Ajouter deux nombres exprimés en base deux

18

Pour faire une addition, l'ordinateur fait comme on lui a appris sur les bancs de l'école.

Dans ce chapitre, nous détaillons l'algorithme de l'addition en base deux, ce qui est surtout un prétexte pour comprendre comment démontrer qu'un algorithme est correct.

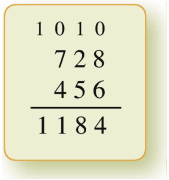
L'algorithme est le même que celui que nous utilisons couramment lorsque nous effectuons une addition ordinaire, c'est-à-dire en base dix. Nous démontrons ensuite que l'algorithme que nous avons programmé calcule bien la somme de deux nombres. Pour cela, nous utilisons la notion importante d'invariant de boucle qui est une propriété vraie à chaque tour de boucle. Nous montrons un telle propriété par récurrence sur le numéro du tour de boucle.

Ada Lovelace (1815-1852) est l'auteur du premier algorithme destiné à être exécuté par une machine. Cet algorithme, qui permettait de calculer une suite de nombres de Bernouilli, devait être exécuté sur la machine analytique conçue par Charles Babbage. Malheureusement, Babbage n'a jamais réussi à terminer sa machine. Ada Lovelace est parfois considérée comme le premier programmeur de l'histoire. Le langage de programmation Ada est ainsi nommé en son honneur.

Nous revenons dans ce chapitre sur l'un des premiers algorithmes que nous ayons appris à l'école, celui qui permet d'ajouter deux nombres entiers, pour nous poser deux questions : comment adapter cet algorithme aux nombres exprimés en base deux ? Et pourquoi cet algorithme calcule-t-il bien la somme des deux nombres ?

L'addition

Commençons par rappeler cet algorithme sur un exemple. On veut ajouter les nombres 728 et 456.

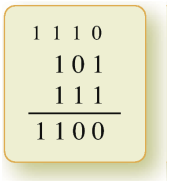


On commence par ajouter les chiffres des unités, 8 et 6. La table de l'addition indique que la somme de ces deux chiffres est 14 ; on pose le chiffre des unités, 4, et on retient le chiffre des dizaines, 1. On ajoute ensuite les chiffres des dizaines et cette retenue, 2, 5 et 1. La table de l'addition indique que la somme de ces trois chiffres est 8 ; on pose le chiffre des unités, 8, et on retient le chiffre des dizaines, 0. On ajoute ensuite les chiffres des centaines et cette retenue, 7, 4 et 0. La table de l'addition indique que la somme de ces trois chiffres est 11 ; on pose le chiffre des unités, 1, et on retient le chiffre des dizaines, 1. Finalement, on pose cette retenue dans la colonne des milliers.

Une irrégularité de cette méthode est que, lors de la première itération, on ajoute deux chiffres, alors qu'en régime permanent, on en ajoute trois. On peut corriger cela en commençant par poser la retenue égale à 0. La première itération se formule alors de la manière suivante : on commence par ajouter les chiffres des unités et la retenue, 8, 6 et 0, etc. Ainsi, cet algorithme n'utilise qu'une seule table, qui indique la somme de chacun des triplets $(a ; b ; c)$ où a, b et c sont des chiffres compris entre 0 et 9, table qu'en général on connaît par cœur.

L'addition pour les nombres exprimés en base deux

Voyons maintenant comment on ajoute des nombres exprimés en base deux, par exemple 101 et 111, c'est-à-dire 5 et 7.



On commence, comme en base dix, par ajouter les chiffres des unités et la retenue, 1, 1 et 0. La table de l'addition indique que la somme de ces deux chiffres est 10 ; on pose le chiffre des unités, 0, et on retient le chiffre des deuzaines, 1. On ajoute ensuite les chiffres des deuzaines et cette retenue, 0, 1 et 1. La table de l'addition indique que la somme de ces trois chiffres est 10 ; on pose le chiffre des unités, 0, et on retient le chiffre des deuzaines, 1. On ajoute ensuite les chiffres des quatraines et cette retenue, 1, 1 et 1. La table de l'addition indique que la somme de ces trois chiffres est 11 ; on pose le chiffre des unités, 1, et on retient le chiffre des deuzaines, 1. Finalement, on pose cette retenue dans la colonne des huitaines. Le résultat est donc 1100, c'est-à-dire 12.

Cette méthode utilise une table qui indique la somme de chacun des triplets $(a ; b ; c)$ où a, b et c sont des chiffres compris entre 0 et 1. Cette table ne contient donc que huit lignes :

a	b	c	a + b + c
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	10
1	0	0	1
1	0	1	10
1	1	0	10
1	1	1	11

En fait, cette méthode se formule mieux en utilisant deux tables. La première indique le chiffre des unités de $a + b + c$:

a	b	c	Unités de a + b + c
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

La seconde indique le chiffre des deuzaines de $a + b + c$:

a	b	c	Deuzaines de a + b + c
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Il s'agit là des tables de deux fonctions booléennes à trois arguments. Comme toutes les fonctions booléennes, elles peuvent s'exprimer avec les fonctions *non*, *et* et *ou* (voir le chapitre 10). Une manière, parmi d'autres, de les exprimer est la suivante :

```
unités(a,b,c)
= (a et non(b) et non(c)) ou (non(a) et b et non(c))
  ou (non(a) et non(b) et c) ou (a et b et c)
deuzaines(a,b,c) = (a et b) ou (b et c) ou (a et c)
```

Pour se convaincre de la correction de ces expressions, il suffit de vérifier qu'elles donnent bien les chiffres des unités et des deuzaines de $a + b + c$ dans chacun des huit cas des tables précédentes. Par exemple, dans le cas $a = 0$, $b = 1$ et $c = 1$, l'expression *a et non*(*b*) *et non*(*c*) prend la valeur 0, les expressions *non*(*a*) *et b et non*(*c*), *non*(*a*) *et non*(*b*) *et c* et *a et b et c* prennent elles aussi la valeur 0 et donc l'expression (*a et non*(*b*) *et non*(*c*)) *ou* (*non*(*a*) *et b* et *non*(*c*)) *ou* (*non*(*a*) *et non*(*b*) *et c*) *ou* (*a et b et c*) prend la

valeur 0 également, ce qui est bien le chiffre des unités de $a + b + c$. De même, les expressions *a et b*, *b et c* et *a et c* prennent respectivement les valeurs 0, 1 et 0 et donc l'expression (*a et b*) *ou* (*b et c*) *ou* (*a et c*) prend la valeur 1, ce qui est bien le chiffre des deuzaines de $a + b + c$.

On peut, par exemple, programmer cette méthode pour ajouter deux nombres de dix chiffres binaires. Le résultat sera donc un nombre de onze chiffres. On choisit de représenter les nombres à ajouter x et y par deux tableaux de booléens de dix cases `n` et `p`, et le résultat par un tableau de booléens `r` de 11 cases. On choisit le booléen `true` pour le chiffre 1 et le booléen `false` pour le chiffre 0. La case 0 d'un tableau contient le chiffre des unités du nombre représenté, la case 1 le chiffre des deuzaines… et la case 9, le chiffre des cinq-cent-douzaines. Le résultat `r` qui a un chiffre de plus a aussi une case 10 pour les mille-vingt-quatraines.

La retenue `c` est d'abord initialisée à 0 (1). Puis on calcule les chiffres du résultat l'un après l'autre par une boucle dont l'indice `i` varie de 0 à 9. À chaque étape, on définit le chiffre `a` comme le `i`-ème chiffre du nombre `n` (2) et `b` comme le `i`-ème chiffre du nombre `p` (3), puis on affecte la case `i` du tableau `r` (4) avec le chiffre des unités de `a + b + c`. Enfin, on affecte la retenue `c` avec le chiffre des deuzaines de `a + b + c` (5). Et une fois la boucle terminée, on affecte la case 10 du tableau `r` avec la dernière des retenues (6).

```
c = false;1
for (i = 0; i <= 9; i = i + 1) {
  a = n[i];2
  b = p[i];3
  r[i] = (a && !b && !c) || (!a && b && !c) || (!a && !b && c)
         || (a && b && c);4
  c = (a && b) || (b && c) || (a && c);}5
r[10] = c;6
```

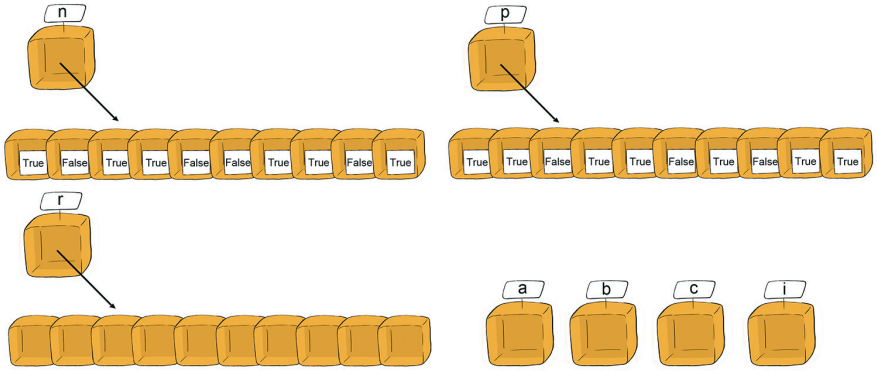
Exercice 18.1

On utilise ce programme pour ajouter les nombres $x = 1011001101$ et $y = 1101101011$. Exécuter l'instruction `c = false;` l'initialisation de la variable `i` et le tour 0 de la boucle revient à exécuter la séquence d'affectations suivante :

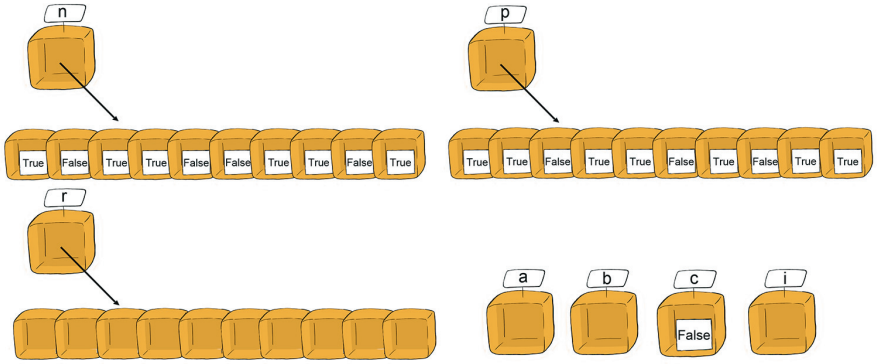
```
c = false;
i = 0;
a = n[0];
b = p[0];
r[0] = (a && !b && !c) || (!a && b && !c) || (!a && !b && c)
       || (a && b && c);
c = (a && b) || (b && c) || (a && c);
i = i + 1;
```

Si on exécute cette séquence dans l'état 1, la première affectation `c = false;` donne l'état 2.

1



2



Dessiner les états successifs produits par l'exécution de chacune de ces affectations. Quel est l'état final produit par l'exécution du tour 0 de la boucle ?

Montrer que dans cet état :

$$(r[0] \times 2^0) + c \times 2^1 = (n[0] \times 2^0) + (p[0] \times 2^0)$$

exécuter le tour 1 de la boucle revient à exécuter la séquence d'affectations suivante :

```
a = n[1];
b = p[1];
r[1] = (a && !b && !c) || (!a && b && !c) || (!a && !b && c)
       || (a && b && c);
c = (a && b) || (b && c) || (a && c);
i = i + 1;
```

Quel est l'état produit par l'exécution de ce tour de boucle ?

Montrer que dans cet état :

$(r[0] \times 2^0 + r[1] \times 2^1) + c \times 2^2 = (n[0] \times 2^0 + n[1] \times 2^1) + (p[0] \times 2^0 + p[1] \times 2^1)$

La démonstration de correction du programme

Quand on conçoit un tel programme, une question se pose naturellement : comment sait-on qu'il calcule la somme des deux nombres entiers ?

Une première manière de s'assurer qu'un programme fait bien ce qu'on attend de lui est de le tester (voir le chapitre 1). Il faut essayer différentes valeurs pour les nombres x et y et vérifier que le programme affiche bien la valeur $x + y$ dans tous les cas. On estime, en général, que le coût du test d'un programme est du même ordre de grandeur que celui de son développement. Cependant, le test présente deux limites importantes : la première est que l'on ne peut pas tester le programme sur toutes les valeurs d'entrée possibles, qui sont souvent très nombreuses, voire en nombre infini. La seconde est que pour tester un programme, il faut savoir ce que l'on attend de lui. Or, ce n'est pas le cas quand on écrit, par exemple, un programme qui calcule la millième décimale du nombre π, ou la position de la Lune dans mille ans, car la raison pour laquelle on écrit un tel programme est précisément que l'on ignore la millième décimale du nombre π ou la position de la Lune dans mille ans. De ce fait, comment le tester ?

Une autre manière de s'assurer qu'un programme fait bien ce qu'on attend de lui est de le démontrer. Par exemple, on peut démontrer que le programme précédent calcule bien la somme des nombres x et y. Plus précisément, on veut démontrer que si, au moment où l'on exécute ce programme, les tableaux `n` et `p` contiennent la représentation binaire de deux entiers de dix chiffres, c'est-à-dire si :

$x = n[0] \times 2^0 + n[1] \times 2^1 + \ldots + n[8] \times 2^8 + n[9] \times 2^9$

et :

$y = p[0] \times 2^0 + p[1] \times 2^1 + \ldots + p[8] \times 2^8 + p[9] \times 2^9$

alors à la fin de l'exécution de ce programme, le tableau `r` contient un nombre de onze chiffres qui est la représentation binaire de $x + y$, c'est-à-dire que :

$x + y =$ `r[0]` $\times 2^0 +$ `r[1]` $\times 2^1 + \ldots +$ `r[9]` $\times 2^9 +$ `r[10]` $\times 2^{10}$.

Le programme qui ajoute deux nombres exprimés en binaire est formé d'une boucle, dans laquelle on calcule d'abord le chiffre des unités, puis les chiffres des deuzaines, des quatraines, etc. du résultat. Après avoir achevé les tours 0…, `i - 1` et au moment de commencer le tour `i`, on a donc calculé la somme des deux nombres formés des `i - 1` premiers chiffres, en partant de la droite, des nombres x et y. C'est l'invariant que l'on va montrer par récurrence. À la fin de la boucle, l'invariant indiquera que l'algorithme a effectué l'addition souhaitée.

Par exemple, si au cours de l'addition de $x =$ 1011001101 et $y =$ 1101101011 (1) on s'arrête après avoir effectué les tours 0 et 1 de la boucle et avant de commencer le tour 2, on a déjà calculé la somme des nombres 01 et 11, c'est-à-dire 1 et 3 (2). Le résultat de cette addition n'est pas exactement le nombre représenté par les deux chiffres déjà posés 00, car il faut tenir compte de la retenue. La propriété exacte est que si on pose la retenue dans la colonne `i`, ce qui donne dans cet exemple le nombre 100 c'est-à-dire 4, on obtient la somme des deux nombres formés des `i - 1` premiers chiffres des nombres x et y.

1

```
1 1 1 1 0 0 1 1 1 1
  1 0 1 1 0 0 1 1 0 1
  1 1 0 1 1 0 1 0 1 1
-------------------
1 1 0 0 0 1 1 1 0 0 0
```

2

```
          1 1 1
  1 0 1 1 0 0 1 1 0 1
  1 1 0 1 1 0 1 0 1 1
-------------------
              0 0
```

Autrement dit, au moment de commencer le tour `i` de la boucle, l'état vérifie la propriété :

(`r[0]` $\times 2^0 + \ldots +$ `r[i-1]` $\times 2^{i-1}$) + `c` $\times 2^i$
$=$ (`n[0]` $\times 2^0 + \ldots +$ `n[i-1]` $\times 2^{i-1}$) + (`p[0]` $\times 2^0 + \ldots +$ `p[i-1]` $\times 2^{i-1}$)

On démontre maintenant cette propriété.

À la première exécution, `i` $= 0$, la somme (`r[0]` $\times 2^0 + \ldots +$ `r[i-1]` $\times 2^{i-1}$) ne contient aucun terme ; elle vaut donc 0. Il en est de même pour les sommes (`n[0]` $\times 2^0 + \ldots +$ `n[i-1]` $\times 2^{i-1}$) et (`p[0]` $\times 2^0 + \ldots +$ `p[i-1]` $\times 2^{i-1}$). Comme par ailleurs, la retenue `c` vaut 0, les deux membres de l'égalité sont nuls.

Invariant

Un *invariant* d'une boucle est une propriété qui est vérifiée à chaque exécution du corps de cette boucle. En général, pour la dernière exécution, cette propriété traduit le fait que la boucle réalise bien la tâche souhaitée. On montre qu'une propriété est un invariant d'une boucle par un raisonnement par récurrence :

- on montre que la propriété est vérifiée à la première exécution du corps de la boucle,
- on montre que si l'invariant est vérifié à une exécution donnée du corps de la boucle, il est encore vérifié à l'exécution suivante.

L'invariant est alors vérifié à la fin de boucle, qui fournit donc le résultat attendu.

On suppose maintenant que cette propriété est vérifiée dans l'état dans lequel s'exécute le tour `i` de la boucle et on veut montrer qu'elle est encore vérifiée dans l'état dans lequel s'exécute le tour suivant. Au début du tour `i`, on a :

$$(r[0] \times 2^0 + \ldots + r[i-1] \times 2^{i-1}) + c \times 2^i$$
$$= (n[0] \times 2^0 + \ldots + n[i-1] \times 2^{i-1}) + (p[0] \times 2^0 + \ldots + p[i-1] \times 2^{i-1})$$

et donc en ajoutant $n[i] \times 2^i + p[i] \times 2^i$ dans les deux membres de l'égalité on obtient que, au début du tour `i` de la boucle :

$$(r[0] \times 2^0 + \ldots + r[i-1] \times 2^{i-1}) + (n[i] + p[i] + c) \times 2^i$$
$$= (n[0] \times 2^0 + \ldots + n[i] \times 2^i) + (p[0] + \ldots + p[i] \times 2^i)$$

Au cours de ce tour de la boucle, on ajoute les trois chiffres `n[i]`, `p[i]` et `c`, et le résultat de cette addition a pour chiffre des unités `r[i]` et pour chiffre des deuzaines la nouvelle valeur de `c`, si bien que, dans l'état atteint à la fin de ce tour de la boucle :

$$(r[0] \times 2^0 + \ldots + r[i-1] \times 2^{i-1}) + (r[i] + 2 \times c) \times 2^i$$
$$= (n[0] \times 2^0 + \ldots + n[i] \times 2^i) + (p[0] \times 2^0 + \ldots + p[i] \times 2^i)$$

c'est-à-dire :

$$(r[0] \times 2^0 + \ldots + r[i] \times 2^i) + c \times 2^{i+1}$$
$$= (n[0] \times 2^0 + \ldots + n[i] \times 2^i) + (p[0] \times 2^0 + \ldots + p[i] \times 2^i)$$

Au début du tour suivant, la variable `i` a été augmentée de 1, si bien que :

$$(r[0] \times 2^0 + \ldots + r[i-1] \times 2^{i-1}) + c \times 2^i$$
$$= (n[0] \times 2^0 + \ldots + n[i-1] \times 2^{i-1}) + (p[0] \times 2^0 + \ldots + p[i-1] \times 2^{i-1})$$

La propriété est donc encore vérifiée au début du tour suivant. Elle est donc vérifiée à chacun des tours de boucles : c'est un invariant de la boucle.

À la fin du dernier tour, `i` est égal à 10 et donc :

$$(r[0] \times 2^0 + \ldots + r[9] \times 2^9) + c \times 2^{10}$$
$$= (n[0] \times 2^0 + \ldots + n[9] \times 2^9) + (p[0] \times 2^0 + \ldots + p[9] \times 2^9)$$
$$= x + y$$

On affecte alors la case 10 du tableau `r` avec la retenue si bien que, quand l'exécution est terminée :

$$r[0] \times 2^0 + \ldots + r[10] \times 2^{10} = x + y$$

C'est ce qu'il fallait démontrer : le tableau `r` contient la représentation binaire du nombre $x + y$.

ALLER PLUS LOIN **L'autonomie de la notion d'algorithme**

Dans ce chapitre, nous avons étudié un programme, écrit en Java, qui additionne deux nombres écrits en base deux. Il est possible d'écrire des programmes très similaires dans d'autres langages de programmation. La méthode pour ajouter deux nombres en base deux est indépendante d'un langage de programmation particulier : c'est une méthode abstraite qui peut s'exprimer dans divers langages. Une telle méthode systématique qui permet de résoudre un problème s'appelle un *algorithme*. Il est important de distinguer un algorithme, méthode indépendante de tout langage, d'un *programme*, qui est l'incarnation d'un algorithme dans un langage particulier.

Cette distinction entre les notions d'algorithme et de programme doit également son importance au fait que nous ayons utilisé des algorithmes pour faire des additions dans diverses bases depuis des millénaires, bien avant que nous ayons pensé à exprimer cet algorithme dans un langage de programmation. Nous avons même utilisé des algorithmes, transmis de génération en génération par observation et imitation, pour fabriquer des objets en céramique, tisser des étoffes, nouer des cordages, préparer les aliments, etc. avant l'invention de l'écriture.

ALLER PLUS LOIN **Définitions algorithmiques et non algorithmiques**

L'apprentissage des mathématiques commence par l'apprentissage d'algorithmes qui permettent d'effectuer des additions, des soustractions, etc.
Même au-delà de ces mathématiques élémentaires, beaucoup de définitions mathématiques sont algorithmiques. Par exemple, la définition du nombre $n - m$ comme le nombre obtenu en mettant n cailloux dans un sac, en en ôtant m et en comptant ceux qui restent est algorithmique. Mais la définition du nombre $n - m$ comme le nombre p tel que $p + m = n$ ne l'est pas : contrairement à la première, cette définition ne dit pas ce que l'on doit faire pour connaître le nombre $n - m$ quand on connaît les nombres n et m.

De même, la définition selon laquelle deux vecteurs non nuls du plan, donnés par leurs coordonnées $(x_1 ; y_1)$ et $(x_2 ; y_2)$ dans une base, sont colinéaires quand $x_1 y_2 = x_2 y_1$ est algorithmique, mais pas celle selon laquelle ces deux vecteurs sont colinéaires s'il existe un facteur de proportion k tel que $x_1 = k x_2$ et $y_1 = k y_2$. Si deux vecteurs sont donnés par leurs coordonnées dans une base, par exemple (4 ; 10) et (6 ; 15), la première définition donne une méthode pour déterminer s'ils sont colinéaires, puisqu'il suffit de calculer 4×15 et 10×6 et de vérifier que l'on obtient bien le même nombre dans les deux cas, mais pas la seconde, qui demande de trouver le facteur de proportion, sans indiquer de méthode pour le faire.

Ai-je bien compris ?

- En quelles bases l'algorithme de l'addition peut-il être utilisé ?
- Que veut-on dire lorsqu'on affirme que l'algorithme de l'addition est correct ?
- Qu'est-ce qu'un invariant ?