Techniques algorithmiques et méthodes de programmation

Paul Zimmermann

Mots-clés : algorithme déterministe, probabiliste, randomisé, en place, par contrat, sans branchement, glouton, programmation dynamique, diviser pour régner, récursivité, parcours en profondeur, en largeur, sentinelle

Résumé : Ce chapitre décrit quelques techniques algorithmiques et méthodes de programmation fondamentales, sans prétendre être exhaustif. Chaque concept est illustré par un exemple.

1 Introduction

Il est difficile de situer précisément l'origine de la notion d'« algorithme ». Certains auteurs citent Euclide (vers 300 avant J.C.) et son algorithme de calcul du plus grand commun diviseur, mais le mot « algorithme » est sans doute apparu bien après : d'autres auteurs pensent que c'est une déformation du nom du mathématicien Al Khwarizmii, qui a vécu au IX^{e} siècle.

Toujours est-il que la notion d'algorithme a connu un essor au XX^{e} siècle avec l'apparition des ordinateurs. Un algorithme est en effet une version « générique » d'un programme ; dit autrement, un programme est une déclinaison d'un algorithme dans un langage de programmation donné. Paradoxalement, il n'existe pas de véritable standard pour décrire les algorithmes (une sorte de langage universel comme l'esperanto) ; ils sont usuellement décrits en « pseudo-code », proche d'un langage de programmation, mais sans forcément en respecter la syntaxe.

Il est important de bien connaître les principales techniques algorithmiques : celles-ci permettent en effet une manipulation efficace des structures de données classiques (voir le chapitre ££« Structures de données »). Les méthodes de programmation quant à elles permettent, pour un problème donné, de trouver une solution meilleure, plus efficace ou plus esthétique.

2 Techniques algorithmiques

2.1 Algorithmes déterministes, probabilistes

Un algorithme est dit *déterministe* lorsque son comportement dépend uniquement des entrées. Une classe importante d'algorithmes non déterministes est celle des algorithmes *probabilistes*, dont l'exécution dépend de paramètres aléatoires ou pseudo-aléatoires. On distingue deux grandes sous-classes d'algorithmes probabilistes :

- les algorithmes *Monte Carlo*, où c'est l'exactitude du résultat qui est aléatoire, mais dont le temps de calcul est déterministe :

  Tri Monte Carlo.
  Entrée : liste $[a_1, \ldots, a_n]$.
  Sortie : liste presque triée.
  pour $i \leftarrow 1$ **à** n^2 **faire**
  $j \leftarrow \mathrm{rand}(1 \ldots n-1)$
  si $a_j > a_{j+1}$ **alors**
  Échange(a_j, a_{j+1})

  Dans l'algorithme ci-dessus, le nombre d'itérations est fixé à n^2, mais la liste résultat n'est pas forcément triée.
  Certains algorithmes d'intégrale numérique multidimensionnelle utilisent des points tirés aléatoirement : ce sont des algorithmes Monte Carlo.
- les algorithmes *Las Vegas*, où c'est le temps de calcul qui est aléatoire, mais dont le résultat est toujours exact :

  Tri Las Vegas.
  Entrée : liste $[a_1, \ldots, a_n]$.
  Sortie : liste triée.
  $i \leftarrow 1$
  tant que $i < n$ **faire**

$j \leftarrow \text{rand}(i \ldots n-1)$
si $a_j \leqslant a_{j+1}$ **alors**
  si $j = i$ **alors** $i \leftarrow i+1$
sinon { $a_j > a_{j+1}$ }
  Échange(a_j, a_{j+1})
  si $j = i > 1$ **alors** $i \leftarrow i-1$

Contrairement au « tri Monte Carlo », le « tri Las Vegas » produit toujours une liste triée. En effet, on peut vérifier l'invariant suivant : les éléments $a_1, \ldots, a_i$ sont triés par ordre croissant. Lorsque a_i est échangé avec a_{i+1}, la sous-liste $a_1, \ldots, a_{i-1}$ reste triée ; sinon, lorsque $a_i \leqslant a_{i+1}$, alors $a_1, \ldots, a_{i+1}$ est triée. Il reste à vérifier que l'algorithme termine. Si le générateur aléatoire est uniforme, il finira par produire $j = i$. Alors de deux choses l'une : soit $a_j > a_{j+1}$ et le nombre d'inversions de la liste diminue strictement, sinon i augmente, ce qui ne peut se faire indéfiniment. (Si on remplace $j \leftarrow \text{rand}(i \ldots n-1)$ par $j \leftarrow i$, on retrouve le tri par insertion.)

Un algorithme reliant deux nœuds d'un graphe connexe, en choisissant des arêtes aléatoirement, est aussi de type Las Vegas, puisqu'il finira par trouver le nœud cible, au bout d'un temps de parcours éventuellement très grand.

2.2 Algorithmes randomisés

Un algorithme est dit *randomisé* lorsqu'il a été modifié pour être insensible à la distribution de ses entrées. Prenons l'exemple du tri : la plupart des algorithmes ont un comportement extrêmement variable suivant la liste à trier. Ainsi, le tri par insertion est très rapide pour une liste presque triée, pour laquelle au contraire le tri rapide (*quicksort*) est très lent.

Un algorithme de tri randomisé commence par permuter de façon aléatoire la liste donnée en entrée. Ainsi, quelle que soit la distribution de cette liste (presque triée, triée en sens inverse, ou quelconque), le coût *en moyenne* de l'algorithme sera le même, puisque la liste permutée suit une distribution uniforme.

Tri randomisé.
Entrée : une liste $l = [a_1, \ldots, a_n]$.
Sortie : liste triée.
Trie(permute(l)).

Toutefois, la liste permutée peut correspondre au pire des cas : la « randomisation » n'a donc d'effet que sur le coût moyen de l'algorithme !

2.3 Algorithmes par contrat

Dans certains cas, on dispose de ressources limitées — temps de calcul et/ou espace mémoire — pour trouver une réponse à un problème donné. C'est la problématique du « temps réel », par exemple pour un joueur d'échec, trouver en moins de cinq minutes le meilleur coup. Une fois le temps imparti dépassé, le résultat n'a plus aucune valeur, aussi bon fût-il. Un algorithme *par contrat* tient compte de la limite imposée pour optimiser la recherche. On pourra ainsi, par exemple, essayer de trouver une première solution par un algorithme rapide mais pas forcément optimal, avant de lancer une recherche plus coûteuse.

L'algorithme ci-dessous illustre la programmation par contrat. Soit A un arbre de degré quelconque, dont chaque nœud n contient une clé entière $k(n)$, et dont on veut chercher une petite clé en un temps imparti T.

La signification des variables de l'algorithme est la suivante : l est la liste des sous-arbres restant à traiter, $k_{\min}$ est la plus petite clé trouvée, t le temps courant, n l'arbre courant qu'on assimile au nœud racine de cet arbre. Les fonctions début et reste renvoient respectivement le premier élément d'une liste, et la liste privée de celui-ci.

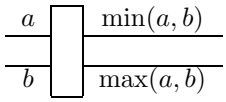
Minimum par contrat.
Entrée : un arbre A, un temps T.
Sortie : un minimum partiel de A.
$l \leftarrow [A]$
$k_{\min} \leftarrow +\infty$;
pour $t \leftarrow 1$ **à** T **tant que** $l \neq []$ **faire**
  $n \leftarrow$ début(l)
  si $k(n) < k_{\min}$ **alors** $k_{\min} \leftarrow k(n)$
  $l \leftarrow$ reste(l) :: fils(n)
Renvoyer $k_{\min}$.

La fonction fils(n) renvoie la liste des fils du nœud n, et $l :: m$ concatène les listes l et m. Cet algorithme parcourt au plus T nœuds de l'arbre A, via un parcours en largeur d'abord (cf. §3.5).

2.4 Algorithmes sans branchement

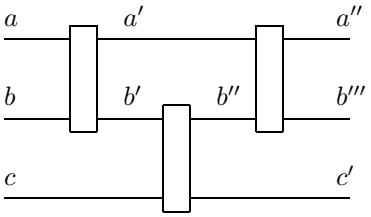
Une classe quelque peu exotique est celle des algorithmes *sans branchement*. Un tel algorithme ne comporte aucun test (**si-alors-sinon**, **tant que** ou **répéter-jusqu'à**). Son comportement est donc complètement déterministe, et peut être décrit par une simple suite d'instructions. Par exemple, un algo-

rithme de tri sans branchement est appelé « réseau de tri ». Un réseau de tri est un circuit contenant n entrées, n sorties, et un nombre quelconque de portes de comparaison, prenant chacune deux entrées a, b, et les renvoyant triées :



Une porte de comparaison.

La figure ci-dessous représente un réseau de tri avec trois portes de comparaison :



Un réseau de tri pour trois nombres.

Il suffit pour s'en convaincre de vérifier que la plus petite valeur se retrouve bien en a'', et la plus grande en c'. Un tel réseau peut aussi s'écrire sous forme linéaire :

trie(l_1, l_2)
trie(l_2, l_3)
trie(l_1, l_2)

où $l = [a, b, c]$ initialement, et la procédure trie(l_i, l_j) échange l_i et l_j si $l_i > l_j$.

L'intérêt d'un algorithme sans branchement est qu'il s'implante très facilement par voie matérielle. Au niveau logiciel, il se compile mieux, puisque le compilateur n'a aucune prédiction de branchement à effectuer, et peut donc effectuer à l'avance un plus grand nombre d'instructions.

3 Méthodes de programmation

3.1 Algorithmes gloutons

De nombreux problèmes d'optimisation sont difficiles à résoudre, car une solution localement optimale ne donne pas forcément une solution (globalement) optimale. Si l'on cherche par exemple le minimum d'une fonction réelle, on peut se trouver « piégé » par un minimum local. Certains problèmes font exception : un algorithme *glouton*, c'est-à-dire faisant un choix optimal localement, peut s'avérer très efficace, voire donner la solution optimale.

Soit par exemple le problème du *rendu de monnaie* : comment rendre la monnaie pour une somme de n euros, avec un nombre minimal de pièces. Si l'on dispose de pièces de 1, 2, et 5 euros (en supposant que cette dernière existe), l'algorithme glouton consiste à utiliser le plus grand nombre possible de pièces de 5 euros, puis de 2 euros, et enfin de 1 euro :

Algorithme RenduMonnaie(n).
$a \leftarrow \lfloor n/5 \rfloor$
$b \leftarrow \lfloor (n - 5a)/2 \rfloor$
$c \leftarrow n - 5a - 2b$
Rendre a pièce(s) de 5 euros,
b pièce(s) de 2 euros,
et c pièce(s) de 1 euro.

On peut montrer que cet algorithme est optimal.

Pour le problème du rendu de monnaie, l'algorithme glouton ne donne pas toujours la solution optimale. Il suffit pour s'en convaincre de considérer des pièces de 1, 3 et 4 euros. Pour rendre la monnaie de 6 euros, l'algorithme glouton utilise trois pièces $(4 + 1 + 1)$, alors que la solution optimale n'en utilise que deux $(3 + 3)$.

3.2 Programmation dynamique

La *programmation dynamique* est une des principales techniques de programmation. L'idée sous-jacente est la suivante. Il arrive souvent qu'on calcule plusieurs fois la même valeur au cours de l'exécution d'un algorithme. Un exemple classique est le calcul du n-ième nombre de Fibonacci, défini par la récurrence $f_n = f_{n-1} + f_{n-2}$, avec les valeurs initiales $f_0 = 0, f_1 = 1$. L'algorithme « naïf » est le suivant :

Algorithme Fib0(n).
si $n \leqslant 1$ **alors** n
sinon Fib0$(n-1)$ + Fib0$(n-2)$.

Le nombre F_n d'appels à Fib0 pour calculer f_n vérifie la récurrence $F_n = F_{n-1} + F_{n-2}$, avec les valeurs initiales $F_0 = F_1 = 1$, ce qui donne $F_n = f_{n+1}$. Sachant que f_n croît exponentiellement avec n, un tel programme peut calculer à la rigueur $f_{30} = 832040$, mais certainement pas $f_{100} = 354224848179261915075$. On constate que Fib0$(n)$ calcule deux fois f_{n-2} : une fois directement via l'appel récursif Fib0$(n-2)$, et une seconde fois lors de l'appel Fib0$(n-1)$.

Une fois identifié le fait que la même valeur est utilisée plusieurs fois au cours d'un algorithme, il existe plusieurs façons de mettre en œuvre la programmation dynamique.

La plus facile est d'utiliser une *mémo-fonction*, si le langage de programmation utilisé le permet. Par exemple avec le langage de calcul formel MAPLE, on peut écrire :

```
f := proc(n) option remember;
  if n < 2
    then n
    else f(n-1) + f(n-2)
  fi
end:
```

La *méthode par recensement* consiste à utiliser un tableau auxiliaire contenant les valeurs déjà calculées, en utilisant une valeur spéciale — ici négative — pour marquer les entrées non initialisées :

Algorithme Fib1(n).
Init : $T_0 = 0$, $T_1 = 1$, $T_n = -1$ sinon.
si $T_n = -1$ **alors**
  $T_n \leftarrow \text{Fib1}(n-1) + \text{Fib1}(n-2)$
Renvoyer T_n.

Dans certains cas, on peut aussi transformer l'algorithme pour ne calculer qu'une seule fois chaque valeur, la stocker directement dans une table, et y faire référence par la suite :

Algorithme Fib2(n).
$T_0 \leftarrow 0$; $T_1 \leftarrow 1$
pour $i \leftarrow 2$ **à** n **faire**
  $T_i \leftarrow T_{i-1} + T_{i-2}$
Renvoyer T_n.

Une telle transformation n'est pas toujours facile ni même possible. On préférera alors la méthode par recensement, qui a l'avantage de garantir qu'on ne calcule pas plus que les valeurs effectivement nécessaires.

Il n'est pas toujours aisé d'identifier un problème se prêtant à la programmation dynamique ; cependant il est primordial de le faire, car cela peut faire passer d'une complexité exponentielle à une complexité polynomiale, comme le montre l'exemple du calcul du n-ième nombre de Fibonacci.

3.3 Diviser pour régner

Le principe « diviser pour régner » est une autre technique algorithmique. Le principe est simple : pour résoudre un gros problème, se ramener à plusieurs problèmes similaires de plus petite taille, et à partir de leurs solutions, reconstruire celle du gros problème.

Dans le domaine du tri, le *tri fusion* illustre parfaitement ce principe :

Algorithme TriFusion.
Entrée : liste $[a_1, \ldots, a_n]$.
Sortie : liste triée.
si $n \leqslant 1$ **alors** renvoyer $[a_1, \ldots, a_n]$
$m \leftarrow \lceil n/2 \rceil$
$b \leftarrow \text{TriFusion}([a_1, \ldots, a_m])$
$c \leftarrow \text{TriFusion}([a_{m+1}, \ldots, a_n])$
Renvoyer Fusion(b, c).

Pour simplifier l'exposition, on a omis le détail de la routine Fusion(b, c), qui fusionne deux listes triées, en sélectionnant au fur et à mesure le plus petit de leurs minima respectifs.

Recherche par dichotomie

Une incarnation importante du principe « diviser pour régner » est la *recherche par dichotomie*. Cette méthode s'applique lorsque l'on cherche une valeur dans une liste triée, et utilise le fait trivial que la valeur cherchée se trouve soit dans la première moitié, soit dans la seconde :

RechercheDichotomique.
Entrée : $[a_1, \ldots, a_n]$ triée, $b \leqslant a_n$.
Sortie : indice j si $a_j = b$, 0 sinon.
$i \leftarrow 0$; $j \leftarrow n$
tant que $i + 1 < j$ **faire**
  $k \leftarrow \lceil \frac{i+j}{2} \rceil$
  si $a_k < b$ **alors** $i \leftarrow k$
  sinon $j \leftarrow k$
Renvoyer (**si** $a_j = b$ **alors** j **sinon** 0).

Il faut faire attention dans une telle recherche que l'instruction **tant que** ne boucle pas indéfiniment, et que l'indice de la valeur testée — ici a_k — soit valide. Pour cela, on utilise un *invariant* de boucle : ici, c'est $a_i < b \leqslant a_j$, avec $0 \leqslant i < j \leqslant n$, et par convention a_0 plus petit que b (noter cependant que a_0 n'est jamais utilisé dans l'algorithme). Lorsque la boucle s'arrête, on a $i + 1 = j$, et par conséquent $a_{j-1} < b \leqslant a_j$, d'où la correction de l'algorithme.

Calcul de rang

Une variante de la recherche par dichotomie est le calcul de rang. Soit s une bijection croissante de $\mathbb{N}^*$ dans un ensemble ordonné S. Pour fixer les idées, soit la fonc-

tion qui à un entier i associe le i-ème nombre premier ; ainsi $s(10^6) = 15485863$. Le problème du calcul de rang est le suivant : étant donné $b \in S$, trouver l'unique entier i tel que $b = s(i)$. Sur notre exemple, il s'agit par exemple de trouver l'entier i tel que 1000003 est le i-ème nombre premier. Ce problème peut se résoudre de la manière suivante. Dans un premier temps on détermine une borne supérieure pour i, par exemple en calculant $s(2^k)$ pour $k = 0, 1, 2, \ldots$, et en comparant l'objet obtenu à b ; on obtient ainsi un encadrement $2^k < i \leqslant 2^{k+1}$. Il suffit ensuite de remplacer a_k par $s(k)$ dans le programme « Recherche-Dichotomique » ci-dessus pour trouver le rang cherché.

3.4 Récursivité

Un programme est dit *récursif* lorsqu'il s'appelle lui-même, comme par exemple le calcul du n-ième nombre de Fibonacci (§3.2). On dit aussi que deux programmes f et g sont *mutuellement récursifs* lorsque f appelle g, et g appelle f.

Une sous-classe importante à (re)connaître est celle des programmes *récursifs terminaux*. Un tel programme est de la forme suivante,

Programme p(arguments).
...
[corps du programme]
...
p(paramètres).

où le corps du programme ne contient aucun appel à p. Par exemple, l'algorithme ci-dessous est récursif terminal :

Algorithme pgcd (récursif).
Entrée : a, b entiers
Sortie : pgcd(a, b)
si $b = 0$ **alors** renvoyer a
Renvoyer pgcd$(b, a \bmod b)$.

Un programme récursif terminal se compile très facilement, puisqu'il suffit de se replacer au début du corps du programme, en ayant simplement remplacé les arguments de p par les paramètres de l'appel terminal. On peut donc facilement transformer un tel programme en un programme itératif, c'est-à-dire sans appel récursif :

Algorithme pgcd (itératif).
Entrée : a, b entiers
Sortie : pgcd(a, b)
tant que $b \neq 0$ **faire**
$(a, b) \leftarrow (b, a \bmod b)$
Renvoyer a.

Néanmoins, la plupart des compilateurs actuels reconnaissent très bien les programmes récursifs terminaux, et une telle transformation n'apporte plus de gain sensible, et au contraire peut même ralentir le programme !

3.5 Parcours d'arbre

Les algorithmes de parcours d'arbre constituent une sous-classe importante. Ils sont par exemple utilisés pour rechercher une occurrence d'un objet dans un arbre, pour lire ou afficher une structure de DAG (*directed acyclic graph* en anglais, soit graphe orienté sans cycle). Il existe de multiples façons de parcourir un arbre. On distingue usuellement le *parcours en profondeur* et le *parcours en largeur*. Dans le parcours en profondeur, on visite la première branche rencontrée jusqu'aux feuilles, avant de passer à la seconde branche. Au contraire, dans le parcours en largeur, on visite d'abord tous les nœuds à profondeur 1, puis ceux à profondeur 2, et ainsi de suite. Les deux types de parcours s'expriment de façon unifiée en utilisant une liste des nœuds à parcourir, que l'on emplit par le début pour le parcours en profondeur, et par la fin pour le parcours en largeur :

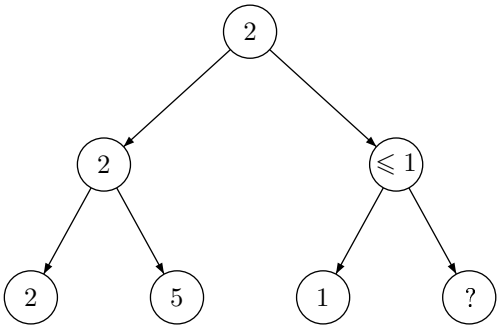
Parcours en profondeur.
Entrée : arbre A.
$l \leftarrow [A]$
tant que $l \neq []$ **faire**
  $n \leftarrow$ début(l)
  visiter n
  $l \leftarrow$ fils(n) :: reste(l)

Parcours en largeur.
Entrée : arbre A.
$l \leftarrow [A]$
tant que $l \neq []$ **faire**
  $n \leftarrow$ début(l)
  visiter n
  $l \leftarrow$ reste(l) :: fils(n)

Ces deux manières de remplir la liste des nœuds à traiter correspondent respectivement aux structures de pile et de file (voir le chapitre ℒℒ« Structures de données »).

Arbre minimax et recherche alpha-bêta

Il arrive souvent qu'on cherche à optimiser une valeur sur un arbre. Considérons par exemple le jeu Othello (ou Réversi). C'est un jeu à deux joueurs, qui se pratique sur un carré de $64 = 8 \times 8$ cases, avec des pions blancs d'un côté et noirs de l'autre. Au départ, les quatre cases centrales sont occupées avec deux pions côté blanc en diagonale, et deux côté noir. À tour de rôle, chaque joueur pose un pion de sa couleur, et retourne tous les pions de la couleur adverse qui sont entourés — en ligne, colonne ou diagonale — par le pion posé et un autre pion de la même couleur. Si on prend comme fonction d'évaluation d'une position le nombre de pions blancs, la stratégie du joueur « blanc » consiste à chercher dans un arbre la branche lui garantissant le gain maximal, sachant qu'au niveau inférieur l'adversaire agit de même, c'est-à-dire cherche à minimiser le gain. Un algorithme déterminant une telle stratégie optimale est dit de type « minimax », comme l'arbre ci-dessous. Suivant la parité du niveau, un nœud reçoit comme valeur le maximum ou le minimum des valeurs de ses fils :



Un arbre minimax.

Une technique importante d'optimisation pour la recherche dans un arbre minimax est connue sous le nom « alpha-bêta ». Illustrons-là sur l'arbre ci-dessus. L'adversaire détermine le minimum des valeurs des deux feuilles 2 et 5 du sous-arbre gauche, soit 2. La valeur du sous-arbre gauche est donc 2. Lors d'un parcours en profondeur d'abord du sous-arbre droit, on obtient la valeur 1, qui permet de déduire que la valeur du sous-arbre droit sera *au plus* 1, quelle que soit la valeur de la feuille notée '?'. En effet, l'adversaire cherche à *minimiser* la valeur du sous-arbre droit. Il en résulte qu'on peut ignorer la branche complètement à droite, puisque le gain maximum sera forcément obtenu via le sous-arbre gauche. Cela permet de diminuer fortement le nombre de nœuds parcourus, et donc à temps constant d'augmenter la profondeur de recherche.

3.6 Parcours de graphe connexe

Avec le parcours de liste et d'arbre, le parcours de graphe est une des figures imposées de l'algorithmique. On considère ici le parcours d'un graphe non orienté connexe, en partant d'un nœud quelconque, le graphe étant donné par l'ensemble de voisins de chaque nœud. La principale difficulté par rapport au parcours d'arbre est qu'il faut éviter de « boucler », c'est-à-dire de passer plusieurs fois au même endroit. Il existe plusieurs techniques pour pallier cette difficulté :

- le marquage des nœuds visités. Chaque fois qu'on parcourt un nœud, on le marque. Cela permet de distinguer les « nouveaux » nœuds de ceux qui ont déjà été traités ;
- avec un langage de programmation fonctionnel, il est parfois plus commode de calculer l'ensemble des nœuds parcourus. Strictement équivalente au marquage sur le papier, cette méthode est cependant moins efficace en pratique, car l'ajout d'un nouveau nœud coûte $O(n)$ au lieu de $O(1)$.

Parcours de graphe.
Entrée : nœud n.
$l \leftarrow [n]$
tant que $l \neq []$ **faire**
  $(n, l) \leftarrow (\text{début}(l), \text{reste}(l))$
  si n non marqué **alors**
    marquer n
    visiter n
    $l \leftarrow l :: \text{voisins}(n)$

3.7 Algorithmes en place

Un algorithme opère *en place* lorsqu'il n'utilise pas de mémoire auxiliaire. Soit l'espace mémoire nécessaire pour stocker le résultat a déjà été alloué, ou bien le résultat remplace directement les données. Voici par exemple un programme de tri par insertion en place :

Tri en place.
Entrée : une liste $[a_1, \ldots, a_n]$.
Sortie : liste triée.

pour $i \leftarrow 2$ **à** n **faire**
 pour $j \leftarrow i-1$ **à** 1
 tant que $a_j > a_{j+1}$ **faire**
 Échange(a_j, a_{j+1})

Les algorithmes en place sont souvent plus efficaces, notamment les algorithmes récursifs (§3.4), car on évite ainsi d'allouer un espace mémoire polynomial voire exponentiel en la taille des données. Cependant, si l'on dispose d'un ramasse-miettes efficace, comme c'est le cas dans les langages de programmation modernes, le coût de la gestion de la mémoire peut s'avérer relativement faible.

3.8 Sentinelles

Une *sentinelle* est une valeur spéciale, qui permet de garantir le bon déroulement d'un algorithme. On peut en donner plusieurs exemples. Lors du tri d'une liste $a_1, \ldots, a_n$ par insertion, on pourra mettre en a_0 une valeur dont on sait *a priori* qu'elle est inférieure à tous les a_i pour $1 \leqslant i \leqslant n$. Cela permet d'éviter un test dans la boucle interne, et contribue à une meilleure lisibilité du programme :

Tri par insertion avec sentinelle.
Entrée : une liste $[a_1, \ldots, a_n]$.
Sortie : la même liste triée.
$a_0 \leftarrow -\infty$
pour $i \leftarrow 2$ **à** n **faire**
 $j \leftarrow i$;
 tant que $a_j > a_{j+1}$ **faire**
 Échange(a_j, a_{j+1}) ; $j \leftarrow j-1$

Sans sentinelle, la boucle interne deviendrait :

tant que $j \geqslant 1$ et $a_j > a_{j+1}$ **faire**
 Échange(a_j, a_{j+1}) ; $j \leftarrow j-1$

Dans l'exemple ci-dessus, la sentinelle est une valeur auxiliaire, mais on peut aussi utiliser une des valeurs triées comme sentinelle. C'est le cas par exemple du tri *quicksort* :

Tri Quicksort.
Entrée : une liste $l = [a_1, \ldots, a_n]$.
Sortie : la même liste triée.
si $n \leqslant 1$ **alors** renvoyer l
$i \leftarrow 1$; $j \leftarrow n+1$; $a_{n+1} \leftarrow a_1 + 1$;
tant que $i + 1 < j$ **faire**
 répéter $i \leftarrow i+1$ **jusqu'à** $a_i > a_1$
 répéter $j \leftarrow j-1$ **jusqu'à** $a_j \leqslant a_1$
 Échange(a_i, a_j)
Quicksort$([a_1, \ldots, a_i])$
Quicksort$([a_{i+1}, \ldots, a_n])$

Ici, on a utilisé les éléments de la liste eux-mêmes comme sentinelles. En effet, la boucle **répéter-jusqu'à** sur j s'arrête forcément à $j = 1$, où la condition $a_j \leqslant a_1$ est réalisée.

4 Conclusion

Nous avons présenté dans ce chapitre quelques techniques algorithmiques (algorithmes déterministes ou probabilites, algorithmes randomisés, algorithmes par contrat, algorithmes sans branchement) et méthodes de programmation (algorithmes gloutons, programmation dynamique, méthode « diviser pour régner », récursivité, parcours d'arbre, parcours de graphe, algorithmes en place, utilisation de sentinelles). Il n'est bien sûr pas possible d'être exhaustif en une dizaine de pages. Pour aller plus loin, le lecteur est invité à consulter les ouvrages mentionnés ci-dessous.

Bibliographie

Le livre (Cormen, Leiserson, Rivest et Stein, 2001) est une bonne introduction à l'algorithmique, assortie de nombreux exemples détaillés et problèmes ; il en existe une traduction française. L'ouvrage (Aho, Hopcroft et Ullman, 1974) est plus ancien, mais reste une bonne référence, surtout pour les questions de complexité. Il en est de même pour *The Art of Computer Programming*, dont trois volumes sont parus (Knuth, 1997a,b, 1998), et d'autres sont en préparation (voir `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`). On pourra consulter aussi (Sedgewick, 1988).

Aho A. V., Hopcroft J. E. et Ullman J. D. (1974), *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

Cormen T. H., Leiserson C. E., Rivest R. L. et Stein Clifford. (2001), *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2e édition, 2001.

Knuth D. E. (1997), *The Art of Computer Programming*, volume 1 : Fundamental Algorithms. Addison-Wesley, 3e édition, 1997a.

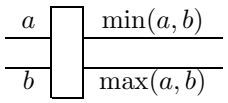
Knuth D. E. (1997), *The Art of Computer Programming*, volume 2 : Seminumerical Algorithms. Addison-Wesley, 3e édition, 1997b.

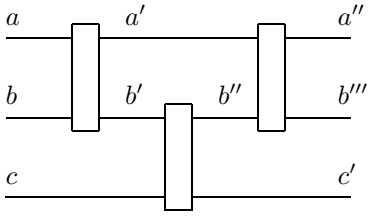
Knuth D. E. (1998), *The Art of Computer Programming*, volume 3 : Sorting and Searching. Addison-Wesley, 2e édition, 1998.

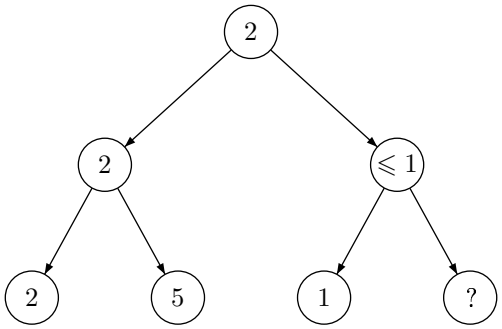
Sedgewick R. (1988), *Algorithms*. Addison-Wesley, Reading, Mass., 2e édition, 1988.

Index

Figures




a
a′
a″
b
b′
b″
b‴
c
c′


2
2
≤ 1
2
5
1
?