

Rapport du devoir #1 :

Système de gestion de cargaison de camions d’un entrepôt

 Scénario 1

Par

 Steven Han, 20129462

 Yassine Hajouji Idrissi, 20038060

et

Chen Zong Jiang, 20122046

Département d’informatique et de recherche opérationnelle

Travail présenté à Samuel Ducharme

Dans le cadre du cours IFT-2015 section A

Structures de données

12 Juin 2019

Auto-évaluation

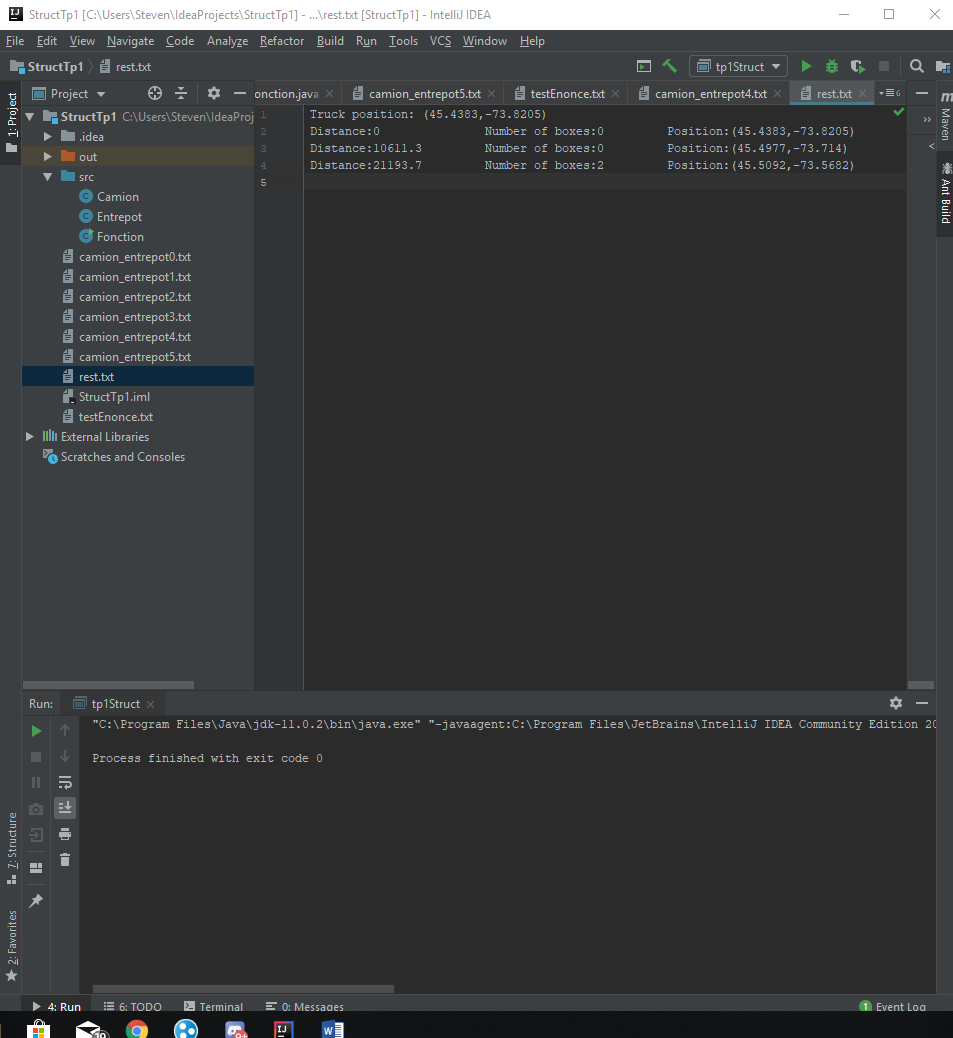
Notre programme fonctionne correctement, car en premier lieu, on trouve bel et bien le max de chaque entrepôt. Voici la position de chaque camion pour chaque entrepôt avec les fichiers des inputs données. Notez que vous pouvez tester le programme pour voir les résultats, sauf s’il y a des erreurs de format.

Sortie de la position du camion exécuté par notre programme

|  |  |
| --- | --- |
| Entrepôts | Truck position (m) |
| 0 | (45.5532,-73.5968) |
| 1 | (45.5766,073.5695) |
| 2 | Erreur |
| 3 | (45.4431,-73619) |
| 4 | (45.554,-73.5363) |
| 5 | (45.5449,-73.5891) |

En vérifiant chaque fichier à l’œil nu et en se basant sur le nombre de boites maximale de chaque entrepôt, on voit bien que le programme sort bel et bien le bon positionnement du camion.

Pour ce qui est de la distance, c’est un peu plus difficile de vérifier, mais si on met le fichier de l’énoncé, on retrouve exactement les mêmes champs.



De plus, on a géré les cas où le format du fichier n’est pas bon (le nombre de boites à livrer dépasse la capacité du camion, le manque d’espace entre les entrepôts et entre le nombre de boite et la coordonné de l’entrepôt).

Il est à noter que notre programme fonctionne s’il y a présence d’un espace à la fin de chaque ligne, car c’est notre indicateur de séparation des paramètres.

Dans l’éventuel que les fichiers futurs ne contiennent pas d’espaces en fin de ligne, il y a une fonction tout en bas du code qui s’intitule mettreEspaceFin qui sera utile.

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

Pour faire cette analyse, on s’attardera aux grosses fonctions du programme (on oublie les déclarations de variables, les accesseurs/mutateurs, les constructeurs et les petites fonctions). Voici le pseudo-code des fonctions utiles :

**Fonction qui prend le fichier contenant les informations et crée un ArrayList d’entrepôts :**

String[] lignes = fichier.split(\n) O(1)

For (i=1 ; i<lignes.length; i++) { O(N)

String[] coordEntrepot = lignes[i].split(“ “) O(1)

For (j=0; j<coordEntrepot.length; j++) { O(N)

Entrepot nouvelEntrepot = new Entrepot (coordEntrepot[n], coordEntrepot[n1], coordEntrepot[n2]) où n = 0,3,6,9…, n1 = 1,4,7,…, n2 = 2,5,8,… O(1)

ArrayList.add(nouvelEntrepot) O(1)

}

}

Ce qui donne pour cette fonction un ordre de grandeur de O(N2) (à cause des boucles imbriqués).

**Fonction qui trouve le max de boite et positionne le camion :**

int maxEntrepot = 0 O(1)

For (listeEntrepot) [ O(N)

If (nbBoiteEntrepotActuel > maxEntrepot) { O(1)

latCamion = latEntrepotActuel O(1)

longCamion = longEntrepotActuel O(1)

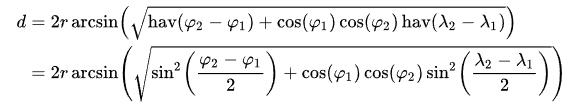
maxEntrepot = nbBoiteEntrepotActuel O(1)

}

}

Ce qui donne comme ordre de grandeur O(N).

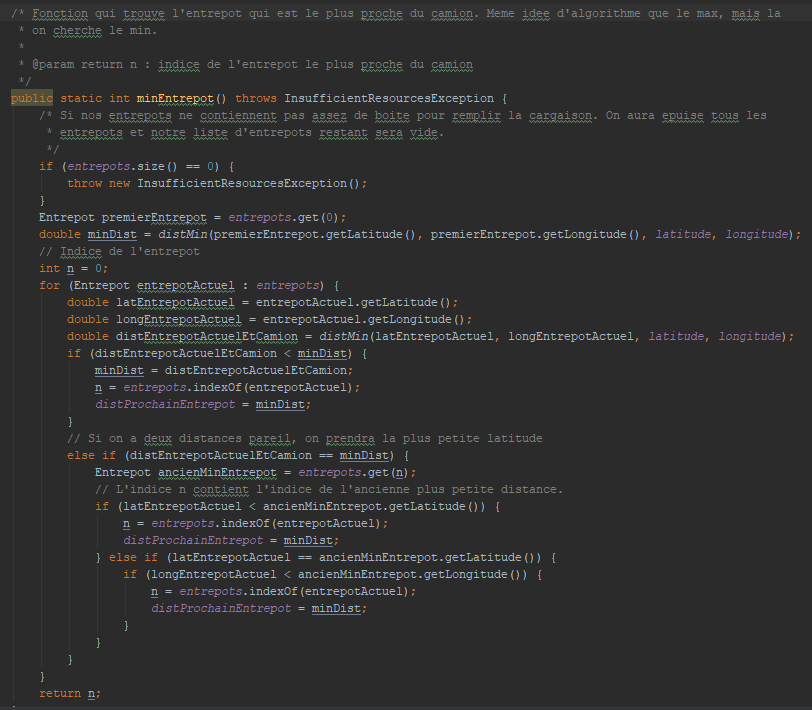
**Fonction qui calcule la distance entre deux points avec la formule de haversine :**



Le code est copier collé de la formule, donc on peut analyser la formule au lieu du code. Les fonctions trigonométriques ont un temps de complexité de O(M(n) log n) où M(n) = O(n log n 2log n) et n est égale aux nombres de bit de précision. Mais comme on a un arcsin au début, peut importe ce qu’il y a dans la parenthèse (dans ce cas, les sinus au carré), l’ordre de grandeur restera de O(M(n) log n).

**Fonction qui calcule la distance la plus proche entre le camion et le prochain entrepôt à visiter :**

Pour celui-là, vu qu’il est gros, je collerai une image du code :



On peut voir que l’algorithme utilise une simple boucle for pour traverser toute la liste des entrepôts. Le reste (les if et les manipulation du ArrayList) prend qu’une étape (O(1)). Donc l’ordre de grandeur est O(N).

**Tous ensemble :**

On a comme ordres de grandeur : O(N2), O(N), O(M(n) log n) où M(n) = O(n log n 2log n), O(N2)

Ce qui donne comme ordre de grandeur finale : O(N2) en pire cas.

Analyse de la complexité temporelle empirique :

Pour l’analyse empirique, on l’a fait sur intelliJ avec la fonction System.nanoTime(). On place un au début du programme et un à la toute fin et on fait une soustraction des deux. Pour le convertir en seconde, il suffit de le diviser par 109. Il est à noter qu’utiliser un IDE ralentit le temps d’exécution du programme comparé à utiliser le terminal. De plus, pour chaque exécution, on s’est assuré d’avoir le moins de chose possible qui roule en arrière-plan et de ne pas faire des tâches simultanées à l’exécution du programme. Après avoir exécuté plusieurs fois le même point, on a fait une moyenne. Ce qui donne le graphique suivant avec le fichier <<entré supplémentaires>> :

Le graphique peut ressembler à une droite linéaire, mais elle est en fait de degré deux, car le coefficient de corrélation est meilleur. Ce qui confirme d’avantage que notre ordre de grandeur théorique est bonne.

Conclusion :

Par notre mini analyse de notre code et des tests fait sur les fichiers donnés, on peut conclure que notre programme fonctionne correctement. De plus, notre analyse théorique et empirique nous donne, en pire cas, un ordre de grandeur de O(N2).