Rapport
IFT 2015 : Devoir 1

Absalon Marion (20211423) et Sooben Vennila (20235256)

12 juin 2023

1 Auto-évaluation

Évaluation du fonctionnement du programme: Correct

Notre code fonctionne correctement sauf preuve du contraire.

1.1 Description du code

Nous avons implémenté une manière de gérer un système de cargaison en Java en utilisant les structures de données suivantes: **Queue** et **ArrayList**.

Notre logique se base sur l'index des *ArrayLists* et du placement du premier élément (on exploitera le principe de FIFO) afin d'utiliser des *Queues*

Analyse Amortie : Nous avons pris la décision de s'autoriser à faire certaines opérations "coûteuses" afin d'ameliorer l'efficacité. En effet, pour pouvoir une meilleure complexité temporelle, nous avons fait le choix de rendre notre complexité spatiale "moins" bonne notamment en créant plusieurs structures de données.

Voici une explication de ce que fait notre code:

1. Lire le fichier passé en paramètre et d'utiliser des techniques de *string manipulation* afin d'en extraire les données. Ces données sont ensuites ajoutés à des *ArrayLists* ou des variables. Nous avons exploité les fonctionnalités d'OOP de Java notamment l'encapsulation.
2. L'étape suivante est de chercher le maximum de boîtes dans l'*ArrayList* car cette position sera la position du camion tout au long de la cargaison. On retournera l'index ainsi de savoir et la position et le nombre de boîtes des 2 *ArrayLists*.

3. On vérifie ensuite si le camion sera déja remplie, si oui on s'arrête ou sinon on continue.

4. Si on continue, la prochaine étape est de calculer les distances entre notre position actuelle et tous les autres entrepôts. La formule d'Haversine est utilisée. Ensuite nous trions le tableau, mais sans le trier en mémoire. Nous effectuons le triage en se basant sur les index uniquement ainsi nous pouvons egalement trier les autres structures de données par rapport à la croissance des distances. Nous ajoutons les index dans un *ArrayList*

5. La prochiane étape est de **get(index)** les elements de l'*ArrayList* Index et ajouter les elements correspondants de chaques *ArrayLists* dans des *Queues*.

6. Maintenant que tout est trié correctement nous commençons l'opération de vider les entrepôts et remplir le camion en utilisant une fontion auxilliaire *calculation()*. On s'arrête quand soit, le camion est rempli, ou les entrepôts sont vides.

7. A chaques itérations de la boucle, notre fichier est modifié.

1.2 Tests effectués

Nous avons testés avec les fichiers mis à notre disposition sur StudiuM.

2 Analyse de la complexité temporelle (pire cas) théorique en notation grand O

2.1 Analyse pire cas PseudoJava

- $O(1)$ pour remove

```
ManipulationFichier fileManip = new ManipulationFichier();

fileManip.readFile(args[0], args[1] ,truck);

maxBoxIndex = maxBox(fileManip.getArrayBox());
maxBox = fileManip.getArrayBox().get(maxBoxIndex);

Coordinates positions = fileManip.getArrayCoord().get(maxBoxIndex);
fileManip.getArrayCoord().remove(maxBoxIndex);
```

Figure 1: Bout de code dans le main permettant de vérifier si le fichier de destination est vide

- $O(1)$ pour remove

```
ManipulationFichier fileManip = new ManipulationFichier();

fileManip.readFile(args[0], args[1] ,truck);

maxBoxIndex = maxBox(fileManip.getArrayBox());
maxBox = fileManip.getArrayBox().get(maxBoxIndex);

Coordinates positions = fileManip.getArrayCoord().get(maxBoxIndex);
fileManip.getArrayCoord().remove(maxBoxIndex);
```

Figure 2: Bout de code dans le main permettant d'écrire la position initiale du camion dans le fichier de destination

- $O(n)$ pour la boucle
- $O(1)$ pour le reste
  $\Rightarrow O(n)$

```
 for (int j = 0; j < fileManip.getArrayCoord().size(); j++) {
    dist = calculeDistance(positions,
        fileManip.getArrayCoord().get(j));
    arrayDistTemp.add(dist);
    arrayDistTempCopy.add(dist);
}
```

Figure 3: Boucle dans le main ajoutant les distances calculées à l'ArrayList de distance et sa copie

- $O(n)$ pour la boucle
- $O(1)$ pour le reste
  $\Rightarrow O(n)$

```
  for (int j=0; j<lengthArrBox;j++){
   int minIndex = maxmind("min", arrayDistTempCopy);
   arrayDistTempCopy.set(minIndex, compa );
   indexArrayList.add(minIndex);
}
```

Figure 4: Boucle dans le main permettant de trier les distances dans l'ordre croissant

- $O(n)$ pour une boucle $\Rightarrow O(n^2)$ pour les boucles (imbriqué)
- le reste $O(1)$
  $\Rightarrow O(n^2)$

```
            for (int d=0; d<lengthArrBox; d++){
             for (int e=d+1; e<lengthArrBox;e++){
                 if (compar == elementBefore){
                     if (elementBeforeLat == elementAfterLat){
                         if (elementBeforeLong > elementAfterLong){
                             int temp = elementAfter;
                             //some setters here
                         }
                     }else if (nearly same exp here as if above){nearly
                          same operations here as if above}
                     }
                 }
             }
        }
```

Figure 5: Code dans le main permettant de choisir les entrepôts en cas de distances égales

- $O(n)$ pour une boucle
- $O(1)$ pour le reste
  $\Rightarrow O(n)$

```
            for (int k = 0; k < lengthArrBox; k++) {
             int index = indexArrayList.get(k);
             queue.add(arrayDistTemp.get(index));
             queueBox.add(fileManip.getArrayBox().get(index));
             queueCoord.add(fileManip.getArrayCoord().get(index));
        }
        truck.setqueueDist(queue);
        truck.setqueueBox(queueBox);
        truck.setqueueCoord(queueCoord);

        int sizequeue= truck.getQueueCoord().size();

        for (i = 0; i < sizequeue; i++) {
            if (truck.getContentTruck() > 0 &&
                truck.getqueueBox().isEmpty()==false) {
                positions = calculation(fileManip, args[1], truck ,
                    positions);
            }
        }
        }
```

Figure 6: Code avec une boucle permettant d'ajouter des données dans la file et boucle qui fait le travail de remplir le camion et vider l'entrepôt; code dans le main() procédure

- $O(n)$ pour la boucle
- $O(1)$ pour le reste
  $\Rightarrow O(n)$

```
        public static int maxmind(String maxOrMin, ArrayList<Double>
            tab){

        double comp=tab.get(0);
        int ind=0;

        for (int i=1; i<tab.size(); i++){
            if (maxOrMin=="max"){
                if (comp<tab.get(i)){
                    comp = tab.get(i);
                    ind = i;
                }
            }else if (maxOrMin=="min"){
                if (comp>tab.get(i)){
                    comp=tab.get(i);
                    ind=i;
                }
            }
        }
        return ind;
    }
```

Figure 7: Fonction maxmind (presque similaire à maxmin) retournant un index correspondant à l'élément maximum ou minimum du tableau

- $O(1)$ remove
- $O(min(ob.getContentTruck(), box))$ au pire cas, si ob.getContentTruck() et box sont initialement > 0, itération min(ob.getContentTruck(), box) fois
  $\Rightarrow O(n)$

```
        public static Coordinates calculation(ManipulationFichier
            fileManip, String namefile, GestionCamion ob, Coordinates
            positions ){

        int minDistIndex=0;

        double minDist=ob.getQueueDist().remove();
        int box= ob.getqueueBox().remove();
        Coordinates pos = ob.getQueueCoord().remove();

        while (ob.getContentTruck() >0 && box > 0) {
            ob.setContentTruck((ob.getContentTruck() - 1));
            box=box-1;
        }

       //some outputs here

        positions = pos;

        return positions;
    }
```

Figure 8: Fonction calculation qui effectue l'opération de vider les entrepôts et accessoirement remplir le camion

- $O(n)$ pour parse
- $O(n)$ pour itérations de boucle
- $O(1)$ pour le reste
  $\Rightarrow O(n)$

```
    public static void setData(String writefilestr, String s,
        GestionCamion c1, String input1){

    ManipulationFichier inputData = new ManipulationFichier();

    try {
        BufferedWriter deletef = new BufferedWriter(new
            FileWriter(writefilestr, false));
        deletef.close();
    } catch (IOException e) {
        System.out.println("Erreur dans l'ecriture du fichier");
    }

    Pattern pattern = Pattern.compile("(\\d+)
        \\((-?\\d+\\.\\d+),(-?\\d+\\.\\d+)\\)");
    Matcher matcher = pattern.matcher(s);

    ArrayList<Integer> box = new ArrayList();
    ArrayList<Coordinates> positions = new ArrayList();
    while (matcher.find()) {
        int number = Integer.parseInt(matcher.group(1));
        double latitude = parseDouble(matcher.group(2));
        double longitude = parseDouble(matcher.group(3));

        box.add(number);

        Coordinates ob1 = new Coordinates(longitude,latitude);

        positions.add(ob1);

    }

    String[] numbers = input1.split("\\s+");
    int firstNumber = Integer.parseInt(numbers[0]);
    int secondNumber = Integer.parseInt(numbers[1]);

    if (firstNumber>secondNumber){
        c1.setContentTruck(secondNumber);
    }else{
        c1.setContentTruck(firstNumber);
    }

    inputData.setArrayBox(box);
    inputData.setArrayCoord(positions);

}
```


Figure 9: Fonction setData qui sépare les données correctement du fichier avec des expressions régulières

- $O(n)$ pour la lecture (Scanner)
- $O(n)$ pour chaques iterations de boucle
- $O(1)$ pour le reste
  $\Rightarrow$ Max $= O(n)$

```
    public static void readFile(String nameFile, String
        writefilename, GestionCamion c1){

    try {
        //Fetches the file and reads it
        File file = new File(nameFile);
        Scanner FileReader = new Scanner(file);
        String lines = "";
        String input1 = FileReader.nextLine();

        //Converts the lines into strings
        while (FileReader.hasNextLine()) {
            String input = FileReader.nextLine();
            String aLine = input + "/n";
            lines = lines + aLine;
        }
        FileReader.close();
        setData(writefilename,lines, c1, input1);
    } catch (FileNotFoundException e) {
        System.out.println("File not found.");
    }

}
```

Figure 10: Fonction pour lecture du fichier

- $O(n)$ pour l'écriture du fichier (n lignes)
- $O(1)$ pour le reste
  $\Rightarrow O(n)$

```
    public static void writeFile(String writefilename, String out) {
    try {
        BufferedWriter outputs = new BufferedWriter(new
            FileWriter(writefilename, true));
        outputs.write(out+"\n");
        outputs.close();
    } catch (IOException e) {
        System.out.println("Erreur dans l'ecriture du fichier");
    }
}
```

Figure 11: Fonction pour l'écriture du fichier

3 Analyse de la complexité temporelle empirique

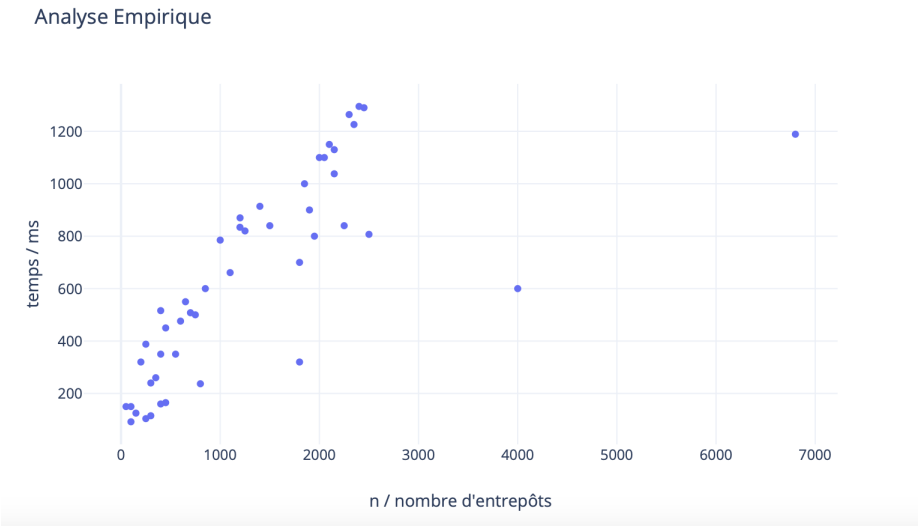
3.1 Graphique de temps d'exécution en fonction de n

L'analyse a été effectuée avec

```
long endTime = System.currentTimeMillis();
System.out.println(endTime - startTime);
System.out.println(Integer.toString(lengthArrBox));
```

Nous avons testés avec les fichiers mis à notre disposition sur StudiuM. Afin d'augmenter notre n, nous avons également fait des fusions de fichiers.

- Utilisé avec $50 < n < 6800$
- On peut apercevoir la boucle de $O(n^2)$



Nous avons testés avec un code Java créant des données et l'outil JFreeChart.

- Utilisé avec $60 < n < 50000$
- On peut apercevoir la boucle de $O(n^2)$ encore une fois.