Quelques Algorithmes simples

Irène Guessarian *

ig@liafa.jussieu.fr

10 janvier 2012

*Je remercie Patrick Cegielski de son aide efficace pour la programmation Java ; la section sur le codage de Huffman a été écrite en collaboration avec Jean-marc Vincent.

Table des matières

1 Introduction

L'algorithmique est l'art de décrire de façon très précise les actions à mener pour effectuer une tâche donnée. La description doit être suffisamment détaillée pour pouvoir être effectuée par un programme sur ordinateur.

L' algorithmique est la discipline la plus étudiée de l'informatique,et ses origines remontent, bien avant la naissance de l'informatique à la Perse du 17eme siècle avant Jésus Christ (avec les algorithmes babyloniens pour résoudre par exemple des équations), et plus "récemment" à Al-Khwârizmî(lui aussi a vécu en Perse, à Bagdad, mais vers 900 après Jésus Christ) qui a donné son nom à l'algorithmique.

2 Algorithmes de tri

Parmi les nombreux problèmes qui peuvent se poser à la sagacité humaine, certains sont des *problèmes algorithmiques* : il s'agit de décrire très précisément une suite d'actions à effectuer pour obtenir la solution au problème.

Considérons un premier problème algorithmique, celui du *tri* (*sort* en anglais) : il s'agit, étant donnée une suite de n nombres, de les ranger par ordre croissant. La suite de n nombres est représentée par un tableau de n nombres et on suppose que ce tableau est entièrement en machine.

Si nous considérons un petit tableau, tel que (7, 1, 15, 8, 2), il est évidemment facile de l'ordonner pour obtenir (1, 2, 7, 8, 15). Mais s'il s'agit d'un tableau de plusieurs centaines, voire millions d'éléments, c'est nettement moins évident. Nous avons besoin d'une stratégie et, à partir d'une certaine taille, d'un outil pour réaliser cette stratégie à notre place car nous ne pouvons plus le faire à la main. Une stratégie pour un problème algorithmique s'appelle un *algorithme*. L'outil utilisé pour réaliser l'algorithme est de nos jours un *ordinateur*.

2.1 Deux algorithmes de tri naïfs

Le premier algorithme auquel on pense généralement pour trier un tableau s'appelle le *tri par sélection* (*selection sort* en anglais) : on recherche le plus petit élément parmi les n éléments et on l'échange avec le premier élément ; puis on recherche le plus petit élément parmi les $n-1$ derniers éléments de ce nouveau tableau et on l'échange avec le deuxième élément ; plus généralement, à la k-ième étape, on recherche le plus petit élément parmi les $n-k+1$ derniers éléments du tableau en cours et on l'échange avec le k-ième élément.

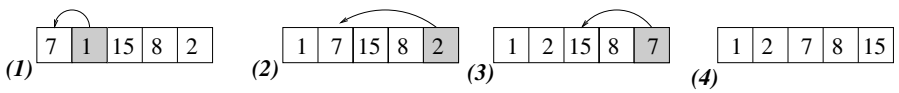


FIGURE 1 – Comment **TRI-SÉLECTION** trie le tableau 7, 1, 15, 8, 2.

On peut formaliser cet algorithme, voir la figure 1 ci-dessous.

Pour montrer en quoi cet algorithme naïf n'est pas suffisant pour les applications courantes, en particulier en gestion, intéressons-nous à la « complexité » de notre algorithme. On peut évaluer la complexité d'un algorithme de plusieurs manières :

1. complexité en espace, par exemple nombre de variables, taille des valeurs, place mémoire, ...

Algorithme 1: Tri par sélection

```
TRI-SÉLECTION(tableau T, entier N)
  Données : Un tableau T de N éléments comparables
  Résultat : Le tableau T contient les mêmes éléments mais rangés par ordre croissant
  Indice entier i, j, min
1 for i = 1 to N − 1 do
2 |   min = i        // On initialise l'indice du minimum des N − i + 1 derniers
  |   éléments
3 |   for j = i + 1 to N do
4 |   |   if (T[j] < T[min]) then
  |   |   |   min = j                  // On actualise l'indice du minimum en cours
5 |   échange (T, i, min)      // min est l'indice du plus petit élément comp ris
  |   entre les indices i et N (inclus) et on permute les éléments d'incices i
  |   et min
  |   /* Le tableau T contient les i plus petits éléments du tableau initial
  |      rangés aux places 1...i                                              */
```

2. complexité en temps : le temps d'exécution $t(n)$ du programme dépendra en général de la taille n des données d'entrée ; on distinguera :
   - la complexité moyenne, c'est-à-dire la valeur moyenne des $t(n)$: elle est en général difficile à évaluer, car il faut commencer par décider quelle donnée est « moyenne » ce qui demande un recours aux probabilités ;
   - la complexité pour le pire des cas, c'est-à-dire pour la donnée d'entrée donnant le calcul le plus long, soit $t(n)$ maximal ;
   - la complexité pour le meilleur des cas, c'est-à-dire pour la donnée d'entrée correspondant au calcul le plus court, soit $t(n)$ minimal.

L'algorithme TRI-SÉLECTION trie les éléments du tableau dans le tableau lui-même et n'a besoin d'aucun espace supplémentaire. Il est donc très économe en espace car il fonctionne en *espace constant* : on trie le tableau sur place, sans avoir besoin de stocker une partie du tableau dans un tableau auxiliaire.

Par contre il n'est pas économe en temps : en effet les lignes 1 et 3 sont deux boucles **for** imbriquées,

- la boucle de la ligne 1 doit être exécutée $N-1$ fois, et elle comprend l'instruction 2 et la boucle **for** de la ligne 3
- la boucle de la ligne 3 effectue $N-1$ comparaisons (ligne 4) à sa première exécution, puis $N-2$, $N-3$, etc. Le nombre de comparaisons effectuées par l'agorithme est donc de $(N-1)+(N-2)+\cdots+1 = N(N-1)/2$, qui est de l'ordre de N^2. Nous verrons plus loin que l'on peut trier un tableau en faisant moins de comparaisons.

Exercice 1. *Justifiez le temps de 1000 jours pour le TRI-SÉLECTION d'un tableau de 100 millions d'éléments.*

Exercice 2. *Ecrire le programme Java qui effectue l'algorithme 1. On supposera d'abord que le tableau a 5 éléments qu'on donne au clavier.*

On modifiera ensuite le programme pour entrer d'abord la taille du tableau à trier, puis les éléments du tableau.

Ce premier exemple d'algorithme nous a permis de voir comment formaliser un algorithme.

On peut penser avoir ainsi résolu le problème algorithmique posé. Prenons cependant un cas concret dans lequel on a besoin de trier de grands tableaux. Il existe en France de l'ordre de cent millions d'opérations bancaires par jour (ce qui ne fait jamais que 1,5 par habitant mais les entreprises en effectuent beaucoup plus que les particuliers). Un organisme national, la Banque de France, a besoin d'effectuer les débits/crédits chaque jour entre les différentes banques. Cet organisme commence par trier chaque opération bancaire par numéro de compte. Avec le tri par sélection cela prendrait environ 1000 jours (voir exercice 1). Il faut donc des algorithmes plus efficaces. Par exemple, le Tri Fusion que nous présenterons ci-dessous, permet de traiter le même nombre de transactions en quelques dizaines de secondes.

Un autre algorithme naïf de tri, celui qu'on fait lorsque par exemple on a en main des cartes à jouer que l'on veut ranger dans un certain ordre, est le tri par insertion.

Le tri par insertion réordonne les nombres du tableau, en commençant par le premier, et de telle sorte que, lorsqu'on traite le j-ème nombre, les $j-1$ premiers nombres sont ordonnés ; pour traiter le j-ème nombre, il suffit alors de l'insérer à sa place parmi les $j-1$ premiers nombres. Voir la figure 18 pour une illustration de l'algorithme. Soit T une liste de n nombres.

Exercice 3. *Donner un algorithme qui réalise le tri par insertion et calculer sa complexité.*

2.2 Tri Fusion (Merge Sort)

Le Tri Fusion utilise une stratégie différente : on divise le tableau à trier en deux parties (de tailles à peu près égales), que l'on trie, puis on interclasse les deux tableaux triés ainsi obtenus. La stratégie sous-jacente est du type *Diviser pour Régner* : on divise un problème sur une donnée de « grande taille » en sous-problèmes de même nature sur des données de plus petite taille, et on applique *récursivement* cette division jusqu'à arriver à un problème de très petite taille et facile à traiter ; ensuite on recombine les solutions des sous-problèmes pour obtenir la solution au problème initial. Pour le Tri Fusion, la très petite taille est 1, dans le cas d'un tableau de taille 1 il n'y a rien à trier.

L'algorithme 2 (Tri Fusion) fait appel (ligne 6) à l'algorithme 3 et se présente comme suit :

Algorithme 2: Tri fusion

TRI-FUSION (tableau T, entier N, l, r)
Données : Un tableau T de N entiers indicés de l à r
Résultat : Le tableau T contient les mêmes éléments mais rangés par ordre croissant

```
1 Indice entier m
2 if l < r then
3 |   m = ⌊(l + r)/2⌋                    // On calcule l'indice du milieu du tableau
4 |   TRI-FUSION(T, l, m)
5 |   TRI-FUSION(T, m + 1, r)
6 |   fusion(T, l, r, m)
```

L'algorithme TRI-FUSION présente plusieurs caractères intéressants :

- c'est un algorithme *récursif* : cela signifie que pour exécuter TRI-FUSION(T, l, r), il faut appeler la même procédure (avec d'autres paramètres) TRIFUSION(T, l, m) et TRI-FUSION$(T, m+1, r)$. Nous verrons plus loin d'autres algorithmes récursifs (*TriRapide* par exemple) ; l'avantage des algorithmes récursifs est en général la grande facilité avec laquelle on peut les écrire, car, souvent, l'algorithme récursif correspond de très près à l'énoncé du problème à résoudre. La difficulté est le choix des parmètres des appels récursifs qui doit à la fois

Algorithme 3: Fusion de deux tableaux

```
fusion (tableau T, entier l, r, m)
  Données : Un tableau T d'entiers indicés de l à r, et tel que l ≤ m < r et que les
            sous-tableaux T[l...m] et T[m+1...r] soient ordonnés
  Résultat : Le tableau T contenant les mêmes éléments mais rangés par ordre croissant
1 Indice entier i, j, k, n1, n2
2 Var tableau d'entiers L, R
3 n1 = m − l + 1
4 n2 = r − m
  /* On utilise des tableaux temporaires L, R et une allocation par blocs
     L = T[l...m] et R = T[m+1...r]                                          */
5 for i = 1 to n1 do
6 |  L[i] = T[l + i − 1]
7 for j = 1 to n2 do
8 |  R[j] = T[m + j]
9 i = 1
10 j = 1
11 L[n1 + 1] = ∞                                  // On marque la fin du tableau gauche
12 R[n2 + 1] = ∞                                   // On marque la fin du tableau droit
13 for k = l to r do
14 |  if (L[i] ≤ R[j]) then
15 |  |  T[k] = L[i]
16 |  |  i = i + 1
17 |  else
18 |  |  T[k] = R[j]
19 |  |  j = j + 1
```

accélérer le temps de calcul et ne pas introduire une non-terminaison de l'algorithme (voir exercice 4).
– c'est un algorithme employant la stratégie *Diviser pour Régner (Divide and Conquer)*, c'est-à-dire :

1. on *divise* le problème en sous-problèmes identiques mais sur des données de plus petite taille (ici on divise le tableau en deux)
2. on *résout* les sous-problèmes récursivement ; si leur taille est suffisamment petite, la solution est immédiate (ici, ordonner un tableau à un élément)
3. on *combine* les solutions des sous-problèmes pour obtenir la solution du problème initial (ici, la fusion de deux tableaux déjà ordonnés).

Voir la figure 2 pour une illustration de l'algorithme 2.

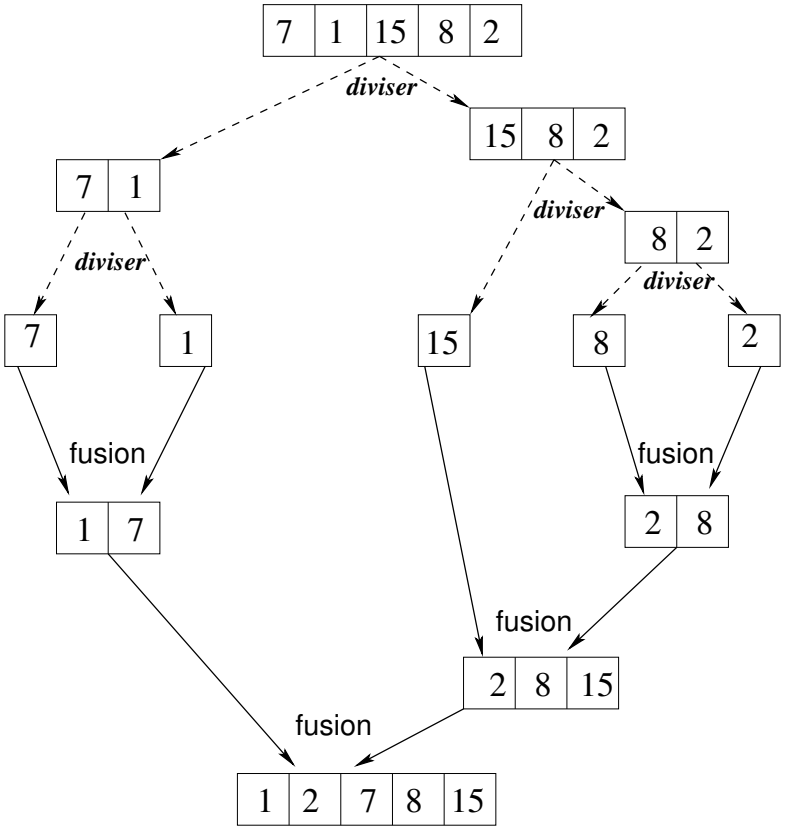


FIGURE 2 – Comment **TRI-FUSION** trie le tableau 7, 1, 15, 8, 2.

Exercice 4. *Soit* $f\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ *définie par :* $f(0,n)=n+1$,

$f(m,0)=f(m-1,1)$,
$f(m,n)=f(m-1,f(m,n-1))$.

1) Montrer que $f(m,n)$ *est définie pour tout couple* $(m,n)\in\mathbb{N}\times\mathbb{N}$.
2) Soit $g\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ *définie par :* $g(0,n)=n+1$,

$g(m, 0) = g(m - 1, 1)$,
$g(m, n) = g(m - 1, g(m, n + 1))$.

Pour quels couples $(m, n) \in \mathbb{N} \times \mathbb{N}$ *la fonction* $g(m, n)$ *est-elle définie ?*

La procédure **fusion** prend un tableau de nombres indicés de l à r, qui est divisé en deux parties triées (de l à m, et de $m + 1$ à r). Pour faire cette fusion, on copie dans deux tableaux annexes L et R les deux parties à trier ; puis on interclasse L et R : on les parcourt et on choisit à chaque étape le plus petit des deux nombres pour le mettre dans T. Lorsque la partie L (resp. R) est vide, on recopie l'autre partie dans T. Pour simplifier l'écriture de l'algorithme 3, on a placé à la fin des tableaux L, R une *sentinelle* ∞, aux lignes 11 et 12, plus grande que tous les nombres présents, et qui évite de tester si L ou R est vide.

TRI-FUSION s'exécute en *espace linéaire* : il faut recopier tout le tableau puisqu'on doit avoir une copie pour pouvoir interclasser : cela est moins bon que l'espace constant. Par contre, sa complexité en temps est excellente : soit $t(n)$ le nombre de comparaisons pour trier un tableau de taille n : la fusion des deux moitiés de tableau prend un temps $t(n)$ compris entre $n/2$ et n, et le tri des deux moitiés de tableau prendra un temps $2t(n/2)$. Donc $t(n)$ satisfait l'équation de récurrence

$$t(n) = 2t(n/2) + n$$

comme $t(1) = 0$, on en déduit $t(n) = O(n \log n)$.

Pour le problème des transactions bancaires, on passe d'un temps $O(100000000^2)$ à un temps $O(100000000 \times \log 100000000) = O(100000000 \times 8 \log 10) = O(100000000 \times 10)$, on a donc divisé le temps par un facteur 10^7, ce qui fait passer de 1000 jours (cf exercice 1) à environ 10 secondes.

Exercice 5. *Ecrire le programme Java qui effectue l'algorithme 2. On pourra d'abord écrire une version simplifiée du programme qui n'utilise pas de "sentinelle" initialisée à "*∞*".*

Exercice 6. *Ecrire le programme Java qui effectue l'algorithme 2 en utilisant une "sentinelle" initialisée à "*inf*" qui sera un entier plus grand que tous les éléments du tableau. Pour ce faire, on commence par parcourir le tableau pour trouver son maximum.*

On commencera par définir une classe "Tableau" avec trois méthodes, max, affiche, echange *qui, respectivement, trouve le maximum d'un tableau, affiche le tableau, et échange deux éléments du tableau.*

2.3 TriRapide (Quicksort)

Donnons l'idée de l'algorithme *TriRapide* : pour trier un tableau de longueur n, on choisit un élément p, appelé *pivot*, dans ce tableau et on permute 2 à 2 les éléments du tableau de sorte à regrouper en début de tableau tous les éléments $\leq p$, et en fin de tableau tous les éléments $> p$; on place le pivot à sa place (entre les éléments $\leq p$, et ceux $> p$), et on recommence l'opération avec la partie de tableau $\leq p$, et la partie de tableau $> p$ (voir la figure 3). Il s'agit ici encore d'une stratégie de type « *Diviser pour Régner* » .

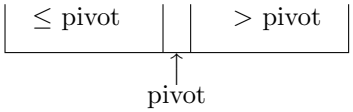


FIGURE 3 – La place du pivot

Soit $T[i \dots j]$ le tableau des éléments que l'on veut trier, l'algorithme qu'on vient de décrire s'écrit (il fait appel à une procédure `segmente` décrite dans l'algorithme 5 donné plus bas) :

Algorithme 4: TriRapide

QUICKSORT (tableau T, entier i, j)
Données : Un tableau T d'éléments indicés de i à j
Résultat : Le tableau T contenant les mêmes éléments mais rangés par ordre croissant

```
1 Indice entier j
2 if (i < j) then
3 |  k =segmente(T, i, j)                    // T(k) est à sa place finale
4 |  QUICKSORT(T, i, k − 1)          // trie les éléments de i à k − 1 (inclus)
5 |  QUICKSORT(T, k + 1, j)          // trie les éléments de k + 1 à j (inclus)
```

QUICKSORT est aussi un algorithme récursif et il emploie également une stratégie de type *Diviser pour Régner*; on pourra remarquer l'élégance et la concision de l'algorithme 4 : élégance et concision sont en général le propre des algorithmes récursifs.

segmente est un algorithme qui choisit pour pivot un élément du tableau $T[i]$ (*ici ce sera le premier élément*) et permute les éléments de $T[i \dots j]$ jusqu'à ce qu'à la fin $T[i]$ soit à sa place définitive dont l'indice est k, et tous les $T[j] \leq T[i]$ soient à gauche de $T[i]$ et tous les $T[j] > T[i]$ soient à la droite de $T[i]$. L'algorithme **segmente** utilise deux compteurs courants pour les $T[j]$, l'un croissant (l), l'autre décroissant (r) : l croît jusqu'à ce que $T[l]$ soit plus grand que le pivot, r décroît jusqu'à ce que $T[r]$ soit plus petit que le pivot et à ce moment on permute $T[l]$ et $T[r]$ et on continue ; ensuite, au moment où r et l se croisent, c'est-à-dire quand l devient supérieur ou égal à r, on a terminé et on peut placer le pivot en k qui est sa place finale, en échangeant $T[i]$ et $T[k]$ (par l'appel de procédure `échange`(T, i, k)) et $T[k]$ se retrouve à sa place définitive (figure 4).

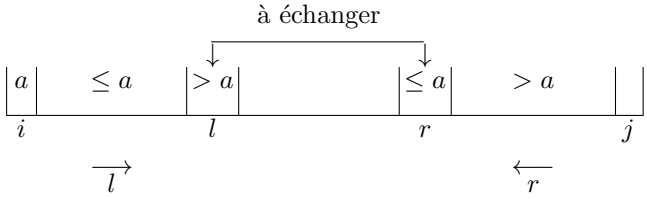


FIGURE 4 – Un échange `échange`(T, l, r)

On pourra étudier l'exemple suivant du tri du tableau : $100, 202, 22, 143, 53, 78, 177$. Les pivots sont soulignés et les sous-tableaux des éléments $\leq k$ et des éléments $> k$ qui restent à trier sont en caractères gras. La partie de tableau qui sera traité à la ligne suivante est entre crochets [...]. La première colonne donne l'appel de procédure dont le résultat est écrit sur la même ligne.

résultat de	[**100**	**202**	**22**	**143**	**53**	**78**	**177**]
segmente(T,1,7)	[**53**	**78**	**22**]	100	**143**	**202**	**177**
segmente(T,1,3)	**22**	53	**78**	100	[**143**	**202**	**177**]
segmente(T,5,7)	22	53	78	100	143	[**202**	**177**]
segmente(T,6,7)	22	53	78	100	143	[**177**]	202

Algorithme 5: segmente

segmente (tableau T, entier i, j)
Données : Un tableau T d'éléments indicés de i à j
Résultat : Le tableau T et l'indice k : T contient les mêmes éléments mais $T[i]$ est placé à sa place finale k, et tous les nombres à gauche de l'indice k sont plus petits que $T[i]$, et tous les nombres à droite de l'indice k sont plus grands que $T[i]$

```
1  Indice entier k, l, r
2  Var élément p
3  l = i + 1
4  r = j
5  p = T[i]
6  while (l ≤ r) do
7      while (T[r] > p) do
8          r = r − 1
9      while (T[l] ≤ p) do
10         l = l + 1
11     if (l < r) then
12         échange (T, l, r)
                        // On échange les valeurs de (T[l] et T[r]) voir figure 4
13         r = r − 1
14         l = l + 1
15 k = r
16 échange (T, l, k))
17 retourner k
```

QUICKSORT s'exécute en *espace constant* : on n'a pas besoin de recopier le tableau à trier, par contre il faut stocker les appels récursifs de **QUICKSORT**, et il peut y avoir $O(n)$ appels récursifs à **QUICKSORT**.

Pour la complexité en temps, comptée en nombre de comparaisons d'éléments :

– Le *pire cas* est obtenu si le tableau de départ est déjà trié et chaque appel de **segmente** ne fait que constater que le premier élément est le plus petit du tableau. Soit $p(n)$ la complexité la pire pour trier un tableau de longueur n, on a donc

$$p(n) = n - 1 + p(n-1) \tag{1}$$

avec $p(2) = 3$. On écrit les égalités 1 pour $i = 2, \ldots, n$, on somme ces égalités, et on en déduit

$$\forall n \geq 2, \qquad p(n) = \frac{n(n-1)}{2} - 3,$$

c'est-à-dire une complexité quadratique, la même que pour le tri par sélection (voir [K98]).

– La *meilleure complexité* en temps pour **TriRapide** est obtenue lorsque chaque appel récursif de **TriRapide** se fait sur un tableau de taille la moitié du tableau précédente, ce qui minimise le nombre d'appels récursifs ; soit $m(n)$ la complexité la meilleure pour trier un tableau de longueur $n = 2^p$, on a donc

$$m(n) = n + 2m(n/2)$$

avec $m(2) = 2$, d'où $m(n) = n\log(n)$.

– La *complexité moyenne* de **TriRapide** est plus difficile à calculer ; la moyenne de la complexité en temps sur toutes les permutations possibles est de l'ordre de $O(n \log n)$ [AG06], la même que pour le tri fusion. La complexité en moyenne est donc égale à la complexité dans le meilleur cas, et de plus c'est la meilleure complexité possible pour un algorithme de tri, ce qui nous permet de dire que **TriRapide** est un bon algorithme.

Le tableau ci-dessous résume les complexités des algorithmes de tri que nous avons vus.

complexité	espace	temps		
		pire	moyenne	meilleure
tri sélection	constant	n^2	n^2	n^2
tri fusion	linéaire	$n \log n$	$n \log n$	$n \log n$
tri rapide	constant	n^2	$n \log n$	$n \log n$

Il y a aussi d'autres méthodes de tri : nous verrons dans la section 4 qu'en organisant les données sous la forme d'un arbre binaire de recherche, on peut construire l'arbre au fur à mesure qu'on lit les données, et ce de telle sorte qu'un parcours judicieusement choisi dans l'arbre donne très rapidement le tableau de données trié.

3 Algorithmes de recherche

Le problème est de rechercher des informations dans une « table » (qui peut être une liste, un tableau, une liste chaînée, un arbre, etc.). Dans la suite, on supposera que :

1. les tables sont des tableaux,

2. les tableaux contiennent des nombres (on pourrait traiter de la même façon des enregistrements quelconques, par exemple (numéro de Sécurité Sociale, nom, adresse) ou (nom, numéro de téléphone)).

3.1 Recherche séquentielle

Cette méthode simple consiste à parcourir le tableau à partir du premier élément, et à s'arrêter dès que l'on trouve l'élément cherché (on ne cherche pas toutes les occurrences d'un élément). Soient T un tableau de n éléments et k l'élément qu'on recherche.

Algorithme 6: Recherche séquentielle

RECHERCHE (tableau T, élément k)
Données : Un tableau T de n éléments et un élément k
Résultat : Le premier indice i où se trouve l'élément k si k est dans T, et sinon la réponse « k n'est pas dans T »

```
1 Indice entier i
2 i = 1
3 while ((i ≤ n) ∧ (T[i] ≠ k))                    // voir section 9.3
4 do
5 |  i = i + 1
6 if (i ≤ n) then
7 |  afficher T[i] = k
8 else
9 |  afficher k n'est pas dans T
```

La complexité en temps de RECHERCHE est linéaire (de l'ordre de n), puisqu'il faudra au pire parcourir tout le tableau.

3.2 Recherche dichotomique

Cette méthode s'applique si le tableau est déjà trié et s'apparente alors à la technique « *Diviser pour Régner* » . Elle suppose donc

1. que les éléments du tableau sont comparables
2. un *prétraitement* éventuel du tableau où s'effectue la recherche : on va, par un précalcul, *trier* le tableau dans lequel on veut chercher.

Il faudra certes ajouter le temps du précalcul au temps de l'algorithme de recherche proprement dit, mais si l'on doit faire plusieurs recherches dans un même tableau, le précalcul permet de gagner un temps considérable car le temps de recherche après précalcul est de $\log n$ contre un temps de n sans précalcul.

On supposera dans la suite que les tableaux ne contiennent que des entiers naturels ; les algorithmes sont similaires quels que soient les éléments d'un ensemble totalement ordonné.

Soient T un tableau déjà trié de n nombres et k le nombre qu'on recherche. On compare le nombre k au nombre qui se trouve au milieu du tableau T. Si c'est le même, on a trouvé, sinon on recommence sur la première moitié (ou la seconde) selon que k est plus petit (ou plus grand) que le nombre du milieu du tableau.

L'algorithme 7 applique aussi la stratégie *Diviser pour Régner*, bien qu'il ne soit pas récursif.

Soit $t(n)$ le nombre d'opérations effectuées par l'algorithme 7 sur un tableau de taille n : $t(n)$ satisfait l'équation de récurrence $t(n) = t(n/2)+1$, comme $t(1) = 1$, on en déduit $t(n) = O(\log n)$. On remarquera que la complexité en temps est réduite de linéaire ($O(n)$) à logarithmique ($O(\log n)$) entre la recherche séquentielle (algorithme 6) et la recherche dichotomique (algorithme 7).

Algorithme 7: Recherche dichotomique

```
RECHERCHEDICHO (tableau T, entier k)
  Données : Un tableau T[1...N] d'entiers déjà ordonné et un entier k
  Résultat : Un indice i où se trouve l'élément k, ou bien -1(par convention si k n'est pas
             dans T)
1  Indice entier i, l, r
2  l = 1
3  r = N
4  i = ⌊(l + r)/2⌋
5  while ((k ≠ T[i]) ∧ (l ≤ r)) do
6  |  if (k < T[i]) then
7  |  |  r = i − 1
8  |  else
9  |  |  l = i + 1
10 |  i = ⌊(l + r)/2⌋
11 if (k == T[i]) then
12 |  retourner i
13 else
14 |  retourner −1
```

Remarquons que la recherche dichotomique dans un tableau déjà ordonné revient à organiser les données du tableau sous la forme d'un arbre (voir figure 6), puis à descendre dans l'arbre jusqu'à soit trouver le nombre cherché, soit arriver à un échec. Par exemple, rechercher 177 dans le tableau

$$T = 22, 53, 78, 100, 143, 177, 202$$

par l'algorithme 7 revient à suivre le chemin indiqué en gras dans la figure 5 **A** ; rechercher 180 dans T revient à suivre le chemin indiqué en gras dans la figure 5 **B**. Nous formaliserons cette utilisation des arbres dans la section suivante (section 4).

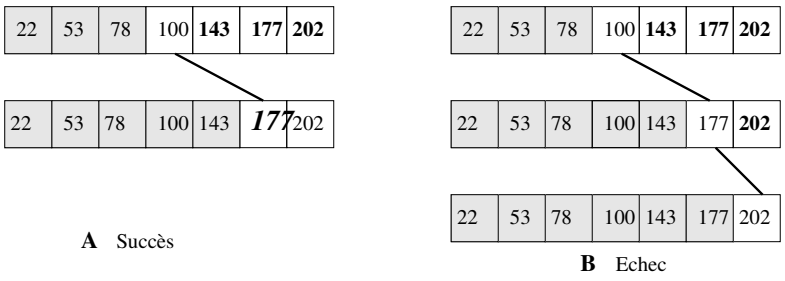


FIGURE 5 – Dans la figure **A** recherche dichotomique de 177 dans $T = 22, 53, 78, 100, 143, 177, 202$; dans la figure **B** recherche dichotomique de 180 dans T.

Exercice 7. *On suppose que la taille N du tableau T est une puissance de 2, $N = 2^n$.*

1. *Modifier l'algorithme 7 pour trouver le premier indice i où se trouve l'élément k.*
2. *Ecrire un algorithme récursif pour trouver le premier indice i où se trouve l'élément k.*

Exercice 8. *Donner les numéros des nœuds de l'arbre de la figure 8.a et leurs étiquettes.*

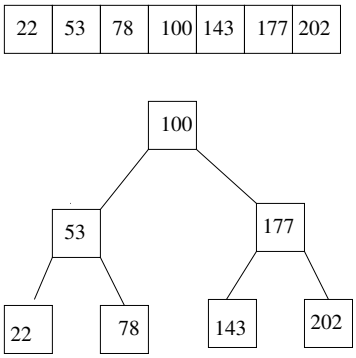


FIGURE 6 – Un arbre représentant $T = 22, 53, 78, 100, 143, 177, 202$; une recherche dichotomique dans T revient à suivre un chemin dans cet arbre.

4 Arbres binaires

4.1 Généralités et définitions

Nous montrons maintenant comment tri et recherche en table peuvent être traités en utilisant des structures d'arbres binaires.

L'ensemble *Arbre* des arbres binaires dont les nœuds sont étiquetés par les éléments d'un alphabet A est défini inductivement par :

- l'ensemble vide est un arbre binaire. Pour abréger l'écriture, l'arbre vide (parfois noté **null**) sera noté $\emptyset$
- si l et r sont des arbres binaires, et a un élément de l'alphabet A, alors le triplet (a, l, r) est un arbre binaire dont la racine est étiquetée par a et dont l (resp. r) est le sous-arbre gauche (resp. droit).

On peut aussi considérer l'ensemble AB des *arbres binaires non étiquetés*, (*i.e.* sans le champ valeur défini au chapitre 2), voir figure 7.

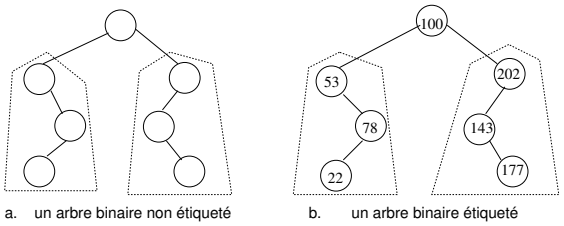


FIGURE 7 – Un arbre non étiqueté et un arbre étiqueté. Les sous-arbres gauches et droits sont entourés de traits en pointillés

Dans la suite tous nos arbres sont étiquetés et l'ensemble A des étiquettes est l'ensemble $\mathbb{N}$ des entiers naturels. On définit l'ensemble des nœuds d'un arbre binaire T, la racine de T, le fils gauche (resp. droit) d'un nœud, le sous-arbre gauche (resp. droit) d'un nœud :

- $noeuds(\emptyset) = \emptyset$
- $noeuds(a, l, r) = \{(a, l, r)\} \cup noeuds(l) \cup noeuds(r)$
- les fils droits et gauches, de même que les sous-arbres droits et gauches ne sont pas définis pour l'arbre vide $\emptyset$

- par convention, on définit $racine(\emptyset) = NIL$
- si $T = (a, l, r)$, avec $l \neq \emptyset \neq r$, alors
  1. $racine(T) = \{(a, l, r)\}$ (on identifie la racine à l'arbre et on la note $\{T\}$),
  2. $filsgauche(T)) = filsgauche(\{(a, l, r)\}) = racine(l)$,
  3. *sous-arbre-gauche*$(\{(a, l, r)\}) =$ *sous-arbre-gauche*$(T) = l$, et symétriquement
  4. $filsdroit(T) = filsdroit(\{(a, l, r)\}) = racine(r)$,
  5. *sous-arbre-droit*$(\{(a, l, r)\}) =$ *sous-arbre-droit*$(T) = r$,
- si l est l'arbre vide $\emptyset$, *i.e.* $T = (a, \emptyset, r)$, alors
  1. $racine(T) = \{(a, \emptyset, r)\}$
  2. $filsgauche(T) = filsgauche(\{(a, \emptyset, r)\}) = NIL$,
  3. *sous-arbre-gauche*$(T) =$ *sous-arbre-gauche*$(\{(a, \emptyset, r)\}) = \emptyset$
  4. $filsdroit(T) = filsdroit(\{(a, \emptyset, r)\}) = racine(r)$,
  5. *sous-arbre-droit*$(T) =$ *sous-arbre-droit*$(\{(a, \emptyset, r)\}) = r$,
- symétriquement si $r = \emptyset$;
- si $l = r = \emptyset$, alors $\{(a, \emptyset, \emptyset)\}$ n'a ni fils gauche ni fils droit (tous deux sont NIL), on dit que la racine $\{(a, \emptyset, \emptyset)\}$ est une *feuille*.

Un nœud n d'un arbre T est identifié par le sous-arbre complet de T qui a pour racine n et tous les nœuds de ce sous-arbre (ils sont « en-dessous » de n) sont appelés les *descendants* de n dans T. On peut de même définir les sous-arbres gauche et droit d'un nœud (qui sont déterminés par les fils gauche et droit de ce nœud).

Pour pouvoir désigner les nœuds d'un arbre, on les numérote par des suites de 0 et de 1 comme suit : la racine est numérotée ε (la suite vide), son fils gauche est numéroté 0 et son fils droit est numéroté 1 ; de manière générale, le fils gauche d'un nœud de numéro n est numéroté $n \cdot 0$ et son fils droit est numéroté $n \cdot 1$ (où $\cdot$ désigne la concaténation, par exemple $01 \cdot 11 = 0111$). Pour simplifier, on notera $n0$ au lieu de $n \cdot 0$, et $n1$ au lieu de $n \cdot 1$. Dans la suite, *on désignera un nœud par son numéro*. Cette numérotation nous permet de définir facilement les relations *père* et *fils* sur les nœuds d'un arbre : $pere(\varepsilon)$ n'est pas défini (la racine n'a pas de père), et $pere(n0) = pere(n1) = n$. Les fils du nœud n sont les nœuds $n0$ (fils gauche) et $n1$ (fils droit) ; par exemple le fils gauche de 01 sera 010, et le père de 01 sera 0.

On définit une application $label_T$ qui donne l'étiquette d'un nœud de T :

- si $T = \emptyset$, $label_T$ est l'application vide ($\emptyset$ n'a pas de nœud)
- si $T = (a, l, r)$, alors
  - $label_T(racine(T)) = a$, et,
  - si le nœud v est un nœud de l, $label_T(v) = label_l(v)$,
  - si le nœud v est un nœud de r, $label_T(v) = label_l(r)$.

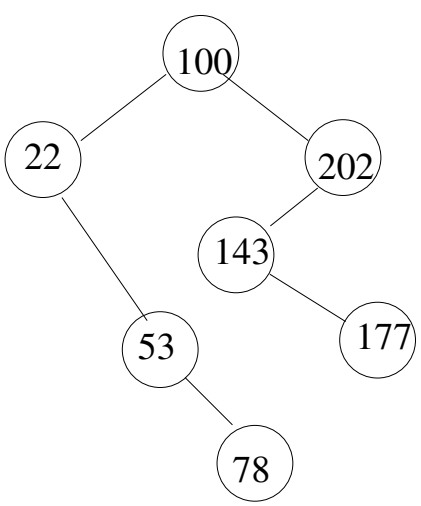
Dans la suite, on simplifiera en écrivant (par abus de notation) $T(v)$ au lieu de $label_T(v)$.

Le seul nœud sans père est la racine de T . Un nœud sans fils (c'est-à-dire dont les fils droits et gauches sont l'arbre vide $\emptyset$) est appelé une *feuille*. Par convention, on dira que l'étiquette d'un nœud de l'arbre vide est NIL.

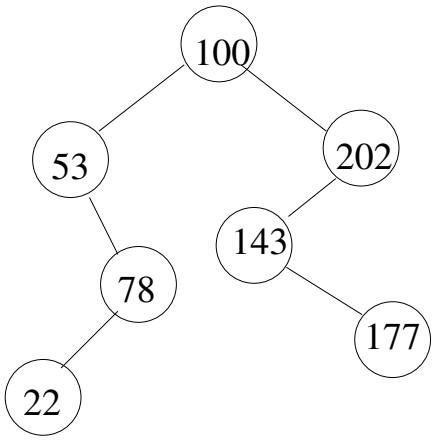
La figure 8 montre deux arbres binaires différents dont les nœuds sont étiquetés par le même ensemble d'étiquettes.

4.2 Parcours dans les arbres

Un parcours d'arbre énumère toutes les étiquettes des nœuds d'un arbre selon un ordre choisi par avance. On distingue :



a. un arbre binaire de recherche



b. un arbre binaire

FIGURE 8 – Deux arbres binaires qui diffèrent par l'ordre de leurs sous-arbres. L'arbre de la figure a est un arbre binaire de recherche (cf. section 4.3).

- le *parcours en largeur (breadth-first)* : on énumère les étiquettes des nœuds niveau par niveau, on commence par la racine, puis on énumère les nœuds qui sont à la distance 1 de la racine, puis les nœuds qui sont à la distance 2 de la racine, etc. (La *distance* entre deux nœuds est le nombre d'arêtes entre ces deux nœuds.)
- les *parcours en profondeur (depth-first)* : ces parcours descendent « en profondeur » dans l'arbre tant que c'est possible. Pour traiter un nœud n, on traite d'abord tous ses descendants et on « remonte » ensuite pour traiter le père de n et son autre fils. Ces parcours en profondeur sont de trois types, et on les définit récursivement de manière fort simple.
  1. le *parcours infixe* traite d'abord le sous-arbre gauche, puis le nœud courant puis son sous-arbre droit : l'étiquette du nœud courant est *entre* les listes d'étiquettes des sous-arbres gauches et droits.
  2. le *parcours préfixe* traite d'abord le nœud courant, puis son sous-arbre gauche, puis son sous-arbre droit : l'étiquette du nœud courant est *avant* les listes d'étiquettes des sous-arbres gauches et droits.
  3. le *parcours suffixe* (ou *postfixe*) traite d'abord le sous-arbre gauche, puis le sous-arbre droit, puis le nœud courant : l'étiquette du nœud courant est *après* les listes d'étiquettes des sous-arbres gauches et droits.

Example 1. *Pour l'arbre de la figure 8.a :*
- *le parcours en largeur donne la liste : 100, 22, 202, 53, 143, 78, 177*
- *le parcours infixe donne la liste : 22, 53, 78, 100, 143, 177, 202*
- *le parcours préfixe donne la liste : 100, 22, 53, 78, 202, 143, 177*
- *le parcours suffixe donne la liste : 78, 53, 22, 177, 143, 202, 100*

On remarquera que la liste obtenue par le parcours infixe est ordonnée (voir figure 9) : cela est dû au fait que l'arbre est un arbre binaire de recherche (cf. section 4.3).

L'algorithme 8 réalise un parcours en largeur d'arbre. T est un arbre, L la liste des étiquettes des nœuds de T quand on le parcourt, et F une file FIFO (FIFO est une abréviation pour

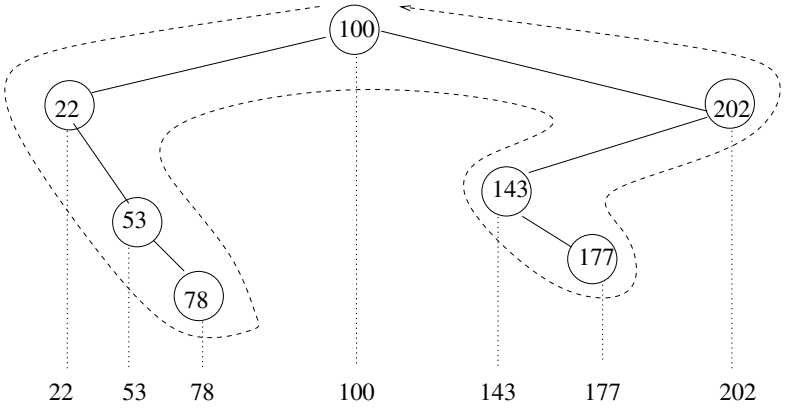


FIGURE 9 – L'arbre binaire de la figure 8.a et son parcours infixe.

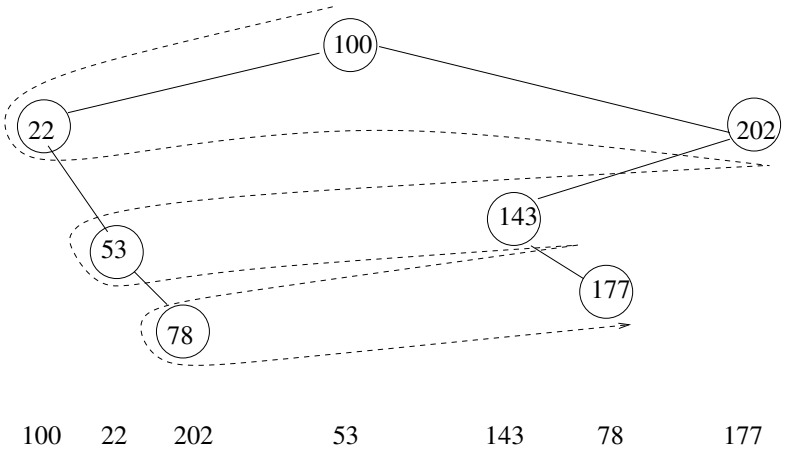


FIGURE 10 – L' arbre binaire de la figure 8.a et son parcours en largeur.

"First In First Out" signifiant que les éléments de cette file sont pris dans l'ordre où on les a mis dans la file) qui contient les nœuds en cours de traitement. Les opérations sur une file FIFO sont : $enfiler(F, n)$) qui ajoute le nœud n à la fin de la file F, et $premier(F)$ (resp. $defiler(F)$) qui prend le premier élément de la file (resp. la file amputée de son premier élément). En programmation, on note $n = defiler(F)$ pour désigner les *deux* opérations consistant à affecter le premier élément de F à n *et* à enlever ce premier élément de F.

Les algorithmes réalisant les parcours en profondeur s'écrivent de manière récursive, beaucoup plus simplement, car ils sont définis par récurrence sur la structure des arbres. Par exemple, le programme réalisant le parcours infixe s'écrit :

On remarquera l'élégance et la concision de l'algorithme récursif 9.

La complexité en temps de PARCOURS-IN sur un arbre T de taille n (c'est-à-dire ayant n nœuds) est linéaire ; en effet, soit $t(T)$ le nombre d'opérations nécessaires pour engendrer le parcours infixe de T. Pour calculer PARCOURS-IN(T), on effectue deux fois l'opération $\cdot$ ($L \cdot L'$ met la liste L' à la suite de la liste L). On suppose que $\cdot$ est une opération primitive qui prend un temps constant c, alors on peut montrer que $t(T) = O(n)$ par récurrence sur

Algorithme 8: Parcours en largeur d'un arbre T

PARCOURS-L (arbre T)
Données : Un arbre T dont les nœuds sont étiquetés par des entiers
Résultat : Le tableau L des étiquettes des nœuds de T quand on le parcourt en largeur

```
1  Var nœud n
2  Var file FIFO F
3  Var tableau L
4  i = 1
5  F = ∅
6  n = racine(T)
7  enfiler(F, n)
8  while (F ≠ ∅) do
9  |  n = defiler(F)
10 |  if (filsgauche(n) ≠ ∅) then
11 |  |  enfiler(F, filsgauche(n))
12 |  if (filsdroit(n) ≠ ∅) then
13 |  |  enfiler(F, filsdroit(n))
14 |  L[i] = T(n)
15 |  i = i + 1
16 retourner L
```

Algorithme 9: Parcours infixe d'un arbre T

PARCOURS-IN (arbre T)
Données : Un arbre T dont les nœuds sont étiquetés par des entiers
Résultat : La liste L des étiquettes des nœuds de T quand on le parcourt en infixe

```
1  Var liste d'entiers L, L1, L2
2  if (T == ∅) then
3  |  L = ∅                                  // Si T est vide, L est la liste vide
4  else
5  |  L1 =PARCOURS-IN(sous-arbre-gauche(T))
6  |  L2 =PARCOURS-IN(sous-arbre-droit(T))
7  |  L = L1 · label_T(racine(T)) · L2       // On concatène L1, la racine de T et L2
8  retourner L
```

le nombre de nœuds de T. On doit d'abord définir le nombre n de nœuds d'un arbre T par $n = nnoeuds(T)$, où *nnoeuds* est la fonction définie par : $nnoeuds(\emptyset) = 0$ et, si $T \neq \emptyset$, $n = nnoeuds(T) = 1 + nnoeuds(sous\text{-}arbre\text{-}gauche(T)) + nnoeuds(sous\text{-}arbre\text{-}droit(T))$. L'hypothèse de récurrence est : si T a n nœuds, alors $t(T) \leq 2c \times n$; on constate en regardant la définition de PARCOURS-IN(T) que $t(T) = 2c + t(sous\text{-}arbre\text{-}gauche(T)) + t(sous\text{-}arbre\text{-}droit(T))$; en utilisant l'hypothèse de récurrence, $t(T) \leq 2c + 2c \times nnoeuds(sous\text{-}arbre\text{-}gauche(T)) + 2c \times nnoeuds(sous\text{-}arbre\text{-}droit(T)) = 2c \times n$, en remarquant que :
$nnoeuds(T) = 1 + nnoeuds(sous\text{-}arbre\text{-}gauche(T)) + nnoeuds(sous\text{-}arbre\text{-}droit(T))$.

Exercice 9. *1. La preuve donnée dans la section 4.2 pour montrer que la complexité en temps du parcours infixe est linéaire est fausse : pourquoi ?*

2. Corriger cette preuve (il faut prendre en compte le cas de base de la récurrence, à savoir le cas où $n = 0$).

Exercice 10. *Modifier le programme PARCOURS-IN pour obtenir le parcours préfixe (resp. suffixe ou postfixe).*

Puisque le parcours infixe d'un arbre binaire de recherche prend un temps linéaire et fournit une liste ordonnée, on peut penser à utiliser les arbres binaires de recherche comme outil de tri. C'est une possibilité, que nous explorons maintenant, en commençant par donner un algorithme de construction d'arbre binaire de recherche.

4.3 Arbres binaires de recherche

Un arbre binaire de recherche est un arbre binaire dont les nœuds sont étiquetés par les éléments d'un ensemble A *totalement ordonné* de sorte que la propriété P suivante soit toujours vraie : si a est l'étiquette d'un nœud n, alors tous les nœuds du sous-arbre gauche de n sont étiquetés par des éléments b tels que $b \leq a$, et tous les nœuds du sous-arbre droit de n sont étiquetés par des éléments b tels que $b > a$.

L'arbre de la figure 8.a est un arbre binaire de recherche, alors que celui de la figure 8.b n'est pas un arbre binaire de recherche, puisque le nœud étiqueté 53 ne satisfait pas la propriété P car 22 apparaît dans le sous-arbre droit de 53 mais est plus petit que 53.

Les arbres binaires de recherche ont la très grande vertu de faciliter le tri d'un ensemble d'éléments : si les éléments sont stockés dans un arbre binaire de recherche, il suffit de parcourir cet arbre dans l'ordre infixe pour ranger les éléments par ordre croissant.

4.4 Construction des arbres binaires de recherche

Un arbre binaire de recherche est une *structure de données dynamique* qui doit pouvoir changer par l'ajout ou la suppression d'éléments.

Si T est l'arbre vide, on le remplace par l'arbre de racine v et de sous-arbres gauche et droit vides. Si T n'est pas l'arbre vide, on compare v à la racine de T et si v est plus petit que cette racine, on l'insère à gauche, s'il est plus grand que la racine on l'insère à droite, et s'il est égal à la racine on a terminé et on ne fait rien. Si donc on cherche à insérer un nombre qui est déjà dans l'arbre, ce dernier n'est pas modifié. L'algorithme récursif 10 est très simple.

Exercice 11. *Décrire un algorithme non récursif pour insérer un entier dans un arbre binaire de recherche. Idée : pour ajouter un nombre v dans un arbre binaire T, on descendra depuis la racine de T jusqu'à arriver à un nœud de T où on pourra ajouter une nouvelle feuille z étiquetée v à sa bonne place.*

Pour ajouter 155 dans l'arbre de la figure 8.a, on descendra le long de la branche dont les nœuds sont étiquetés 100, 202, 143, on ajoutera un nœud z de numéro 100 et d'étiquette 155. On

Algorithme 10: Insertion d'un nombre à une feuille dans un arbre T

```
INSERER (arbre T, entier v)
  Données : Un arbre binaire de recherche T dont les nœuds sont étiquetés par des entiers
            et un entier v à insérer dans T
  Résultat : L'arbre T avec une nouvelle feuille étiquetée v
1 Var nœud x
2 x = racine(T)
3 if (T == ∅) then
4 |  T = (v, ∅, ∅)
5 else
6 |  if (v < T(x)) then
7 |  |  INSERER(sous-arbre-gauche(T),v)
8 |  else
9 |  |  if (v > T(x)) then
10 |  |  |  INSERER(sous-arbre-droit(T),v)
```

voit que la complexité du programme **INSERER** est linéaire en la hauteur de l'arbre binaire de recherche. On conçoit donc qu'un arbre binaire le moins haut possible optimisera le temps.

Exercice 12. *1. Quel type de liste donne la complexité la pire pour **INSERER** ?*
2. Quel est l'ordre de grandeur de cette complexité la pire ?

On peut montrer (cf. [K98] page 431), que pour générer à partir de $\emptyset$ un arbre binaire de recherche contenant tous les nombres d'une liste de n nombres, il faut *en moyenne* un temps $O(n \log n)$, et que la construction de cet arbre effectue des comparaisons similaires à *TriRapide*. En reprenant maintenant l'idée de trier une liste en l'implantant comme un arbre binaire de recherche, que l'on parcourt ensuite de manière infixe, on voit que le temps de tri est le temps de parcours, $O(n)$, auquel il faut ajouter le temps de création de l'arbre binaire qui représente la liste donnée, soit $O(n \log n)$, et donc en tout $O(n \log n) + O(n) = O(n \log n)$; ce temps est comparable à celui de *TriRapide*. [K98] en conclut que, si l'on s'intéresse à trier et aussi à faire d'autres opérations sur une liste de nombres, il est judicieux d'implanter cette liste sous la forme d'un arbre binaire de recherche.

Exercice 13. *1. Définir une classe Java "Node", avec des méthodes qui permettent de retrouver le label d'un nœud, de définir ses fils gauches et droits, et de les retrouver.*
2. Définir une classe Java "Arbre", avec des méthodes qui permettent de définir l'arbre vide, de définir et retrouver la racine d'un arbre, d'insérer un élément dans un arbre, de faire le parcours préfixe (infixe) d'un arbre en imprimant les valeurs de ses nœuds.

Exercice 14. *Définir un programme Java utilisant la classe "Arbre" définie dans l'exercice 13 pour construire l'arbre binaire de recherche de la figure 8.a .*

4.5 Opérations sur les arbres binaires de recherche

Les opérations usuelles sur les arbres binaires de recherche sont : rechercher un nombre, supprimer un nombre, rechercher le nombre maximum, etc. Toutes ces opérations se feront en temps $O(h)$ si h est la hauteur de l'arbre binaire de recherche.

Par exemple, l'algorithme de recherche d'un nombre k dans un arbre binaire de recherche s'écrit :

Algorithme 11: Recherche dans un arbre binaire de recherche T

```
RECHERCHER (arbre T, entier k)
  Données : Un arbre binaire de recherche T dont les nœuds sont étiquetés par des entiers
            et une valeur k à rechercher dans T
  Résultat : Un nœud de T étiqueté k s'il en existe
1 Var nœud n
2 n = racine(T)
3 while ((T(n) ≠ NIL) ∧ (T(n) ≠ k)) do
4 |   if (k < T(n)) then
5 |   |   n = filsgauche(n)
6 |   else
7 |   |   n = filsdroit(n)
8 if (T(n) ≠ NIL) then
9 |   afficher k est dans T
10 else
11 |   afficher k n'est pas dans T
```

L'algorithme de recherche du maximum d'un arbre binaire de recherche est encore plus simple (il suffit de descendre toujours à droite) :

Algorithme 12: Recherche du maximum d'un arbre binaire de recherche T

```
MAX (arbre T)
  Données : Un arbre binaire de recherche non vide T dont les nœuds sont étiquetés par
            des entiers
  Résultat : Le maximum des étiquettes des nœuds de T
1 Var nœud n
2 n = racine(T)
3 while (T(filsdroit(n)) ≠ NIL) do
4 |   n = filsdroit(n)
5 retourner T(n)
```

La plupart des opérations se font en temps $O(h)$ sur un arbre binaire de recherche de hauteur h (cf. [CLRS09]). Comme, en moyenne, la hauteur h d'un arbre binaire de recherche ayant n nœuds est en $O(\log n)$, on peut donc implanter une liste de longueur n par un arbre binaire de recherche en temps $O(n \log n)$: on fait n insertions dont chacune prend un temps $O(n \log n)$ à partir l'arbre vide. On peut en conclure que les arbres binaires de recherche donnent un bon outil de travail sur les listes.

Exercice 15. *1. Ecrire un algorithme **MIN** qui trouve le minimum d'un arbre binaire de recherche.*

2. Ecrire un algorithme qui trouve l'étiquette du successeur d'un nœud d'un ABR dans l'ordre donné par le parcours infixe de l'arbre (on suppose que tous les nœuds ont des étiquettes distinctes)
.

Exercice 16. *Ecrire un algorithme **récursif** qui cherche le maximum d'un arbre binaire de recherche (cf. algorithme 12).*

Exercice 17. *Ecrire un algorithme* ***récursif*** *qui cherche un élément dans un arbre binaire de recherche (cf. algorithme 11).*

5 Quelques algorithmes classiques sur les graphes

Les arbres que nous avons étudiés dans la section 4 sont un cas particulier des graphes auxquels nous allons intéresser maintenant.

Un *graphe (non orienté)* G est un triplet (S, A, δ), où :
- S est un ensemble de *sommets* (*vertices* en anglais),
- A est un ensemble d'*arêtes*, disjoint de S, (*edges* en anglais),
- $\delta\colon A \longrightarrow S \times S$ associe à chaque arête deux sommets non nécessairement distincts (les extrémités de l'arête).

Voir les figures 11 et 12 pour des exemples de graphes.

Un graphe non orienté est *connexe* si pour toute paire (s, s') de sommets distincts, il existe un chemin joignant s et s'. Dans un graphe non orienté connexe, la *distance* $d(s, s')$ entre deux sommets s et s' est égale à la longueur du plus court chemin joignant ces deux sommets, (en particulier $d(s, s) = 0$).

Tous nos graphes sont non orientés et connexes.

La différence essentielle entre graphes et arbres est qu'un arbre n'a aucun cycle : si on part d'un nœud on ne pourra *jamais* y revenir en suivant les arêtes de l'arbre et « sans faire marche arrière », ceci car entre deux nœuds d'un arbre il y a toujours *un et un seul* « chemin » ; par contre entre deux nœuds (qu'on appelle sommets) d'un graphe, il peut y avoir plusieurs chemins. Tout arbre est connexe, alors qu'un graphe peut ne pas l'être (et dans ce cas, entre deux sommets du graphe il peut n'y avoir aucun chemin). Toutefois, comme nous nous restreindrons aux graphes connexes, nos arbres sont des cas particuliers de graphes, et les problèmes de parcours se posent pour les graphes comme pour les arbres (il faudra un peu plus de travail pour les résoudre dans le cas des graphes du fait de la pluralité de chemins entre deux sommets).

Les graphes sont une très riche source de problèmes algorithmiques. Les premiers algorithmes sur les graphes furent motivés par des problèmes pratiques importants :
- la conception du réseau d'électrification de la Moravie conduisit à concevoir le premier algorithme glouton (cf. section 5.1) de construction d'un arbre couvrant minimal ;
- le problème de savoir comment un voyageur de commerce doit organiser un circuit pour visiter un certain nombre de villes avec un coût le plus petit possible est la source de nombreux problèmes.

Cela a conduit à généraliser aux graphes les problèmes de parcours que nous avons vus dans les arbres ; de plus, des problèmes d'optimisation (recherche de plus court chemin, de coût minimum, etc.) se posent pour les graphes. Nous illustrerons la technique dite d'algorithme glouton (*greedy algorithm*) qui est assez fréquente pour les graphes.

Les arêtes peuvent être munies d'un *poids*, qui est donné par une application $poids\colon A \longrightarrow \mathbb{N}$; par exemple, si le graphe représente le réseau SNCF, les sommets seront les villes desservies, une arête reliera deux villes entre lesquelles un train peut circuler, et son poids sera la distance entre ces deux villes (on suppose des poids dans $\mathbb{N}$ pour simplifier les notations).

5.1 Parcours dans les graphes

Comme pour les arbres, on souhaite définir des parcours dans les graphes, de telle sorte que chaque sommet soit traité une fois et une seule. Le problème est plus complexe car dans un graphe, il peut y avoir plusieurs chemins pour aller d'un sommet à un autre, et il n'y a pas un ordre naturel sur les successeurs d'un sommet. Il faut donc annoter les sommets déjà visités pour

ne pas les traiter une seconde fois : pour ce faire, S est partitionné, à tout instant de l'algorithme, en trois ensembles disjoints *blanc, gris, noir*. L'idée est que

- *blanc* est l'ensemble des sommets non traités,
- *gris* est l'ensemble des sommets visités et en cours de traitement : dès qu'un sommet est coloré en gris, on le traite, et on le met dans la file des sommets dont les sommets adjacents doivent être visités,
- *noir* est l'ensemble des sommets traités et dont tous les sommets adjacents ont été visités, marqués en gris et traités.

Au départ, tous les sommets sont blancs, sauf le sommet d'où l'on commence le parcours qui est gris ; à la fin de l'algorithme, tous les sommets sont noirs.

L'algorithme ci-dessous généralise l'algorithme de parcours en largeur **PARCOURS-L** dans les arbres au cas des graphes. On souhaite calculer un *arbre couvrant* de G (en anglais *spanning tree*), c'est-à-dire un arbre qui contienne tous les sommets de G et dont les arêtes sont aussi des arêtes de G, voir figure 11 ; l'arbre couvrant engendré ici préserve les distances au sommet s, *i.e.* le nombre d'arêtes entre un sommet n et s dans l'arbre est le nombre minimal d'arêtes pour aller de s à n dans G. Le traitement d'un sommet n consiste à trouver en quel nœud de l'arbre couvrant on doit le placer : pour cela il suffit de préciser quel est son père dans l'arbre, on calculera donc un tableau $pred[n]$ où $pred[n]$ est le sommet de G qui sera le père de n dans l'arbre couvrant (voir la section 9.4 pour la manière de programmer le calcul du tableau *pred*). Remarquons qu'un arbre couvrant peut ne pas être binaire (voir figure 11 où l'arbre A_2 n'est pas binaire).

L'ensemble des sommets en cours de traitement (ici les sommets gris) est géré par une file FIFO, comme dans **PARCOURS-L**. Pour chaque sommet s de G on dispose de la liste $ADJ(s)$ des sommets reliés à s par une arête ; on utilise aussi dans l'algorithme un tableau C donnant la couleur courante associée aux sommets, et le tableau *pred* donnant le prédécesseur de chaque sommet dans le parcours.

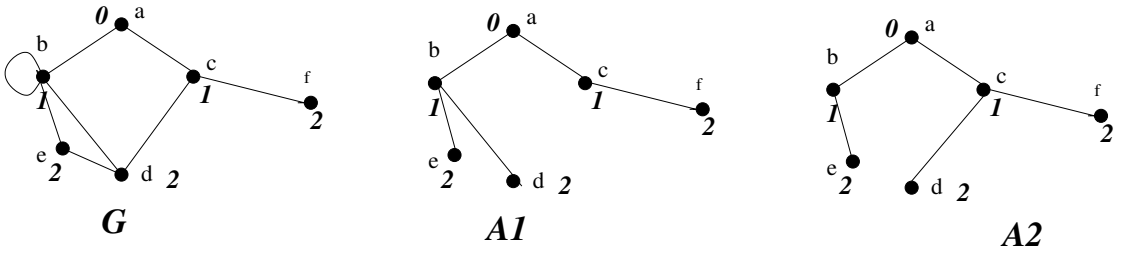


FIGURE 11 – Un graphe G et deux arbres couvrants A_1 et A_2 de G préservant les distances au sommet étiqueté a : à côté de chaque chaque sommet on voit une lettre qui est son étiquette et un nombre qui est sa distance à s.Les deux arbres A_1 et A_2 sont obtenus par l'algorithme 13.

L'algorithme 13 **PARCOURS-LG** est de type *glouton*, *i.e.* à chaque étape il fait le meilleur choix instantané possible (on n'est pas sûr que ce soit soit le meilleur dans l'absolu) ; il parcourt G en calculant les distances de chaque nœud n au nœud s choisi comme point de départ ; le parcours de G engendre en même temps un arbre couvrant préservant les distances au sommet s ; voir la figure 11 pour un exemple de graphe et deux arbres couvrants préservant les distances au sommet a obtenus par **PARCOURS-LG** : la figure 11 montre que cet algorithme est *non déterministe* : il peut faire plusieurs choix (ligne 13) et produire plusieurs résultats, par exemple les arbres couvrants $A1$ et $A2$.

Exercice 18. *Modifier **PARCOURS-LG** pour calculer la distance à s de tous les sommets de*

Algorithme 13: Parcours en largeur d'un graphe G préservant les distances à un sommet s

```
PARCOURS-LG (graphe G, sommet s)
  Données : Un graphe G et un sommet s de G
  Résultat : Un arbre couvrant de G de racine s tel que la distance à s de chaque sommet
             de G soit préservée dans l'arbre
1  Var sommet v, u
2  Var file FIFO F
3  Var tableau de couleurs C
4  Var tableau de sommets pred
5  for (v ∈ S \ {s}) do
6      C[v] = blanc
7  C[s] = gris
8  pred[s] = NIL                          // On donne la racine s à l'arbre couvrant
9  F = ∅
10 enfiler(F, s)
11 while (F ≠ ∅) do
12     u = defiler(F)
13     for ( v ∈ ADJ(u)) do
14         if (C[v] == blanc)             // On vérifie que v n'a pas déjà été visité
15         then
16             C[v] = gris pred[v] = u    // Actualise l'arbre couvrant par ajout de v
               et son père
17             enfiler(F, v)              // Actualise la file F en y ajoutant v
18     C[u] = noir                        // u et tous ses voisins ont été traités
```

G (i.e. le traitement d'un sommet n consiste à calculer la distance entre n et s).

Le problème des arbres couvrants est utile lorsqu'on veut relier par un ensemble de câbles (par exemple téléphone, télévision) un ensemble de points. Ce problème admet des variantes : si les arêtes de G sont étiquetées par des *poids* ou *coûts*, on cherchera souvent un arbre couvrant de coût minimal. Dans l'exemple de couverture d'un réseau de points par des câbles, certaines sections de câble peuvent coûter plus cher, car le câble doit être enterré plus profondément, ou doit être mieux protégé, ou est plus long etc. On souhaitera alors trouver un arbre couvrant de *coût (ou poids) minimum.*

La figure 12 montre un exemple de graphe avec poids sur les arêtes, deux arbres couvrants de poids non minimaux et un arbre couvrant de poids minimum obtenu par l'algorithme de Prim (algorithme 14) qui calcule efficacement un arbre couvrant de poids minimum.

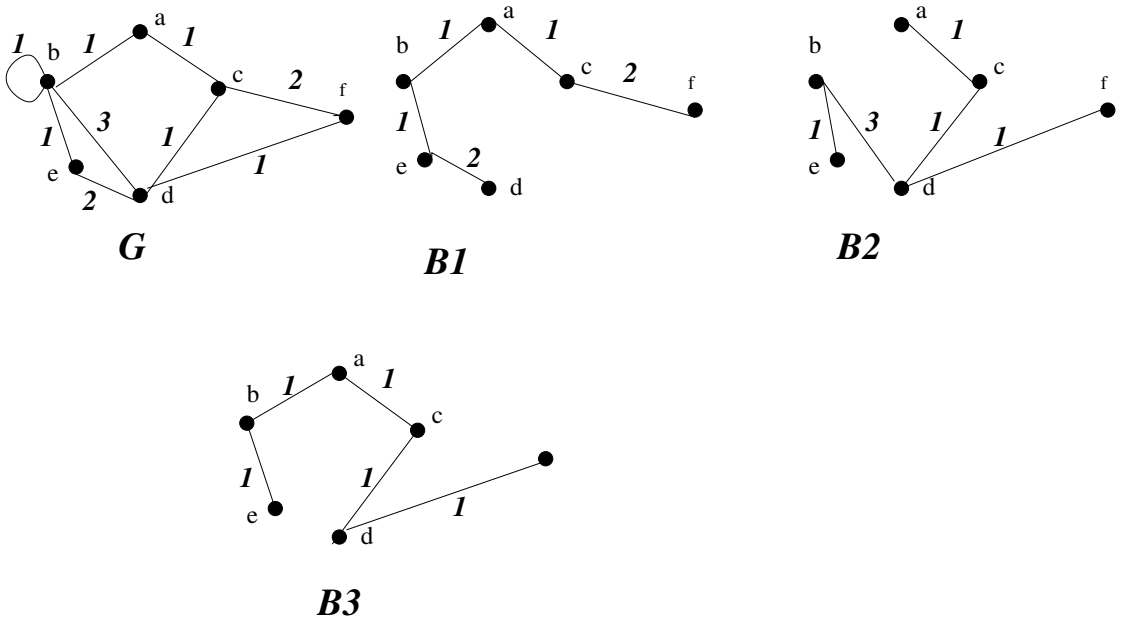


FIGURE 12 – Un graphe G : à côté de chaque sommet on voit une lettre qui est son étiquette et à côté de chaque arête un nombre qui est son coût. Trois arbres couvrants de G : B_1 (de poids non minimum 7) et B_2 (de poids non minimim 6), et un arbre couvrant B_3 de poids minimum 5 (obtenu par l'algorithme 14).

L'algorithme de Prim suit aussi une stratégie gloutonne pour calculer un arbre couvrant de poids minimum. L'arbre couvrant commence d'un sommet s choisi comme racine, auquel on ajoute à chaque étape une arête de poids minimal reliant un des sommets de l'arbre à un sommet non encore connecté à l'arbre. On utilise :

- le tableau *poids* où $poids(x, y)$ donne le poids de l'arête reliant le sommet x au sommet y (si elle existe)
- pour chaque sommet x la liste $ADJ(x)$ des sommets *différents de* x et reliés à x par une arête
- une liste P de sommets non encore connectés à l'arbre, initialisée à s
- un tableau *pred* de paires de sommets (v, n) donnant le prédécesseur (ou père) de chaque sommet v de G dans l'arbre couvrant, initialisée à (s, NIL)

– un tableau *cle* qui donnera pour chaque sommet x de G, le poids minimum d'une arête reliant x à l'arbre couvrant en cours de construction

Algorithme 14: Algorithme de Prim : calcul d'un arbre couvrant de poids minimum d'un graphe G

```
PRIM (graphe G sommet s)
  Données : Un graphe G et un sommet s de G
  Résultat : Un arbre couvrant de G de racine s obtenu avec un parcours en largeur de G
1 Var liste d'entiers L
2 Var sommet v, u
3 Var ensemble de sommets P
4 Var tableau d'entiers cle , poids
5 Var tableau de sommets pred
6 for (v ∈ S \ {s}) do
7     cle[v] = ∞                                          // Initialisations
8 cle[s] = 0
9 P = S
10 pred[s] = NIL
11 while (P ≠ ∅) do
12     u =un sommet de cle minimale dans P        // Appel à la file de priorité P
13     P = P \ {u}
14     for (v ∈ ADJ(u)) do
15         if ((v ∈ P) ∧ (poids(u, v) < cle[v])) then
16             cle[v] = poids(u, v)
17             pred[v] = u
```

On remarquera que l'arbre couvrant ainsi construit n'est pas unique car plusieurs choix sont possibles à la ligne 12.

[

A propos des algorithmes gloutons] Un algorithme glouton fait à chaque étape le choix le meilleur possible à cette instant ; si ce choix est aussi le meilleur possible dans l'absolu, l'algorithme glouton sera alors optimal ; toutefois, il y a des problèmes pour lesquels le choix le meilleur à chaque étape ne donne pas le meilleur choix global possible. Le problème du voyageur de commerce consiste, étant donné un graphe G dont les arêtes sont munies de poids et un sommet s de ce graphe, à trouver un cycle de poids total minimum, partant de s et passant une fois et une seule par chaque sommet. Pour ce problème, l'algorithme glouton choisira à chaque étape la ville « la plus proche » non encore visitée, et il se peut que ce choix soit fort mauvais globalement, voir la figure 13 : sur le graphe G, et en partant du sommet A, la stratégie gloutonne qui consiste à choisir à chaque instant l'arête de poids minimum menant à un sommet non encore visité donne le parcours TG (parcours glouton $ABCDEA$) de poids 55, alors que le parcours optimal TO ($ABECDA$) a un poids de 8.

Comme pour les arbres, on peut parcourir les graphes « en profondeur » , c'est-à-dire visiter tous les descendants du sommet courant s avant de retourner en arrière pour explorer les sommets qui sont « au même niveau » que s. Donnons un algorithme **PARCOURS-GP** de parcours en profondeur de graphe. Comme pour le parcours en largeur, **PARCOURS-GP** utilise le tableau

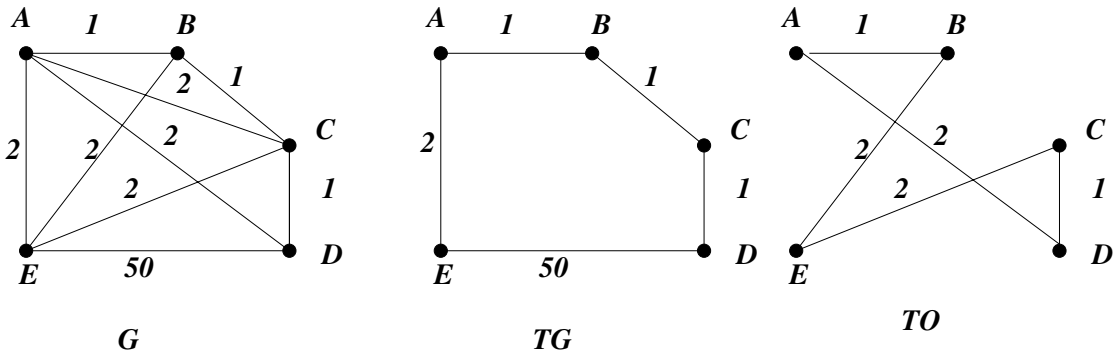


FIGURE 13 – Un graphe G et deux parcours de voyageur de commerce : parcours glouton TG et parcours optimal TO.

de couleurs blanc (non encore visité), gris (en cours de traitement) et noir (traité) pour les sommets, et un tableau *pred* qui donne le prédécesseur (ou père) d'un sommet dans le parcours. On utilise en outre une variable *temps* qui permet d'estampiller chaque sommet x par deux dates, la date $d(x)$ de la première visite (où il est coloré en gris) et la date $f(x)$ de la dernière visite (où il a été traité ainsi que tous ses voisins et où il est coloré en noir). P est la liste des sommets non encore visités (colorés en blanc).

Algorithme 15: Parcours en profondeur de G à partir du sommet s

```
PARCOURS-GP (graphe G, sommet s)
  Données : Un graphe G et un sommet s de G
  Résultat : Parcours en profondeur de G à partir du sommet s
1 Var entier temps
2 Var tableau d'entiers d, f
3 Var tableau de couleurs C
4 Var tableau de  sommets pred
5 Var sommet v
6 Var tableau de  sommets pred
7 for (v ∈ S) do
8 |   C[v] = blanc
9 |   d[v] = f[v] = ∞
10|   pred[v] = NIL
11 temps = 0
12 VISITEP(G, s, temps, f, C, pred)
```

Cet algorithme de parcours en profondeur des graphes est un algorithme récursif (comme **PARCOURS-IN** pour les arbres).

Exercice 19. *1. Calculer la borne inférieure m des poids des chemins parcourant un graphe G en partant d'un sommet et en revenant à ce sommet s après avoir parcouru tous les sommets de G.*

2. Donner une heuristique qui calcule en temps polynomial (en le nombre de sommets de G) un chemin de poids au plus $2m$.

Algorithme 16: Parcours en profondeur de G à partir d'un sommet n

VISITEP (graphe G, sommet u, entier $temps$, tableau entier f, tableau couleur C, tableau sommets $pred$)
Données : Un graphe G avec un marquage indiquant les sommets déjà visités et leurs prédécesseurs, un sommet n de G
Résultat : Parcours en profondeur de G à partir du sommet n

```
1  Var tableau d'entiers d
2  Var sommet v
3  temps = temps + 1
4  C[u] = gris
5  d[u] = temps                                   // Début traitement du sommet u
6  for v ∈ ADJ(u) do
7      if (C[v] == blanc) then
8          C[v] = gris
9          pred[v] = u
10         VISITEP(G, v, temps, f, C, pred)
11     C[u] = noir
12     temps = temps + 1                          // Fin traitement du sommet u
13     f[u] = temps
```

6 Recherche de motifs

Nous étudierons la recherche d'un motif dans un texte, c'est-à-dire étant donnés un motif p et un texte t qui sont deux chaînes de caractères, quelles sont les occurrences de p dans t. En anglais on parle de *pattern matching* . Supposons donnés un texte t sous forme d'un tableau de n lettres prises dans un alphabet $\mathcal{A}$, et un motif p, qui est un tableau de m lettres prises dans $\mathcal{A}$.

6.1 Algorithme naïf de recherche de motif

L'algorithme naïf fait glisser le motif le long du texte, en commençant à la première lettre $t[1]$, puis en comparant lettre à lettre, et ce jusqu'à la position $n - m$ du texte.

Algorithme 17: Recherche des occurrences du motif p dans le texte t

PATTERN-MATCH (chaîne de caractères t, p)
Données : Une chaîne de caractères t et une chaîne de caractères p
Résultat : Les occurrences de p dans t

```
1  Var entier n, m, i
2  n = length(t)
3  m = length(p)
4  for i = 0 to n − m do
5      if (p[1...m] == t[i + 1...i + m]) then
6          afficher p apparaît à la position i dans t
```

Cet algorithme fait $m \times (n - m + 1)$ comparaisons, dont un certain nombre sont inutiles : en effet après chaque échec, on recommence à comparer à partir de l'indice $i + 1$ sans tenir compte

des comparaisons de i à $i+m-1$ qui ont déjà été faites. Certaines de ces comparaisons inutiles peuvent être évitées si l'on prétraite le motif p pour construire un *automate fini* avec lequel on peut ensuite traiter le texte t avec une lecture directe sans retour en arrière.

6.2 Algorithme amélioré utilisant un automate fini

Au motif p on associe un automate fini $A(p)$ qui reconnaît toutes les suites de lettres ayant p pour suffixe. Si $p = ababac$, $A(p)$ est dessiné dans la figure 14 et on peut construire $A(p)$ par l'algorithme 18.

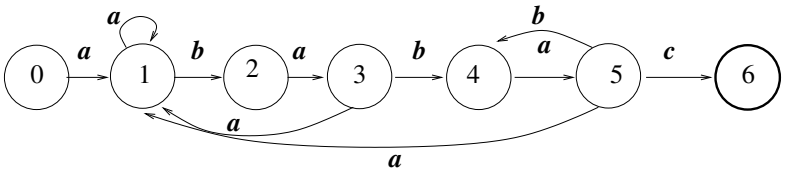


FIGURE 14 – L'automate du motif $p = ababac$: toutes les transitions non dessinées vont à l'état initial 0 ; l'état final est l'état 6. L'état 1 reconnaît les mots se terminant par a, l'état 2 reconnaît les mots se terminant par ab, l'état 3 reconnaît les mots se terminant par aba, l'état 4 reconnaît les mots se terminant par $abab$, l'état 5 reconnaît les mots se terminant par $ababa$, et l'état 6 reconnaît les mots se terminant par $ababac$, c'est-à-dire le motif recherché.

L'algorithme 19 se décompose en deux parties. AUTOM prétraite p et calcule l'automate $A(p)$: ce précalcul de l'algorithme 18 prend un temps $O(m^3 \times S)$, où S est le nombre de lettres de l'alphabet $\mathcal{A}$: en effet les boucles **for** (lignes 4 et 5) sont exécutées $m \times S$ fois, la boucle **repeat** peut s'exécuter $m+1$ fois et le test de la ligne 9 peut effectuer m comparaisons. Ensuite l'algorithme PATTERN parcourt le texte t pour trouver toutes les occurrences de p en temps linéaire en la taille de t en utilisant $A(p)$.

Algorithme 18: Calcul de l'automate des suffixes $A(p)$ associé au motif p

```
AUTOM (chaîne de caractères, p)
  Données : Une chaîne de caractères p
  Résultat : La table des transitions de l'automate A(p)
1 Var entier k j
2 Var tableau δ: {0,...,m} × A ⟶ {0,...,m}
  /* L'automate a m+1 états 0,...,m et ses transitions sont représentées
     par la fonction δ                                                  */
3 m = length(p)
4 for j = 0 to m do
5 |   for (a ∈ A) do
6 |   |   k = min(m+1, j+2)
7 |   |   repeat
8 |   |   |   k = k − 1
9 |   |   until (p[1]...p[j]a est un suffixe de p[1]...p[k])
10|   |   δ(j, a) = k
11 return δ
```

Algorithme 19: L'algorithme de recherche de motif utilisant l'automate des suffixes calculé par l'algorithme 18

```
PATTERN(chaîne de caractères t, p)
  Données : Une chaîne de caractères t et une chaîne de caractères p
  Résultat : Les occurrences de p dans t
1 Var entier i , j
2 n = length(t)
3 m = length(p)
4 δ=AUTOM(p)
5 j = 0                                          // On initialise l'état de l'automate
6 for i = 1 to n do
7 |   j = δ(j, t[i])
8 |   if (j == m)        // L'état final est atteint, une occurrence du motif est
  |   trouvée
9 |   then
10|   |   afficher "p apparaît à la position i + 1 − m dans t"
```

6.3 Algorithme KMP (Knuth-Morris-Pratt)

Cet algorithme améliore encore l'algorithme 19 en évitant même de précalculer la fonction de transition δ de l'automate des suffixes (algorithme 18) : au lieu de précalculer $A(p)$, on précalcule un « tableau des préfixes » $pref(p)$ du motif p ; ensuite, lorsqu'on lit le texte t, on a recours à $pref(p)$ pour calculer *efficacement* et « à la volée » les transitions δ de $A(p)$, uniquement lorsqu'on en aura besoin (et non plus pour chaque lettre $t[i]$ du texte). Illustrons sur un exemple l'idée de KMP. Soit le motif $p[1 \dots 6] = ababac$ et le texte $t[1 \dots 9] = ababcaba$. La comparaison entre le texte et le motif échoue à la cinquième lettre du motif ; toutefois, une observation du motif montre qu'il est inutile de recommencer la comparaison à partir de la deuxième lettre du texte (un b puisqu'elle correspond à la deuxième lettre du motif et pas à la première) ; il faut donc décaler le motif de 2 lettres et repartir de la troisième lettre du texte, dont on sait qu'elle est identique à la première lettre du motif. De plus, il est inutile de comparer les deux premières lettres du motif avec les troisième et quatrième lettres du texte, car on sait déjà qu'elles sont identiques, il suffit donc de repartir de la cinquième lettre du texte, c'est-à-dire du point où l'on avait échoué : au final, on ne parcourra le texte qu'une seule fois, soit un nombre n de comparaisons. Voir la figure 15.

Pour implanter cet algorithme, il faut un prétraitement du motif et calculer un tableau qui donne pour chaque i, $1 \leq i \leq m$, le décalage à faire si la comparaison vient d'échouer à la position i du motif. Dans ce tableau, $pref(p)[i]$ est la longueur du motif initial le plus long se terminant en $p[i]$, *i.e.* la longueur du plus long préfixe de p qui soit un suffixe de $p[1 \dots i]$.

7 Compléments

- Les algorithmes que nous avons donnés pour les arbres binaires dans la section 4 se généralisent facilement aux arbres non binaires (i.e. chaque nœud peut avoir un nombre fini quelconque de fils), du moins lorsque le nombre de fils d'un nœud est *borné*.
- Nous avons choisi une représentation des arbres binaires de recherche qui optimise les opérations standard comme la recherche du maximum, le tri,etc. On peut avoir des représentations différentes : si l'on sait à l'avance quelles sont les opérations qui seront effectuées le plus sou-

Algorithme 20: Algorithme de Knuth-Morris-Pratt : recherche des occurrences du motif p dans le texte t

```
KMP (chaîne de caractères t, p)
  Données : Une chaîne de caractères t et une chaîne de caractères p
  Résultat : Les occurrences de p dans t
1 Var entier n, m, i j
2 Var tableau d'entiers pref
3 n = length(t)
4 m = length(p)
5 i = 0
6 pref =CALCULPREF(p)
7 for j = 0 to n do
8     while ((i > 0) ∧ (p[i + 1] ≠ t[j])) do
9         i = pref[i]
10    if (p[i + 1] == t[j]) then
11        i = i + 1
12    if (i == m) then
13        afficher p apparaît à la position j − m dans t
14        i = pref[i]
```

Algorithme 21: Précalcul du tableau des préfixes de p pour l'algorithme de Knuth-Morris-Pratt

```
CALCULPREF(chaîne de caractères p)
  Données : Une chaîne de caractères p
  Résultat : Le tableau des préfixes de p
1 Var entier n, m, i j
2 Var tableau d'entiers pref
3 m = length(p)
4 i = 0
5 pref[1] = 0
6 for j = 2 to m do
7     while ((i > 0) ∧ (p[i + 1] ≠ p[j])) do
8         i = pref[i]
9     if (p[i + 1] == t[j]) then
10        i = i + 1
11        pref[j] = i
```

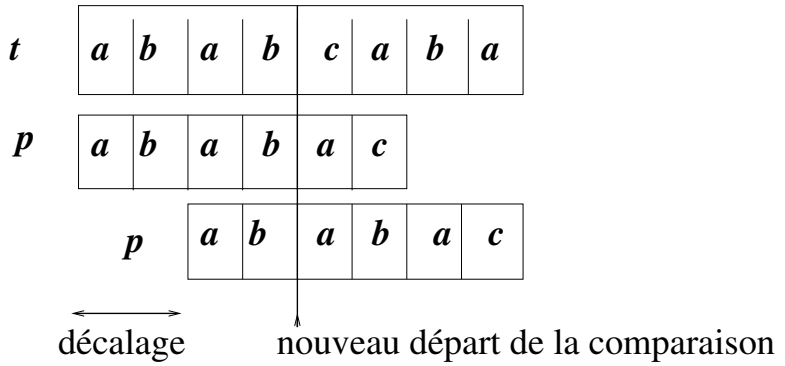


FIGURE 15 – Décalage du motif $p = ababac$ sur le texte $t = ababcaba$ après le premier échec.

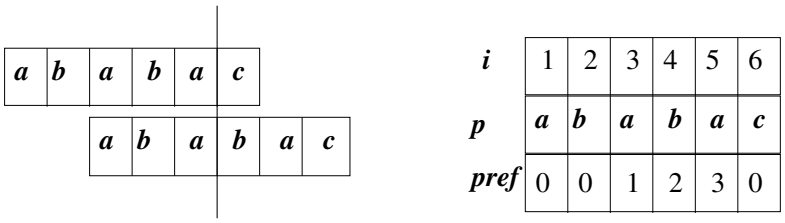


FIGURE 16 – Le tableau $pref(p)$ du motif $p = ababac$.

vent sur ces arbres, on peut choisir des représentations qui minimisent le temps d'exécution des opérations les plus fréquentes.

- L'algorithme Tri Rapide et la construction des arbres binaires de recherche sont similaires et s'appuient sur le même algorithme récursif de type *Diviser pour Régner* (voir [K98] pour une excellente comparaison de ces deux algorithmes).
- Pour des problèmes particulièrement difficiles, dont la solution exacte demande un algorithme qui a un temps d'exécution rédhibitoire (par exemple exponentiel), on peut se contenter d'une *heuristique* qui fournit une solution non optimale, ou une solution approchée, mais en un temps raisonnable, par exemple linéaire. Un exemple est l'algorithme glouton pour le problème du voyageur de commerce voir la figure 13.
- **L'agorithme KMP** se généralise à des recherche de motifs « avec trous » , par exemple le motif *vie* apparaît dans le mot *ville* si l'on permet des trous (pour reprendre la publicité « dans *ville* il y a *vie* »). Cette problématique est particulièrement importante dans les études portant sur la génomique ou la fouille de données.

8 Algorithme de codage de Huffman

Coder consiste à représenter des objets par des mots écrits avec des lettres choisies dans un alphabet de référence et dont le sens est partagé. Les exemples historiques sont nombreux, le codage en Morse, le télégraphe Chappe, le code secret de César et bien d'autres. Actuellement, le codage numérique de l'information représente une partie essentielle de l'informatique. Dans ce cas l'information est représentée par des mots binaires (écrits avec des 0 et des 1), qui pourront

facilement être manipulés par un ordinateur. Le code ASCII (*American Standard Code for Information Interchange*) est l'un des codes standardisés les plus utilisés permettant de représenter des caractères de l'alphabet latin pur.

Une des propriétés fondamentales d'un code est sa **non-ambiguïté**, c'est-à-dire qu'il n'existe aucun texte codé qui puisse avoir deux transcriptions différentes. Le code ASCII, qui code les lettres de l'alphabet sur un octet (8 bits dont les 7 premiers codent les 127 lettres, et le 8ème bit est un zéro) est non-ambigu car il y a correspondance entre un octet et une lettre.

Un code $\mathcal{C}$ est donc une bijection entre un ensemble (alphabet) de symboles $\mathcal{A} = \{a_1, \cdots, a_K\}$ et des mots binaires $m_1, m_2, \cdots, m_K$. On dira qu'un code est préfixe si aucun des codages m_i n'est un préfixe (c'est-à-dire le début) d'un autre codage.

Exercice 20. *Montrer qu'un code ayant la propriété du préfixe n'est pas ambigu.*

On peut représenter tout code par un arbre binaire : certains nœuds de l'arbre sont étiquetés par des lettres de $\mathcal{A}$ et le codage de la lettre qui étiquette n est le numéro du nœud n (avec la sémantique déjà donnée pour numéroter les nœuds : fils gauche codé par 0 et fils droit par 1). Un code est préfixe si seules les feuilles de l'arbre binaire sont étiquetées par les lettres de l'alphabet $\mathcal{A}$. La longueur du codage d'une lettre dans un code préfixe est la distance entre la racine de l'arbre et la feuille correspondante. Sur la figure 17.**A**(resp. 17.**B**) on a un code non préfixe (resp. un code préfixe).

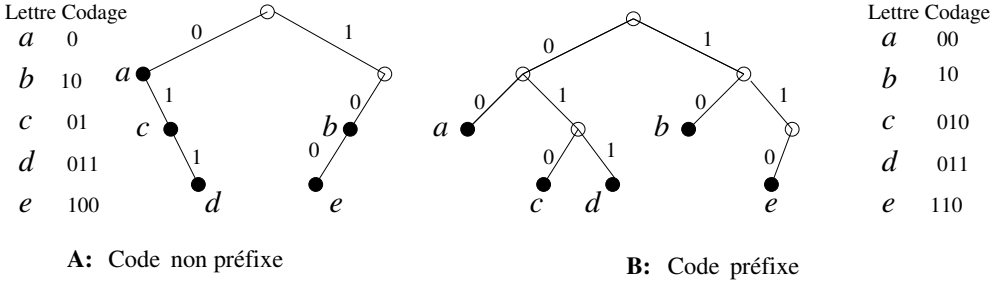


FIGURE 17 – Un code non préfixe **A** et un code préfixe **B**.

L'intérêt d'un code de longueur variable est d'adapter la longueur du codage de chaque lettre à sa fréquence d'apparition dans le texte à coder : plus une lettre est fréquente plus son code doit être court. On notera l_i la longueur du codage m_i et p_i la proportion (ou fréquence ou probabilité) de la lettre S_i dans le texte à coder.

Le problème à résoudre est de trouver un code binaire préfixe qui minimise la longueur moyenne du code,

$$L(\mathcal{C}) = \sum_{i=1}^{K} p_i l_i.$$

Ceci revient à construire un arbre binaire pour lequel la moyenne pondérée des feuilles soit minimale. Pour réaliser cet arbre, on construit une solution en partant des feuilles les plus profondes puis on construit l'arbre progressivement en les combinant deux à deux.

L'algorithme de Huffman produit un code préfixe de longueur moyenne optimale. C'est un autre exemple d'algorithme **glouton**. L'algorithme nécessite donc de l'ordre de K opérations d'insertion et d'extraction de F et, si la file de priorité est implantée dans une structure de données adéquate (par exemple un tas), le coût d'insertion et d'extraction est majoré par $\log K$. Ce qui donne à cet algorithme une complexité de l'ordre de $\mathcal{O}(K \log K)$.

Algorithme 22: Algorithme de Huffman (1951)

```
ALGORITHME_HUFFMAN
  Données : Un ensemble S de K symboles avec leurs pondérations p
  Résultat : Un arbre optimal de codage (nœud racine)
  Indice i
  Indice x,y,z
  File de Priorité F
  /* F est une file de priorité dans laquelle on insère des couples (nœud,
     priorité) et on extrait les couples selon leur priorité              */
  /* pour cet algorithme, l'extraction se fera par poids de nœud croissant */
  for (s ∈ S)                                        // Initialisation de la forêt
  do
     z=nouveau_nœud() ; z.symbole=s ; z.poids=p(s)
     Insérer (F,z)
  for i = 1 to K − 1                                          // Itération principale
  do
     // Il existe un arbre de codage optimal dont les arbres contenus dans F
        sont des sous-arbres
     // F contient K − i + 1 nœuds
     x=Extraire (F) ; y=Extraire (F) ;
     z=nouveau_nœud() ; z.gauche=x ; z.droit=y
     z.poids=x.poids+y.poids
     Insérer (F,z)
  Renvoie Extraire (F)
```

Exercice 21. *Pour un alphabet de $K = 4$ symboles donner un code optimal en fonction de $p_1, \cdots, p_4$. Indication : on peut supposer sans perte de généralité que les p_i sont ordonnés, on peut également supposer que l'arbre est complet (argument d'optimalité). Il y a 2 types d'arbres complets ayant 4 feuilles et le choix de l'arbre se fait sur la comparaison de p_1 à $p_3 + p_4$.*

Exercice 22. *Soit A un arbre binaire ayant K feuilles dont les distances à la racine sont respectivement $l_1, \cdots, l_K$. Montrer l'inégalité de Kraft*

$$\sum_i 2^{-l_i} \leq 1.$$

9 Questions d'enseignement

Il y a d' innombrables et excellents ouvrages sur l'algorithmique ; par exemple on peut tout trouver dans [CLRS09, K98].

On pourra utiliser les exemples et exercices de la section 9.5 pour donner une première intuition sur les algorithmes. D'autres exemples sont donnés ci-dessous :

- l'algorithme d'Euclide pour calculer le pgcd.

- la résolution d'une équation du second degré permet d'introduire les *tests* ; la méthode est presqu'un vrai algorithme mais il y a la difficulté du test de nullité du discriminant : il faut tester la valeur du discriminant Δ, et la suite des calculs est différente selon que $\Delta = 0$ ou non.

- le calcul du pgcd permet de montrer la nécessité d'aller au-delà de l'*algorithme naïf*.

- l'établissement d'une table de logarithmes permet de montrer la possibilité d'erreurs dues à des causes diverses lors de l'exécution à la main et l'utilité des *bureaux de calcul*.

Il faudra aussi expliquer qu'un algorithme *ne se réduit pas à* un programme : un algorithme est la description précise d'une méthode pour résoudre un problème, alors qu'un programme est l'expression concrète d'un algorithme dans un langage de programmation, destinée à être exécutée sur une machine. De même que « *to work* » et « *travailler* » expriment une même action selon qu'on parle anglais ou français, de même « $i = i + 1$ » et « $i{+}{+}$ » expriment le même algorithme « *ajouter 1 à i* » selon qu'on « parle » *Java* ou *C*.

9.1 Propriétés des algorithmes

Après avoir donné quelques exemples d'algorithmes, nous conseillons vivement à l'enseignant de dire explicitement quelles sont les propriétés qu'un algorithme doit avoir ; nous reprenons ici la description donnée par [K98], qui caractérise un algorithme comme étant une « recette de cuisine » , ou encore un nombre fini de règles qu'on applique dans l'ordre donné dans la « recette » pour obtenir la solution d'un problème. Toutefois, cette « recette » doit satisfaire aux contraintes suivantes [K98] :

- Définition non ambiguë : chaque étape d'un algorithme doit pouvoir être effectuée par un ordinateur ou un robot et ce d'une seule manière possible. Par exemple l'algorithme de PRIM donné plus haut ne satisfait pas cette condition de non-ambiguïté puisque à la ligne 12, plusieurs choix sont possibles. Notons toutefois que les « algorithmes » non-déterministes et probabilistes sont de plus en plus utilisés.
- Définition effective : chaque étape d'un algorithme doit pouvoir être effectuée par un humain de manière exacte et en un temps fini. Par exemple il n'y a pas d'algorithme pour diviser deux réels donnés par une représentation décimale infinie.
- Données : l'algorithme prend en entrée des données qui sont précisées par le problème à résoudre.
- Résultats : on peut montrer que l'algorithme fournit des résultats qui sont ceux demandés par le problème à résoudre.

9.2 Preuves des algorithmes

Nous avons illustré la *complexité des algorithmes* (voir les algorithmes 1, 23, etc.). On peut aussi considérer l'aspect *preuve de correction* des algorithmes, à savoir, le résultat de l'algorithme est-il bien le résultat attendu ? Ceci peut se faire au moyen d'assertions que l'on met à des lignes stratégiques de l'algorithme et que l'on doit vérifier (voir par exemple les assertions mises entre /* */ dans les algorithmes 1, 23). Toutefois, il faut ensuite *démontrer* ces assertions formellement : on pourra se reporter au chapitre 14.3 de [AG06] pour des exemples détaillés de preuves d'algorithmes simples.

9.3 Questions d'implantation

Remarquons que les $\wedge$ qui figurent dans les conditions d'arrêt des boucles **while** (ligne 3 de l'algorithme 6, ligne 5 de l'algorithme 7, ligne 3 de l'algorithme 11, etc.) sont implantés comme des opérateurs *paresseux* ou *séquentiels*, c'est-à-dire : on évalue d'abord le premier booléen de la conjonction, *ensuite* et *uniquement* si ce premier booléen est vrai, on évalue le booléen suivant de la conjonction. Si l'on évaluait en parallèle, ou dans un autre ordre, ces booléens, on pourrait avoir des erreurs : par exemple ligne 6 de l'algorithme 23, si $i = 0$, $T[i]$ n'est pas défini.

9.4 Questions de programmation

Dans l'algorithme 13 par exemple, on doit calculer plusieurs tableaux indexés par des sommets, or les indices d'un tableau sont normalement des entiers : comment faire ? Par exemple, pour calculer *pred*, on calculera un tableau de couples $(n, pere(n))$ où $pere(n)$ est le sommet de G qui sera le père de n dans l'arbre couvrant : on déclarera un tableau *pred* de paires de sommets (Var tableau de couples de sommets *pred*) indexé par un entier (Indice entier i, initialisé à $i = 1$) ; la ligne 8 $pred(s) = NIL$ de l'algorithme 13 sera remplacée par $pred(1) = (s, NIL)$, et la ligne 16 $pred(v) = n$ sera remplacée par les deux lignes

$i = i + 1$

$pred(i) = (v, n)$.

De même le tableau de couleurs C sera programmée comme un tableau à trois colonnes (indice, sommet, couleur).

9.5 Exercices du Rosen

Les exercices suivants sont tirés de la page WEB de l'excellent livre de Rosen ; voir `http://www.mhhe.com/math/advmath/rosen/` pour des compléments.

Exercice 23. *Ecrire un algorithme qui donne le chiffre des dizaines d'un nombre.*

Exercice 24. *Donner un algorithme qui donne l'indice du premier entier pair dans un tableau d'entiers, et 0 s'il n'y a aucun nombre pair dans ce tableau.*

Exercice 25. *Donner un algorithme qui détermine si un tableau est trié (i.e. ses éléments sont rangés par ordre croissant).*

Exercice 26. *Donner un algorithme qui détermine s'il existe un élément $T[i]$ d'un tableau qui soit la moyenne d'un élément d'indice inférieur à i et d'un élément d'indice supérieur à i.*

10 Résumé des définitions principales

– **Algorithme** Description précise des actions à mener pour résoudre un problème donné en un nombre fini d'étapes.

- **Algorithme de tri** Algorithme qui, étant donnée une suite de n nombres, les range par ordre croissant.
- **Algorithme de recherche** Algorithme qui recherche un élément dans une structure (qui peut être une liste, un tableau, un arbre, etc.).
- **Algorithme de recherche linéaire ou séquentielle pour les tableaux** Algorithme qui recherche un élément dans un tableau en parcourant le tableau et en lisant tous ses éléments. Exemple : l'algorithme RECHERCHE (6).
- **Algorithme de recherche dichotomique (ou de type "diviser pour régner")** Algorithme qui recherche un élément dans un tableau ordonné en coupant le tableau en deux à chaque étape pour ne chercher que dans une des deux parties. Exemples : l'algorithme RECHERCHEDICHO (7), QUICKSORT (4), l'algorithme TRI-FUSION (3), et aussi les algorithmes MAX ou MIN dans les arbres binaires de recherche (12).
- **Parcours** Un parcours d'un arbre (resp. graphe) est une liste des sommets de cet arbre (resp. graphe).
- **Algorithme glouton** Un algorithme glouton est un algorithme qui optimise en faisant à chaque étape le meilleur choix instantané possible à cette étape. Exemples : les algorithmes PARCOURS-LG (13), PRIM(14), HUFFMAN (22).
- **Arbre couvrant** Un arbre qui contienne tous les sommets d'un graphe.

11 Bibliographie

[AG06] André Arnold, Irène Guessarian, *Mathématiques pour l'Informatique*, 4eme ed., Edi-Science, Dunod (Paris), 2005.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, 3eme ed., MIT Press, London, 2009.

[K98] Donald E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, 2eme ed., Addison-Wesley, Reading, Massachusetts, (1998).

[R99] Kenneth H. Rosen, *Discrete Mathematics and its Applications*,McGrawHill, New York, 1999.

12 Solutions de certains exercices

Exercice 1

En utilisant un ordinateur actuel cadencé à 3GHz et en supposant qu'une comparaison s'effectue en dix cycles machines (en général c'est beaucoup plus) on a besoin de (en secondes)

$$\frac{(100000000)^2}{3000000000/10} = \frac{10^8}{3 \times 10^8} = \frac{10^8}{3}$$

soit $10^8/180$ minutes, ou

$$\frac{10^8}{540} \sim 10^6/5 \sim 2 \times 10^5 \text{heures}$$

soit environ $10^3 = 1000$ jours.

Exercice 2

```
public class tris0
{
    public static void main (String args[])
```

```
    {
int[] Tab;
Tab=new int[5];
        saisie(Tab);
        tri(Tab);
affiche(Tab);
    }
    public static void saisie (int[] T)
    {T[0]=7;
    T[1]=1;
    T[2]=15;
    T[3]=8;
    T[4]=2;
    }
    public static void tri(int[] T)
    {int i,j,min;
    for ( i=0;i<T.length-1;i++)
{min=i;
for (j=i+1;j<T.length;j++)
    if (T[j]<T[min])
                min=j;
echange(T,i,min);
        }
    }
    public static void affiche(int[] T)
    {
for (int i=0;i<T.length;i++)
    System.out.print(T[i]+"  ");
System.out.println();
    }
    public static void echange(int[] T, int a, int b)
    {int c;
    c=T[a];
    T[a]=T[b];
    T[b]=c;
    }
}
```

Pour pouvoir travailler sur un tableau de dimension non précisée à l'avance, il faut modifier le début du programme précédent en :

```
   static final int MAX=100;

   public static void main (String args[]) throws IOException
   {
        int[] Tab;
        int[] dim;
        Tab=new int[MAX];
        dim = new int[1];
        saisie(Tab, dim);
        tri(Tab,0,dim[0]-1);
```

```
        affiche(Tab,dim[0]);
    }

    static void saisie (int[] T, int[] dim) throws IOException
   {
   InputStreamReader isr;
   isr=new InputStreamReader(System.in);
   BufferedReader br;
   br= new BufferedReader(isr);
   String s;

   System.out.print("Dimension du tableau = ");
   s = br.readLine();
   dim[0] = Integer.parseInt(s);

   for (int i=0 ; i<dim[0];i++)
        {
            System.out.print("T["+i+"] = ");//("Dimension du tableau svp :  ");
            s = br.readLine();
            T[i]=Integer.parseInt(s);
            //T=new int[n];
         }
    }
```

Exercice 3

Algorithme 23: Tri par insertion

INSERTION (tableau T, entier n)
Données : Un tableau T de n entiers
Résultat : Le tableau T contient les mêmes éléments mais rangés par ordre croissant

```
1 Var entier k
2 Indice entier i, j
3 for j = 1 to n do
4 |   k = T[j]
  |   /* Les j − 1 premiers éléments de T sont ordonnés et on va insérer T[j] à
  |      sa place dans le tableau T[1..(j − 1)]                                  */
5 |   i = j − 1
6 |   while ((i > 0) ∧ (T[i] > k)) do
7 |   |   T[i + 1] = T[i]                          // On décale le i-ème élément
8 |   |   i = i − 1
9 |   T[i + 1] = k
  |   /* Les j premiers éléments de T sont ordonnés                              */
```

Le tri par insertion fonctionne en *espace constant* : on n'utilise qu'un nombre constant de nombres hors de la liste.

Soit $t(n)$ la *complexité en temps*, comptée en nombre d'opérations :

– le meilleur cas est celui où la liste est déjà triée, et la complexité est *linéaire* car $t(n)$ est une fonction linéaire de n,

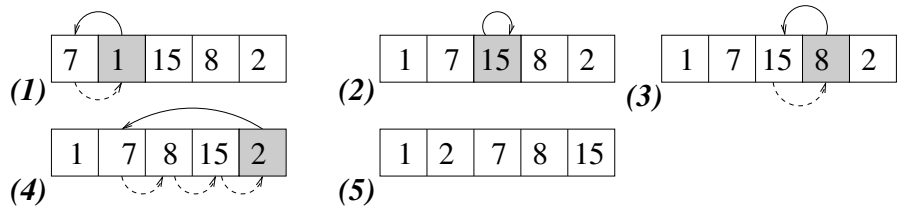


FIGURE 18 – Comment INSERTION trie la liste 7, 1, 15, 8, 2.

- le pire cas est celui où la liste est en ordre décroissant, alors pour chaque j entre 2 et n, il faut faire dans la boucle **while** $C \times j$ opérations, et la complexité en temps est de l'ordre de $\sum_2^n C \times j = C \times n(n-1)/2$, c'est une fonction quadratique de n,
- la complexité en moyenne, en supposant qu'en moyenne dans la boucle **while** la moitié des nombres seront inférieurs à $T[j]$, sera aussi quadratique car il faudra faire $C \times j/2$ opérations pour chaque itération de la boucle **while**.

Exercice 5

```
import java.io.*;

public class trif1
{
   static final int MAX=100;

   public static void main (String args[]) throws IOException
   {
int[] Tab;
        int[] dim;
        Tab=new int[MAX];
        dim = new int[1];
        saisie(Tab, dim);
        tri(Tab,0,dim[0]-1);
affiche(Tab,dim[0]);
   }

   static void saisie (int[] T, int[] dim) throws IOException
   {
   InputStreamReader isr;
   isr=new InputStreamReader(System.in);
   BufferedReader br;
   br= new BufferedReader(isr);
   String s;

   System.out.print("Dimension du tableau = ");
   s = br.readLine();
   dim[0] = Integer.parseInt(s);

   for (int i=0 ; i<dim[0];i++)
{
```

```
    System.out.print("T["+i+"] = ");            s = br.readLine();
    T[i]=Integer.parseInt(s);
    //T=new int[n];
         }
   }

   static void tri(int[] T, int l,int r )
   {
   int m;
   if (l<r)
{
    m=(l+r)/2;
    tri(T,l,m);
              tri(T,m+1,r);
              fusion(T,l,r,m);
         }
    }

    static void fusion(int[] T,int l,int r,int m)
      {
       int i,j,k,n1,n2;
       int[] L,R;

       n1=m-l+1;
       n2=r-m;

       L=new int[n1];
       R=new int[n2];
       for(i=0;i<n1;i++) L[i]=T[l+i];
       for(j=0;j<n2;j++) R[j]=T[m+1+j];
       i=0;
       j=0;

       System.out.print("L = ");affiche(L,n1);

       System.out.print("R = ");affiche(R,n2);

       for (k=l; k<=r; k++)
  {
    if ((i<n1)&&(j<n2))
     {
              if (L[i]<=R[j])
  {
  T[k]=L[i];
  i++;
                   }
              else
  {
                   T[k]=R[j];
  j++;
```

```
  }
     }
  else if ((j>=n2)&&(i<n1))
      {
  T[k]=L[i];
  i++;
     }
  else if ((i>=n1)&&(j<n2))
      {
  T[k]=R[j];
  j=j++;
     }


  if ((i>n1)&&(j>n2)) {System.out.print("T = ");affiche(T,MAX);}
  }

      }
     static void affiche(int[] T, int n)
    {
for (int i=0;i<n;i++)
    System.out.print(T[i]+"  ");
System.out.println();
    }

}
```

Exercice 6 La classe Tableau est définie comme suit

```
import java.io.*;

public class Tableau
{
    //public static final int MAX=100;

   public static void saisie(int[] T, int[] dim)  throws IOException
   {
       //T=new int[MAX];
       //dim = new int[1];

   InputStreamReader isr;
   isr=new InputStreamReader(System.in);
   BufferedReader br;
   br= new BufferedReader(isr);
   String s;

   System.out.print("Dimension du tableau = ");
   s = br.readLine();
   dim[0] = Integer.parseInt(s);

   for (int i=0 ; i<dim[0];i++)
```

```
        {
            System.out.print("T["+i+"] = ");
            s = br.readLine();
            T[i]=Integer.parseInt(s);
            //T=new int[n];
         }
    }

    public static int max (int[] T, int n) throws IOException
    {
        int m=T[0];
        int i=0;

        while (i<n)
    { i++;
    if (T[i]>m) m=T[i];
    };
return m; }

    public static void affiche(int[] T, int n)
    {
        for (int i=0;i<n;i++)
            System.out.print(T[i]+"  ");
        System.out.println();
    }
    public static void echange(int[] T, int a, int b)
    {int c;
    c=T[a];
    T[a]=T[b];
    T[b]=c;
    }

}
```

L'algorithme 3 s'écrit maintenant

```
import java.io.*;

public class trifsent
{
   static final int MAX=100;

   public static void main (String args[]) throws IOException
   {
        int[] Tab;
        int[] dim;
        Tab=new int[MAX];
        dim = new int[1];
        Tableau.saisie(Tab, dim);
        int m1=Tableau.max(Tab,dim[0]);
```

```
        int inf=m1;
        tri(Tab,0,dim[0]-1,inf);
        Tableau.affiche(Tab,dim[0]);
    }

    static void tri(int[] T, int l,int r,int M )
    {
    int m;
    if (l<r)
{

m=(l+r)/2;/
tri(T,l,m,M);
       tri(T,m+1,r,M);
       fusion(T,l,r,m,M);
       }

    }

    static void fusion(int[] T,int l,int r,int m,int M)
      {
       int i,j,k,n1,n2;
       int[] L,R;

       n1=m-l+1;
       n2=r-m;
       System.out.println("n1 = " + n1+ "  n2 = " + n2 );

       L=new int[n1+1];
       R=new int[n2+1];


       for(i=0;i<n1;i++) L[i]=T[l+i];
       for(j=0;j<n2;j++) R[j]=T[m+1+j];
       L[n1]=M;
       R[n2]=M;
       i=0;
       j=0;

       k=l;
       for (k=l; k<=r; k++)
   if ((i<n1)&&(L[i]<=R[j]))
       {
   T[k]=L[i];
   i++;
       }
   else
       {
   T[k]=R[j];
   j++;
```

```
        }


      }
}
```

On pourrait remplacer la ligne
`int inf=m1 ;`
par
`int inf=m1+1 ;`
ce qui permettrait d'écrire plus simplement
`if (L[i]<=R[j])`
au lieu de
`if ((i<n1)&&(L[i]<=R[j]))`
i.e. de ne pas tester si on est à la fin du tableau `L` : toutefois cette méthode conduira à une erreur si `m1` est le plus grand élément exprimable dans le type int (car alors `m1+1` est égal à `-m1-1`).

Exercice 7

Algorithme 24: Recherche dichotomique de la première occurrence de k dans un tableau à 2^n éléments

RECHERCHE-DIC-PUIS2 (tableau T, entier k)
Données : Un tableau $T[1 \dots 2^n]$ d'entiers déjà ordonné et un entier k
Résultat : Le premier indice i où se trouve l'élément k, ou bien -1(par convention si k n'est pas dans T)

```
1  Indice entier i, l, r
2  l = 1
3  r = 2^n
4  i = 2^{n-1}
5  while ((k ≠ T[i]) ∧ (l ≤ r)) do
6  |  if (k < T[i]) then
7  |  |  r = i
8  |  else
9  |  |  l = i + 1
10 |  n = n − 1
11 |  i = l + 2^{n-1}
12 if (k == T[i]) then
13 |  repeat
14 |  |  i = i − 1
15 |  until (k = T[i])
16 |  retourner i
17 else
18 |  retourner −1
```

On remarquera l'utilisation d'une boucle **repeat**.

2. L'algorithme récursif naïf

On remarquera que l'algorithme récursif 25 est beaucoup plus concis. Toutefois, cet algorithme récursif naïf prendra toujours un temps $\log n$, alors que l'algorithme itératif peut être beaucoup

Algorithme 25: Recherche dichotomique récursive de la première occurrence de k dans un tableau à 2^n éléments

RECH-DIC-REC-PUIS2 (tableau T, entier k)
Données : Un tableau $T[l+1 \dots l+2^n]$ d'entiers déjà ordonné et un entier k
Résultat : Le premier indice i où se trouve l'élément k, ou bien -1(par convention si k n'est pas dans T)

```
1  Indice entier i
2  i = l + 2^{n-1}
3  if (k ≤ T[i]) then
4  |  RECH-DIC-REC-PUIS2(T[1 ... 2^{n-1}], k)
5  else
6  |  RECH-DIC-REC-PUIS2(T[2^{n-1} + 1 ... 2^n], k)
7  if ((i == l) ∧ (k == T[i])) then
8  |  retourner i
9  else
10 |  retourner −1
```

plus rapide (si par exemple k se trouve au milieu de la liste, l'algorithme 24 terminera en temps constant) ; bien sûr la complexité dans le pire cas des deux algorithmes est $O(\log n)$.

Exercice 8

La racine est numérotée ε et son étiquette est $T(\varepsilon) = 100$, les autres nœuds : $T(0) = 22$, $T(1) = 202$, $T(01) = 53$, $T(10) = 143$, $T(011) = 78$, $T(101) = 177$ et il y a deux feuilles étiquetées 78, 177.

Exercice 9

1. On a compté pour 0 le temps de parcours d'un arbre vide (base de la récurrence), or il faut au moins tester si l'arbre est vide. Cet exercice montre la difficulté d'évaluer la complexité d'un algorithme, même simple, *et* le soin qu'il faut apporter aux preuves par récurrence.

2. Soit c' le temps d'exécution de la ligne 3 de PARCOURS-IN, alors, pour le cas de base de la récurrence $t(0) = c'$, et ensuite, en posant l'hypothèse de récurrence : $t(T) \leq (c' + 2c)k$ si k est le nombre de nœuds de T, on a $t(T) \leq c' + 2c + 2c\ nnoeud(sous\text{-}arbre\text{-}gauche(T)) + 2c\ nnoeud(sous\text{-}arbre\text{-}droit(T)) = (c' + 2c)n$, ce qui donne encore un temps linéaire.

Exercice 12

Le pire cas se produit lorsque la liste est triée, alors l'arbre binaire de recherche correspondant est un arbre filiforme, qui se réduit à sa branche d'extrême droite, et pour ajouter le $(i+1)$ème élément de la liste, il faudra parcourir les i nœuds déjà crées, d'où le temps de calcul pour une liste de longueur n : $t(n) = C \times \sum_{i=1}^{n-1} i = C \times n(n-1)/2 = O(n^2)$, $t(n)$ est donc quadratique. On remarquera que, pour Tri Rapide aussi, la pire complexité est obtenue pour une liste triée et elle est quadratique aussi.

Exercice 13

```
class Node
{
    //attributs
    int val;
    Node leftChild;
    Node rightChild;
    //methodes
```

Algorithme 26: Insertion (itérative) d'un nombre à une feuille dans un arbre T

```
INSERER (arbre T, entier v)
  Données : Un arbre binaire de recherche T dont les nœuds sont étiquetés par des entiers
            et un entier v à insérer dans T
  Résultat : L'arbre T avec une nouvelle feuille étiquetée v
 1 Var nœud x, y
 2 Var arbre T'
 3 T' = T
 4 y = NIL
 5 x = racine(T)
 6 while (x ≠ NIL) do
 7 |   y = x
 8 |   if (v < T(x)) then
 9 |   |   T' = sous-arbre-gauche(T')
10 |   if (v > T(x)) then
11 |   |   T' = sous-arbre-droit(T')
12 |   x = racine(T')
13 pere(x) = y
14 if (T == ∅) then
15 |   T = (v, ∅, ∅)
16 else
17 |   if (v < T(y)) then
18 |   |   sous-arbre-gauche(y) = (v, ∅, ∅)
   |   /* le sous-arbre gauche vide de y est remplacé par (v, ∅, ∅)          */
19 |   if (v > T(y)) then
20 |   |   sous-arbre-droit(y) = (v, ∅, ∅)
   |   /* le sous-arbre droit vide de y est remplacé par (v, ∅, ∅)           */
```

```
    public Node(int v, Node lc, Node rc)
    {
val=v;
leftChild=lc;
rightChild=lc;
    }

    public int getVal()
    {
return val;
    }
    public Node getLC()
    {
return leftChild;
    }
    public Node getRC()
    {
        return rightChild;
    }
    public void putLC(Node lc)
    {
        leftChild = lc;
    }
    public void putRC(Node rc)
    {
        rightChild = rc;
    }

}

public class Arbre
{
    //attributs
    Node root;
    //Arbre filsG;
    //methodes

    public Arbre()
    {
     root=null;
    }
    public Arbre(Node nd)
    {
root = nd;
    }
    public Node getRoot()
    {
return root;
    }
    public void insert(int n)
```

```
    {
if (root==null) root=new Node(n,null,null);
       else
   if (n<root.getVal())
                     {
 Arbre filsG;
 filsG=new Arbre(root.getLC());
 filsG.insert(n);
 root.putLC(filsG.getRoot());
             }
   else
       if (n>root.getVal())
   {
       Arbre filsD;
       filsD=new Arbre(root.getRC());
       filsD.insert(n);
       root.putRC(filsD.getRoot());
   }

    }

    public void parcoursInfixe()
    {
if (root==null) System.out.print("ff");
else
{
    Arbre filsG;
    filsG=new Arbre(root.getLC());
    filsG.parcoursInfixe();
    System.out.print(root.getVal()+ "  ");
             Arbre filsD;
    filsD=new Arbre(root.getRC());
    filsD.parcoursInfixe();

                }
    }

    public void parcoursPrefixe()
    {
        if (root==null) System.out.print("f");
        else
    {
System.out.print(" "+root.getVal()+ "  ");
                Arbre filsG;
filsG=new Arbre(root.getLC());
filsG.parcoursPrefixe();
       Arbre filsD;
filsD=new Arbre(root.getRC());
filsD.parcoursPrefixe();
    }
```

```
    }
}
```

Exercice 14

```
public class constArbre
{
    public static void main(String[] args)
    {
Arbre A;
A=new Arbre();
A.insert(100);
A.insert(22);
A.insert(202);
A.insert(143);
A.insert(53);
A.insert(177);
A.insert(78);

A.parcoursInfixe();
        System.out.println();
A.parcoursPrefixe();
        System.out.println();
    }
}
```

Exercice 15

1. Il suffit de remplacer « droit » par « gauche » dans l'algorithme **MAX**.

2. Si le sous-arbre droit de n est non vide, le successeur de n est le minimum de ce sous-arbre ; sinon,
   - soit n est le plus grand nombre de l'arbre T
   - sinon, si n n'est pas le plus grand nombre de T, son successeur est l'ancêtre x de n le plus proche de n tel que le fils gauche de x soit aussi un ancêtre de n (la relation ancêtre est réflexive, c'est-à-dire que n est un ancêtre de lui-même). Par exemple dans la figure 8 le successeur du nœud étiqueté 78 est le nœud étiqueté 100. Le programme s'écrit :

Exercice 19 Voir [CLRS09] section 2.1.1.

Exercice 23 Voir l'algorithme 28.

Exercice 24 Le premier algorithme 29 parcourra tout le tableau, le second algorithme 30 sera meilleur car il s'arrête dès qu'on trouve un nombre pair. L'inconvénient de cet algorithme est de parcourir tout le tableau, même après avoir trouvé un nombre pair ; il vaut mieux faire une boucle **While** qui s'arrête dès qu'on trouve le premier nombre pair.

Exercice 25 Voir l'algorithme 31.

Exercice 26 Voir l'algorithme 32. On remarquera qu'il y a trois boucles imbriquées, car pour chaque indice i, et chaque $j < i$ il faut essayer tous les éléments d'indice $k > i$.

Algorithme 27: Recherche du successeur d'un nœud d'un arbre binaire de recherche T

SUCCESSEUR (arbre T nœud n)
Données : Un arbre binaire de recherche T dont les nœuds sont étiquetés par des entiers deux à deux distincts et un nœud n
Résultat : L'étiquette du nœud de T qui est le successeur de n dans le parcours infixe

```
1  Var nœud n, x, y
2  if (T(filsdroit(n)) ≠ NIL) then
3  |  MIN(sous-arbre-droit(n))
4  else
5  |  x = pere(n)
6  |  y = n
7  while ((T(x) ≠ NIL) ∧ (y == filsdroit(x))) do
8  |  y = x
9  |  x = pere(x)
10 afficher le successeur de T(n) est T(x)
11 retourner T(x)
```

Algorithme 28: Chiffre des dizaines de N

DIZAINE(entier N, entier N, entier D, entier U)
Données : Un entier N
Résultat : Le chiffre des dizaines D de N

$N' = N - 100 \left\lfloor \frac{N}{100} \right\rfloor$ // N' est le nombre formé par le chiffre des dizaines de N suivi du chiffre des unités de N

$U := N' - 10 \left\lfloor \frac{N'}{10} \right\rfloor$ // U est le chiffre des unités de N

$D = N' - U$

Algorithme 29: Indice du premier nombre pair d'un tableau

PREMIER-PAIR(tableau T, entier N)
Données : Un tableau T de N éléments comparables
Résultat : L"indice du premier nombre pair dans T, et 0 si T ne contient aucun nombre pair

```
Indice entier i, j
j = 0 ;                                   // On initialise le résultat j
for i = 1 to N do
|  if ((j = 0) ∧ (T[i] ≡ 0 mod 2)) then
|  |  j = i        // On actualise j si on n'a pas encore trouvé de nombre pair
```

Algorithme 30: Indice du premier nombre pair d'un tableau (algorithme amélioté)

PREM-PAIR(tableau T, entier N)
Données : Un tableau T de N éléments comparables
Résultat : L"indice du premier nombre pair dans T, et 0 si T ne contient aucun nombre pair
Indice entier i, j
$j = 0$ `// On initialise le résultat` j
$i = 1$
while $((j == 0) \wedge (i \leq N))$ **do**
  if $(T[i] \equiv 0 \mod 2)$ **then**
    $j = i$ `// On actualise` j `dès qu'on trouve un nombre pair`
  $i = i + 1$

Algorithme 31: Détermine si le tableau T est trié

TABLEAU-TRIE(tableau T, booleén b)
Données : Un tableau T de $N \geq 2$ éléments comparables
Résultat : $b = 1$ si T est trié, et $b = 0$ sinon
Indice entier i
$i = 1$ `// On initialise l'indice` i
$b = 1$ `// On initialise le résultat` b
while $((b == 1) \wedge (i \leq N))$ **do**
  if $(T[i] < T[i-1])$ **then**
    $b = 0$
  $i = i + 1$

Algorithme 32: Détermine s'il existe un élément du tableau qui soit la moyenne entre un de ses prédécesseurs et un de ses successeurs

TABLEAU-MOYENNE(tableau T, booleén b)
Données : Un tableau T de $N \geq 3$ éléments
Résultat : $b = 1$ s'il existe des indices $j < i < k$ tels que $T[i] = \dfrac{T[j] + T[k]}{2}$, et $b = 0$ sinon
Indice entier i, j, k
$b = 0$ `// On initialise le résultat` b
$i = 2$
while $((b == 0) \wedge (i < N))$ **do**
  $j = 1$
  while $((b == 0) \wedge (j < i))$ **do**
    $k = i + 1$
    while $((b == 0) \wedge (k \leq N))$ **do**
      if $T[i] == \dfrac{T[j] + T[k]}{2}$ **then**
        $b = 1$
      $k = k + 1$
    $j = j + 1$
  $i = i + 1$