1 Représentation des entiers naturels

1.1 Système de numération à position dans une base

Depuis le Moyen Âge, on écrit les nombres entiers naturels dans un système de numération à position en base 10. Pour représenter un entier naturel n, on imagine n objets, que l'on groupe par paquets de dix, puis on groupe ces paquets de dix objets en paquets de dix paquets, et ainsi de suite ... A la fin, il reste :

- un nombre d'objets isolés compris entre neuf et dix ;
- un nombre de paquets de dix objets compris entre neuf et dix ;
- ...

On représente cet entier naturel en écrivant de droite à gauche, le nombre d'objets isolés, le nombre de paquets de dix, le nombre de paquets de cent, le nombre de paquets de mille, ... Chacun de ces nombres étant compris entre zéro et neuf, seuls dix chiffres sont nécessaires : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9.
Ainsi 256 représente exprime un entier naturel formé de 6 unités, 5 dizaines et 2 centaines : $256 = 2 \times 10^2 + 5 \times 10 + 6$.
On peut choisir de faire des paquets de deux, de cinq, de seize, de soixante ... On écrit alors les nombres entiers naturels un système de numération à position en base 2, 5, 8, 16 ou 60 et le nombre de symboles nécessaires pour représenter les entiers naturels est 2, 5, 8, 16 ou 60.

Exercice 1

1. Déterminer l'écriture décimale de l'entier dont l'écriture en base 5 est 4021_5 puis celle de l'entier dont l'écriture en base 2 est 101_2.
2. Convertir en base 2 puis en base 5 les entiers d'écritures décimales 7, 34 et 128.

Exercice 2

1. Ecrire en Python une fonction `nbchifres` qui retourne le nombre de chiffres dans l'écriture en base 10 d'un entier n (entré en base 10 par l'utilisateur).
2. Ecrire en Python une fonction `base5to10` qui retourne l'écriture en base 10 d'un entier entré par l'utilisateur en base 5 (dont les chiffres sont 1, 2, 3, 4 et 5).

1.2 Numération en base 2

1.2.1 Numération en base 2 et informatique

Définition 1

La mémoire des ordinateurs est constituée d'une multitude de petits circuits électroniques (des transistors) et chacun ne peut être que dans deux états électriques : notés arbitrairement 0 et 1.
La valeur 0 ou 1 d'un circuit mémoire élémentaire s'appelle un **chiffre binaire**, un **booléen** ou un bit (abréviation de **binary digit**).
L'état d'un circuit composé de plusieurs circuits mémoires à 1 bit s'exprimer sous la forme d'une suite finie de bits appelée **mot**.
Depuis les années 1970, l'unité de mesure de l'espace (disque ou mémoire) est **l'octet** ou `byte`. Un octet correspond à 8 bits. Un octet peut donc prendre $2^8 = 256$ valeurs différentes.
Les préfixes multiplicateurs les plus utilisés sont définis en base 10 : 1 kilooctet (Ko) = 10^3 octets, 1 mégaoctet (Mo) = 10^6 octets, 1 gigaoctet (Go) = 10^9 octets.
Il existe aussi des préfixes multiplicateurs en base 2 : 1 kibioctet (Kio) = 2^{10} = 1024 octets, 1 mébioctet (Mio) = $2^{20} \approx 1{,}049$ mégaoctet, 1 gibioctet (Gio) = $2^{30} \approx 1{,}074$ gigaoctet.
Un microprocesseur est d'autant plus rapide que les mots qu'il peut traiter à chaque cycle d'horloge sont longs. Désormais les microprocesseurs manipulent souvent des mots de 64 bits et non plus de 32 bits.

Exercice 3

1. Si on code les entiers successifs sur un octet à partir de 0, quel est le plus grand entier qu'on peut représenter ?
2. Calculer le nombre d'états possibles pour un mot de 32 bits, puis pour un mot de 64 bits.

   Si on considère qu'une adresse mémoire permet d'adresser un bloc de 4 Kio. Quel volume maximal (en Tio) permet d'adresser un mot de 32 bits ? de 64 bits ?
3. Un disque dur a une capacité de 500 Go. Exprimer cette capacité en Gio.
4. Sachant qu'une minute de musique au format mp3 occupe un espace de 1 Mo, combien d'heures de musique peut-on stocker sur un baladeur de 18 Go.
5. Avec un débit de 80 Mbits/s, combien de temps faut-il pour télécharger un fichier de 1,8 Go ?
6. En mode `Truecolor` la couleur d'un pixel est donnée par un triplet (R,V,B) donnant les nuances de rouge, vert et bleu, chacune codée sur un octet. Calculer le nombre de couleurs possibles en `Truecolor`.

1.2.2 Principe de la numération en base 2

Il y a 10 *sortes de personnes, celles qui comprennent la numération en binaire et les autres.*

Exemple 1

Soit un entier n dont l'écriture en base 2 est 1101_2. Ecrire n en base 10.

Théorème-Définition 1

Tout entier naturel n (ou entier non signé) peut s'écrire de façon unique en base 2 sous la forme $c_k c_{k-1} \cdots c_1 c_0$ telle que :

$$n = \sum_{i=0}^{k} c_i \times 2^i \text{ avec } \forall i \in [\![0;k]\!],\ c_i \in [\![0;1]\!]$$

c_k est le bit de poids fort et c_0 est le bit de poids faible. Pour distinguer l'écriture en base 2 de l'écriture en base 10 on pourra écrire 101_2 pour par exemple l'écriture de 5 en base 2.

Exercice 4

1. Quel est le plus grand entier non signé qu'on peut représenter en base 2 avec un mot de 8 bits ? un mot de 64 bits ?
2. C'est en 11110010000_2 qu'Alan Turing a défini sa machine à calculer universelle. Exprimer ce nombre en base dix.
3. Montrer qu'avec un mot de n bits on peut représenter les nombres de 0 à $2^n - 1$.

Exercice 5 *Calculs en base 2*

1. Réaliser les additions ou soustractions suivantes en base 2 :

   a. $1_2 + 1_2 + 1_2 + 1_2$
   b. $1100_2 + 1000_2$
   c. $1100_2 - 1000_2$
   d. $1100_2 - 101_2$

2. Réaliser les multiplications ou divisions suivantes en base 2 :

a. $1011_2 \times 11_2$
b. $1100_2 \times 101_2$
c. $100100_2 \div 100_2$
d. $100100_2 \div 11_2$

Exercice 6

Méthode *Conversion en base 2, algorithme des divisions en cascade*

Pour écrire les entiers en base deux, on utilise deux chiffres : 0 et 1. Si on se donne un entier naturel n, on imagine qu'il représente n objets et on les groupe par paquets de deux, puis on groupe ces paquets en paquets de deux paquets, ...Ainsi,on fait une succession de divisions par 2, jusqu'à obtenir un quotient égal à 0 et l'écriture en base 2 de n est la liste des restes successifs obtenus (0 ou 1, appelés bits) mais dans l'ordre inverse (le premier bit obtenu est le bit de poids faible et le dernier celui de poids fort).
Avec cette *méthode des divisions en cascades* voici l'exemple de la conversion du nombre décimal 13 en base 2 :

Etape	N (dividende)	Q (Quotient)	R (Reste)
$13 = 2 \times 6 + 1$	13	6	1
$6 = 2 \times 3 + 0$	6	3	0
$3 = 2 \times 1 + 1$	3	1	1
$1 = 0 \times 2 + 1$	1	0	1

L'écriture de 13 en base 2 est 1101_2 et on a :

$$13 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

puis on applique l'algorithme de Horner pour l'évaluation d'un polynôme

$$13 = (2^2 + 2 + 0) \times 2 + 1$$

à la fin on voit apparaître la suite des chiffres obtenus par l'algorithme des divisions en cascade

$$13 = ((1 \times 2 + 1) \times 2 + 0) \times 2 + 1$$

1. Convertir en base 2 des entiers naturels suivants avec l'algorithme des divisions en cascade :

- 7
- 9
- 35
- 255
- 512
- 1025

2. Vérifier les résultats précédents dans la console Python avec la primitive `bin`.

```
>>> bin(9) #pour convertir de décimal en binaire
'0b1001'
>>> int('0b1001',2) #pour convertir de binaire en décimal
9
```

conversions binaire décimale

3. Ecrire en Python une fonction `lenb2` qui prend en argument un entier naturel n écrit en base 10 et qui retourne la longueur de son écriture en base 2.
Tester cette fonction sur les entiers de 1., on pourra aussi vérifier les résultats dans la console Python avec la méthode `bit_length` du type `int` qu'on utilise ainsi pour longueur de l'écriture en binaire de 13 : `(13).bit_length()`.

4. Ecrire une fonction `base10to2` qui prend en argument un entier naturel n écrit en base 10 et qui retourne un entier représentant son écriture en base 2.

On pourra implémenter cette version de l'algorithme des divisions en cascades :

Passage de l'écriture d'un entier N en base 10 à son écriture en base 2

Entrée	Demander N
Traitement	$0 \mapsto$ K ; $0 \mapsto$ A Tant que N > 0 A+RESTE(N/2)$\times 10^{K} \mapsto$ A Quotient de la division de N par 2 $\mapsto$ N K + 1 $\mapsto$ K Fin Tant Que
Sortie	Afficher A

En remplaçant 2 par B, on peut imaginer que cet algorithme retourne l'écriture en base B à partir de l'écriture en base 10, mais est-ce que celà va fonctionner pour toutes les bases ?

Ecrire une fonction `base10to2bis` qui implémente le même type d'algorithme mais retourne les chiffres de l'écriture binaire sous la forme d'un tableau.

5. Compléter le code des fonctions récursive `base10to2rec` ci-dessous pour qu'elles retournent l'écriture binaire d'un entier n écrit en base 10 sous la forme d'une chaîne de caractères :

```
1    def base10to2rec(n):
2    if n <= 1:
3        return str(n%2)
4    return .....
```

6. Ecrire la fonction `base2to10` réciproque de la précédente qui prend en argument un entier représentant l'écriture binaire d'un entier n et qui retourne l'écriture décimale sous forme d'entiers.

Ecrire une fonction `base10to2bis` qui implémente le même type d'algorithme mais retourne les chiffres de l'écriture binaire sous la forme d'un tableau.

Méthode *Conversion en base 2, algorithme n°2*

On commence par déterminer p la plus grande puissance de 2 inférieure ou égale à l'entier n.
On effectue ensuite une boucle sur toutes les puissances de 2 inférieures ou égale à p.

- ☞ si la puissance de 2 est inférieure ou égale à l'entier variable, le bit correspondant de la représentation binaire de l'entier initial n vaut 1, puis on modifie l'entier variable en lui retranchant cette puissance de 2 ;
- ☞ sinon le bit correspondant de la représentation binaire de l'entier initial n vaut 0.

Entier	Puissance de 2	Bit
13	2^3	1
$13-2^3=5$	2^2	1
$5-2^2=1$	2^1	0
1	2^0	1

On retrouve ainsi l'écriture binaire de 13 qui est 1101_2.

7. Ecrire une fonction `puissance2dans(n)` qui retourne la plus grande puissance de 2 inférieure ou égale à un entier `n`.

8. Ecrire une fonction `b10to2(n)` qui prend en argument un entier naturel n écrit en base 10 et qui retourne une chaîne de caractères représentant son écriture en base 2 en appliquant l'algorithme 2 ci-dessus.

9. Compléter le code d'une fonction `b10to2rec(n)` qui prend en argument un entier naturel n écrit en base 10 et qui retourne une chaîne de caractères représentant son écriture en base 2 en appliquant l'algorithme 2 ci-dessus.

```
def b10to2rec(n):
    """Conversion de base dix en base deux. Fonction récursive avec enveloppe"""

    def b10to2rec2(n, p):
        if p == 0:
            return ''#terminaison de la récursion
        elif p <= n:
            return .........
        else:
            return .........

    p2 = puissance2dans(n)
    return b10to2rec2(n, p2)
```

Exercice 7

Exemple de conversion en base 2 du nombre décimal 6.59375 :

- on convertit d'abord la partie entière en base 2 selon la méthode exposée ci-dessus soit $6 = 110_2$;
- on convertit ensuite la partie fractionnaire en multipliant successivement par 2 pour obtenir les bits de poids 2^i avec i exposant négatif jusqu'à épuisement de la partie fractionnaire :
  - ❄ $0.59375 \times 2 = 1.1875 \longrightarrow$ Poids binaire 1×2^{-1} ;
  - ❄ $0.1875 \times 2 = 0.375 \longrightarrow$ Poids binaire 0×2^{-2} ;
  - ❄ $0.375 \times 2 = 0.75 \longrightarrow$ Poids binaire 0×2^{-3} ;
  - ❄ $0.75 \times 2 = 1.5 \longrightarrow$ Poids binaire 1×2^{-4} ;
  - ❄ $0.5 \times 2 = 1 \longrightarrow$ Poids binaire 1×2^{-5} ;
- On en déduit que l'écriture binaire de 6.59375 est 110.10011_2.

1. Convertir en binaire les nombres décimaux 35.1875 et 14.015625

2. L'écriture en binaire de 0.1 est-elle finie ?

3. Ecrire en Python une fonction `base10to2flottant(n,digits)` qui retourne un flottant qui est la représentation en base 2 du flottant n avec une précision digits bits après la virgule.

a. Retrouver avec cette fonction le développement en base 2 de 6.59375.

b. Déterminer avec cette fonction le développement en base 2 du nombre décimal 0.1 avec un nombre de digits croissant : 5, 10, 20, 50 , 100, 500. Que peut-on conjecturer sur la longueur de son développement en base 2 ? La représentation affichée en console comporte-t-elle tous les digits réellement calculés (qu'on pourra stocker dans une liste) ?

Exercice 8

Pour multiplier par dix un entier naturel exprimé en base dix, il suffit d'ajouter un 0 à sa droite, par exemple, $14 \times 10 = 140$. Quelle est l'opération équivalente pour les entiers naturels exprimés en base deux ? Exprimer en base deux les nombres 5, 10 et 20 pour vérifier cette remarque.
Dans la console Python (voir ci-dessous), on peut aussi utiliser l'opérateur « qui dans permet de décaler l'écriture binaire d'un entier d'un nombre fixé de bits vers la gauche. L'opérateur » permet de décaler l'écriture binaire d'un entier d'un nombre fixé de bits vers la droite.

```
>>> 34>>1,34<<1
(17, 68)
>>> bin(34),bin(34>>1),bin(34<<1)
('0b100010', '0b10001', '0b1000100')
```

opérations binaires

Exercice 9

1. Déterminer l'écriture en binaire de 67 puis celle de 188.
2. Soit n un entier compris entre 0 et 255, représentable en binaire sur un octet. Soit n' l'entier dont la représentation en binaire est obtenue à partir de celle de n en remplaçant les 0 par des 1.
   Montrer que $n+n'=255$. On appellera n' le complément binaire sur 8 bits de n.
3. En déduire l'écriture en Python d'une fonction `complement`, qui prend en argument un entier n compris entre 0 et 255 et retourne la représentation binaire du nombre $255-n$ ou 'erreur' si n n'est pas dans la plage souhaitée.

Exercice 10 *extrait d'un TP de Stéphane Gonnord*

Soit a un réel positif. Pour calculer a^n, si on utilise l'algorithme naif : $a\times a\times\cdots\times a$ cela nécessite $n-1$ multiplications.
Pour réduire le coût en opérations, on peut remarquer que :

$$a=\begin{cases}\left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair}\\ a\times\left(a^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair}\end{cases}$$

On peut alors naturellement programmer un algorithme d'exponentiation rapide récursif qui nécessite au plus $\log_2(n)$ multiplications où $\log_2(n)$ est le logarithme en base 2 de n.
La partie entière de $\log_2(n)$ est le nombre de chiffres de l'écriture binaire de n moins 1, de la même façon que la partie entière du logarithme décimal d'un entier n est le nombre de chiffres de son écriture décimale moins 1.
Si on note $\lfloor x\rfloor$ la partie entière de x on a :

$$2^{\log_2(n)}=n \text{ et } 2^{\lfloor\log_2(n)\rfloor}\leqslant n<2^{\lfloor\log_2(n)\rfloor+1}$$

```
def expo_rapide_rec(a,n):
    if n==0:
        return 1
    elif n%2==0:
        return expo_rapide_rec(a**2,n//2)
    return a*expo_rapide_rec(a**2,n//2)
```

exponentiation rapide version récursive

On peut implémenter l'exponentiation rapide avec un algorithme itératif en regardant le calcul suivant :

$$\begin{aligned}a^{42} &= \underbrace{1}_{res}.\underbrace{(a}_{P})^{\overbrace{42}^{N}}=\underbrace{1}_{res}.\underbrace{(a^2}_{P})^{\overbrace{21}^{N}}=\underbrace{a^2}_{res}.\underbrace{(a^4}_{P})^{\overbrace{10}^{N}}\\ &= \underbrace{a^2}_{res}.\underbrace{(x^8}_{P})^{\overbrace{5}^{N}}=\underbrace{(a^2.a^8)}_{res}.\underbrace{(a^{16}}_{P})^{\overbrace{2}^{N}}=\underbrace{(a^2.a^8)}_{res}.\underbrace{(a^{32}}_{P})^{\overbrace{1}^{N}}\end{aligned}$$

et celui-ci :

$$a^{42}=a^{0\times2^0+1\times2^1+0\times2^2+1\times2^3+0\times2^4+2^5}=a^{2+8+32}$$

Regarder comment les quantités res, P et N évoluent au cours du temps, et en déduire une fonction `expo_rapide_iter(a,n)` qui retourne a^n calculé avec un algorithme d'exponentiation rapide itératif.

1.3 Numération en base quelconque

Théorème-Définition 2

- Soit b un entier naturel non nul, tout entier naturel n peut s'écrire de façon unique en base b sous la forme $c_k c_{k-1} \cdots c_1 c_0$ telle que :
$$n = \sum_{i=0}^{k} c_i \times b^i \text{ avec } \forall i \in [\![0; k]\!],\ c_i \in [\![0; b-1]\!]$$
c_k est le bit de poids fort et c_0 est le bit de poids faible. Pour distinguer l'écriture en base b de l'écriture en base 10 on pourra écrire 13_4 pour par exemple l'écriture de 7 en base 4.
- Pour déterminer les chiffres de l'écriture en base b d'un entier n à partir de son écriture décimale, on effectue les divisions successives par b jusqu'à obtenir un quotient nul. Les chiffres sont obtenus dans l'ordre inverse de leur poids dans l'écriture de n en base b.

Exercice 11

Modifier la fonction `base10to2` précédente pour écrire une fonction `base10tob` qui prend en argument un entier naturel n écrit en base 10 et qui retourne sous forme de liste son écriture en base b où $b \geqslant 1$. Les écritures des entiers de 0 à $b-1$ en base 10 représenteront les chiffres en base b (pas de confusion possible puisque le type de données retourné est une liste).

Exercice 12

En informatique, on utilise souvent la représentation en base 16 (dite **hexadécimale**).
En base 16, on a besoin de 16 chiffres : 0, 1, 2, 3, 4, 5 , 6, 7, 8, 9, puis A (10), B (11), C (12), D (13), E (14) et F (15).
Les entiers en base 16 sont souvent précédés des symboles `0x`, ainsi `0xF5` représente en base 16 l'entier $15 \times 16 + 5 = 245$ et `0xFF` représente l'entier $15 \times 16 + 15 = 16^2 - 1 = 255$.

1. Déterminer à la main l'écriture en base 16 des entiers 17, 35, 171, 255 et 865.
2. Convertir en écriture décimal les nombres hexadécimaux `0x3AE`, `0xFFF` et `0x6AF`.
3. Modifier la fonction `base10tob` en une fonction `base10to16` qui retourne la liste des chiffres de l'écriture en base 16 (ou une chaîne de caractères) d'un entier n en respectant la nomenclature ci-dessus. On pourra utiliser un dictionnaire pour la traduction des chiffres.
Cette fonction réalise la même action que la primitive `hex` de Python.
4. La représentation hexadécimale s'obtient facilement à partir de la représentation binaire. Puisque $2^4 = 16$, pour convertir un octet de binaire en hexadécimal, il suffit de regrouper les bits par 4.
Ainsi 154 représenté en binaire sur un octet par $\overbrace{1001}^{9_{16}}\overbrace{1010}^{A_{16}}{}_2$ a pour représentation hexadécimale $9A$ car $9 \times 16 + 10 = 154$.
Convertir de même en hexadécimal les octets suivants :
   - 11101010
   - 10011011
   - 1100000
5. La notation hexadécimale est plus compacte que la représentation décimale. Par exemple, l'adresse MAC (Medium Access Control) d'une carte réseau d'ordinateur est constitué de six entiers compris entre 0 et 255 (six octets) qu'on note sous forme hexadécimale.
Traduire en notation décimale puis en binaire l'adresse MAC suivante : $A1-FB-64-3B-AB-FF$. Ecrire une fonction Python qui traduit une adresse MAC en notation binaire.
6. Pour représenter les couleurs on utilise souvent en informatique le mode Rouge Vert Bleu (RGB en anglais). Avec le principe de synthèse additive des couleurs, une couleur est ainsi représentée par les valeurs de ses trois composantes primaires mesurées sur une échelle de 0 à 255. Par exemple le rouge se code (255,0,0), le vert (0,255,0), le bleu (0,0,255), le jaune (255,255,0), le noir (0,0,0) et le blanc (255,255,255).
De plus on utilise souvent la notation HTML des couleurs qui est une chaîne constituée de 7 caractères :

- un symbole # pour commencer
- deux symboles correspondant au niveau de rouge codé en hexadécimal
- deux symboles correspondant au niveau de vert codé en hexadécimal
- deux symboles correspondant au niveau de bleu codé en hexadécimal

Par exemple le jaune (255,255,0) a #FFFF00 pour notation HTML, et le blanc (255,255,255) a #FFFFFF pour notation HTML.

a. Retrouver le triplet (R,G,B) en notation décimale correspondant aux couleurs de notation HTML : #8B8B46, #535346 et #83D07B.

b. Ecrire la notation HTML des couleurs de triplets (R,G,B) : (123,184,208), (228,44,185).

c. Ecrire en Python une fonction `rgb2html` qui prend en entrée 3 entiers R,G et B entre 0 et 255 et qui retourne une chaîne de caractères correspondant à la notation HTML de la couleur (R,G,B). Ecrire la fonction réciproque `html2rgb`.

Exercice 13

Convertir 3 et 2 en base 4 puis 32_4 en base 2. Convertir 11 puis 10 en base 4 puis en base 2 puis BA_{16} en base 4 puis en base 2.
Comment passer rapidement de la base 4 à la base 2 ? de la base 16 à la base 4 puis à la base 2 ?

Exercice 14

Les Shadoks ne possèdent que 4 mots pour compter : `GA (pour 0) - BU (pour 1) - ZO (pour 2) - MEU (pour 3)`, et comptent donc en base 4...
Pour un cours de numération Shadok, voir le site : `http://pedago52.fr/guillemin/maths/shadoks/doc.htm`

1. Ecrire un programme qui prend en entrée un entier positif et qui le convertit en numération Shadok (sous forme d'une liste de chaînes de caractères prise parmi "GA" - "BU" - "ZO" - "MEU").
2. Ecrire un programme qui prend en entrée un entier positif écrit en numération Shadok et qui retourne l'écriture décimale correspondante.

2 Représentation des entiers relatifs

Théorème-Définition 3

Pour représenter avec un mot de n bits les entiers relatifs (ou entiers signés) deux solutions sont possibles :

- une première solution simple est de réserver le premier bit au signe (1 pour positif et 0 pour négatif) mais deux problèmes se posent :
  - il y a deux zéros ;
  - avec ce codage, $a-b \neq a+(-b)$ et il faut programmer différemment l'addition et la soustraction
- une autre solution appelée **complément à 2,** consiste à répartir les entiers naturels codés sur n bits en deux groupes :
  - un entier x compris entre 0 et $2^{n-1}-1$ représente l'entier positif x ;
  - un entier x compris entre 2^{n-1} et 2^n-1 représente l'entier négatif $x-2^n$;

  On peut ainsi représenter sur n bits les entiers positifs compris entre 0 et $2^{n-1}-1$ et les entiers négatifs compris entre -2^{n-1} et -1.

  Si y compris entre -2^{n-1} et -1 sa représentation en complément à 2 est l'écriture binaire de $y+2^n$.

  Il y a un entier négatif de plus mais avec le codage en complément à 2 on a bien $a-b=a+(-b)$

Exemple 2

En Python, il n'existe qu'un type `int` pour des entiers signés, il n'existe pas de type pour des entiers non signés comme en C.
Si les entiers signés sont représentés sur un nombre fini de bits, on peut imaginer qu'un dépassement de capacité (ou *overflow*) se produit lorsqu'on dépasse ces limites.Par défaut les entiers signés sont représentés sur 32 bits ou 64 bits selon l'architecture de la machine. Lorsque celà se produit, Python change automatiquement le type de représentation et rajoute des bits.
En Python2, le suffixe L (pour entier long) apparait après l'entier dans sa représentation canonique obtenue avec la fonction `repr()` (l'affichage peut différer de celui donné par `print()` car par principe on doit avoir `eval(repr(objet)) == objet`).
En Python3, c'est transparent et on peut considérer que les calculs sur les entiers se font en *précision arbitraire* (limitée quand même par la mémoire de la machine).
Tester les commandes suivantes en Python2 sur sa machine pour déterminer si elle l'architecture utilisée est 32 bits ou 64 bits :

```
>>> for i in range(64):
...     print i,'->',repr(2**i-1)
>>> import sys
>>> sys.maxsize,2**31-1 #plus grand entier signé représentable sur 1 mot (ici de 32 bits)
(2147483647, 2147483647L)
```

entiers en Python2

Exemple 3

On donne ci-dessous la représentation en complément à deux des entiers signés sur 3 bits :

Représentation binaire	000	001	010	011	100	101	110	111
Entiers	0	1	2	3	-4	-3	-2	-1

Exercice 15

1. Déterminer la représentation binaire en complément à 2 sur 8 bits des entiers signés :
   - 54 | −128 | −1 | −2 | 127 | −127
2. Quelles plages d'entiers positifs et négatifs peut-on représenter en complément à 2 sur 8 bits ? sur 16 bits ?
3. Déterminer l'entier représenté en complément à 2 sur 8 bits par :
   - 10000001 | 11111111 | 11111101 | 01111110
4. Notons r la représentation binaire en complément à 2 d'un entier n.

   Si $n \in [\![1\,;127]\!]$, on a $r = n$ et la représentation en complément à 2 de $-127 \leqslant -r \leqslant -1$ est celle de $2^8 - r = 256 - r = (255-r)+1$.

   Si $n \in [\![-127\,;-1]\!]$, on a $r = 256+n$ soit $-n = 256-r$. Comme $-n \in [\![1\,;127]\!]$ il est égal à sa représentation binaire en complément à 2 et donc la représentation binaire en complément à 2 de $-n$ est encore $256-r = (255-r)+1$.

   On a vu dans l'exercice 9 que si on change tous les bits de la représentation binaire d'un entier n compris entre 0 et 255 alors on obtient l'écriture binaire de $n' = 255-n$.
   Pour tout entier $n \in [\![-127\,;127]\!]$ privé de 0, on obtient donc la représentation binaire en complément à 2 de $-n$ en inversant tous les bits de la représentation de n et en ajoutant 1.

   Pour 0, son écriture binaire sur huit bits est 00000000_2, si on inverse tous les bits celà donne 11111111_2 et si on ajoute 1 celà donne $1\overbrace{00000000}^{8\text{ bits}}$ et comme on est limité à huit bits on tronque le neuvième bit dit de débordement et on retrouve bien 00000000_2. Ainsi on a bien $-0 = 0$.

   Pour le plus petit entier négatif sur huit bits qui est $-128 = 10000000_2$, si on inverse tous les bits et qu'on ajoute 1 on obtient $01111111_2 + 1_2 = 10000000_2$ c'est-à-dire la représentation en complément à 2 de -128. C'est un débordement qui génère un résultat faux, ce qui est logique puisque sur huit bits seuls les entiers positifs de 0 à 127 sont représentés.

   Avec cette méthode, déterminer l'écriture binaire sur 8 bits en complément à 2 de l'opposé des entiers ci-dessous codés en complément à 2 sur 8 bits, puis préciser la valeur de l'entier et de son opposé :

• 100	• 1100100	• 11111000	• 1	• 1111111	• 10000011	• 10000001

En Python, on inverse les bits d'un entier en le préfixant avec l'opérateur `~`, on pourra tester la commande `~100+1`.

5. Ecrire en Python une fonction `oppose_complement2(n)` qui retourne l'écriture binaire en complément à deux de l'opposé d'un entier `n` compris entre -127 et 127 ou un message d'erreur de débordement sinon.
6. Effectuer en binaire l'addition $94+38$. Quel est l'entier relatif obtenu avec un codage sur 8 bits ? Pourquoi est-il négatif ?
7. Déterminer le codage binaire en complément à 2 sur 8 bits des entiers 19 et -6, effectuer en binaire l'addition $19+(-6)$ et vérifier qu'on obtient le même résultat que $19-6$ (le résultat est codé sur 8 bits et s'il y a un bit qui « dépasse », il ne compte pas ...)

3 Représentation des réels : les flottants

Définition 2

- En informatique, les nombres réels qui ne sont pas des entiers signés sont représentés par des approximations appelées **flottants**. La notation des flottants correspond à **la notation scientifique**, elle se présente sous la forme $s \times m \times b^e$ où s est le signe (1 ou -1), m la mantisse (ou nombre de chiffres significatifs), b la base de représentation (2 en informatique) et e l'exposant (qui positionne la virgule). Le terme flottant fait référence à la position mobile de la virgule.
- Dans une représentation en virgule fixe en base 2 avec une mantisse sur deux chiffres et un chiffre après la virgule on ne peut représenter que sept nombres :

  $$-1,1 \quad -1,0 \quad -0,1 \quad 0,0 \quad 0,1 \quad 1,0 \quad 1,1$$

  Dans ce cas l'écart entre deux flottants consécutifs est constant et on n'a pas besoin de coder la virgule fixe.
- Dans une représentation en virgule flottante en base 2 avec une mantisse toujours sur deux chiffres mais avec une virgule flottante qui peut prendre deux positions (exposants -1 ou 0) on peut représenter onze nombres :

  $$-11, \quad -10, \quad -1,1 \quad -1,0 \quad -0,1 \quad 0,0 \quad 0,1 \quad 1,0 \quad 1,1 \quad 10, \quad 11,$$

  On remarque d'ailleurs un problème d'unicité de la représentation puisque 0 peut se noter 00, ou 0,0 et -1 peut se noter $-1,0$ ou $-01,$. C'est pourquoi on imposera en pratique une notation normalisée avec un premier bit de mantisse non nul et on représentera le 0 par l'exposant le plus petit possible (virgule la plus à gauche) et une mantisse égale à 1 suivie uniquement de 0 (Voir théorème suivant).

  En virgule flottante, on peut donc représenter plus de nombres mais l'écart entre deux flottants consécutifs est variable et il faut un espace mémoire supplémentaire pour coder la place de la virgule (l'exposant).
- Les performances des ordinateurs se mesurent en vitesse d'opérations sur les flottants ou FLOPS.

  Le site `http://www.top500.org/` établit chaque année un classement des supercalculateurs les plus rapides. En juin 2013, le numéro 1 était le chinois Tianhe-2 avec une vitesse de calcul de 33,86 petaflops/s. classement

Exemple 4

En Python, un nombre réel (il existe un type `complex`) est représenté par un entier ou par un flottant.
Les flottants n'offrent qu'une précision d'une quinzaine de décimales. Néanmoins cette précision peut suffire si l'on calcule avec des grandeurs physiques dont la plupart sont connues avec une dizaine de chiffres significatifs.
Si on a besoin de valeurs exactes, il vaut mieux calculer avec des entiers. En finance par exemple, on manipule des sommes d'une précision de 10^{-4} euros. On pourrait calculer en euros avec des flottants mais comme on a besoin de résultats exacts, on calcule avec des entiers en changeant d'unité (on compte en dix-millième d'euro).

```
>>> type(2),type(2.)
(<class 'int'>, <class 'float'>)
>>> 2.**1023
8.98846567431158e+307
>>> 2.**1024
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
>>> 2**(-1074), 2**(-1075)
5e-324, 0.0
>>> a = 0
>>> for i in range(10):
...     a += 0.1
>>> a
0.9999999999999999
>>> a == 1
False
>>> a = 0
>>> a = a + 10*0.1
>>> a
1.0
```

Des flottants surprenants

Exercice 16

1. Tester dans la console Python les instructions suivantes. Commentez les résultats obtenus.

```
>>> x = 0.1
>>> 3*x-0.3
>>> y = 0.5
>>> 3*y-1.5
>>> 1+10**(-15) == 1 ; 1+10**(-16) == 1
>>> 1+2**(-52) == 1; 1+2**(-53) == 1 ; 1+2**(-53) == 1+10**(-16); 2**(-53) == 10**(-16)
```

calculs déroutants

2. Tester le programme suivant. Interpréter les résultats obtenus.

```
a,b, i = 1., 2., 1 # affectations parallèles, a et b de type float d'où le .
b = 2.
i = 1
while i<11:
    a, b, c = b, b*a, 2**(2**i)
    print(type(b),b,' ; ',type(c),c)
    i += 1
```

limites des flottants

Théorème-Définition 4

1. On ne peut pas représenter les nombres réels en précision infinie.

2. Pour représenter les réels, on les approche par un ensemble fini de rationnels de la forme $\frac{a}{2^n}$, appelés flottants et qui sont choisis de telle sorte qu'il y ait autant de flottants entre $2^{-1}=\frac{1}{2}$ et $2^0=1$, qu'entre 1 et $2=2^1$, qu'entre 2 et $4=2^2$ …

3. Pour les flottants, les ordinateur affichent en sortie des valeurs notées en base 10, sous forme décimale ou scientifique. Mais en interne, les flottants sont représentés les selon la norme IEEE754, par un triplet unique : **(signe, exposant, mantisse)**. C'est une représentation similaire à la notation scientifique mais en base 2.
Par exemple pour $-12.015625=-2^3\times\left(1+\frac{1}{2}+\frac{1}{2^9}\right)$ le signe est -1, l'exposant est 3 et la mantisse est $\frac{1}{2}+\frac{1}{2^9}$.
Les valeurs affichés en sortie sont par défaut notés en base 10, sous forme décimale ou scientifique.

- Le **signe** est codé par un bit de signe s (0 pour positif et 1 pour négatif).
- L'**exposant** (positif ou négatif) est codé sur n bits par un entier naturel e. Les valeurs extrêmes 0 et 2^n-1 sont réservées et on ne peut utiliser en fait que 2^n-2 valeurs pour coder comprises entre 1 et 2^n-2. L'exposant réel codé par e est $e-(2^{n-1}-1)$.
- La **mantisse** m est la partie fractionnaire en base 2.
  Si m est codé sur p bits par $b_1\,b_2\cdots b_p$ alors $1.m=1+\frac{b_1}{2}+\frac{b_1}{2^2}+\cdots+\frac{b_p}{2^p}$.

Un flottant f codé par le triplet (s,e,m) avec e codé sur n bits et tel que $0<e<2^n-1$ est dit normalisé :

$$f=(-1)^s\times 2^{e-(2^{n-1}-1)}\times(1.m) \tag{1}$$

Valeurs particulières :

- Si $e=0$ et $m=0$ alors $f=0$ (deux zéros puisque le bit de signe s peut prendre deux valeurs)
- Si $e=0$ et $m\neq 0$, le flottant est dénormalisé et sa valeur est $2^{2-2^{n-1}}\times 0.m$.
- Si $e=2^n-1$ et $m=0$ alors $f=\infty$ ($+\infty$ ou $-\infty$ suivant la valeur du bit de signe s).
- Si $e=2^n-1$ et $m\neq 0$ alors $f=$ NaN (Not a Number).

Exemple 5 *représentation au format double de la norme IEEE754*

Si on considère la représentation des flottants au format double (sur $n=64$ bits) , s se code sur 1 bit, e sur 11 bits et la mantisse m sur 52 bits.

1. Dans ce format, le décimal $12,625=2^3+2^2+2^{-1}+2^{-3}=2^3\times\left(1+2^{-1}+2^{-4}+2^{-6}\right)$ de développement binaire 1100.101 se codera ainsi :

- le nombre est positif donc $s=0$;
- $2^3\leqslant 12<2^4$ donc l'exposant réel est 3 représenté sous la forme $3+2^{11-1}-1=3+1023=1026$ soit donc $e=10000000010$ en binaire ;
- Avec $12,625=2^3\times\left(1+2^{-1}+2^{-4}+2^{-6}\right)$ ou avec la notation exponentielle binaire $1100.101=1.\underbrace{100101}_{mantisse}$ E 03 on voit que la mantisse est $m=100101$ codée $100101\underbrace{00\cdots00}_{46\text{ zéros}}$ sur 52 bits.

Finalement le codage de 12,625 au format double est : $\underbrace{0}_{s}\,\underbrace{10000000010}_{e}\,\underbrace{100101\cdots\cdots00}_{\text{mantisse sur 52 bits}}$

2. Le décimal 0,1 présente un développement infini (mais périodique) en binaire qui est $0.0001100110011001\ \underbrace{1001}_{période}$. Il est donc représenté de façon approchée, ce qui explique que pour la machine $3\times 0,1\neq 0,3$.
Pour déterminer sa représentation au format double on passe en notation binaire exponentielle :
$0.0001100110011001\ \underbrace{1001}_{période}=1.\underbrace{1001}_{période}$ E -04 (car $2^{-4}\leqslant 0,1<2^{-3}$).
On en déduit que :

- $s = 0$ car 0,1 est positif ;
- l'exposant réel est -4 donc sa représentation est $-4 + 2^{11-1} - 1 = -4 + 1023 = 1019$ soit donc $e = 01111111011$ en binaire sur onze bits ;
- Dans la notation binaire exponentielle du décimal 0,1, on peut lire sa mantisse $m = 1001\cdots \underbrace{1001}_{période}$.

  Comme la précision de l'ordinateur n'est pas infinie, cette mantisse n'est codée que sur 52 bits au format double (la période est donc répétée treize fois car $4 \times 13 = 52$).

Finalement le codage de 0,1 au format double est : $\underbrace{0}_{s}\ \underbrace{01111111011}_{e}\ \underbrace{1001\cdots\cdots\cdots 1001}_{\text{mantisse}}$.

D'après les tests effectués avec le programme `base10to2flottant.py`, en Python, le $52^{ième}$ bit de la mantisse pour 0,1 n'est pas exact (1 au lieu de 0). Il est arrondi au bit le plus proche 0 car le bit suivant est un 0 :

```
Entrez un flottant : 0.1
Entrez un nombre de digits : 56
Nombre de bits de la partie fractionnaire en binaire réellement calculés ; 55
Liste des bits de la partie fractionnaire en binaire réellement calculés :
 [0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1,
      1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1]
La liste des bits affichés est réduite à 20 bits après la virgule au maximum car le
     nombre affiché est un flottant :
Développement en base 2 de 0.1 : 0.00011001100110011001
```

Exercice 17

On considère un codage des flottants sur 6 bits avec 1 bit de signe s, l'exposant e codé sur 3 bits et la mantisse m codée sur 2 bits. Le décalage de l'exposant est donc de $2^{e-1} - 1 = 2^2 - 1 = 3$.

exposant e	mantisse m	valeur	type
$0 < e < 2^3 - 1$	m	$(-1)^s \times 2^{e-3} \times (1.m)$	normalisé
$e = 0$	$m \neq 0$	$(-1)^s \times 2^{-2} \times (0.m)$	dénormalisé
$e = 0$	$m = 0$	$(-1)^s \times 0$	zéro
$e = 7$	$m = 0$	$(-1)^s \times \infty$	infini
$e = 7$	$m \neq 0$	pas un nombre	NaN

1. Ecrire un programme en python qui affiche tous les flottants positifs qu'on peut ainsi représenter.
2. Ecrire un programme en python qui affiche tous les flottants positifs qu'on peut représenter avec `p` bits d'exposant et `q` bits de mantisse.

Exercice 18

D'après la norme IEEE754, les **flottants en double précision**, (type `float` de Python ou `double` de Java) sont codés sur 64 bits avec 1 bit pour le signe s, 11 bits pour l'exposant e et 52 bits pour la mantisse m.
La mantisse correspond environ à 16 chiffres décimaux.

1. Rechercher sur internet la signification du sigle IEEE.
2. Comment sont représentés les nombres suivants :
   - 2^{-1022} | -2^{1023} | $2^5 \times 1.5$ | $2^{-4} \times 1.1875$
3. Quel est le plus grand flottant positif qu'on peut représenter avec ce codage ?
4. Quel est le plus petit flottant normalisé positif qu'on peut représenter avec ce codage ? et le plus petit flottant dénormalisé ?
5. Tester les commandes suivantes dans la console Python et expliquer les résultats obtenus :

```
>>> import sys #on importe le module système
>>> sys.float_info #les infos systèmes sur les flottants
>>> 1+2**(-52) == 1; 1+2**(-53) == 1
>>> 2+2**(-51) == 2 ; 2+2**(-52) == 2
>>> 2**(1023)*(2-2**(-52)); 2**(1023)*(2-2**(-53))
>>> 2.0**1024; 2**1024
>>> 2**(-1022)*2**(-52);2**(-1022)*2**(-53)
>>> 2.**(2**10-1) ; 2.**(2**10)
```

console et flottants

6. Pour déterminer la représentation décimale d'un flottant dont on donne la représentation binaire, il faut d'abord identifier de gauche à droite, les bits correspondant à s, e et m. On peut alors reconnaître une valeur particulière ou décoder un flottant normalisé en adaptant la formule (1).
   Déterminer les flottants codés par les mots :
   a. 11000100011010010011110000111000000000000000000000000000000000000
   b. 00010000001111010011100101011000000000000000000000000000000000000
7. Représenter en binaire selon la norme IEEE754 le fottant 6,625.(Voir pages 26 et 27 du livre « Technologie des ordinateurs et des réseaux »).

Exercice 19

1. Tester le programme suivant et expliquer le traitement effectué par la procédure `mystère`.

```
#-*- coding: Utf-8 -*

def mystere():
    x = 1.
    y = x+1.
    i = 0
    while (y-x == 1.):
        i += 1
        x = x*2.
        y = x+1.
    print(i,x,y)

if __name__ == "__main__":
    mystere()
```

mystère

2. On vérifiera qu'en Python, les flottants sont notés sous forme décimale jusqu'à 2^{54} non compris, puis en notation scientifique.

3.1 Problèmes d'approximation avec les flottants

Exercice 20

Partie A : Erreur dans le test d'égalité de deux flottants

A cause de la représentation machine approchée, il n'est en général pas raisonnable de tester l'égalité de deux flottants.
On peut avoir de mauvaises surprises si un test d'égalité de deux flottants doit décider de l'arrêt.
On donne ci-dessous deux exemples où celà se passe mal.
Dans le premier on part du triplet Pythagoricien $(a,b,c) = (5,4,3)$ vérifiant $a^2 = b^2 + c^2$ et on divise tous les éléments du triplet par 30. On devrait avoir encore un triplet Pythagoricien ...
Dans le second on lance une boucle `while` en prévoyant dix itérations et on bloque la machine dans une boucle infinie!!!

```
>>> a,b,c = 5,4,3
>>> a**2 == b**2+c**2
True
>>> a,b,c = a/30,b/30,c/30
>>> a**2 == b**2+c**2
False
>>> i = 0
>>> while i!=1:
...     i += 0.1
```

Pour éviter ce genre de désagrément, on peut remplacer le test d'égalité par un test avec une inégalité.
Pour tester si $f_1 == f_2$ on peut écrire $|f_1 - f_2| < \varepsilon$ où ε est l'erreur choisie.

Partie B : Perte de précision par soustraction de nombres très proches (ou cancellation)

Tester les instructions suivantes dans la console.
Que semble-t-il se passer lorsqu'on retranche des nombres de plus en plus proches ?

```
>>> 0.1-0.0999999
>>> 0.1-0.0999999999999 # douze '9'
>>> 0.1-0.09999999999999999 # seize '9'
>>> 0.1-0.099999999999999999 # dix-sept '9'
```

Partie C : Perte de précision par somme, avec absorption

1. Tester les calculs suivants en console et interpréter les résultats.

```
>>> 2**(-53)+2**(-53)+1
>>> 1+2**(-53)+2**(-53)
>>> 1+2**(-53)+2**(-53) == 2**(-53)+2**(-53)+1
```

2. Les fonctions `somme1` et `somme2` ci-contre qui calculent $\sum_{k=1}^{n} \frac{1}{k^4}$.

Dans `somme2`, on applique le principe de la photo de classe (on commence par sommer les petits).

Tester les fonctions pour les entrées $n = 10\,000$ et $n = 100\,000$. On donne $\lim_{n\to+\infty} \sum_{k=1}^{n} \frac{1}{k^4} = \frac{\pi^4}{90}$, quelle méthode de calcul semble la plus précise ? Donner une interprétation.

```
def somme1(n):
    s = 0
    for i in range(1,n+1):
        s += 1/i**4
    return s


def somme2(n):
    s = 0
    for i in range(n+1,0,-1):
        s += 1/i**4
    return s
```

cancellation

Exercice 21

1. Tester le programme `recurrence`. Quels sont les résultats attendus, interpréter les résultats affichés par l'ordinateur.

2. Tester le programme `boucle` ci-dessous, qui se propose de calculer $\sum_{k=1}^{10} f\left(1+\frac{k}{10}\right)$ avec $f(x)=x^2$. Que se passe-t-il ?

   Et si on remplace le test `a != 2` par `a<=2` ?

```
def f(x):
    return x**2

a, pas, s = 1.1, 0.1, 0
while a != 2:
    print(a)
    s = s+f(a)
    a += pas
print(s)
```

boucle

```
def recurrence(n):
    u0 = 1/3
    u1 = 0
    for i in range(n) :
        u1 = 4*u0 - 1
        u0 = u1
    return u1

#programme principal
print('4/3 - 1 == 1/3 est',4/3 - 1
      == 1/3)
for i in range(1,50):
    res = recurrence(i)
    print(('recurrence(%s)=%.8f')%(
          i,res))
```

recurrence

Table des matières