{\ln(b)}(\ln N)^2)$.

Voyons maintenant un algorithme plus intéressant et moins coûteux. L'idée est la suivante : on décompose la puissance b en binaire, on calcule les carrés successifs de a $(a, a^2, a^4, a^8, \dots)$, puis on effectue le produit des carrés de a correspondant aux puissances de 2 ayant un facteur 1 dans l'écriture binaire de b.

Exemple.

Calculons $14^{201}[23]$ avec cette méthode. On commence par écrire 201 en base 2 :

$$201 = 2^7 + 2^6 + 2^3 + 2^0 = 11001001_2$$

Calculons maintenant les 7 carrés successifs de $14[23]$

$$14^{2^0} \equiv 14[23] \quad 14^{2^1} \equiv 12[23] \quad 14^{2^2} \equiv 12^2 \equiv 6[23] \quad 14^{2^3} \equiv 6^2 \equiv 13[23]$$
$$14^{2^4} \equiv 13^2 \equiv 8[23] \quad 14^{2^5} \equiv 8^2 \equiv 18[23] \quad 14^{2^6} \equiv 18^2 \equiv 2[23] \quad 14^{2^7} \equiv 2^2 \equiv 4[23]$$

On en déduit alors :

$$14^{201} \equiv 14^{2^7} \times 14^{2^6} \times 14^{2^3} \times 14^{2^0}[23] \equiv 4 \times 2 \times 13 \times 14[23] \equiv 7[23].$$

Voici une rédaction possible de cet algorithme (appelé algorithme d'exponentiation rapide) :

```
Fonction exponentiation(N,b,a)
c <- 1[N] ; p <- b ; x <- a [N] ;
Tant que (p<>1) faire
    Si (p est pair) Alors
        p <- p/2
    Sinon
        c <- c * x [N] ; p <- (p-1)/2
    Fin Si
    x <- x^2 [N]
Fin Tant que
c <- c * x [N]
Retourner c
Fin Fonction
```

On laisse le soin au lecteur de montrer la terminaison et la correction de cet algorithme. Penchons nous sur sa complexité. Notons pour cela $k = \lfloor \log_2(b) \rfloor + 1$ le nombre de chiffre dans l'écriture en base 2 de b. La boucle est itérée k fois, et à chaque passage dans la boucle, on effectue au plus 2 multiplications modulo N (les autres opérations sont négligeables en coût puisqu'elles correspondent à regarder ou supprimer le bit des unités de p). Cet algorithme est donc en $O(\ln(b)\ln(N)^2)$. On a une très nette amélioration de la complexité de l'algorithme.

2.3 Cryptographie

La cryptographie est un aspect essentiel de la sécurité des systèmes informatiques. Il en existe deux types :

- les *codes à clé privée* (pour les « gros » volumes de données), très rapides, mais qui nécessitent l'utilisation d'une même clé de taille fixée pour le cryptage et le décryptage. Les exemples historiques sont le chiffre de Jules César, de Vigenère ou de Hill. Les algorithmes couramment utilisés aujourd'hui sont l'AES et le DES.
- les *codes à clé publique* très lents mais où une partie des informations sur la clé peut être échangée sans sécurité (donc réservés à de petits volumes de données, quelques milliers de bits au plus). On dit que la clé est « rendue publique », c'est-à-dire qu'elle est communiquée de manière non cryptée. Ces systèmes servent souvent à échanger, de manière sécurisée, une clé pour un code à clé privée, les données étant ensuite cryptées avec AES à partir de cette clé. Les exemples les plus connus sont les systèmes d'El-Gamal et RSA.

Codes à clé privée

Citons seulement quelques codes à clé privée qu'on pourrait développer dans la leçon 170 *Méthodes de chiffrement ou de codage. Illustrations.* :

- le chiffre de Jules César,
- le chiffre de Vigenère
- le chiffre de Hill.

Codes à clé publique

La sécurité des cryptosystèmes à clé publique actuelle est basée sur la notion de *fonction à sens unique*. Il s'agit, intuitivement, d'une fonction $f : E \to F$ telle que :

- Il est possible de calculer simplement $f(x)$ à partir de n'importe quel $x \in E$,
- Pour la plupart des $y \in f(E)$, il n'est pas possible de trouver un x tel que $f(x) = y$, à moins d'exécuter un nombre prohibitif d'opérations, ou d'avoir une chance sur laquelle il est déraisonnable de compter.

Voici deux exemples classiques de fonctions à sens unique :

- le calcul de puissance dans $\mathbb{Z}/p\mathbb{Z}$ avec p premier suffisamment grand sur lequel repose le système de cryptage d'El-Gamal.
- la factorisation de grands entiers sur laquelle repose la sécurité du cryptage par RSA.

On développe dans la suite le fonctionnement de ces deux systèmes de cryptage.

Système d'El-Gamal.

C'est un système de cryptage inventé en 1985 par Taher Elgamal. Supposons que Bob désire envoyer un message $M \in \mathbb{N}$ crypté à Alice.

Principe.

1. Bob ou Alice choisit et publie un grand entier premier $p > M$ de grande taille (de l'ordre de 1000 bits, c'est à dire $2^{1000} \simeq 10^{301}$) et un entier g tel que $g[p]$ soit un générateur de $(\mathbb{Z}/p\mathbb{Z})^*$.
2. Bob choisit alors un entier $2 \leq b \leq p-2$ qu'il garde secret, calcule $y \equiv g^b[p]$ soit un générateur de $(\mathbb{Z}/p\mathbb{Z})^*$ et rend public y. Alice choisit un entier $2 \leq a \leq p-2$ qu'elle garde secret, calcule $x \equiv g^a[p]$ et rend public x.
3. Bob et Alice calculent $c \equiv g^{ab}[p] \equiv x^b[p] \equiv y^a[p]$ qui sera leur clef privée.

4. Bob code son message M et $T \equiv M \times c[p]$ qu'il envoie à Alice.

5. Alice calcule $c^{-1}[p]$ en utilisant l'algorithme d'Euclide étendu. Elle retrouve alors M par la formule $M \equiv T \times c^{-1}[p]$

Alice et Bob sont les seuls à pouvoir chiffrer et déchiffrer un message. En effet, tout le monde connait $p, g, x \equiv g^a[p], y \equiv g^b[p]$ et T. Il est cependant très difficile d'en déduire la valeur de a ou b nécessaire pour obtenir c et donc de retrouver M. C'est ce qu'on appelle le *problème du logarithme discret*. Il existe des algorithmes pour obtenir a, mais le temps de calcul est prohibitif, ce qui rend le système relativement sûr (voir à ce propos [ ]).

Ce système repose donc sur une fonction à sens unique : $x \mapsto g^x[p]$ se calcule en temps polynomial (en $O(\ln(x)\ln(p)^2$ par l'algorithme d'exponentiation rapide), mais la résolution de l'équation $y \equiv g^x[p]$ d'inconnue x se fait au mieux en temps sous-exponentiel.

Le système RSA.

Ce système a été mis au point en 1983 par Rivest, Shamir et Adelman (d'où son nom qui reprend les initiales de chacun des auteurs). Supposons encore une fois que Bob veut envoyer un message confidentiel $M \in \mathbb{N}$ à Alice, codé par RSA.

Principe.

1. Alice choisit deux nombres premiers p et q de grande taille. Elle note $n = pq > M$ le produit de ces deux nombres.

2. Alice choisit ensuite un autre entier e tel que e et $(p-1)(q-1)$ soient premiers entre eux. Alors on sait (par la formule de Bezout) qu'il existe un entier $d \in \mathbb{N}$ tel que $ed \equiv 1[(p-1)(q-1)]$.

3. Alice rend publique la clef (e, n), dont Bob aura besoin pour coder le message, et elle garde secret le nombre d qui est nécessaire au décryptage (les autres entiers ne servent plus).

4. Bob code son message M et $T \equiv M^e[n]$, qu'il transmet à Alice.

5. Alice, qui a reçu le message codé T, peut retrouver M par la formule $M \equiv T^d[n]$. En effet, on a :
$$T^d \equiv M^{ed}[n] \equiv M^{1-k(p-1)(q-1)}[n] \equiv M[n].$$
Pour s'en convaincre, il suffit de vérifier ces égalités modulo p et q, le théorème chinois permettant de conclure modulo n.

Notons ici que la clé de cryptage est différente de la clé de décryptage, qui ne peut en être déduite facilement : on parle de *systèmes à clés asymétriques* (par opposition aux *systèmes à clés symétriques*, dont El-Gamal fait partie, où la même clé sert à crypter et décrypter).

La sûreté du code RSA repose sur la fonction à sens unique $f : (p, q) \in \mathbb{P} \times \mathbb{P} \mapsto p \cdot q \in \mathbb{N}$. On a en effet des algorithmes performants pour calculer $p \cdot q$ même pour des grands nombres. Mais pour $n \in \mathbb{N}$ produit de deux grands nombres premiers p et q, on ne sait pas déterminer p et q facilement. La méthode du crible d'Eratosthène va demander un temps exponentiel et est donc inapplicable. D'autres techniques plus efficaces existent (voir notamment [ ]), mais elles ont une complexité exponentielle ou sous-exponentielle au mieux, et restent donc très lentes.

Mise en oeuvre des systèmes El-Gamal et RSA.

Pour pouvoir utiliser ces systèmes à clé public de manière efficace (avec un temps de calcul faible), il faut savoir calculer rapidement de grandes puissances modulaires, ce qu'on sait faire via l'algorithme

d'exponentiation rapide. Il faut aussi trouver de grands nombres premiers, ce qui n'est pas une chose aisée. En effet, il n'existe pas d'algorithme capable de certifier qu'un nombre est ou non premier en un temps polynomial. Par contre, il existe des algorithmes probabilistes donnant une certaine chance qu'un nombre entier n soit premier. L'un des plus connu est le test de Rabin-Millet qui repose entre autres sur le petit théorème de Fermat (voir à ce sujet [1]).

3 Algèbre linéaire

On se penche sur la résolution d'un système linéaire $AX = B$ où $A \in \mathscr{M}_n(\mathbb{R})$ est la matrice des coefficients du système, $B \in \mathscr{M}_{n,1}(\mathbb{R})$ est la matrice colonne du second membre, et $X \in \mathscr{M}_{n,1}(\mathbb{R})$ est la matrice colonne des inconnues.

3.1 Méthode de remontée

Traitons pour commencer du cas où $A \in \mathscr{M}_n(\mathbb{K})$ triangulaire (supérieure par exemple) et inversible. On utilise la *méthode de remontée* :

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n & = b1 \\ a_{2,2}x_2 + \cdots + a_{2,n}x_n & = b_2 \\ \quad \ddots \quad \vdots & = \vdots \\ \qquad a_{n,n}x_n & = b_n \end{cases}$$

Puisque les $a_{i,i}$ sont tous non nuls, on a :

$$\begin{cases} x_n & = a_{n,n}^{-1} b_n \\ x_{n-1} & = a_{n-1,n-1}^{-1}(b_n - a_{n-1,n}x_n) \\ & \cdots \\ x_1 & = a_{1,1}^{-1}(b_1 - a_{1,n}x_n - \cdots - a_{1,2}x_2) \end{cases}$$

Propriété 4

La complexité de la « remontée » est en $\dfrac{n(n-1)}{2}$ additions et multiplications, et n divisions, soit de l'ordre de $O(n^2)$ opérations.

Toutes les décompositions présentées dans la suite permettent de ramener la résolution d'un système linéaire à celle plus simple d'un ou plusieurs systèmes linéaires mis sous forme triangulaire. Il faudra alors appliquer la méthode de remontée pour terminer la résolution des systèmes considérés. On verra que son coût est négligeable par rapport au coût de la mise sous forme triangulaire.

3.2 Pivot de Gauss et applications

On suppose ici que A est une matrice carrée inversible. L'algorithme de Gauss est à la base du résultat suivant.

Propriété 5

Pour toute matrice A inversible, il existe une matrice M inversible telle que MA soit triangulaire supérieure inversible.

Preuve. On applique l'algorithme du pivot de Gauss pour ramener par opérations élémentaires **sur les lignes** la matrice A à une matrice triangulaire supérieure. Chaque opération élémentaire correspondant au produit **à gauche** par une matrice de transvection ou de dilatation (inversible), on en déduit l'existence de M. □

Propriété 6

La résolution d'un système linéaire $AX = B$ (avec A inversible) par l'algorithme de Gauss se fait en $O\left(\frac{n^3}{3}\right)$ additions et multiplications, et $O\left(\frac{n^2}{2}\right)$ divisions.

Conséquence. Le coût du calcul du déterminant, de A^{-1}, voir plus généralement du rang d'une matrice, est en $O(n^3)$ additions et multiplications et en $O(n^2)$ divisions. Par comparaison, le calcul de M^2 nécessite $2n^3$ additions et multiplications.

Remarque. On pourrait utiliser un algorithme récursif pour calculer le déterminant d'une matrice A en utilisant la formule de développement du déterminant par rapport à la première ligne (par exemple) :

$$\det(A) = \sum_{j=1}^{n} (-1)^{j+1} a_{1,j} \Delta_{1,j}$$

où $\Delta_{i,j}$ est le mineur d'ordre (i,j) de A. Si on note u_n le nombre d'opérations pour calculer un déterminant d'ordre n, on a la relation de récurrence :

$$u_n = n u_{n-1} + 2n.$$

On a donc :

$$u_n \geq n u_{n-1} \geq \cdots \geq n(n-1)\dots 3 u_2 = n!$$

Cette méthode est donc très loin d'être aussi efficace que l'algorithme de Gauss. Pour $n = 20$ par exemple, et en s'appuyant sur une base de 10^9 opérations par secondes, on obtient :

- par la méthode de Gauss : $\frac{20^3}{10^9} = 8\mu s$;
- en développant suivant une ligne : $\frac{20!}{10^9} \simeq 2,5 \times 10^9 s$ soit plus de 79 années.

Outre le problème du coût temporel de la résolution du système, se pose également la question de l'*instabilité numérique* (erreurs d'arrondi par la machine menant à des résultats complètement faux). Commençons par un exemple illustrant les problèmes numériques que l'on peut rencontrer.

Exemple.

Considérons le système :

$$(\mathscr{S}) : \quad \begin{cases} 10^{-4}x - y &= -1 \\ x + y &= 2 \end{cases}$$

dont la solution exacte est :

$$x = \frac{1}{1+10^{-4}} = 0,99989998\dots \quad , \quad y = \frac{1+2\cdot 10^{-4}}{1+10^{-4}} = 1,00010001\dots$$

Les écritures avec trois chiffres significatifs des solutions sont donc $x = 1$, $y = 1$. On va résoudre ce système en faisant les calculs eux-même avec trois chiffres significatifs (taille de la mantisse). Rappelons que c'est ce qui se passe en machine (avec une précision plus importante bien sûr) du fait de la représentation des réels par des flottants. Résolvons ce système par deux méthodes.

- En prenant pour pivot le premier coefficient 10^{-4} (très petit par rapport aux autres coefficients !). On trouve :

$$(\mathscr{S}) \underset{L_2 \leftarrow L_2 - \frac{1}{10^{-4}} L_1}{\Leftrightarrow} \left\{ \begin{array}{rcl} 10^{-4}x - y & = & -1 \\ (1+10^4)y & = & 2+10^4 \end{array} \right.$$

Mais avec trois chiffres significatifs, on a $10^4 + 1 = 10^4$ et $10^4 + 2 = 10^4$ en machine, donc le système est devenu :

$$\left\{ \begin{array}{rcl} 10^{-4}x - y & = & -1 \\ 10^4 y & = & 10^4 \end{array} \right.$$

avec comme solution $x = 0$ et $y = 1$. On est alors très loin des solutions que l'on aimerait trouver !

- Cela se passe beaucoup mieux en prenant pour pivot le deuxième coefficient de la première colonne (coefficient le plus grand en valeur absolue). On trouve dans ce cas :

$$(\mathscr{S}) \underset{L_1 \leftarrow L_1 - 10^{-4} L_2}{\Leftrightarrow} \left\{ \begin{array}{rcl} -(1+10^{-4})y & = & -(1+10^{-4}) \\ x + y & = & 2 \end{array} \right.$$

On a $1 + 10^{-4} = 1$ à trois chiffres significatifs près, et donc le système est devenu dans ce cas :

$$\left\{ \begin{array}{rcl} -y & = & -1 \\ x + y & = & 2 \end{array} \right.$$

avec comme solution $x = 1$ et $y = 1$, c'est-à-dire les véritables solutions à trois chiffres significatifs.

L'exemple précédent montre que des erreurs de calculs importantes peuvent survenir du fait de la représentation des réels par les flottants en machine. Il montre également que ces phénomènes d'instabilité numérique peuvent être limités dans le cas de l'algorithme de Gauss en cherchant le meilleur pivot à chaque étape : il s'agit du coefficient le plus grand en valeur absolue.

3.3 Factorisation LU

Propriété 7 (*Décomposition LU*)

Soit $A \in \mathscr{M}_n(\mathbb{R})$ inversible, ainsi que tous ses mineurs principaux. Alors A se décompose sous la forme :

$$A = LU$$

avec L triangulaire inférieure avec des 1 sur la diagonale, et U triangulaire supérieure, la décomposition étant unique.

Preuve. Cette décomposition repose là encore sur l'algorithme de Gauss. L'hypothèse sur A et ses mineurs permet d'appliquer l'algorithme en prenant systématiquement pour pivot le coefficient diagonal (on peut en effet montrer que sous ces hypothèse, ce coefficient sera toujours non nul). La mise sous forme triangulaire de A fournit une matrice L' triangulaire inférieure avec des 1 sur la diagonale, produit de matrices de transvections (afin d'annuler tous les coefficients sous la diagonale), de sorte que :

$$L'A = U$$

où U est triangulaire supérieure. On obtient la décomposition $A = LU$ en posant $L = L'^{-1}$. L'unicité est laissée en exercice. □

Remarque. Si A est une matrice inversible, on peut montrer qu'il existe une matrice de permutation P telle que les mineurs principaux de la matrice PA soient tous inversibles (la matrice PA se déduit

de A par permutation de ses lignes). En appliquant le résultat précédent, on montre donc qu'il existe une matrice L triangulaire inférieure avec des 1 sur la diagonale, et U triangulaire supérieure, telles que :

$$PA = LU.$$

On parle encore de décomposition LU pour A.

Intérêt. Cette décomposition permet de ramener la résolution d'un système linéaire à la résolution de deux systèmes linéaires triangulaires qu'on résoudra par remontée :

$$AX = B \quad \Leftrightarrow \quad L(UX) = B \quad \Leftrightarrow \quad \begin{cases} LY = B \\ UX = Y \end{cases}.$$

Cette méthode devient intéressante quand on a plusieurs systèmes linéaires avec des second membres différents. En effet si on change B, on peut ré-exploiter notre décomposition LU, contrairement à la méthode de Gauss qui nécessiterait de reprendre les calculs.

Complexité. La complexité de la décomposition LU est la même que celle de l'algorithme de Gauss.

Inconvénient. L'inconvénient de cette méthode, comparée à celle du pivot de Gauss, est qu'on ne cherche pas le meilleur pivot par colonne (c'est à dire le plus grand en valeur absolue), puisqu'on prend systématiquement le coefficient diagonal. Cela peut provoquer de l'instabilité numérique comme on l'a vu sur un exemple plus haut, et donc des erreurs de calculs plus importants qu'avec la méthode de Gauss.

3.4 Méthode de Cholesky

Propriété 8 (*Méthode de Cholesky*)

Si A est symétrique définie positive, alors il existe une matrice B triangulaire inférieure telle que :

$$A = B\ {}^tB.$$

Cette décomposition est de plus unique si on suppose de plus que $B_{i,i} > 0$ pour tout i.

Preuve. Je vous renvoie à [1]. □

Intérêt. Il est ici le même que pour la décomposition LU : on se ramène à des systèmes triangulaires, pour lesquels il suffit de faire la remontée.

Complexité. On montre que pour cette méthode, la complexité est en $\frac{n^3}{6}$ additions et multiplications, $\frac{n^2}{2}$ divisions et n racines carrées.

Remarque. Noter que si A est seulement inversible, alors le système $AX = B$ est équivalent au système ${}^tAAX = {}^tAB$, et que ${}^tAA$ est symétrique définie positive. On peut donc appliquer la méthode de Cholesky par cette astuce pour résoudre le système linéaire.

3.5 Méthode QR

Propriété 9 (*Méthode QR*)

Soit $A \in \mathscr{M}_n(\mathbb{K})$. Alors A peut s'écrire sous la forme :

$$A = QR$$

où Q est unitaire et R est triangulaire supérieure dont les coefficients diagonaux sont ≥ 0. La décomposition est de plus unique si A est inversible.

Preuve. Cette décomposition est basée sur l'algorithme de Gram-Schmidt. Je vous renvoie à [1]. □

Complexité. On peut montrer qu'elle est en $O(2n^3)$ opérations.

Intérêt. On ramène la résolution d'un système linéaire essentiellement à la résolution d'un système triangulaire, l'inversion de Q étant directe puisque $Q^{-1} = {}^t\overline{Q}$.

Étant donné sa complexité, cette méthode n'a pas d'intérêt si la matrice A est inversible, car les méthodes précédentes sont plus efficaces. Par contre elle permet d'obtenir un système triangulaire même si A n'est pas inversible, et peut s'étendre aux matrices non carrées.

Inconvénients. Cette méthode est très instable numériquement. Une manière de résoudre cette difficulté est d'utiliser les matrices de Householder. Voir à ce sujet [1].

3.6 Méthode de Cramer

Propriété 10 (*Formules de Cramer*)

Supposons A inversible, et notons $C_1, \dots, C_n$ les colonnes de A.
Les solutions du système de Cramer $AX = B$ sont :

$$\forall k \in [\![1, n]\!], \quad x_k = \frac{\det(C_1, \dots, \boxed{B}, \dots, C_n)}{\det(C_1, \dots, \boxed{C_k}, \dots, C_n)}.$$

L'utilisation de ces formules n'est en pratique pas intéressante : en effet, elle nécessite le calcul de $n+1$ déterminants dont le coût (à l'aide de l'algorithme de Gauss) est en $O(n^3)$, soit un coût total de $O(n^4)$. C'est donc beaucoup plus qu'en appliquant l'algorithme de Gauss. Cette méthode est de plus numériquement instable : même pour un système d'ordre 2, le résultat peut être complètement faux. Il s'agit donc de la pire méthode, à ne jamais utiliser en pratique.

En fait ces formules de Cramer ont seulement un intérêt théorique. Elles permettent par exemple de justifier de la continuité des solutions du système en fonction du second membre.

Annexe : Représentation d'un réel en base b

Commençons par donner la définition de la représentation d'un réel en base b, b étant un entier supérieur ou égal à 2.

Définition.

Soit $x \in \mathbb{R}$. Soit $k \in \mathbb{N}$, $s = \pm 1$, $(n_j)_{0 \leq j \leq k}$ et $(m_j)_{j \in \mathbb{N}^*}$ deux familles d'entiers de $\{0, \dots, b-1\}$. On dit que k, s et les deux familles forment une représentation de x en base b si :

$$s \times x = \sum_{j=0}^{k} n_j b^j + \sum_{j=1}^{+\infty} m_j b^{-j}.$$

On le note alors $x = s \times n_k \dots n_0, m_1 m_2 \dots_b$.

Remarque. Cette écriture à toujours un sens. En effet, la série $\sum_{j=1}^{+\infty} m_j b^{-j}$ est convergente car :

$$\sum_{j=1}^{+\infty} m_j b^{-j} \leq (b-1) \sum_{j=1}^{+\infty} b^{-j} = (b-1) \times \frac{1}{b} \frac{1}{1-\frac{1}{b}} = 1.$$

Exemple.

Étudions les représentations binaires de 145 et $-23,6875$. On a :

$$145 = 128 + 16 + 1 = 2^7 + 2^4 + 2^0.$$

Donc 10010001_2 est une représentation binaire de 145. De la même façon, on a :

$$\begin{aligned} 23,6875 &= 16 + 4 + 2 + 1 + \frac{11}{16} = 16 + 4 + 2 + 1 + \frac{8+2+1}{16} \\ &= 2^4 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-3} + 2^{-4} \end{aligned}$$

Donc $-10111,1011$ est une représentation binaire de $-23,6875$.

Voyons le cas particulier des entiers, qui peuvent s'écrire de façon unique avec une famille $(m_j)_{j \in \mathbb{N}^*}$ constante nulle.

Propriété 11

Soit $b \in \mathbb{N}$, $b \geq 2$. Soit $n \in \mathbb{Z}^*$.
Il existe un unique $k \in \mathbb{N}^*$, un unique $s = \pm 1$ et une famille $\{n_0, \dots, n_k\}$ d'entiers dans $\{0, \dots, b-1\}$, $n_k \neq 0$, tels que :

$$n = s \times n_k n_{k-1} \dots n_{0\,b} = s \times \sum_{j=0}^{k} n_j b^j.$$

Preuve. On effectue la division euclidienne de $|n|$ par b : il existe un unique couple (q_0, n_0) d'entiers tels que :

$$n = q_0 b + n_0 \quad \text{avec} \quad 0 \leq n_0 \leq b-1.$$

Si $0 = q_0$, on pose $k = 0$ et on obtient l'existence de la décomposition. De même si $0 < q_0 \leq b-1$, on pose $k = 1$, $n_1 = q_0$ et on a l'existence. Sinon on effectue la division euclidienne de q_0 par b : il existe un couple d'entiers (q_1, n_1) tels que :

$$q_0 = q_1 b + n_1 \quad \text{avec} \quad 0 \leq n_1 \leq b-1$$

de sorte que :

$$n = q_0 b + n_0 = q_1 b^2 + n_1 b + n_0.$$

On discute alors de la valeur de q_1, et on répète si besoin cette procédure. Notons que la suite d'entiers naturels ainsi construite $(q_i)_i$ ainsi construite est strictement décroissante. Il existe donc $k \in \mathbb{N}$ tel que $q_k \leq 0$ et $q_{k+1} = 0$, et par construction on obtient l'existence également des $n_i \in [\![0, \dots, b-1$ avec $n_k = q_k \neq 0$ tels que :

$$|n| = \sum_{i=0}^{k} n_i b^i.$$

L'unicité est laissée en exercice (voir éventuellement [1]). □

Remarque. Si $n = n_k \dots n_1 n_{0\,b}$, on a :

$$b^k \leq n < b^{k+1} \quad \Rightarrow \quad k \leq \frac{\ln(n)}{\ln(b)} < k+1.$$

Ainsi on a $k = \log_b(n)$.

On a donc une seule écriture d'un entier non nul sans chiffre après la virgule, quelle que soit la base b. Cependant, si on considère les entiers comme des rationnels, l'unicité n'est plus assurée. Par exemple, un calcul précédent montre que :

$$1 = 1_b = 0, ccc \dots_b$$

où $c = b - 1$. En particulier en base décimale, on a :

$$1 = 0, 999 \dots$$

En fait, on peut montrer que les rationnels sont les seuls réels pouvant s'écrire sous plusieurs formes dans certaines bases. Un irrationnel n'aura qu'une écriture possible, quelle que soit la base choisie.

En machine, n'importe quel réel (en fait, n'importe quel caractère) est représenté par des 0 et des 1 (des bits), donc en base $b = 2$.

Bibliographie

[1] Loïc Foissy, Alain Ninet. Algèbre et calcul formel. Edition Ellipses.

[2] Roger Mansuy. Option informatique - MPSI-MP/MP*.

[3] Bernard Parisse. Algorithmique (Agrégation Interne). Disponible à l'adresse `https://www-fourier.ujf-grenoble.fr/~parisse/agregint.html`