Saé S1.02 : comparaison d'approches algorithmiques

Quelques critères de performance

Introduction

Quand on compare deux programmes qui réalisent la même chose, il est difficile de dire lequel est le meilleur sans savoir sur quel critère on les compare. Par exemple, la recherche dichotomique est-elle "meilleure" que la recherche séquentielle ? Ça dépend sur quoi on les compare : la recherche dichotomique est plus "rapide" que la recherche séquentielle, mais moins "universelle" puisqu'elle ne peut s'appliquer que sur une collection ordonnée.

Pour pouvoir comparer plusieurs algorithmes entre eux, il faut donc mettre en place des critères ou indicateurs de performance. La plupart du temps ces critères concernent soit le temps de calcul, soit la quantité de ressources utilisées (la quantité de mémoire nécessaire par exemple).

Nous allons aujourd'hui nous intéresser à des indicateurs concernant le temps de calcul pour différents programmes de tri. Vous devrez plus tard appliquer les principes abordés dans cette séance à votre projet de Saé 2.

Nous partirons du principe que pour les programmes de tri, ce qui "prend du temps" ce sont les comparaisons entre deux éléments du tableau et les permutations de deux éléments. Un programme qui fera moins de comparaisons et/ou moins de permutations qu'un autre sera sans doute plus rapide. Nous allons donc compter le nombre de permutations et de comparaisons effectuées lors de l'exécution.

Enfin nous verrons comment mesurer la durée d'exécution d'un programme.

1 – Quelques programmes de tri

Un algorithme de tri est chargé d'ordonner une collection d'objets selon une relation d'ordre. Nous utiliserons ici des tableaux d'entiers. Un tri est dit **en place** s'il ne nécessite aucune structure de données secondaire pour opérer. Un tri est dit **stable** s'il conserve la place relative de deux élements identiques.

1.1 – Tri par insertion

Pour le **tri par insertion**, on considère qu'à l'étape `i`, la partie `[0, i-1]` est déjà triée. L'étape `i` consiste alors à placer `t[i]` à sa bonne place dans l'intervalle `[0,i]` : on décale vers la droite tout élément supérieur à t[i] de manière à créer un "trou" pour l'insertion.

Algorithme :

```
procédure tri_insertion(tableau T)
début
       pour i de 1 à taille-1 faire
            // x représente l'élément à placer au bon endroit
            x := T[i];

            // décaler les éléments T[0]..T[i-1] qui sont plus grands
            // que x, en partant de T[i-1]
            j := i;
            tant que j > 0 et T[j - 1] > x faire
                     T[j]:= T[j-1];
                     j := j-1;
            fin tant que
            // placer x dans le "trou" laissé par le décalage
            T[j] := x;
      fin pour
fin
```

Exercice 1

Question 1

Programmez en C l'algorithme de tri par insertion. Testez-le avec un tableau de 10 entiers déclaré comme ceci dans le main :

```
tableau T = {5,3,9,8,7,5,2,1,9,1};
```

À des fins de vérification, vous pouvez écrire une procédure qui affiche les éléments du tableau.

Question 2

Ajoutez à votre programme une procédure qui initialise le tableau avec des valeurs aléatoires. Changez la taille du tableau et passez-la à 300000. Testez votre tri.

1.2 – Tri rapide

Le **tri rapide** ("quick sort" en anglais), ou tri de Hoare du nom de son inventeur, est un tri en place mais non stable.
La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Algorithme :

```
fonction partition(tableau:t,entier:debut,entier:fin,entier:pivot)
délivre entier
début
    permuter t[pivot] et t[fin]
    j:= debut;
    pour i de debut à (fin-1) faire
        si t[i] <= t[fin] alors
            permuter t[i] et t[j]
            j := j+1;
        finsi
    fin pour
    permuter t[fin] et t[j]

    délivre j;
fin

procédure triRapide(tableau:t, entier:debut, entier:fin){
début
    si debut < fin alors
        pivot := (debut+fin)/2;
        pivot := partition(t, debut, fin, pivot);
        triRapide(t, debut, pivot-1);
        triRapide(t, pivot+1, fin);
    finsi
fin
```

Exercice 2

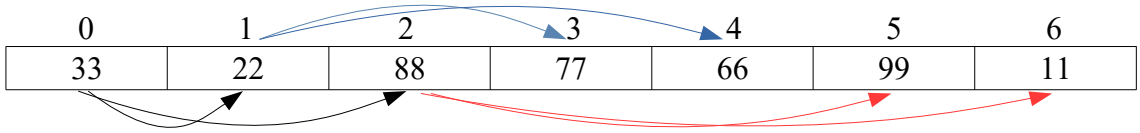
Dans un nouveau programme, implémentez en C l'algorithme de tri rapide. Testez-le d'abord sur un tableau de 10 entiers (*cf.* exercice 1) puis sur un tableau de 300000 entiers générés de manière aléatoire.

1.3 – Tri par tas

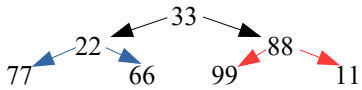
Le **tri par tas**, ("heap sort" en anglais) est un tri **en place**, mais **non stable**. Le principe général du tri par tas est donné ici juste pour information, l'algorithme est fourni page suivante.

Le tableau vu comme un arbre binaire

Dans un tableau, on peut imaginer que chaque élément d'indice `i` a ses deux fils aux indices `2*i+1` et `2*i+2`. Le premier élément du tableau représente la racine de l'arbre.



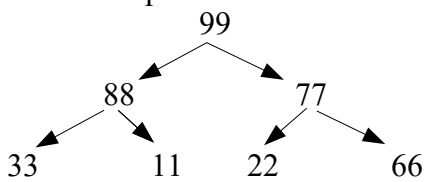
33 a pour fils 22 et 88
22 a pour fils 77 et 66
88 a pour fils 99 et 11



Définition d'un tas

Un tas est un arbre binaire équilibré tel que chaque nœud est de valeur supérieure à celles de ses deux fils. On en déduit que la valeur maximale est celle de la racine de l'arbre.

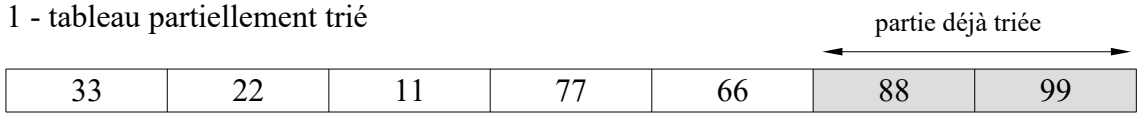
Exemple de tas (sa racine est 99 qui est aussi la valeur maximale) :



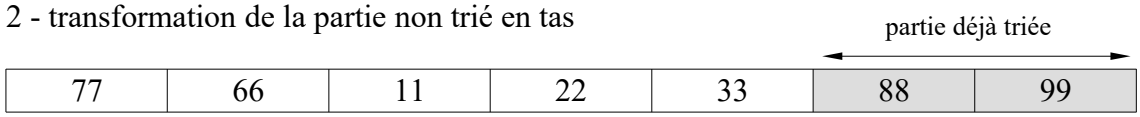
Le principe général du tri par tas est de transformer la partie non encore triée en tas (on dit "tamiser") puis d'en extraire la racine, c'est-à-dire le premier élément, pour la rejeter en fin de tableau, en fin de partie non triée.

Exemple d'une étape de tri

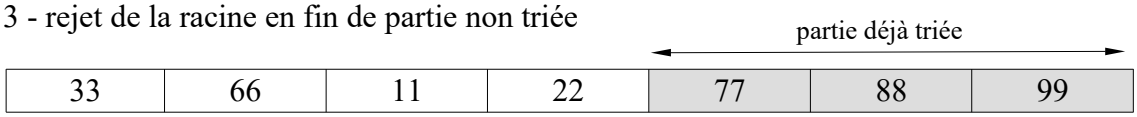
1 - tableau partiellement trié



2 - transformation de la partie non trié en tas



3 - rejet de la racine en fin de partie non triée



Algorithme (N : taille du tableau)

Voici une version récursive du tri par tas. Elle fait appel à une procédure `tamiser` qui transforme le tableau `t` en tas.

```
procédure tamiser(tableau t, entier noeud, entier n)
début
    fils := 2*noeud + 1;

    si ( fils < n  et  t[fils+1] > t[fils] )alors
        fils := fils + 1;
    finsi

    si ( fils <= n  ET  t[noeud] < t[fils] ) alors
        permuter t[noeud] et t[fils]
        tamiser(t,fils,n);
    finsi

fin


procédure triParTas(tableau t)
début
    pour i de (TAILLE/2 - 1) à 0 faire
      tamiser(t,i,TAILLE-1);
    fin pour

    pour i de TAILLE-1 à 0 faire
        permuter t[0] et t[i]
        tamiser(t, 0, i-1)
    fin pour
fin
```

Exercice 3

Dans un nouveau programme, implémentez en C l'algorithme de tri par tas. Testez-le d'abord sur un tableau de 10 entiers (*cf.* exercice 1) puis sur un tableau de 300000 entiers générés de manière aléatoire.

2 – Premiers indicateurs

Les comparaisons et les permutations d'éléments du tableau prennent du temps au cours de l'exécution (elles "coûtent cher"). Nous allons ajouter aux programmes précédents :

- une variable globale pour compter le **nombre de comparaisons** effectuées entre deux éléments du tableau,

- une variable globale pour compter le **nombre de permutations** réalisées entre deux éléments du tableau.

Exercice 4

Dans chacun de vos trois programmes, ajoutez ces deux compteurs et faîtes en sorte que leur valeur soit affichée à l'écran en fin d'exécution.

3 – Temps d'exécution

Remarque : les fonctions et types abordés dans cette section se trouvent dans `time.h`.

3.1 – Première méthode : les fonctions `time()` et `difftime()`

La fonction `time()` permet d'obtenir le temps écoulé (en général le nombre de secondes) depuis le premier janvier 1970 à 00 h 00 mn 00 s, sous forme d'un entier positif : on parle de "timestamp". Le résultat est de type `time_t`.

Deux utilisations de cette fonction sont possibles : le temps écoulé peut être récupéré soit comme résultat de la fonction, soit comme paramètre de sortie.

Exemple 1
Utilisation comme une fonction, on ne se sert pas du paramètre :

```
time_t  horaire = time(NULL);
```

Exemple 2
Utilisation d'un paramètre de sortie :

```
time_t  horaire;
time(&horaire);
```

La fonction `difftime()` calcule la différence entre deux "timestamp", c'est-à-dire le nombre de secondes entre un temps de fin et un temps de début. Le résultat est de type `double`. Bien sûr le temps de début doit être antérieur au temps de fin.

Exemple :

```
//debut et fin sont deux "timestamp"
  double duree = difftime(fin, debut);
```

Exercice 5

Question 1
Écrivez puis testez ce petit programme d'illustration (il ne fait rien d'intéressant) :

```
time_t debut = time(NULL);
for (int i=0 ; i<100000 ; i++){
      for (int j=0 ; j<100000 ; j++){
      }
}
time_t fin = time(NULL);
printf("duree = %.3f secondes\n", difftime(fin, debut) );
```

Question 2
En vous inspirant du programme précédent, faites en sorte d'afficher le temps d'exécution de votre tri par insertion (*cf.* exercice 1).

3.2 – Deuxième méthode : la fonction `clock()`

Un inconvénient de la méthode précédente est que le résultat obtenu est un nombre entier de secondes, ce qui est une précision insuffisante pour des exécutions rapides.

D'autre part, le résultat correspond au temps qui s'est écoulé entre le début et la fin du programme (ou de la partie du programme qu'on souhaite chronométrer). Mais un système multi-tâche alloue à tour de rôle un laps de temps CPU à chaque processus en cours. Le temps d'exécution mesuré précédemment ne donne donc pas le temps d'exécution réel c'est-à-dire le temps réellement consacré par le processeur à votre programme.

La fonction `clock()` retourne le nombre de "ticks" (les "tops d'horloge") consommé par l'application en cours d'éxécution. Cela correspond à sa consommation CPU. Le résultat retourné est de type `t_clock`.

Pour connaître le temps CPU utilisé par un programme il faut donc calculer le nombre de ticks consommé et le diviser par le nombre de ticks par seconde (constante CLOCKS_PER_SEC).

Exercice 6

Question 1
Écrivez puis testez ce petit programme d'illustration (il ne fait rien d'intéressant) :

```
clock_t begin = clock();
for (int i=0 ; i<100000 ; i++){
      for (int j=0 ; j<100000 ; j++){
      }
}
clock_t end = clock();
double  tmpsCPU = ((end - begin)*1.0) / CLOCKS_PER_SEC;
printf( "Temps CPU = %.3f secondes\n",tmpsCPU);
```

Question 2
Complétez ce programme pour qu'il affiche le temps d'exécution enregistré avec `time()` en plus du temps CPU enregistré avec `clock()` et comparez.

Question 3
Pour obtenir une différence encore plus flagrante, ajoutez l'instruction `sleep(3);` à votre programme. Cette instruction va allonger le temps d'exécution de 3 secondes, mais pas le temps CPU, puisque le programme va "dormir" pendant ces 3 secondes et ne consommer aucun temps processeur.

4 – Bilan

On souhaite comparer les programmes de tri au regard des trois critères : nombre de comparaisons, nombre de permutations et durée d'exécution.

Exercice 7

Question 1

Modifiez vos programmes de tri pour qu'ils affichent en fin d'exécution la durée d'exécution (temps CPU) de leur phase de tri.

Question 2

Exécutez vos programmes et complétez le fichier `Bilan.ods` disponible sur Moodle.