1. Auto-évaluation :

Le programme fonctionne correctement.

1. **Analyse théorique de la complexité**

L’analyse théorique de la complexité sera présentée une fonction à la fois, selon l’exécution du programme dans le pire cas. Nous considérons une entrée de n bâtiments. La méthode que nous avons choisie est d’additionner les O(n) de chaque fonction majeure (ayant un temps non constant) afin de trouver le O(n) de l’ensemble du programme. Vous trouverez le nombre d’opérations des méthodes écrit en commentaire dans le code, à l’exception des temps constants.

D’abord, la première fonction importante appelée par *Main* est *parse(InputFile)*. Cette fonction fait des opérations constantes, jusqu’à ce qu’elle entre dans une boucle while. Le nombre d’opérations est d’approximativement 6n, ce qui donne O(n).

Ensuite, *Main* appelle *initialLoad(truck, warehouse)*. Cette fonction appelle par la suite *warehouse.maxBoxes()* afin de trouver le bâtiment avec le plus de boîtes. Dans le pire cas, *maxBoxes()* passe au travers n-1 bâtiments dans sa boucle for, ce qui donne O(n). Les autres opérations de *initialLoad(truck, warehouse)* sont en temps constant.

*distanceTruckBuildings(truck, ArrayList buildings)* est la fonction suivante appelée par *Main*. Cette fonction contient une boucle for qui passe au travers tous les bâtiments, donc elle est de O(n).

Après, c’est *warehouse.quickSortDistance(buildings, 0, lowIndex, highIndex)* qui est appelée par *Main*. Il s’agit d’un tri rapide (quick sort) traditionnel qui tri les éléments selon leur distance au camion. Le tri rapide consiste à choisir un pivot et à permuter les éléments du tableau pour que ceux ayant une distance inférieure au pivot se retrouvent à sa gauche, et ceux ayant une distance supérieure soient à droite. Ensuite, quick sort est appelé récursivement sur les deux sous-tableaux formés par les éléments inférieurs et les éléments supérieurs au pivot. Si le pivot sélectionné est la plus grande ou la plus petite distance, un des sous-tableau sera de taille n-1. Si cela se produit à chaque appel récursif, *quickSortDistance* sera appelé avec n-1, n-2, n-3,… éléments à chaque fois pour le grand sous-tableau, et avec 1 pour le petit sous-tableau. Comme il y a n/2 appels de quick sort avec n-1, n-2, n-3,… nous obtenons n((n-1)/2) et nous pouvons ignorer les n/2 appels avec 1 qui sont insignifiants. Ainsi, par distribution, nous obtenons (n2-n)/2, ce qui donne O(n2) dans le pire cas. Heureusement, le cas moyen du tri rapide est O(nlogn).

Par la suite, *Main* appelle n fois *warehouse.closestBuildingToTruck(buildings)* dans le pire cas afin de remplir le camion. En effet, dans le pire cas, le camion devra passer par les n bâtiments pour être rempli. Cette fonction appelle une fois une autre méthode : *findNextUnvisitedBuilding(buildings)* qui peut passer au travers les n bâtiments dans une boucle while. Après son retour à *closestBuildingToTruck*, cette fonction exécute une boucle while qui, dans le pire cas, passe au travers n éléments. Comme cette méthode, est d’ordre n, nous pouvons conclure qu’elle est O(n). Nous avons donc que *Main* appelle n fois une fonction, ce qui donne O(n2) pour cette portion du programme.

Finalement, *Main* appelle *loadOutPut(outputFilePath)* qui retire les bâtiments visités de la file afin d’afficher leurs informations dans un fichier. Dans le pire cas, tous les n bâtiments ont été visités et doivent être retirés de la pile, pour un O(n).

En conclusion, en regardant l’analyse théorique de toutes les fonctions, nous remarquons que les plus significatives sont deux méthodes de O(n2). La complexité du programme est donc quadratique puisqu’il est O(n2).

1. Analyse Expérimentale :