Informatique PCSI
Les nombres entiers : représentation

1 Entiers multi-précision

1.1 Programmation

La plupart des processeurs calculent avec des nombres binaires de taille limitée à 32 ou 64 bits. Avec certains langages, le nombre d'octets avec lequel on travaille sur les entiers peut être précisé (un octet, deux octets, quatre octets, huit octets). Les calculs sont en général gérés directement par le processeur. Il y a certains risques qu'il faut connaître.

Pour simplifier, supposons que les entiers sont codés sur deux octets. Si le langage nous permet de travailler uniquement sur des entiers naturels, des entiers non signés, nous avons à notre disposition les nombres de 0 à 65535. Si notre programme est amené à demander le calcul $65500 + 40$, le résultat de l'addition sera 4. (En binaire, le résultat est $1\,000\ldots0100$ et le 1 à gauche disparaît). De même le résultat du produit 32×2048 sera 0. Avec des entiers signés, toujours sur deux octets, nous disposons des nombres de -32768 à 32767. Donc, par exemple, le calcul $32000 + 1000$ aura pour résultat -32536.

Pour traiter ce genre de problème, par exemple le dépassement de capacité dans une addition, on peut faire une remarque simple sur les entiers signés. Si les deux opérandes sont du même signe, il y a dépassement si le résultat n'est pas du même signe. Par exemple, avec quatre bits, $6 + 2$ donne un nombre négatif : $0110 + 0010 = 1000$. Il faut alors gérer ce type de problème.

Si les deux opérandes ne sont pas du même signe, il n'y a jamais de dépassement.

Par exemple $6 - 2$ donne bien 4 (la soustraction $6 - 2$ est remplacée par l'addition $6 + (-2)$), soit : $0110 - 0010 = 0110 + 1110 = 0100$.

Par contre, avec deux entiers signés p et q sur n bits, si $p + q \geq 2^{n-1}$ ou si $p + q < -2^{n-1}$, il y a un dépassement de capacité positif dans le premier cas, négatif dans le second cas.

1.2 Langage Python

Lorsque nous programmons en langage Python, nous n'avons pas, à priori, à nous soucier de la représentation des entiers.

Remarque : avec la représentation des entiers signés en complément à deux, il est facile d'augmenter le nombre de bits utilisés pour la représentation. Si l'entier est positif, on ajoute devant le nombre de 0 nécessaires, si l'entier est négatif, on ajoute devant le nombre de 1 nécessaires.

Le langage Python gère la taille, illimitée dit-on, mais limitée quand même par la capacité de la mémoire. Cependant, si un nombre dépasse la taille maximale utilisable par le processeur, ce nombre est découpé en deux ou plusieurs parties et Python s'occupe des différentes opérations à effectuer. Cela prend alors plus de temps et d'espace en mémoire et peut ralentir sérieusement un programme.

On peut vérifier avec le programme suivant que la première boucle s'exécute environ mille fois plus vite que la deuxième. Le temps affiché est environ 1/1000 de seconde pour la première boucle et plus d'une seconde pour la seconde. La différence s'explique par la gestion des entiers longs dans la deuxième boucle.

```
from time import time

top = time()
n = 0
for i in range(1800):
    n = n * 2 ** i
print(time() - top)

top = time()
```

```
n= 2**180
for i in range(1800):
    n = n * 2 ** i
print(time() - top)
```

1.3 Compléments

En Python, une instruction comme `x = 537` définit une variable `x` dont la valeur est 537. Plus précisément, le nom `x` est lié à l'objet entier enregistré en mémoire dont la valeur est 537. En mémoire, diverses informations sont enregistrées en plus de la valeur. Sur une machine en 64 bits, ces informations sont stockées sur des suites, ou blocs, de huit octets. La valeur peut être très grande et être stockée sur plusieurs blocs. Le nombre de blocs utilisés est donc lui aussi enregistré en mémoire.

Dans la représentation ci-dessous, chaque case représente un octet écrit en hexadécimal.

01	00	00	00	00	00	00	00	19	02	00	00

L'ordre d'enregistrement des octets dépend de la machine : on parle d'ordre *little endian* ou *big endian*. Dans l'exemple, le bloc de huit octets représente le nombre 1, pour un bloc utilisé. Le nombre 19 en hexadécimal représente le nombre $1 \times 16 + 9$ soit 25. Le bloc de quatre octets représente dont le nombre $25 + 2 \times 256^1 + 0 \times 256^2 + 0 \times 256^3 = 537$.

Si Python autorise la manipulation d'entiers sans taille limite autre que celle de la mémoire de la machine, la machine travaille elle avec des entiers codés sur des mots de taille fixe. Cette taille est limitée en particulier par le processeur (32 ou 64 bits) et les bus qui transmettent les données entre les mémoires et le processeur.

Nous pouvons obtenir facilement certaines informations avec le module `sys`.

```
>>> import sys
>>> sys.maxsize
9223372036854775807
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

Le nombre `maxsize` est un entier donnant la valeur maximale de la taille des entiers gérés par la machine, soit $2^{31} - 1$ sur une plateforme 32 bits et $2^{63} - 1$ sur une plateforme 64 bits.

Le nombre `bits_per_digit` est le nombre de bits utilisés pour chaque mot en Python. Les entiers Python sont stockés en interne en base 2^{30}.

Le nombre `sizeof_digit` est la taille en octets d'un mot Python, la taille d'un bloc.

Le nombre maximal de blocs de 4 octets est donc de l'ordre de $2^{63}/4$, ce qui nous donne une idée du plus grand entier utilisable par Python.

Pour un entier positif, on trouve donc en mémoire le nombre de blocs de 4 octets utilisés k. L'entier est ensuite codé suivant sa taille (le nombre de blocs de 4 octets) sur 4 octets, ou 8 octets, ou 12 octets, etc. Chaque bloc de 4 octets est un nombre compris entre 0 et $2^{30} - 1$ bornes comprises. Le dernier bloc ne peut être 0.

La valeur du nombre est donc `bloc[0]` $+$ `bloc[1]` $\times 2^{30} +$ `bloc[2]` $\times 2^{60} + \ldots$.

Par exemple avec $n = 537 + 5 \times 2^{30} + 7 \times 2^{60}$, on obtient trois blocs :

03	00	00	00	00	00	00	00	19	02	00	00	05	00	00	00	07	00	00	00

Les huit premiers octets donnent le nombre de blocs. Ensuite quatre octets pour le nombre 537, quatre octets pour le nombre 5 et quatre octets pour le nombre 7.

1.4 Calculs

On pourrait utiliser des nombres de type float pour représenter des grands nombres mais même les calculs les plus simples sont alors des calculs approchés.

Alors que les calculs effectués avec des entiers sont des calculs exacts. Cependant, pour des entiers très grands, stockés en Python sur plusieurs blocs, il faut gérer les opérations qui sont données à exécuter au processeur, en particulier utiliser des algorithmes de calcul performants.

Prenons par exemple le produit de deux entiers positifs $p = a + b \times 2^{30}$ et $q = c + d \times 2^{30}$. Une simple application de la distributivité donne pour résultat $r = ac + (ad + bc) \times 2^{30} + bd \times 2^{60}$.

Il y a donc quatre produits à calculer avec des nombres compris entre 0 et 2^{30}. La valeur de chaque produit est comprise entre 0 et 2^{60} et le résultat final est compris entre 0 et 2^{120}.

Un algorithme comme celui de Karatsuba, développé dans les années 1960, utilise une stratégie plus performante.

1.5 Algorithme de Karatsuba

On utilise la base dix pour simplifier la présentation et on considère $p = a10^m + b$ et $q = c10^m + d$.

Par exemple, avec $m = 2$, si $n = 6738$ alors $n = 67 \times 10^2 + 38$.

La stratégie utilisée lorsqu'on pose la multiplication des nombres p et q est une application de la distributivité. On obtient $pq = ac10^{2m} + (ad + bc)10^m + bd$

On est ramené aux quatre produits ac, ad, bc et bd et en terme de complexité on peut montrer que l'on ne gagne rien.

On écrit alors $pq = ac10^{2m} + (ac + bd - (a - b)(c - d))10^m + bd$. Sous cette forme il n'y a que trois produits à calculer : ac, bd et $(a - b)(c - d)$. On montre alors que la gain en terme de complexité est effectif.

Pour l'algorithme, nous séparons les produits à calculer en transformant l'expression :

$$pq = (10^{2m} + 10^m)ac + 10^m(b - a)(c - d) + (10^m + 1)bd$$

Cette formule peut être utilisée dans un processus récursif.

```
def karatsuba(p, q, m=9):
    if p < q:
        p, q = q, p
    if q == 0:
        return 0
    if q <= 10**m:
        return p * q
    a, b = p // 10**m, p % 10**m
    c, d = q // 10**m, q % 10**m
    ac = karatsuba(a, c)
    bd = karatsuba(b, d)
    b_a_c_d = karatsuba(b-a, c-d)
    r = (10**(2*m) + 10**m) * ac + 10**m * b_a_c_d + (10**m + 1) * bd
    return r
```

La valeur par défaut du paramètre `m` est 9. Cela correspond à une taille de mot 10^9, soit un peu moins que 2^{30}.

L'algorithme est de type *diviser pour régner*.