Travaux Pratiques n°5

L'ensemble des exercices de ce TP pourra être réalisé dans un même fichier `.c`. Tous les tableaux dans ce TP sont des tableaux statiques.

Exercice 1

Ecrire et tester une fonction `void affiche_tab(int tab[], int n)` qui affiche à l'écran les n premiers éléments du tableau d'entiers `tab`. Si le tableau est de taille `n`, la fonction affichera l'ensemble des éléments de `tab`.

Exercice 2

Ecrire et tester une fonction `void init_tab(int max, int tab[], int n)` qui initialise les n premiers éléments du tableau d'entiers `tab` à des valeurs aléatoires comprises entre `0` et `max-1`. Si le tableau est de taille `n`, la fonction initialisera l'ensemble des éléments de `tab`.

Exercice 3

Ecrire une fonction `void ech(int tab[],int i, int j)` qui échange l'élément d'indice i et l'élément d'indice j dans le tableau `tab`.

Exercice 4 (Tri par sélection)

Ecrire et tester une fonction `void tri_selec(int tab[], int n)` qui implémente l'algorithme de tri par sélection basé sur le principe suivant : On sélectionne tout d'abord l'élément le plus petit du tableau, c.à d. on trouve l'entier p tel que $\forall 0 \leq i \leq n, t[i] \geq t[p]$. Une fois cet emplacement trouvé, on échange les élements $t[0]$ et $t[p]$. Puis on recommence ces opérations pour le reste du tableau (c.à d. les éléments compris entre les indices 1 et $n-1$. On recherche alors le plus petit élément de cette nouvelle suite de nombre et on échange avec $t[1]$. Et ainsi de suite jusqu'au moment où on a placé tous les élements du tableau. `n` représente le nombre d'éléments à trier.

Exercice 5 (Tri à bulle)

Ecrire une fonction `void tri_bulle(int tab[], int n)`qui implémente l'algorithme de tri à bulle. `n` représente le nombre d'éléments à trier.

Le principe du tri à bulle est de comparer successivement tous les éléments adjacents d'un tableau (en commençant par le premier) et de les échanger si l'élément d'indice i est supérieur à l'élément d'indice $i+1$. On recommence cette opération tant que tous les éléments ne sont pas triés.

Exercice 6 (Tri par insertion)

Ecrire une fonction `void tri_insertion(int tab[], int n)`qui implémente l'algorithme de tri par insertion. `n` représente le nombre d'éléments à trier.

Le tri par insertion consiste à prendre l'élément se trouvant juste après la partie déjà triée du tableau et de trouver sa place dans cette dernière (insertion). Le premier élément à insérer est le deuxième élément du tableau, le premier étant forcément déjà trié puisqu'il est tout seul. On recommence ce procédé jusqu'au dernier élément du tableau.

Exercice 7 (Tri rapide)

Le principe du tri rapide (quicksort) consiste à choisir un élément quelconque de l'ensemble à trier, on nomme cet élément le pivot, puis à partitionner en deux sous-ensembles les éléments restants de part et d'autre du pivot. Le premier sous-ensemble contient tous les éléments inférieurs ou égaux au pivot, le second contient tous les éléments strictement supérieurs au pivot. On applique à nouveau le partitionnement des deux sous-ensembles créés et on continue ainsi récursivement jusqu'à ce que les sous-ensembles créés soient réduits à 1 seul élément. On rassemble enfin le tout dans l'ordre et c'est fini.

Ecrire la fonction C qui implémente le tri rapide. Afin de faire le partitionnement, vous allez définir deux nouveaux tableaux. Le premier est destiné à contenir les éléments supérieurs au pivot et le second les éléments inférieurs au pivot . Sachant que la taille du tableau n'est pas connue à l'avance, nous ne pouvons pas déclarer ces tableaux comme habituellement. Nous allons faire appel à deux fonctions supplémentaires *nouveau_tableau* (qui renvoie un tableau de taille t) et la fonction *detruire_tableau* (qui détruit un tableau de taille t) dont les prototypes sont les suivants :

```
int *nouveau_tableau(int t);
void detruire_tableau(int *tab);
```

Exercice 8 (Comparaison des méthodes de tri implémentées)

On va ensuite exécuter chacun des algorithmes de tri sur un tableau et comparer le temps mis par chacun des algorithmes de tri précédents.

Pour donner une idée, on prendra un tableau de taille 50000 contenant des entiers entre 0 et 20000. On pourra ensuite faire varier la taille des tableaux pour voir l'évolution du temps pris par les algorithmes en fonction de la taille.

On pourra alors tracer un graphique du temps en fonction de la taille N. Quelle courbe obtient-on ?

Pour mesurer le temps (en secondes) d'exécution d'une séquence d'instruction, on utilise la bibliothèque `time.h`. On déclare une variable `p` de type `double` et deux variables `start` et `end` de type `clock_t` :

```
double p;
clock_t start, end ;
```

Au début et à la fin de la séquence d'instruction (ici séquence permettant de trier le tableau), on consulte l'horloge de l'ordinateur :

```
start=clock();
//Début de la séquence d'instruction
(...)
```

```
//Fin de la séquence d'instruction
end=clock();
```

On calcule et affiche la différence en secondes :

```
p=(double) (end - start) / CLOCKS_PER_SEC ;
printf("Le temps d'execution est : %lf secondes\n",p);
```

Attention : Afin que l'estimation du temps de chaque tri soit significative, il convient de faire la moyenne sur plusieurs tris pour chacune des méthodes, et de trier toujours le même tableau lorsqu'on compare deux méthodes.