

Codage des nombres

Arnaud Labourel

1 Introduction au codage

Il s'agit, dans ce chapitre, de donner les bases sur les manières de coder de l'information. On abordera le codage des nombres et donc sur les principes des systèmes de numération. On s'intéressera surtout à la base dix, dite décimale (la manière la plus commune pour représenter des nombres) et à la base deux, dite binaire (la manière utilisée pour coder les nombres dans un ordinateur).

2 Représentation des nombres

2.1 Définition

Les *systèmes de numération* permettent de représenter commodément tous les entiers naturels à l'aide d'un nombre fini de symboles. Un système de numération est défini par un entier $\beta > 1$ appelé *base* du système et d'un ensemble de β symboles appelés *chiffres* qui représentent chacun un entier compris entre 0 et $\beta - 1$ ¹.

Le théorème suivant est essentiel pour la représentation en base β d'un entier.

Soit x un élément de $\mathbb{N}$. Il existe des entiers $a_0, a_1, \dots, a_n$ compris 0 et $\beta - 1$ tels que :

$$x = a_n\beta^n + a_{n-1}\beta^{n-1} + a_{n-2}\beta^{n-2} + \dots + a_1\beta + a_0$$

Les entiers $n, a_0, a_1, \dots, a_n$ sont déterminés de manière unique si l'on convient que a_n n'est pas nul.

En identifiant les entiers $a_0, a_1, \dots, a_n$ à des chiffres, on obtient la représentation en base β de x . Cette représentation est le mot $a_n a_{n-1} a_{n-2} \dots a_2 a_1 a_0$.

L'emploi d'un système de numération offre plusieurs avantages :

- Il suffit d'un ensemble fini de symboles pour représenter tous les entiers.
- La comparaison de deux entiers est facile.
- Pour effectuer toutes les opérations de calcul algébrique, il suffit de connaître les tables d'addition et de multiplication pour les nombres entre 0 et $\beta - 1$.

Les systèmes de numération les plus couramment utilisés sont décrits dans le tableau suivant.

	Système	Base	Chiffres
décimal	dix	0 1 2 3 4 5 6 7 8 9	
binaire	deux	0 1	
octal	huit	0 1 2 3 4 5 6 7	
hexadécimal	seize	0 1 2 3 4 5 6 7 8 9 A B C D E F	

Puisqu'on utilise les mêmes chiffres 0 et 1 pour toutes les bases, il est souvent impossible de déterminer la base d'une représentation. Par exemple, 11 vaut onze en base dix mais seulement trois en base deux. Afin d'éviter la confusion, on mettra en indice de la représentation la valeur de la base. Par exemple, on a $11_2 = 3_{10}$.

¹ On identifie souvent les chiffres avec les entiers qu'ils représentent

Pour représenter les entiers relatifs, il suffit d'ajouter un signe $+$ ou $-$ devant la représentation de leur valeur absolue. Il existe une méthode pour représenter les nombres réels. Néanmoins, cette représentation n'est pas forcément unique, ni finie (c'est-à-dire utilisant une suite finie de symboles). On utilise le théorème suivant :

Soit x un nombre réel strictement positif. Alors il existe au moins une suite d'entiers $a_n, a_{n-1}, \dots, a_1, a_0, d_1, d_2, d_3, \dots$ tels que la différence :

$$x - \left(\sum_{i=0}^n a_i \beta^i + \sum_{i=1}^k d_i \beta^{-i} \right)$$

soit positive et tende vers 0 quand k tend vers l'infini.

Ce résultat permet de représenter² le nombre réel x par :

$$x = a_n a_{n-1} \dots a_2 a_1 a_0, d_1 d_2 \dots$$

Pour les nombres réels négatifs, on calcule la représentation de leur valeur absolue et on ajoute le signe $-$ ou $+$ comme on le fait pour les entiers relatifs.

Le nombre $x = \frac{-3}{100}$ peut s'écrire en base dix :

$$x = -0,030000\dots \quad \text{ou} \quad x = -0,029999\dots$$

2.2 Conversion

Connaissant l'écriture d'un entier dans la base β , il arrive qu'on ait besoin de connaître son écriture dans une autre base β' . On nomme *conversion* le passage d'une écriture dans une base vers une autre base.

Méthode générale de conversion de la base β à la base β'

Pour convertir un entier x de la base β à la base β' , on passe par les étapes suivantes :

Étape 1 : On fait la division entière de x (écrit en base β) par β' . On divise le quotient obtenu par β' et on continue à diviser le quotient obtenu jusqu'à qu'on arrive à un quotient nul.

Étape 2 : On convertit tous les restes des divisions en les écrivant en base β' . Les restes sont compris entre 0 et $\beta' - 1$ et correspondent donc chacun à un chiffre de la base β' .

Étape 3 : On écrit les restes convertis dans l'ordre inverse dans lequel on les a trouvés de gauche à droite. Cela nous donne l'écriture en base β' de x .

Écriture de 10111011111_2 en base dix.

Étape 1 : On a $\beta' = 1010$ en base deux. On effectue les divisions³ :

² En théorie, car en pratique on ne peut pas écrire une infinité de chiffres.

³ On verra plus en détail comment faire une division en binaire dans la suite du chapitre

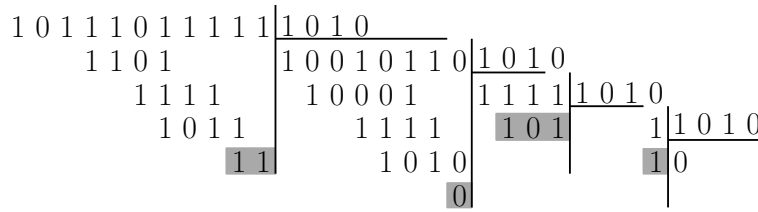


FIGURE 1 – image

Étape 2 : conversion des restes :

$$\begin{array}{cccc}
 11 & 0 & 101 & 1 \\
 \downarrow & \downarrow & \downarrow & \downarrow \\
 3 & 0 & 5 & 1
 \end{array}$$

Étape 3 : On écrit les restes dans le sens inverse ce qui nous donne 1 5 0 3. On a donc $1011101111_2 = 1503_{10}$.

On peut vérifier le résultat en effectuant la conversion inverse, c'est-à-dire en convertissant 1503_{10} en base deux :

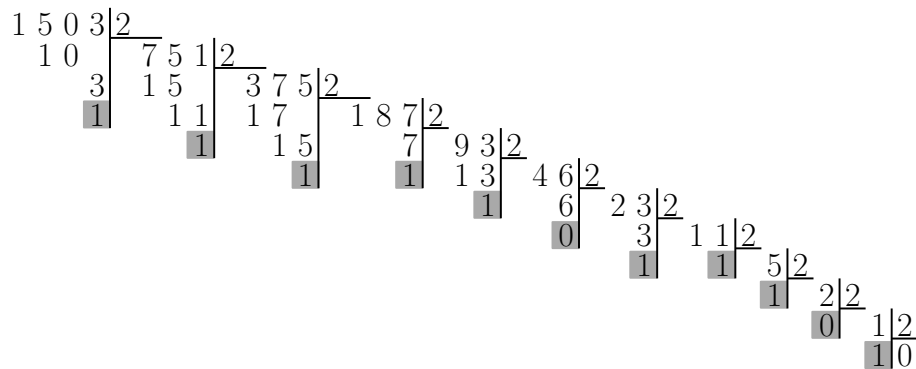


FIGURE 2 – image

Il est ici trivial de convertir les restes car les chiffres 0 et 1 correspondent dans toutes les bases aux nombres zéro et un. Il suffit donc de les écrire en sens inverse pour obtenir : $x = 1011101111_2$.

Méthode pratique de conversion de la base β à la base β'

La méthode générale de conversion peut poser problème si la base de départ n'est pas la base décimale. Réaliser une division dans une base quelconque n'est pas évident, car bien que les principes de calculs restent les mêmes, les détails changent. Généralement, il est plus simple de commencer à convertir la représentation du nombre en base dix. Pour cela, il suffit d'appliquer la formule du théorème [th:formule] et de calculer la somme en base décimale.

On cherche à convertir 1234_5 de la base 5 vers la base 16. Pour cela, on commence à convertir 1234_5 en base 10. On a $1234_5 = (1 \times 5^3 + 2 \times 5^2 + 3 \times 5^1 + 4 \times 5^0)_{10} = (125 + 50 + 15 + 4)_{10} = (194)_{10}$. Ensuite, on convertit le résultat obtenu en base 16.

$$\begin{array}{r|l}
 194 & 16 \\
 16 & 12 \\
 34 & 120 \\
 2 &
 \end{array}$$

FIGURE 3 – image

Le résultat final est $C2_{16}$.

La conversion d'une base à l'autre se simplifie considérablement quand une des bases est une puissance entière d'une autre. Nous allons distinguer deux cas suivant que l'ancienne base est une puissance de la nouvelle ou l'inverse.

Si la première base est une puissance de l'ancienne, c'est-à-dire que $\beta' = \beta^k$ avec k entier, alors l'entier β' s'écrit dans la base β avec le chiffre 1 suivi de k chiffres 0. Le reste de la division euclidienne d'un entier x par β' s'obtient donc en recopiant les k derniers chiffres de cet entier. Le quotient quant à lui est obtenu en recopiant les k chiffres restants. Par exemple, le quotient de la division de 123023_{10} par 100_{10} est 1230_{10} et le reste est 23_{10} . Les divisions en cascade de l'étape 1 de la méthode de conversion universelle ont donc pour effet de découper l'écriture de x en tranche de k chiffres. Le découpage se fait de droite à gauche, la dernière tranche de chiffres pouvant être incomplète.

Méthode de conversion de la base β à la base β^k

Pour convertir un entier x de la base β à la base β^k , on passe par les étapes suivantes :

Étape 1 : On découpe l'écriture en base β du nombre x en tranche de k chiffres en partant de la droite.

Étape 2 : On convertit chaque tranche de k chiffre en la remplaçant par le chiffre correspondant de la nouvelle base, ce qui nous donne l'écriture de x en base β^k .

On cherche à faire la conversion de 1011101111_2 en base huit. Appliquons d'abord la méthode générale. on a $\beta' = 1000_2$ et l'étape 1 donne :

$$\begin{array}{r|l}
 1011101111 & 1000 \\
 1110 & 10111011 & 1000 \\
 1101 & 1110 & 10111 & 1000 \\
 1011 & 1101 & 111 & 10 & 1000 \\
 1111 & 1011 & 10 & 0 \\
 1111 & 011 &
 \end{array}$$

FIGURE 4 – image

On constate que les restes trouvés : 111, 011, 111 et 10 s'obtiennent bien en découplant 1011101111 en tranches de trois chiffres à partir de la droite.

Ensuite l'étape 2 donne :

$$\begin{array}{cccc} 111 & 011 & 111 & 10 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 7 & 3 & 7 & 2 \end{array}$$

On obtient donc $10111011111_2 = 2737_8$. Avec la méthode simplifiée, il suffit de découper x en tranche de trois chiffres à partir de la droite puis de convertir chaque tranche en base huit :

$$\begin{array}{cccc} 10 & 111 & 011 & 111 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 2 & 7 & 3 & 7 \end{array}$$

Maintenant passons aux conversions dans l'autre sens, c'est-à-dire quand l'ancienne base est une puissance de la nouvelle : $\beta = \beta'^k$.

Cette conversion est encore plus simple car il suffit de remplacer chaque chiffre de x par son écriture dans la nouvelle base.

Pour convertir 5437_8 en base deux, il suffit d'écrire :

$$\begin{array}{cccc} 5 & 4 & 3 & 7 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 101 & 100 & 011 & 111 \end{array}$$

On obtient $5437_8 = 101100011111_2$.

Ce type de conversion s'applique surtout aux bases deux, huit et seize. En informatique, on travaille souvent avec mots binaires de longueur huit qu'on appelle *octets*. Pour pouvoir représenter un octet avec un seul symbole, il faudrait $2^8 = 256$ symboles différents, ce qui est beaucoup trop⁴. Pour représenter un octet de manière compacte, on coupe celui-ci en deux tranches de 4 bits qu'on remplace par des chiffres correspondant de la base seize ou hexadécimale.

Pour représenter l'octet 01001101 on le coupe en deux, ce qui donne 0100 et 1101. Ces deux mots binaires sont représentés en base seize par les chiffres 4 et D ⁵ et donc l'octet 01001101 est représenté par $4D$.

2.3 Propriétés d'une représentation en base β

Considérons un entier x strictement positif. Quand on l'écrit en base β , on a $x = a_n a_{n-1} \dots a_1 a_0$. Le nombre n est défini de manière unique si on impose que $a_n \neq 0$. $n + 1$ est le nombre de chiffres nécessaire pour écrire x dans la base β . On pose $N_\beta(x) = n + 1$. Le théorème suivant nous donne exactement la valeur de $N_\beta(x)$ en fonction de x .

$N_\beta(x)$ est le plus petit entier strictement supérieur à $\log_\beta(x)$.

⁴ pour un être humain !

⁵ En base 16, les lettres A, B, C, D, E et F sont des chiffres qui correspondent respectivement aux nombres dix, onze, douze, treize, quatorze et quinze.

On rappelle que $\log_\beta(x)$ est le logarithme de base β du nombre x . Le logarithme de base β peut se calculer à partir du logarithme népérien (noté $\ln$) grâce à la formule suivante :

$$\log_\beta(x) = \frac{\ln(x)}{\ln(\beta)}$$

On considère $x = 1503_{10}$. On a $\log_2(x) = 10,6$. Par conséquent, il faut 11 chiffres pour représenter x en base deux. Ce résultat est bien vérifié car $x = 1011101111_2$ d'après l'exemple [ex:1503].

Le théorème [th:log] montre que la quantité de chiffres nécessaires pour représenter un nombre entier dans un système de numération est à peu près proportionnel au logarithme de ce nombre. Comme le logarithme d'un nombre croît beaucoup moins vite que le nombre lui-même, ceci explique la supériorité des systèmes de numérations en base β sur la représentation par bâton aussi appelé système unaire ou système monadique⁶ qui nécessite autant de symboles que la valeur du nombre. Plus une base est petite, plus il faut de chiffres pour écrire les nombres entiers dans cette base. Le théorème suivant précise cette idée.

Le rapport $\frac{N_\beta(x)}{N_{\beta'}(x)}$ tends vers $\frac{\ln(\beta')}{\ln(\beta)}$ quand x tend vers l'infini.

D'après le théorème [th:log], il existe deux nombres ϵ et ϵ' avec $0 < \epsilon, \epsilon' \leq 1$ tels que : $N_\beta(x) = \frac{\ln(x)}{\ln(\beta)} + \epsilon$ et $N_{\beta'}(x) = \frac{\ln(x)}{\ln(\beta')} + \epsilon'$. On en déduit que :

$$\frac{N_\beta(x)}{N_{\beta'}(x)} = \frac{(\ln(x) + \epsilon \ln(\beta)) \ln(\beta')}{(\ln(x) + \epsilon' \ln(\beta')) \ln(\beta)}$$

Par conséquent, on a :

$$\frac{\ln(x) \ln(\beta')}{(\ln(x) + \ln(\beta')) \ln(\beta)} \leq \frac{N_\beta(x)}{N_{\beta'}(x)} \leq \frac{(\ln(x) + \ln(\beta)) \ln(\beta')}{\ln(x) \ln(\beta)}$$

On a donc $\frac{N_\beta(x)}{N_{\beta'}(x)}$ qui tend vers $\frac{\ln(\beta')}{\ln(\beta)}$ quand x tend vers l'infini car $\frac{\ln(x) + \ln(\beta)}{\ln(x)}$ et $\frac{\ln(x)}{\ln(x) + \ln(\beta')}$ tendent vers 1 quand x tend vers l'infini.

Pour écrire un nombre en base deux il faut environ 3,32 fois plus de chiffres qu'en base dix puisque $\frac{\ln(10)}{\ln(2)} = 3,32 \dots$

De même, puisque $\frac{\ln(16)}{\ln(10)} = 1,20 \dots$, il faut environ 20% de chiffres en plus pour écrire un nombre en base dix qu'en base seize.

2.4 Calcul en binaire

Les opérations classiques sur des représentations binaires de nombres entiers comme l'addition ou la multiplication suivent à peu près les mêmes règles que pour la base décimale. Néanmoins, il existe des différences. Nous allons décrire les méthodes de calcul en binaire pour les quatre opérations de base : addition, soustraction, multiplication et division.

⁶ Le nombre 25_{10} devient ||||| quand on le représente par bâtons.

2.4.1 Addition en binaire

Une addition en binaire se pose exactement comme une addition en décimal. En fait, l'addition en binaire est beaucoup plus simple car il y a moins de possibilités à considérer. En effet, si on se restreint à l'addition de deux nombres il n'y a que 6 cas à considérer lorsqu'on additionne deux chiffres :

- les deux chiffres sont égaux à 0 et il n'y a pas de retenue :
Dans ce cas le résultat est 0. On écrit donc 0 et il n'y a pas de retenue pour le prochain chiffre.
- **Un des deux chiffres est 1 et l'autre est 0 et il n'y a pas de retenue** Dans ce cas le résultat est 1. On écrit donc 1 et il n'y a pas de retenue pour le prochain chiffre.
- les deux chiffres sont égaux à 1 et il n'y a pas de retenue :
Dans ce cas le résultat est $2_{10} = 10_2$. On écrit donc 0 et on pose une retenue pour le prochain chiffre.
- les deux chiffres sont égaux à 0 et il y a une retenue :
Dans ce cas le résultat est 1. On écrit donc 1 et il n'y a pas de retenue pour le prochain chiffre.
- Un des deux chiffres est 1 et l'autre est 0 et il y a une retenue :
Dans ce cas le résultat est $2_{10} = 10_2$. On écrit donc 0 et on pose une retenue pour le prochain chiffre.
- les deux chiffres sont égaux à 1 et il y a une retenue :
Dans ce cas le résultat est $3_{10} = 11_2$. On écrit donc 1 et on pose une retenue pour le prochain chiffre.

Addition de 110101100 et 101101010.

$$\begin{array}{r}
 1 1 1 1 0 0 0 \\
 1 1 0 1 1 0 0 \\
 + 1 0 1 1 1 1 0 \\
 \hline
 1 1 0 0 1 1 1 0
 \end{array}$$

2.4.2 Soustraction en binaire

Pour la soustraction, le principe est le même qu'en base 10. Il n'y a 8 cas à considérer lorsqu'on soustrait deux chiffres

- $0 - 0$ sans retenue :
Dans ce cas le résultat est 0. On écrit donc 0 et il n'y a pas de retenue pour le prochain chiffre.
- $1 - 0$ sans retenue :
Dans ce cas le résultat est 1. On écrit donc 1 et il n'y a pas de retenue pour le prochain chiffre.
- $1 - 1$ sans retenue :
Dans ce cas le résultat est 0. On écrit donc 0 et il n'y a pas de retenue pour le prochain chiffre.
- $0 - 1$ sans retenue :
Dans ce cas on met une retenue sur le prochain chiffre afin d'effectuer le calcul $10 - 1$ qui donne 1. On écrit donc 1 et on met une retenue pour le prochain chiffre.
- $0 - 0$ avec retenue :
Dans ce cas on met une retenue sur le prochain chiffre afin d'effectuer le calcul $10 - 0 - 1$ qui donne 1. On écrit donc 1 et on met une retenue pour le prochain chiffre.
- $1 - 0$ avec retenue :

Dans ce cas le résultat est $1 - 0 - 1 = 0$. On écrit donc 0 et il n'y a pas de retenue pour le prochain chiffre.

— 1 - 1 avec retenue :

Dans ce cas on met une retenue sur le prochain chiffre afin d'effectuer le calcul $11 - 1 - 1$ qui donne 1. On écrit donc 1 et on met une retenue pour le prochain chiffre.

— 0 - 1 avec retenue :

Dans ce cas on met une retenue sur le prochain chiffre afin d'effectuer le calcul $10 - 1 - 1$ qui donne 0. On écrit donc 0 et on met une retenue pour le prochain chiffre.

Soustraction de 110001001 et 11010100.

$$\begin{array}{r}
 \begin{array}{ccccccccc}
 & -1 & & -1 & & -1 & & -1 & & -1 \\
 & 1 & & 1 & & 0 & & 0 & & 0 & & 1 & & 0 & & 0 & & 1 \\
 + & 0 & & 1 & & 1 & & 0 & & 1 & & 0 & & 1 & & 0 & & 0 \\
 \hline
 & 0 & & 1 & & 0 & & 1 & & 1 & & 0 & & 1 & & 0 & & 1
 \end{array}
 \end{array}$$

2.4.3 Multiplication en binaire

La multiplication utilise aussi les mêmes principes qu'en décimal. On pose la multiplication en mettant le plus grand nombre en premier. Ensuite on va multiplier ce nombre par chaque chiffre du deuxième nombre. Si le chiffre est 1 on recopie le premier nombre avec le décalage approprié (égal à la position du nombre dans le deuxième nombre) et si le chiffre est 0, on a rien à faire. Ensuite, il suffit d'additionner les nombres ainsi obtenus pour avoir le résultat.

Multiplication de 110001001 par 10101.

$$\begin{array}{r}
 110001001 \\
 \times 10101 \\
 \hline
 110001001 \\
 +11000100100 \\
 +1100010010000 \\
 \hline
 10000000111101
 \end{array}$$

Le résultat de la multiplication de 110001001 par 10101 est 10000000111101.

2.4.4 Division en binaire

La division utilise aussi les mêmes principes qu'en décimal. On utilise la méthode de division par colonne. En binaire, cette technique est simple car à chaque fois on multiplie dénominateur par un nombre de la forme $10 \dots 0$. Il faut donc à chaque étape trouver le décalage maximal (en ajoutant des 0) que l'on peut appliquer au dénominateur de telle sorte qu'il reste plus petit que le numérateur et ensuite retrancher cette valeur au numérateur. On répète le processus avec le résultat obtenu comme nouveau numérateur.

Division de 10111011111 par 1010.

$$\begin{array}{r}
1011101111 \mid 1010 \\
- 1010 \\
\hline
0001101111 \\
- 1010 \\
\hline
00111111 \\
- 1010 \\
\hline
010111 \\
- 1010 \\
\hline
00011
\end{array}$$

FIGURE 5 – image

On obtient pour quotient 10010110 et pour reste 11.

2.5 Application de l'écriture binaire d'un nombre

2.5.1 Exponentiation rapide

On dispose d'une machine sachant réaliser des multiplications et on souhaite l'utiliser pour calculer la puissance d'un nombre. Pour calculer x^y avec x entier, on peut successivement calculer les valeurs $x^2, x^3, x^4, \dots, x^{y-1}$ et x^y en multipliant à chaque fois le résultat précédent par x . Cette méthode a donc recours à $y - 1$ multiplications. On peut se demander s'il on peut obtenir le même résultat en effectuant moins d'opérations. C'est possible en utilisant une méthode appelée *exponentiation rapide*. Cette méthode repose sur la représentation en base 2 de l'entier y . On a :

$$y = (y_n y_{n-1} y_{n-2} \dots y_2 y_1 y_0)_2 = y_n 2^n + y_{n-1} 2^{n-1} + y_{n-2} 2^{n-2} + \dots + y_2 2^2 + y_1 2^1 + y_0 2^0$$

On peut donc écrire :

$$\begin{aligned}
x^y &= x^{y_n 2^n + y_{n-1} 2^{n-1} + \dots + y_1 2^1 + y_0 2^0} \\
&= (x^{2^n})^{y_n} \times (x^{2^{n-1}})^{y_{n-1}} \times \dots \times (x^{2^1})^{y_1} \times (x^{2^0})^{y_0}
\end{aligned}$$

Puisqu'on a écrit y en base 2, les chiffres y_i sont égaux à 0 ou 1. On a donc pour tout i tel que $0 \leq i \leq n$:

$$(x^{2^i}) = \begin{cases} x^{2^i} & \text{si } y_i = 1 \\ 1 & \text{si } y_i = 0 \end{cases}$$

Pour calculer x^y , il suffit donc de calculer successivement les nombres :

$$x^2 = x \times x \quad x^4 = x^2 \times x^2 \quad \dots \quad x^{2^n} = (x^{2^{n-1}}) \times (x^{2^{n-1}})$$

et ensuite de multiplier entre eux les nombres x^{2^i} tel que $y_i = 1$.

On souhaite calculer x^{1503} . On convertit donc 1503 en base 2, ce qui donne : $(1503)_{10} = 10111011111$ d'où :

$$(1503)_{10} = 1 + 2 + 4 + 8 + 16 + 64 + 128 + 256 + 1024$$

Ensuite on calcule par récurrence $x^2, x^4, x^8, \dots, x^{1024}$ et enfin on effectue le produit :

$$x^{1503} = x^1 x^2 x^4 x^8 x^{16} x^{64} x^{128} x^{256} x^{1024}$$

Par cette méthode il ne faut que 18 multiplications pour obtenir le résultat : 10 pour calculer les valeurs $x^2, x^4, x^8, \dots, x^{1024}$ et 8 pour multiplier les 9 facteurs.

De manière générale, si $y = (y_n y_{n-1} y_{n-2} \dots y_2 y_1 y_0)_2$ et m est égal au nombre de chiffres égaux à 1 dans cette représentation en base deux alors le calcul de x^y utilise n multiplication pour le calcul des valeurs $x^2, x^4, x^8, \dots, x^{2^n}$ et $m - 1$ multiplication pour le calcul final. Puisque $m \leq n$, le calcul utilise donc au plus $2n$ multiplications. En utilisant le Théorème [th:log], on peut donc calculer x^y en moins de $2\lceil \log_2(y) \rceil$ multiplications puisque $n \leq \lceil \log_2(y) \rceil$. Cette méthode est bien plus efficace. L'exemple suivant montre que le gain est très important si y est grand.

On souhaite calculer $3^{1000000001}$ avec une machine pouvant réaliser une multiplication par milliseconde. En utilisant la méthode naïve, on a besoin de faire 1000000000 multiplications et donc 1000000000 millisecondes ce qui représente un peu plus de 11 jours et demi. Avec l'exponentiation rapide, on a seulement besoin de $2\lceil \log_2(1000000001) \rceil = 60$ millisecondes.

2.5.2 Algorithme d'Euclide en base 2

L'algorithme de division euclidienne permet de calculer le PGCD (Plus Grand Commun Diviseur). Il existe une variante de cet algorithme (décrit par Donald Knuth dans son ouvrage The Art of Computer Programming, volume 3) qui est particulièrement adaptée aux nombres écrits en base 2 et donc au calcul du PGCD par un ordinateur. Cette méthode ne requiert que des soustractions, des multiplications par deux et des divisions par 2. Les multiplications et divisions par deux sont des opérations très simples en base deux car elles consistent respectivement à rajouter un zéro à la fin de la représentation du nombre ou bien à l'enlever. Cette variante de l'algorithme d'Euclide repose sur quatre propriétés du PGCD et des nombres représentés en base deux :

- Un entier dont la représentation en base 2 se termine par exactement k chiffres zéro est de la forme $a \times 2^k$ avec a impair. Par exemple, $24 = (11000)_2$ est égal à 3×2^3 .
- si a et b sont impairs alors $\text{pgcd}(a \times 2^p, b \times 2^q) = \text{pgcd}(a, b) \times 2^r$ avec $r = \min\{p, q\}$. En particulier, pour $q = 0$ on a $\text{pgcd}(a \times 2^p, b) = \text{pgcd}(a, b)$. Par exemple, on a $\text{pgcd}(24, 16) = \text{pgcd}(3 \times 2^3, 1 \times 2^4) = \text{pgcd}(3, 1) \times 2^3 = 8$.
- Si on a $a > b$ alors $\text{pgcd}(a, b) = \text{pgcd}(a - b, b)$.
- On a $\text{pgcd}(a, a) = a$.

2.5.3 Méthode de calcul du $pgcd(a, b)$ en base deux

Étape 0 : On compte le plus petit nombre r de zéros à la fin entre les représentations de a et b . On enlève ce nombre de zéros des représentations de deux nombres. On retient que le résultat obtenu par la suite devra être multiplié par 2^r .

Étape 1 : Si l'un des deux nombres est pair (se terminant par au moins une zéro), on enlève tous les zéros à la fin de sa représentation.

Étape 2 : Si $a = b$ alors $pgcd(a, b) = a$. sinon on passe à la phase suivante.

Étape 3 : Si $a > b$ alors on recommence l'étape 1 en remplaçant a par $a - b$. Sinon on recommence à l'étape 1 en remplaçant b par $b - a$.

Calcul de $x = pgcd(2352, 2568)$

On convertit les deux nombre en base 2. On obtient $a = 2352 = (100100110000)_2$ et $b = 2568 = (101000001000)_2$.

Étape 0 : puisque a termine par 4 zéros et b termine par 3 zéros, on enlève 3 zéros à chacun et on obtient que $x = (1000)_2 \times y = 8y$ et $y = pgcd(100100110, 101000001)$.

Étape 1 : On enlève un zéro au premier nombre qui est pair et on obtient $y = pgcd(10010011, 101000001)$.

Étape 2 : les deux nombres sont différents. On continue donc avec l'étape 3.

Étape 3 : $y = pgcd(10010011, 101000001 - 10010011) = pgcd(10010011, 10101110)$

Étape 1 : $y = pgcd(10010011, 1010111)$

Étape 3 : $y = pgcd(10010011 - 1010111, 1010111) = pgcd(111100, 1010111)$

Étape 1 : $y = pgcd(1111, 1010111)$

Étape 3 : $y = pgcd(1111, 1010111 - 1111) = pgcd(1111, 1001000)$

Étape 1 : $y = pgcd(1111, 1001)$

Étape 3 : $y = pgcd(1111 - 1001, 1001) = pgcd(110, 1001)$

Étape 1 : $y = pgcd(11, 1001)$

Étape 3 : $y = pgcd(11, 1001 - 11) = pgcd(11, 110)$

Étape 1 : $y = pgcd(11, 11)$

Étape 2 : $y = (11)_2 = (3)_{10}$

On a donc $pgcd(2352, 2568) = x = 8y = 8 \times 3 = 24$.

3 Codage d'information dans un ordinateur

3.1 Généralités

Un ordinateur peut être défini comme une machine permettant le traitement automatique d'informations. Afin que l'ordinateur puisse traiter de l'information (images, paroles, textes, nombres), il faut d'abord la convertir dans un format qui lui est compréhensible. Les ordinateurs (passés et présents) fonctionnent quasiment tous avec le système binaire. L'information doit donc toujours être codée sous forme de 0 et de 1. Afin de pouvoir transformer l'information, il faut définir une méthode de transformation appelée *codage*. Le codage peut se faire en plusieurs étapes et avoir des utilités différentes. Les types de codage les plus courants sont :

- **Numérisation** : La numérisation permet de construire une représentation discrète d'un objet du monde réel. Cette étape est indispensable afin qu'un ordinateur puisse traiter l'information. Par exemple, un appareil photo numérique va transformer une vue en un fichier binaire (et donc ne contenant que des 0 et des 1). La résolution de l'image, c'est-à-dire son nombre de points (ou pixels) est finie. Pour chaque point, une couleur est attribuée parmi une palette finie (jusqu'à 16 millions dans la plupart des cas). Il y a souvent perte d'information lors de la numérisation car on est obligé de coder des informations très complexes en une représentation finie et compacte.
- **Compression** : La compression de données consiste à transformer une suite de bits en une suite de bits plus courte, contenant les mêmes (ou presque les mêmes) informations. La compression est essentielle pour l'informatique actuelle, car souvent l'information non-compressée prendrait trop de place. Un exemple classique de données compressées est la vidéo. Un flux vidéo non-compressé prend beaucoup plus de place qu'un flux compressé. Un tel flux ne pourrait être stocké avec les moyens de stockage actuels. La plupart des formats de fichier de vidéo (avi, mp4, ...) utilisent un algorithme de compression. Il est à noter qu'il existe deux types de compression : la compression sans pertes dans laquelle l'information reste la même après compression et la compression avec pertes dans laquelle une partie de l'information est perdue. La compression avec pertes ne s'applique qu'aux données perceptibles, en général sonores ou visuelles, qui peuvent subir une modification, parfois importante, sans que cela ne soit perceptible par un humain.
- **Détection/correction d'erreurs** : Un code détecteur/correcteur est une technique de codage basée sur la redondance. Elle est destinée à corriger les erreurs potentielles lors d'une transmission d'information sur une voie de communication peu fiable. Les supports physiques étant sujets à des altérations souvent imprévisibles, il est souvent impossible de garantir la fiabilité d'une transmission. Par exemple les communications radios peuvent être perturbées par d'autres communications, les disques compacts rayés, ... Il est donc souvent nécessaire de soit détecter les erreurs pour demander un renvoi de l'information (code détecteur par exemple utilisés dans le protocole TCP/IP des réseaux d'ordinateurs utilisé pour Internet) ou bien de les corriger, c'est-à-dire de reconstruire l'information manquante grâce à la redondance (technique utilisée dans les disques compacts).
- **Chiffrement** : Le chiffrement est un procédé grâce auquel on souhaite rendre la compréhension d'un document impossible à toute personne qui n'a pas la clé de (dé)chiffrement. Le chiffrement est très utilisé sur internet. En effet, les données sensibles passant par internet (mots de passe, coordonnées bancaires) doivent être chiffrées afin de ne pas pouvoir être interceptées par une tierce personne.

Il est à noter que ces quatre techniques sont souvent combinées dans une utilisation courante. Par exemple

le format bluray pour la vidéo implique le plus souvent l'utilisation combinée de ces quatre techniques. Il y a d'abord numérisation afin d'obtenir le flux vidéo non-compressé. Ensuite, il y a la compression du flux vidéo (le plus souvent via la norme MPEG-4). Le disque utilise des codes correcteurs afin de corriger les erreurs liées au support (impuretés sur le disque et imprécision du capteur). Finalement le flux vidéo entre le lecteur et la télévision peut être crypté (via le protocole HDCP) afin d'éviter l'écoute du flux par un appareil tierce et donc le piratage.

3.2 Code détecteurs/correcteurs d'erreur

Dans cette section, nous allons parler plus en détail des codes détecteurs/correcteurs d'erreurs. En informatique, la transmission des informations se fait habituellement sous forme de 0 et de 1 qui sont transmis sous forme de courant électrique (câble réseau), d'ondes radios (wifi) ou bien de signaux lumineux (fibre optique). Malheureusement, quand une transmission se prolonge, des erreurs finissent toujours pas se produire : des 0 sont remplacés par des 1 ou inversement. Si l'on transmet un bit toutes les microsecondes, un million de bits sont expédiés en une seconde et donc même si le risque d'erreur est faible, au bout de quelques minutes beaucoup de bits sont transmis de manière erronée. Il est donc indispensable de se prémunir contre ces erreurs de transmission.

Une manière de se prémunir contre les erreurs de transmission est d'utiliser des codes détecteurs d'erreurs. Généralement, la technique du codage par blocs est utilisée pour ces codes. Le principe est de grouper les bits à envoyer par bloc (par exemple par octet, c'est-à-dire un groupe de 8 bits). Pour l'expéditeur, cette technique consiste à :

1. À découper les bits à transmettre en *blocs* ayant tous une certaine longueur (noté k) fixée à l'avance.
2. À coder chaque bloc en lui ajoutant un nombre fixé (noté r) de *bits de contrôles* qui dépendent du bloc, afin d'obtenir un *mot de code* (de longueur $n = k + r$).
3. À transmettre le mot de code.

Pour le receveur, cette technique consiste à :

1. À vérifier que le mot binaire reçu est bien un mot du code (c'est la détection).
2. À extraire le bloc (les bits d'informations) du mot de code si le mot est accepté ou bien demander le renvoi du mot de code dans le cas contraire.

Le rapport $\tau = r/n$ qui mesure, parmi les bits transmis (taille du mot de code n) la proportion de ceux réellement porteur d'information (taille du bloc originel r), s'appelle le *rendement* ou le *taux* du code. Bien évidemment, on a tout intérêt à ce que le rendement soit le plus élevé possible, car plus il est élevé et plus on transmettra d'information en un temps donné. En quelque sorte, le bloc constitue la marchandise à envoyer, et les bits de contrôle sont l'emballage qui la protège. Plus l'emballage est résistant, plus le paquet est lourd et plus le prix à payer est élevé. Les bons codages sont ceux qui trouvent le bon compromis entre sécurité et coût.

L'un des codes détecteurs d'erreur les plus simples est connu sous le nom de test de parité. Pour ce code, on choisit tout d'abord la taille des blocs. Pour notre exemple, on va prendre des blocs de 3 bits ($k = 3$). À chaque bloc, on rajoute un bit de parité qui correspond au bit de contrôle ($r = 1$). Le bit de contrôle sera choisi de telle

sorte que le mot de code ait toujours un nombre pair de 1. Par exemple, si le bloc de départ est 001, on rajoute un 1 pour obtenir le mot de code 0011. Si le bloc de départ est 011, on rajoute un 0 pour obtenir le mot de code 0110. Le code est défini par la table suivante :

TABLE 1: Table de codage pour le codage par bit de parité

bloc de départ	mot de code
000	0000
001	0011
010	0101
011	0110
100	1001
101	1010
110	1100
111	1111

Le rendement du code est de $3/4$. La détection pour ce code est très simple. Le receveur compte le nombre de 1. Si ce nombre est impair (par exemple il reçoit le mot $R = 1011$), il a la certitude qu'au moins une erreur a été commise. Malheureusement, il ne sait pas combien exactement d'erreurs ont été commises ni leur emplacement. Il lui est donc impossible de les corriger. Si le receveur reçoit $R' = 1111$, le message est accepté car il fait partie du code. Bien que le message semble correct, il n'y a aucune certitude qu'il n'y ait pas eu d'erreurs. En effet, Il se peut que l'expéditeur ait envoyé le mot $E' = 0011$ et que deux erreurs se soient produites sur les deux premiers bits. Dans ce cas, le mot reçu est aussi R' . Dans le doute, il fait comme si le message était bon et il le décode pour obtenir l'information 001. En pratique, cette hypothèse est le plus souvent correcte. En effet, la probabilité qu'une erreur se produise est souvent très faible, ce qui fait qu'il est bien plus probable qu'aucune erreur se soit commise. Il est possible d'évaluer exactement la probabilité qu'un message faux soit accepté par le receveur. Si la probabilité d'erreur est d'un sur mille (en moyenne, un bit transmis sur mille est transmis avec erreur), alors parmi les messages faux, il y en a seulement 0,15% qui ne sont pas détectés. Si on transmet les bits sans codage, la proportion de bits faux non-détectés est de 100%.

Parfois, il n'est pas suffisant de détecter les erreurs, il faut aussi être capable de les corriger. Les raisons à cela peuvent être multiples. Cela peut être des problèmes de support : une impureté sur un CD fera que certains bits sont toujours mal lus quelque soit le nombre de lecture sur ces bits. Il y a aussi parfois des contraintes temporelles pour le flux d'information qui font que demander le renvoi n'est pas envisageable. Par exemple pour la téléphonie mobile, demander le renvoi d'information peut prendre trop de temps et créer un silence dans la transmission. Pour ces raisons, des codes correcteurs d'erreurs sont souvent utilisés. Le principe des codes correcteurs d'erreur est similaire à celui des codes détecteurs. La seule différence réside dans le fait que la redondance d'information est plus grande et donc le receveur peut non seulement détecter les erreurs mais aussi les corriger.

L'un des codes correcteurs d'erreur les plus simples est connu sous le nom de codage par répétition. Pour ce code, on choisit des blocs d'un bit ($k = 1$). A chaque bloc et donc à chaque bit, on rajoute deux bits identiques

($r = 2$). Par exemple, si le bloc de départ est 0, on rajoute deux 0 pour obtenir le mot de code 000. Si le bloc de départ est 1, on rajoute deux 1 pour obtenir le mot de code 111. Le rendement du code est de $1/3$. Pour reconstituer l'information, le receveur va supposer qu'au plus une erreur de transmission s'est produite sur le mot de code. Si le mot reçu contient au moins deux 1, il va considérer que le bit envoyé est 1 et 0 si la majorité des bits sont des 0. Le décodage est défini par la table suivante :

TABLE 2: Table de décodage pour le codage par répétition

mot de code	bloc après décodage
000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

Si au plus une erreur s'est produite lors de l'envoi du code, alors le receveur va décoder la bonne information car la majorité des bits reçus est correcte. Par contre, si deux erreurs se sont produites alors, le receveur va décoder la mauvaise information. En pratique, la probabilité qu'une erreur se produise est souvent très faible et donc il y a peu de chance que deux bits sur trois soient transmis de manière erronée. Il est possible d'évaluer exactement la probabilité qu'un message reçu soit décodé de manière erronée par le receveur. Lorsque la probabilité p qu'un bit soit transmis de manière erronée est proche de zéro, la probabilité qu'un message soit mal décodé est proche de p .

3.3 Codage des nombres

Nombres entiers

Le codage des entiers naturels ($\mathbb{N}$) dans un ordinateur utilise tout simplement la représentation binaire du nombre. Il existe deux formats courants pour les entiers dans les machines actuelles : "short" sur 32 bits et "long" sur 64 bits. Les machines plus anciennes pouvaient aussi coder les entiers sur 8 ou 16 bits. Pour tous ces formats, on rajoute des 0 devant la représentation en base deux du nombre afin d'obtenir la longueur voulue (8, 16, 32 ou 64). Le format "short" permet d'encoder les entiers de 0 à 4 294 967 295 ($2^{32} - 1$) alors que le format "long" permet d'encoder les entiers de 0 à environ 18×10^{18} ($2^{64} - 1$). Dans tous les cas, il existe des entiers qui sont trop grands pour être représentés. C'est malheureusement inévitable car un ordinateur (réel) a une mémoire finie et par conséquent, l'infinité des entiers ne pourra jamais être encodée dans un tel ordinateur.

L'entier 1503_{10} a pour code en 16 bits le mot : 0000 0101 1101 1111. L'entier 32767_{10} a pour code en 16 bits le mot : 0111 1111 1111 1111.

Pour les entiers signés (appartenant à $\mathbb{Z}$), il existe deux façons classiques de les encoder. La première consiste

tout simplement à réserver un bit pour coder le signe de l'entier. Le premier bit du mot binaire codant l'entier donne le signe : positif si le bit est 0 et négatif s'il est à 1.

Le format "short" signé permet d'encoder les entiers de $-2^{31} - 1$ à $2^{31} - 1$ alors que le format "long" signé permet d'encoder les entiers d'environ 9×10^{18} ($-2^{63} - 1$) à environ 9×10^{18} ($2^{63} - 1$). Il est à noter que le nombre 0 a deux représentations dans ce système (une négative et une positive).

L'entier 1503_{10} a pour représentation en 16 bits le mot : 0000 0101 1101 1111. L'entier -1503_{10} a pour représentation en 16 bits le mot : 1000 0101 1101 1111. L'entier 0 a deux représentations possibles en 16 bits le mot : 1000 0000 0000 0000 et le mot 0000 0000 0000 0000.

Une autre méthode de représentation des entiers est le complément à deux. Avec cette méthode, le signe est toujours indiqué par le premier bit (positif s'il est à 0 et négatif s'il est à 1) mais la valeur absolue est codée de manière différente. Afin de calculer la représentation d'un nombre dans ce système, on procède en deux étapes :

1. On exprime la valeur absolue du nombre en base 2 (en rajoutant éventuellement des 0 pour obtenir la longueur souhaitée : 8, 16, 32 ou 64)
2. Si le nombre est négatif, on inverse tous les bits, c'est-à-dire que les 1 deviennent des 0 et inversement. Puis, on ajoute une unité au résultat obtenu.

Dans cette représentation, il n'y a qu'une seule façon de coder le 0 (avec un mot ne contenant que des 0). La plage des nombres représentable dans ce système est quasiment la même que pour la précédente méthode. On peut juste coder un entier négatif de plus. Par exemple, -32768 peut être représenté en 16 bits avec ce système alors que ce n'était pas le cas précédemment.

On considère une représentation sur 16 bits. Pour -3 on obtient le mot 0000 0000 0000 0011 à la première étape. En inversant les bits on : 1111 1111 1111 1100. L'ajout d'une unité donne finalement 1111 1111 1111 1101. On utilise la même méthode pour obtenir la représentation des nombres suivants :

0	=	0000000000000000
1	=	0000000000000001
-1	=	1111111111111111
3	=	0000000000000011
32767	=	0111111111111111
-32767	=	1000000000000001
-32768	=	1000000000000000

On peut se demander quel est l'avantage de cette représentation sur la précédente. En fait, cette représentation permet de calculer facilement la représentation de la somme de deux nombres. Pour faire la somme de deux nombres, il suffit de les sommer de manière normale (somme en base deux) puis ensuite de tronquer le résultat sur le nombre de bits de la représentation en enlevant éventuellement un 1 à gauche.

Pour faire l'addition de 3 et -1 sur 16 bits, on pose :

$$\begin{array}{r} 0000000000000011 \quad 3_{10} \\ +1111111111111111 \quad +(-1_{10}) \\ \hline 1000000000000010 \quad 2_{10} \end{array}$$

Ce qui nous donne $0000000000000010 = 2_{10}$ après avoir tronqué le 1.

Nombres réels

Il est parfois utile de pouvoir représenter des nombres réels. Pour cela, on a besoin de définir une méthode pour coder les chiffres après la virgule. Il existe deux solutions classiques, selon que le nombre de chiffres situés après la virgule est ou non fixé :

- Virgule fixe
- Virgule flottante

Fixer le nombre de chiffres après la virgule a plusieurs avantages. Cela permet de rendre les opérations plus simples et donc de permettre des processeurs moins onéreux. Le codage est constitué de deux parties :

- la partie entière codée en binaire (complément à deux).
- la partie décimale dans laquelle chaque bit correspond à l'inverse d'une puissance de 2.

On cherche à calculer la représentation en virgule fixe de $-3,625_{10}$ avec 16 bits pour coder la partie entière et 16 bits pour coder la partie fractionnaire. On a :

$$-3,625_{10} = \underbrace{1111 \ 1111 \ 1111 \ 1101}_{-3} \quad \underbrace{1010 \ 0000 \ 0000 \ 0000}_{0,625 = 0,5 + 0,125 = \frac{1}{2} + \frac{0}{4} + \frac{1}{8}}$$

Il existe une méthode générale pour calculer la représentation de la partie décimale d'un nombre.

Méthode générale de calcul de la partie fractionnaire d'un nombre

On part de la partie x fractionnaire d'un nombre. Pour calculer chaque chiffre en partant de la gauche, on :

1. multiplie x par deux ($x \leftarrow 2 \times x$). La valeur du bit est déterminée par la nouvelle valeur de x .
 - Si $x < 1$ le bit est égal à 0.
 - Si $x \geq 1$, le bit est égal à 1 et on retranche 1 à x ($x \leftarrow x - 1$).
2. On répète le processus avec la nouvelle valeur de x pour calculer le bit suivant. On arrête le calcul si x devient égal à 0 ou si on a calculé assez de bits pour la représentation.

Le calcul de la représentation de 0,347 sur 10 bits donne :

$0,347 \times 2 = 0,694 < 1 \implies$ on pose 0: $0,347 = 0,0$
 $0,694 \times 2 = 1,388 > 1 \implies$ on pose 1: $0,347 = 0,01$
 $0,388 \times 2 = 0,766 < 1 \implies$ on pose 0: $0,347 = 0,010$
 $0,766 \times 2 = 1,552 > 1 \implies$ on pose 1: $0,347 = 0,0101$
 $0,552 \times 2 = 1,104 > 1 \implies$ on pose 1: $0,347 = 0,01011$
 $0,104 \times 2 = 0,208 < 1 \implies$ on pose 0: $0,347 = 0,010110$
 $0,208 \times 2 = 0,416 < 1 \implies$ on pose 0: $0,347 = 0,0101100$
 $0,416 \times 2 = 0,832 < 1 \implies$ on pose 0: $0,347 = 0,01011000$
 $0,832 \times 2 = 1,664 > 1 \implies$ on pose 1: $0,347 = 0,010110001$
 $0,664 \times 2 = 1,328 > 1 \implies$ on pose 1: $0,347 = 0,0101100011$

Malgré ses avantages, cette représentation n'est pas souvent utilisée. Le plus grand problème de cette représentation est le gaspillage de précision que l'on a sur les petits nombres. En effet dans ce cas les chiffres à gauche correspondant à la partie entière sont gaspillés car non utilisés. De plus, cette représentation ne permet de représenter que des nombres de $2^{n-1} - \frac{1}{2^d}$ à $-2^{n-1} - 1 + \frac{1}{2^d}$ avec n le nombre de bits pour la partie entière et d celui de la partie fractionnaire. On peut aussi remarquer que cette représentation ne peut pas représenter de nombre plus petit que $\frac{1}{2^d}$. Pour toutes ces raisons, le codage par virgule flottante, bien que plus difficile, est souvent préféré.

Il existe essentiellement deux formats de représentation en virgule flottante qui sont utilisés dans les machines actuelles : une version 32 bits et une version 64 bits. Dans les deux cas, on utilise une représentation similaire à la représentation scientifique en base 10. On représente un nombre avec un signe s égal à 1 ou -1 , une mantisse m comprise entre 1 et 2, et un exposant e . La valeur représentée x est égal à $sm2^e$.

En base dix, on a :

$$-1540,412654 = -1,540412654 \times 10^3$$

En base deux, on a :

$$101110,001101 = 1,01110001101 \times 2^5$$

Le nombre de bits utilisés pour les deux formats sont les suivants :

	taille	bits pour s	bits pour e	bits pour m	valeur
Simple précision	32 bits	1	8	23	$-1^s m 2^{e-127}$
Double précision	64 bits	1	11	52	$-1^s m 2^{e-1023}$

On peut remarquer qu'en base 2, le chiffre de la mantisse à gauche de la virgule est toujours 1. Il est donc inutile de le coder. Par conséquent, la mantisse ne code que les chiffres à droite de la virgule.

- Binaire (valeur absolue) : 110,101
- Normalisation de la mantisse : 1,10101.2²
- Occupation des 23 bits de mantisse 10101000000000000000000
- Décalage de l'exposant (décalage de 127) donc exposant = (2 + 127)₁₀ = 10000001₂
- Signe = 1 car négatif

On représente donc 110,101 par :

$$\begin{array}{ccc}
 \textit{signe} & \textit{exposant} & \textit{mantisse} \\
 \boxed{1} & \boxed{10000001} & \boxed{10101000000000000000000}
 \end{array}$$

En plus des nombres classiques, la norme définit :

- l'infini (∞) : L'infini est codé avec un exposant composé que de "1" et une mantisse composée que de "0". On peut avoir $+\infty$ et $-\infty$ suivant le signe. Par exemple, $+\infty$ est représenté en 32 bits par :

$$\begin{array}{ccc}
 \textit{signe} & \textit{exposant} & \textit{mantisse} \\
 \boxed{0} & \boxed{11111111} & \boxed{000000000000000000000000}
 \end{array}$$

- NaN (not a number) représente le résultat d'une opération qui n'est pas un nombre réel. Par exemple, la racine carré d'un nombre négatif qui est imaginaire donne pour résultat un NaN. Les NaN (de différents types) sont codés par un exposant composé que de "1" et par une mantisse contenant au moins un "1". Par exemple, on peut avoir la représentation suivant pour un NaN en 32 bits :

$$\begin{array}{ccc}
 \textit{signe} & \textit{exposant} & \textit{mantisse} \\
 \boxed{0} & \boxed{11111111} & \boxed{000000000000000000000001}
 \end{array}$$

3.4 Approximations et erreurs

La représentation des nombres dans les ordinateurs posent de nombreux problèmes. Comme on l'a vu précédemment, la représentation des entiers utilise la plupart du temps un nombre fixe de bits. Par conséquent, elle ne peut être utilisée que pour une plage bornée d'entiers. Sur 16 bits, on ne peut coder que les entiers de 0 à 65535. Malheureusement, il peut arriver que le résultat d'une opération sorte de la plage des entiers représentables. Par exemple, si on calcule $(39001 + 27446)_{10} = 66447_{10}$ sur 16 bits, on a :

$$\begin{array}{r}
 1001100001011001 \\
 + \quad 0110101100110110 \\
 \hline
 = 1 \underbrace{0000001110001111}_{= (911)_{10}}
 \end{array}$$

Dans la plupart des machines, le dépassement ne sera pas détecté et le bit supplémentaire sera simplement ignoré (ce qui nous donne $(911)_{10}$ dans l'exemple).

Pour les nombres réels, il y a des problèmes de dépassement mais aussi des problèmes d'approximations. La mantisse étant de taille bornée, certains nombres ne peuvent pas être représentés de manière exacte. Par exemple, en 64 bits (avec mantisse de 52 bits), le nombre $2^{60} + 1$ est approximé par 2^{60} car il faudrait une mantisse d'au moins 60 bits pour pouvoir le représenter de manière exacte. Une conséquence est que $(1 + 2^{60}) + (-2^{60})$ donnera 0. C'est d'autant plus grave que $1 + (2^{60} + (-2^{60}))$ donnera bien 1, ce qui fait que l'addition n'est plus associative.