TP : Algorithme de tri

Objectif du TP : implémenter, tester et évaluer les performances de divers algorithmes de tri.

Documents à rendre :

- Un programme source par algorithme de tri (utilisé pour la phase de mise au point de l'algorithme : codage + tests).
Dans ce programme source, l'algorithme de tri devra être implémenté dans une **fonction** appelée dans la fonction main().
Exemple de déclaration de fonction :

```
/* Fonction permettant de trier les éléments d'un tableau d'entiers
en utilisant l'algorithme du tri à bulles :
- Paramètre Adr : adresse du 1er élément du tableau à trier
- Paramètre Nb : Nombre d'éléments du tableau
Pas de valeur de retour */
void triABulle(int * Adr, int Nb);
```

Les tests nécessiteront **l'affichage du tableau non trié** puis **l'affichage du tableau trié** après appel de la fonction de tri.

- une synthèse (document Libre Office Writer) évaluant les performances des divers algorithmes de tri.

Remarques :
- Pour initialiser les éléments des tableaux (initialement non triés), il est conseillé d'utiliser la fonction *rand()* (cf. exercice 4 du TP sur les fonctions).
- Pour fixer la taille du tableau, vous devez utiliser une directive *#define*

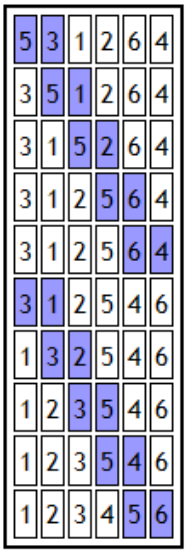
Travail à réaliser :

Exercice n°1 : **le tri à bulles**
Écrire un programme qui implémente l'algorithme de tri à bulles décrit ci-dessous.

L'algorithme du tri bulle (ou bubble sort) consiste **à regarder les différentes valeurs adjacentes** d'un tableau, et à les permuter si le premier des deux éléments est supérieur au second. L'algorithme se déroule ainsi : les deux premiers éléments du tableau sont comparés, si le premier élément est supérieur au second, une permutation est effectuée. Ensuite, sont comparés et éventuellement permutés les valeurs 2 et 3, 3 et 4 jusque (n-1) et n. Une fois cette étape achevée, il est certain que le dernier élément du tableau est le plus grand. L'algorithme reprend donc pour classer les (n-1) éléments qui précédent. L'algorithme se termine quand il n'y a plus de permutations possibles. Pour classer les n valeurs du tableau, il faut, au pire, effectuer l'algorithme (n-1) fois.

Cet algorithme porte le nom de tri bulle car, petit à petit, les plus grands éléments du tableau remontent, par le jeu des permutations, en fin de tableau. Dans un aquarium il en va de même : les plus grosses bulles remontent plus rapidement à la surface que les petites qui restent collés au fond.

Exemple : Evolution du tableau au fil de l'algorithme (en bleu, les éléments qui sont comparés, et éventuellement permutés, pour passer à la ligne suivante).



L'algorithme se termine car il n'y a plus de permutations possibles. Ce fait sera constaté grâce à un dernier parcours du tableau ou aucune permutation n'a lieu.

Exercice n°2 : **le tri par sélection**
Écrire un programme qui implémente l'algorithme de tri par sélection décrit ci-dessous.

Le tri par sélection est l'un des tris les plus instinctifs. Le principe est que pour classer n valeurs, il faut rechercher la plus petite valeur et la placer en 1ère position, puis la plus petite valeur dans les valeurs restantes et la placer en seconde position et ainsi de suite...

Considérons un tableau à n éléments. Pour effectuer le tri par sélection, il faut rechercher dans ce tableau la position du plus petit élément. Le plus petit élément est alors échangé avec le premier élément du tableau. Ensuite, on réitère l'algorithme sur le tableau constitué par les (n-p) derniers éléments où p est le nombre de fois où l'algorithme a été itéré. L'algorithme se termine quand p=(n-1), c'est à dire quand il n'y a plus qu'une valeur à sélectionner ; celle ci est alors la plus grande valeur du tableau.

Exemple : grandes étapes de l'évolution du tableau au fil de l'algorithme. En bleu, les valeurs déjà traitées.

5	3	1	2	6	4
1	3	5	2	6	4
1	2	5	3	6	4
1	2	3	5	6	4
1	2	3	4	6	5
1	2	3	4	5	6

Exercice n°3 : **le tri par insertion**

Écrire un programme qui implémente l'algorithme de tri par insertion décrit ci-dessous :

Le tri par insertion consiste à piocher une à une les valeurs du tableau et à les insérer, au bon endroit, **dans le tableau trié** constitué des valeurs précédemment piochées et triées. Les valeurs sont piochées dans l'ordre où elles apparaissent dans le tableau. Soit p l'indice de la valeur piochée, les (p-1) premières valeurs du tableau constituent le tableau trié dans lequel va être inséré la $p^{ième}$ valeur. Au début de l'algorithme, il faut considérer que la liste constituée du seul premier élément est trié : c'est vrai puisque cette liste ne comporte qu'un seul élément. Ensuite, on insère le second élément (p=2), puis le troisième (p=3) etc. Ainsi, p varie de 2 à n, où n est le nombre total d'éléments du tableau.

Le problème de cet algorithme est qu'il faut parcourir le tableau trié pour savoir à quel endroit insérer le nouvel élément, puis décaler d'une case toutes les valeurs supérieures à l'élément à insérer. En pratique, le tableau classé est parcouru de droite à gauche, c'est à dire dans l'ordre décroissant. Les éléments sont donc décalés vers la droite tant que l'élément à insérer est plus petit qu'eux.. Dés que l'élément à insérer est plus grand qu'un des éléments du tableau triée il n'y a plus de décalage, et l'élément est inséré dans la case laissée vacante par les éléments qui ont été décalés.

Exemple : grandes étapes de l'évolution du tableau au fil de l'algorithme. En bleu, le tableau trié, en rouge, la valeur de la mémoire qui contient la valeur à insérer.

Tableau						Memoire
5	3	1	2	6	4	3
3	5	1	2	6	4	1
1	3	5	2	6	4	2
1	2	3	5	6	4	6
1	2	3	5	6	4	4
1	2	3	4	5	6	

Exercice n°4 : **le tri rapide**

Écrire un programme qui implémente l'algorithme de tri rapide décrit ci-dessous :

L'algorithme de tri rapide, "quick sort" en anglais, est un algorithme de type dichotomique. Son principe consiste à séparer l'ensemble des éléments en deux parties. Pour effectuer la séparation, une **valeur pivot** est choisie. Les valeurs sont réparties en deux ensembles suivant qu'elles sont plus grandes ou plus petites que le pivot. Ensuite, les deux ensembles sont triés séparément, suivant la même méthode. L'algorithme est **récursif**. Le résultat du tri est égal au tri de l'ensemble dont les valeurs sont inférieures au pivot concaténé à l'ensemble des valeurs supérieures au pivot, ce dernier étant pris en sandwich entre les deux ensembles.

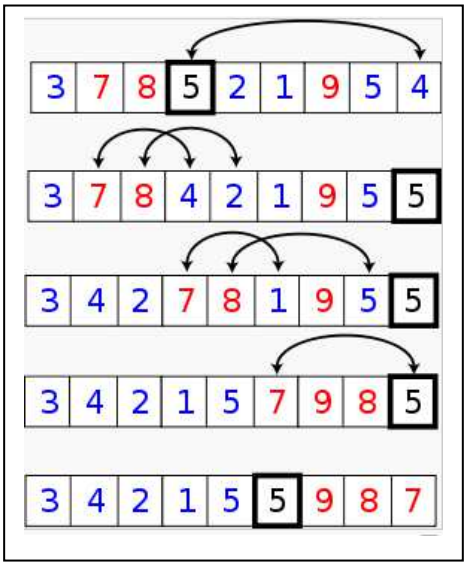
Choix du pivot : le choix du pivot est le problème central de cet algorithme. En effet, l'idéal serait de pouvoir répartir les deux ensembles en deux parties de taille à peu prés égales. Cependant, la recherche du pivot qui permettrait une partition parfaite de l'ensemble en deux parties égales aurait un coût trop important. C'est pour cela que le pivot est choisi de façon aléatoire parmi les valeurs de l'ensemble. Dans la pratique, le pivot est le premier ou le dernier élément de l'ensemble à fractionner. En moyenne, les deux ensembles seront donc de taille sensiblement égale.

Proposition d'algorithme:

```
Fonction partitionner(tableau T, entier prem, entier der, entier pivot): entier
    Variables
        entier j,i
    Début
        échanger T[pivot] et T[der]
        j ← prem
        Pour i  allant de prem à der - 1 par Pas de 1 Faire
                Si (T[i] <= T[der]) alors
                        Si (i ≠ j) alors
                                échanger T[i] et T[j]
                        Fin Si
                        j ←j + 1
                FinSi
        FinPour
        échanger T[der] et T[j]
        renvoyer j
    Fin


 Fonction tri_rapide(tableau t, entier premier, entier dernier) : vide
   Variables
        entier pivot
   Début
        Si (premier < dernier) alors
                pivot ← choix_pivot(t,premier,dernier)
                pivot ← partitionner(t,premier, dernier,pivot)
                tri_rapide(t,premier,pivot-1)
                tri_rapide(t,pivot+1,dernier)
        FinSi
   Fin
```

Exemple de partitionnement d'une petite liste de nombres :



Partie n°5 : **synthèse : évaluation des performances des algorithmes de tri implémentés**

On va ici tester des programmes déjà au point. Il faut supprimer les affichages des tableaux avant et après tri (**les mettre en commentaires**).

Pour évaluer les performances d'un programme, vous pouvez utiliser la commande **time**.

Une commande de la forme :

time *executable*

lance l'exécution de l'*executable* et fournit à la fin de son exécution le résultat de 3 mesures :

- le temps total d'horloge écoulé entre le début et la fin du processus (i.e. programme en cours d'exécution) tel qu'il pourrait être mesuré par l'utilisateur à l'aide d'un chronomètre (**real**);
- le temps passé par le processus en mode utilisateur (temps CPU en mode utilisateur), c'est à dire pour exécuter les instructions de son propre code (**user**);
- le temps passé par le processus en mode noyau (temps CPU en mode système), c'est à dire pour exécuter les instructions contenues dans le système **(sys**).

Compléter le tableau suivant avec le **temps CPU en mode utilisateur** correspondant à l'exécution d'un programme qui trie un tableau d'entiers de la **taille** spécifiée.

	Tri à bulles	Tri par sélection	Tri par insertion	Tri rapide
Taille = 1000				
Taille = 10000				
Taille = 100000				
Taille = 1000000				