Chapitre 13

Comparaison d'algorithmes et tris

Cette séance de révision doit vous permettre de mettre en application toutes les connaissances acquises lors des séances passées. Afin de progresser dans cette séance vous devrez faire preuve de rigueur dans le développement du code demandé.

Le tri est une opération qui vise à réordonner les éléments d'un tableau selon un ordre défini à l'avance. Dans cette séance nous chercherons à ordonner les éléments d'un tableau d'entiers selon l'ordre croissant.

Exemple :
Un tableau non-trié :

7	3	9	5	1

Le *même* tableau (*ie* contenant les *même* valeurs) trié :

1	3	5	7	9

Pour faire cela, de nombreux algorithmes ont été proposés. Ils ne sont pas tous équivalents en terme d'efficacité et peuvent notamment demander plus ou moins d'opérations élémentaires et donc être plus ou moins longs à exécuter.

Nous chercherons lors de cette séance à comparer différents algorithmes classiques par des calculs empiriques de complexité. Nous étudierons notamment comment évolue le temps de calcul ainsi que le nombre d'opérations nécessaires pour ordonner le tableau en fonction de la taille du tableau.

Nous nous attendons à ce que le nombre d'opérations comme le temps de calcul augmente lorsque la taille du tableau (et donc le nombre d'éléments à trier) augmente. Cependant, nous mettrons en évidence que le nombre d'opérations et le temps de calcul augmentent plus ou moins vite selon les algorithmes.

Les algorithmes qui seront testés lors de ce TP sont tous des algorithme de *tri en place*, c'est-à-dire qu'ils n'utilisent pas de tableau auxiliaire. Ces algorithmes sont :

— le tri rapide (quicksort),

— le tri à bulle,

— le tri par sélection.

Afin de vous guider dans le développement de ces tests nous mettons à votre disposition un premier algorithme, celui du tri rapide. Celui-ci est disponible dans le fichier suivant :

`/home/TP/tpinfo/m1102_2016/tri.cpp`

Exercice 1 : Suivre les spécifications - Exécution de `tri_rapide`

Après avoir copié dans votre répertoire courant le fichier `tri.cpp`, compléter celui-ci en suivant scrupuleusement les instructions données dans le fichier.

Notamment, après avoir complété les fonctions manquantes l'affichage affichage produit doit être IDENTIQUE à celui qui est indiqué dans le fichier, soit :

```
| 4 | 2 | 3 | 1 |
| 1 | 2 | 3 | 4 |
Nombre d'echanges = 5
```

Exercice 2 : Génération d'un tableau de valeurs aléatoires

Afin de pouvoir tester nos algorithmes de tri nous souhaitons disposer de tableaux contenant un nombre variable d'entiers aléatoires à trier (la taille maximale des tableaux sera fixée par une constante `CAPACITE_MAX` que vous pourrez initialement fixer à 10^6).

Définir une fonction `init_tab_aleat()` qui prend en paramètre un tableau d'entiers (et son nombre d'éléments) et qui initialise le tableau avec des nombres aléatoires compris entre 0 et 100.

Tester la fonction `init_tab_aleat` et afficher des tableaux de taille 10, 100 et 1 000, avant et après tri par la fonction `tri_rapide()`.

Remarque : : La fonction `init_tab_aleat()` a déjà été développée dans un précédent exercice (Exercice 7, TP 9).

Exercice 3 : Nombre d'échanges lors du tri par `tri_rapide`

Proposer un programme appelant l'algorithme `tri_rapide` 5 fois de suite sur le *même* tableau de taille 10 000. Comment se comporte le nombre d'échanges ? Est-il identique ? Pourquoi ?

Remarque : : On entend par même tableau, un tableau de même taille, comportant les mêmes éléments placés dans le même ordre.

Exercice 4 : Nombre moyen d'échanges de l'algorithme `tri_rapide`

Proposer un programme principal permettant d'afficher la complexité moyenne (nombre moyen d'échanges sur le tableau durant l'exécution) du tri d'un tableau de taille 10 000. La moyenne sera calculée à partir du tri de 100 tableaux (distincts) tirés aléatoirement.

Exercice 5 : Nombre moyen d'échanges de `tri_rapide` en fonction de la taille du tableau

On souhaite évaluer comment varie le nombre d'échanges lors du tri rapide, quand la taille des tableaux augmente.

Modifier le programme principal pour qu'il permette d'évaluer le nombre moyen d'échanges pour des tableaux de taille 10^n où $n \in [1, 5]$.

Ce programme produira un affichage ne comportant que 5 lignes, chaque ligne donnant un couple de valeurs `{taille, nb_echanges_moy}` avec :

- `taille` est la taille du tableau à trier,
- `nb_echanges_moy` est le nombre d'échanges lors du tri d'un tableau de taille n.

Exercice 6 : Comparaison du temps moyen d'exécution et du nombre moyen d'opérations

Afin d'évaluer la complexité temporelle, on veut mesurer et afficher le temps d'exécution de notre algorithme de tri. Pour cela on met en place un système de chronométrage de la durée d'exécution.

La librairie `<time.h>` propose la fonction `clock()` dont une partie de la documentation est traduite et reproduite ci-dessous. La documentation de cette fonction et celle de la libr sont disponibles airie dans leur intégralité aux adresses suivantes :

- `<time.h>` [1]

1. http ://pubs.opengroup.org/onlinepubs/7908799/xsh/time.h.html

— `clock()` [2]

```
NOM
   clock - permet d'évaluer la durée CPU associée à un programme.

UTILISATION
   #include <time.h>
   clock_t clock(void);

DESCRIPTION
      La fonction clock() renvoie la meilleur approximation du temps processeur
      utilisé par un processus depuis l'invocation et le lancement de ce processus.

VALEUR DE RETOUR
      Le type de la valeur de retour est clock_t.
      Pour déterminer le temps en secondes, la valeur retournée par clock() doit
      être divisée par la valeur de la constante CLOCKS_PER_SEC définie elle aussi
      dans la librairie <time.h>.

USAGE
      Afin de mesurer le temps passé pour exécuter un programme, la fonction
      clock() doit être appelée au début et à la fin du programme et les deux
      valeurs doivent être soustraites .
      La valeur retournée pas la fonction clock() est définie pour assurer une
       compatibilité inter-systèmes qui peuvent avoir des horloges fournissant des
       résolutions différentes (la résolution d'un système peut ne pas être la
      microsecondes).

      La valeur retournée par la fonction clock() peut avoir été réinitialisée
      sur certains systèmes. Par exemple, avec une machine sur laquelle clock_t
      est codée sur 32 bits, le chronométrage est remis à zéro après 2147 secondes
      soit un peu moins de 36 minutes.
```

Proposer une explication au comportement de la fonction `clock()` décrit dans le dernier paragraphe.

En fonction de la taille du tableau à trier, proposer un programme permettant de comparer les temps moyens d'exécution aux nombres moyens d'échanges.

Par exemple, le programme pourra afficher pour chaque taille de tableau une ligne donnant 4 valeurs `{taille, nb_echanges_moy, tps_exec_moy, r}`, avec :

— `taille` est la taille du tableau à trier,

— `nb_echanges_moy` est le nombre moyen d'échanges lors du tri d'un tableau de taille n.

— `tps_exec_moy` est le temps moyen d'exécution lors du tri d'un tableau de taille n,

— `r` le rapport $r = \frac{nb\_echanges\_moy}{tps\_exec\_moy}$.

Exercice 7 : Implantation de l'algorithme du tri par sélection

On peut mener la même analyse de complexité (nombre moyen d'opérations en fonction de la taille du tableau à trier) pour d'autres algorithmes de tri et comparer les résultats.

Appliquer, dans un premier temps, l'étude décrite dans l'Exercice 6 à l'algorithme du tri par sélection et comparer ensuite les résultats à ceux obtenus avec le tri rapide.

Le tri par sélection (https://fr.wikipedia.org/wiki/Tri_par_sélection) est ainsi décrit dans Wikipedia

Sur un tableau de n éléments (numérotés de 1 à n), le principe du tri par sélection est le suivant :

2. http ://pubs.opengroup.org/onlinepubs/7908799/xsh/clock.html

— *rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 1,*

— *rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 2,*

— *continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.*

En pseudo-code, l'algorithme s'écrit ainsi :

```
procédure tri_selection(tableau t, entier n)
    pour i de 1 à n - 1
        min ← i
        pour j de i + 1 à n
            si t[j] < t[min], alors min ← j
        fin pour
        si min ≠ i, alors échanger t[i] et t[min]
    fin pour
fin procédure
```

Implanter cet algorithme comme une fonction dont l'en-tête devra être la suivante :
`void tri_par_selection( int tab [], int taille, int & compteur)`
avec :

— `tab` le tableau contenant les valeurs à trier,

— `taille` le nombre d'éléments contenus dans le tableau `tab`,

— `compteur` une variable entière passée par référence.

Utiliser la fonction `echange(...)` précédemment développée. La variable `compteur` passée en paramètre de la fonction `tri_par_selection` comptera le nombre d'échanges permettant à cet algorithme de trier les valeurs du tableau. La variable `compteur` sera donc passée comme dernier paramètre de la fonction `echange`.

Tester la fonction obtenue et comparer les résultats obtenus avec ceux obtenus pour `tri_rapide`.

Exercice 8 : Implantation de l'algorithme du tri à bulles

Appliquer, dans un premier temps, l'étude décrite dans l'Exercice 6 à l'algorithme du tri à bulle et comparer ensuite les résultats à ceux obtenus avec le tri rapide et avec le tri par sélection.

Le tri à bulle (https://fr.wikipedia.org/wiki/Tri_à_bulles) est ainsi décrit dans Wikipedia.

L'algorithme parcourt le tableau et compare les éléments adjacents. Lorsque les éléments ne sont pas dans l'ordre croissant, ils sont échangés.

— *Après un premier parcours complet du tableau, le plus grand élément est forcément en fin de tableau, à sa position définitive. En effet, aussitôt que le plus grand élément est rencontré durant le parcours, il est mal trié par rapport à tous les éléments suivants, donc échangé à chaque fois jusqu'à la fin du parcours.*

— *Après le premier parcours, le plus grand élément étant à sa position définitive, il n'a plus à être traité. Le reste du tableau est en revanche encore en désordre. Il faut donc le parcourir à nouveau, en s'arrêtant à l'avant-dernier élément.*

— *Après ce deuxième parcours, les deux plus grands éléments sont à leur position définitive. Il faut donc répéter les parcours du tableau, jusqu'à ce que les deux plus petits éléments soient placés à leur position définitive.*

Le pseudo-code suivant est repris de *Bubble Sort : An Archaeological Algorithmic Analysis qui le reprend de Knuth* :

```
tri_à_bulles(Tableau T)
   pour i allant de taille de T - 1 à 1
       pour j allant de 0 à i - 1
           si T[j+1] < T[j]
               échanger(T[j+1], T[j])
```

Une optimisation courante de ce tri consiste à l'interrompre dès qu'un parcours des éléments possiblement encore en désordre (boucle interne) est effectué sans échange. En effet, cela signifie que tout le tableau est trié. Cette optimisation nécessite une variable supplémentaire.

```
1 tri_à_bulles_optimisé(Tableau T)
2     pour i allant de taille de T - 1 à 1
3         tableau_trié :← vrai
4         pour j allant de 0 à i - 1
5             si T[j+1] < T[j]
6                 échanger(T[j+1], T[j])
7                 tableau_trié :← faux
8         si tableau_trié
9             fin tri_à_bulles_optimisé
```

Implanter cet algorithme comme une fonction dont l'en-tête devra être la suivante :
`void tri_a_bulles( int tab [], int taille, int & compteur)`
avec :

— `tab` le tableau contenant les valeurs à trier,

— `taille` le nombre d'éléments contenus dans le tableau `tab`,

— `compteur` une variable entière passée par référence.

Utiliser la fonction `echange(...)` précédemment développée. La variable `compteur` passée en paramètre de la fonction `tri_par_selection` comptera le nombre d'échanges permettant à cet algorithme de trier les valeurs du tableau. La variable `compteur` sera donc passée comme dernier paramètre de la fonction `echange`.

Tester la fonction obtenue et comparer les résultats obtenus avec ceux obtenus pour `tri_rapide` et pour `tri_par_selection`.