LOGIQUE & CALCUL

Le calculateur amnésique

Un calculateur sans mémoire est-il sérieusement limité ? Les résultats de l'algorithmique in situ *montrent que non : avec un peu d'astuce, il s'en sortira toujours.*

Jean-Paul DELAHAYE

J'ai la mémoire qui flanche
Je me souviens plus très bien
De quelle couleur étaient ses yeux
Je crois pas qu'ils étaient bleus
Étaient-ils verts ? Étaient-ils gris ?
Étaient-ils vert-de-gris ?
Ou changeaient-ils tout le temps de couleur
Pour un non, pour un oui ...
Jeanne Moreau, Cyrus Bassiak
et François Rauber

Si vous avez déjà écrit des programmes informatiques, vous avez nécessairement été confronté au problème suivant : deux nombres sont stockés dans les variables A et B, par exemple $A = 44$ et $B = 13$, et vous voulez échanger le contenu de A et B pour obtenir $A = 13$ et $B = 44$. La méthode classique, que tout le monde réinvente, consiste à utiliser une variable C et à écrire la séquence de trois instructions : $C := A$; $A := B$; $B := C$.

Le signe := est celui de l'affectation. L'instruction X := Y signifie que nous lisons le contenu de Y et que nous le mettons dans X, ce qui efface le contenu de X sans changer celui de Y. Cette séquence de trois affectations pour l'échange de deux valeurs exige l'utilisation d'une variable C supplémentaire. Or nous pouvons échapper à l'introduction de ce complément de mémoire par la séquence suivante : $A := A + B$; $B := A - B$; $A := A - B$. Le déroulement du calcul donne :

	A	B
Au départ :	44	13
Après $A := A + B$:	57	13
Après $B := A - B$:	57	44
Après $A := A - B$:	13	44

La suite d'instructions est valable avec n'importe quelles valeurs à la place de 44 et 13. Elle fonctionne aussi avec des valeurs booléennes où les signes + et – sont remplacés par *xor* qui désigne l'opération « ou exclusif » (0 xor 0 = 0 ; 0 xor 1 = 1 ; 1 xor 0 = 1 ; 1 xor 1 = 0). La suite d'instructions est :
$A := A$ xor B ; $B := A$ xor B ; $A := A$ xor B, où A et B valent 0 ou 1.

Vérifions que la séquence permute bien les valeurs de A et B en examinant l'effet de chaque opération. La première affectation met (A xor B) dans A. La deuxième met donc ((A xor B) xor B) dans B ; mais, d'après l'associativité du xor, cela vaut (A xor (B xor B)) c'est-à-dire A, puisque (B xor B) = 0 et que A xor 0 = A. La troisième affectation met donc (A xor B) xor A dans A ; comme (A xor B) xor A = (A xor (B xor A)), cela donne, par commutativité, (A xor (A xor B)) = ((A xor A) xor B) = (0 xor B) = B. Il y a bien échange.

Ce type de calcul est dénommé calcul *in situ* ou *calcul sans mémoire*. L'idée est d'opérer les calculs sans faire appel à des variables de stockage autres que celles qui communiquent les données au programme, que l'on utilise pour calculer et pour placer les résultats une fois les opérations terminées. L'étude de ce calcul *in situ* a pour fonction d'identifier ce que peut calculer un amnésique qui, ne disposant de rien d'autre que l'espace mémoire des données du calcul, le réutilise car il n'a en propre aucune mémoire à sa disposition.

Avant de donner des détails sur les résultats obtenus par les chercheurs qui étudient cette algorithmique *in situ*, et pour bien en mesurer la difficulté, posons-nous une petite énigme.

Considérons une permutation circulaire de trois données A, B, C, où l'on cherche à passer de $A = 3$, $B = 50$, $C = 700$ à $A = 50$, $B = 700$, $C = 3$. Nous voulons pour ce faire utiliser un programme *in situ*, c'est-à-dire n'utilisant aucune autre variable que A, B et C, et bien sûr nous voulons qu'il fonctionne pour tout jeu de données.

Permutation circulaire de trois valeurs

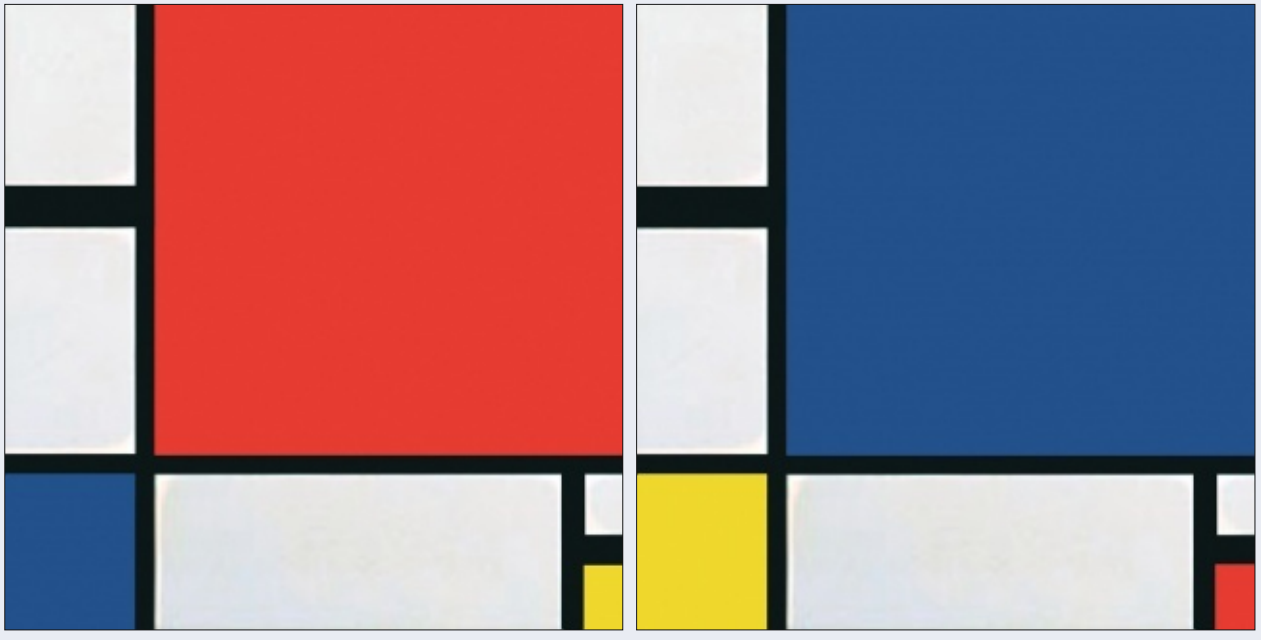
En voici un, avec le détail de l'état de chacune des variables lors du calcul :

	A	B	C
	3	50	700
$A := A + B$	53	50	700
$B := B + C$	53	750	700
$C := A - B + C$	53	750	3
$B := B - A + C$	53	700	3
$A := A - C$	50	700	3

En remplaçant les + et les – par des *xor*, on obtient, comme précédemment, un programme pour variables booléennes. Une autre méthode aurait consisté à échanger les valeurs de A et B, puis à échanger les valeurs de B et C en utilisant deux fois la méthode vue précédemment. Cela aurait nécessité six affectations alors que notre proposition n'en utilise que cinq. Peut-on faire mieux que cinq affectations ? La solution est donnée dans l'encadré ci-contre.

Expliquons maintenant avec plus de détails ce que les chercheurs dénomment calcul *in situ*. Un ensemble fini S est donné

1. La permutation circulaire



La question posée est : quel est le minimum d'affectations dont on a besoin dans un calcul *in situ* pour échanger circulairement les contenus de trois variables A, B, C, (A, B, C) devenant (B, C, A) ? Dans le tableau de Mondrian et son faux, trois couleurs ont été ainsi permutées. Petit exercice : quel est le vrai, quel est le faux ?

Notons d'abord que trois affectations ne suffiront pas. Il faudrait en effet, dès la première affectation, placer une bonne valeur là où on fait l'affectation, ce qui ferait disparaître la valeur qui y est stockée, dès lors irrécupérable.

En revanche, réaliser la permutation circulaire avec quatre affectations est possible, donc 4 est le nombre minimum d'affections nécessaires à la permutation circulaire de trois nombres. Voici la solution en quatre affectations :

	A	B	C
Au départ	A	B	C
A := A + B + C	A+B+C	B	C
C := A – B – C	A+B+C	B	A
B := A – B – C	A+B+C	C	A
A := A – B – C	B	C	A

Le procédé se généralise pour opérer une permutation circulaire entre n mémoires, c'est-à-dire pour passer de $(A_1A_2A_3 \ldots A_{n-1}A_n)$ à $(A_2A_3 \ldots A_{n-1}A_nA_1)$.

À la première étape, on place dans A_1 la somme $A_1 + A_2 + A_3 + \ldots + A_{n-1} + A_n$, puis, à chaque étape, toutes les valeurs initiales des variables sont présentes dans $A_2, A_3, \ldots A_{n-1}, A_n$ sauf une, ce qui permet, grâce à A_1, de la placer au bon endroit.

	A_1	A_2	A_3	.....	A_{n-1}	A_n
Au départ	A_1	A_2	A_3	.....	A_{n-1}	A_n
$A_1 := A_1 + A_2 + \ldots + A_n$	$A_1 + A_2 + \ldots + A_n$	A_2	A_3	.....	A_{n-1}	A_n
$A_n := A_1 - A_2 - \ldots - A_n$	$A_1 + A_2 + \ldots + A_n$	A_2	A_3	.....	A_{n-1}	A_1
$A_{n-1} := A_1 - A_2 - \ldots - A_n$	$A_1 + A_2 + \ldots + A_n$	A_2	A_3	.....	A_n	A_1
.....	.....	...	...	.....	...	...
$A_1 := A_1 - A_2 - \ldots - A_n$	A_2	A_3	A_4		A_n	A_1

L'affectation opérée pour A_n place donc bien la valeur initiale de A_1 en A_n, puis l'affectation opérée pour A_{n-1} place donc bien la valeur initiale de A_2 en A_{n-1}, etc. Le calcul est présenté avec des + et des –, et fonctionne avec des nombres entiers, réels ou même complexes.

Le calcul s'adapte aussi avec des nombres calculés modulo k, ou aux variables booléennes (en remplaçant les + et les – par *xor*, ce qui correspond d'ailleurs au + quand on calcule modulo 2). Cette remarque est importante car elle permet de répondre à une objection naturelle, mais essentielle, à la méthode proposée pour opérer une permutation circulaire de n entiers.

Voici l'objection : quand on calcule avec des entiers, la méthode de calcul *in situ* proposée triche, car elle stocke dans A_1 la somme des valeurs de toutes les variables, ce qui exige un espace mémoire plus grand pour la variable A_1 : s'il y a par exemple 10 variables variant entre 0 et 10, il faut pour stocker A_1 une variable capable de recevoir toutes les valeurs entre 0 et 100.

Voici la réponse à l'objection. Oui, cela est vrai pour la méthode avec des + et des –, c'est-à-dire avec les opérations arithmétiques habituelles entre nombres entiers. Cependant, puisqu'on peut mener un calcul modulo k (où k est un entier fixé), on dispose dans ce cas d'une méthode qui n'utilise pas plus d'espace pour A_1.

Une autre idée pour répondre à l'objection consiste à écrire les nombres entiers en base 2 et, au lieu de faire des additions ou des soustractions entre deux nombres X et Y, on calcule le nombre dont les chiffres binaires sont ceux obtenus en faisant le *xor* entre les chiffres binaires de X et de Y (exemple 1001 xor 0111 = 1110, autrement dit, 9 xor 7 = 14) et l'on retombe bien sur ses pieds à la fin.

L'espace nécessaire pour stocker A_1 reste encore identique à celui nécessaire pour stocker chaque nombre (p chiffres binaires par exemple si on manipule des nombres de p chiffres binaires). Les procédés décrits (traduits avec des calculs modulo k ou des *xor*) réalisent donc un calcul *in situ* sans avoir à considérer un espace mémoire particulier pour A_1.