TP3: Recherche dichotomique et tris

1 Préliminaires

Procédez comme pour le TP 1 et 2 pour **importer** le projet `/share/l1ie/SI1/TPrechDicho/TPrechDicho.zip`

2 Recherche dichotomique dans les tableaux d'entiers

Dans le projet `TPrechDicho`, le package `main` contient le fichier `Main.java`. Vous devrez compléter ce fichier.

Programmez la fonction de recherche dichotomique vue en cours Cette fonction recherche un entier dans un tableau trié et retourne l'index d'une occurence de cet entier dans le tableau ou -1, s'il n'y figure pas. La signature de cette fonction est :

```
int rechercheDicho(int cherche, int[] t)
```

Testez la fonction de recherche dichotomique à l'aide de JUnit Dans le cas de la recherche dichotomique il est conseillé de tester un grand nombre de cas possibles. On rappelle que vos tableaux **devront être triés**.

- **Il faut** commencer par des tests sur des petits tableaux (taille 0, puis 1, puis 2, puis 3, etc.). Les bugs seront plus faciles à localiser et à corriger sur des petits exemples ;
- **Il est inutile** de tester plusieurs fois des cas similaires ;
- Parmi les cas à tester (pour chaque taille) : la valeur apparaît (au début, à la fin, ou au milieu du tableau), la valeur n'apparaît pas, la valeur apparaît plusieurs fois, etc. Creusez-vous la tête !
- Pour vérifier si une fonction a été suffisamment testée avec JUnit, vous pouvez utiliser **Pitest** (Regardez le tutoriel vidéo). Pour lancer la couverture de tests, faire un clic droit sur le fichier `MainTest.java` sélectionner "Run as>PIT Mutation Testing".

On rappelle qu'il ne faut **jamais** effacer un test même si celui-ci réussit !

Evaluer les performances

1. Copiez d'autres fonctions de recherche de vos TP précédents dans `Main.java` et comparez les performances de `rechercheDicho` par rapport à vos autres fonctions de recherche en vous servant de `ACX.tracerRecherche`.
2. Déterminez une fonction de référence qui approxime correctement votre fonction `rechercheDicho`. **Attention !** Les fonctions de référence doivent avoir du sens vis-à-vis des complexités supposées de vos fonctions de recherche. Affichez sur deux graphiques distincts :
   - une comparaison des fonctions de recherche simple et dichotomique
   - votre fonction de recherche dichotomique avec sa fonction de référence. Que constatez-vous ?
3. Sauvegardez les graphiques dans votre espace disque. Faites contrôler le résultat par votre encadrant de TP.

Remarque : `ACX.tracerRecherche` appellera bien votre fonction `rechercheDicho` avec des tableaux triés.

3 Recherche dans les tableaux de chaînes de caractères

Programmez une fonction de **recherche simple** et une fonction de **recherche dichotomique** pour les tableaux de chaînes de caractères (`String`). Les chaînes de caractères seront comparées à l'aide de l'ordre lexicographique, qui est utilisé par exemple dans un dictionnaire. Pour comparer deux chaînes `s1` et `s2` vis à vis de l'ordre lexicographique, utiliser `s1.compareTo(s2)` dont le résultat est un entier. Cet entier est strictement négatif si `s1` est plus petite que `s2` dans l'ordre lexicographique, il est strictement positif dans le cas contraire et il est nul si les deux chaînes sont égales. Les signatures des fonctions de recherche seront :

```
int recherche(String cherche, String[] t)
int rechercheDicho(String cherche, String[] t)
```

et renverront l'indice de l'élément dans le tableau contenant la chaîne cherchée, et -1 si elles n'y sont pas. Testez vos fonctions sur des exemples de tableaux de chaînes avec JUnit. On rappelle que de tels tableaux peuvent être définis de la façon suivante : `String[] t1= {"def","abc","ab","jed"};`

4 Recherche dans un lexique et correction orthographique

On souhaite chercher les mots d'un **texte** dans un **lexique** français et signaler les mots mal orthographiés. Dans le répertoire `lib` du projet Eclipse vous trouverez un **lexique** du français courant dans le fichier `dico.txt`. Celui-ci comporte plus de 280.000 mots de la langue française. Vous trouverez également un extrait du roman Germinal d'Émile Zola `germinalExtrait.txt`.

Dans le fichier `Main.java`, complétez la fonction `main` de façon à réaliser ce qui suit.

- A l'aide de la fonction `String[] ACX.lectureDico(String nomFichier)` récupérez le **lexique** sous la forme d'un tableau de `String`. Le nom de fichier est `"lib/dico.txt"`.
- A l'aide de la fonction `String[] ACX.lectureTexte(String nomFichier)`, récupérez le **texte** sous la forme d'un tableau de `String`. Le nom de fichier est `"lib/germinalExtrait.txt"`.
- Cherchez chaque mot du **texte** dans le **lexique** et affichez chaque mot du **texte** en l'annotant par un symbol `*` s'il ne figure pas dans le **lexique**.

Attention ! Le lexique est imparfaitement trié, vous ne pourrez donc pas utiliser la recherche dichotomique (pour l'instant) pour cette étape.

5 Bonus : proposition d'algorithmes de tri

Pour que la recherche dichotomique (plus performante) opère il faut que le tableau soit trié. On propose maintenant de réfléchir à un algorithme de tri pour les tableaux. Définir une fonction qui trie un tableau de `String`. La signature de cette fonction sera : `void tri(String[] t)`

Pour évaluer les performances (et la correction) de votre fonction de tri, vous pouvez utiliser :

- `ACX.tracerTriString(String[] fonctions)`, qui trace les courbes de temps de calcul pour les fonctions dont les noms figurent dans le tableau `fonctions`.
- `ACX.tracerTriString(String[] fonctions, String[] fonctionsReference)`, qui trace les courbes de temps de calcul pour les fonctions dont les noms figurent dans le tableau `fonctions` et trace les courbes pour des fonctions de référence dont les noms figurent dans le tableau `fonctionsReference`. La signature des fonctions de référence doit être de la forme `int f(int x)`.

Pour finir, tentez de trier le tableau de `String` extrait du lexique `dico.txt` et utilisez la recherche dichotomique pour repérer les mots mal orthographiés du texte de Zola. A défaut, tentez de trier `petit_dico.txt` qui ne contient que 8500 entrées.