TP4 : Fichier et complexité expérimentale

Le but de ce TP est de comparer les temps d'exécution de différents algorithmes de tri, lorsqu'on les fait tourner sur des volumes de données relativement grands. Les données à trier seront ici des nombres complexes, stockés dans un fichier. Récupérez sur le site de l'UE les fichiers random.txt, sorted.txt et reverse.txt.

Exercice 1 : Préambule : surchage d'opérateurs

Afin de faciliter l'écriture des instructions utiles pour les entrées/sorties sur fichier et console, et les tris, vous allez surcharger certains opérateurs de la classe `NbComplexe` que vous avez créé au TP précédent. Reprenez votre fichier `NbComplexe.cpp` (ou celui du corrigé) et ajoutez les opérateurs suivants : = (affectation), < (strictement inférieur, qui remplace la fonction membre `estPlusPetit`), * (multiplication, qui remplace la fonction membre `multiplier`), << (écriture sur un flux, qui remplace la fonction membre d'affichage) et >> (lecture sur un flux, qui remplace la fonction membre de saisie). Vous conserverez à l'identique la fonction membre `module`, les constructeurs et le destructeur. Mettez à jour les appels dans les fonctions globales `trierParSelection` et `trierParInsertion` afin d'utiliser ces opérateurs au lieu des anciennes fonctions membres. Testez les opérateurs et les fonctions de tri.

```
NbComplexe& operator = (const NbComplexe& operandeDroite) {
  re = operandeDroite.re;
  im = operandeDroite.im;
  return *this;
}

bool operator < (const NbComplexe& operandeDroite) const {
  return module() < operandeDroite.module();
}

NbComplexe operator * (const NbComplexe& operandeDroite) const {
  NbComplexe resultatMult;
  resultatMult.re = re*operandeDroite.re - im*operandeDroite.im;
  resultatMult.im = im*operandeDroite.re + re*operandeDroite.im;
  return resultatMult;
}

friend ostream& operator << (ostream& flux, const NbComplexe& c) {
  flux << c.re << " ";
  if (c.im >= 0.0) flux << "+";
  flux << c.im << " i";
  return flux;
}

friend istream& operator >> (istream& flux, NbComplexe& c) {
  string mot_i;
  flux >> c.re;
  flux >> c.im;
  flux >> mot_i; // afin de lire le caractère 'i'
  return flux;
}
```

```
void trierParSelection (NbComplexe * tab, int taille) {
  NbComplexe minComplexe;
  for (int i=0; i<taille-1; i++) {
    int indmin = i;
    for (int j=i+1; j<taille; j++){
      if (tab[j] < tab[indmin]) indmin = j;
```

```
    }
    minComplexe = tab[indmin]; // opérateur d'affectation
    tab[indmin] = tab[i];
    tab[i] = minComplexe;
  }
}

void trierParInsertion (NbComplexe * tab, int taille) {
  NbComplexe complexeAPlacer;
  for (int i=1; i<taille; i++) {
    complexeAPlacer = tab[i];
    int j = i - 1;
    while ( j>=0 && complexeAPlacer < tab[j] ) {
      tab[j+1] = tab[j];
      j--;
    }
    tab[j+1]=complexeAPlacer;
  }
}
```

Exercice 2 : Lire un fichier texte de nombres complexes

Ouvrez les fichiers .txt fournis pour en comprendre la structure, puis fermez-les. Ajoutez ensuite dans le fichier `NbComplexe.cpp` une fonction globale pour lire le contenu d'un fichier et remplir un tableau avec les nombres complexes contenus dans le fichier. Ensuite le programme triera le tableau avec l'algorithme de votre choix et l'affichera à l'écran. Attention, votre programme devra s'adapter automatiquement (i.e. sans qu'il y ait besoin de recompiler) au nombre de complexes contenus dans le fichier.

L'entête de cette procédure de lecture du fichier est la suivante :

Procédure lireTabNbComplexeDepuisFichier (tab : tableau de nombres complexes, taille : entier, nom_fichier : chaîne de caractères)
Précondition : tab n'est pas alloué, la procédure est en charge d'allouer la mémoire pour contenir les nombres complexes lus depuis le fichier. Le fichier est au format texte et commence par une ligne contenant le nombre de complexes à lire.
Postcondition : tab contient les nombres complexes lus depuis le fichier de nom nom_fichier
Paramètres en mode donnée : nom_fichier
Paramètres en mode donnée-résultat : tab, taille

Testez votre procédure en construisant un tableau de nombres complexes depuis l'un des trois fichiers fournis, puis affichez-le à l'écran. N'oubliez pas d'inclure la librairie `fstream` lors de vos tests.

```
void lireTabNbComplexeDepuisFichier (NbComplexe*& tab, int& taille, const string& nom_fichier)
{
  ifstream fichierLecture(nom_fichier.c_str());
  if (!fichierLecture.is_open()) {
    cout << "Erreur dans l'ouverture en lecture du fichier : " << nom_fichier << endl;
    return;
  }
  fichierLecture >> taille;
  if (taille <= 0) return;
  tab = new NbComplexe [taille];
  for (int i=0; i<taille; i++) fichierLecture >> tab[i];
  fichierLecture.close();
}
```

Appel depuis le main:

```
NbComplexe * tabLu = NULL;
int tailleLue = 0;
lireTabNbComplexeDepuisFichier(tabLu,tailleLue,"random.txt");
delete [] tabLu;
```

Exercice 3 : Ecrire un fichier texte de nombres complexes

Ajoutez dans votre programme une procédure qui écrit un tableau de nombres complexes dans un fichier txt, en respectant le format des fichiers fournis sur le site de l'UE. Testez votre procédure en l'appelant sur le tableau après un tri par insertion ou par sélection.

L'entête de cette procédure d'écriture du fichier est la suivante :

Procédure ecrireTabNbComplexeDansFichier (tab : tableau de nombres complexes, taille : entier, nom_fichier : chaîne de caractères)
Précondition : tab contient les nombres complexes à écrire dans le fichier.
Postcondition : le fichier de nom nom_fichier est au format texte et commence par une ligne contenant le nombre de complexes du fichier, et il contient ensuite les nombres complexes du tableau tab
Paramètres en mode donnée : tab, taille, nom_fichier

```
void ecrireTabNbComplexeDansFichier (const NbComplexe * tab, int taille, const string & nom_fichier) {
  ofstream fichierEcriture(nom_fichier.c_str());
  if (!fichierEcriture.is_open()) {
    cout << "Erreur dans l'ouverture en ecriture du fichier : " << nom_fichier << endl;
    return;
  }
  fichierEcriture << taille;
  for (int i=0; i<taille; i++) fichierEcriture << endl << tab[i];
  fichierEcriture.close();
}
```

Appel depuis le main :

```
ecrireTabNbComplexeDansFichier(tab,taille,"desNombresComplexes.txt");
```

Exercice 4 : Mesurer les temps d'exécution des algorithmes de tri

Dans cet exercice vous allez tester le comportement du tri par sélection et du tri par insertion en exécutant votre programme sur les trois fichiers txt fournis. Notez à chaque fois le temps d'exécution pour compléter le tableau suivant. Vous pouvez vous reportez à l'annexe C pour apprendre comment mesurer le temps d'exécution d'un bout de code.

Fichier à trier	Tri par sélection	Tri par insertion
random.txt		
sorted.txt		
reverse.txt		

Lequel des deux algorithmes est le plus performant sur un fichier aléatoire ? Même question pour un fichier déjà trié et un fichier trié dans l'ordre inverse ?

Les instructions pour mesurer le temps de tri sont :

```
clock_t tempsExec = clock();
trierParSelection(tab,taille); // ou trierParInsertion(tab,taille);
tempsExec = clock() - tempsExec;
cout << "tri en " << ((float)tempsExec)/CLOCKS_PER_SEC << " secondes" << endl;
```

Exemple sur une machine donnée (temps en secondes):

Fichier à trier	Tri par sélection	Tri par insertion
random.txt	17.36	8.852
sorted.txt	17.29	0.002
reverse.txt	17.46	17.42