Rapport du TP1
Structure de Donnée

Lallia Diakité - 20256054
Naromba Conde - 20251772

05 Juin 2024

1 Auto-Evaluation

Notre programme a pour but de gérer le processus de cargaison des camions dans un entrepôt. La classe principale de notre code est Tp1. Cette classe est responsable de l'initialisation et de l'exécution des différentes étapes du programme.

La classe Warehouse joue un rôle central dans notre programme. Elle gère la lecture des données d'entrée, le calcul des distances entre les bâtiments et le camion, et le tri des distances. Elle coordonne également les opérations de chargement et de déchargement.

Nous avons choisi les méthodes de tri qui nous semblait les plus efficace pour notre programme afin d'optimiser le processus de cargaison des camions. Le tri par insertion a été sélectionné pour sa simplicité et son efficacité sur de petites tailles de données et Le tri rapide (Quick Sort) pour sa performance moyenne en O(n log n) et sa rapidité sur des ensembles de données de taille moyenne à grande.

En conclusion, après avoir validé notre programme avec plusieurs tests fournis par la professeure, nous pouvons affirmer qu'il fonctionne correctement (à 100%) . Il répond aux exigences du TP et fournit une gestion efficace du processus de cargaison des camions dans un entrepôt.

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

2.1 Tri par Insertion

Pseudo Code du tri par Insertion

```
Procédure triParInsertion(liste)
    pour i de 1 à longueur(liste) - 1 faire  // O(n)
        clé = liste[i]  // O(1)
        j = i - 1  // O(1)
        tant que j >= 0 et liste[j].distance > clé.distance faire  // O(i) dans le pire cas
            liste[j + 1] = liste[j]  // O(1)
            j = j - 1  // O(1)
        fin tant que
        liste[j + 1] = clé  // O(1)
    fin pour
Fin Procédure
```

Donc, la complexité temporelle asymptotique de l'algorithme de tri par insertion est $O(n^2)$

Pseudo-code du Tri Rapide (Quick Sort)

```
Procédure sortBuildingsByDistanceQuickSort()
```

```
    pour chaque building dans buildings faire  // O(n)
        building.calculerDistance(truck.getLatitude(), truck.getLongitude())  // O(1)
    fin pour
    quickSort(0, buildings.size() - 1)
Fin Procédure

Procédure quickSort(low, high)
    si low < high alors  // O(1)
        pi = partition(low, high)  // O(n) dans le pire cas
        quickSort(low, pi - 1)  // T(n/2) dans le pire cas
        quickSort(pi + 1, high)  // T(n/2) dans le pire cas
    fin si
Fin Procédure

Procédure partition(low, high)
    pivot = buildings.get(high)  // O(1)
    i = (low - 1)  // O(1)
    pour j de low à high - 1 faire  // O(n)
        si buildings.get(j).getDistance() <= pivot.getDistance() alors  // O(1)
            i = i + 1  // O(1)
            échanger buildings[i] avec buildings[j]  // O(1)
        fin si
    fin pour
    échanger buildings[i + 1] avec buildings[high]  // O(1)
    retourner (i + 1)  // O(1)
Fin Procédure
```

Donc,si le pivot choisi est le plus grand ou le plus petit élément à chaque partition, la complexité devient $T(n) = T(n-1) + O(n)$, ce qui résout à $O(n^2)$.

3 Analyse empirique de la complexité temporelle

3.1 Tri Java

Analyse des Temps d'Exécution

Fichier 1

— Taille des données : 492

— Temps d'exécution : 8000 µs

Fichier 2

— Taille des données : 349

— Temps d'exécution : 6000 µs

Fichier 3

— Taille des données : 3500

— Temps d'exécution : 7000 µs

Fichier 4

— Taille des données : 283

— Temps d'exécution : 10000 µs

Fichier 5

— Taille des données : 453

— Temps d'exécution : 6000 µs

Fichier 6

— Taille des données : 800

— Temps d'exécution : 6000 µs

Graphe de Performance

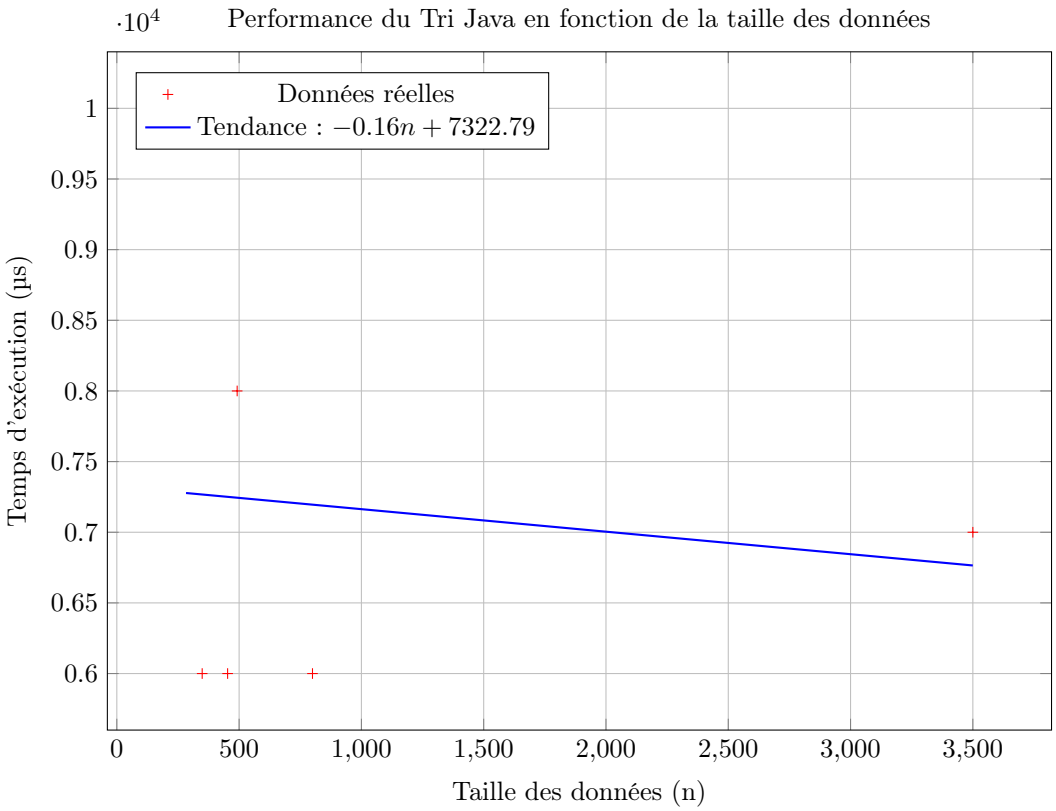

FIGURE 1 – Performance du Tri Java en fonction de la taille des données

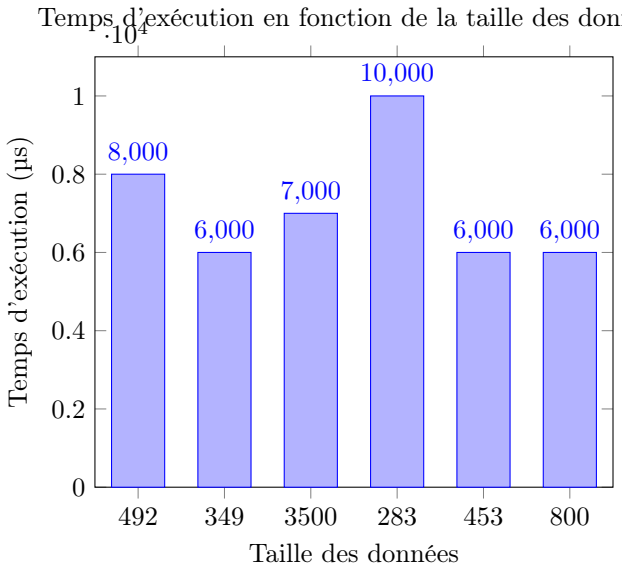

FIGURE 2 – Temps d'exécution du Tri Java en fonction de la taille des données

3.2 Tri Par Insertion

Données et Résultats

Fichier 1

— Taille des données : 492
— Temps d'exécution : 349 µs

Fichier 2

— Taille des données : 350
— Temps d'exécution : 290 µs

Fichier 3

— Taille des données : 3500
— Temps d'exécution : 237 µs

Fichier 4

— Taille des données : 283
— Temps d'exécution : 339 µs

Fichier 5

— Taille des données : 453
— Temps d'exécution : 317 µs

Fichier 6

— Taille des données : 800
— Temps d'exécution : 342 µs

Graphe de Performance

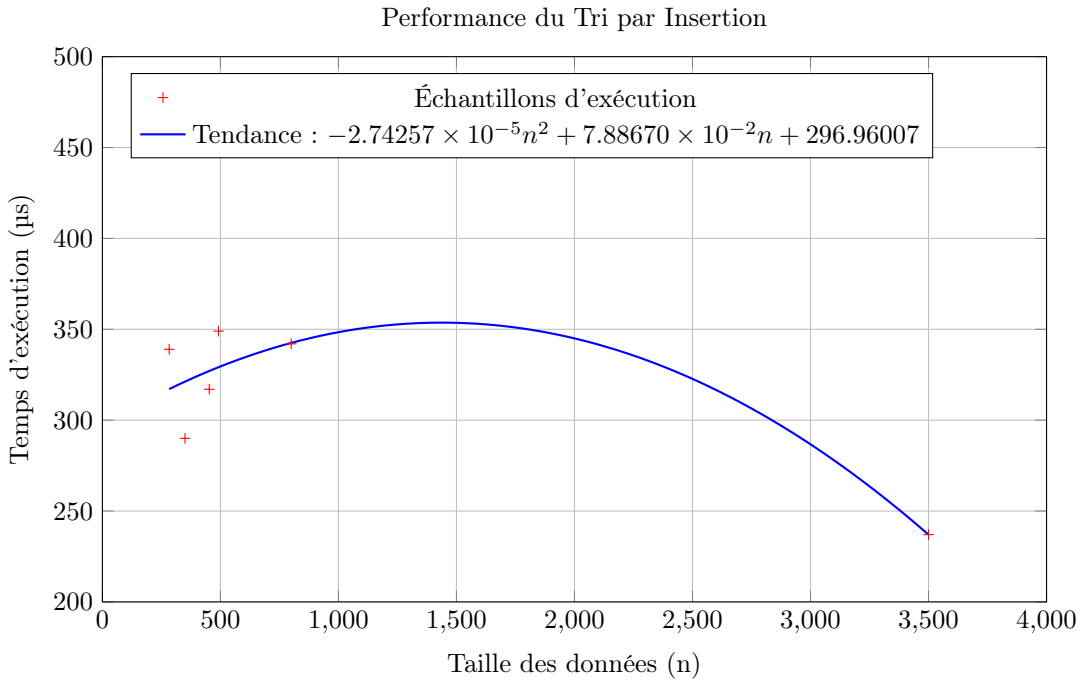

FIGURE 3 – Performance du Tri par Insertion en fonction de la taille des données

Diagramme en Bâton

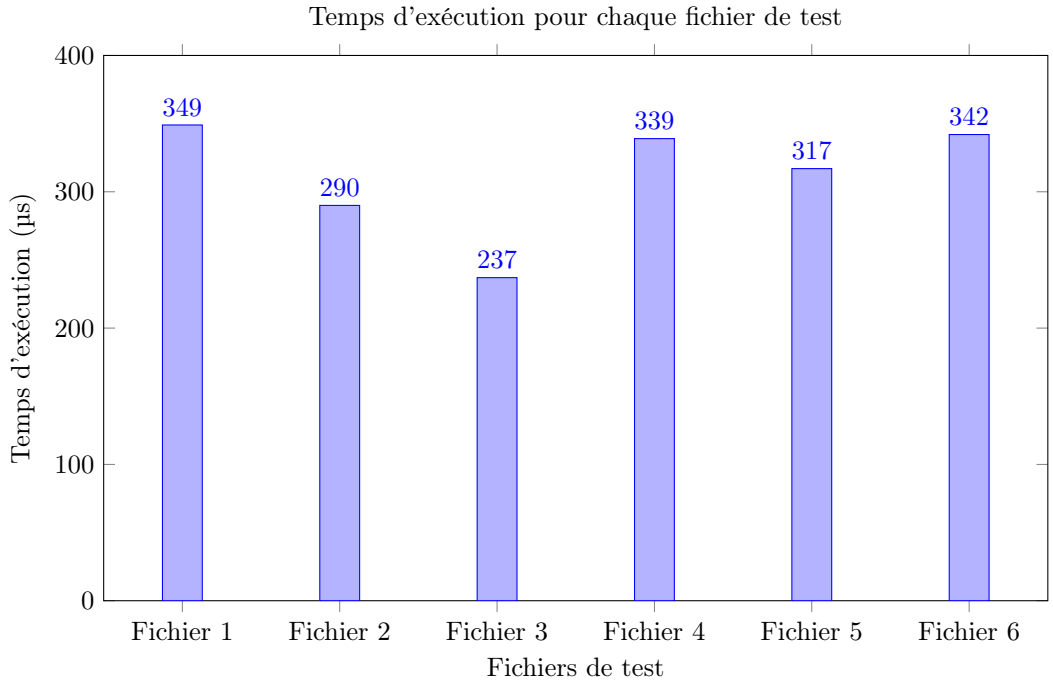

FIGURE 4 – Temps d'exécution pour chaque fichier de test

3.3 Tri rapide

Fichier 1

— Taille des données : 492
— Temps d'exécution : 465 µs

Fichier 2

— Taille des données : 350
— Temps d'exécution : 448 µs

Fichier 3

— Taille des données : 3500
— Temps d'exécution : 417 µs

Fichier 4

— Taille des données : 283
— Temps d'exécution : 391 µs

Fichier 5

— Taille des données : 453
— Temps d'exécution : 370 µs

Fichier 6

— Taille des données : 800
— Temps d'exécution : 444 µs

Graphe de Performance

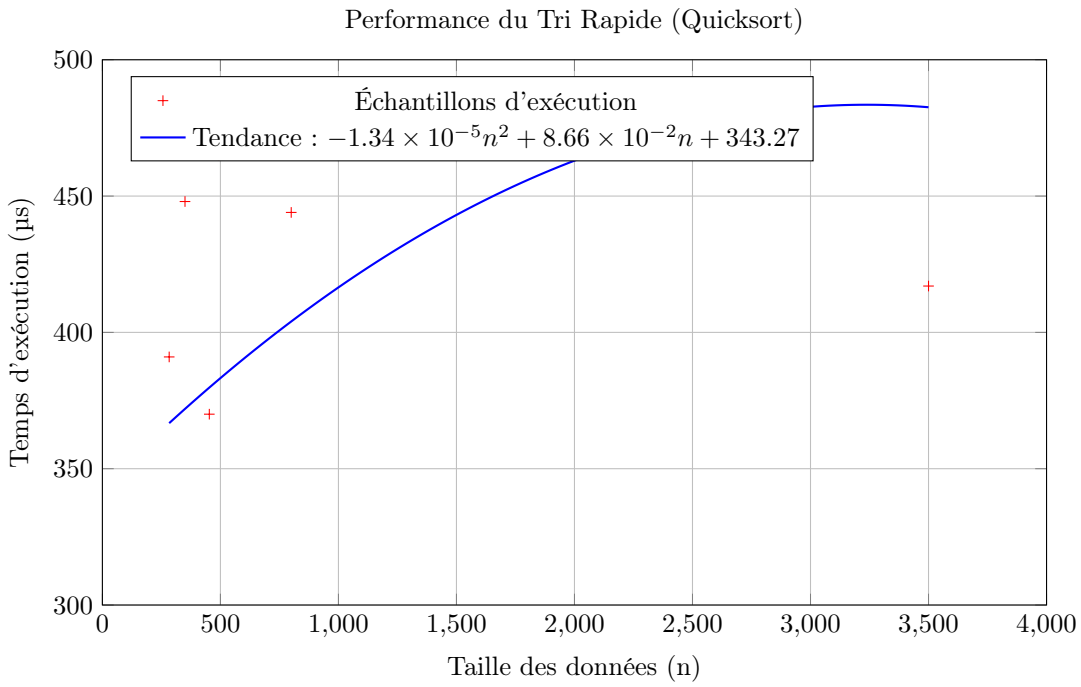

FIGURE 5 – Performance du Tri Rapide (Quicksort) en fonction de la taille des données

Diagramme en Bâton

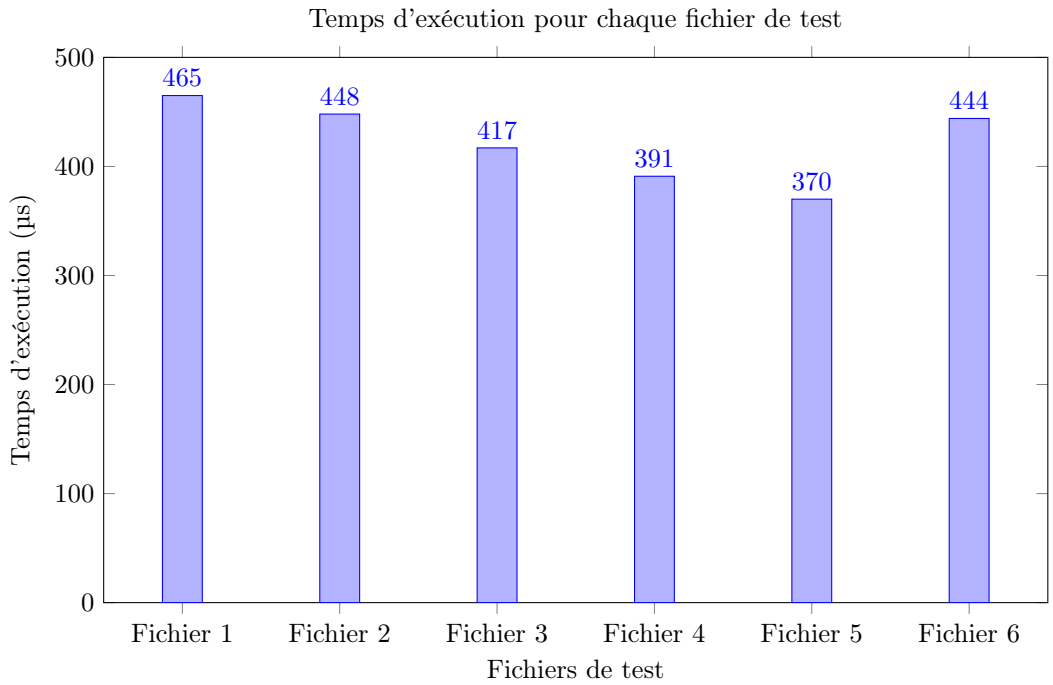

FIGURE 6 – Temps d'exécution pour chaque fichier de test

Instructions pour Compiler et Exécuter le Code Java

Étapes pour Compiler et Exécuter le Code

Cette section décrit les étapes nécessaires pour compiler et exécuter le programme Java qui gère le processus de cargaison des camions dans un entrepôt.

Compilation

— La première étape consiste à compiler les fichiers Java. Ouvrez un terminal ou une invite de commande.

— Accédez au dossier où se trouvent vos fichiers Java.

— Compilez les fichiers Java en utilisant la commande suivante :

```
javac *.java
```

— Cette commande compilera tous les fichiers Java présents dans le dossier et générera les fichiers `.class` correspondants.

Exécution

— Pour exécuter le programme, utilisez la commande suivante en fournissant les arguments nécessaires (le chemin du fichier d'entrée et le chemin du fichier de sortie) :

```
java Tp1 <chemin_du_fichier_d_entrée> <chemin_du_fichier_de_sortie>
```

— Par exemple, si votre fichier d'entrée est `input.txt` et vous souhaitez que le fichier de sortie soit `output`, utilisez :

```
java Tp1 input.txt output
```

— Cette commande exécutera votre programme et générera les fichiers de sortie avec les suffixes `_java.txt`, `_insertion.txt`, et `_quicksort.txt`.

Exemple

Supposons que vous avez un fichier d'entrée nommé `input.txt` dans le même dossier que vos fichiers Java. Vous souhaitez générer les fichiers de sortie dans le même dossier.

— **Compilation :**

```
javac *.java
```

— **Exécution :**

```
java Tp1 input.txt output
```

— Après l'exécution, vous trouverez les fichiers de sortie suivants dans votre dossier :

  — `output_java.txt`
  — `output_insertion.txt`
  — `output_quicksort.txt`

Ces fichiers contiendront les résultats du tri des bâtiments par distance en utilisant les différents algorithmes de tri implémentés dans votre code.