Représentation des nombres

C. Charignon

Il y a 10 sortes de personnes : ceux qui comprennent la base 2 et les autres.

Anonyme

On présente dans ce chapitre les principales manières dont sont enregistrés en mémoire les nombres. Il est important de retenir les limitations que ceci apporte, en premier lieu les problèmes d'erreurs d'arrondis.

La mémoire d'une ordinateur est composée d'un grand nombre de circuits électriques pouvant être dans deux états différents. Un de ces états sera appelé « 0 » et l'autre « 1 ». Ainsi, toute information dans un ordinateur est enregistrée comme une suite de zéros et de uns. Le but de ce chapitre est de voir succintement comment on peut représenter un nombre par une suite de zéros et de uns.

Remarquons tout de suite que la mémoire d'un ordinateur est finie, alors que le nombre de nombres est infini : quelque soit la méthode employée, il y aura toujours des nombres qu'il sera impossible d'enregistrer ! Les nombres pouvant être exactement enregistrés en mémoire sont même plutôt rares en proportion.

Les processeurs actuels sont souvent en sur 64 bits. Cela signifie que chaque nombre est codé sur 64 bits, c'est-à-dire par une suite de 64 zéros ou uns. Ceci laisse donc 2^{64} nombres différents possibles.
Remarque : En Caml, pour des raisons techniques, seuls 63 bits sont utilisés pour codes les entiers.

Table des matières

Première partie
Cours

1 Entiers

Dieu a créé les nombres entiers, tout le reste est l'œuvre de l'homme.

Leopold Kronecker

1.1 Base 2

1.1.1 Principe

Définition 1.1. *Soit $b \in \mathbb{N}$ tel que $b \geq 2$. Soit $n \in \mathbb{N}$ et $(a_1, ..., a_n) \in [\![0, b-1]\!]^{n+1}$. On note alors*

$$(a_n a_{n-1} ... a_1 a_0)_b = \sum_{k=0}^{n} a_k b^k$$

On dit que l'écriture $(a_n a_{n-1} ... a_1 a_0)_b$ est l'écriture de $\sum_{k=0}^{n} a_k b^k$ en base b.

Une deuxième notation existe : $\overline{a_n \, ... \, a_0}^b$.

On peut démontrer que pour tout $N \in \mathbb{N}$, il existe un unique $n \in \mathbb{N}$, et un unique n-uplet $(b_0, ..., b_n) \in [\![0, b-1]\!]^{n+1}$ tel que $N = (a_n a_{n-1} ... a_1 a_0)_b$ et $a_n \neq 0$. Autrement dit, l'écriture en base b de N existe et est unique.

Nous utiliserons essentiellement la base 2, donc en général on prendra $b = 2$ et les chiffres possibles seront 0 et 1. Cependant en informatique, les bases 8 ou 16 (hexadécimal) sont également utilisées, car un nombre écrit en base 8 ou 16 peut facilement être écrit en base 2.

Pour la base 16, les chiffres plus grand que 9 sont notés A,B,C,D,E,F. Par exemple $(A108C)_{16}$ signifie $12 + 8 \cdot 16 + 1 \cdot 16^3 + 10 \cdot 16^4$.

1.1.2 Remarques et commentaires

En Python, on peut rentrer un nombre entier en le tapant directement en base 2, 8, ou 16 avec la syntaxe suivante :

- *base 2 :* taper `0bmon_nombre_en_base_2` (« b » comme « binaire »)
- *base 8 :* taper `0mon_nombre_en_base_2`
- *base 16 :* taper `0xmon_nombre_en_base_2` (« x » comme « hexadécimal »)

En résumé : quand on commence un nombre par 0, Python comprend qu'il n'est pas en base 10. Si on indique ensuite un b (comme binaire) c'est en base 2, un x (comme hexadécimal) c'est en base 16, et rien, c'est en base 8.

Les ordinateurs actuels fonctionnent en général sur 64 bits, ce qui signifie que le processeur est capable de faire les opérations élémentaires pour des nombres codés à l'aide de 64 bits. Si nous codions simplement chaque chiffre à l'aide d'un bit, le plus grand nombre possible est :

$$(111...1)_2 = \sum_{k=0}^{63} 1 \cdot 2^i = \frac{1 - 2^{64}}{1 - 2} = 2^{64} - 1$$

De manière plus générale, à l'aide de n bit, on peut coder en base 2 les entiers de 0 à $2^n - 1$.

Mais ce n'est pas la manière retenue par Python car elle ne permet pas de coder les entiers négatifs !

Attention aux dépassements de mémoire ! Si on effectue $(2^{64} - 1) + 1$ (c'est-à-dire en base 2 $(1...1)_2 + (0...01)_2$), on va obtenir 0 ! En effet, on aurait du obtenir $(10...0)_2$ (65 chiffres) mais comme la mémoire est limitée à 64 bits, le 65-ème disparaît !

Mais Python, depuis sa version 3, contourne ce problème : lorsqu'un entier nécessite plus que 64 bits, il va être découpé en plusieurs blocs de 64 bits.

1.1.3 Conversion base 2 / base 10

La formule de la définition permet aisément de convertir un nombre écrit en base b vers la base 10 habituelle. Voyons comment effectuer l'opération réciproque.

Soit donc $N \in \mathbb{N}$ et $(a_n....a_0)_2$ son écriture en base 2, ce qui signifie que $N = \sum_{k=0}^{2} a_i 2^i$.

Déjà, l'écriture

$$N = a_0 + 2\sum_{k=1}^{n} a_i 2^{i-1}$$

montre que a_0 est le reste de la division euclidienne de N par 2. Et le quotient est $q = \sum_{k=1}^{n} a_i 2^{i-1}$.

Maintenant, q peut aussi s'écrire $q = \sum_{k=0}^{n-1} a_{i+1} 2^i = (a_n...a_1)_2$.

Mais alors on peut obtenir a_1 facilement : c'est le reste de la division euclidienne de q par 2. Et le quotient sera $(a_n...a_2)_2$.

Et ainsi de suite...

Notez qu'on obtient les chiffres formant N "à l'envers" : a_0 en premier et a_n en dernier. Si on veut écrire un algorithme calculant l'écriture de n en base 2, et si on veut afficher le résultat dans l'ordre de lecture habituel, il faudra retourner le résultat à la fin.

On peut donner l'algorithme suivant :

Entrées : une liste L
Sorties : la liste contenant les éléments de L dans l'ordre inverse
Variables locales : R

```
1 début
2 |  R ← []
3 |  n ← longueur de L pour i de 1 à n :
4 |  |  rajouter L[n − i] à la fin de R
5 |  fin
6 |  renvoyer(R)
7 fin
```

Algorithme 1 : retourne

Entrées : n entier, écrit en base 10
Sorties : la liste des chiffres pour écrire n en base 2
Variables locales : q,r entier, L liste

```
1 début
2 |  q ← n
3 |  tant que q ≠ 0 :
4 |  |  q, r ← quotient et reste de la division euclidienne de q par 2
5 |  |  rajouter r à la fin de L.
6 |  fin
7 |  renvoyer ( retourne(L))
8 fin
```

Algorithme 2 : écriture en base 2

N.B. En python, pour obtenir en même temps le quotient et le reste de la division euclidienne de q par 2, utiliser `divmod(q,2)`.

N.B. Bien sûr, on peut remplacer 2 par n'importe quel nombre $b \in [\![2, \infty[\![$ pour obtenir un algorithme de conversion en base b.

1.2 Nombres relatifs

Pour coder un entier éventuellement négatif, l'idée naïve est d'utiliser un bit pour coder le signe. Par exemple, -14 serait codé, en décidant d'utiliser 8 bits et de garder le premier pour le signe, par `10001110`.

En 64 bits, par cette méthode, il resterait 63 bits pour coder $|N|$, et on pourrait coder les nombres entre $-(2^{63}-1)$ et $2^{63}-1$. Une remarque : 0 aurait deux codages : `00...0` mais aussi `10...0` (car $0 = -0$).

Mais il existe une méthode plus pratique, qui va simplifier les additions. Cette méthode, appelée « complément à deux » n'est pas au programme, même si j'en donne une description dans ce document.

Signalons qu'à l'aide de 64 bits, il est possible de coder 2^{63} nombres différents. Par exemple on pourrait coder l'intervalle $[\![2^{63}-1, 2^{63}[\![$. Mais il se trouve que l'intervalle codé par la méthode du complément à deux est $[\![-2^{63}, 2^{63}-1[\![$.

1.3 Complément à deux (hors programme)

- Un entier positif $N \in [\![0, 2^{63}-1]\!]$ sera codé par son écriture en base 2 en 63 chiffres, précédée d'un 0.
- Un entier négatif $N \in [\![-2^{63}, -1]\!]$, sera codé par l'écriture en base 2 de $2^{64}+N$. On a $2^{64}+N \in [\![2^{64}-2^{63}, 2^{64}-1]\!] = [\![2^{63}, 2^{64}-1]\!] = [\![(100...0)_2, (111...1)_2]\!]$. Ainsi ce codage commencera par un 1, et il est possible de distinguer le codage d'un nombre négatif de celui d'un nombre positif.

Remarque : On peut coder tous les éléments de $[\![-2^{63}, 2^{63}-1]\!]$: il se trouve qu'il y a un nombre négatif de plus que de nombres positifs.

1.3.1 Intérêt pour les additions

En quoi ceci est-il pratique pour effectuer des additions ? En fait, le même circuit qui calcule les additions de nombres positifs va marcher également pour les nombre négatifs !

En effet, soit $(N, M) \in [\![-2^{63}, 2^{63}-1]\!]^2$. On les code par complément à deux et on envoie les deux codages au circuit d'addition.

- Si $N \geqslant 0$ et $M \geqslant 0$, on récupère $N+M$ sans problème.
- Si $N \geqslant 0$ et $M < 0$: on a en fait envoyé au circuit N et $2^{64}+M$. Celui-ci va donc renvoyer $N+2^{64}+M$.
  - ◇ Si le résultat $N+M<0$: ceci est bien le codage de $N+M$ en complément à 2, le résultat correct est renvoyé.
  - ◇ Si $N+M \geqslant 0$, il aurait fallu obtenir $N+M$. Or, $N+M+2^{64} \in [\![2^{64}, 2^{64}+2^{63}-2]\!]$.
    Donc le codage en base 2 de $N+M+2^{64}$ est $10a_{62}...a_0$, où $(a_{62}...a_0)_2 = N+M$. Le 1 est en 65-ème position. Mais l'ordinateur fonctionne en 64 bits... le 65 bit est juste oublié ! Et le résultat renvoyé sera $(0a_{62}...a_0)$, qui est bien le codage de $N+M$.
- Autres cas similaires.

1.3.2 Comment coder et décoder un entier en complément à 2 ?

- Lorsque $N \geq 0$, on garde son écriture en base 2 sur 63 bits, précédée d'un zéro.
- Lorsque $N < 0$:
  - ◇ On calcule $|N|$ en base 2, sur 64 bits.
  - ◇ On inverse tous les bits : on obtient alors $\overline{\underbrace{11...1}_{64 \text{ uns}}}^{\,2} - |N|$.
  - ◇ On rajoute 1. On obtient alors $1 + \overline{\underbrace{11...1}_{64 \text{ uns}}}^{\,2} - |N| = \overline{1\underbrace{00...0}_{64 \text{ zéros}}}^{2} + N = 2^{64} + N$.

Et donc pour décoder un nombre écrit en complément à 2 :

- si l'écriture commence par un zéro, c'est un nombre positif écrit simplement en base 2 ;
- sinon : retrancher 1 puis inverser les bits. La suite de bits obtenue est alors l'écriture en base 2 de $-N$.

1.4 Une remarque en Python ≥ 3

À partir de python 3, les entiers peuvent aussi grand que nécessaire. Un entier plus grand que 2^{63} sera codé en deux (ou plus) paquets de 64 bits. Le dépassement de mémoire n'arrive donc pas lorsqu'on utilise des entiers avec python≥ 3. (Sauf si la mémoire totale de l'ordinateur est remplie, ce qui est un autre problème.)

Cependant, le temps de calcul va augmenter lorsque le nombre de paquets de bits utilisé augmente.

1.5 Exemple : multiplication égyptienne

cf exercice : 6

Soient $(N, M) \in \mathbb{N}^2$. Voici la méthode que les égyptiens utilisaient pour calculer $N \times M$.

On décompose un des deux nombres en base 2. En général le plus petit des deux, disons M ici. Soit $(a_n...a_0)_2$ l'écriture en base de 2 de M.

Alors :

$$N \times M = N \times \sum_{i=0}^{n} a_i 2^i = \sum_{i=0}^{n} a_i N 2^i.$$

Nous allons alors calculer successivement tous les $N2^i$, et nous les rajouterons dans une variable de résultat à chaque fois que $a_i = 1$. (Les a_i étant calculés comme dans notre algorithme de décomposition en base 2.)

Ceci donne en Python :

```
def mult(n,m):
    """multiplie n par m par la méthode égyptienne."""

    if n<m: #Pour coller le plus possible à la méthode, c'est le plus petit des deux nombres
    # qu'on décompose en base 2.
        return( mult(m,n))

    R=0
    nFoisPuissanceDe2=n
    q=m

    while q!=0:
        r=q%2

        if r==1:
            R+=nFoisPuissanceDe2
        q=q//2
        nFoisPuissanceDe2*=2
    return(R)
```

FIGURE 1 – Le papyrus Rhind qui décrit entre autre la méthode de multiplication égyptienne.

1.6 Bonus : opérations bit à bit

Il existe des commandes Python qui agissent directement sur les bits de la représentation d'un nombre entier. L'intérêt est que ces opérations sont en général câblées directement dans le processeur, leur exécution est donc très rapide.

- `&` effectue un « et »
- `|` effectue un « ou »
- `^` effectue un « ou exclusif »
- `~` effectue un « non », c'est-à-dire inverse tous les bits
- `a << b` décale les bits de `a` de n cases vers la gauche, ce qui revient à multiplier par 2^b.
- `a >> b` décale les bits de `a` de n cases vers la droite, ce qui revient à diviser par 2^b.

cf exercice : 5

2 Flottants

Les flottants ne sont pas du tout codés de la même manière que les entiers. Au passage, cela signifie que les conversions d'un flottant vers un entier ou réciproquement (les fonctions `int` et `float`) sont relativement complexes : on aura donc intérêt à les éviter autant que possible.

2.1 La norme IEEE 754

2.1.1 Principe général

Déjà, l'écriture en base 2 s'étend naturellement aux nombres à virgule : Pour tout $(n, m) \in \mathbb{N}^2$ et $(a_n, ..., a_0, a_{-1}, ..., a_{-m}) \in \{0, 1\}^{n+m+1}$, on définit :

$$(a_n...a_0, a_{-1}...a_{-m})_2 = \sum_{k=-m}^{n} a_i 2^i$$

Soit $x \in \mathbb{R}$. Pour encoder x, on utilise «la notation scientifique», mais en base 2. On écrit x sous la forme :

$$x = \pm(1, m_1 \dots m_{52})_2 \times 2^e,$$

$(1, m_1 \dots m_{52})_2$ s'appelle alors la « mantisse », et e « l'exposant ». Notez qu'on utilise en général 52 bits pour la mantisse. On décide que le nombre avant la virgule sera toujours 1 : ceci évite d'avoir à utiliser un bit pour l'enregistrer (0. sera codé par un cas particulier).

Le signe est codé ainsi : 0 pour + et 1 pour −. Notons s ce bit de signe.

Comment code-t-on e ? Si nous disposons de 64 bits pour coder x, il nous reste 11 bits pour e. On pourrait coder les entiers de $[\![-2^{10}, 2^{10} - 1]\!]$. Mais non !

Déjà, on se limite à $e \in [\![-1022, 1023]\!]$, et on encode par l'écriture en base 2 de $e + 1023$. On a $e + 1023 \in [\![1, 2046]\!]$, donc l'écriture en base 2 est entre $(0000000001)_2$ et $(1111111110)_2$.

Cela permet de simplifier les comparaisons, au détriment des additions (sachant qu'une addition des exposants intervient pour la multiplication des flottants).

Il reste deux codages inutilisés : $(0000000000)_2$ et $(11111111111)_2$. Ceux-ci serviront pour les cas particuliers : 0, $+\infty$, $-\infty$ et « NAN » (voir plus bas, mais c'est hors programme).

Notons ci-dessous $e_1...e_{11}$ le codage de l'exposant.

Alors le codage enregistré sera :

$$s\ e_1...e_{11}\ m_1...m_{52}.$$

Remarque : Pour comparer deux flottants, il suffit d'utiliser l'ordre lexicographique sur les bits.

Exemple : Quel nombre représente le codage 1 101...0 0 ... 11001 ?

- le signe est −
- la mantisse est $(1, 101....0)_2$, donc $1 + \frac{1}{2} + \frac{1}{8}$.
- l'exposant est $(0 \dots 11001)_2 - 1023 = 25 - 1023 = -998$.

En conclusion, le nombre codé est $+\left(1 + \frac{1}{2} + \frac{1}{8}\right) \times 2^{-998}$.

2.1.2 Cas particuliers (hors programme)

- 0 ne peut pas s'écrire sous la forme $\pm 1, m_1...m_{52}^e$! On convient de le coder par 00...0. (D'où l'intérêt d'avoir gardé libre le codage 0000000000 pour l'exposant.
  En fait, 1000....0 représente 0^-, alors 000...0 est 0^+. L'ordinateur utilisera les règles usuelles de calcul de limites. Par exemple $1/0^+ = +\infty$.
- $+\infty$ est codé par 0 11111111111 000...0 : les bits d'exposant sont tous à 1, ceux de mantisse à 0. On arrive à $+\infty$ dès qu'on dépasse le plus grand nombre pouvant être codé par le cas général, en l'occurrence $1, \underbrace{1...1}_{52\text{uns}} \times 2^{1023}$.
- $-\infty$ est codé par 1 11111111111 000...0.
  Remarque : D'où l'intérêt d'avoir laissé libre le codage 1111111111 au niveau de l'exposant.

- Les combinaison de type "(0 ou 1) 1111111111 *suite_non_nulle_de_52_bits* » donnent ce qui s'appelle des « NAN » (not a number).
  On obtient un NAN lors d'un calcul non défini. Par exemple $+\infty - \infty$.

Essayer les commandes suivantes :

```
x=2.**1023
x=x*2
print(x)
print(1/x)
print(x*(1/x))
```

Remarque : Pour comparer deux nombres flottants, le même algorithme, et donc le même circuit du processeur, que pour les entiers positifs. On regarde d'abord le bit le plus à gauche, puis en cas d'égalité on passe un cran à droite, etc...
Remarque : Les informations sur la représentation des flottants (taille de la mantisse, plus grand nombre représentable...) peux être obtenu par la commande `sys_info` de la bibliothèque `sys`.

2.1.3 Comment calculer le codage d'un nombre ?

Exemple : Quelle est la représentation de $x = 0,375$?
on commence par mettre sous la forme $1,... \times 2^{...}$:

- $2.x = 0,75$
- $4.x = 1,5$. Donc $x = 1,5.2^2$.

Maintenant, il reste à convertir $1,5$ en base 2, et à donner le codage de l'exposant, qui est 2.

- Pour l'exposant : on rajoute 1023, et on écrit en base 2 : c'est donc 1025. Or $1025 = 1024 + 1 = 2^{10} + 1 = (01000000001)_2$.
- Pour la mantisse : on cherche $m_1...m_{52}$ tels que $1,5 = (1, m_1...m_{52})_2$, c'est-à-dire $0,5 = (0, m_1...m_{52})_2$.
  Pour récupérer le 2 : il suffit de multiplier par 2 :
  $1, = (m_1, m_2...m_{52})_2$. Donc $m_1 = 1$ et les autres sont nuls.

On peut utiliser l'algorithme 3
Attention : on n'obtient de la sorte une flottant qui *approche* seulement le réel fourni en entrée.
exemple : quelle est la représentation de 0.1 ?

2.2 Exemples de problèmes rencontrés

2.2.1 Dépassement de capacité

Un nombre plus grand que 2^{1024} sera transformé en $+\infty$, un nombre inférieur à -2^{1024} devient $-\infty$, un nombre dans l'intervalle $\left[0, 2^{-1023}\right[$ sera transformé en 0^+, un nombre dans $]2^{-1023}, 0[$ devient 0^-. Les opérations suivent alors les règles sur les calculs de limite. Mais en cas de forme indéterminée, le flottant renvoyé sera NAN : not a number.

cf exercice : 10
Remarque : Concernant le 0^+ et le 0^-, Python s'arrange pour améliorer un peu... Il utiliser ce qu'on appelle des nombres « dénormalisés », ce qui signifie qu'il ne suivent plus tout à fait la norme IEEE 754.

2.2.2 L'égalité n'existe pas avec des flottants

Étudions l'exemple suivant :

```
1 .1+.1==.2
2
3 .1+.1+.1==.3
```

Et oui, les erreurs d'arrondi rendent les test de type `x==y` inefficaces. Concrètement, ne jamais utiliser un test d'égalité avec des flottants ! On les remplacera par des tests de type `abs(x-y) < eps`, en choisissant pour `eps` un nombre de l'ordre de grandeur des erreurs d'arrondi attendues...

Remarque : Pour afficher toutes les décimales d'un nombre, on peut utiliser la commande `Decimal` de la bibliothèque `decimal`. Essayez donc le code :

Entrées : un réel x
Sorties : le codage de x, sous forme d'une liste de 0 et 1
Variables locales : L : liste qui va contenir le résultat, y : réel, e et i : entiers

début
  $L \leftarrow []$
  # le signe
  si $x \geq 0$ **:**
    rajouter 0 dans L
    $y \leftarrow x$
  sinon :
    rajouter 1 dans L
    $y \leftarrow -x$
  fin

  # Calcul de l'exposant :
  $e \leftarrow 0$
  tant que $y \leq 2$ **:**
    $y \leftarrow y/2$
    $e \leftarrow e - 1$
  fin
  tant que $y < 1$ **:**
    $y \leftarrow y * 2$
    $e \leftarrow e + 1$
  fin
  # Maintenant, $y \in [1, 2[$ et e contient l'exposant.
  $L \leftarrow L+$ convertionEnBase2$(e + 1023)$

  } En avant pour la mantisse :

  $y \leftarrow y - 1$
  pour *i de 1 à 52* **:**
    $y \leftarrow y * 2$
    si $y \geq 1$ **:**
      rajouter 1 dans L
      $y \leftarrow y - 1$
    sinon :
      rajouter 0 dans L
    fin
  fin
  renvoyer L
fin

Algorithme 3 : codage d'un flottant selon la norme IEEE 754

```
import decimal
decimal.Decimal(.1)
```

cf exercice : 8

2.2.3 Erreurs d'arrondi

Exemple : Calculons $1 + 2^{-53} - 1$. Essayons `(1+2**(-53)) - 1` puis `(1-1) + 2**53`.

Que se passe-t-il dans le premier cas ? On commence par le calcul de `1+2^{-53}`. L'exposant du résultat sera 0. Par conséquent, il faut écrire 2^{-53} en « écriture scientifique en base 2 avec exposant 0 », ceci donne :

$$2^{-53} = 0,\ \underbrace{0...0}_{52 \text{ zéros}}\ 1 \times 2^0$$

Le 1 arrive à la 53-ième place après la virgule. Mais nous n'enregistrons que 52 bits de mantisse ! Ce 53-ième chiffre est

alors purement et simplement effacé. Et donc 2^{-53} est assimilé à 0. Le résultat de l'opération `1.0 + 2**-53` est donc `1.0`.

Intuitivement, on peut considérer que 2^{-53} est négligeable devant 1. On peut donc considérer que $1 + 2^{-52}$ est équivalent à 1. Et lors du calcul de `(1+2**(-53)) - 1`, on a somnmé des équivalents !

De manière générale, il faut éviter autant que possible de sommer des termes qui n'ont pas le même ordre de grandeur.

cf exercice : ??, 12

Exemple : Calcul d'une série : mieux vaut commencer par les plus petits termes.

Deuxième partie

Exercices

Exercices : représentation des nombres

1 Entiers

Exercice 1. **! Opérations élémentaires sur les entiers

Le but de cet exercice est de simuler la gestion des entiers par un ordinateur. Un nombre sera représenté par un tableau de 64 cases contenant chacune 0 ou 1. On écrira les bits de gauche à droite, comme dans l'écriture mathématique habituelle.

1. Écrire une fonction `addition` qui additionne deux nombres écrits en base deux. On pourra utiliser une variable locale `retenue`.
2. Écrire une fonction `multiplication` qui effectue le produit de deux entiers. On pourra écrire une fonction intermédiaire chargée de décaler les bits vers la gauche (en mettant des 0 à droite).
3. Écrire une fonction `compare` qui prend deux entier et indique si le premier est plus petit que le second.

Exercice 2. ** Écriture en base 2, version récursive

On va écrire une fonction `enBase2` pour calculer l'écriture d'un entier en base 2.

Soit $n \in \mathbb{N}$ et $(a_k \dots a_0)_2$ son écriture en base 2. On rappelle la méthode de calcul de l'écriture en base 2 de n : déjà `a0=n%2`. Puis pour obtenir la suite, on « recommence avec `n//2` ».

On peut traduire le « on recommence avec `n//2` » par un réappel de la même fonction, c'est-à-dire par `enBase2(n//2)`.

Écrire la fonction utilisant cette idée.

Exercice 3. *! Place mémoire occupée

Un randonneur part traverser les Pyrénées. La traversée lui prend les mois de juillet et août. Chaque jour il prend 15 photos. L'appareil enregistre les photos au format 5000×3000 pixels. Chaque pixel est enregistré comme trois entiers positifs codés sur 8 bits (un pour chaque couleur élémentaire : rouge, vert, et bleu).

Sa carte mémoire de 16Go sera-t-elle suffisante ?

Remarque : $1\,\text{Go} = 1 \times 10^9$ o. Ne pas confondre avec le Gio qui vaut 1024^3 octets.

Exercice 4. * Nombre de bits pour écrire un entier

Soit $n \in \mathbb{N}$. Montrer que le nombre de chiffres pour écrire n en base 2 est $1 + \left\lfloor \log_2 n \right\rfloor$.

Exercice 5. * Codage d'ensembles**

On va coder les sous-ensembles de $[\![0, 63[\![$ [1] de la manière suivante : pour tout $X \in \mathcal{P}([\![0, 63[\![)$, on note n_X le nombre tel que pour tout $i \in [\![0, 63[\![$, le bit i de n_X vaut 1 ssi $i \in X$.

1. Calculer n_X dans le cas où $X = \emptyset$, $X = \{1, 4, 6\}$ puis $X = [\![0, 64[\![$.
2. En utilisant les opérations bit à bit de Python, écrire en une ligne une fonction qui calcule la réunion, puis l'intersection de deux ensembles décrits comme ci-dessus par des nombres entiers. Puis une fonction pour le complémentaire dans $[\![0, 64[\![$.
3. Écrire également une fonction pour tester si un élément appartient à un ensemble, puis pour ajouter un élément à un ensemble.

   Voici la syntaxe de quelques opérations bit à bit en Python :
   (a) `|` est le « ou » bit à bit. Par exemple, $3|5 = \overline{011}^2|\overline{101}^2 = \overline{111}^2 = 7$.
   (b) `&` est le « et » bit à bit. Par exemple $3\&5 = \overline{011}^2\&\overline{101}^2 = \overline{001}^2 = 1$.
   (c) $\sim$ est le « non » bit à bit. Par exemple ~ 3 est le nombre dont l'écriture sur 64 bits est $111 \dots 100$.
   (d) `<<` permet de décaler les bits vers la gauche et `>>` vers la droite. Par exemple `2 << 3` renvoie `16`, et `2 >> 1` renvoie `1`.
4. Écrire les fonctions de conversion pour transformer un ensemble en entier et réciproquement.

Exercice 6. ** Multiplication égyptienne

Soient $(N, M) \in \mathbb{N}^2$. Voici la méthode que les égyptiens utilisaient pour calculer $N \times M$.

On décompose un des deux nombres en base 2. En général le plus petit des deux, disons M ici. Soit $(a_n \dots a_0)_2$ l'écriture en base de 2 de M.

Alors :

$$N \times M = N \times \sum_{i=0}^{n} a_i 2^i = \sum_{i=0}^{n} a_i N 2^i.$$

Pour obtenir un programme efficace, on utilise une variable qui contiendra, pour tout $i \in [\![0, n]\!]$, $N2^i$ à l'étape i. En outre, les a_i sont les chiffres pour écrire M en base 2, on les calcule comme d'habitude.

1. Et pas $[\![0, 64[\![$ car dans Python un des 64 bits sert aux nombres négatifs qui ne sont pas au programme.

1. Programmer cette méthode.
2. On peut utiliser une opération bit à bit comme dans l'exercice précédent : multiplier un entier x par 2 revient juste à décaler tous les bits d'une case vers la gauche, en rajoutant un zéro à droite. En Python, on peut utiliser l'expression `x << 1`. (Plus généralement, pour tout $i \in \mathbb{N}$, `x << 1` renvoie le nombre obtenu en décalant de i bits vers la gauche les bits qui forment x.

Exercice 7. ** Persistance d'un entier

Pour tout $n \in \mathbb{N}$, nous noterons $p(n)$ le produit des chiffres de n (en base 10).

1. Programmer la fonction p.
2. Démontrer que pour tout $n \in \mathbb{N}$, le nombre de chiffres de $p(n)$ est strictement inférieur au nombre de chiffres de n.
3. On appelle persistance d'un entier n le plus petit entier k tel que $p^k(n)$ n'a plus qu'un seul chiffre. Écrire une fonction qui calcule la persistance d'un entier.
4. Une conjecture prétend que la persistance d'un entier est toujours inférieure à 11. Écrire une fonction prenant en entrée un entier N et qui vérifie cette conjecture sur $[\![0, N[\![$.
5. (***) On peut augmenter la rapidité du programme précédent en gardant en mémoire la persistance de tous les entiers déjà traités.

2 Flottants

Exercice 8. *! Équation de degré 2

1. Retrouver ou réécrire une fonction `secondDegre(a,b,c)` pour calculer les solutions réelles de l'équation $ax^2 + bx + c = 0$ d'inconnue $x \in \mathbb{R}$.
2. Résoudre les équations : $x^2 + 0{,}2x + 0{,}01 = 0$ puis $x^2 + 1{,}4x + 0{,}49 = 0$. Expliquer le résultat. Proposer une amélioration de la fonction pour éviter ces problèmes.

Exercice 9. * Associativité et autres propriétés algébriques

L'ensemble des flottants forme-t-il un corps pour les opérations habituelles ? Trouver un contre-exemple pour chaque propriété qui n'est pas vérifiée.

Exercice 10. *! Pour déterminer le nombre de bits utilisé par le système

1. Essayer le programme suivant :

```
def fonctionMystère():
    x=1.
    c=0
    while x!=x+1.:
        x*=2.
        c+=1
    return(c)
```

   Que renvoie-il ? Expliquer. Que vaut x à la fin ?
2. De même, trouver un programme permettant d'afficher l'exposant maximal d'un flottant.

Exercice 11. * Plus grand et plus petit flottant

Quels sont :

1. le plus petit flottant > 0 ?
2. le plus grand flottant non infini ?
3. Quel est le plus grand flottant x tel que $1 + x = 1$?

Exercice 12. ** Calcul de somme

Pour tout $n \in \mathbb{N}$, on pose $H_n = \sum_{k=1}^{n} \frac{1}{k}$. La suite H s'appelle la série harmonique.

1. Écrire deux fonctions pour calculer H_n. L'une sommera les terme de 1 jusqu'à n, l'autre de n jusqu'à 1.
2. Comparer les résultats. Laquelle des deux est la plus précise ?

Remarque : Cette somme est obtenue par exemple quand on approche $\int_1^{n+1} \frac{1}{t} dt$ par la méthode des rectangle. Elle est donc proche de $\ln(n+1)$.

Exercice 13. * Suite de Müller et Kahan**

On définit la suite u par : $\begin{cases} u_0, u_1 = \frac{11}{2}, \frac{61}{11} \\ \forall n \in \mathbb{N},\ u_{n+2} = 111 - \dfrac{1130 - \frac{3000}{u_n}}{u_{n+1}} \end{cases}$.

1. Supposons que cette suite converge vers une limite ℓ. Montrer que $\ell \neq 0$. Puis montrer qu'il n'y a que trois valeurs possibles pour ℓ que l'on précisera.
2. Programmer cette suite, et tester pour de grandes valeurs de n.

Quelques indications

7 1) Il s'agit de décomposer en base 10. La méthode est la même que pour décomposer en base 2.

9

11 1.

2.

3. Écrire x et 1 sous leur forme « scientifique en base 2 ».

13 1. Partir du fait que $\lim_{n\to\infty} u_{n+1} = \ell$ et $\lim_{n\to\infty} u_{n+2} = \ell$ (suites extraites).
On obtiendra une équation de degré 3 dont 6 est racine « évidente ».

Quelques solutions

1

2

3

4 Soit d le nombre de chiffres pour écrire n en base 2. Ainsi :

$$\begin{array}{ll} & 2^{d-1} \leqslant n < 2^n \\ \text{donc} & d-1 \leqslant \log_2 n < n \\ \text{donc} & d = \left\lfloor \log_2 n \right\rfloor + 1 \end{array}$$

5

6

7

8

9

10
1. La boucle s'arrête dès que $x = 2^{53}$. En effet, à partir de cette valeur, on a $x + 1 = x$ (en flottants.
2. Remplacer la condition de la boucle par `x!=2*x`.

11
1.
2.
3. Le nombre 1 s'écrit sous la forme $(1, \underbrace{0 \ldots 0}_{52\text{zeros}})_2 \times 2^0$.
   Lorsqu'on ajoute un nombre plus petit, il sera ramené à un exposant nul. Ainsi 2^{-53} sera mis sous la forme $(0, 0 \ldots 01)_2 \times 2^0$. Le 1 étant en 53-ième position sera éliminé !
   Au final, le plus grand flottant qui sera négligé face à 1 est $(1, 1 \ldots 1)_2 \times 2^{-53}$.

12 2) Les erreurs de calculs sont d'autant plus grandes qu'on ajoute des termes d'ordre de grandeur différent (un grand avec un petit).

À l'extrême, dès que deux nombres x et y sont tels que $x/y \geqslant 2^{53}$, on aura $x + y = x$. Ainsi, dès que $\dfrac{H_n}{1/n} > 2^{53}$ la somme des termes « à l'endroit » reste constante.

Il vaut donc mieux calculer la somme à l'envers : les petits termes seront additionnés entre eux. Alors que dans la somme "à l'endroit", la dernière addition par exemple est $H_{n+1} + \dfrac{1}{n}$, or H_{n-1} est assez grand, alors que $\dfrac{1}{n}$ est petit : l'erreur de calcul risque d'être grosse.

13 Déjà, si $\ell = 0$, on voit que $\lim_{n\to\infty} -\dfrac{1130 - \frac{3000}{u_n}}{u_{n+1}} = -\infty$, d'où contradiction. Donc $\ell \neq 0$.

Alors par passage à la limite, on trouve que $\ell^3 - 111\ell^2 + 1130\ell - 3000\ell = 0$. On factorise par $\ell - 6$: $\ell = 6$ ou $\ell^2 - 105\ell - 500 = 0$. D'où les trois valeurs possibles pour ℓ.

Mais la suite « informatique » (c'est-à-dire avec des flottants) converge vers 100...