IPT : Cours 2

La représentation informatique des nombres

(3 ou 4 heures)

MPSI-Cauchy : Prytanée National Militaire

Pascal Delahaye

27 septembre 2016

1 Codage en base 2

DÉFINITION 1 : Tout nombre positif s'écrit sous la forme :

$$x = n + f \quad \text{avec} \quad \begin{cases} n \in \mathbb{N} \\ f \in [0,\ 1[\end{cases}$$

- n est appelée la partie entière du nombre x
- f est appelée la partie fractionnaire du nombre x

1.1 Codage binaire de la partie entière

`Exemple 1.` Considérons le nombre entier $x = 241$

1. x se décompose de la façon suivante : $x = \mathbf{2}.10^2 + \mathbf{4}.10^1 + \mathbf{1}.10^0$.
   On dit alors que $x = 241$ est l'écriture décimale ou l'écriture en base 10 de x.

2. Plus généralement, si b et un entier supérieur ou égal à 2, en effectuant des divisions euclidiennes successives de x par b, il est possible d'écrire x sous la forme :

   $$x = \mathbf{a_p}.b^p + \mathbf{a_{p-1}}.b^{p-1} + \cdots + \mathbf{a_1}.b^1 + \mathbf{a_0}.b^0 \quad \text{avec} \quad (a_p,\ a_{p-1},\ \ldots,\ a_1,\ a_0) \in [\![0, b-1]\!]^{p+1}$$

   On dit alors que $x = \overline{a_p a_{p-1} \ldots a_1 a_0}^{(b)}$ est l'écriture en base b de x.

Remarque 1. Les bases les plus couramment utilisées sont : la base 10 (mathématiques courantes), la base 2 ou *base binaire* (informatique - codage dans la mémoire de l'ordinateur) et la base 16 dite *base hexadécimale* (informatique - codage logiciel).

PROPOSITION 1 : **Codage binaire de la partie entière**
Soit $n \in \mathbb{N}^*$.
Il existe $p \in \mathbb{N}$ et $(a_p,\ a_{p-1},\ \ldots,\ a_1,\ a_0) \in \{0,\ 1\}^{p+1}$ tel que :

$$n = \mathbf{a_p}.2^p + \mathbf{a_{p-1}}.2^{p-1} + \cdots + \mathbf{a_1}.2^1 + \mathbf{a_0}.2^0 \quad \text{avec} \quad a_p = 1$$

Cette décomposition est unique.
L'écrite de n sous la forme : $n = \overline{a_p a_{p-1} \ldots a_1 a_0}^{(2)}$ s'appelle l'écriture binaire (ou en base 2) de n.

Remarque 2. Nous pouvons obtenir la valeur de p grâce à la formule : $p = \lfloor \frac{\ln n}{\ln 2} \rfloor$.

Méthode de décomposition d'un entier en base 2

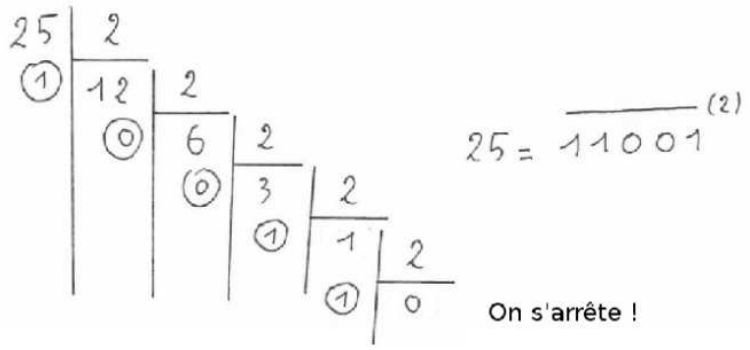
Les valeurs $a_0,\ a_1,\ \ldots,\ a_p$ s'obtiennent en effectuant les divisions euclidiennes des quotients successifs par 2.
Plus précisément, $\forall k \in [\![0, p]\!]$:

$$q_{-1} = n \quad \text{et} \quad \begin{cases} a_k \text{ est le reste} \\ q_k \text{ est le quotient} \end{cases} \quad \text{de la division euclidienne de } q_{k-1} \text{ par } 2$$

	Nombre	Quotient de la division par 2	Reste de la division par 2	
n=	25	12	1	a0
q0=	12	6	0	a1
q1=	6	3	0	a2
q2=	3	1	1	a3
q3=	1	0	1	a4

La décomposition de $x = 25$ en base 2 est donc : $x = \overline{11001}^{(2)}$.
Vérification : $1.2^4 + 1.2^3 + 1.2^0 = 16 + 8 + 1 = 25 = x$.

Décomposition à la main :



Algorithme de décomposition de n en base b :

```
# Initialisation des variables

  Q = n       # Q permet de stocker la suite des quotients
  L = []      # La liste L contiendra les coefficients de n en base b

# Boucle tant que "le quotient Q n'est pas nul"

  while Q != 0 :
        L.append(Q%b)    # on ajoute le reste à la liste
        Q = Q//b         # on calcule le nouveau quotient

# On inverse la liste et on affiche le résultat

  L.reverse()
  print(L)
```

Remarque 3. Quelques valeurs de référence :

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
1	2	4	8	16	32	64	128	256

Codage / décodage rapide

On peut utiliser le tableau précédent pour :

1. obtenir plus rapidement la décomposition en binaire d'un nombre.
   Exemple : $n = 14 = 8 + 6 = 8 + 4 + 2 = 1.2^3 + 1.2^2 + 1.2^1 + 0.2^0 = \overline{1110}^{(2)}$
2. retrouver l'écriture décimale d'un nombre donné en binaire.
   Exemple : $n = \overline{10101}^{(2)} = 1 + 4 + 16 = 21$

`Exemple 2.` $(*)$

1. Déterminer l'écriture binaire des nombres entiers suivants : $n_1 = 7$, $n_2 = 13$, $n_3 = 26$.
2. Déterminer les nombres entiers dont l'écriture binaire est : $n_1 = \overline{10101}^{(2)}$, $n_2 = \overline{1010}^{(2)}$, $n_3 = \overline{10011}^{(2)}$.
3. Combien de nombres entiers peut-on coder avec une écriture binaire comportant au plus p chiffres ? Quel est le plus grand ?
4. Comment reconnaît-on un nombre impair à partir de son écriture binaire ?

Remarque 4. Sous Python, la transcription d'un entier en binaire se fait avec la fonction `bin()`.
En tapant `bin(7)` dans l'interpréteur (ou la console, c'est pareil !), on obtient le résultat : `0b111` que l'on interprète par : $7 = \overline{111}^{(2)}$

`Exemple 3.`

Conversion en binaire en Python

```
for n in range(16):
  print(n,'->',bin(n),'\n')
```

```
0 -> 0b0        4 -> 0b100      8 -> 0b1000     12 -> 0b1100
1 -> 0b1        5 -> 0b101      9 -> 0b1001     13 -> 0b1101
2 -> 0b10       6 -> 0b110      10 -> 0b1010    14 -> 0b1110
3 -> 0b11       7 -> 0b111      11 -> 0b1011    15 -> 0b1111
```

PROPOSITION 2 : **Somme de deux entiers naturels en binaire**
Le principe d'addition est le même que celui en base 10 avec une retenue de 1 à propager sur le terme suivant lorsqu'on effectue $1+1$.

`Exemple 4.` Effectuer la somme des trois entiers binaires suivants : $n_1 = \overline{10101}^{(2)}$, $n_2 = \overline{1110}^{(2)}$, $n_3 = \overline{10011}^{(2)}$

1.2 Codage binaire de la partie fractionnaire

`Exemple 5.` Considérons le nombre $x = 0.241$

1. x se décompose de la façon suivante : $x = \mathbf{2}.10^{-1} + \mathbf{4}.10^{-2} + \mathbf{1}.10^{-3}$.
   On dit alors que $x = 0,241$ est l'écriture décimale ou l'écriture en base 10 de x.

2. Plus généralement, si $x \in [0,\ 1[$ et si b et un entier supérieur ou égal à 2, il est possible d'écrire x sous la forme :

$$x = \mathbf{a_{-1}}.b^{-1} + \mathbf{a_{-2}}.b^{-2} + \cdots + \mathbf{a_{-q}}.b^{-q} + R_{-q}(x) \quad \text{avec} \quad \begin{cases} (a_{-1},\ a_{-2},\ \ldots,\ a_{-q}) \in [\![0, b-1]\!]^q \\ 0 \leq R_{-q}(x) < b^{-q} \end{cases}$$

   On dit alors que $x = 0,\overline{a_{-1}a_{-2}\ldots a_{-q}}^{(b)}$ est une approximation de x en base b à b^{-q} près.

PROPOSITION 3 : **Codage binaire de la partie fractionnaire**
Soit $f \in]0,\ 1[$.
Pour tout $q \in \mathbb{N}^*$, il existe $(a_{-1},\ \ldots,\ a_{-(q-1)},\ a_{-q}) \in \{0,\ 1\}^q$ tel que :

$$f = a_{-1}.2^{-1} + a_{-2}.2^{-2} + \cdots + a_{-q}.2^{-q} + R_{-q}(x) \quad \text{avec} \quad 0 \leq R_{-q}(x) < 2^{-q}$$

Cette décomposition est unique.
L'écriture de f sous la forme : $n = 0,\overline{a_{-1}\ldots a_{-(q-1)}a_{-q}\ldots}$ s'appelle l'écriture binaire (ou en base 2) de f.

Remarque 5. Nous allons voir que, contrairement aux nombres décimaux qui ont un développement décimal fini, le codage en base 2 de la partie fractionnaire n'est pas toujours fini.

Méthode de décomposition de la partie fractionnaire f en base 2

On obtient les différentes valeurs de a_{-k} :
- en partant des valeurs $\begin{cases} x = f \\ k = 1 \end{cases}$
- en effectuant autant que nécessaire les trois opérations suivantes :

```
a <- Partie entière de (2x)      (a donne la valeur de a_{-k})
x <- 2x - a
k <- k+1
```

`Exemple 6.` Décomposition binaire de $x = 0.15$:

Nombre (n ← 2n – a)	2n	a = E(2n)	
0,15	0,30	0	a-1
0,30	0,60	0	a-2
0,60	1,20	1	a-3
0,20	0,40	0	a-4
0,40	0,80	0	a-5
0,80	1,60	1	a-6
0,60	1,20	1	a-7
0,20	0,40	0	a-8
0,40	0,80	0	a-9
0,80	1,60	1	a-10
0,60	1,20	1	a-11
0,20	0,40	0	a-12
0,40	0,80	0	a-13
0,80	1,60	1	a-14
∘	∘	∘	∘
∘	∘	∘	∘
∘	∘	∘	∘

On constate que $x = 0,15$ admet une écriture binaire infinie :
$x = 0,\overline{0010011001...}^{(2)}$ (on observe la périodicité de $\overline{1001}^{(2)}$).

Algorithme de la décomposition à p coefficients de f en base b :

```
# Initialisation des variables

  x = f       # Variable pour stocker la suite des nombres n
  L = []      # Liste qui contiendra les coefficients de la décomposition

# Boucle for

  for k in range(1,p+1) :
      a = floor(2*x)      # on calcule le nouveau coefficient binaire
      L.append(a)         # on le stocke dans la liste L
      x = 2*x - a         # on calcule le nouveau nombre x
# Résultat

 print(L)     # Pour afficher le contenu de L
```

Remarque 6. Nous venons de voir que certains nombres décimaux admettent un développement en base 2 infini. Réciproquement, est-ce que les nombres admettant un développement fini en base 2 peuvent avoir un développement décimal infini ?

Remarque 7. Une autre procédure simple pour obtenir la décomposition binaire de f à 2^{-q} près consiste à décomposer en binaire $\lfloor 2^q f \rfloor$.

Remarque 8. Quelques valeurs de référence :

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
0.5	0.25	0.125	0.0625	0.03125

`Exemple 7.` $(*)$

1. Déterminer l'écriture binaire des parties fractionnaires suivantes : $f_1 = 0.5$, $f_2 = 0.375$, $n_3 = 0.24$.
2. Déterminer les nombres entiers dont l'écriture binaire est : $f_1 = 0.\overline{101}$, $f_2 = 0.\overline{1101}$, $f_3 = 0.\overline{10011}$.
3. Combien de parties fractionnaires peut-on coder avec une écriture binaire comportant au plus p chiffres ? Quelle est la plus petite ?

COROLLAIRE 4 : **Ecriture binaire d'un nombre décimal positif**
Tout nombre décimal positif x se décompose de façon unique sous la forme :

$$x = (a_p.2^p + a_{p-1}.2^{p-1} + \cdots + a_1.2^1 + a_0.2^0) + (a_{-1}.2^{-1} + a_{-2}.2^{-2} + \cdots + a_{-q}.2^{-q}) + R_{-q}$$

où :

1. $p,\ q \in \mathbb{N}$
2. les valeurs a_k sont dans $\{0,\ 1\}$ avec $a_p = 1$
3. $0 \leq R_{-q}(x) < 2^{-q}$.

L'écriture de x sous la forme : $n = \overline{a_p a_{p-1} \dots a_1 a_0\ ,\ a_{-1} \dots a_{-(q-1)} a_{-q} \dots}^{(2)}$ s'appelle l'écriture binaire de x.

Remarque 9. En prenant $b \in \mathbb{N}^*$ avec $b \geq 2$, à la place de 2, on obtient une écriture de x en base b.

Exercice : 1
Donner la décomposition binaire du nombre $x = 17,2$.

2 Codage informatique des nombres entiers

2.1 Les unités (ou mots) 'mémoire'

DÉFINITION 2 : **bits et octets**

- BIT : Dans la mémoire VIVE d'un ordinateur, un *bit* (contraction de *Binary Digit*) correspond à une case mémoire qui peut uniquement contenir 0 ou 1. Cette contrainte est liée à la technologie utilisée : polarisation magnétique, charge, courant, tension électrique, intensité lumineuse...

- OCTET : Un groupement de 8 bits est appelé un *octet*.
  Tous les caractères utilisables par l'ordinateur sont codés en binaire sur un octet (on a donc un total de $2^8 = 256$ caractères disponibles) : ce codage est appelé le code ASCII.

  Sous Python :

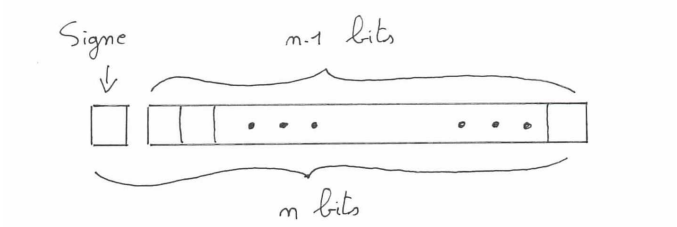
  (a) on obtient le code ASCII du caractère & en tapant "`ord('&')`".
  On obtient : 38.

  (b) on obtient le caractère correspondant au code ASCII 255 en tapant "`chr(255)`".
  On obtient : `'ÿ'`.

  ⚠ le terme anglais *byte* désigne un octet et non un bit.

2.2 Codage d'un entier

Le codage d'un entier par un ordinateur se fait en général sur 32 bits ou 64 bits.
Pour gagner en généralité, nous considérerons que ce codage utilisera n bits.
Le premier bit (appelé le *bit fort*) désigne le signe du nombre tandis que les $(n-1)$ bits restant permettent de coder la valeur absolue du nombre.



PROPOSITION 5 : Avec n bits, il est possible de coder tous les entiers de l'intervalle $[\![-2^{n-1}, 2^{n-1} - 1]\!]$.

Principe de codage des entiers
Lorsque les entiers sont codés sur 8 bits :

- Codage des entiers positifs :

  Le codage des entiers positifs est analogue au codage en binaire.
  Exemple : pour le nombre $n = 77 = \overline{1001101}$, le codage en machine est de la forme suivante :

  0100 1101

- Codage des entiers négatifs :

  En revanche, pour coder des entiers strictement négatifs, on procède ainsi :
  1. On code la valeur absolue du nombre en binaire
  2. On effectue un "complément à 2", c'est à dire qu'on échange tous les 1 avec des 0 et inversement
  3. On ajoute 1 au nombre binaire obtenu

  Exemple : pour le nombre $n = -77$, le codage en machine est de la forme suivante :

  10110011

L'avantage de coder ainsi les entiers négatifs est qu'elle est compatible avec l'addition de deux entiers.
En particulier, lorsqu'on additionne un nombre et son opposé en binaire, on obtient bien 0 (à vérifier !).

Remarque 10. On dit que les entiers sont codés en binaires *sur n bits signés en complément à 2.*

`Exemple 8.` $(*)$ Donner le codage machine sur un octet des entiers suivants : $n_1 = \pm\ 7$, $n_2 = \pm\ 14$ et $n_3 = \pm\ 42$

Pour retrouver un nombre entier connaissant son codage binaire en machine :

On effectue la démarche inverse...
On identifie le signe de l'entier grâce au premier bit.
1. Si le nombre est positif : celui-ci est codé en binaire et son décodage ne pose donc pas de difficulté.
2. Si le nombre est négatif :
   (a) On enlève "1" à l'entier codé sur les $(n-1)$ bits restants.
   Pour cela, il suffit de mettre à "0" le dernier bit égal à "1" et de mettre à "1" tous les bits précédents.
   (b) On effectue le complément à 2 en échangeant tous les "1" et les "0"
   (c) On obtient alors le codage binaire de la valeur absolue du nombre.

`Exemple 9.` $(*)$ Pouvez-vous donner la valeur des entiers codés en machine de la façon suivante :

1. $n_1 = 0\ 001\ 1011$
2. $n_2 = 1\ 010\ 1011$
3. $n_3 = 1\ 110\ 1000$

Remarque 11. Lorsque les entiers sont codés sur 32 bits :
— On peut représenter en machine tous les entiers positifs de 0 à $2^{32-1} - 1 = 2147483647$
— On peut représenter en machine tous les entiers négatifs de -1 à $-2^{32-1} = -2147483648$
Et si on codait les entiers sur 64 bits ?....

3 Codage informatique des nombres réels

3.1 Codage des nombres flottants

En informatique, les nombres réels sont codés sous la forme de nombres décimaux dits *nombres flottants.*
Pour leur représentation, les nombres flottants utilisent en général 32 bits (simple précision) ou 64 bits (double précision) : ils ne peuvent donc ni couvrir entièrement l'ensemble infini des nombres décimaux ni à fortiori celui des nombres réels. Beaucoup de nombres réels (décimaux ou pas) seront donc manipulés par l'ordinateur sous une

forme approximative.

Remarque 12. Cependant, il existe certains logiciels dits *de calcul formel*, capables de manipuler les nombres décimaux et même certains nombres rationnels sous leur forme exacte.

PROPOSITION 6 : **Approximation binaire d'un réel**

D'après la partie I de ce cours, nous pouvons déduire que tout nombre réel x non nul peut s'écrire sous la forme :

$$x = \pm \overbrace{(1.2^p + a_{p-1}.2^{p-1} + \cdots + a_0.2^0)}^{n} + \overbrace{(a_{-1}.2^{-1} + \cdots + a_{-(q-p)}.2^{-(q-p)} + R_{-(q-p)})}^{f} \quad \text{avec } 0 \leq R_{-(q-p)} < 2^{-(q-p)}$$

Ou encore : $$\boxed{x = \underbrace{\pm\, 2^p(1 + a_{-(1-p)}.2^{-1} + \cdots + a_{-(q-p)}.2^{-q})}_{\tilde{x}} + R_{-(q-p)} \quad \text{avec} \quad 0 \leq R_{-(q-p)} < 2^{-(q-p)}}$$

Ainsi, pour coder le nombre réel x, on décide de coder le nombre $\tilde{x} = \pm\, 2^p(1 + a_{-(1-p)}.2^{-1} + \cdots + a_{-(q-p)}.2^{-q})$ qui est le nombre flottant correspondant à x. Ce nombre $\tilde{x}$ est alors une approximation de x à $2^{-(q-p)}$ près.

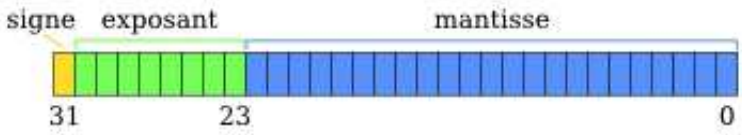
Le codage de $\tilde{x}$ se décompose en 3 étapes :

1. le signe
2. l'exposant p
3. le nombre $m = a_{p-1}.2^{-1} + \cdots + a_{-(q-p)}.2^{-q}$

Inutile en effet de coder le "1" puisque celui-ci est systématiquement présent (on dit que ce 1 correspond à un *bit caché* que l'on ajoute naturellement à la formule permettant de reconstituer le nombre dans le système décimal).

Principe de codage des nombres flottants sur 32 bits

Ce codage est imposé par la norme internationale IEEE754-1985.



On commence par exprimer le nombre x à coder sous la forme approximative :

$$\tilde{x} = \pm(1 + a_{-1}.2^{-1} + \cdots + a_{-q}.2^{-q})2^p \quad \text{avec} \quad \begin{cases} q = 23 \\ p \in [\![-126, 127]\!] \end{cases}$$

1. Le premier bit représente le signe s du nombre : $\begin{cases} 0 \text{ si le nombre est positif} \\ 1 \text{ si celui-ci est négatif} \end{cases}$.

2. Les 8 bits suivants codent l'exposant p.
   Le codage sur 8 bits donnant un entiers positif $e \in [\![0, 2^8 - 1 = 255]\!]$, on convient alors que $p = e - 127$.
   ⚠ Les cas où $e = 0$ ou $e = 255$ correspondent autre chose que des nombres (HP).
   Nous considèrerons donc que $e \in [\![1, 254]\!]$ et ainsi, nous pouvons coder tous les exposants p compris entre -126 et 127. Les nombres flottants codés de cette façon sont dits *normalisés*.

3. Les 23 derniers bits codent la partie $m = a_{-1}.2^{-1} + \cdots + a_{-q}.2^{-q}$ en base 2, appelée la mantisse de x.

Le nombre codé $\tilde{x}$ est alors donné par la formule : $\boxed{\tilde{x} = (-1)^s.(1 + m).2^{e-127}}$

Ce codage assez complexe a été choisi pour optimiser la vitesse de calcul et non pour faciliter la lecture humaine...

	Encodage	**S**igne	**E**xposant	**M**antisse	Valeur
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^S.M.2^{E-127}$
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^S.M.2^{E-1023}$

La norme IEEE754

Remarque 13.

1. Le plus grand nombre flottant normalisé est donc :
   (a) Avec 32 bits : $(1+(2^{-1}+2^{-2}+\cdots+2^{-23}))2^{127} = (2-2^{-23}).2^{127} = 3.40282346638528859811704183484516925440.10^{38}$
   (b) Et avec 64 bits ?

2. Le plus petit nombre flottant positif non nul normalisé est donc :
   (a) Avec 32 bits : $2^{-126} = 1.17549435082228750796873653722224567781866555677208752150 8.10^{-38}$
   (b) Et avec 64 bits ?

3. La plus grande précision que l'on peut espérer obtenir pour un réel $x \geq 1$:
   (a) Avec 32 bits ?
   Un tel réel s'écrit $x = 2^p(1 + \cdots + a_{-23}2^{-23}) + R_{p-23}$ avec le reste $R_{p-23} < 2^{p-23}$ qui est négligé par l'ordinateur. Dans le meilleur des cas (cad lorsque $p = 0$), la partie de x inférieure à $2^{-23} \approx 1,2.10^{-7}$ est négligée.
   (b) Et avec 64 bits ?
   C'est la partie de x inférieure à $2^{-52} \approx 2,210^{-16}$ qui est négligée.

`Exemple 10.`

1. Que vaut le nombre flottant codé par : $x = 0\ 01111100\ 01000000000000000000000$?

   (a) Le bit correspondant au signe est nul, et donc il s'agit d'un nombre positif,
   (b) L'exposant est $0.2^7 + 1.2^6 + 1.2^5 + 1.2^4 + 1.2^3 + 1.2^2 + 0.2^1 + 0.2^0 - 127 = 124 - 127 = -3$,
   (c) La partie significative est $1 + 0.2^{-1} + 1.2^{-2} = 1,25$.

   Le nombre représenté est donc $x = +1.25 \times 2^{-3} = +0.15625$.

2. Comment est codé en machine le nombre $x = 25,375$?

   (a) On décompose en binaire 25 et $0,375$: cela nous donne :
   $$x = 1.2^4 + 1.2^3 + 0.2^2 + 0.2^1 + 1.2^0 + 0.2^{-1} + 1.2^{-2} + 1.2^{-3}$$
   (b) On exprime x sous la forme adaptée :
   $$x = +(1 + 1.2^{-1} + 0.2^{-2} + 0.2^{-3} + 1.2^{-4} + 0.2^{-5} + 1.2^{-6} + 1.2^{-7})2^4$$
   (c) On détermine les différents élements du codage binaire :
      i. Le signe : 0
      ii. L'exposant : $4 + 127 = 131 = \overline{10000011}^{(2)}$
      iii. La mantisse : $\overline{10010110000000000000000}^{(2)}$ (on ne tient pas compte du bit caché !)

   Finalement :
   $$x = 0\ 10000011\ 10010110000000000000000$$

Exercice : 2

1. Déterminer le nombre flottant codé par : $x = 1\ 00101101\ 11010100000000000000000$?
2. Déterminer le codage machine du nombre : $1234,65625$.

3.2 Nombre de chiffres significatifs

Nous venons de voir qu'un réel x se décompose sous la forme :

$$\boxed{x = \tilde{x} + R_{p-q} \quad \text{avec} \quad \tilde{x} = \pm 2^p(1 + a_{-1}.2^{-1} + \cdots + a_{-q}.2^{-q}) \quad \text{et} \quad 0 \leq R_{p-q} < 2^{p-q}}$$

Nous avons ainsi :

$$2^p \leq x \quad \text{ce qui nous donne} \quad \left|\frac{x - \tilde{x}}{x}\right| \leq 2^{-q}$$

La version de Python avec laquelle nous allons travailler cette année code les flottants en double précision (sur 64 bits). Que dire de la précision avec laquelle les nombres réels seront représentés ?

En double précision, nous avons $q = 52$, ce qui nous donne une majoration de l'erreur relative :

$$\boxed{\left|\frac{x - \tilde{x}}{x}\right| \leq 2,3.10^{-16}}$$

Ce résultat signifie qu'en double précision, la valeur réelle de x n'est connue qu'avec seulement 16 chiffres significatifs.

3.3 Les erreurs éventuelles

1. Les erreurs d'arrondi :

   Les nombres flottants ne traduisent les réels qu'avec un nombre de chiffres significatifs limité ce qui engendre des erreurs dans les calculs effectués. Ces erreurs peuvent éventuellement s'accroître avec le nombre de calculs.

   En particulier :
   (a) lorsque $a << b$ on obtient $a + b = b$ car la valeur a est considérée comme nulle par l'ordinateur !!
   (b) lorsque $a \approx b$ l'erreur relative (cad le nombre de chiffres significatifs) sur le résultat de $a - b$ est très importante.

   Il faudra parfois être attentif à l'ordre des opérations. En double précision :
   (a) le calcul de $(2^{60} - 2^{60}) + 1$ donne 1
   (b) le calcul de $(2^{60} + 1) - 2^{60}$ donne 0

2. Les erreurs de dépassement :

   Lorsqu'on travaille avec des nombres très grands ou très petits, on peut parfois dépasser les valeurs extrêmes autorisées pour les nombres flottants. On rappelle qu'en simple précision (32 bits), les flottants positifs sont compris dans l'intervalle $[1,17.10^{-38}\ ;\ 3,41.10^{38}]$.

`Exemple 11.` **L'exemple du calcul de e^x :**

Il est théoriquement possible de calculer une approximation de e^x en utilisant la formule :

$$e^x = \sum_{k=0}^{n} \frac{x^k}{k!} \quad \text{pour} \quad n \in \mathbb{N}^*$$

Pour des valeurs négatives de x, les premiers termes de cette somme sont très grands et de signe contraire. Les premiers calculs engendrent alors des erreurs relatives très importantes. Le tableau ci-dessous montre la différence entre les résultats théoriques et les résultats obtenus par l'ordinateur (Python 3.4) pour différentes valeurs de x en prenant $n = 2000$.

Python
```
from math import factorial

def expon(x,n) :
        res = 0
        for k in range(0,n+1) :
         res = res + (x**k)/factorial(k)
        return res
```

x	valeur trouvée par l'ordinateur	e^x
-1	0.3678794411714	0.367879...
-5	0.0067379469990	0.0067379...
-10	0.0000453999294	0.0000454...
-20	0.0000000005478	0.0000000020612...
-30	-0.000085530164	0.000000000000093576
-40	0.1470264494805	0.0000000000000000042

4 Ce qu'il faut savoir et/ou savoir faire :

1. Sur les entiers :

   (a) Savoir coder / décoder un entier naturel en binaire
   (b) Savoir programmer en python le codage et le décodage d'un entier naturel en binaire
   (c) Savoir calculer le nombre de bits nécessaires pour coder un entier en binaire
   (d) Savoir déterminer le codage machine sur n bits d'un entier naturel
   (e) Savoir déterminer le codage machine sur n bits d'un entier strictement négatif
   (f) Savoir retrouver la valeur d'un entier dont on nous donne le codage binaire en machine
   (g) Connaître les plus petits et plus grands entiers relatifs que l'on peut coder avec n bits.

2. Sur les nombres flottants :

   (a) Savoir coder / décoder un nombre décimal positif en binaire avec p chiffres après la virgule
   (b) Savoir programmer en python le codage et le décodage en binaire d'un réel $x \in]0,\ 1[$
   (c) Savoir déterminer le codage machine d'un nombre flottant sur 32 bits
   (d) Savoir déterminer la valeur du nombre décimal dont on nous donne le codage machine sur 32 bits
   (e) Savoir expliquer pourquoi l'ordinateur ne reconnaît pas un nombre aussi simple que 0.1
   (f) Savoir expliquer pourquoi sous Python 3.4 les nombres flottants ne comportent que 16 chiffres significatifs
   (g) Savoir retrouver les valeurs du plus petit et du plus grand flottant normalisé strictement positif que l'on peut coder avec n bits
   (h) Avoir conscience des conséquences possibles des erreurs d'approximation dues au codage sous la forme de nombres flottants.

3. Entraînement :

   Faire un programme Python de décodage d'un nombre flottant donné par son codage machine sur 32 bits.
   On pourra considérer que le codage est donné sous la forme d'une liste L de longueur 32.
   On pourra également commencer par concevoir 2 sous-programmes, l'un calculant l'exposant p et l'autre calculant la mantisse.