Université du Havre Normandie

UFR Sciences et Techniques

Projet d'algorithmique : Complexité des algorithmes de tri

Baouche Mohamed Djaouad
Bennabi Ghiles Rayane

Ce projet nous apprendra à étudier et à comparer les algorithmes de tri, un élément clé de notre domaine.

Table des matières

Introduction

Ce projet porte sur l'analyse de la complexité de trois algorithmes de tri :

- **Tri par sélection du maximum**
- **Tri rapide (QuickSort)**
- **Tri par dénombrement**

L'objectif est de comparer leur performance en termes de **temps d'exécution** et de **nombre d'opérations élémentaires** (comparaisons et affectations).

Le programme prend en entrée un tableau d'éléments, chaque élément étant caractérisé par une **clé** (un entier) et une **valeur** (un réel).

Les tests sont réalisés sur des tableaux de différentes tailles, générés soit **manuellement** (petites tailles), soit **aléatoirement** (grandes tailles).

Présentation des Algorithmes

1 Tri par sélection du maximum

Le tri par sélection du maximum est un algorithme de tri simple qui fonctionne en sélectionnant le plus grand élément de la partie non triée du tableau et en l'échangeant avec le dernier élément de cette partie.

1.1 Fonctionnement

1.2 Principe

—On parcourt le tableau et on cherche l'élément ayant la plus grande clé.
—On échange cet élément avec le dernier élément non trié.
—On répète ce processus jusqu'à ce que le tableau soit entièrement trié.

Bien que cet algorithme soit facile à comprendre et implémenter, il est inefficace pour les grands tableaux en raison de sa complexité quadratique.

1.3 Complexité

1.3.1 Pire Cas ($O(n^2)$)

- Le pire cas se produit **quelle que soit la disposition initiale du tableau** car le tri par sélection **parcourt toujours toute la partie non triée**.
- La **boucle extérieure** s'exécute en **n - 1 fois**.
- La **boucle intérieure** effectue **$(n - 1) + (n - 2) + ... + 1 = n(n - 1) / 2 = O(n^2)$**.

1.3.2 Cas Moyen ($O(n^2)$)

- Comme le tri sélection ne dépend pas de l'ordre initial, le **nombre total de comparaisons reste le même** que dans le pire cas.
- **Même raisonnement que le pire** : **$O(n^2)$**.

1.3.3 Meilleur Cas ($O(n^2)$)

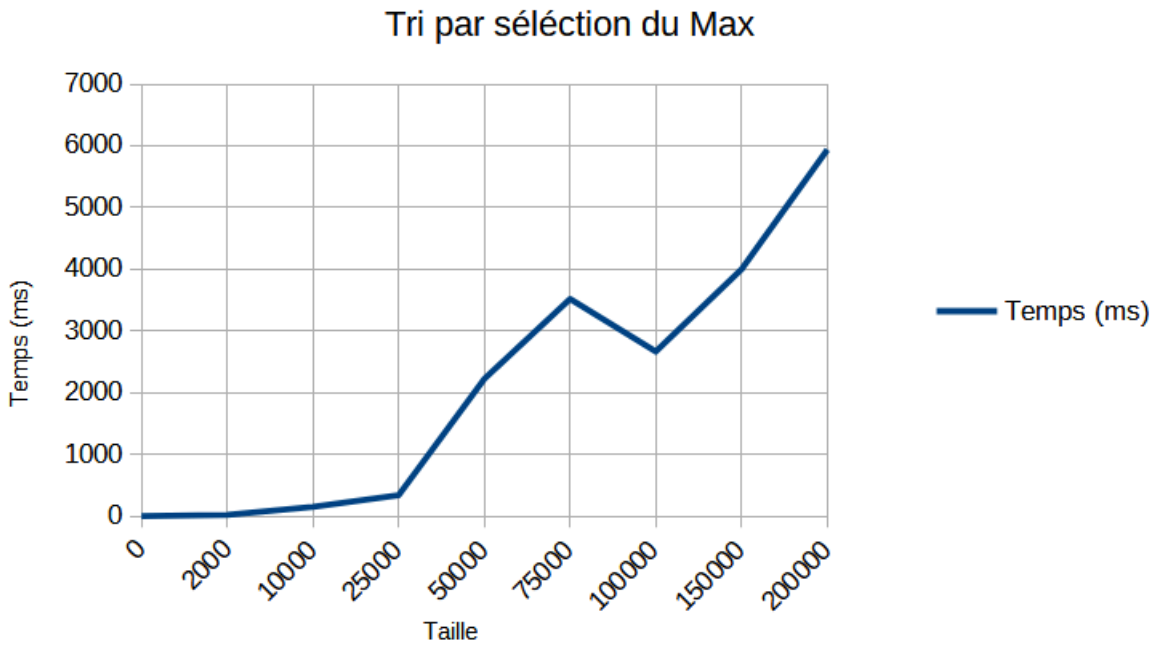
- Même si le tableau est **déjà trié**, **le tri continue de parcourir toutes les itérations de la boucle extérieure et intérieure, vérifiant chaque élément.**
- **Même complexité que les autres cas : $O(n^2)$**.

Le tri par sélection du maximum effectue $O(n^2)$ comparaisons dans tous les cas, ce qui le rend inefficace pour les grands tableaux.

1.4 Implémentation en Java

```java
// Tri par sélection du maximum
public static void selectionMax(Element[] tab) {
    int indice_max;
    int indice_fin = tab.length - 1;

    for (int j = 1; j < tab.length; j++) {
        indice_max = 0;

        // Trouver l'élément avec la clé maximale
        for (int i = 1; i <= indice_fin; i++) {
            totalComparaisons++;
            if (tab[i].getCle() > tab[indice_max].getCle()) {
                indice_max = i;
            }
        }

        // Échanger l'élément maximum avec la fin de la partie non triée
        echanger(tab, indice_max, indice_fin);
        totalAffectations += 3;

        indice_fin--; // Réduire la partie non triée
    }
}
```

1.5 Graphe



1.6 Analyse du graphe

Cette courbe illustre l'évolution du temps d'exécution du tri par sélection du maximum en fonction de la taille du tableau. Il est observé que, à mesure que la taille du tableau augmente, le temps d'exécution croît de manière exponentielle, en particulier pour les grandes tailles. Cela est dû à la complexité algorithmique de cet algorithme, qui est de l'ordre de $O(n^2)$, rendant ainsi son efficacité limitée pour les ensembles de données volumineux. Pour des tableaux de petite taille, le tri reste relativement rapide, mais dès que la taille du tableau augmente, le temps de traitement se prolonge de façon significative. Lorsque la taille dépasse les 75 000 éléments, le temps d'exécution devient excessivement long, ce qui démontre que cet algorithme n'est pas adapté pour traiter de grandes quantités de données. En raison de cette lenteur, il est rarement employé dans des situations nécessitant le tri de grands ensembles de données.

2 Tri rapide (QuickSort)

Le tri rapide est un algorithme basé sur le paradigme "diviser pour régner". Il utilise un **pivot** pour diviser le tableau en deux sous-tableaux :

- Les éléments inférieurs ou égaux au pivot à gauche.
- Les éléments supérieurs au pivot à droite.

Puis, l'algorithme s'exécute récursivement sur chaque sous-tableau jusqu'à ce que tout soit trié.

2.1 Fonctionnement

2.2 Principe

1.Trouver le pivot qui est le premier élément dans le tableau.

2. Réorganisation des éléments : Deux indices sont utilisés pour parcourir le tableau :

- **L'indice `i` avance de gauche à droite** pour trouver un élément **plus grand que le pivot**.
- **L'indice `j` recule de droite à gauche** pour identifier un élément **plus petit ou égal au pivot**.

Lorsque ces deux indices repèrent des valeurs qui ne sont pas à leur place, elles sont **échangées**.

Le processus se poursuit jusqu'à ce que `i` et `j` se croisent. À ce moment-là, le **pivot est placé à sa position définitive**.

3. Découpage et récursion : Une fois le pivot correctement positionné, le tri est appliqué **récursivement** sur les deux parties obtenues :

- **La partie gauche**, contenant les valeurs plus petites.
- **La partie droite**, avec les valeurs plus grandes.

Cette division continue **jusqu'à ce que chaque sous-tableau ne contienne qu'un seul élément**, ce qui signifie qu'il est trié

4. Suivi des performances :

Des variables sont utilisées pour **compter le nombre de comparaisons et d'échanges effectués** pendant l'exécution de l'algorithme.

Une fois le tri terminé, ces statistiques sont affichées à l'écran

2.3 Complexité

2.3.1 Pire Cas ($O(n^2)$)

- Si le tableau est **déjà trié** ou **quasi trié**, **le pivot choisi est toujours le plus petit ou le plus grand élément**.
- Cela entraîne **une partition déséquilibrée** : chaque partition contient **n - 1 élément** et **0 élément**.
- **Nombre total d'appels récursifs : n → $O(n^2)$**

2.3.2 Cas Moyen (O(n log n))

- En moyenne, le pivot divise le tableau en **deux sous-tableaux de taille ≈ n/2**.
- À chaque niveau, on fait **O(n) comparaisons**, et **La récursion est logarithmique log n**.
- **Complexité totale : O(n log n)**.

2.3.3 Meilleur Cas (O(n log n))

- Si le tableau est toujours parfaitement **divisé en deux sous-tableaux égaux**, alors on aura :
  - **La récursion est logarithmique log n**
  - **O(n) opérations à chaque niveau**
- **Complexité totale : O(n log n)**.

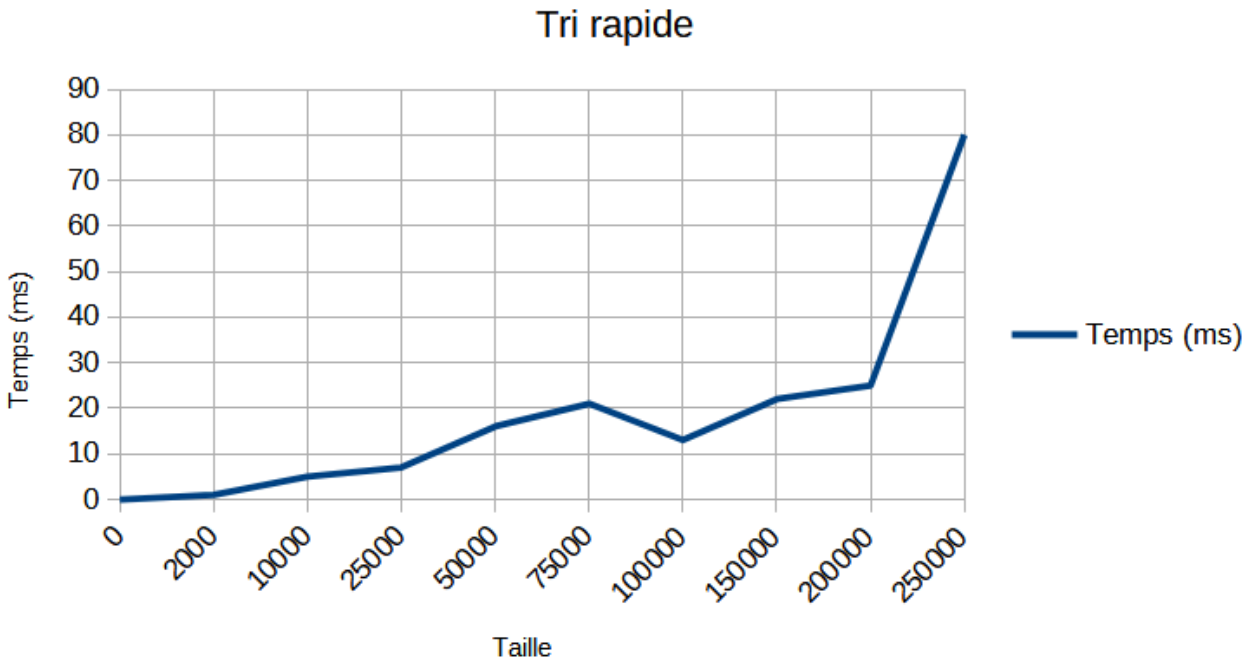
Dans le meilleur des cas, la complexité est $O(n \log n)$, mais elle peut être $O(n^2)$ dans le pire des cas si le pivot est mal choisi.

2.4 Implémentation en java

```java
// Tri rapide (QuickSort)
public static void triRapide(Element[] tab, int debut, int fin) {
    if (debut < fin) {
        int pivotIndex = partition(tab, debut, fin);
        triRapide(tab, debut, pivotIndex - 1);
        triRapide(tab, pivotIndex + 1, fin);
    }

    // Affichage des résultats après le dernier appel récursif
    if (debut == 0 && fin == tab.length - 1) {
        System.out.println("************************** TRI RAPIDE : **********************");
        System.out.println("Comparaisons tri rapide : " + totalComparaisons);
        System.out.println("Affectations tri rapide : " + totalAffectations);
        System.out.println("Opérations élémentaires tri rapide : " + (totalComparaisons + totalAffectations));
    }
}
```

```java
private static int partition(Element[] tab, int debut, int fin) {
    int pivot = tab[debut].getCle();
    int i = debut + 1;
    int j = fin;

    while (i <= j) {
        // Trouver un élément plus grand que le pivot à gauche
        while (i <= fin && tab[i].getCle() <= pivot) {
            i++;
            totalComparaisons++;
        }

        // Trouver un élément plus petit ou égal au pivot à droite
        while (tab[j].getCle() > pivot) {
            j--;
            totalComparaisons++;
        }

        // Échanger si i < j
        if (i < j) {
            echanger(tab, i, j);
            totalAffectations += 3;
        }
    }

    // Placer le pivot à sa position finale
    echanger(tab, debut, j);
    totalAffectations += 3;

    return j;
}
```

2.5 Graphe



2.6 Analyse du graphe

La courbe présentée montre l'évolution du temps d'exécution du tri rapide en fonction de la taille du tableau. Pour des tableaux relativement petits (jusqu'à 10 000 éléments), le temps d'exécution reste très court, atteignant seulement 4 ms pour un tableau de 10 000 éléments. Même lorsque la taille du tableau augmente, le temps d'exécution reste modéré, avec 13 ms pour 100 000 éléments et 22 ms pour 150 000 éléments. Cette performance est en adéquation avec la complexité moyenne du tri rapide, qui est de O(n log n). Toutefois, il est important de noter que dans des scénarios défavorables, comme lorsque le tableau est déjà trié, le tri rapide peut voir ses performances se dégrader. Dans de telles situations, la complexité peut atteindre $O(n^2)$. Cette dégradation n'apparaît pas sur la courbe, car les tests ont probablement été réalisés sur des données aléatoires, correspondant ainsi au cas moyen.

3 Tri par dénombrement

Le tri par dénombrement est un algorithme non comparatif qui fonctionne en comptant le nombre d'occurrences de chaque clé et en déterminant leur position finale dans le tableau trié, si deux clés ou plus sont identiques alors il garde leurs positions initiales.

3.1 Fonctionnement

3.2 Principe

Initialisation :

- Un compteur `comp` pour les comparaisons et un tableau `counts` pour les occurrences sont définis.

Vérification des conditions initiales

- Si le tableau `tab` est nul ou vide, la fonction termine immédiatement sans effectuer de tri.

Identification de la valeur maximale (`max`) :

- `max` est initialisé à la première clé. On parcourt le tableau pour trouver la valeur maximale, en comptabilisant les comparaisons dans `comp` et les affectations dans `aff`.

Création du tableau de comptage (`counts`) :

- Un tableau `counts` de taille `max + 1` est créé pour compter le nombre d'occurrences de chaque clé dans le tableau `tab`.
- On initialise toutes les cases du tableau `counts` à 0, et cela est comptabilisé dans le compteur `aff`.

Comptage des occurrences :

- On parcourt chaque élément du tableau `tab`. Pour chaque élément, on incrémente la case correspondante dans le tableau `counts` (c'est-à-dire,

pour chaque clé `cle`, on incrémente `counts[cle]`).

Mise à jour du tableau de comptage :

- On parcourt le tableau `counts` et on met à jour chaque case en ajoutant la valeur de la case précédente à la case actuelle. Cette étape permet de préparer le tableau de comptage pour déterminer la position finale de chaque élément dans le tableau trié.

Reconstruction du tableau trié :

- Un tableau temporaire `res` est créé pour stocker les éléments triés.
- On parcourt le tableau `tab` de la fin vers le début. Cette approche permet de garantir la stabilité du tri, c'est-à-dire de conserver l'ordre relatif des éléments ayant la même clé.
- Pour chaque élément du tableau `tab`, on consulte le tableau `counts` pour connaître sa position dans le tableau `res`, on y place l'élément, et on décrémente la valeur correspondante dans `counts`.

Copie des éléments triés dans le tableau d'origine :

Après avoir reconstruit le tableau trié dans `res`, on copie les éléments du tableau `res` dans le tableau `tab` pour que ce dernier contienne les valeurs triées.

3.3 Complexité

3.3.1 Pire Cas (O(n + k))

Lorsque les valeurs des clés sont très dispersées, c'est-à-dire lorsque $k \gg n$, l'allocation du tableau de comptage devient coûteuse. Ce cas se produit lorsqu'on doit trier un grand nombre d'éléments dans une plage de valeurs très large.

Exemple

trier un million de nombres entre 0 et un milliard nécessite une allocation mémoire importante pour le tableau de comptage, ce qui rend l'algorithme

moins efficace.

Complexité

- **Phase 1** : Le comptage des occurrences prend **O(n)**.
- **Phase 2** : Le parcours du tableau de comptage pour reconstruire le tableau trié prend **O(k)**.

Si k devient beaucoup plus grand que n, l'algorithme devient très long en raison de la mémoire et du temps nécessaires pour traiter k.

3.3.2 Cas Moyen (O(n + k))

- Le nombre d'opérations est linéaire **O(n + k)** tant que **k** reste raisonnable.

3.3.3 Meilleur Cas (O(n))

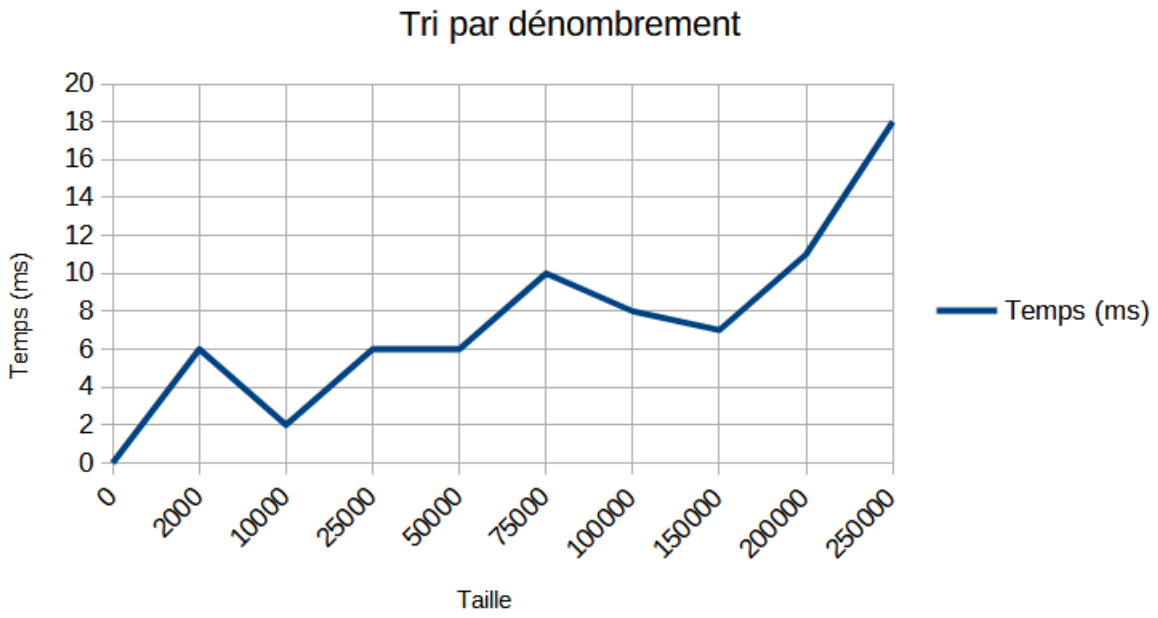
- Si **k est proche de n**, le tri se fait en **temps linéaire O(n+k)**.

3.4 Implémentation en Java

```java
// Tri par dénombrement (conservé)
public static void denombre(Element[] tab) {
    int comp = 0; //nombre de comparaisons

    if (tab == null || tab.length == 0) {
        return;
    }

    int max = tab[0].getCle();//On compte pas ces initialisation(choix conventionnelle)
    int aff = 0; //nombre d'affectations
    for (int i = 1; i < tab.length; i++) {
        comp++;
        if (tab[i].getCle() > max) {
            max = tab[i].getCle();
            aff++;
        }
    }

    int[] counts = new int[max + 1]; //Création du tableau d'occurrences

    for (int i=0;i<counts.length;i++) {
        counts[i]=0;
        aff++;
    }

    for(int j = 0; j < tab.length; j++) {//Calcule des occurrences
        counts[tab[j].getCle()]++;
    }

    for(int k = 1; k <= max; k++) { //mise à jour du tableau
        counts[k] = counts[k] + counts[k-1];
        aff++;
    }
```

```java
// On crée le tableau
Element[] res = new Element[tab.length];

for(int p = tab.length - 1; p >= 0; p--) { //le tableau trié
    res[--counts[tab[p].getCle()]] = tab[p];
    aff++;
}

for(int i = 0; i < tab.length; i++) { //On copie les élèments dans notre tableau de base
    tab[i] = res[i];
    aff++;
}
```

3.5 Graphe

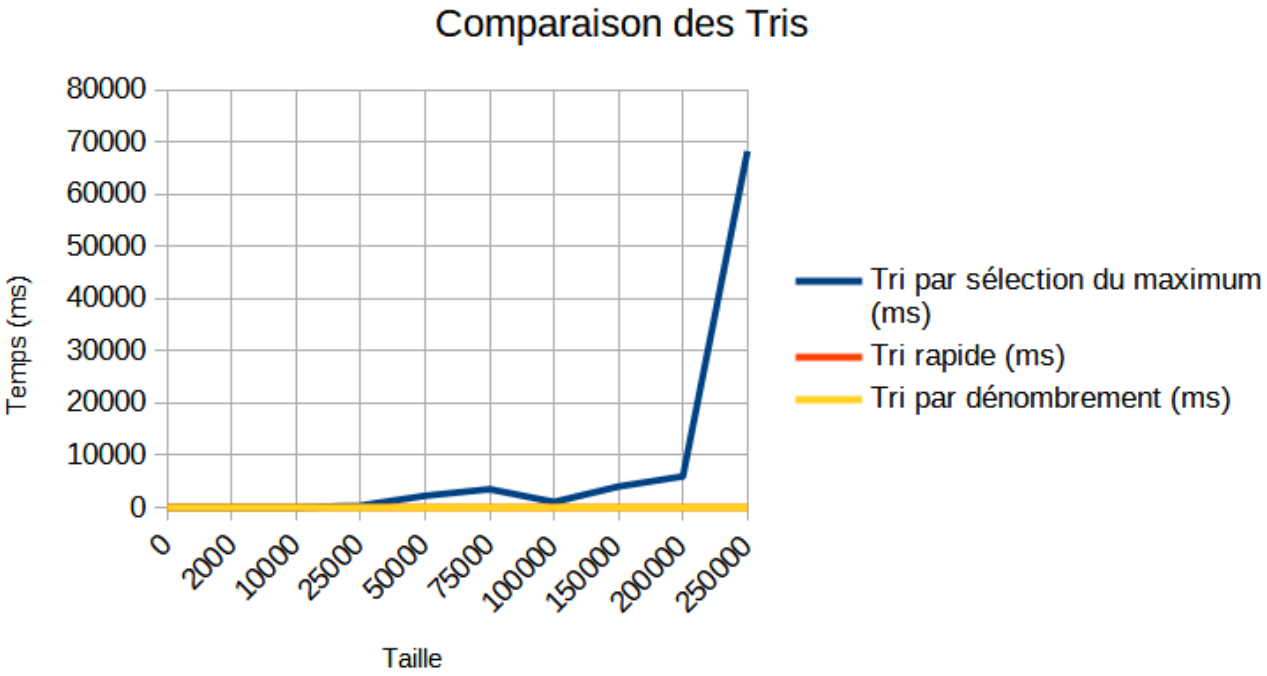


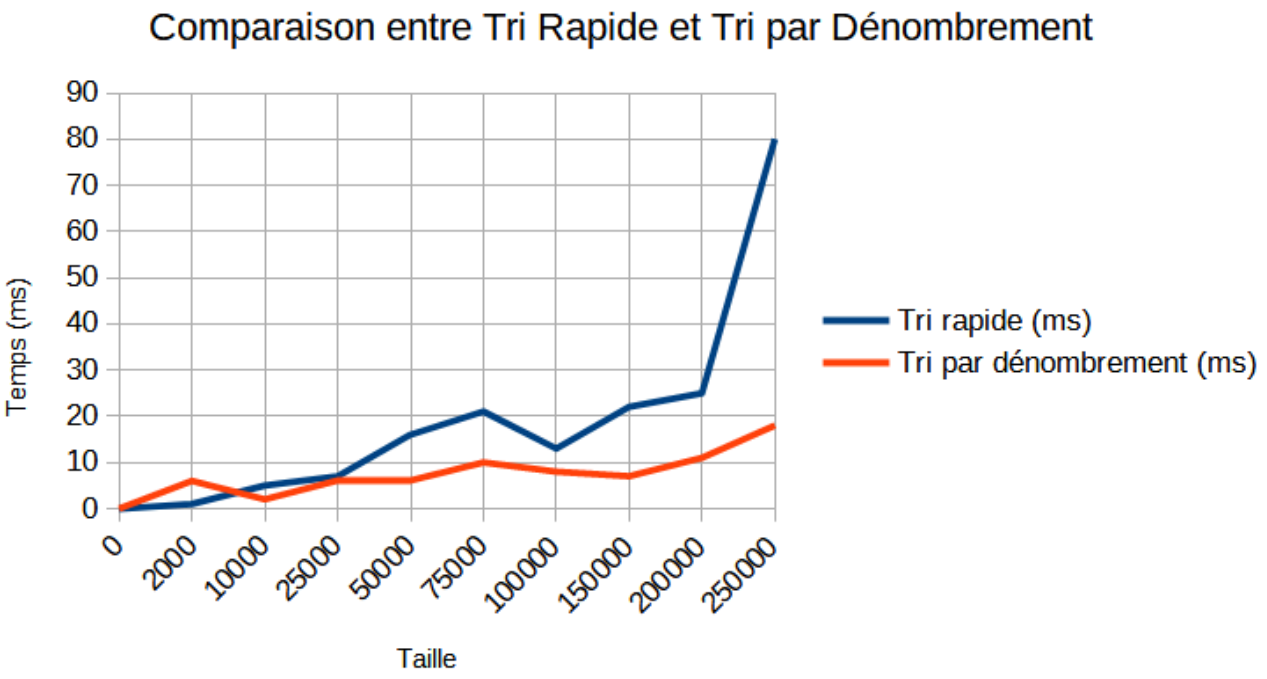
3.6 Analyse du graphe

La courbe présentée montre que le tri par dénombrement reste extrêmement rapide, même pour des tableaux de grande taille. Par exemple, pour un tableau de 100 000 éléments, le temps d'exécution est de seulement 8 ms, et pour 250 000 éléments, il atteint 18 ms. Ces résultats sont en adéquation avec la complexité linéaire O(n+k) de l'algorithme, où n est le nombre d'éléments du tableau et k la plage des valeurs possibles.

Cependant, il est important de souligner que cette efficacité est limitée par la taille de k. Si la plage des valeurs possibles (k) est très grande, la complexité peut augmenter considérablement. En conséquence, bien que le tri par dénombrement soit particulièrement adapté pour trier rapidement de grands ensembles de données lorsque la plage de valeurs est étroite, il devient moins performant lorsque cette plage devient trop large.

4 Comparaison des tris





Ces courbes illustrent la comparaison des temps d'exécution des algorithmes de tri rapide, de tri par dénombrement et de tri par sélection du maximum en fonction de la taille du tableau.

On constate que le **tri par dénombrement** se démarque par son efficacité, maintenant un **temps d'exécution très faible**, même pour des ensembles de grande taille. Ce comportement est en accord avec sa **complexité en O(n + k)**, qui le rend particulièrement performant lorsque la plage des valeurs possibles reste restreinte.

Le **tri rapide**, quant à lui, présente également de **bonnes performances**, avec une **croissance modérée du temps d'exécution**. Sa complexité moyenne en **O(n log n)** en fait un choix équilibré et performant pour le tri de données générales.

À l'inverse, le **tri par sélection du maximum** est clairement **moins efficace**, car son **temps d'exécution augmente quadratiquement ($O(n^2)$)**, le rendant **inefficace aux grandes tailles de données**.

Cette analyse met en évidence que le **tri par dénombrement** est le plus rapide dans des cas spécifiques où la plage des valeurs reste limitée, tandis que le **tri rapide** constitue un **compromis performant** pour des données variées. En revanche, le **tri par sélection du maximum** est peu adapté aux grandes structures en raison de sa **complexité élevée**, ce qui limite son usage en pratique.

5 Analyse de la Complexité des Algorithmes

Algorithme	Meilleur Cas	Cas Moyen	Pire Cas
Tri par Sélection du Max	$O(n^2)$	$O(n^2)$	$O(n^2)$
Tri Rapide (QuickSort)	O(n log n)	O(n log n)	$O(n^2)$
Tri par Dénombrement	O(n + k)	O(n + k)	O(n + k)

Le **tri rapide** est généralement le plus performant grâce à sa complexité **O(n log n)** en moyenne. Cependant, dans le pire cas (tableau déjà trié), sa complexité devient **$O(n^2)$**.

Le **tri par dénombrement** est le plus rapide lorsque les clés sont limitées en plage, avec une complexité linéaire **O(n + k)**.

Le **tri par sélection du maximum**, en revanche, reste inefficace pour de grands tableaux en raison de sa complexité**(n^2)**.

6 Résultats Expérimentaux

Nous avons testé les trois algorithmes sur différentes tailles de tableaux et mesuré leur **temps d'exécution**.

Taille du tableau	Tri par sélection du maximum (ms)	Tri rapide (ms)	Tri par dénombrement (ms)
0	0	0	0
2000	17	1	6
10000	145	5	2
25000	338	7	6
50000	2220	16	6
75000	3515	21	10
100000	1061	13	8
150000	3999	22	7
200000	5937	25	11
250000	68246	80	18

7 Conclusion

Notre étude a mis en évidence les forces et faiblesses des trois algorithmes :

- **Tri par sélection du maximum** : simple à implémenter mais inefficace pour de grandes tailles.
- **Tri rapide** : un bon compromis, rapide en moyenne mais sensible aux pires cas.
- **Tri par dénombrement** : extrêmement rapide mais dépendant de la plage des clés.

Ainsi, pour des **grands tableaux avec une large plage de valeurs**, le **tri rapide** est recommandé. Pour des **clés limitées en plage**, le **tri par dénombrement** est idéal. En revanche, le **tri par sélection du maximum** est peu adapté aux grandes tailles.

Ces résultats confirment que le choix d'un algorithme de tri doit être fait en fonction des caractéristiques des données à trier.

8 Références

- Wikipedia. ”Algorithme de tri.” Wikipedia, The Free Encyclopedia. [En ligne] Disponible sur : https://fr.wikipedia.org/wiki/Algorithme_de_tri. (Consulté le 10 Mars 2025).

- Interstices.”Les algorithmes de tri.” [En ligne] Disponible sur : https://interstices.info/les-algorithmes-de-tri/. (Consulté le 10 Mars 2025).

- GeeksforGeeks. ”Time Complexities of all Sorting Algorithms.” [En ligne] Disponible sur : https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/. (Consulté le 11 Mars 2025).