Farah Mohamed, Matricule: **20246646**
Hervé Ng'isse, Matricule : **20204609**

TP2 2015, 29 juillet 2023.

1. Auto-évaluation:

Notre code fonctionne correctement. En effet, tous les tests effectués sur les fichiers d'entrée ont produit les mêmes résultats que les exemples de sorties.

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

2.1.main:

V est le nombre de sommets dans le graphe.
E est le nombre d'arêtes(rues) dans le graphe.

- **Lecture fichier**

-la lecture du fichier et l'initialisation des variables ont une complexité temporelle **O(1)**
-les deux boucles qui ajoutent à chaque rue (arête) ses deux sommets(de départ et d'arrivé) ont a eux deux une complexité temporelle de **O(E).**

```
while ((line = bufferedReader .readLine()) != null) {
    //si une ligne  commence  par une lettre et que la longueur de la ligne >=6
    // alors c'est surement une rue (une arête)
    if (line.length() <= 4 && (Character.isLetter(line.charAt(0)))){
        line = line.replace( target: " ", replacement: "");
        Sommet sommet= new Sommet(line,  visited: false, new ArrayList<Rue>());
        listSommet.add(sommet);
    }
    else if(Character.isLetter(line.charAt(0))) {
        line = line.replace( target: ";",  replacement: "");
        line = line.replace( target: ":",  replacement: "");
        line = line.trim().replaceAll( regex: "\\s+",  replacement: " ");
        String[] newLine = line.split( regex: " ");
        String nomArete = newLine[0];
        Sommet sommetDepart = new Sommet(newLine[1],  visited: false);
        Sommet sommetArrivee = new Sommet(newLine[2],  visited: false);
        int poidsArete = Integer.parseInt(newLine[3]);
        Rue rue = new Rue(nomArete,sommetDepart,sommetArrivee,poidsArete);
        //ajoute a chaque rue (arête)ses deux sommets(de départ et d'arrivé)
        for (Sommet sommet: listSommet) {
            if (sommet.getSommet().equals(sommetDepart.getSommet())){
                sommet.addRue(rue);
                rue.setSommetDepart(sommet);
                break;
            }
        }
        for (Sommet sommet: listSommet) {
            if (sommet.getSommet().equals(sommetArrivee.getSommet())){
                sommet.addRue(rue);
                rue.setSommetArrivee(sommet);
                break;
            }
        }
        listArete.add(rue);
    }
}
```

- **l'algorithme de Prim-Jarnik**

- Insertion dans la File de Priorité **O(V)**
-Extraction de l'élément de la file de priorité : **O(log V)**
-les opérations des vérifications de la disponibilité de sommet et l'initialisation de la disponibilité du sommet,l'écriture dans le fichier de sortie ont une complexité temporelle : **O(1)**

- l'Ajout des arêtes non visitées dans la file de priorité a une complexité temporelle : **O(E log V)**

Donc l'algorithme de Prim-Jarnik a une complexité temporelle de **O(E log V)**

```
//le sommet de depart choisit est le premier élément de notre liste de sommets
Sommet startSommet = listSommet.get(0);
startSommet.setVisited(true);
PriorityQueue<Rue> ruePriorityQueue = new PriorityQueue<>();
//on ajoute tous les arêtes de notre sommet de départ à notre priorityQueue en suivant une  priorité
//lié au poids ensuite a l'ordre alphanumerique
for (Rue rue: startSommet.getRueConnecte()){
    ruePriorityQueue.add(rue);
}
int compteur = 0;
int poids = 0;
//implementation l'algorithme de Prim-Jarnik.
while (!(ruePriorityQueue.isEmpty()) ) {
    Rue ruePrise = ruePriorityQueue.poll();
    compteur +=1;
    //si les deux sommets sont déjà visités la condition permet d'ignorer l'arête
    if ((ruePrise.getSommetArrivee().isVisited()) && (ruePrise.getSommetDepart().isVisited())) {
        continue;
    }
    //changer la valeur du sommet courant en True
    ruePrise.setVisited(true);
   try (BufferedWriter writerSommet = new BufferedWriter(new FileWriter(outputFile,  append: true))) {
       //la condition permet écrire les sommets selon leurs ordres de lecture et si un sommet de depart
       //est déjà lu alors on écrit le sommet d'arrivé
       String printAreteString = ruePrise.getSommetDepart().getSommet();
       boolean b = !(printArete.contains(printAreteString));
       if (b) {
            printArete.add(printAreteString);
            writerSommet.write(printAreteString);
            writerSommet.newLine();
       }else{
           printAreteString=ruePrise.getSommetArrivee().getSommet();
           printArete.add(printAreteString);
           writerSommet.write(printAreteString);
           writerSommet.newLine();
       }
    } catch (IOException e) {
    throw new RuntimeException(e);
    }
   //instancie avec un constructeur vide le prochain sommet pour éviter de prendre un sommet déjà
   //visité
    Sommet prochainSommet = new Sommet();
```

```
    //instancie avec un constructeur vide le prochain sommet pour éviter de prendre un sommet déjà
    //visité
     Sommet prochainSommet = new Sommet();
    //condition pour vérifier si le prochain sommet est déjà visité
     //si oui prendre le sommet d'arrivé
     if (ruePrise.getSommetArrivee().isVisited()) {
         prochainSommet = ruePrise.getSommetDepart();
     } else {
         prochainSommet = ruePrise.getSommetArrivee();
     }
     //changer la valeur du sommet de départ en True
     prochainSommet.setVisited(true);
     //les arête sont les rues ainsi on ajoute dans la priorityqueue,les rues qui n'ont pas encore été
     //visité avec comme sommet courant,le prochain sommet de notre sommet precedent
     for (Rue rue: prochainSommet.getRueConnecte()){
         if (!(rue.isVisited())){
             ruePriorityQueue.add(rue);
         }
     }
     //itération du chemin déjà parcouru
     poids += ruePrise.getPoidsArete();
     //ajoute les informations lies a la rue dans une liste afin d'obtenir une sortie:
     //nom sommet + rue a chaque itération
     //mais sommets  ensuite toutes les rues
     outputPrint.add(ruePrise.getNomArete() +"  "+ ruePrise.getSommetDepart().getSommet() +
             "  "+ ruePrise.getSommetArrivee().getSommet()+ "  " + ruePrise.getPoidsArete());
}
```

- **Ecriture des rues et de la longueur du parcours**

cette opération dépend du nombre d'éléments dans la liste des rues à écrire plus la longueur du chemin donc la complexité temporelle est **O(E).**

```
try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile,  append: true))) {
    for (String element : outputPrint) {
        writer.write(element);
        writer.newLine();
    }
    writer.write( str: "-----");
    writer.newLine();
    writer.write(String.valueOf(poids));
```

- **complexite temporelle globale**

$O(E) + O(E \log V) + O(E)$,la complexite temporelle totale du code dans le pire cas est **$O(E \log V)$**