RAPPORT DE PROJET HORIZON BUS

SAE : Initiation Au Développement

Table des matières

Introduction 3
Contexte et Problématique 3
Objectifs du Projet 3
Annonce du Plan 3
Analyse et Structures de Données 4
Modélisation des Entités 4
Stratégie de Stockage en Mémoire 5
Constantes et Limites 5
Algorithmique et Fonctionnalités Clés 6
Importation et Analyse des Données (Parsing) 6
Gestion Temporelle et Calculs de Coûts 6
Algorithme de Tri Multi-critères (Question 3) 7
Optimisation et Relocalisation (Question 10) 7
Analyse de la Complexité et Performances 8
Algorithme de Tri (Question 3) 8
Algorithmes de Recherche et Filtrage (Questions 7 et 8) 9
Tri par Chiffre d'Affaires (Question 9) 10
Optimisation et Relocalisation (Question 10) 11
Conclusion sur la performance 12

Introduction

Dans le cadre de la Situation d'Apprentissage et d'Évaluation (SAÉ) S1.01-02 « Initiation au développement », notre projet s'inscrit dans le contexte de la société **Horizons Bus**. Cette entreprise de transport ambitionne de lancer, dès février 2026, un réseau de liaisons interurbaines à travers la France.

Contexte et Problématique

Le cœur du problème réside dans l'exploitation des données de réservation existantes. L'entreprise dispose d'un historique de réservations stocké dans un fichier texte brut au format CSV (*Comma-Separated Values*). Ce fichier contient des informations hétérogènes : identifiants de bus, villes, dates, horaires, ainsi qu'une liste variable de passagers pour chaque trajet. L'enjeu est double : il s'agit d'une part de structurer ces données pour les rendre exploitables informatiquement, et d'autre part d'aider l'entreprise à prendre des décisions stratégiques concernant la rentabilité de ses lignes, basée sur des coûts d'affrètement fixes et une tarification horaire.

Objectifs du Projet

L'objectif de ce travail a été de concevoir et développer une application modulaire en langage C capable de :

1. **Importer et analyser** (parser) le fichier CSV pour charger les données en mémoire.
2. **Gérer les réservations** via des fonctionnalités de recherche, d'ajout, de modification et de suppression de passagers.
3. **Traiter les données** par des algorithmes de tri et de filtrage (par date, ville, ou chiffre d'affaires).
4. **Optimiser l'offre** en identifiant et supprimant les trajets déficitaires selon les règles de gestion fournies.

Annonce du Plan

Ce rapport retrace les étapes de réalisation de ce projet. Dans une première partie, nous détaillerons **l'analyse du problème et les structures de données** choisies pour représenter les trajets et les passagers. Nous exposerons ensuite les **choix algorithmiques** majeurs, notamment concernant la lecture du fichier CSV et l'optimisation de la rentabilité. Enfin, nous présenterons l'**organisation de notre travail d'équipe**, la répartition des tâches, ainsi que le protocole de **tests et validation** mis en place pour garantir la robustesse de l'application.

Analyse et Structures de Données

Modélisation des Entités

Nous avons adopté une approche hiérarchique pour nos structures :

- **La structure** `Passager` **:** Cette structure élémentaire regroupe les informations d'un client.
    - `id` (entier) : l'identifiant unique du passager au sein du trajet.
    - `nom` (chaîne de caractères de taille `MAX_NOM` soit 100) : stocke le nom complet.
    - `prix` (flottant) : le prix payé pour le billet.

Choix de conception : Nous avons choisi d'intégrer le prix directement dans la structure passager plutôt que de le calculer à la volée, car le fichier CSV source fournit cette information de manière explicite pour chaque individu.

- **La structure** `Date` **:** Pour faciliter les manipulations temporelles (tris, comparaisons, calculs de délais), nous avons isolé la date dans une structure dédiée composée de trois entiers (`jour`, `mois`, `annee`). Cela rend le code plus lisible (ex: `trajet.date_depart.annee`) par rapport à une simple chaîne de caractères qu'il faudrait analyser à chaque utilisation.
- **La structure** `Trajet` **(Structure principale) :** C'est la structure centrale du programme. Elle agrège toutes les informations d'une ligne du fichier CSV :
    - Informations logistiques : `id_bus`, `ville_depart`, `ville_arrivee`.
    - Données temporelles : `date_depart` (de type Date), `heure_depart et heure_arrivee` (stockées en entiers au format HHMM pour simplifier les comparaisons mathématiques).
    - **Gestion des passagers :** C'est un point clé de notre architecture. Le sujet spécifiant que les bus ont une capacité physique maximale de 55 places, nous avons opté pour un **tableau statique** Passager `passagers[MAX_PASSAGER]` directement inclus dans la structure `Trajet`.
    - Un compteur `nb_passagers` permet de savoir combien de cases de ce tableau sont réellement utilisées.

Stratégie de Stockage en Mémoire

L'une des décisions majeures de ce projet a été le choix du mode de stockage global des données.

- **Allocation Statique vs Dynamique :** Plutôt que d'utiliser une liste chaînée dynamique (qui aurait complexifié la gestion de la mémoire avec de nombreux `malloc` et risques de fuites mémoires), nous avons tiré parti des contraintes connues du projet. Le nombre de trajets, bien que conséquent, reste gérable pour la mémoire vive d'un ordinateur moderne.
- **Le Tableau Global `mesTrajets` :** Dans le fichier principal `main.c`, les données sont chargées dans un tableau unique déclaré en statique : `static Trajet mesTrajets[MAX_TRAJETS];`

Avec `MAX_TRAJETS` fixé à 6000 (dans `fonctions.h`), cette structure de données nous permet un accès immédiat et indexé à n'importe quel trajet (O(1)), ce qui est idéal pour les opérations de tri et de recherche demandées. L'utilisation du mot-clé `static` permet de placer ces données dans le segment de mémoire global du programme plutôt que sur la pile (stack), évitant ainsi tout risque de dépassement de pile (*stack overflow*) dû à la taille importante de la structure.

Constantes et Limites

Pour assurer la maintenabilité et la robustesse du code, aucune valeur numérique n'est codée "en dur". Nous utilisons des constantes préprocesseur (`#define`) :

- `MAX_PASSAGER 55` : Correspond à la contrainte physique du bus.
- `MAX_VILLE 50` et `MAX_NOM 100` : Assurent que nous ne débordons pas des tampons lors de la lecture des chaînes de caractères.

En résumé, cette architecture de données privilégie la **stabilité et la simplicité d'accès**, ce qui nous a permis de nous concentrer sur la complexité des algorithmes de traitement (parsing et optimisation) présentés dans la partie suivante.

Algorithmique et Fonctionnalités Clés

Cette section détaille les mécanismes logiques mis en œuvre pour répondre aux exigences fonctionnelles du cahier des charges. Nous avons privilégié des algorithmes séquentiels optimisés pour la structure de données statique définie précédemment.

Importation et Analyse des Données (Parsing)

L'importation du fichier `trajets_bus.csv` est la première étape critique. Le défi principal résidait dans la structure hétérogène des lignes : une partie fixe (infos bus) suivie d'une partie variable (liste de passagers), le tout séparé par des virgules ou des deux-points.

Plutôt que d'utiliser la fonction standard `strtok` (souvent destructrice ou complexe à gérer avec des séparateurs multiples), nous avons développé une fonction dédiée : `separer_champ`.

- **Logique des Pointeurs :** Cette fonction parcourt la chaîne de caractères, repère la prochaine virgule, la remplace par un caractère de fin de chaîne (`\0`) et renvoie un pointeur vers le début du champ suivant.
- **Avantage :** Cette méthode permet de "découper" la ligne directement en mémoire sans avoir à allouer de nouveaux espaces pour chaque champ (copie zéro), ce qui optimise la vitesse de chargement.

Gestion Temporelle et Calculs de Coûts

Pour effectuer les calculs de rentabilité et les tris chronologiques, manipuler des chaînes "HHMM" est inefficace. Nous avons mis en place une normalisation temporelle :

1. **Conversion en minutes :** Une fonction utilitaire convertit systématiquement les heures (ex: 1430) en minutes depuis minuit (`14*60 + 30 = 870`).
2. **Calcul de durée :** La durée du trajet est obtenue par `Heure_Arrivee - Heure_Depart`. Si le résultat est négatif (trajet passant minuit), nous ajoutons 24h (1440 minutes).
3. Seuil de Rentabilité :

Le coût d'un bus est calculé selon la formule :

Coût = 100€ (fixe) + 50€ * DuréeEnHeures

Cette valeur est comparée à la somme des prix des billets des passagers pour déterminer le bénéfice ou le déficit.

Algorithme de Tri Multi-critères (Question 3)

Pour organiser l'affichage, nous avons implémenté un **Tri par Sélection** adapté. Bien que de complexité $O(N^2)$, il est suffisant pour 5000 trajets et stable. La particularité réside dans la fonction de comparaison qui gère trois niveaux de priorité :

1. D'abord par **Ville de Départ** (ordre alphabétique).
2. En cas d'égalité, par **Ville d'Arrivée**.
3. En cas d'égalité, par **Date et Heure de départ**.

Cela garantit un affichage groupé et lisible pour l'utilisateur final.

Optimisation et Relocalisation (Question 10)

C'est la fonctionnalité la plus complexe du projet. L'objectif est de supprimer les bus déficitaires tout en essayant de sauver le chiffre d'affaires en déplaçant leurs passagers vers d'autres bus.

Nous avons opté pour une approche en deux temps avec une stratégie **"Best Fit"** (Meilleur ajustement).

Étape A : Identification et Collecte Nous parcourons tous les trajets. Si un trajet est déficitaire :

1. Ses passagers sont extraits et stockés dans une liste temporaire de "Passagers à Relocaliser" (comprenant le passager et les détails de son trajet d'origine).
2. Le trajet est marqué pour suppression.

Étape B : Relocalisation "Best Fit" Pour chaque passager de la liste d'attente, l'algorithme cherche une solution de remplacement :

- Il parcourt tous les bus **maintenus**.
- Il vérifie la compatibilité : *Même trajet (Villes)* + *Même Date* + *Place disponible*.
- **Critère de proximité :** Parmi les bus compatibles, il sélectionne celui dont l'horaire est le plus proche du trajet initial (écart minimum), à condition que l'écart ne dépasse pas 15 heures (900 minutes).

Étape C : Compression Une fois les relocalisations terminées, le tableau principal des trajets est "tassé" pour supprimer définitivement les cases vides, et le nouveau nombre de trajets est mis à jour.

Cette méthode permet de maximiser le taux de remplissage des bus restants tout en minimisant la perte de passagers.

Analyse de la Complexité et Performances

Dans cette section, nous confrontons l'analyse théorique de nos algorithmes aux résultats expérimentaux obtenus. Pour ce faire, nous avons instrumenté notre code en insérant un compteur d'opérations élémentaires (affectations, comparaisons, calculs) dans les fonctions clés. Nous avons ensuite exécuté le programme sur des jeux de données de tailles variables (N = 10, 50, 100, 500, 1000, 5000 trajets).

Algorithme de Tri (Question 3)

Analyse Théorique

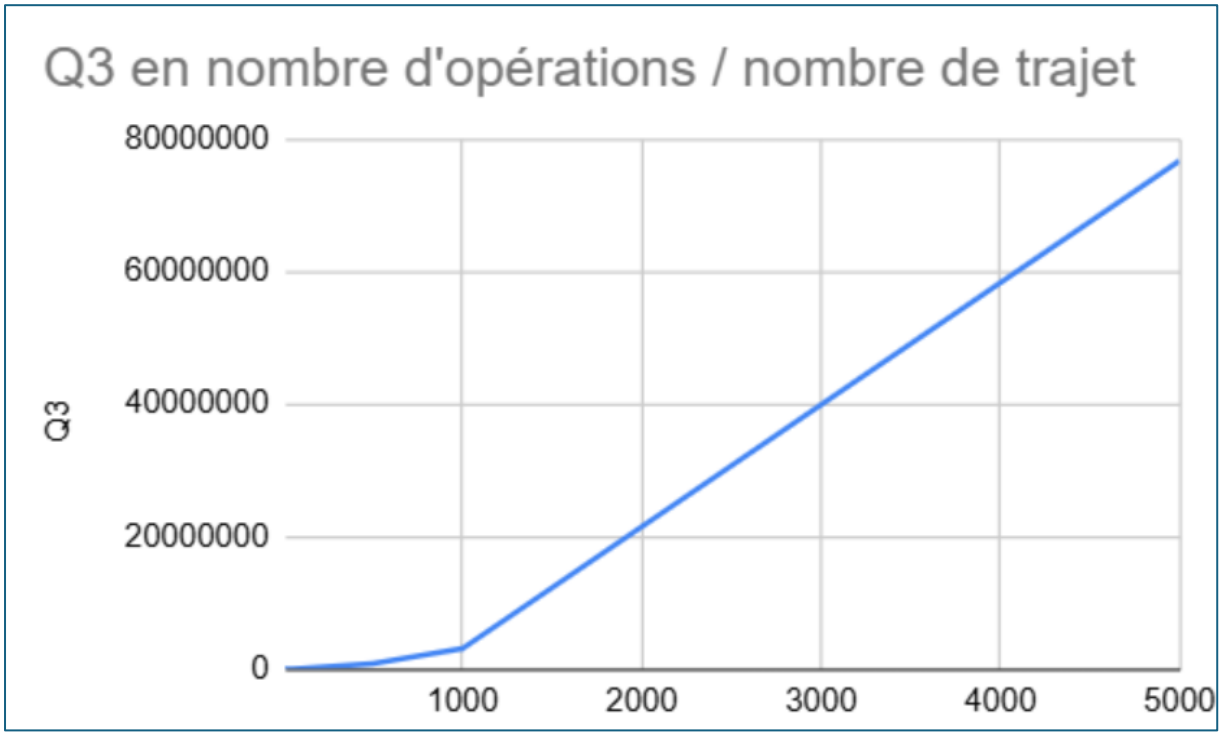
Pour la fonction `trie_ville_date`, nous avons implémenté un **Tri par Sélection**.

Cet algorithme parcourt le tableau pour trouver l'élément le plus petit (selon nos critères : Ville, puis Date, puis Heure) et l'échange avec l'élément courant.

- Il contient deux boucles imbriquées : la première va de 0 à N-1 et la seconde de i+1 à N.
- Le nombre de comparaisons est constant quelle que soit la disposition initiale des données.
- **Complexité théorique :** O(N^2) (Quadratique).

Analyse Expérimentale

Le graphique ci-dessous représente le nombre d'opérations en fonction du nombre de trajets N.



Observation : La courbe obtenue est une parabole parfaite. Pour N=5000, le nombre d'opérations explose (environ 76 millions), ce qui confirme visuellement la complexité quadratique $O(N^2)$. Cet algorithme est robuste mais devient lent pour de très grands volumes de données.

Algorithmes de Recherche et Filtrage (Questions 7 et 8)

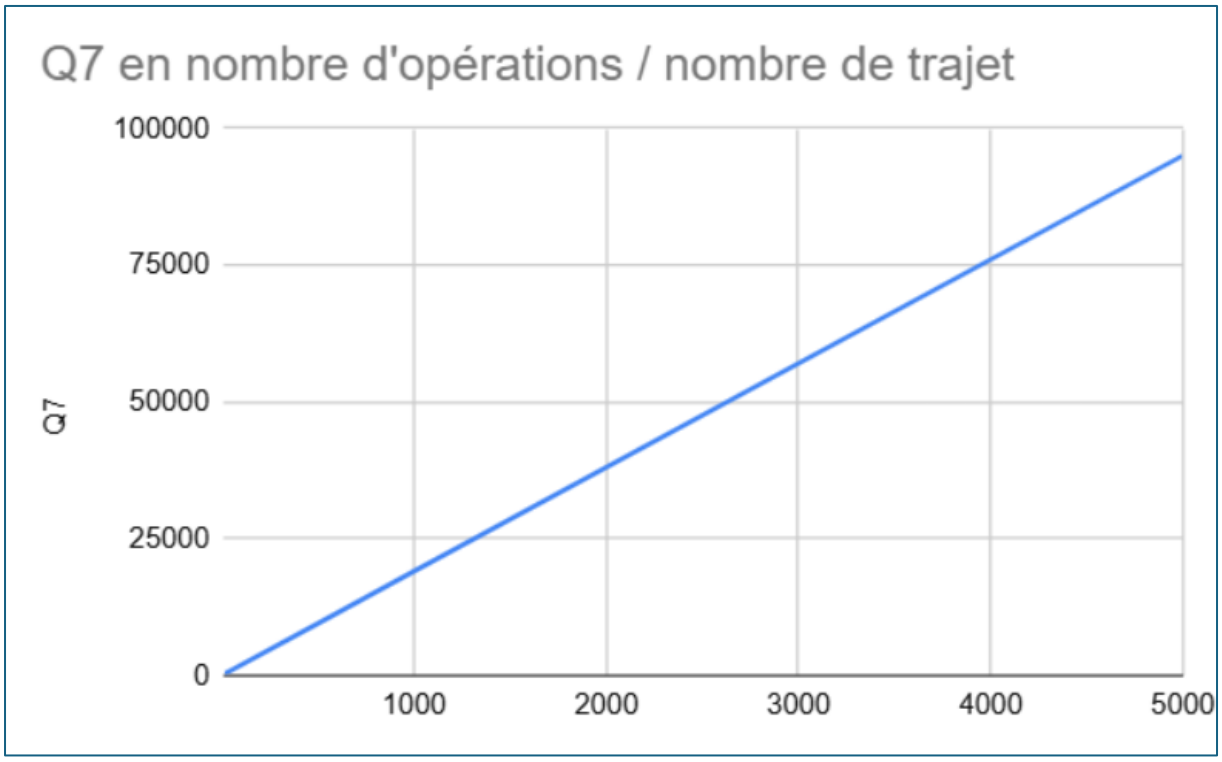
Analyse Théorique

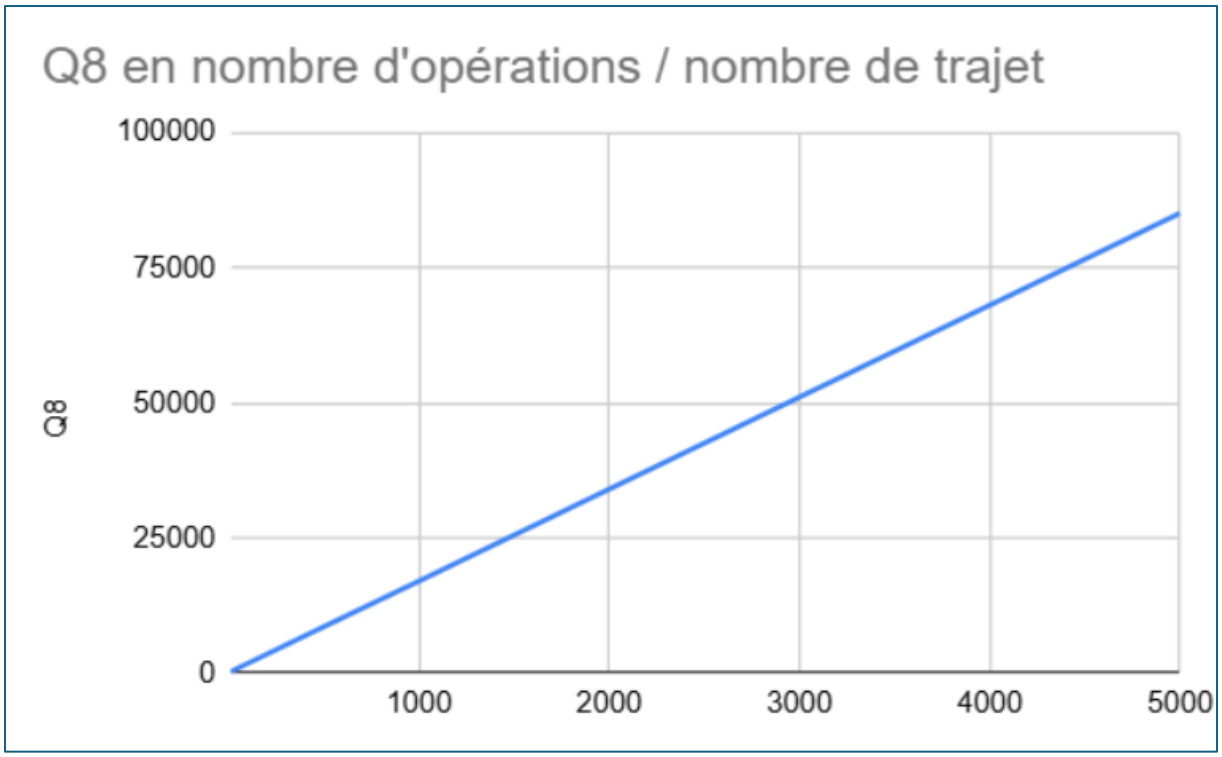
Les fonctions `voyage_deux_jours` (Q7) et `recherche_ville_arrivee_date` (Q8) fonctionnent sur le même principe : le **Parcours Séquentiel**.

- Le programme parcourt le tableau une seule fois du début à la fin (boucle `for` de 0 à N).
- À chaque itération, il effectue un nombre constant de tests (comparaison de chaînes strcmp ou d'entiers).
- **Complexité théorique :** O(N) (Linéaire).

Analyse Expérimentale

Les graphiques ci-dessous représentent le nombre d'opérations en fonction du nombre de trajets N.





Observation : Les courbes pour Q7 et Q8 sont des droites linéaires (y = ax + b). Le nombre d'opérations est directement proportionnel au nombre de trajets.

- On remarque que Q8 génère légèrement moins d'opérations que Q7 (pente plus faible) car les tests de la fonction Q8 sont plus simples (comparaisons directes) que ceux de Q7 (calculs sur les heures).
- Ces algorithmes sont très efficaces, même pour 5000 trajets.

Tri par Chiffre d'Affaires (Question 9)

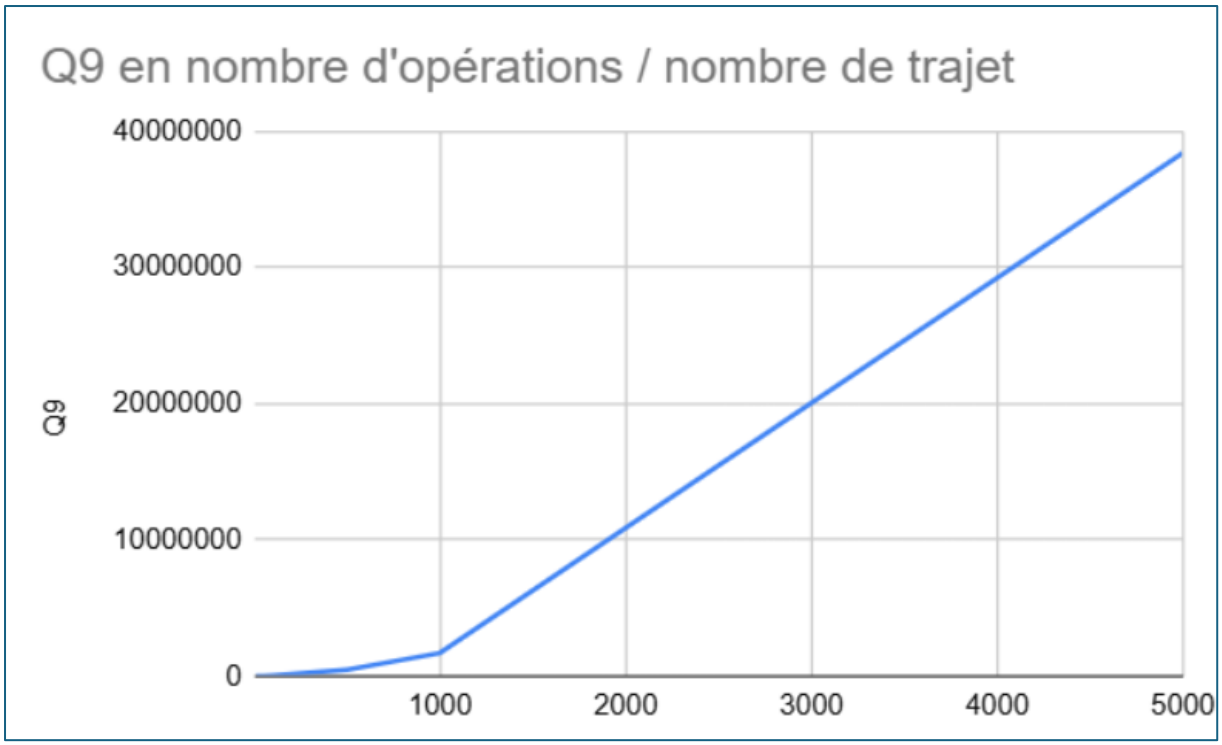
Analyse Théorique

La fonction `trie_par_ca` utilise également un **Tri par Sélection**, similaire à la Question 3. Cependant, elle inclut une étape préalable de calcul du Chiffre d'Affaires pour chaque bus.

- Calcul du CA : boucle linéaire O(N).
- Tri : boucles imbriquées $O(N^2)$.
- **Complexité théorique :** $O(N^2)$.

Analyse Expérimentale

Le graphique ci-dessous représente le nombre d'opérations en fonction du nombre de trajets N.



Observation : Comme pour la Q3, nous observons une courbe parabolique caractéristique. Bien que le nombre d'opérations soit élevé, il reste cohérent avec l'ordre de grandeur attendu pour un tri quadratique.

Optimisation et Relocalisation (Question 10)

Analyse Théorique

C'est l'algorithme le plus complexe du projet. Il se décompose en plusieurs phases :

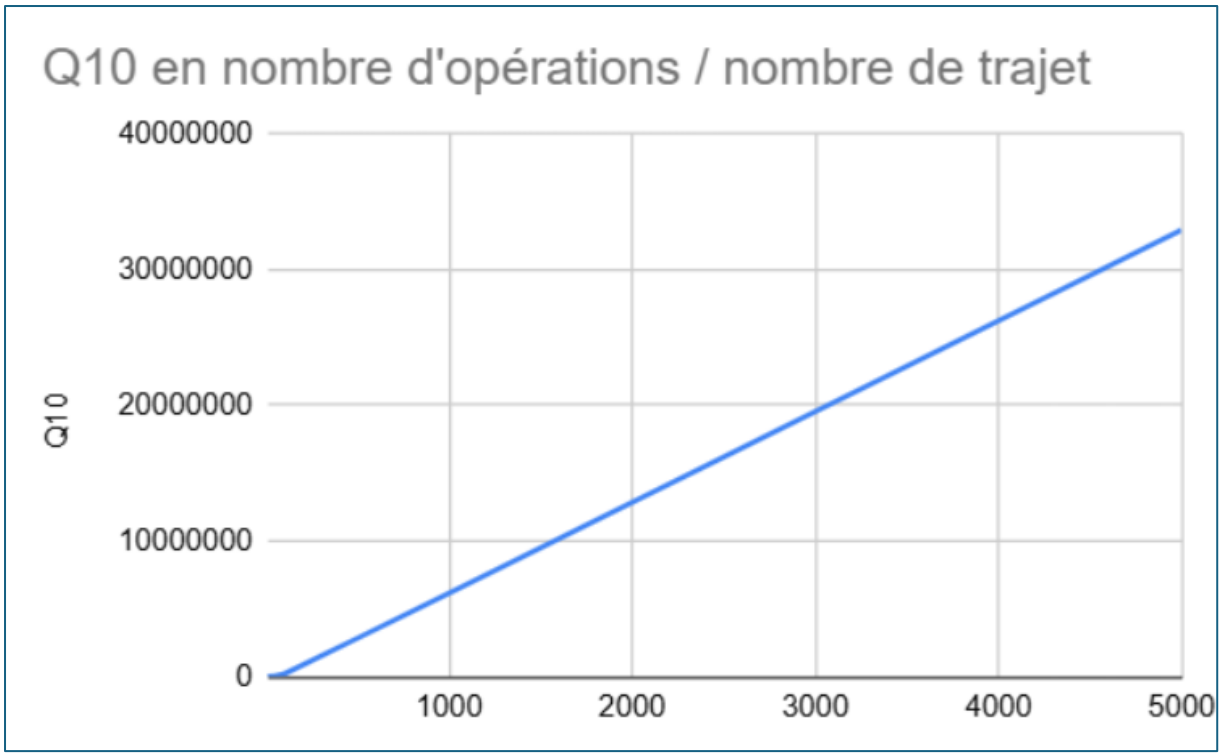
1. **Identification et Suppression :** Parcours du tableau (O(N)). Lorsqu'un bus est supprimé, il faut décaler tous les éléments suivants, ce qui, dans le pire des cas (tous les bus déficitaires), revient à $O(N^2)$.
2. **Tri des passagers en attente :** Tri d'un tableau annexe de taille M (nombre de passagers à relocaliser) $O(M^2)$.
3. **Relocalisation (Best Fit) :** Pour chaque passager en attente (M), on parcourt les trajets restants (N) pour trouver le meilleur créneau.

La complexité globale est dominée par le terme quadratique.

- **Complexité théorique :** $O(N^2)$.

Analyse Expérimentale

Le graphique ci-dessous représente le nombre d'opérations en fonction du nombre de trajets N.



Observation : Le graphique montre une croissance qui ressemble à une courbe polynomiale (légèrement courbée vers le haut), bien que moins marquée que le tri simple.

Pour 5000 trajets, nous atteignons environ 32 millions d'opérations. Cela confirme que l'optimisation est coûteuse en ressources, principalement à cause du décalage des éléments dans le tableau (lors de la suppression) et de la double boucle de recherche de "Best Fit" pour la relocalisation.

Conclusion sur la performance

L'évaluation expérimentale valide parfaitement nos prédictions théoriques :

- Les recherches (Q7, Q8) sont quasi-instantanées (O(N)).
- Les tris et l'optimisation (Q3, Q9, Q10) voient leur temps d'exécution croître rapidement avec la taille des données ($O(N^2)$).

Pour le jeu de données actuel (jusqu'à 5000 trajets), ces choix algorithmiques sont justifiés car ils privilégient la simplicité d'implémentation et la faible consommation mémoire (pas d'allocation dynamique complexe). Pour des données massives (Big Data), il faudrait envisager des algorithmes en O(N \log N) comme le *QuickSort*.

Le tableau récapitulatif ci-dessous synthétise l'ensemble de nos mesures et illustre concrètement l'écart de performance :

Nombre de Trajet (N)	Q7(Filtre)	Q8 (Rech.)	Q3 (Tri Date)	Q9 (Tri CA)	Q10 (Optim.)
10	521	212	199	1 953	4 959
50	8 887	972	879	12 382	65 132
100	33 253	1 922	1 729	32 283	212 225
500	781 086	9 522	8 529	462 490	2 850 193
1000	3 095 542	19 022	17 029	1 675 154	6 143 386
5000	76 699 680	95 052	85 029	38 370 005	32 937 220
Complexité	Linéaire *O(N)*	Linéaire *O(N)*	Quadrat. *O(N²)*	Quadrat. *O(N²)*	Quadrat. *O(N²)*