TD 1 – Algorithmes de tri

D'après certaines estimations, environ un quart des ressources des microprocesseurs sont occupées à trier des données : tri de fichiers clients, de tickets de transactions financières, tri de facettes pour l'affichage 3D... Si la puissance des ordinateurs augmente régulièrement, les quantités de données traitées augmentent beaucoup plus vite, ce qui fait qu'on a besoin d'algorithmes toujours plus performants.

Le but de ce TD est de présenter quelques méthodes simples de tri de tableaux d'entiers (le tri de données d'autres types suivant en général des principes rigoureusement similaires avec une autre fonction de comparaison), souvent utilisées pour trier un petit nombre de données, et d'en analyser le fonctionnement et les performances.

En ce qui concerne justement l'analyse des performances, on comptera essentiellement le nombre de comparaisons entre éléments, et le nombre de lectures/écritures en mémoire. Ces nombres dépendant souvent de l'état du tableau à trier, on essayera de distinguer le meilleur et le pire cas possibles.

1) Tri d'un petit nombre d'éléments

Lorsqu'on veut trier un très petit nombre d'éléments, il est plus simple d'écrire directement les comparaisons à effectuer que de se lancer dans la conception d'un programme compliqué. On peut pour cela s'aider d'un *arbre de décision*.

a) Écrire une procédure `tri2` qui prend en argument un tableau de type `tableau2 = array[1..2] of integer`, et qui trie ce tableau.

b) Écrire les procédures `tri3`, `tri4` et `tri5` triant des tableaux de respectivement 3, 4 et 5 éléments.

c) Pourquoi semble-t-il vain d'aller beaucoup plus loin sur cette voir ?

2) Tri par insertion

L'idée de cet algorithme est de trier les éléments du tableau un par un, en insérant chaque élément à la bonne place dans la partie du tableau déjà triée. C'est ainsi que procèdent certains joueurs de bridge pour ranger leurs cartes dans l'ordre.

Voici un algorithme mettant en œuvre cette idée :

```
procedure insertion(t : tableau[1..N] d'entiers);

  // Avant l'entrée dans le corps de boucle, t[1..i-1] est trié.
  // Après le corps de boucle, t[i] a été inséré à la bonne place
  // dans le tableau t[1..i-1], et t[1..i] est trié.
  pour i de 2 à N faire
    j <- i-1
    // on fait remonter t[i] à sa place
    tant_que j>0 et t[j+1] < t[j] faire
      echanger t[j] et t[j+1]
    fin_tant_que
  fin_pour
```

a) Implémenter cet algorithme en Pascal (attention au fait que pour que le tableau soit effectivement trié, il faut le transmettre *par adresse* et non *par valeur*).

b) Le test de la boucle `tant_que` est dangereux : que se passe-t-il si `j=0` ?

Améliorer cet algorithme à l'aide d'une *sentinelle* de manière à ce que le bon fonctionnement de la procédure ne dépende pas du bon vouloir du compilateur !

c) Combien de comparaisons sont nécessaire dans le pire cas ? Combien d'échanges ? Et au fait, quel(s) tableaux correspondent à ce pire cas ?

d) Que se passe-t-il dans le meilleur cas ? Quel(s) tableaux correspondent à ce meilleur cas ?

e) En fait, on peut singulièrement améliorer cet algorithme en analysant ce qui se passe lors de la phase d'insertion de `t[i]` : on répète l'opération

```
temp <- t[j+1]; t[j+1] <- t[j]; t[j] <- temp
```

jusqu'à ce que l'élément à insérer soit à la bonne position. Or on constate que `temp` reçoit toujours la même valeur ! Il suffirait donc de sauvegarder une fois pour toutes cette valeur, et de n'effectuer que des décalages en profitant de la place libre ainsi libérée.

Voici un exemple d'exécution montrant comment le petit tableau `[55; 41; 59; 26]` est trié (`temp` montre la valeur de la variable `temp`, `comp` compte le nombre de comparaisons entre éléments du tableau, `aff` le nombre d'affectations) :

```
                    temp  comp  aff
55 41 59 26  |
55 ## 59 26  |  41          1
## 55 59 26  |  41     1    2
41 55 59 26  |              3
41 55 ## 26  |  59          4
41 55 59 26  |         2    5
41 55 59 ##  |  26          6
41 55 ## 59  |  26     3    7
## 41 55 59  |  26     4    8
26 41 55 59  |              9
```

Réécrire l'algorithme de tri par insertion, l'implémenter et reprendre l'étude de l'efficacité dans le meilleur et dans le pire cas.

f) Écrire un programme illustrant cette méthode de tri, en montrant les évolutions successives du tableau à trier comme ci-dessus. On pourra supposer que les valeurs du tableau à trier sont comprises entre 10 et 99.

Observer le travail effectué sur quelques exemples bien choisis : un tableau trié en ordre croissant, un autre trié en ordre décroissant, et quelques tableaux obtenus par mélange aléatoire.

3) **Tri par sélection**

Une autre méthode simple de tri est le *tri par sélection* : on cherche le plus petit élément du tableau `t[1..N]`, et on l'échange avec `t[1]`. Puis on recommence en recherchant le plus petit élément `t[2..N]`, qu'on échange avec `t[2]`, et ainsi de suite.

Voici une version de cet algorithme :

```
procedure selection(t : tableau[1..N] d'entiers);

  // Avant l'entrée dans le corps de boucle, t[1..i-1] contient les
  // i-1 plus petits éléments du tableau t, rangés dans l'ordre croissant.
  // Après le corps de boucle, le plus petit élément de t[i..N] a
  // été échangé avec t[i].
  pour i de 1 à N faire
    // On recherche le plus petit élément de t[i..N]
    min <- i;
    pour j de i+1 à N faire
      si t[j] < t[min] alors min <- j fin_si
    fin_pour
    // on l'échange avec t[i]
    echanger t[min] et t[i]
  fin_pour
```

a) Implémenter cet algorithme sous forme d'une procédure `selection`.

b) Analyser le meilleur cas, le pire cas, en termes de nombre de comparaisons, et de nombre d'échanges.

c) En quoi cet algorithme peut-il être préférable au tri par insertion ?

4) **Tri à bulle**

Le tri à bulle est plus une curiosité amusante qu'une méthode de tri effective. En effet, il est peu efficace, même pour un algorithme élémentaire.

Voici l'algorithme définissant ce tri :

```
procedure bulle(t : tableau[1..N] d'entiers);

  répéter
    temp <- t[1]
    pour j de 2 à N faire
      si t[j-1] > t[j] alors
        temp <- a[j-1]; a[j-1] <- a[j]; q[j] <- temp
      fin_si
    fin_pour
  jusqu'à (t = a[1]);
```

a) Expliquer le fonctionnement de cet algorithme, en montrant son action sur l'entrée

`[57; 43; 59; 26; 53; 58; 97; 93]`

b) Au regard de l'analyse précédente, voyez-vous un moyen d'améliorer un peu la boucle interne ?

c) L'implémenter en Pascal, et construire un programme permettant d'en visualiser le fonctionnement.

d) Compter le nombre de comparaisons et d'échanges dans le meilleur et le pire cas.
Cet algorithme vous semble-t-il efficace ?

5) **Tri par distribution**

Voici un autre algorithme, appelé *tri par distribution*, qui trie une série de données dont les valeurs sont comprises entre 0 et `M-1`, en utilisant deux tableaux auxiliaires :

```
procedure distribution(t : tableau[1..N] d'entiers);

  // variables temporaires :
  //   compte : tableau[0..M-1] d'entiers
  //   tabaux : tableau[1..N] d'entiers

  pour j de 0 à M-1 faire compte[j] <- 0 fin_pour
  pour i de 1 à N faire
    compte[t[i]] <- compte[t[i]] + 1
  fin_pour
  pour j de 0 à M-1 faire
    compte[j] <- compte[j-1] + compte[j]
  fin_pour
  pour i de N à 1 faire
    tabaux[compte[t[i]]] <- t[i]
    compte[t[i]] <- compte[t[i]] - 1
  fin_pour
  pour i de 1 à N faire t[i] <- tabaux[i] fin_pour
```

Expliquer le fonctionnement de cet algorithme qui a l'air bien compliqué ! Quel est son avantage principal ? Et son plus gros défaut ?

6) **D'autres tris**

L'Internet regorge de pages recensant d'autres méthodes de tri plus ou moins compliquées (et plus ou moins adaptées aux situations particulières). N'hésitez pas à en lire la description et à essayer de les implémenter.

Nous en verrons deux plus efficaces après le cours sur la récursivité : le *tri rapide* et le *tri fusion*.