INF2010

Contrôle périodique 1

Mon nom de famille est : Bourai

9/10

Mon prénom est : Sami

Mon matricule est : 2041659

Directives :

- Le contrôle périodique 1 prend la forme d'un devoir à faire chez soi ;
- Vous avez jusqu'au 19 octobre pour compléter le travail ;
- L'examen est noté sur un total de 10 points et compte pour 15% de la note finale du cours ;
- Ne posez pas de question concernant l'examen. En cas de doute sur le sens d'une question, énoncez clairement toute supposition que vous faites.
- Ne communiquez avec personne au sujet de l'examen. Cet examen doit être réalisé de manière individuelle.
- Quand vous aurez fini, remettez un PDF comportant vos réponses aux questions ;
- La remise s'effectue sur Moodle ;
- Les remises par courriel ne sont pas acceptées ;
- La remise doit se faire au plus tard le 19 octobe à 23h59.

Réécrivez ci-après la phrase « J'affirme sur mon honneur avoir fait cet examen sans l'aide de personne » :

J'affirme sur mon honneur avoir fait cet examen sans l'aide de personne

---

Bonne chance!

On vous demande de contribuer à la réalisation d'un logiciel du jeu de Scrabble. L'implémentation de certaines fonctionnalités vous est déjà fournie dans le code source du fichier compressé `intra1_a2020.zip`. Le fichier compressé contient les fichiers suivants :

Fichier	Description
`Intra1A2020.java`	`Classe contenant la fonction principale main()`
Autres fichiers `*.java`	Diverses classes déjà implémentée
Fichier texte `ods.txt`	Dictionnaire officiel du Scrabble

L'examen se subdivise en quatre parties. Vous devrez changer la valeur de certains booléens dans la fonction principale `main()` pour les exécuter.

Partie 1 :

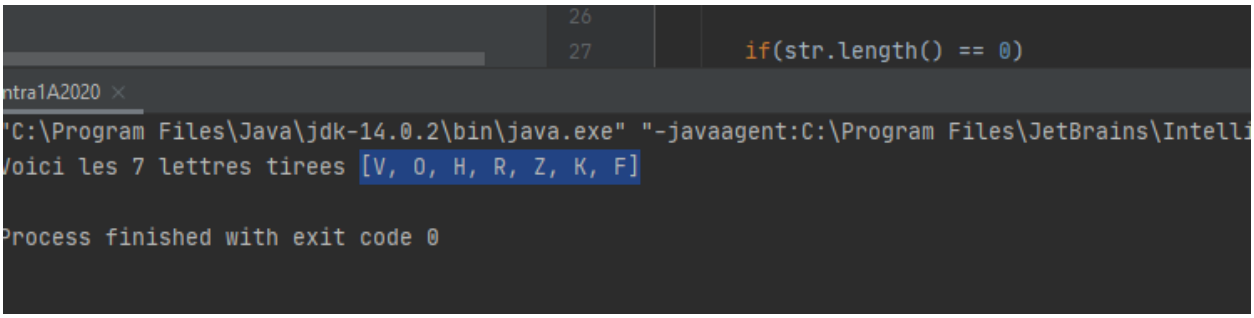
Pour que le code compile, vous devez modifier la ligne indiquée ci-après dans le fichier `Util.java` avec vos informations personnelles.

```
public class Util {

   private static final int MON_MATRICULE = ; // <= A COMPLETER
```

En exécutant le programme, vous devriez voir s'afficher un texte similaire à ceci (les lettres affichées dépendent du matricule ; ici, nous avons utilisé 1625144) :

```
Voici les 7 lettres tirées [E, S, Z, O, L, S, M]
```

1) **(0.5 point)** Donnez l'affichage résultant de l'ajout de votre matricule au fichier `Util.java`. 0,5/0,5



2) **(1 point)** La classe `Scrabble` décrite dans le fichier `Scrabble.java` contient la méthode `getLetters(int nbLetters)` qui prend en entrée un entier indiquant le nombre de lettres demandées et qui retourne une liste de lettres tirées au hasard parmi les lettres disponibles. Donnez la complexité en pire cas de cette méthode en fonction du nombre de lettres demandées, soit `nbLetters`, qu'on notera m, et le nombre de lettres disponibles, soit la taille de `availableLetters`, qu'on notera n. Justifiez adéquatement votre réponse.

C'est soit la complexité de sheffleLetter ou celle de la boucle while …. Ça se décide entre la plus grande d'entre elles ….a voir

Or la complexite de sheffleLetter est de o(n^2) . Dans les paramètres de celle-ci on y retrouve List qui est de type linkedList. Plus bas on a une boucle for qui va jusqu'à n élément et dans celle-ci on fait usage de la fonction get() sur la linkedList List, ce qui donne comme complexité O(n). alors la complexité de shuffleLetter devient O(n*n)= O(n^2). 1/1

3) **(1 point)** La classe `Scrabble` décrite dans le fichier `Scrabble.java` contient la méthode `evaluate(List<Character> myLetters)` qui prend en entrée une liste de caractères et donne la valeur cumulée des points des lettres fournies. Donnez la complexité en pire cas de cette méthode en fonction de la taille de la liste de caractères `myLetters` (qu'on notera n) et la taille du dictionnaire `letterValues` (notée k). Justifiez adéquatement votre réponse.

O(n) 1/1

La complexité de contain() dans une map est de o(1) et c'est de même pour les get o(1). Donc, on tient plus compte de k.Alors la complexité asymptotique dépend seulement de la taille de myLetter qui est (n) ce qui veut dire que c'est automatiquement O(n).

Partie 2 :

L'Officiel Du Scrabble (ODS) est le dictionnaire faisant référence pour savoir si une combinaison de lettres est permise dans le jeu. Le dictionnaire vous est fourni sous la forme d'un fichier texte (`ods.txt`). Ce fichier est utilisé par la classe `ODS`, décrite dans `ODS.java`. Pour pouvoir compléter le travail, vous devez vous assurer que la classe ODS dispose du bon chemin vers le fichier texte. Au besoin, modifiez la ligne correspondante :

```
public class ODS {

   String fileName = "src/ods.txt"; // <= Donnez le chemin vers le fichier
```

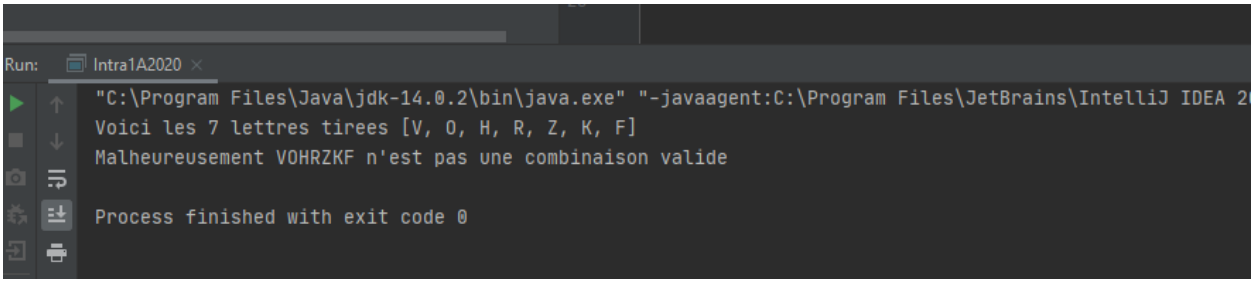
Également, pour les besoins de ce travail, la classe `QuadraticProbingHashTable` du livre de Weiss a été empruntée et modifiée. Notamment, elle implémente l'interface `Collection`, bien que certaines méthodes de l'interface n'aient pas été implémentées :

```
public class QuadraticProbingHashTable<AnyType> implements Collection<AnyType>
```

4) **(0.5 point)** Si vous donnez le bon chemin pour le fichier ods.txt et que vous mettez le booléen `partie2` à **`true`** comme suit, l'affichage du programme est modifié. Reproduisez l'affichage obtenu :

```
   public static void main(String args[]) {

      boolean partie2 = true; // <= Mettre à true pour la partie 2
```

0,5/0,5



5) **(1 point)** Donnez la complexité en pire cas de la méthode `contains(String str)` de la classe `ODS`. Justifiez adéquatement votre réponse.

Contrains(String str) fait appelle à la fonction isActive() qui est de complexité O(1). De plus, cette dernière fait appelle à la méthode findPos(object). Lorsqu'on va chercher findPos(), on constate que cette dernière contient une boucle while.Le pire cas étant celui où on cherche l'élément à partir d'une certaine position, mais qu'à partir de cette dernière, il existe beaucoup de collisions donc on sera obligé de revenir au début de la table et continuer à chercher. Ceci fait en sorte qu'on parcours presque la table au complet. Donc au pire cas la complexité est de O(n) pour findPos(). Alors, contains() est de complexité O(n).

1/1

La complexité est celle de contains de la classe QuadraticProbingHashTable, soit O(1).

6) **(1 point)** La classe `QuadraticProbingHashTable` définit la classe imbriquée `QphtIterator` qui implémente l'interface `iterator`. Cette classe a été partiellement implémentée pour vous. Donnez la complexité en pire cas de la méthode `next()` de la classe `QphtIterator` en fonction du nombre d'éléments présents dans la table (noté k) et la taille de la table, soit `array.length`, notée n pour cette question. Justifiez adéquatement votre réponse.

La fonction isActive() permet de rentrer dans le if seulement si l'élément est actif. Alors, le if se répète (k) fois ou k est la size de la table (case remplies et actives). Par contre, la boucle while implique qu'on entre dans celle-ci (n) fois dépendamment de ce que le hash à donner comme index. Donc la boucle while se fait n fois quand même. Alors il n'est même pas nécessaire de tenir compte de k. la complexité est automatiquement O(n).

1/1

7) **(1 point)** Peut-on améliorer la complexité algorithmique de la méthode `next()` discutée à la question 6 ? Si oui, dites comment, sinon justifiez.

Je pense que la complexité algorithmique de cette méthode est à son meilleur. On peut donc pas améliorer cette dernière. Vu qu'on a besoin de return un élément actif on a besoin d'une boucle qui va parcourir tous les éléments afin de savoir si ces derniers sont bel et bien actifs ou non. Donc le fait de parcourir, une liste de (n) élément implique automatiquement une complexité de O(n). Il n'est pas possible de réduire la complexité, car il faut parcourir les éléments pour savoir lequel est actif ou non.

0/1

Oui, si les indices des positions des éléments sont conservées en mémoire.

8) **(1 point)** La méthode `remove()` de la classe `QphtIterator` est incomplète. Proposez-en une ci-après:

```
public void remove( )
{
    if( !okToRemove )
        throw new IllegalStateException( );

    QuadraticProbingHashTable.this.remove(array[current-1].element);

}
```

1/1

Partie 3 :

On désire implémenter dans une classe de joueur (`Player`) un algorithme qui renvoie toutes les combinaisons pouvant être formée pour l'ensemble des lettres tirées . Pour ce faire, l'algorithme exploite des méthodes statiques disponibles dans la class `Util` (`Util.java`).

L'algorithme proposé procède comme suit :

Partant des lettres disponibles

1. Il construit toutes les sous-chaines possibles de manière récursive.
2. Pour chaque sous-chaine, il détermine toutes les permutations possibles de ses lettres de manière récursive.

La classe Player retourne toutes ces combinaisons à l'appel de sa méthode :

```
public Collection<String> getCombinations(boolean useHashTable)
```

qui retourne une `Collection` qui est soit un `ArrayList`, soit une `QuadraticProbingHashTable`, selon la valeur du booléen qu'elle reçoit en paramètre.

9) **(1 point)** Mettez le booléen `partie3` à **`true`**. Comparez les affichages obtenus selon que le booléen `useHashTable` soit à **`true`** ou à `false`. Décrivez les différences que vous observez et donnez en une explication.

Lorsqu'on met le Bolean useHashTable = true; on active l'utilisation de la hash table. Donc c'est elle qui sera utilisée comme contenant. Ainsi, on obtient 6 combinaisons sans aucune répétition. En voilà la preuve :

```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 202
Voici les 7 lettres tirees [V, O, H, R, Z, K, F]
Malheureusement VOHRZKF n'est pas une combinaison valide
Vous pourriez essayer de jouer:
Mot:    HO  Valeur: 5
Mot:    OH  Valeur: 5
Mot:    OR  Valeur: 2
Mot:    FOR Valeur: 6
Mot:    ROKH    Valeur: 16
Mot:    RHO Valeur: 6
```

Ainsi, lorsqu'on va dans la classe QuadraticProabingHashTable ensuite dans la fonction add() de celle-ci, on remarque que celle-ci n'ajoute pas un élément si ce dernier est déjà présent. C'est pour cela qu'on voit seulement les 6 combinaisons possibles sans aucune répétition. 1/1

Dans le cas contraire, lorsqu'on met le Bolean useHashTable = false; la fonction getCombinations() utilise un arrayList < > comme contenant. Or, ce dernier sa fonction add() ajoute tous les éléments qu'il soit présent déjà ou non, ce dernier est quand même ajouté. C'est pour cela qu'on obtient énormément d'éléments. Voici une mini-capture d'écran :

Partie 4 :

Pour cette partie, on désire trouver le meilleur choix parmi toutes les combinaisons valides trouvées. Tous les mots valides ont été insérés dans un tableau avec leur valeur. On vous demande d'appeler un algorithme de tri parmi trois tirés du livre de Weiss : `insertionSort`, `mergeSort` ou `quickSort`.

10) **(1 point)** Quel est selon vous l'algorithme de tri le plus performant pour le problème traité ? Justifiez clairement votre réponse. 1/1

Dans ce cas-ci, on a pas énormément de données. En effet, on a 6 combinaisons et parmi celles-ci on doit trouver celle qui en vaut le plus. Donc la size de ma hashTable est de 6 données.

Allons par élimination. D'abord le merge sort, cette façon de trier à une vitesse de tri constante et ce pour n'importe quelle taille. Elle opère en faisant usage d'un tableau externe. De plus, cette méthode est souvent privilégiée quand on a des Linked List, ce qui n'est pas le cas dans cet intra .Ainsi, on l'élimine, malgré qu'il offre la meilleure complexité (O(n log n)) en pire cas, car c'est un petit tableau a trier et on ne veut pas avoir un rendement pareil au petit tableau.

Quant au quick Sort, ce dernier offre une complexité O(n log n) en meilleur cas et O(n*n) en pire cas. Il est utilisé de préférence pour les Arrays et plus précisément pour les Arrays de petite taille (size >20) puisqu'il est rapide pour ces derniers. Or ce dernier fait appelle à la récursivité ce qui ralenti la vitesse de tri pour mon petit tableau. Donc, on l'élémine lui aussi.

Donc, l'idéal c'est le tri par insertion et ce, même si cela représente une complexité en pire cas de O(n*n). En fait, pour des petits tableaux (avec un nombre d'éléments <= 20)ce dernier est plus rapide que le quickSort, puisqu'il ne fait pas appel à la récursivité afin de découper le tableau. En fait, on économise 15 pourcents du temps en utilisant le tri par insertion selon le livre de Weiss chapitre 7 section 7.7.3 (Small Arrays). Page 294.

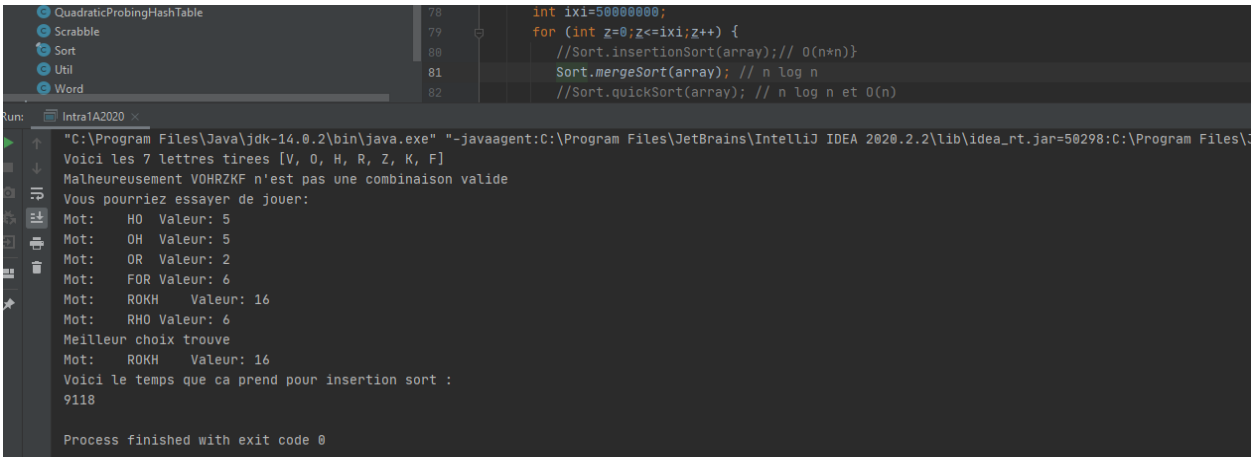
Voici maintenant des preuves un peu plus pratiques. Pour prouver cela j'ai du créée deux variables de type long (tempsDebut et tempsFin) d'avoir le delta temps que prend la partie 4 lors de l'appel de la fonction de tri. J'ai aussi inséré une boucle qui se répète 50 000 000 lorsqu'on appel la fonction de tri tel que la suite :

```
long tempsDebut = System.currentTimeMillis();

   Word[] array = new Word[validCombinations.size()];
   int i = 0;
   for (Word word : validCombinations)
      array[i++] = word;

   System.out.println("Meilleur choix trouve");
   // Décommentez une des options suivante
int ixi=50000000;
for (int z=0;z<=ixi;z++) {
   Sort.insertionSort(array);// O(n*n)}
   //Sort.mergeSort(array); // n log n
   //Sort.quickSort(array); // n log n et O(n)
}
long tempsFin = System.currentTimeMillis();
long temps = tempsFin-tempsDebut;
System.out.println( array[array.length-1] );
System.out.println("Voici le temps que ca prend pour insertion sort : ");
System.out.println(temps);
```

Ce test a été effectué pour les trois différentes façons de tri et voici les résultats dans l'utilisation du tri par insertion :

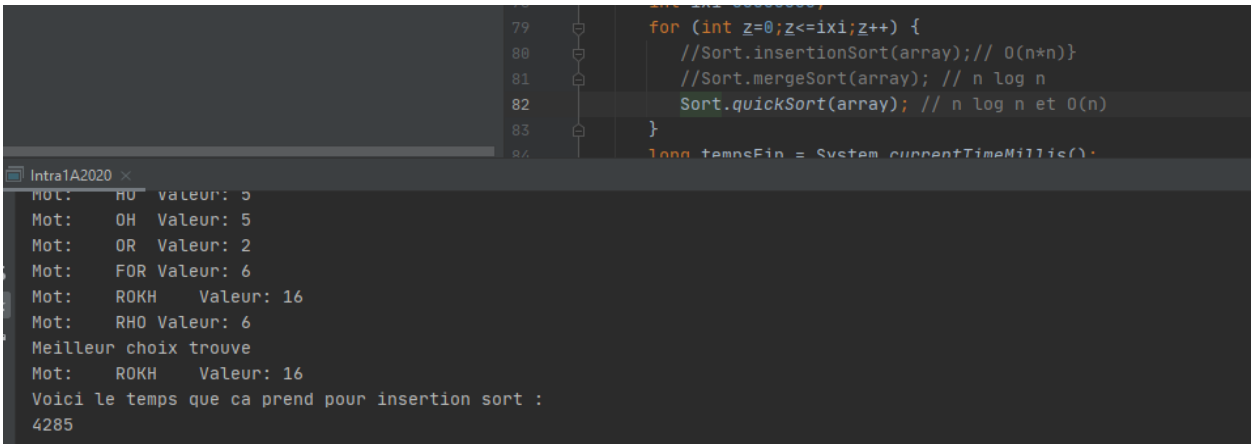
911 ms.

Test pour merge sort :



9118 ms.

Test pour quick sort :



4285 ms.

Tableau 1 : Moyennes pour chaque type de tri en ms:

Essaie	insertionSort (ms)	mergeSort (ms)	quickSort (ms)
1	864	9932	2797
2	859	9962	2781
3	853	10344	2743
4	849	9253	2774
Moyenne	856	9873	2774

Donc, en moyenne, c'est insertion sort qui offre le meilleur temps.

Enfin, ce que j'ai fait, c'est compter combien de fois est utilisée chaque fonction de tri et ce pour chaque tri. Voici un exemple de code:

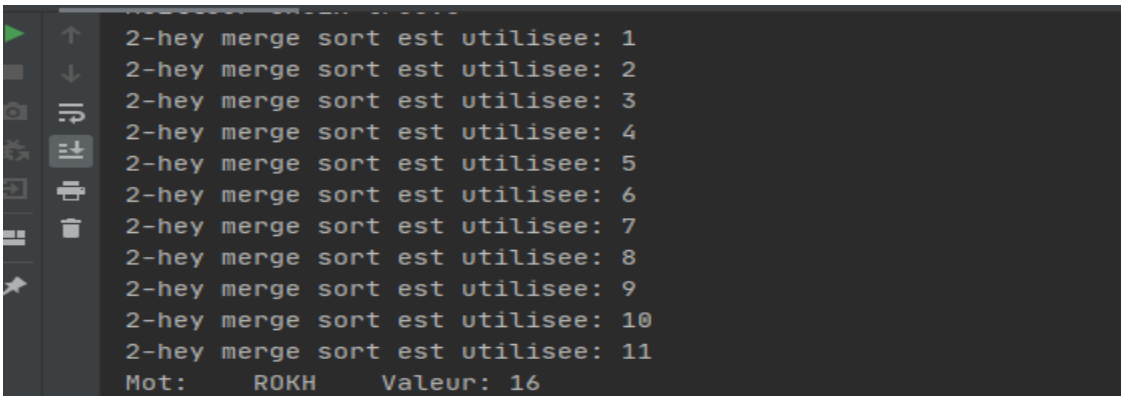
```
void mergeSort( AnyType [ ] a, AnyType [ ] tmpArray, int left, int right )
{
System.out.println("2-hey merge sort est utilise"+ compteurRecursif++);///ici
   if( left < right )
   {
      int center = ( left + right ) / 2;
      mergeSort( a, tmpArray, left, center );
      mergeSort( a, tmpArray, center + 1, right );
      merge( a, tmpArray, left, center + 1, right );
   }
}
```

Donc, la variable compteurRecursif compte combien de fois chaque fonction est appelée et elle affiche cela a l'écran. Cette ligne se répète pour chaque méthode de tri qu'il y a dans la classe Sort.

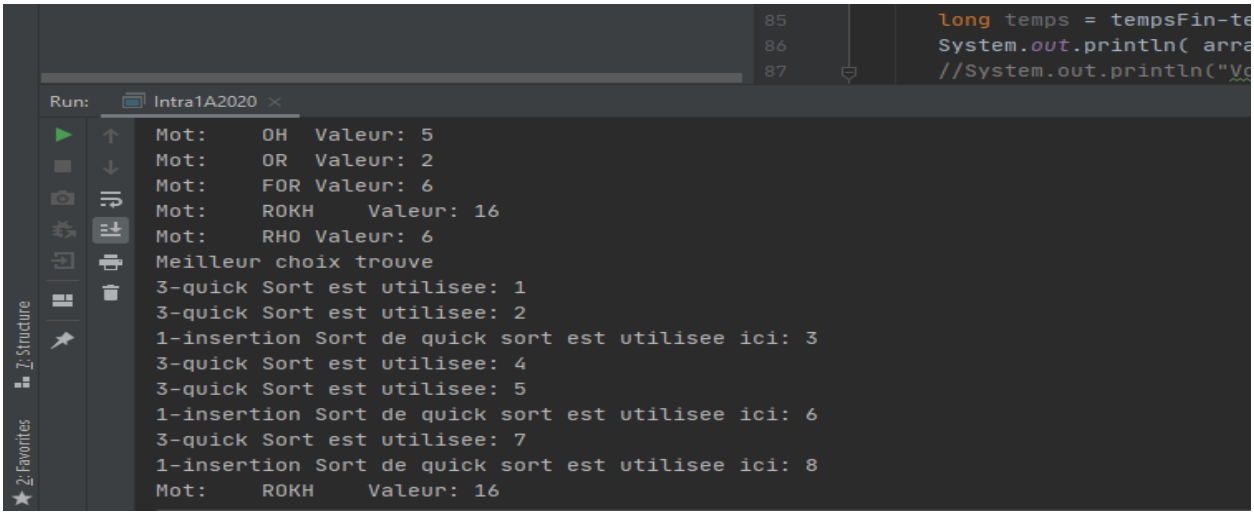
Voici l'affichage pour insertionSort :

```
Mot:    ROKH    Valeur: 16
Mot:    RHO Valeur: 6
Meilleur choix trouve
1-1 insertion sort est utilisee ici : 1
Mot:    ROKH    Valeur: 16

Process finished with exit code 0
```

L'affichage pour mergeSort :



L'affichage pour quickSort :



En conclusion, insertionSort est appellee une fois puisqu'elle n'utilise pas de récursivité. Quant à mergeSort(), la fonction est appelée 11 fois ce qui explique l'énorme temps que ça prenais (9873 ms). quickSort(), elle est appelée 8 fois et sur les fois on traite le pire cas en faisant appelle à la fonction insertionSort() 3 fois. Cela démontre aussi pourquoi on a obtenu (2774 ms) en temps moyen pour 50 000 000 que cette fonction est exécutée. Ce qui est plus élevé que insertionSort() (856 ms).

11) **(1 point)** Mettez le booléen `partie4` à **`true`** et décommentez l'option de tri choisie. Quel est la meilleure combinaison de lettres que vous trouvez ? Le résultat vous semble-t-il logique? Justifiez brièvement.

La meilleure combinaison de lettre que j'ai trouvée est ROKH et c'est de valeur 16. Le résultat est tout à fait logique puisqu'on cherche la combinaison qui donne la plus grande valeur possible. Ainsi, selon le tri choisi on va chercher l'élément avec la valeur la plus grande.

1/1

Sortie pour quickSort | | mergeSort | | insertionSort :

```
Mot:    OH  Valeur: 5
Mot:    OR  Valeur: 2
Mot:    FOR Valeur: 6
Mot:    ROKH    Valeur: 16
Mot:    RHO Valeur: 6
Meilleur choix trouve
Mot:    ROKH    Valeur: 16
```