Représentation des entiers en machine, entiers relatifs

Représentation en machine et dépassement

En Python, le type `int` permet de représenter des nombre entiers de taille arbitraire (*Big Integers*) sans limite de taille. Lorsqu'on stocke un entier dans une variable, Python alloue dynamiquement un nombre de bits suffisants pour le représenter en mémoire. Cela permet d'effectuer des calculs avec des nombres entiers extrêmement grands, limités uniquement par la mémoire disponible sur l'ordinateur.

```
>>> 2**100
1267650600228229401496703205376
>>> 2**100 * 3**50
910043815000214977332758527534256632492715260325658624
```

Mais ce n'est pas le cas de tous les langages informatiques. Dans la plupart des langages (C, Java, etc.), **les entiers sont généralement stockés en utilisant un nombre de bits fixe prédéfini**. Les tailles standard sont des puissances de 2 en bits, correspondant souvent à la taille des registres du processeur, souvent 32 ou 64 bits.

L'utilisation d'une taille fixe pose le problème du **dépassement** (*overflow*). Lorsqu'une opération arithmétique (comme l'addition ou la multiplication) produit un résultat qui sort de la plage de valeurs permise par le nombre de bits alloués, il y a dépassement.

Prenons l'exemple d'une addition de deux entiers naturels stockés sur 8 bits : $150 + 150 = 300$. L'ordinateur effectue une somme en binaire équivalente, 1001 0110 + 1001 0110, mais le résultat comporte 9 bits à cause de la retenue : 1 0010 1100. Cela dépasse la capacité de stockage de 8 bits, le premier bit est alors ignoré et l'ordinateur ne garde que les 8 derniers bits, 0010 1100, c'est-à-dire 44 ce qui est bien sûr faux !

Même si certains langages informatiques offrent des types d'entiers *Big Integer* équivalents à `int` en Python qui offrent l'avantage d'éliminer les erreurs de dépassement, les opérations sont plus lentes que les opérations natives d'un processeur sur des entiers tailles fixes et il n'est pas toujours intéressant de les utiliser. Il est souvent préférable de comprendre le nombre de bits utilisés et de choisir un type d'entier adapté à la taille des nombres manipulés.

Nombres de bits

On a vu précédemment qu'un nombre qui s'écrit dans le système binaire avec n bits $b_{n-1}b_{n-2}\dots b_2b_1b_0$ (chaque b_i est un bit valant 0 ou 1) a une valeur décimale égale à :

$$b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

ou encore avec la formule mathématique d'une somme de 0 à $n-1$: $\sum_{i=0}^{n-1} b_i \times 2^i$

On peut écrire les 2 nombres 0 et 1 avec 1 seul bit, 4 nombres allant de 0 à 3 avec 2 bits, 8 nombres allant de 0 à 7 avec 3 bits, ... 2^n nombres allant de 0 à $2^n - 1$ avec n bits.

Cours

Avec n bits, on peut écrire 2^n nombres entiers naturels (**positifs**), allant de 0 à $2^n - 1$.

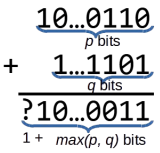
Exercice corrigé

1. En C, le type `unsigned short int` permet de stocker les entiers positifs sur 2 octets (16 bits). Quel est le plus grand nombre entier accepté ?
2. On donne les puissances de 2 suivantes : $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$ et $2^8 = 256$. Combien de bits doit-on utiliser au minimum pour représenter le nombre entier 72 ?

Réponse

Nombre de bits nécessaires pour une somme et un produit

Si on additionne deux nombres entiers qui s'écrivent respectivement dans le système binaire avec p et q bits, la somme s'écrit avec le plus grand nombre de bits entre p et q, auquel on ajoute éventuellement 1 pour prendre en compte le cas d'une retenue sur le bit de poids le plus fort. La somme s'écrit donc avec au maximum $max(p, q) + 1$ bits.



Prenons par exemple $a = 13$ (4 bits) et $b = 7$ (3 bits). D'après cette formule, la somme de a et b s'écrit donc sur $max(4, 3) + 1 = 5$ bits ou moins. En effet, $a + b = 20$ et 20 est inférieur ou égal à $2^5 - 1 = 31$, donc 5 bits suffisent.

De la même façon, si on multiplie deux nombres entiers qui s'écrivent respectivement dans le système binaire avec p et q bits, on obtient le produit en ajoutant jusqu'à $q - 1$ bits aux p bits du premier nombre auxquels il faut encore ajouter 1 pour prendre en compte le cas d'une retenue sur le bit de poids le plus fort. Le produit s'écrit donc avec au maximum $p + q$ bits.

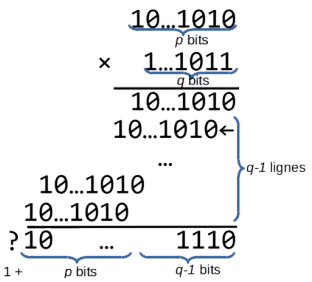
Reprenons notre exemple $a = 13$ (4 bits) et $b = 7$ (3 bits). D'après cette formule, le produit de a et b s'écrit donc sur $4 + 3 = 7$ bits ou moins. En effet, $a \times b = 91$ et 91 est inférieur ou égal à $2^7 - 1$ ($2^7 = 128$), donc 7 bits suffisent.

Cours

Si a et b sont deux nombres entiers, les nombres de bits nécessaires pour écrire leur somme et leur produit vérifient :

- $bits(a+b) \leq max(bits(a), bits(b)) + 1$
- $bits(a \times b) \leq bits(a) + bits(b)$

où $bits(n)$ est le nombre de bits nécessaires pour écrire n en binaire.



Exercice corrigé

Si $a = 200$ et $b = 58$, combien de bits au maximum sont nécessaires pour écrire $a + b$? Pour $a \times b$?

Réponse

Représentation binaire des entiers relatifs

On a vu comment les ordinateurs encodent naturellement les entiers positifs en binaire. Mais comment faire pour les entiers relatifs qui peuvent être positifs ou négatifs ? En informatique, on dit que ces entiers sont **signés**, car ils ont un signe « + » ou « - ». Dans ce cas, l'encodage le plus souvent utilisé est le **complément à 2** qui offre l'avantage de simplifier les calculs en ne distinguant pas le signe et de la valeur.

Principe du complément à 2

Avec n bits on peut représenter les 2^n entiers positifs entre 0 et $2^n - 1$. Par exemple sur 4 bits, on peut représenter les 16 (=2^4) entiers positifs compris entre 0 et 15 :

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010
0	1	2	3	4	5	6	7	8	9	10

L'idée du complément à 2 et de partager la plage de nombres binaires disponibles sur n bits en deux parties égales :

- La première moitié, contenant les 2^{n-1} premiers nombres binaires, permet de représenter les entiers positifs entre 0 et $2^{n-1} - 1$. Par exemple sur 4 bits :

0000	0001	0010	0011	0100	0101	0110	0111
0	1	2	3	4	5	6	7

- La seconde moitié, contenant les 2^{n-1} nombres binaires suivants, permet de représenter les entiers positifs entre -2^{n-1} et -1. Par exemple sur 4 bits :

1000	1001	1010	1011	1100	1101	1110	1111
-8	-7	-6	-5	-4	-3	-2	-1

Autrement dit, on convertit en binaire un nombre négatif en ajoutant 2^n à sa valeur[1]. Par exemple sur 4 bits :

- -1 est représenté par $-1 + 16 = 15$ en binaire : 1111.
- -2 est représenté par $-2 + 16 = 14$ en binaire : 1110.
- ...
- -8 est représenté par $-8 + 16 = 8$ en binaire : 1000.

On observe immédiatement un premier avantage du complément à 2 : il permet d'identifier directement le signe d'un nombre juste avec son premier bit : 0 pour les nombres négatifs, 1 pour les positifs. Ce premier bit est appelé « **bit de signe** ».

On peut toujours représenter 2^n nombres entiers relatifs sur n bits avec le complément à 2, mais à la différence des nombres positifs, les entiers représentés par le complement à deux vont de -2^{n-1} à $2^{n-1} - 1$.

Voici les plages d'entiers que l'on peut représenter avec les nombres de bits les plus courants :

n bits	Plage d'entiers naturels (non signés)	Plage d'entiers relatifs (complément à 2)
4 bits	$[0, 2^4 - 1] = [0, 15]$	$[-2^3, 2^3 - 1] = [-8, 7]$
8 bits	$[0, 2^8 - 1] = [0, 255]$	$[-2^7, 2^7 - 1] = [-128, 127]$
16 bits	$[0, 2^{16} - 1] = [0, 65\,535]$	$[-2^{15}, 2^{15} - 1] = [-32\,768, 32\,767]$
32 bits	$[0, \approx 4,29 \times 10^9]$	$[\approx -2,14 \times 10^9, \approx 2,14 \times 10^9]$

n bits	Plage d'entiers naturels (non signés)	Plage d'entiers relatifs (complément à 2)
64 bits	$[0, \approx 1,84 \times 10^{19}]$	$[\approx -9,22 \times 10^{18}, \approx 9,22 \times 10^{18}]$

Noter qu'il y a toujours un nombre négatif de plus que les positifs car 0 est représenté avec les positifs.

Cours

Le **complément à 2** permet de stocker en machine les nombres entiers relatifs, dit « entiers **signés** », en binaire sur un nombre de bits fixé.

Avec un codage sur n bits :

- Les entiers **positifs** (y compris 0) sont représentés de manière usuelle par **leur valeur**.
- Les entiers **négatifs** sont représentés par **leur valeur à laquelle on ajoute** 2^n.
- Le premier bit, dit « **bit de signe** », indique le signe du nombre : **0 s'il est positif, 1 s'il est négatif**. Les $n-1$ bits suivants indiquent la valeur du nombre (⚠️ attention au risque de dépassement).
- Cette méthode permet de représenter les 2^n nombres entiers de -2^{n-1} à $2^{n-1}-1$.

Avantage du complément à 2

L'avantage du complément à 2 est d'encoder les entiers relatifs de telle façon que la somme bit à bit d'un nombre et de son opposé, en ignorant le dépassement éventuel, est égale à 0.

Par exemple 5 et encodé en 0101 sur 4 bits et −5 en 1011. Quand on ajoute 5 et -5 en binaire, on obtient bien 1011 + 0101 = 0000 car la dernière retenue disparait sur 4 bits. Cela permet d'effectuer toutes les opérations d'addition et de soustraction sur des entiers relatifs de la même façon que pour les entiers positifs, sans distinction du signe des nombres (tant qu'on reste dans la plage $[-2^{n-1}, 2^{n-1}-1]$).

Par exemple, calculons −5 + 2 comme le ferait un ordinateur sur 4 bits :

- −5 sur 4 bits : 1011
- 2 sur 4 bits : 0010
- Somme binaire : 1011 + 0010 = 1101. On obtient le résultat attendu, −3 !

Encoder un entier négatif sur n bits

Pour encoder un nombre négatif sur n bits, la machine effectue un calcul directement au niveau des bits qui permt d'ajoute 2^n à une valeur avec la technique du complément à 2.

Cours

En machine, le complément à 2 d'un nombre négatif est obtenu en effectuant les opérations suivantes :

1. Écrire en binaire le nombre positif correspondant.
2. Inverser tous les bits (« **complément à 1** ») :
   - 0 devient 1
   - 1 devient 0
3. Ajouter 1, en ignorant le dépassement éventuel (« **complément à 2** »).

Par exemple, pour coder -5 sur 4 bits :

1. 5 en binaire → 0101
2. Complément à 1 → 1010
3. +1 → 1011

Donc -26 est encodé par 1011 sur 4 bits.

Noter que sur 8 bits (1 octet) on obtient :

1. 5 en binaire → 0000 0101
2. Complément à 1 → 1111 1010
3. +1 → 1111 1011

Exercice corrigé

Encoder -3, -13 et -26 sur 1 octet ?

Réponse

Décoder un nombre négatif sur n bits

On constate que si on fait deux fois le complément à 2 d'un nombre, on retrouve le nombre original !

Prenons par exemple le nombre 5 sur 4 bits, il s'écrit 0101. Si on effectue un premier complément à 2, on obtient 1011, c'est à dire -5. Effectuons un second complément à 2, on obtient à nouveau 0101, c'est à dire le nombre de départ, 5.

Mathématiquement c'est logique puisque le complément à 2 est une opération qui calcule l'opposé d'un nombre : si x est un nombre, son complément à 2 donne -x [2].

Pour décoder un nombre négatif, il suffit donc d'appliquer une nouvelle fois le complément à 2.

Cours

En machine, le complément à 2 d'un nombre négatif (dont le bit de signe est 1) est obtenu en effectuant les opérations suivantes :

1. Inverser tous les bits (« **complément à 1** ») :
   - 0 devient 1
   - 1 devient 0
2. Ajouter 1, en ignorant le dépassement éventuel (« **complément à 2** »)
3. Décoder le nombre binaire et prendre son opposé.

Exercice corrigé

Trouver la valeur du nombre binaire 1110 0111

Réponse

1. D'où le nom « complément à 2 puissance n », tronqué en « complément à 2 ». ↩
2. Il y a une exception avec le nombre le plus négatif représentable (par exemple -8 sur 4 bits : 1000). Si on fait son complément à 2, on obtient... 1000 à nouveau, car il n'y a pas de +8 représentable en complément à 2 sur 4 bits (le maximum est +7). ↩