

# TP n° 10

## Algorithmes de recherche et de tri

L'objectif de cette séance de travaux pratiques est d'expérimenter en Java divers algorithmes de recherche et de tri.

### Exercice 1 – Recherche par dichotomie

L'existence d'un ordre total (i.e. tous les éléments de la liste sont comparables 2 à 2) sur les éléments d'une liste permet d'obtenir une liste triée et ainsi optimiser significativement la recherche d'un élément par l'utilisation de l'algorithme de recherche pas dichotomie donné ci-dessous :

**Fonction** recherche\_dichotomie(*l* : Liste<Élément>, *e* :Élément) **résultat** Entier

inf, sup, p : Entier

**début**

inf ← 0; // la borne inf est incluse

sup ← longueur(*l*); // la borne sup est exclue

**tant que** *inf* ≠ *sup* - 1 **faire**

*p* ← (*inf* + *sup*)/2;

**si** *e* ≥ *ième*(*l*,*p*) **alors** *inf* ← *p*; **sinon** *sup* ← *p*;

**fin**

**si** *ième*(*l*,*inf*) = *e* **alors**

    retourner *inf*;


**fin**


**sinon**

    erreur "élément introuvable"

**fin**


**fin**


 Dans un nouveau projet Eclipse, créer une nouvelle classe **ListeEntier** qui dérive d'une **ArrayList<Integer>**. Dans le constructeur de cette classe, cette liste est initialisée comme suit : On parcourt tous les entiers dans l'intervalle  $[1, 10^5]$  et pour chacun, on effectue un tirage aléatoire booléen. Si le tirage est Vrai, alors on ajoute l'entier dans la liste.

 Dans le programme principal, construire une nouvelle **ListeEntier** et vérifier qu'après sa construction, cette liste contient (en moyenne) 5.000 éléments.

 Dans le programme principal, mettre en œuvre un premier test de performance qui consiste

à tirer aléatoirement un nombre dans l'intervalle  $[1, 10^5]$  et vérifier s'il est contenu dans la liste. On effectue 1000 fois ce test en mesurant i) le taux de succès de la recherche (résultat théorique : 50%) et ii) le temps passé à effectuer les 1000 tests. Pour mesurer le temps passé, on pourra utiliser la méthode statique `System.currentTimeMillis()`.

 Implémenter la méthode `rechercher` qui prend en paramètre un entier et retourne le booléen `True` si l'entier est contenu dans la liste, `False` sinon. On implémentera une recherche par dichotomie, en adaptant l'algorithme donné ci-dessus.

 Dans le programme principal, mettre en œuvre un second test identique au précédent, mais en utilisant la méthode `rechercher`. Vous devriez constater une amélioration significative des performances, comment pouvez-vous l'expliquer ?

## Exercice 2 – Tri par sélection


Dans cet exercice, on cherche à ordonner tous les caractères d'une chaîne de caractères. Pour cela, nous allons mettre en œuvre un tri par sélection, dont l'algorithme est donné ci-dessous :


**Fonction** `tri_sélection(l : Liste<Élément>)` **résultat** `Liste<Élément>`


```

i, j, imin : Entier
début
    i ← 0;
    tant que i < longueur(l)-1 faire
        imin ← i;
        j ← i+1;
        tant que j < longueur(l)-1 faire
            si ième(l,j) < ième(l,imin) alors imin ← j;
            j ← j + 1;
        fin
        permuter(l,i,imin);
        i ← i + 1;
    fin
    retourner l
fin

```

 Dans un nouveau paquetage de votre projet, créer une classe `TriSelection`. Dans le programme principal, déclarer une chaîne de caractères `texte` prenant comme valeur, par exemple : “les sanglots longs des violons de l’automne blessent mon coeur d’une langueur monotone”.

 Dans le programme principal, trier tous les caractères de la chaîne, en implémentant le tri par sélection donné ci-dessus. Afin de trier les caractères, il est nécessaire de transformer préalablement la chaîne de caractères en tableau de caractères.

 Afficher la chaîne de caractères “triée” et constater que les caractères sont triés par ordre alphabétique.


## Exercice 3 – Tri rapide (quicksort)


Dans cette exercice, nous mettons en œuvre un tri rapide, un des algorithmes les plus efficaces pour le tri des éléments d’une collection. Il s’agit d’un algorithme récursif, dont le principe consiste à partitionner la liste en 2 sous-listes autour d’un pivot. L’algorithme général est donné ci-dessous :

**Fonction** tri\_rapide( $l$  : Liste<Élément> ;  $inf$ ,  $sup$  : Entier) **résultat** Liste<Élément>

```

     $i_{pivot}$  : Entier
    début
        si  $inf < sup$  alors
             $i_{pivot} \leftarrow \text{partitionner}(l, inf, sup)$ 
             $l \leftarrow \text{tri\_rapide}(l, inf, pivot-1)$ 
             $l \leftarrow \text{tri\_rapide}(l, pivot+1, sup)$ 
        fin
    retourner  $l$ 
fin
```

 Dans un nouveau paquetage de votre projet, créer une classe **TriRapide** qui dérive d’une **ArrayList<Integer>**. Dans le programme principal, créer un nouveau **TriRapide** et le remplir avec 30 éléments tirés aléatoirement dans  $[0, 100]$ .

 Écrire le code de la méthode **partitionner** qui prend en paramètre deux entiers (une borne  $inf$  et une borne  $sup$ ) et qui retourne un entier (la position du pivot). La fonction opère sur la sous-liste constituée de tous les éléments du **TriRapide** compris entre les bornes  $inf$  (inclue) et  $sup$  (inclue). Le principe est le suivant : elle choisit un “pivot” parmi les éléments de la sous-liste (par exemple le dernier élément). Elle va ensuite chercher à placer ce pivot à sa position finale dans la sous-liste, en respectant la propriété suivante : tous les éléments inférieurs au pivot doivent être à gauche du pivot, et tous les éléments supérieurs ou égaux doivent être à droite. Elle retourne ensuite l’indice du pivot, afin que le programme principal puisse trier récursivement les 2 partitions : la sous-liste à gauche du pivot et la sous-liste à droite du pivot. Voici quelques exemples de résultat de partitionnement de listes :

$[21, 34, 42, 39, 37, 6, 3, 2, 23, 26, 35] \rightarrow [21, 34, 6, 3, 2, 23, 26] \textbf{ 35 } [42, 39, 37]$

$[21, 34, 6, 3, 2, 23, 26] \rightarrow [21, 6, 3, 2, 23] \textbf{ 26 } [34]$

$[21, 6, 3, 2] \rightarrow \textbf{ 2 } [6, 3, 21]$

 Implémenter la méthode **trier** en vous aidant de l’algorithme donné ci-dessus.

## Annexe

Illustration des étapes de l'algorithme de tri rapide sur une collection d'entiers.

Choix du pivot

[ 2 , 8 , 7 , 5 , 4 , 1 , 9 , 5 , (3) ]

Placement pivot + appel récursif sur les 2 partitions

[ 2 , 1 ] [3] [ 5 , 4 , 8 , 9 , 5 , 7 ]

Dans les partitions d'au moins 2 éléments, choix d'un pivot

[ 2 , (1) ] [3] [ 5 , 4 , 8 , 9 , 5 , (7) ]

Placement des pivots + appel récursif sur les nouvelles partitions

[] [1] [ 2 ] [3] [ 5 , 4 , 5 ] [7] [ 8 , 9 ]

Choix des pivots dans les partitions d'au moins 2 éléments

[] [1] [] [2] [] [3] [ 5 , 4 , (5) ] [7] [ 8 , (9) ]

Placement pivot + appel récursif sur les nouvelles partitions

[] [1] [] [2] [] [3] [ 4 ] [5] [ 5 ] [7] [ 8 ] [9] []

Pas d'appel récursif car toutes les partitions sont vides ou contiennent 1 seul élément

[] [1] [] [2] [] [3] [] [4] [] [5] [] [5] [] [7] [] [8] [] [9] []

La liste est triée car tous les pivots ont été placés