Combien de bits pour un entier ?

Doc. 4,2

Objectif

Savoir combien de bits il faut pour écrire un entier, une somme ou un produit, et comprendre ce qui se passe lorsqu'un résultat ne tient plus dans la place que la machine lui a réservée.

Le 4 juin 1996, la fusée Ariane 5 explose 37 secondes après son premier décollage. L'enquête montre qu'un programme de navigation a voulu ranger une vitesse dans une case mémoire de 16 bits. La valeur était trop grande pour cette case : le calculateur a produit une erreur, et la fusée a perdu le contrôle de sa trajectoire.

Q1. Avant toute recherche : combien de nombres entiers différents peut-on écrire avec 3 bits ? Écrire toutes les possibilités. Quel est le plus grand ?

Réponse : Avec 3 bits, on peut écrire 8 nombres : 000, 001, 010, 011, 100, 101, 110 et 111, c'est-à-dire les entiers de 0 à 7. Le plus grand est $(111)_2 = 4 + 2 + 1 = 7$.

§1. Ce que l'on peut écrire avec n bits

Chaque bit a deux valeurs possibles. Ajouter un bit double donc le nombre de possibilités : à chaque écriture déjà possible, on peut ajouter un 0 ou un 1 devant. Avec 1 bit, on a 2 possibilités ; avec 2 bits, $2 \times 2 = 4$; avec 3 bits, $2 \times 2 \times 2 = 8$.

À retenir

Avec n bits, on peut écrire 2^n nombres différents. Pour les entiers positifs, ce sont les entiers de 0 à $2^n - 1$.

Un **octet** est un groupe de 8 bits : il permet d'écrire les entiers de 0 à $2^8 - 1 = 255$.

Les processeurs manipulent les entiers dans des cases de taille fixe, le plus souvent de 8, 16, 32 ou 64 bits.

Q2. Compléter le tableau en calculant les puissances de 2 avec Python, par exemple `2 ** 16 - 1`.

taille	nombre d'octets	plus grand entier positif
8 bits		
16 bits		
32 bits		
64 bits		

Réponse :

taille	nombre d'octets	plus grand entier positif
8 bits	1	$2^8 - 1 = 255$
16 bits	2	$2^{16} - 1 = 65\,535$

32 bits	4	$2^{32} - 1 = 4\,294\,967\,295$
64 bits	8	$2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$

Chaque fois qu'on double le nombre de bits, le plus grand entier est à peu près élevé au carré : le nombre de chiffres décimaux double.

§2. Combien de bits pour écrire un entier donné ?

Pour savoir combien de bits sont nécessaires pour écrire un entier N, on cherche la plus petite puissance de 2 strictement supérieure à N.

Exemple résolu : combien de bits pour 614 ?

On encadre 614 entre deux puissances de 2 consécutives : $2^9 = 512 \leq 614 < 1024 = 2^{10}$.

Avec 9 bits, on ne peut pas dépasser $2^9 - 1 = 511$: c'est trop peu. Avec 10 bits, on va jusqu'à $2^{10} - 1 = 1023$: cela suffit. Il faut donc **10 bits** pour écrire 614.

Q3. Combien de bits faut-il pour écrire 255 ? 256 ? 1000 ? 2026 ? Vérifier avec Python : `len(bin(2026)) - 2` donne le nombre de chiffres de l'écriture binaire de 2026. Pourquoi faut-il retirer 2 ?

Réponse :
- $2^7 = 128 \leq 255 < 256 = 2^8$: 8 bits.
- $2^8 = 256 \leq 256 < 512 = 2^9$: 9 bits. Un seul de plus que 255, et il faut un bit de plus.
- $2^9 = 512 \leq 1000 < 1024 = 2^{10}$: 10 bits.
- $2^{10} = 1024 \leq 2026 < 2048 = 2^{11}$: 11 bits.

`bin(2026)` retourne `'0b11111101010'` : les deux premiers caractères `0b` ne sont pas des chiffres, il faut les retirer du compte.

Q4. Écrire la fonction `nb_bits`, sans utiliser `bin`, en cherchant la première puissance de 2 strictement supérieure à `n` :

```python
def nb_bits(n: int) -> int:
    """
    Retourne le nombre de bits nécessaires pour écrire
    l'entier n en base 2.
    Précondition : n >= 1.

    >>> nb_bits(614)
    10
    >>> nb_bits(255)
    8
    >>> nb_bits(256)
    9
    >>> nb_bits(1)
    1
```

```
    """
```

Coup de pouce

Utiliser deux variables : le nombre de bits `k` et la puissance `2 ** k`. Tant que la puissance n'est pas strictement supérieure à `n`, on ajoute un bit.

Réponse :

```python
def nb_bits(n: int) -> int:
    assert n >= 1, "n doit être supérieur ou égal à 1"
    k = 1
    while 2 ** k <= n:
        k += 1
    return k
```

On part de 1 bit. Tant que $2^k \leq n$, les k bits ne suffisent pas, et on en ajoute un. Une autre méthode consiste à compter les divisions par 2 nécessaires pour atteindre 0, comme dans la fonction `decompose` du Doc. 4,1.

§3. La taille d'une somme et d'un produit

§3.1. Poser une addition en binaire

On additionne en base 2 comme en base 10, en posant l'opération et en propageant les retenues de droite à gauche. La seule différence est qu'en base 2, on obtient une retenue dès que la somme atteint 2 : $1 + 1 = (10)_2$, on pose 0 et on retient 1.

Exemple résolu : calculer $(0110)_2 + (0111)_2$

retenues	1	1		
	0	1	1	0
+	0	1	1	1
	1	1	0	1

Colonne de droite : $0 + 1 = 1$. Deuxième colonne : $1 + 1 = (10)_2$, on pose 0 et on retient 1. Troisième colonne : $1 + 1 + 1 = (11)_2$, on pose 1 et on retient 1. Dernière colonne : $0 + 0 + 1 = 1$.

On vérifie en base 10 : $6 + 7 = 13 = (1101)_2$.

Q5. Poser les additions suivantes en binaire, puis vérifier en base 10 :

1. $(1011)_2 + (0110)_2$;
2. $(11111111)_2 + (00000001)_2$.

Réponse :

1. $(1011)_2 + (0110)_2 = (10001)_2$. En base 10 : $11 + 6 = 17 = 16 + 1$.
2. $(11111111)_2 + (00000001)_2 = (100000000)_2$. Toutes les colonnes produisent une retenue. En base 10 : $255 + 1 = 256 = 2^8$. Le résultat compte 9 chiffres : il ne tient plus sur un octet.

§3.2. Combien de bits pour le résultat ?

Q6. On additionne puis on multiplie deux entiers qui tiennent chacun sur 8 bits.

1. Quel est le plus grand résultat possible pour la somme ? pour le produit ?
2. Combien de bits faut-il, au plus, pour écrire la somme ? pour le produit ? Utiliser la fonction `nb_bits`.
3. Reprendre les mêmes questions avec deux entiers de 4 bits.

Réponse :

1. Le plus grand entier sur 8 bits est 255. La plus grande somme est $255 + 255 = 510$, le plus grand produit $255 \times 255 = 65\,025$.
2. `nb_bits(510)` vaut 9 et `nb_bits(65025)` vaut 16. Il faut 9 bits pour la somme et 16 bits pour le produit.
3. Le plus grand entier sur 4 bits est 15. $15 + 15 = 30$ s'écrit sur 5 bits et $15 \times 15 = 225$ sur 8 bits.

Q7. En s'appuyant sur ces exemples, proposer une règle qui donne le nombre de bits d'une somme, puis d'un produit, de deux entiers de n bits. Justifier la règle pour la somme.

Coup de pouce

Comparer $2 \times (2^n - 1)$ à 2^{n+1}.

Réponse : La somme de deux entiers de n bits tient sur $n + 1$ bits, et leur produit sur $2n$ bits.

Justification pour la somme : chaque entier est au plus égal à $2^n - 1$, donc la somme est au plus égale à $2 \times (2^n - 1) = 2^{n+1} - 2$, qui est inférieur à 2^{n+1}. Elle s'écrit donc avec au plus $n + 1$ bits. Plus généralement, la somme d'un entier de n bits et d'un entier de m bits tient sur $\max(n, m) + 1$ bits, et leur produit sur $n + m$ bits.

§4. Quand le résultat ne tient plus : le dépassement de capacité

Une case mémoire de 8 bits ne peut pas contenir de neuvième bit. Lorsqu'un calcul produit un résultat trop grand, le bit en trop est perdu : seuls les 8 bits de droite sont conservés. On parle de **dépassement de capacité** (en anglais *overflow*).

Q8. On calcule $200 + 100$ avec des entiers rangés sur 8 bits.

1. Écrire 200, 100 et 300 en base 2 avec Python.
2. Que reste-t-il de 300 si l'on ne garde que les 8 bits de droite ? Quelle valeur la machine obtient-elle ?
3. Vérifier que cette valeur est égale à `300 % 256`. Expliquer pourquoi.

Réponse :

1. `bin(200)` donne `'0b11001000'`, `bin(100)` donne `'0b1100100'` et `bin(300)` donne `'0b100101100'`, qui compte 9 bits.
2. On perd le bit de gauche, qui vaut $2^8 = 256$. Il reste $(00101100)_2 = 32 + 8 + 4 = 44$. La machine affirme que $200 + 100 = 44$.
3. `300 % 256` vaut bien 44 : supprimer le bit de poids 2^8 revient à retirer 256, c'est-à-dire à prendre le reste de la division par 256.

C'est ce qui s'est passé sur Ariane 5 : la valeur à ranger dépassait la capacité d'une case de 16 bits. Ce type d'erreur ne se voit pas à la lecture du programme : il n'apparaît que pour certaines valeurs.

Q9. Dans la console, calculer `2 ** 100`, puis `2 ** 100 + 1`. Python signale-t-il un dépassement de capacité ? Combien de bits faut-il pour écrire `2 ** 100` ?

Réponse :

```
>>> 2 ** 100
1267650600228229401496703205376
>>> 2 ** 100 + 1
1267650600228229401496703205377
```

Python ne signale aucune erreur et le calcul est exact. Il faut 101 bits pour écrire 2^{100}, qui s'écrit 1 suivi de cent 0 en base 2.

En Python, les entiers sont **de taille arbitraire** : Python utilise autant de bits qu'il en faut, tant que la mémoire de l'ordinateur le permet. Ce n'est pas le cas de la plupart des langages, ni du processeur lui-même, qui travaille sur des cases de 32 ou 64 bits.

§5. Ajouter ou retirer un zéro à droite

En base 10, ajouter un 0 à droite d'un nombre le multiplie par 10 : 42 devient 420.

Q10.
1. Que vaut $(1101)_2$? Que vaut $(11010)_2$, obtenu en ajoutant un 0 à droite ?
2. Prévoir ce que donne l'ajout de trois 0 à droite. Vérifier avec `int("1101000", 2)`.
3. Que se passe-t-il quand on retire le chiffre de droite de $(1101)_2$? Quelle opération Python donne directement ce résultat ?

Réponse :
1. $(1101)_2 = 13$ et $(11010)_2 = 16 + 8 + 2 = 26$: ajouter un 0 à droite multiplie par 2, car chaque chiffre monte d'une position et voit son poids doubler.
2. Trois 0 multiplient par $2^3 = 8$: $13 \times 8 = 104$, et `int("1101000", 2)` vaut bien 104.
3. On obtient $(110)_2 = 6$: retirer le chiffre de droite revient au quotient de la division par 2. En Python, `13 // 2` vaut 6. C'est exactement le calcul effectué à chaque étape de la méthode des divisions successives.

Ces opérations s'appellent des **décalages** : décaler vers la gauche de k positions multiplie par 2^k, et décaler vers la droite divise par 2^k (division entière). Les processeurs les effectuent beaucoup plus vite qu'une multiplication quelconque.

§6. Bilan

À l'issue de cette activité, je sais…

- expliquer qu'avec n bits on écrit 2^n nombres, les entiers de 0 à $2^n - 1$;
- citer les tailles courantes des entiers (8, 16, 32 et 64 bits) et le plus grand entier positif d'un octet ;
- déterminer le nombre de bits nécessaires pour écrire un entier, à la main et avec une fonction ;
- poser une addition en binaire en propageant les retenues ;
- évaluer le nombre de bits d'une somme ($n + 1$) et d'un produit ($2n$) de deux entiers de n bits ;
- expliquer ce qu'est un dépassement de capacité et calculer le résultat obtenu par la machine ;
- savoir que les entiers de Python sont de taille arbitraire ;
- multiplier ou diviser par une puissance de 2 en décalant l'écriture binaire.

Dans le Doc. 4,3, on se demande comment écrire un entier négatif avec seulement des 0 et des 1. L'addition binaire et le dépassement de capacité y joueront un rôle inattendu.