ITC – cours n°17

Représentation binaire des données

Un ordinateur est composé de différentes parties, dont des entrées (clavier, souris, port USB...) et des sorties (carte graphique (CGU en anglais), carte son...). Tous ces composants sont à base de transistors qui ne gèrent que 2 états : le courant passe ou le courant ne passe pas. L'ordinateur ne « connaît » donc que ces 2 états, et on utilise les deux chiffres 0 et 1 pour traduire le passage (ou pas) d'un courant :

- le 0 représente l'état bas (pas de courant) ;
- le 1 représente l'état haut (courant qui passe).

Toutes les informations (nombres, chaînes de caractères, vidéos, images...) sont alors représentées par ces 2 chiffres, appelés chiffres binaires (ou bit pour « binary digit » en anglais). Afin de représenter des données plus complexes, les bits sont regroupés en **mots machines** de plus grande taille (8 bits, 32 bits, 64 bits par exemple).

Un processeur gère en général les bits par « paquets » de taille minimale, appelés **bytes**. Dans l'usage courant, la norme est que 1 byte = 8 bits, également appelé un **octet**. Ainsi, un mot machine de 32 bits pèse 4 bytes ou 4 octets.

I. Représentation des entiers

1.1. Bases décimale et binaire

On représente un nombre en base décimale avec les 10 chiffres usuels de 0 à 9 :

$$n_{10} = \sum_{k=0}^{d} a_{10,k} 10^k \quad \text{, par exemple} \quad 185_{10} = 1\times 10^2 + 8\times 10^1 + 5\times 10^0$$

De la même manière, on peut représenter un nombre entier en base 2 avec les chiffres 0 et 1 :

$$n_2 = \sum_{k=0}^{d} a_{2,k} 2^k \quad \text{, par exemple} \quad 5_{10} = 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 101_2$$

Les mots machines ne pouvant utiliser que les chiffres 0 et 1, c'est cette représentation qui est naturelle pour les entiers machine.

1.2. Représentation d'un entier positif sur un mot de taille fixe

a) Définition

Considérons un entier positif encodé sur un mot de taille N (en général $N = 32$ bits, dans les exemples de ce cours nous prendrons $N = 8$ bits). Dans ce cas, la représentation est simple : un bit correspond à un chiffre. On obtient alors les propriété suivantes :

- 0 est codé par 00000000, 1 par 00000001, 2 par 00000010...
- l'intervalle accessible est $[\![0; 2^N - 1]\!]$, soit $[\![0; 255]\!]$ pour $N = 8$ et $[\![0; 4294967296]\!]$ pour $N = 32$.

b) Opérations

L'addition marche simplement avec gestion de la retenue. Si changer un bit est considéré comme une opération élémentaire, alors l'addition coûte exactement N : sur un mot de taille fixée, il s'agit d'un coût constant.

Il y a un risque associé à l'intervalle fini accessible : celui de dépassement de capacité. En effet,

$$11111111 + 00000001 = 1\ 00000000 \qquad (255 + 1 = 256 = 0 \bmod 2^8)$$

qui ne peut donc pas exister sur les N bits réservés. En pratique, le bit excédentaire est perdu, donc on revient à 0 : les nombres sont calculés modulo 2^N.

La multiplication par 2^p peut être effectuée avec le même nombre d'opérations élémentaires que l'addition : il suffit de décaler les bits de p cases vers la gauche, et d'ajouter des zéros à droite, en effet :

$$n_2 \times 2^p = \sum_{k=0}^{N-1} a_k \times 2^{k+p} = \sum_{k=p}^{N+p-1} a_k \times 2^k$$

En Python, il existe un opérateur pour cette opération : $n \ll p$. De même, on peut très rapidement diviser par 2^p en décalant les bits vers la droite (en Python $n \gg p$).

On peut proposer un algorithme naïf de multiplication quelconque en décomposant en N décalages de bits et sommes, par exemple :

$$\begin{aligned} 00010010 \times 00001011 &= 00010010 \times (00001000 + 00000010 + 00000001) \\ &= 00010010 \ll 3 + 00010010 \ll 1 + 00010010 \ll 0 \\ &= 10010000 + 00100100 + 00010010 \\ &= 11000110 \end{aligned}$$

Cet algorithme simple a donc une complexité N^2 ; dans la pratique, il existe des algorithmes plus performants, par exemple en $\mathcal{O}(N^{\log_3(5)}) \simeq \mathcal{O}(N^{1,465})$; celui de meilleure complexité est en $\mathcal{O}(N \log N)$, mais il est inutile en pratique car il ne devient meilleur que les autres que pour des entiers codés sur au moins 1729^{12} bits.

Globalement, si N est fixé, la complexité des opérations arithmétiques peut être considérée constante.

1.3. Représentation d'un entier relatif sur un mot de taille fixe

a) Approche naïve : la représentation par valeur absolue

On peut sacrifier le premier bit pour représenter le signe du nombre ($0 \leftrightarrow +$, $1 \leftrightarrow -$) et les $N - 1$ bits restants représentent alors la valeur absolue :

$$00100100_2 = 40_{10} \quad \text{et} \quad 10100100_2 = -40_{10}$$

Cette approche présente cependant deux problèmes :
- problème mineur : 0 a deux représentations (00000000 et 10000000)
- problème majeur : l'algorithme d'addition binaire simple ne fonctionne plus ! Par exemple, avec $3 + (-4)$:

$$00000011 + 10000100 = 10000111$$

qui représente -7.

Plutôt que de changer l'algorithme d'addition, on utilise une meilleure représentation.

b) Approche correcte : complément à deux

« Complément à deux » est l'abbréviation de « complément à 2^N » :
- les nombres positifs sont représentés normalement sur les $N-1$ derniers bits ;
- les nombres négatifs sont représentés par $2^N - |x|$.

De cette manière, le dépassement de capacité sert directement à gérer le signe dans l'addition. Pour prendre l'opposé de x, on peut suivre l'algorithme simple suivant : on inverse les bits puis on ajoute 1.

$$(-4)_{10} = (-00000100)_2 = (11111011)_2 + 1 = 11111100_2$$

On constate bien que l'algorithme usuel de l'addition fonctionne à nouveau :

$$(-4+5)_{10} = 11111100_2 + 00000101 = 1\ 00000001 = 00000001 \bmod 2^8$$

Ainsi, les opérations arithmétiques conservent la même complexité qu'avec les nombres positifs.

1.4. Entiers multi-précision

Python utilise une sur-couche pour éviter les dépassements : il adapte le nombre de bits à la valeur du nombre. On peut le voir à l'aide de l'instruction `bit_length` :

```
In [1]: (15).bit_length()
Out[1]: 4
In [2]: (255).bit_length()
Out[2]: 8
In [3]: (2**71).bit_length()
Out[3]: 72
```

Cela demande un temps de calcul supplémentaire, et surtout les opérations arithmétiques ne sont plus en temps constant, mais en $\mathcal{O}(N) = \mathcal{O}(\log_2 n)$. Ainsi, supposer que les opérations élémentaires sont en temps constant n'est pas tout à fait vrai en Python (mais c'est un modèle valide dans d'autres langages). Un calcul de complexité d'un algorithme est donc soumis à un **choix de modèle** pour les opérations élémentaires, et selon le langage concret d'implémentation, le choix de modèle est plus ou moins adapté.

II. Représentation des réels

2.1. Rappels : décimaux, rationnels, réels

On rappelle que l'ensemble des nombres réels $\mathbb{R}$ est un ensemble continu, tandis que les nombres rationnels $\mathbb{Q}$ sont ceux qui s'écrivent comme un rapport d'entiers. Les nombres décimaux sont ceux qui s'écrivent avec un développement décimal fini, donc qui peuvent être écrits sous la forme

$$d = \frac{n}{10^p} \quad , \quad (n,p) \in \mathbb{Z} \times \mathbb{N}$$

Ces ensembles sont ordonnés ainsi :

$$\mathbb{D} \subset \mathbb{Q} \subset \mathbb{R}$$

On peut représenter un nombre décimal en notation scientifique avec des entiers uniquement, appelés **exposant** et **mantisse** :

$$1{,}4325 = \underbrace{14325}_{\text{mantisse}} \times \overbrace{10^{-4}}^{\text{exposant}}$$

2.2. Représentation à virgule flottante

Les nombres dits « à virgule flottante » sont une représentation binaire des nombres réels : elle se présente en trois parties (sur N bits)
- 1 bit de signe S ;
- e bits codant l'exposant $E \in [\![0; 2^e - 1]\!]$;
- m bits codant la mantisse $M \in [\![0; 2^m - 1]\!]$;

avec $1 + e + m = N$; le nombre s'écrit alors

$$x = S \times M \times 2^{E-d} \quad \text{avec} \quad d = 2^{e-1}$$

Dans la pratique, deux types de nombres à virgule flottante sont très utilisés : ils sont définis par la norme IEEE-754 :

Nom	N	e	d	m	bornes approchées	plus petite valeur
simple précision	32 bits	8 bits	127	23 bits	$\pm 3{,}4 \cdot 10^{38}$	$2{,}8 \cdot 10^{-45}$
double précision	64 bits	11 bits	1023	52 bits	$\pm 1{,}8 \cdot 10^{308}$	$4{,}9 \cdot 10^{-324}$

Le système de virgule flottante est une bonne représentation binaire pour les réels, mais il faut déjà remarquer quelque chose : certains réels ne sont **pas représentables** ainsi. Au-delà de l'argument mathématique (N bits implique un nombre fini de nombres représentables, tandis que les réels sont infinis dans n'importe quel intervalle), on peut le constater sur certains exemples avec la commande `format` pour afficher le nombre souhaité de décimales :

```
In [1]: format(0.1, '.24g')
Out[1]: 0.10000000000000005551115
In [2]: format(-1/3, '.24g')
Out[2]: -0.33333333333333314829616
```

2.3. Problèmes d'arrondi

Une conséquence directe de la limite des nombres non représentables est la difficulté à faire des calculs stables avec des nombres à virgule flottante. D'une part, lorsqu'un résultat n'est pas représentable, il est arrondi vers le nombre représentable le plus proche :

```
In [1]: 5e-324 / 10
Out[1]: 0.0
```

Par ailleurs, les opérations élémentaires ne sont plus commutatives :

```
In [1]: 0.1 + 1e-15 - 0.1
Out[1]: 9.992007221626409e-16
In [2]: 0.1 - 0.1 + 1e-15
Out[2]: 1e-15
```

Il faut bien comprendre que ce ne sont pas des bugs, mais une limitation intrinsèque de l'informatique : tout doit être représenté en binaire, il n'y a donc pas de représentation parfaitement satisfaisante des réels.

Un problème particulièrement subtil est celui des comparaisons de deux nombres réels :

```
In [1]: 0.1 + 0.1 + 0.1 == 0.3
Out[1]: False
# Explication :
In [2]: format(0.1 + 0.1 + 0.1, '.24g')
Out[2]: 0.30000000000000004440892l
In [3]: format(0.3, '.24g')
Out[3]: 0.29999999999999998889777
```

Pour contourner ce problème, il existe plusieurs façons de comparer le fait que deux nombres sont « assez proches », par exemple

```
In [1]: x, y = 0.1 + 0.1 + 0.1, 0.3
In [2]: x == y                          # méthode naïve
Out[1]: False
In [3]: atol, rtol = 1e-8, 1e-5
In [4]: abs(x - y) < atol               # comparaison en valeur absolue
Out[2]: True
In [5]: abs(x - y) < rtol*abs(y)        # comparaison en valeur relative
Out[3]: True
In [6]: abs(x - y) < atol + rtol*abs(y)  # mélange des deux
Out[4]: True
```

Puisque NumPy est pensé pour le calcul numérique, elle implémente déjà une comparaison d'entiers selon la troisième méthode : il s'agit de la fonction `np.isclose`. On peut y régler les paramètres de tolérance `rtol` et `atol`, les valeurs prises dans l'exemple ci-dessus étant les valeurs par défaut dans `np.isclose`.

2.4. Cas particuliers

La norme IEEE-754 définit également deux types d'entités particulières :

- les nombres infinis, disponibles en Python avec `float('inf')` :

```
In [1]: x = float('inf')  # majorant de tout autre nombre représentable
In [2]: y = -x            # minorant de tout autre nombre représentable
In [3]: x < 462973819e6352
Out[1]: False
In [4]: x + x
Out[2]: inf
```

- les « Not A Number » (ou NaN) qui servent à représenter les résultats de formes indéterminées

```
In [5]: z = x - x  # inf - inf
In [6]: z
Out[3]: nan
```

et que l'on peut obtenir directement avec `float('nan')`.