

Chapitre 1

Le tri

1.1 Le tri

Au chapitre précédent, nous avons vu qu'une règle, pour accéder rapidement à des données, est de les représenter dans une structure, tableau, liste ou arbre, ordonnée. Cela généralise l'idée qu'il vaut mieux ordonner les mots dans un dictionnaire, les noms dans un annuaire ou les livres dans une bibliothèque, pour les retrouver rapidement. Un problème lié à la recherche en table est donc celui de la construction de structures ordonnées : le *tri*.

Une relation \leq est une relation de *pré-ordre* si elle est réflexive et transitive

$$\begin{array}{c} x \leq x \\ \text{si } x \leq y \text{ et } y \leq z \text{ alors } x \leq z \end{array}$$

une relation de pré-ordre est une relation d'*ordre* si elle est antisymétrique

$$\text{si } x \leq y \text{ et } y \leq x \text{ alors } x = y$$

et elle est *totale* si deux éléments sont toujours comparables

$$x \leq y \text{ ou } y \leq x$$

Un exemple de relation de pré-ordre qui ne soit pas un ordre est la relation définie sur les couples d'entiers par $(x, y) \leq (x', y')$ si $x \leq x'$. Dans ce cas, le couple $(1, 2)$ est inférieur au couple $(1, 3)$ et vice-versa sans que ces deux couples soient égaux.

Un algorithme de *tri* est un algorithme qui s'applique à un tableau contenant des données sur lesquelles existe une relation de pré-ordre total. Le résultat de l'algorithme est un tableau — le même ou un autre — qui contient les mêmes éléments que le tableau initial, avec la même multiplicité, et qui est ordonné, c'est-à-dire tel que $t[0] \leq t[1] \leq \dots \leq t[n - 1]$.

1.2 Le tri par sélection et le tri par insertion

L'un des algorithmes de tri les plus simples, l'algorithme de *tri par sélection*, consiste à chercher un élément minimal dans le tableau et à le permuter avec la première case. On cherche ensuite un élément minimal parmi ceux qui restent, et on le permute avec la deuxième case, et ainsi de suite jusqu'à la fin du tableau.

```
static void select (final int [] t, final int n) {
  for (int i = 0; i < n; i = i + 1) {
    int min = i;
    for (int j = i + 1; j < n; j = j + 1) {if (t[j] <= t[min]) min = j;}
    int z = t[i]; t[i] = t[min]; t[min] = z;}
}
```

Dans cet algorithme on construit petit à petit le tableau trié.

Une alternative consiste à trier petit à petit le tableau initial. C'est le *tri par insertion*. Quand les k premiers éléments du tableau sont triés, on insère le $(k + 1)$ -ème à sa place, en décalant ceux des éléments déjà triés qui lui sont supérieurs, et on obtient un tableau dont les $k + 1$ premiers éléments sont triés.

Exercice 1.1 *Programmer le tri par insertion en Java.*

Le tri par sélection comporte deux boucles imbriquées dont la première fait varier un indice i de 0 à $n - 1$ et le second de $i + 1$ à $n - 1$. Le corps de cette boucle est exécuté $(n - 1) + (n - 2) + \dots + 1 + 0$ fois, soit $(n^2 - n) / 2$ fois. Quand n tend vers l'infini, cette fonction est équivalente à $n^2 / 2$, et donc à n^2 à une constante près. On considère que l'accès à l'élément d'un tableau se fait en temps constant, et donc que le corps de la boucle s'exécute en temps constant. Le temps nécessaire pour trier un tableau de taille n avec cet algorithme est donc équivalent à n^2 à une constante près. Cet algorithme est quadratique. On peut démontrer qu'il en est de même pour le tri par insertion.

Exercice 1.2 (Le tri par remontée de bulles) *Le tri par remontée de bulles consiste à parcourir le tableau et à permuter un élément avec son successeur chaque fois qu'ils ne sont pas dans le bon ordre. Quand on a fini le parcours du tableau, on recommence. Montrer que n parcours du tableau suffisent à le trier. Donner un exemple de tableau qui demande n parcours pour être trié. Montrer que cet algorithme est quadratique.*

Programmer le tri par remontée de bulles en Java.

1.3 Diviser pour régner

On peut trouver des algorithmes beaucoup plus efficaces pour trier un tableau, mais, pour cela, il faut abandonner l'idée de récurrence, qui consiste à trier un tableau de taille n en se ramenant à un tableau de taille $n - 1$, pour se ramener plutôt à deux tableaux de taille $n / 2$. Cette variante de la récurrence s'appelle *diviser pour régner*.

Comme au paragraphe précédent, selon que l'on divise le tableau trié ou le tableau à trier, on obtient deux algorithmes différents : le *tri rapide* et le *tri fusion*.

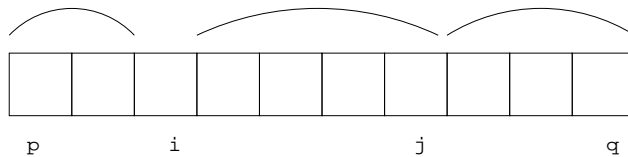
1.3.1 Le tri rapide

Si l'on divise le tableau trié, on obtient un algorithme appelé le *tri rapide*. Le premier élément du tableau à trier s'appelle le *pivot*. On range tous les éléments du tableau inférieurs au pivot à gauche du tableau, tous les éléments supérieurs au pivot à droite et le pivot au milieu. Ces deux parties ont une taille $(n - 1) / 2$ en moyenne. Il ne reste plus qu'à les trier l'une et l'autre.

```
static void rapide (final int [] t, final int p, final int q) {
    if (p < q) {
        int i = p;
        int j = q;
        while (i < j) {
            if (t[i+1] <= t[i]) {int z = t[i];t[i] = t[i+1];t[i+1] = z;i = i+1;}
            else {int z = t[j];t[j] = t[i+1];t[i+1] = z;j = j-1;}
            rapide(t,p,i-1);
            rapide(t,i+1,q);}
}
```

La fonction `rapide` trie la partie du tableau située entre l'indice `p` et l'indice `q` inclus. Tout d'abord, si l'intervalle à trier est vide, on n'a rien à faire. Si ce n'est pas le cas, il y a au moins une case `t[p]` dans le tableau, c'est le pivot. On classe les éléments situés entre `p + 1` et `q` en mettant les éléments inférieurs au pivot à gauche et les éléments supérieurs au pivot à droite. En régime permanent

- les éléments `p, ..., i - 1` sont inférieurs au pivot,
- l'élément `i` est le pivot,
- les éléments `i + 1, ..., j` restent à classer,
- les éléments `j + 1, ..., q` sont supérieurs au pivot



Si l'élément `i + 1` est inférieur au pivot, on le permute avec le pivot, et on incrémente `i`, s'il est supérieur, on le permute avec l'élément `t[j]` et on décrémente `j`. Ainsi, quand on sort de la boucle, `i` est égal à `j`, la case `i` contient le pivot, les cases `p, ..., i - 1` les éléments inférieurs au pivot, et les cases `i + 1, ..., q` les éléments supérieurs. On les trie en appelant récursivement la fonction `rapide`.

Pour trier n éléments, avec cet algorithme, on doit d'abord exécuter une boucle qui demande un temps majoré par $a(n + 1)$ pour une certaine constante a . En considérant que le temps nécessaire pour exécuter la boucle est égal à $a(n + 1)$, on obtiendra un majorant de la complexité. Ensuite on doit trier k

et $n - 1 - k$ éléments, où k est un nombre compris entre 0 et $n - 1$, toutes ces valeurs recevant la même pondération quand on fait la moyenne. Toutes les valeurs de k ayant la même pondération, le temps moyen nécessaire pour trier une liste est

$$C_n = a (n + 1) + \frac{1}{n} \sum_{k=0}^{n-1} (C_k + C_{n-1-k})$$

$$C_n = a (n + 1) + \frac{2}{n} \sum_{k=0}^{n-1} C_k$$

Quand on calcule $(n + 1) C_{n+1} - n C_n$, la sommation disparaît, et on obtient la relation de récurrence

$$(n + 1) C_{n+1} - n C_n = 2 a (n + 1) + 2 C_n$$

d'où on tire

$$C_n = (n + 1)(C_0 + 2 a \sum_{k=2}^{n+1} 1/k)$$

Comme la série $\sum_{k=1}^n 1/k$ est équivalente à $\ln(n)$ on obtient que C_n est asymptotiquement équivalent à $2 a n \ln(n)$.

Rappelons que cette valeur est une majoration du temps moyen nécessaire pour trier un tableau. Le temps moyen nécessaire pour trier un tableau est donc asymptotiquement majoré par $n \ln(n)$ à une constante près.

Cependant, ce résultat n'est vrai qu'en moyenne, il y a des cas dans lesquels le tri rapide est quadratique. C'est en particulier le cas si le tableau à trier est déjà ordonné. Dans ce cas, la partition des $n - 1$ éléments en deux sous-ensembles est la pire que l'on puisse imaginer : l'une des parties contient les $n - 1$ éléments et l'autre est vide.

1.3.2 Le tri fusion

Si au lieu de diviser le tableau trié, on divise le tableau à trier, on obtient un autre algorithme, le *tri fusion* qui est en $n \ln(n)$ dans tous les cas, mais qui ne semble pas pouvoir se programmer *en place*, c'est-à-dire sans utiliser un tableau auxiliaire. Cet algorithme utilise donc deux fois plus de mémoire que le tri rapide.

Dans le tri fusion, on divise le tableau en deux parties égales, que l'on trie séparément. On obtient alors deux tableaux triés qu'il faut fusionner. Il suffit pour cela de comparer successivement les plus petits éléments des deux tableaux et de sélectionner le plus petit des deux.

```
static void fusion (final int [] t, final int [] aux,
                  final int p, final int q) {
    if (p < q) {
        int k = (p + q) / 2;
        fusion (t,aux,p,k);
```

```

fusion (t,aux,k+1,q);

int r = p;
int s = k + 1;
for (int m = p; m <= q; m = m + 1) {
  if (r > k) {aux[m] = t[s];s = s + 1;}
  else if (s > q) {aux[m] = t[r];r = r + 1;}
  else if (t[r] <= t[s]) {aux[m] = t[r]; r = r + 1;}
  else {aux[m] = t[s]; s = s + 1;}}
for (int m = p; m <= q; m = m + 1) t[m] = aux[m];}}

```

Ici le temps nécessaire pour trier un tableau de taille n vérifie la relation de récurrence

$$C_n \leq a(n + 1) + 2 C_{n/2}$$

en supposant que n est une puissance de 2 pour éviter les questions d'arrondi.

On montre, par récurrence sur n , que cette suite est inférieure à $2 a n \log_2(n)$. En effet, supposons que c'est vrai pour $n / 2$. On a

$$C_n \leq a(n + 1) + 4 a(n / 2) \log_2(n / 2)$$

$$C_n \leq 2 a n \log_2(n) - a(n - 1)$$

et donc

$$C_n \leq 2 a n \log_2(n)$$

Le temps nécessaire pour trier un tableau de n éléments est donc asymptotiquement majoré par $n \ln(n)$ à une constante près.

1.4 Insérer en temps logarithmique

Une autre idée pour trier un tableau en temps $n \ln(n)$ est de construire une structure ordonnée en insérant successivement les n éléments dans une structure vide au départ. Comme on peut insérer un élément en temps logarithmique, dans un arbre équilibré, insérer n éléments demande un temps asymptotiquement majoré par $n \ln(n)$ à une constante près. Dans une seconde phase de l'algorithme, on recherche et supprime n fois l'élément maximal de cette structure. Encore une fois, cela demande un temps asymptotiquement majoré par $n \ln(n)$ à une constante près. Et on obtient ainsi tous les éléments du tableau dans l'ordre décroissant.

Ce type d'algorithme est particulièrement utilisé avec des tas, car comme nous l'avons vu, les tas peuvent se représenter par des tableaux. L'algorithme obtenu s'appelle le *tri en tas*. En utilisant les fonctions `insert` et `suppress` de la Section ??, le tri en tas se programme ainsi

```
static void tas (final int [] t, final int n) {
    for (int i = 0; i < n; i = i + 1) {insert(t,i);}
    for (int i = n-1; i >=0; i = i - 1) {
        int a = t[0];suppress(t,i);t[i] = a;}}
```

Pendant la première phase de l'algorithme, la partie gauche du tableau contient un tas et la partie droite les éléments non encore insérés. Pendant la seconde, la partie gauche contient un tas, et la partie droite les éléments déjà supprimés. Ainsi, contrairement au tri fusion, le tri en tas ne demande pas d'utiliser de tableau auxiliaire : c'est un tri en place.

Le tri en tas demande donc un temps asymptotiquement majoré par $n \ln(n)$ à une constante près — contrairement aux tris par insertion et par sélection — dans tous les cas et non uniquement en moyenne — contrairement au tri rapide — et il est en place — contrairement au tri fusion.

1.5 La borne inférieure

Peut-on faire mieux que $n \ln(n)$ et trouver un jour un algorithme plus rapide, par exemple un algorithme linéaire dans tous les cas ? On peut démontrer que c'est impossible si l'on fait les hypothèses suivantes

- le tableau initial peut être n'importe laquelle des permutations du tableau trié,
- la seule opération autorisée sur les éléments à trier est la comparaison pour la relation de pré-ordre.

En effet, un tel algorithme devrait fonctionner pour toutes les relations de pré-ordre et donc en particulier pour toutes les relations d'ordre. S'il est linéaire en la taille du tableau, il effectue un nombre de comparaisons inférieur à $a \cdot n$ où n est la taille du tableau et a une constante. Le résultat de chaque comparaison est un booléen et donc le nombre de suites de résultats possibles pour ces comparaisons est inférieur à 2^{an} . Or, pour des valeurs suffisamment grandes de n , $n! \geq 2^{an}$ et donc, il y a au moins deux permutations différentes pour lesquelles cette suite de résultats est identique. L'algorithme ne peut pas distinguer ces deux permutations et exécute les mêmes instructions pour l'une et pour l'autre. Et si la relation de pré-ordre est une relation d'ordre, dans au moins un de ces deux cas, le résultat n'est pas un tableau trié.

Plus généralement, pour pouvoir exécuter des instructions différentes pour chacune des $n!$ permutations, le programme doit effectuer au moins $\log_2(n!)$ comparaisons. Or

$$\log_2(n!) = \sum_{k=1}^n \log_2(k)$$

Parmi ces n termes, $(99 / 100) \cdot n$ sont supérieurs à $\log_2(n / 100) = \log_2(n) - \log_2(100)$ et donc

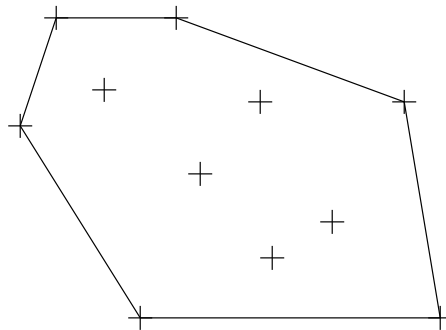
$$\log_2(n!) \geq (99 / 100) \cdot n (\log_2(n) - \log_2(100))$$

La fonction $\log_2(n!)$ est donc asymptotiquement minorée par $(99 / 100) n \log_2(n)$, soit $n \ln(n)$ à une constante près.

Le nombre de comparaisons, et *a fortiori* d'opérations, effectuées par un algorithme de tri est donc asymptotiquement minoré par $n \ln(n)$ à une constante près. Le tri fusion et le tri en tas sont donc optimaux à une constante près.

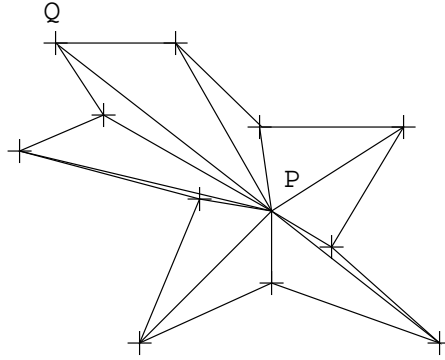
Les deux hypothèses ci-dessus sont nécessaires. On peut concevoir un algorithme de tri plus rapide si l'on dispose d'informations sur le tableau initial. Par exemple si on sait qu'il est ordonné, on peut le trier en temps nul. La seconde hypothèse est également nécessaire : si on sait que les données à trier sont les coordonnées de n points dont les abscisses sont $0, \dots, n - 1$, on peut les trier par abscisse croissante en temps linéaire, même si les $n!$ permutations sont possibles.

Exercice 1.3 (L'enveloppe convexe) *Les algorithmes de tri sont utilisés dans de nombreux domaines, en particulier en géométrie algorithmique où ils servent par exemple au calcul de l'enveloppe convexe d'un ensemble fini de points.*



Pour calculer cette enveloppe convexe, on commence par choisir un pôle P , qui est dans l'enveloppe convexe, mais distinct de chacun des points. Une manière de choisir ce point est de choisir un point A dans l'ensemble et de prendre le milieu du segment AB où B est un point de l'ensemble de distance minimale à A . On doit ensuite choisir un point Q de l'ensemble qui fait partie de la frontière de l'enveloppe convexe. On peut par exemple prendre un point de distance maximale au pôle.

Puis on calcule les coordonnées polaires de chacun des points en prenant P comme pôle et le vecteur PQ comme origine des angles. On trie ensuite les points par angle polaire croissant. On obtient un tableau τ , dont le point Q est nécessairement le premier élément. On ajoute le point Q une seconde fois à la fin du tableau de manière à fermer la boucle.



Il ne reste plus qu'à parcourir le tableau pour supprimer les points d'angle négatif. On obtient ainsi un polygone convexe qui est la frontière de l'enveloppe convexe de l'ensemble initial.

Écrire un programme en Java qui dessine l'enveloppe convexe d'un ensemble quelconque de points.

Quelle est la complexité de ce programme ?

