R1.01 – TP9 (SAE 1.02)
Vecteurs en ArrayList – Coût en nombre de comparaisons

L'objectif de ce TP est de retravailler encore les algorithmes de recherche et de tri dans des vecteurs en ArrayList et de mettre en œuvre leur outillage pour être en mesure de les comparer.

Pour ce TP, le baromètre de calcul de la complexité d'un algorithme est exclusivement le nombre de comparaisons entre deux éléments du type du vecteur traité par une fonction ou une procédure.

- Reportez-vous au cours, pour vérifier si le nombre de comparaisons que vous obtenez est cohérent avec sa classe de complexité.
- Pour chaque algorithme demandé, réfléchissez aux facteurs d'influence de leur performance

Avant de commencer...

- ✓ Lisez entièrement ce sujet
- ✓ Ouvrez un terminal et placez-vous dans votre répertoire `R1.01`
- ✓ Lancez `IJ`, puis à partir du menu `File` (fichiers), créez un nouveau projet `TP9`

1. Algorithmes de tri d'un ArrayList<Integer>

1.1. Dans le projet `TP9`, créez une classe `AlgoIntegerTri` où vous ajouterez, puis coderez les fonctions suivantes :

a) Tri à bulles amélioré, outillé :

```
public static int triBulleOutille(ArrayList<Integer> vInt) {
    // { vInt quelconque }
    // => { * vInt a été trié par la méthode du tri à bulles amélioré
    //      * résultat = nombre de comparaisons entre deux éléments de vInt }
```

b) Tri par sélection, outillé :

```
public static int triSelectOutille(ArrayList<Integer> vInt) {
    // { vInt quelconque }
    // => { * vInt a été trié par la méthode du tri par sélection
    //      * résultat = nombre de comparaisons entre deux éléments de vInt }
```

c) Tri par insertion, outillé :

```
public static int triInsertOutille(ArrayList<Integer> vInt) {
    // { vInt quelconque }
    // => { * vInt a été trié par la méthode du tri par insertion
    //      * résultat = nombre de comparaisons entre deux éléments de vInt }
```

d) Tri fusion outillé

1 – le modèle

```
public static int triFusionOutille(ArrayList<Integer> vInt, int inf, int sup) {
    // { vInt[inf..sup] non vide }
    //  => { * vInt[inf..sup] trié
    //       * résultat = nombre de comparaisons entre deux éléments }
```

2 – le worker

```
public static int fusionTabGTabDOutille(ArrayList<Integer> vInt,
                                        int inf, int m, int sup) {
    //{ inf <= sup, m = (inf+sup)/2, vInt[inf..m] trié,vInt[m+1..sup] trié }
    //  => { * vInt[inf..sup] trié
    //       * résultat = nombre de comparaisons entre deux éléments }
```

1.2. Dans la classe `AlgoIntegerTri`, ajoutez une procédure `main` où vous testerez, *au fur et à mesure*, chacun des tris, avec chaque vecteur défini ci-dessous :

```
CAS 1 : vecteur initial = [12, 7, 9, 14, 5, 17, 6, 8, 12]
CAS 2 : vecteur initial = [-44, -45, 9, -12, 30, 56, 7, -3, 19, -45, 9, 23, 11, 150, 28, 34, 1, 25]
CAS 3 : vecteur initial = [17, 14, 12, 12, 9, 8, 7, 6, 5]
```

Instructions à écrire :

- ✓ déclaration d'un `ArrayList<Integer>` nommé `vIntBase`
- ✓ déclaration d'un autre `ArrayList<Integer>` de nom `vInt`
- ✓ autres déclarations nécessaires au test de chaque tri
- ✓ test (*à répéter, au fur et à mesure du codage des tris outillés*), pour chaque vecteur des cas 1, 2 et 3 :
  - l'initialisation de vIntBase avec les éléments du vecteur initial
  - l'affichage du vecteur initial et de son nombre d'éléments
  - l'initialisation de `vInt` avec l'instruction : `vInt = new ArrayList(vIntBase);`
  - l'affichage de `vInt` avant tri
  - l'appel de la fonction à tester
  - l'affichage du vecteur trié
  - l'affichage du nombre de comparaisons effectuées

EXEMPLE de Trace attendue (Traitement du CAS 1 - *le nombre de comparaisons est masqué*)

```
Vecteur initial : [12, 7, 9, 14, 5, 17, 6, 8, 12]
Nombre d'éléments : 9

A - Tri à bulles amélioré
   * Vecteur initial : [12, 7, 9, 14, 5, 17, 6, 8, 12]
   * Vecteur trié    : [5, 6, 7, 8, 9, 12, 12, 14, 17]
   * Nombre de comparaisons : ■

B - Tri par sélection
   * Vecteur initial : [12, 7, 9, 14, 5, 17, 6, 8, 12]
   * Vecteur trié    : [5, 6, 7, 8, 9, 12, 12, 14, 17]
   * Nombre de comparaisons : ■

C - Tri par insertion
   * Vecteur initial : [12, 7, 9, 14, 5, 17, 6, 8, 12]
   * Vecteur trié    : [5, 6, 7, 8, 9, 12, 12, 14, 17]
   * Nombre de comparaisons : ■

D - Tri fusion
   * Vecteur initial : [12, 7, 9, 14, 5, 17, 6, 8, 12]
   * Vecteur trié    : [5, 6, 7, 8, 9, 12, 12, 14, 17]
   * Nombre de comparaisons : ■
```

1.3. Testez…

2. Algorithmes de recherche dans un ArrayList<Integer> trié

2.1. Créez une classe `PaireResCompteur` dans laquelle vous collerez le code donné ci-dessous après avoir supprimé la 1ère ligne créée automatiquement :

```
/** Classe générique pour outiller une fonction
 * Elle propose uniquement un constructeur et 2 getters
 * @param <R> : R est le type du résultat de la fonction outillée
 */
public class PaireResCompteur <R>  {
    private R res; // résultat de la fonction outillée
    private int compteur; // compteur du code observé

    public PaireResCompteur(R res, int compteur) {
        this.res = res;
        this.compteur = compteur;
    }
    public R getRes() {
        return res;
    }
    public int getCompteur() {
        return compteur;
    }
}
```

2.2. Créez une classe `AlgoIntegerRech` dans laquelle vous ajouterez et coderez les fonctions suivantes :

a) Recherche séquentielle outillée de l'indice de la 1ère occurrence d'un entier donné

```
public static PaireResCompteur<Integer> indRechSeqO(ArrayList<Integer> vInt,
                                                    int unInt) {
    //{ vInt non vide, trié }
    // => { à la fin du traitement, ind est l'indice de la 1ère occurrence
    //      de unInt dans vInt, ou est égal à -1 si unInt n'est pas dans vInt
    //
    //      résultat = variable de type PaireResCompteur avec :
    //                 - res = ind
    //                 - compteur = nombre de comparaisons effectuées
    //                   entre inInt et un élément de vInt
    //      RECHERCHE SÉQUENTIELLE }
```

b) Recherche dichotomique itérative outillée de la 1ère occurrence d'un entier donné

```
public static PaireResCompteur<Integer> indRechDichoItO(ArrayList<Integer> vInt,
                                                        int unInt) {
    //{ vInt non vide, trié }
    // => { à la fin du traitement, ind est l'indice de la 1ère occurrence
    //      de unInt dans vInt,ou est égal à -1 si unInt n'est pas dans vInt
    //
    //      résultat = variable de type PaireResCompteur avec :
    //                 - res = ind
    //                 - compteur = nombre de comparaisons effectuées
    //                   entre inInt et un élément de vInt
    //      RECHERCHE DICHOTOMIQUE ITÉRATIVE }
```

c) Recherche dichotomique récursive outillée de la 1ère occurrence d'un entier donné

1 – le Modèle

```
public static PaireResCompteur<Integer> indRDichoRecO(ArrayList<Integer> vInt,
                                                      int unInt) {
    //{ vInt non vide, trié }
    // => { à la fin du traitement, ind est l'indice de la 1ère occurrence
    //      de unInt dans vInt, ou est égal à -1 si non trouvé
    //
    //      résultat = variable de type PaireResCompteur avec :
    //                 - res = ind
    //                 - compteur = nombre de comparaisons effectuées
    //                   entre inInt et un élément de vInt[inf..sup] }
```

2 – le Worker

```
public static PaireResCompteur<Integer> indRDichoRecWorker0(ArrayList<Integer> vInt,
                                                            int unInt, int inf,
                                                            int sup) {
    //{ vInt[inf..sup] non vide, trié }
    // => { à la fin du traitement, ind est l'indice de la 1ère occurrence
    //      de unInt dans vInt[inf..sup], ou est égal à -1 si non trouvé
    //
    //      résultat = variable de type PaireResCompteur avec :
    //                 - res = ind
    //                 - compteur = nombre de comparaisons effectuées
    //                   entre inInt et un élément de vInt[inf..sup]
    //      RECHERCHE DICHOTOMIQUE RÉCURSIVE }
```

2.3. Dans la classe `AlgoIntegerRech` créez une procédure `main` dans laquelle vous ajouterez :

- ✓ la déclaration d'un `ArrayList<Integer>` initialisé avec les valeurs (-45, -45, -10, 9, 20, 30, 75)
- ✓ l'affichage de cet `ArrayList` et de son nombre d'éléments
- ✓ la saisie d'un entier quelconque
- ✓ pour chaque fonction de recherche à tester :
  - L'affichage d'un message indiquant quel est l'algorithme de la fonction testée (EX : RECHERCHE SÉQUENTIELLE)
  - Si l'entier saisi a été trouvé dans le vecteur, l'affichage de son indice et s'il n'a pas été trouvé, l'affichage d'un message indiquant l'échec de la recherche
  - L'affichage du nombre de comparaisons effectuées par la fonction

2.4. Testez…

3. Algorithmes dans un ArrayList<String> non trié

3.1. Créez une classe `AlgoString` dans laquelle vous collerez le code de la procédure ci-dessous :

```
public static void affichVect(ArrayList<String> vString) {
    // { vString non vide } =>
    //  { le contenu de vString est affiché  de façon "lisible" sur un petit écran }
    System.out.println();
    System.out.println("-- VECTEUR D'INSTRUMENTS DE CUISINE --");
    System.out.print("[");
    int i = 0;
    int nb = 0;
    while (i < vString.size() - 1) {
        System.out.print(vString.get(i) + ", ");
        i++;
        nb++;
        if (nb % 7 == 0 && nb < vString.size() - 1) {
            System.out.print("\n ");
        }
    }
    System.out.println(vString.get(vString.size() - 1) + "]");
    System.out.println();
    System.out.println("Nombre d'éléments : " + vString.size());
    System.out.println("---------------------------------------");
    System.out.println();
}
```

3.2. Créez dans la classe `AlgoString` une procédure `main`, où vous :

- ✓ collerez la déclaration suivante :

```
ArrayList<String> vInstruments = new ArrayList<>(Arrays.asList("casserole", "fourchette",
"cuillère", "couteau","passoire", "tamis", "spatule", "fouet", "éplucheur", "bain-marie",
"râpe", "presse-ail", "presse-agrumes","ouvre-boîte", "thermomètre", "pince", "balance",
"presse-purée", "louche", "minuteur", "ciseaux", "bol","mandoline", "doseur", "shaker"));
```

- ✓ afficherez le contenu de ce vecteur en appelant la procédure `affichVect`

3.3. Ajoutez à la classe `AlgoString` les fonctions suivantes (avant sa procédure `main`) :

a) Recherche séquentielle outillée, du nombre de caractères minimum d'un élément d'un vecteur de String

```
public static PaireResCompteur<Integer> nbCarMinOutille(ArrayList<String> vString) {
    // { vInstruments non vide }
    // => { à la fin du traitement, min est le nombre de caractères
    //       minimum d'un élément du vecteur
    //
    //       résultat = variable de type PaireResCompteur avec
    //                  - res = min
    //                  - compteur = nombre de comparaisons effectuées pour
    //                    produire min
    //      RECHERCHE SÉQUENTIELLE }
```

b) Recherche récursive "Diviser pour Régner" outillée, du nombre de caractères minimum d'un élément d'un vecteur de String

1 – le Modèle

```
public static PaireResCompteur<Integer> nbCarMinDPRO(ArrayList<String> vString) {
    //{ vString non vide }
    // => { à la fin du traitement, min est le nombre de caractères
    //      minimum d'un élément de vString
    //
    //      résultat = variable de type PaireResCompteur avec
    //                 - res = min
    //                 - compteur = nombre de comparaisons effectuées pour
    //                   produire min
```

2 – le Worker

```
public static PaireResCompteur<Integer> nbCarMinDPRWorkerO(ArrayList<String> vString,
                                                           int inf, int sup) {
    // { vString non vide }
    // => { à la fin du traitement, min est le nombre de caractères
    //      minimum d'un élément de vString[inf..sup]
    //
    //      résultat = variable de type PaireResCompteur avec
    //                 - res = min
    //                 - compteur = nombre de comparaisons effectuées pour
    //                   produire min
    //      RECHERCHE RÉCURSIVE DIVISER POUR RÉGNER }
```

3.4. Complétez la `procédure` main de la classe `AlgoString` avec les déclarations et instructions nécessaires au test des fonctions que vous avez codées en **3.3**

4. EXTENSION : nombre de comparaisons sur grands vecteurs

OBJECTIF : Étudier plus finement l'influence de la taille du vecteur, de l'ordre de ses éléments (non ordonnés, vs. ordonnés) et éventuellement du fait qu'ils comportent ou non des doublons.

Pour cela, vous utiliserez des vecteurs de taille croissante (ex : 10, 50, 100, 500, 1000 éléments) dont les éléments seront générés aléatoirement avec la fonction `Math.random()` qui retourne une valeur de type `double`

EXEMPLE : *affectation à un entier d'un entier aléatoirement "choisi" dans l'intervalle [0, 300]*

```
int val = (int) (Math.random()*301 ;
```

NOTE : des appels successifs de Math.random() peuvent produire un résultat identique

- Créez une classe `Utilitaire` dans laquelle vous ajouterez les fonctions suivantes :
  - ✓ *fonction retournant un* `ArrayList<Integer>` *de taille* ***n****, dont les éléments sont générés par tirage aléatoire d'une valeur comprise entre* ***0*** *et* ***2*n***
  - ✓ *fonction retournant un* `ArrayList<Integer>` *de taille* ***n****, dont les éléments sont générés par tirage aléatoire d'une valeur comprise entre* ***0*** *et* ***2*n****, sans doublons*
  - ✓ *procédure (non outillée) de tri d'un* `ArrayList<Integer>` *- Méthode de tri au choix…*
- Ajoutez dans la classe `AlgoIntegerRech` la déclaration de vecteurs de taille croissante, dont les éléments sont générés aléatoirement, et testez sur ces vecteurs les différentes fonctions de recherche.