Représentation des données

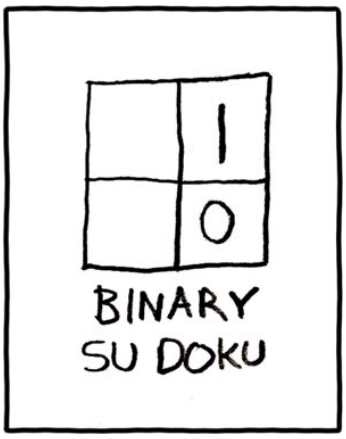

Table des matières

Un ordinateur possède une mémoire vive, appelée Ram pour « Random Access Memory ». C'est cette mémoire qui matérialise l'état du système. Concrètement, les barrettes mémoire contiennent des milliards de condensateurs qui peuvent être chargés ou déchargés. Lorsqu'un condensateur est chargé, il représente le *bit* 1. S'il est déchargé, il représente le bit 0.

condensateur	0	1	2	3	4	5	6	7	8	9	10	...
état	0	1	0	0	1	0	1	1	0	0	1	...

Une succession de 8 bits est appelée un *octet* : c'est la plus petite quantité de mémoire adressable par un ordinateur. Les quantités de mémoire se comptent en kilooctets ($1\,000 \approx 2^{10}$ octets), mégaoctets ($10^6 \approx 2^{20}$ octets), gigaoctets ($10^9 \approx 2^{30}$ octets) et téraoctets ($10^{12} \approx 2^{40}$ octets).

Dans ce chapitre, nous allons voir comment une succession de 0 et de 1 peut être utilisée pour représenter des entiers et des nombres flottants.

1 Les entiers

1.1 Décomposition en base b

L'écriture de l'entier 1984 décrit un nombre formé de 4 unités, 8 dizaines, 9 centaines et 1 millier :

$$1984 = 4 \times 10^0 + 8 \times 10^1 + 9 \times 10^2 + 1 \times 10^3.$$

Le choix de faire des paquets en utilisant des puissances de 10 est cependant arbitraire. On peut tout aussi bien décider d'utiliser des puissances de 2, 12, 16 ou 60. Si l'on choisit d'utiliser des puissances de b, on dit qu'on décompose notre nombre en base b.

Proposition 1.1

Soit $b \geqslant 2$ un entier et $w \in \mathbb{N}$. Alors, pour tout $n \in [\![0, b^w[\![$, il existe un unique w-uplet $(d_0, d_1, \dots, d_{w-1})$ d'éléments de $[\![0, b[\![$ tel que

$$n = \sum_{k=0}^{w-1} d_k b^k.$$

Remarques

- On parle de *décomposition en base b* de l'entier n. Les d_k sont appelés *chiffres* de n en base b.
- Les chiffres correspondant aux plus grandes puissances de b sont dits *plus significatifs* ou de *poids fort*. Ceux qui correspondent aux petites puissances de b sont dits *moins significatifs* ou de *poids faible*.
- Pour tout $n \in \mathbb{N}^*$, il existe un unique $w \in \mathbb{N}^*$ et un unique w-uplet $(d_0, d_1, \dots, d_{w-1})$ d'éléments de $[\![0, b[\![$ tel que $d_{w-1} \neq 0$ et $n = \sum_{k=0}^{w-1} d_k b^k$. On écrit
$$n = \underline{d_{w-1} \cdots d_1 d_0}_b.$$
Lorsqu'on parle de *la* décomposition de n en base b, c'est de cette écriture qu'il s'agit. Par exemple $13 = \underline{13}_{10}$ car $13 = 3 \times 10^0 + 1 \times 10^1$ et $13 = \underline{1101}_2$ car $13 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3$. Par convention, la décomposition de 0 est vide, quelle que soit la base.
- En base 2, les valeurs possibles pour un chiffre sont 0 et 1 ; on parlera indifféremment de *bit* ou de chiffre. Étant donné l'importance de la base 2 en informatique, il est bon de connaitre les premières puissances de 2.

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1 024

- Si la base est supérieure à dix, on a un problème pour l'écriture : il y a moins de chiffres usuels que de chiffres de la base. Le seul cas que l'on rencontre en pratique est celui de la base 16 dite *hexadécimale*. La convention est d'utiliser les lettres de `A` à `F` pour représenter les chiffres de 10 à 15.
- Les bases 2, 16 et dans une moindre mesure 8, sont couramment utilisées en informatique. Par conséquent, il est possible d'écrire les littéraux directement dans ces bases. En Python, la syntaxe est :

```
In [1]: 0b10010
Out[1]: 18

In [2]: 0xff
Out[2]: 255

In [3]: 0o77
Out[3]: 63
```

- Si l'on veut obtenir les chiffres de 137 en base 10, on commence par écrire que $137 = 13 \times 10 + 7$: on effectue la division euclidienne de 137 par 10. Le chiffre de poids faible d_0 est donc 7, et avant cela on a les chiffres de 13. De même, $13 = 1 \times 10 + 3$, donc $d_1 = 3$, et on continue en remplaçant 13 par 1. Enfin $1 = 0 \times 10 + 1$, donc $d_2 = 1$. On remplace 1 par 0 et on a terminé car 0 n'a « pas de chiffre ». On a donc $137 = \underline{137}_{10}$.
- Si au contraire on dispose de la liste des chiffres en base b et que l'on souhaite obtenir n, le plus efficace est de remarquer que
$$\sum_{k=0}^{w-1} d_k b^k = d_0 + b\left(d_1 + b\left(d_2 + \cdots b\left(d_{w-2} + b d_{w-1}\right)\right)\right).$$
Cette écriture est à la base de l'algorithme de Horner : on part de 0 et on multiplie successivement notre valeur par 10 avant de lui ajouter d_k, pour toutes les valeurs de k allant en décroissant de $w-1$ à 0. Si l'on souhaite calculer $\underline{137}_{10}$, on effectue donc les calculs
$$0 \xrightarrow[\times 10+1]{} 1 \xrightarrow[\times 10+3]{} 13 \xrightarrow[\times 10+7]{} 137.$$

Exercices 1

- Calculer $\underline{1000110}_2$ et $\underline{C7}_{16}$.
- Donner l'écriture binaire de 59 et de 31. Quel phénomène général peut-on remarquer dans le deuxième cas ?
- Comment s'écrit $\underline{102}_3$ en base 5 ?

La fonction `eval_lsd(b: int, d: list[int]) -> int` (`lsd` pour least significant digit) renvoie l'entier dont l'écriture en base b est

$$\underline{d_{w-1} \dots d_0}_b$$

en utilisant l'algorithme de Horner.

```python
def eval_lsd(b, d):
    """eval_lsd(b: int, d: list[int]) -> int"""
    w = len(d)
    n = 0
    for k in range(w):
        n = n * b + d[w - 1 - k]
    return n
```

La fonction `digits_lsd(b: int, n: int) -> list[int]` renvoie la liste des chiffres de n en base b, le chiffre le moins significatif étant en premier.

```python
def digits_lsd(b, n):
    """digits_lsd(b: int, n: int) -> list[int]"""
    d = []
    while n > 0:
        d.append(n % b)
        n = n // b
    return d
```

Ces deux fonctions sont bien entendu à connaitre sur le bout des doigts.

Proposition 1.2

Soit $b \geqslant 2$ un entier. Un entier $n > 0$ s'écrit avec $w = 1 + \lfloor \log_b(n) \rfloor$ chiffres en base b.

Les opérations d'addition et de multiplication apprises à l'école primaire fonctionnent tout aussi bien en base $b \geqslant 2$ qu'en base 10. Entrainez-vous avec les exercices suivants pour vous convaincre de cela.

Exercices 2

- ⇨ Effectuer l'addition $\underline{100110}_2 + \underline{1011}_2$ en base 2, c'est-à-dire sans jamais convertir un nombre en base 10.
- ⇨ Effectuer la multiplication $\underline{100110}_2 \times \underline{1011}_2$ en base 2.
- ⇨ Écrire les tables de multiplication en base 3. Calculer le produit $\underline{1022}_3 \times \underline{221}_3$ en travaillant en base 3.

La décomposition en base 2 nous permet de revenir sur l'algorithme d'exponentiation rapide dont nous avons donné une version récursive dans le chapitre sur les fonctions et dont nous donnons ici une version itérative. Étant donné $n \in \mathbb{N}$, on effectue sa décomposition en base 2

$$n = \sum_{k=0}^{w-1} d_k 2^k$$

et on remarque que pour tout x

$$x^n = x^{\sum_{k=0}^{w-1} d_k 2^k} = \prod_{k=0}^{w-1} \left[x^{(2^k)}\right]^{d_k}.$$

Comme $d_k \in \{0, 1\}$, quel que soit $k \in [\![0, w-1]\!]$, le terme entre crochets est soit présent dans le produit (si $d_k = 1$) soit absent (si $d_k = 0$). De plus, il est aisé de calculer les valeurs successives de x^{2^k} car chaque terme est le carré du précédent. On obtient ainsi l'algorithme d'exponentation rapide dans sa version itérative.

```python
def expo(x, n):
    """expo(x: int, n: int) -> int"""
    ans = 1
    y = x
    m = n
    while m > 0:
        if m % 2 == 1:
            ans = ans * y
        y = y * y
        m = m // 2
    return ans
```

1.2 Représentation mémoire des entiers non signés

La décomposition en base 2 nous donne un moyen de représenter les nombres positifs à l'aide d'une séquence de bits. Comme cette décomposition s'effectue uniquement pour les entiers positifs, on parle de représentation des entiers *non signés*. C'est cette représentation que les processeurs utilisent pour manipuler les entiers non signés. Il ne sont cependant capables que de travailler avec des entiers ayant une largeur w fixée. Aujourd'hui, un processeur 64 bits peut travailler avec des entiers codés sur 8, 16, 32 ou 64 bits.

Proposition 1.3

Pour une largeur de $w \in \mathbb{N}$, le plus grand entier non signé représentable est $2^w - 1$.

Avec 8 bits, on peut coder tous les entiers entre 0 et $2^8 - 1 = 255$. Avec 16 bits, on peut coder tous les entiers entre 0 et $2^{16} - 1 = 65\,535$. Avec 32 bits on peut coder tous les entiers entre 0 et $2^{32} - 1 = 4\,294\,967\,295$, soit quelques milliards. Avec 64 bits on peut coder tous les entiers entre 0 et quelques milliards de milliards, ce qui est suffisant pour de nombreuses applications.

Ces représentations sur une largeur fixe ont cependant un défaut : la somme et le produit de deux entiers représentables ne sont pas toujours représentables. Par exemple, avec une largeur de 8 bits, 250 et 12 sont représentables, mais $250 + 12$ ne l'est pas. Lorsque ce problème survient, on dit qu'il y a un *dépassement de capacité* (*overflow* en anglais) et le résultat obtenu est calculé modulo 256. Dans notre exemple, le calcul sur 8 bits de $250 + 12$ donnera donc 6 ! Certains langages comme le C ne cachent pas cette caractéristique des processeurs et le programmeur a la responsabilité de s'assurer que ce problème n'arrive jamais (ou d'agir en conséquence). Python travaille quant à lui avec des entiers de taille variable. L'avantage est qu'il peut manipuler des entiers aussi grands que l'on souhaite ; on ne risque pas de dépassement de capacité. L'inconvénient est que les opérations usuelles sur ces entiers sont bien plus lentes qu'en C et que leur temps d'exécution dépend de la taille des entiers.

1.3 Représentation mémoire des entiers signés

Les entiers considérés pour le moment étaient supposés positifs, mais les processeurs proposent bien évidemment de travailler avec des types *signés*, qui permettent de représenter des valeurs négatives. D'une manière ou d'une autre, il est clair que le signe nous « coutera » un bit : il faut stocker l'information + ou −. Il est assez naturel d'imaginer la stratégie suivante :

— Le bit le plus significatif détermine le signe : un 1 signifie que le nombre est positif, un 0 qu'il est négatif.
— La valeur absolue du nombre est stockée de manière standard sur les autres bits.

Implicitement, nous considérons ici que l'on travaille avec des entiers d'une largeur w fixée. Ainsi, l'expression *bit le plus significatif* désigne le bit d_{w-1} de poids maximal dans cette largeur, ce qui explique pourquoi il peut être égal à zéro.

Bien que cette méthode paraisse raisonnable, elle a deux défauts :

— Le nombre 0 a deux représentations : $00\ldots0$ et $10\ldots0$. Cela a pour conséquence de ne pouvoir stocker que $2^w - 1$ valeurs différentes, par exemple les entiers de $-(2^{w-1} - 1)$ à $2^{w-1} - 1$ inclus. On perd une place puisque zéro en prend deux.
— Les opérations arithmétiques usuelles ne sont pas très simples à effectuer : essentiellement, pour ajouter deux nombres, on est obligé de regarder leur bit de poids fort pour déterminer leur signe et de distinguer les cas.

En réalité, aucun ordinateur n'utilise cette représentation pour les entiers signés. L'immense majorité utilise la représentation par *complément à deux*.

Proposition 1.4

Soit $w \in \mathbb{N}^*$. Pour tout $n \in [\![-2^{w-1}, 2^{w-1}[\![$, il existe un unique w-uplet $(b_0, b_1, \ldots, b_{w-1})$ de bits tel que

$$n = \left(\sum_{k=0}^{w-2} b_k 2^k\right) - b_{w-1}2^{w-1}.$$

Remarques

⇨ Le bit b_{w-1} est nul si et seulement si $n \geqslant 0$. Si tel est le cas $(b_0, \ldots, b_{w-1})$ est la décomposition de n en base 2. Sinon $n < 0$, $b_{w-1} = 1$ et $(b_0, \ldots, b_{w-1})$ est la décomposition de $n + 2^w$ en base 2.

⇨ On appelle valeur en *complément à deux* de la suite de bits $(b_0, \ldots, b_{w-1})$ l'entier

$$\left(\sum_{k=0}^{w-2} b_k 2^k\right) - b_{w-1}2^{w-1}.$$

On dit que le bit de poids fort a un poids négatif.

⇨ Une même suite de bits $(b_0, \dots, b_{w-1})$ dans une largeur w, correspondra donc à deux entiers différents : $\sum_{k=0}^{w-1} b_k 2^k$ en non signé et $\sum_{k=0}^{w-2} b_k 2^k - b_{w-1}2^{w-1}$ en signé par complément à deux. Fixons par exemple $w := 4$, et notons respectivement $\mathrm{v}_4(b_3b_2b_1b_0)$ et $\mathrm{vs}_4(b_3b_2b_1b_0)$, les valeurs non signées et signées associées à une suite de bits.

— $\mathrm{vs}_4(0000) = \mathrm{v}_4(0000) = 0$.
— $\mathrm{vs}_4(0100) = \mathrm{v}_4(0100) = 4$.
— $\mathrm{vs}_4(1100) = -8 + 4 = -4$ et $\mathrm{v}_4(1100) = 8 + 4 = 12$.
— $\mathrm{vs}_4(1111) = -8 + 4 + 2 + 1 = -1$ et $\mathrm{v}_4(1111) = 8 + 4 + 2 + 1 = 15$.

Exercice 3

⇨ On fixe $w := 8$.

1. Quel est le plus grand entier non signé représentable, c'est-à-dire la plus grande valeur $\mathrm{v}_8(bits)$ que l'on peut obtenir ?
2. Quels sont les plus petits et plus grands entiers signés représentables ?
3. Quelle suite de bits donne $\mathrm{vs}_8(bits) = 0$? 127 ? -1 ? -128 ?

Proposition 1.5

Pour une largeur $w \in \mathbb{N}^*$
— Le plus grand entier signé représentable en complément à 2 est $2^{w-1} - 1$.
— Le plus petit entier signé représentable en complément à 2 est -2^{w-1}.

Remarques

⇨ *Explosion d'Ariane 5* : Le vol inaugural de la fusée Ariane 5 a eu lieu le 4 juin 1996. Comme le montre l'illustration ci-dessous, il s'est terminé, un peu moins de 37 secondes après le décollage, par ce que nous appellerons pudiquement un Rud (*Rapid Unplanned Dissassembly*).

Dommage.

La fusée et son chargement avaient couté 500 millions de dollars. La commission d'enquête a rendu son rapport au bout de deux semaines. Il s'agissait d'une erreur de programmation dans le système inertiel de référence. À un moment donné, un nombre codé en virgule flottante qui représentait la vitesse horizontale de la fusée par rapport à la plateforme de tir était converti en un entier signé sur 16 bits. Malheureusement, le nombre en question était plus grand que 32 767 et la conversion a été incorrecte.

```
L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
                                G_M_INFO_DERIVE(T_ALG.E_BV));
if L_M_BV_32 > 32767 then
   P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
   P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M
end if;

P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                ((1.0/C_M_LSB_BH) *
                                G_M_INFO_DERIVE(T_ALG.E_BH)))
end LIRE_DERIVE;
```

Extrait du code source (en Ada) d'Ariane 5. On peut voir un certain nombre de conversions d'un entier 32 bits vers un entier 16 bits avec protection contre les dépassements de capacité, et, soulignée en rouge, une conversion non protégée d'un flottant vers un entier 16 bits.

⇨ L'année 2012 a été marquée par une contribution majeure au patrimoine culturel de l'humanité : la vidéo *Gangnam Style*. À cette époque, le nombre de vues d'une vidéo YouTube était codé sur un entier 32 bits signé. Bien évidemment, ce choix, qui limite le nombre de vues à 2 147 483 647, n'était absolument pas adapté à un chef-d'œuvre de cette ampleur. Début 2014, il est devenu évident qu'on allait bientôt avoir un dépassement de capacité. Fort heureusement, YouTube a apporté la modification nécessaire à temps : les vues sont maintenant codées sur un entier signé de 64 bits, ce qui laisse de la marge, la valeur maximale étant 9 223 372 036 854 775 807. *Baby shark* peut rester tranquille pour de nombreuses années.

Proposition 1.6

Pour toute largeur $w \in \mathbb{N}$ et toute suite de bits $(b_0, \dots, b_{w-1})$, on a

$$\mathrm{v}_w(bits) \equiv \mathrm{vs}_w(bits) \mod 2^w.$$

Remarques

⇨ Cette propriété est la raison d'être de la représentation par complément à deux.

⇨ Nous n'allons pas rentrer dans les détails qui ne nous intéressent pas vraiment, mais l'avantage principal de la représentation en complément à deux, illustré par l'exercice ci-dessous, est qu'on peut utiliser essentiellement le même circuit physique pour les opérations arithmétiques sur les entiers signés et non signés.

Exercice 4

⇨ 1. Poser les additions binaires suivantes :

	1011		1001		1001		0101
+	0111	+	0011	+	1011	+	0011

2. Interpréter ces additions comme des opérations sur des entiers non signés de 4 bits. On ne gardera donc que les quatre bits les moins significatifs du résultat. Mathématiquement, cela revient à faire quoi ?
3. Reprendre la question précédente en faisant cette fois une interprétation signée, en complément à deux, sur quatre bits.
4. Quel critère simple, portant sur les deux bits de retenue de poids les plus forts, peut-on utiliser pour déterminer s'il y a eu un « vrai » dépassement de capacité ? Par « vrai » dépassement de capacité, on entend une situation dans laquelle le résultat mathématique de l'opération n'est pas représentable avec la largeur fixée.

2 Les nombres flottants

2.1 Représentation mémoire des flottants

Pour écrire un nombre réel de manière approchée, les physiciens ont pris l'habitude d'utiliser l'écriture scientifique. Par exemple

$$\mathrm{e}^{\pi} \approx 23.14 = 2.314 \times 10^1 = \left(2 + \frac{3}{10} + \frac{1}{10^2} + \frac{4}{10^3}\right) \times 10^1.$$

On dit qu'un tel nombre est représenté avec 4 chiffres significatifs. Remarquons que contrairement aux entiers naturels, qui admettent tous une décomposition en base b, seuls les nombres décimaux peuvent s'écrire de manière exacte sous la forme

$$\pm \left(\sum_{k=0}^{p-1} \frac{m_k}{10^k}\right) \times 10^e$$

où $m_k \in [\![0, 9]\!]$. Même certains nombres rationnels comme $1/3$ ne peuvent pas s'écrire de la sorte. Bien entendu, ces remarques faites en base 10 sont aussi valables en base 2, plus familière des ordinateurs.

Définition 2.1

Soit $p \in \mathbb{N}^*$. On dit qu'un réel x est un nombre flottant représentable avec une mantisse de p bits lorsqu'il existe $m_0, \dots, m_{p-1} \in \{0,1\}$ et $e \in \mathbb{Z}$ tels que

$$x = \pm \left(\sum_{k=0}^{p-1} \frac{m_k}{2^k} \right) 2^e.$$

Si x est non nul, il est possible d'imposer $m_0 = 1$; cette écriture est alors unique. L'ensemble des nombres flottants représentables avec une mantisse de p bits est noté $\mathcal{F}_p$.

Remarques

- Par exemple $2.5 = (1 + 0/2 + 1/4) \times 2^1 \in \mathcal{F}_3$.
- Si x est non nul et $m_0 = 1$, cette écriture est appelée *décomposition en base 2 normalisée*. Les autres écritures comme $x = (0 + 1/2 + 1/4) \times 2^4$ sont dites *dénormalisées*
- Tous les entiers compris entre $-2^p + 1$ et $2^p - 1$ sont des éléments de $\mathcal{F}_p$.

Exercice 5

- Montrer que les rationnels $r = \pm a/b$ (avec a et b premiers entre eux) tels que b n'est pas une puissance de 2 ne sont pas des éléments de $\mathcal{F}_p$. En particulier $0.1 = 1/10$ et $1/3$ n'appartiennent pas à $\mathcal{F}_p$.

Proposition 2.2

Soit $x \in \mathbb{R}$. Alors il existe un élément f de $\mathcal{F}_p$ minimisant la distance de x à $\mathcal{F}_p$. De plus

$$|x - f| \leqslant u_p \, |x| \quad \text{avec } u_p \coloneqq 2^{-p}.$$

Remarques

- Les éléments de $\mathcal{F}_p$ permettent donc d'approcher n'importe quel réel avec une *erreur relative* inférieure à u_p. Cet élément u_p est appelé *epsilon* de $\mathcal{F}_p$.
- Pour une valeur de x, f est unique sauf dans le cas particulier où x est au milieu de deux éléments successifs de $\mathcal{F}_p$. Dans ce cas, un seul de ces deux éléments a une décomposition en base 2 normalisée telle que $m_{p-1} = 0$. C'est cet élément f qu'on appelle *arrondi* de x à la *précision* $\mathcal{F}_p$.

Définition 2.3

Si $p, q \in \mathbb{N}^*$, on note $\mathcal{F}_{p,q}$ l'ensemble formé
- des réels x de la forme
$$x = \pm \left(m_0 + \frac{m_1}{2} + \frac{m_2}{4} + \cdots + \frac{m_{p-1}}{2^{p-1}} \right) 2^e$$
où $m_0, \dots, m_{p-1} \in \{0,1\}$ et $-2^{q-1} + 2 \leqslant e \leqslant 2^{q-1} - 1$.
- des éléments $+\infty$ et $-\infty$.
- de l'élément noté NaN (Not a Number)

Remarques

- Il y a en fait deux zéros, notés 0^+ et 0^-, mais ils sont le plus souvent affichés de la même façon.
- Si x est non nul, le plus souvent, il est possible d'imposer $m_0 = 1$. Ce n'est cependant pas possible pour les nombres de la forme
$$x = \pm \left(0 + \frac{m_1}{2} + \frac{m_2}{4} + \cdots + \frac{m_{p-1}}{2^{p-1}} \right) 2^e$$
pour $e = -2^{q-1} + 2$. De tels nombres sont dits *dénormalisés*.
- Les processeurs actuels proposent en général deux types de nombres flottants.
  - Le format simple précision, codé sur 32 bits, utilise $p = 24$ et $q = 8$. On a alors $u = 2^{-24} \approx 5.9 \times 10^{-8}$.
  - Le format double précision, codé sur 64 bits, utilise $p = 53$ et $q = 11$. On a alors $u = 2^{-53} \approx 1.1 \times 10^{-16}$.

  C'est le format double précision auquel Python nous donne accès avec son type `float`. Le plus petit nombre strictement positif représentable est de l'ordre de 10^{-324} et le plus grand nombre représentable est de l'ordre de 10^{308}.
- Représenter un réel à l'aide d'une succession de bits revient donc à coder son signe, sa mantisse et son exposant. Avec les nombres flottants double précision de la norme IEEE 754, on se donne 64 bits pour stocker ces trois données. Si $x \in \mathcal{F}_{53,11}$ est non nul et $-1022 \leqslant e \leqslant 1023$, on code, dans l'ordre
  - Le *signe*, qui ne nécessite qu'un seul bit : 0 code le signe positif et 1 le signe négatif.

— L'*exposant*, qui se code sur 11 bits. L'entier e est compris entre -1022 et 1023 on code l'écriture binaire de $e + 1023$ qui est un entier entre 1 et 2046.
— La *mantisse*, qui se code sur 52 bits : comme $m_0 = 1$, il suffit de coder $m_1, \ldots, m_{52}$.

⇨ Dans la plage de 11 bits dévolue à l'exposant, les entiers $\underline{0\ldots0}_2 = 0$ et $\underline{1\ldots1}_2 = 2047$ ne sont pas utilisés dans l'explication précédente. Le nombre 0 est représenté par un exposant e égal à -1023, donc codé $\underline{0\ldots0}_2 = 0$, avec une mantisse dont tous les bits sont nuls. Il y a bien deux 0, un 0^+ et un 0^- puisque le bit donnant le signe peut prendre les deux valeurs 0 ou 1 (les 63 autres bits sont à zéro). L'exposant décalé égal à 2047 est utilisé pour les situations particulières ($+\infty$, $-\infty$ et d'autres choses plus compliquées comme NaN). Dans ce cas, tous les bits dévolus à l'exposant sont égaux à 1.

Charles Leclerc semble avoir un problème sur l'un de ses capteurs de vitesse

2.2 Problèmes liés à l'arithmétique des nombres flottants

2.2.1 Overflow et underflow

Le problème le plus simple à comprendre est celui du dépassement arithmétique (*overflow* en anglais), qui survient lorsqu'on dépasse le plus grand nombre représentable, qui est de l'ordre de 10^{308} en double précision. Dans ce cas, le résultat est $+\infty$ ou $-\infty$.

```
In [1]: a = 2.0**1023

In [2]: a
Out[2]: 8.98846567431158e+307

In [3]: 2.0 * a
Out[3]: inf
```

De même, pour les flottants strictement positifs, il peut y avoir dépassement par valeurs inférieures, ou *underflow*. Dans ce cas, le nombre renvoyé est 0 (plus précisément 0^+ dans notre cas).

```
In [4]: 2.0**(-1074)
Out[4]: 5e-324

In [5]: 2.0**(-1075)
Out[5]: 0.0
```

Des études ont montré que même lors de calculs intermédiaires, de tels nombres n'apparaissaient presque jamais. Ces problèmes peuvent donc essentiellement être ignorés.

2.2.2 Inexactitude de la représentation, arrondis

Les arrondis sont plus problématiques et sont liés au fait que les nombres flottants n'ont qu'un nombre fini de chiffres significatifs. Deux types d'arrondis entrent en jeu.

Le premier est dû au fait que les ordinateurs travaillent en base 2 alors qu'ils échangent le plus souvent des informations avec l'utilisateur et le programmeur en base 10. Par exemple, lorsqu'on écrit

```
In [1]: x = 0.1
```

```
In [2]: x
Out[2]: 0.1
```

on pourrait facilement être dupé et croire que 0.1 est représentable de manière exacte en double précision. Or $0.1 = 1/10$ n'appartient à aucun des $\mathcal{F}_p$ puisque 10 n'est pas une puissance de 2. Il n'est donc pas représentable de manière exacte par un nombre flottant, un peu comme $1/7 = 0.142857\underline{142857}$ n'est pas un nombre décimal (le souligné signifie que le groupe de chiffres se répète à l'infini). Si l'on effectue un développement de 1/10 en base 2, on obtient $1.100\underline{1100} \times 2^{-4}$ (encore une fois, le groupe de chiffres se répète à l'infini). Lorsque 0.1 est entré dans le shell, Python va effectuer son développement en base 2. Comme ce développement est infini et qu'il travaille en double précision, il ne va garder que les 53 premiers bits et arrondir 0.1 à un nombre légèrement différent, avant de stocker ce nombre dans x. C'est un *arrondi de conversion* de bases. Pour afficher la valeur de x à l'utilisateur, il va le reconvertir en base 10 et un autre arrondi de conversion va faire qu'il affiche 0.1. Mais il ne faut pas oublier que ce n'est pas 0.1 qui est stocké dans x.

Le second problème est plus fondamental : les ensembles $\mathcal{F}_p$ ne sont stables ni par addition, ni par soustraction, ni par multiplication ; encore moins par division. Par exemple

$$x \coloneqq (1 + 0/2 + 1/4) \times 2^1 \in \mathcal{F}_3 \quad \text{et} \quad y \coloneqq (1 + 0/2 + 0/4) \times 2^{-2} \in \mathcal{F}_3,$$

mais

$$x + y = (1 + 0/2 + 1/4 + 1/8) \times 2^1 \notin \mathcal{F}_3$$

car $x + y$ possède 4 chiffres significatifs. Pour chaque opération élémentaire (addition, soustraction, multiplication, division) entre deux nombres flottants, le processeur se trouve donc dans l'incapacité de représenter le résultat exact par un nombre flottant : il va devoir effectuer un *arrondi arithmétique*. Même si `+` et `*` restent commutatives, ces arrondis leur font perdre leur associativité. Plus le nombre d'opérations élémentaires va être grand, plus ces arrondis vont nous éloigner du résultat attendu.

Exercice 6

⇨ Observez les lignes suivantes et commentez.

```
In [1]: 1.0 + 2.0**(-53) + (-1.0)
Out[1]: 0.0

In [2]: 1.0 + (-1.0) + 2.0**(-53)
Out[2]: 1.1102230246251565e-16
```

Ces phénomènes ne sont pas anodins et ne doivent pas être pris à la légère. Par exemple, si l'on veut tester l'égalité de deux flottants et qu'il existe une légère différence entre eux due aux arrondis, on aura un résultat surprenant !

```
In [3]: 0.1 + 0.1 + 0.1 == 0.3
Out[3]: False

In [4]: import math

In [5]: math.sqrt(10)**2 == 10.0
Out[5]: False
```

On retiendra qu'il ne faut jamais faire de test d'égalité entre deux nombres flottants. En pratique, on ne se posera pas la question de savoir si `a == b`, mais plutôt de savoir si `abs(a - b) <= eps * abs(a)` où `eps` est un nombre « petit », à choisir selon notre application ($\varepsilon \coloneqq \sqrt{u} \approx 10^{-8}$ est souvent un bon choix).

Nous avons vu pour le moment des calculs où les erreurs introduites par les arrondis étaient négligeables devant les grandeurs manipulées. Mais ce n'est pas toujours le cas. Supposons par exemple que l'on souhaite calculer

$$u_n \coloneqq \int_0^1 x^n \mathrm{e}^x \,\mathrm{d}x$$

pour tout $n \in \mathbb{N}$. Un encadrement élémentaire de l'intégrande nous montre que

$$\forall n \in \mathbb{N}, \quad 0 \leqslant u_n \leqslant \int_0^1 x^n \mathrm{e} \,\mathrm{d}x = \frac{\mathrm{e}}{n+1}$$

ce qui prouve par le théorème des gendarmes que u_n tend vers 0 lorsque n tend vers $+\infty$. Si l'on veut calculer explicitement u_n, on remarque que $u_0 = \mathrm{e} - 1$ et une intégration par partie nous donne

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \mathrm{e} - (n+1)u_n.$$

Le programme suivant permet donc de calculer u_n.

```python
import math

def integrale(n):
    """integrale(n: int) -> float"""
    u = math.exp(1.0) - 1.0
    for k in range(n):
        u = math.exp(1.0) - (k + 1) * u
    return u
```

```
In [6]: [integrale(n) for n in [0, 5, 10, 20]]
Out[6]: [1.718281828459045, 0.395599547802016,
         0.22800151529345358, -129.26370813285942]
```

Les premières valeurs de u_n calculées sont réalistes, mais u_{20} est totalement faux. Ce phénomène était prévisible, car si $\alpha \in \mathbb{R}$ et (v_n) est la suite définie par

$$v_0 \coloneqq \alpha, \quad \text{et} \quad \forall n \in \mathbb{N}, \quad v_{n+1} \coloneqq \mathrm{e} - (n+1)v_n$$

alors, en définissant l'erreur $\varepsilon_n \coloneqq v_n - u_n$, on obtient facilement $\varepsilon_{n+1} = -(n+1)\varepsilon_n$ et donc

$$\forall n \in \mathbb{N}, \quad \varepsilon_n = (-1)^n n! \varepsilon_0.$$

Si α est une valeur approchée de $\mathrm{e} - 1$ telle que $|\varepsilon_0| = |\alpha - (\mathrm{e} - 1)| \propto 10^{-16}$, comme $20! \propto \times 10^{18}$, on en déduit que $|v_{20} - u_{20}| \propto 100$. Donc si l'on effectue une erreur de calcul de l'ordre de 10^{-16} pour u_0, même si les calculs suivants sont exacts, l'erreur absolue obtenue pour le 20^e terme de la suite (u_n) est de l'ordre de 100, ce qui est beaucoup plus grand que l'ordre de grandeur de u_{20} puisque $0 \leqslant u_{20} \leqslant \mathrm{e}/21 \approx 0.13$. Certains algorithmes numériques comme celui-ci sont *instables* et rendent les calculs avec des nombres flottants inexploitables. D'autres sont *stables* et peuvent donc être utilisés avec des nombres flottants. L'étude de la stabilité des algorithmes numériques dépasse le cadre du programme des classes préparatoires.

2.2.3 Quelques catastrophes dues à une mauvaise utilisation des nombres flottants

Il y a un nombre de catastrophes qui sont attribuables à une mauvaise gestion de l'arithmétique des nombres flottants. Dans le premier exemple, cela s'est payé en vies humaines.

— *Missile Patriot* : En février 1991, pendant la guerre du Golfe, une batterie américaine de missiles Patriot, à Dharan (Arabie Saoudite), a échoué dans l'interception d'un missile Scud irakien. Le Scud a frappé un baraquement de l'armée américaine et a tué 28 soldats. La commission d'enquête a conclu à un calcul incorrect du temps de parcours du scud, dû à un problème d'arrondi. Les nombres étaient représentés en virgule fixe sur 24 bits. Le temps était compté par l'horloge interne du système en dixièmes de seconde. Malheureusement, $1/10$ n'a pas d'écriture finie dans le système binaire : $1/10 = 0.1$ (dans le système décimal) $= 0.0001100110011001100110011...$ (dans le système binaire). L'ordinateur de bord arrondissait $1/10$ à 24 chiffres, d'où une petite erreur dans le décompte du temps pour chaque dixième de seconde. Au moment de l'attaque, la batterie de missile Patriot était allumée depuis environ 100 heures, ce qui a entrainé une accumulation des erreurs d'arrondi de 0.34 s. Pendant ce temps, un missile Scud parcourt environ 500 m, ce qui explique que le Patriot soit passé à côté de sa cible. Ce qu'il aurait fallu faire c'est redémarrer régulièrement le système de guidage du missile.

— *Bourse de Vancouver* : Un autre exemple où les erreurs de calcul ont conduit à une erreur notable est le cas de l'indice de la Bourse de Vancouver. En 1982, elle a créé un nouvel indice avec une valeur nominale de 1000. Après chaque transaction boursière, cet indice était recalculé et tronqué après le troisième chiffre décimal et, au bout de 22 mois, la valeur obtenue était 524.881, alors que la valeur correcte était 1098.811. Cette différence s'explique par le fait que toutes les erreurs d'arrondi étaient dans le même sens : l'opération de troncature diminuait à chaque fois la valeur de l'indice.

3 Caractères et chaines de caractères

3.1 Codes ASCII et Unicode

Le code ASCII associe un caractère à chaque entier entre 0 et 127, ce qui correspond à 7 bits non signés. Ces caractères peuvent être classés en trois grandes catégories :

— *Caractères alphanumériques :* les chiffres et les lettres minuscules et majuscules. Seules les lettres utilisées en anglais font partie du code ASCII, donc pas de « é », de « ñ », de « ß »...

— *Autres caractères imprimables :* les signes de ponctuation, quelques symboles ()+, *, }, etc) et l'espace.

— *Caractères non imprimables :* la tabulation, les différents caractères correspondant à un retour à la ligne, le caractère nul, etc.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30				!	"	#	$	%	&	’
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Comme les caractères étaient presque systématiquement codés sur 8 bits, les codes 128 à 255 étaient « libres » : ils ont pendant très longtemps été utilisés pour coder les caractères spécifiques aux différentes langues (signes diacritiques, lettres supplémentaires, etc). Cependant ces extensions posaient deux problèmes :

— Elles n’étaient pas standardisées, puisqu’elles différaient d’une langue à l’autre et qu’il y avait même plusieurs extensions concurrentes pour une même langue. En pratique, jusqu’à la fin des années 2000, il y avait une chance sur deux que tous les caractères accentués contenus dans un mail soient remplacés par une bouillie infâme avant de parvenir au destinataire.

— Elles permettaient plus ou moins de gérer les langues européennes ou au moins les langues basées sur l’alphabet latin, mais étaient totalement inappropriées au chinois, au japonais, etc.

Le standard Unicode a été développé à partir de la fin des années 1980. À l’heure actuelle, il définit des codes pour 144 697 caractères, ce qui permet de gérer l’ensemble des langues, ainsi que de nombreux caractères supplémentaires comme les Emojis, par exemple. Ce standard définit trois représentations binaires UTF-8, UTF-16 et UTF-32, et il est assez complexe : nous ne rentrerons pas dans les détails. Dans tous les cas, les *codepoints* (l’entier associé à un caractère) ne sont pas modifiés pour les caractères appartenant au code ASCII. On peut obtenir un caractère à partir de son code Unicode grâce à la fonction `chr`.

```
In [1]: ord('A')
Out[1]: 65

In [2]: ord('é')
Out[2]: 233
```

On peut obtenir un caractère à partir de son code Unicode grâce à la fonction `chr`. C’est d’ailleurs le moyen le plus simple d’obtenir des caractères non disponibles sur votre clavier.

```
In [3]: "I " + chr(9829) + " les Lazos."
Out[3]: 'I ❤ les Lazos.'
```

3.2 Lecture et écriture dans un fichier

Python permet d’ouvrir des fichiers texte en lecture ou en écriture. Pour Python, un fichier n’est qu’une séquence de caractères. Une fois que le fichier aura été ouvert par le programme, celui-ci maintiendra un marqueur fictif à la position courante qui nous indique où sera lue ou écrit la prochaine séquence. Un fichier est ouvert avec la fonction `open` :

```
f = open("/Users/fayard/Desktop/fichier.txt", 'r')
```

Le premier argument est une chaine de caractères contenant le chemin complet du fichier. Le second argument est le mode d’ouverture du fichier : `'r'` (pour read) pour une ouverture en lecture seule, `'w'` (pour write) pour une ouverture avec les droits d’écriture et enfin `'a'` (pour append) pour une ouverture avec les droits d’écriture et la position du marqueur en fin de fichier. Une fois que le travail dans le fichier sera fini, il faudra prendre soin de bien fermer le fichier avec la commande

```
f.close()
```

Une fois ouvert, la fonction `read` permet de lire le fichier en entier et le renvoie sous la forme d’une chaine de caractères :

```
texte = f.read()
```

Dans la même famille, la fonction `readlines()` permet de lire le fichier en entier, mais renvoie non pas une seule chaine de caractères, mais une liste de chaines de caractères, chaque chaine correspondant à une ligne du fichier. Attention, cette ligne se terminera par le caractère de retour à la ligne.

```
lignes = f.readlines()
```

Mais en général, on lira et on traitera le fichier ligne par ligne avec le fonction `readline()` qui lit une ligne et la renvoie en tant que chaine de caractères. Bien entendu, cette chaine finira aussi par un caractère de retour à la ligne. Après cet appel, le curseur sera positionné au début de la ligne suivante.

```
ligne = f.readline()
```

On saura qu'on est à la fin du fichier lorsque cette méthode nous renverra une chaine de caractères vide.

Mais la méthode sans doute la plus utilisée pour parcourir un fichier en lecture est de remarquer que le descripteur du fichier `f` est un itérable. La boucle

```
for ligne in f:
```

va donc permettre d'effectuer une boucle sur toutes les lignes du fichier. Cette boucle est équivalente à

`for ligne in f.readlines():`

mais a l'avantage de faire lire le fichier à notre programme ligne après ligne alors que l'utilisation de `readlines()` va charger le fichier en mémoire en entier avant même de traiter la première ligne.

Il est courant de lire des fichiers textes contenant des données organisées, comme par exemple un fichier Csv (Comma Separated Values) qui est le format le plus simple pour enregistrer les données d'un tableur. Le fichier décrit dans un fichier texte chaque ligne du document, où chaque colonne est séparée par une virgule. Si par exemple, vous souhaitez avoir une liste des élèves de la classe avec leur âge, le fichier Csv correspondant sera simplement

```
Linus,Torvalds,53
Donald,Knuth,85
Master,Yoda,900
```

Afin de séparer proprement ce type de ligne, on utilisera la fonction `split`. Par exemple, si `ligne` est égal à `"Linus,Torvalds,53"`, la commande `data = ligne.split(',')` stockera dans la variable `data` la liste

`["Linus", "Torvalds", "53"].`

Si le fichier est ouvert en écriture, on peut écrire dedans à l'aide de la méthode `write`.

```
f.write("Ma jolie histoire.")
```

Le programme spécifie que la documentation de ces fonctions doit vous être rappellée mais il est important de savoir les utiliser proprement une fois ce rappel fait.