Nom, Prenom:_____ **Groupe:**_____

TP3 : Tri [4 heures]

Préparation

Dans ce TP, nous allons écrire les méthodes d'une classe Sort dont les attributs suivants sont membres et par conséquent accessibles depuis chaque fonction membre de la classe :

```
int         *x;        // Pointeur vers le premier élément du tableau à trier
int         NbElements;// Nombre d'éléments dans le tableau
```

L'allocation mémoire et le remplissage du tableau sont réalisés lors de l'appel du constructeur de la classe.

a.Ecrire une méthode qui retourne l'indice du plus grand élément du tableau x compris entre les indices [Start ; End[.

```
// Retourne l'indice du plus grand élément du tableau
// Compris entre les indices Start (inclus) et End (non inclus)
int Sort::Max(int Start,int End)
```

b. Proposer un algorithme permettant de trier le tableau x dans l'ordre décroissant.

Nom, Prenom:_____ **Groupe:**_____

Travail pratique

Nous allons compléter la classe Sort qui pemet de trier un tableau de données. Les premières méthodes de la classe sort vous sont founies:

```
// Constructeur
// Crée un tableau de 10 valeurs
// Et le remplit au hasard.
Sort::Sort()
```

```
// Destructeur
// Libère la mémoire allouée pour le tableau
Sort::~Sort()
```

```
// Rempli le tableau avec des valeurs aléatoires.
// La graine du générateur pseudo aléatoire
// est passée en paramètre, pour
// permettre la répétabilité
void Sort::FillRandom(int n, int seed)
```

```
// Affiche le contenu du tableau
void Sort::Print()
```

Voici un exemple de code permettant de créer un tableau, de le remplir avec 150 valeurs aléatoires puis de l'afficher.

```
Sort Tri;
Tri.FillRandom(150,20);
Tri.Print();
```

Notez que le deuxième paramètre de la fonction FillRandom permet d'initialiser le générateur pseudo-aléatoire. Ce paramètre est appelé la 'graine' du générateur. Deux tirages utilisant la même graine fournissent la même séquence.

Exercice 1: Tri par sélection

a. Implémenter la méthode Permute qui permet de permuter les éléments i1 et i2 du tableau x.

```
// Permute les élements i1 et i2 du tableau
void Sort::Permute(int i1,int i2)
```

b. Implémenter la méthode Max qui retourne l'indice du plus grand élément du tableau x compris entre les indices [Start ; End[.

```
// Retourne l'indice du plus grand élément du tableau
// Compris entre les indices Start (inclus) et End (non inclus)
int Sort::Max(int Start,int End)
```

c. Implementer la méthode Tri_Selection_Decroissant qui permet de trier le tableau x dans l'ordre décroissant en utilisant la méthode de tri par sélection.

```
// Trie les élements de x dans l'ordre décroissant
// en utilisant la méthode de tri par sélection
void Sort::Tri_Selection_Decroissant()
```

d. Compter le nombre de permutations nécessaires pour trier un tableau de 1250 cellules.

Nombre de permutation(s) pour un tableau de 1250 cellules : _____

e. Compter le nombre de permutations nécessaires pour trier un tableau déjà trié.

Nombre de permutation(s) pour un tableau trié croissant de 1250 cellules : _____

Nombre de permutation(s) pour un tableau trié décroissant de 1250 cellules : _____

Exercice 2: Tri par insertion

a. Implémenter la méthode décale qui décale tous les éléments compris entre [Start;Stop[de une cellule vers la droite. La valeur située à l'indice stop est retrounée par la méthode. Nous supposerons que Start est inférieur à Stop.

```
// Décale tous les élément de 1 cellule
// de start vers stop. x[stop] est perdu
// Start < Stop
int Sort::Decale(int Start,int Stop)
```

b. Implémenter une méthode qui insert l'élément x[Stop] à sa place dans le tableau entre les éléments [Start;Stop[en supposant que cette portion du tableau est déjà ordonnée dans l'ordre décroissant.

```
// Insere l'élément x[Stop] dans le tableau
// entre les éléments Start et Stop
// Les éléments sont décalés vers l'indice Stop
void Sort::Insert(int Start,int Stop)
```

c. Implémenter la méthode Tri_Insertion_Decroissant qui permet de trier le tableau x dans l'ordre décroissant en utilisant la méthode de tri par insertion.

```
// Trie les élements de x dans l'ordre décroissant
// en utilisant la méthode de tri par insertion
void Sort::Tri_Insertion_Decroissant()
```

d. Compter le nombre d'appels de la fonction Decale nécessaires pour trier un tableau de 1250 cellules. La graine du générateur pseudo-aléatoire sera 0.

Nombre d'appels de la fonction Decale pour un tableau de 1250 cellules : _____

e. Compter le nombre d'appels de la fonction Decale nécessaires pour trier un tableau déjà trié.

Nombre de d'appels pour un tableau trié croissant de 1250 cellules : _____

Nombre d'appels pour un tableau trié décroissant de 1250 cellules : _____

Nom, Prenom:_____ **Groupe:**_____

Exercice 3: Tri à bulles

a. Implémenter la méthode Tri_Bulles qui permet de trier le tableau x dans l'ordre décroissant en utilisant la méthode de tri à bulles.

```
// Trie les élements de x dans l'ordre décroissant
// en utilisant la méthode de tri à bulles
void Sort::Tri_Bulles_Decroissant()
```

b. Compléter le tableau suivant qui représente le nombre de permutation(s) nécessaire(s) au tri du tableau. La graine du générateur pseudo-aléatoire sera 0.

Nombre d'éléments	**10**	**100**	**1000**	**10000**
Tableau aléatoire				
Tableau trié décroissant				
Tableau trié croissant				

c. En déduire la complexité en terme de permutations de l'algorithme dans le meilleur cas, dans le pire cas et dans le cas moyen.

Meilleur cas	**Cas moyen**	**Pire cas**

Exercice 4: QuickSort

a. Ecrire la méthode Pivote_Decroissant qui prend l'élément x[Start] comme pivot et le place à son emplacement définitif. Tous les éléments avant le pivot doivent être supérieurs et tous les élements après le pivot doivent être inférieurs au pivot. La fonction retourne l'indice du pivot après traitement.

```
// Prends l'élément x[Start] comme pivot et
// le place de façon définitive afin que tous les
// éléments avant le pivot soient plus petits,
// et tous les élements après le pivot soient plus
// grands. Le fonction retourne l'indice du pivot.
int Sort::Pivote(int Start, int Stop)
```

Nom, Prenom:_____ **Groupe:**_____

b. Implémenter la méthode QuickSort_Recursif_Decroissant qui permet de trier le tableau x dans l'ordre décroissant en utilisant la méthode de tri rapide sur l'intervalle [Start;Stop[.

```
// Trie les élements de x dans l'ordre décroissant
// en utilisant la méthode de tri rapide
void Sort::QuickSort_Recursif_Decroissant(int Start, int Stop)
```

c. Ecrire la méthode QuickSort_Decroissant qui permet de trier l'ensemble du tableau x dans l'ordre décroissant en utilisant la méthode de tri rapide.

```
// Trie les élements de x dans l'ordre deroissant
// en utilisant la méthode de tri rapide
void Sort::QuickSort_Decroissant()
```

d. Compléter le tableau suivant qui représente le nombre de permutation(s) nécessaire(s) au tri du tableau. La graine du générateur pseudo-aléatoire sera 0.

Nombre d'éléments	**10**	**100**	**1000**	**10000**
Tableau aléatoire				
Tableau trié décroissant				
Tableau trié croissant				

c. En déduire la complexité en terme de permutations de l'algorithme dans le meilleur cas, dans le pire cas et dans le cas moyen.

Meilleur cas	**Cas moyen**	**Pire cas**

Nom, Prenom:_____ **Groupe:**_____

Exercice 5: Complexité

a. Pour chaque algorithme, faites un relevé de l'évolution du temps de calcul en fonction de la taille du tableau à trier, dans le meilleur cas, dans le pire cas et dans le cas moyen.

Imprimer vos courbes et joignez-les au compte-rendu.

b. Pour chaque algorithme, expliquer quel est le meilleur cas et le pire cas.

c. Déterminer l'ordre de grandeur de la complexité de chaque algorithme.

	Selection	**Insertion**	**Bulles**	**QuickSort**
Meilleur cas				
Cas moyen				
Pire cas				