

Rapport d'Analyse : Étude Comparative des Algorithmes de Tri

Sujet : Analyse d'un programme C pour la mesure et la comparaison des performances de plusieurs algorithmes de tri.

Table des Matières

1. Introduction

- 1.1. Contexte et Importance du Tri
- 1.2. Objectif du Projet
- 1.3. Algorithmes Étudiés
- 1.4. Structure du Rapport

2. Méthodologie et Vue d'Ensemble du Code

- 2.1. Structure Générale du Programme
- 2.2. Génération des Données
- 2.3. Mesure du Temps d'Exécution
- 2.4. Génération des Scripts GNUPlot

3. Analyse Détaillée des Algorithmes de Tri

- 3.1. Algorithmes de Complexité Quadratique ($O(n^2)$)
 - 3.1.1. Tri à Bulles (Bubble Sort)
 - 3.1.2. Tri par Insertion (Insertion Sort)
 - 3.1.3. Tri par Sélection (Selection Sort)
- 3.2. Algorithmes de Complexité Quasi-Linéaire ($O(n \log n)$)
 - 3.2.1. Tri Rapide (Quick Sort)
 - 3.2.2. Tri Fusion (Merge Sort)
 - 3.2.3. Tri de Shell (Shell Sort)
 - 3.2.4. TimSort

4. Analyse du Code C et Optimisations

- 4.1. Gestion de la Mémoire
- 4.2. Immutabilité des Données de Test
- 4.3. Gestion des Cas Extrêmes ($O(n^2)$)

- 4.4. Portabilité et Automatisation

5. Résultats Attendus et Interprétation

- 5.1. Performance des Algorithmes $O(n^2)$
- 5.2. Performance des Algorithmes $O(n \log n)$
- 5.3. Analyse Comparative
- 5.4. Impact de la Constante Masquée

6. Conclusion

7. Annexe : Code Source Complet

1. Introduction

1.1. Contexte et Importance du Tri

Le tri est l'une des opérations les plus fondamentales en informatique. Il s'agit de ranger les éléments d'un ensemble dans un ordre défini (numérique, lexicographique, etc.). Une grande partie des algorithmes plus complexes, comme la recherche binaire, dépendent de la précondition que les données soient triées. L'efficacité d'un algorithme de tri est donc un sujet d'étude majeur, car elle a un impact direct sur les performances de nombreuses applications.

1.2. Objectif du Projet

Ce rapport présente une analyse détaillée d'un programme C conçu pour effectuer une étude comparative des performances de plusieurs algorithmes de tri classiques. L'objectif est de :

- Implémenter divers algorithmes de tri.
- Mesurer leur temps d'exécution sur des ensembles de données de taille croissante.
- Analyser les résultats empiriques pour les confronter à la complexité théorique de chaque algorithme.
- Générer des graphiques pour visualiser les différences de performance.

1.3. Algorithmes Étudiés

Le programme analyse huit algorithmes de tri, regroupés par complexité théorique :

- **Complexité $O(n^2)$** : Tri à Bulles, Tri par Insertion, Tri par Sélection.
- **Complexité $O(n \log n)$** : Tri Rapide, Tri Fusion, Tri de Shell, TimSort.

1.4. Structure du Rapport

Ce rapport est organisé comme suit : la section 2 décrit la méthodologie utilisée. La section 3 détaille chaque algorithme implémenté. La section 4 analyse les aspects techniques et les choix d'implémentation du code C. La section 5 interprète les résultats attendus, et la section 6 conclut l'étude.

2. Méthodologie et Vue d'Ensemble du Code

2.1. Structure Générale du Programme

Le code est structuré de manière modulaire, favorisant la lisibilité et la maintenance :

- **Déclarations des fonctions** : Une section claire en tête de fichier pour lister les prototypes.
- **Implémentation des algorithmes** : Le cœur du programme, où chaque algorithme de tri est implémenté dans sa propre fonction.
- **Fonctions utilitaires** : Des fonctions pour la génération de données (`generateRandomArray`) et la mesure du temps (`measureTime` , `measureTimeQuick` , `measureTimeMerge`).
- **Fonction principale (`main`)** : Orchestre l'ensemble du processus : itération sur les tailles de tableau, lancement des mesures, affichage des résultats et génération des scripts pour GNUPlot.

2.2. Génération des Données

Pour garantir l'équité des tests, chaque algorithme est testé sur le même ensemble de données pour une taille donnée. La fonction `generateRandomArray` crée un tableau d'entiers aléatoires. La graine du générateur de nombres aléatoires est initialisée avec `srand(time(NULL))` pour s'assurer que chaque exécution du programme utilise des données différentes, évitant ainsi les biais liés à un jeu de données spécifique.

2.3. Mesure du Temps d'Exécution

La mesure du temps est effectuée à l'aide de la fonction `clock()` de la bibliothèque `<time.h>`. La méthodologie est rigoureuse :

1. **Copie du tableau original** : Pour chaque algorithme, une copie du tableau de données original est créée avec `memcpy`. C'est crucial pour garantir que chaque algorithme travaille sur des données non triées et identiques.
2. **Calcul du temps CPU** : La différence `clock() - start` mesure le temps CPU utilisé, ce qui est plus précis que le temps réel (wall-clock time), car il n'est pas affecté par d'autres processus s'exécutant sur la machine.
3. **Fonctions de mesure dédiées** : Des fonctions `measureTime`, `measureTimeQuick` et `measureTimeMerge` sont créées pour gérer les signatures de fonction légèrement différentes des algorithmes (par ex., `quickSort` et `mergeSort` prennent des indices de début et de fin).

2.4. Génération des Scripts GNUPlot

Le programme ne se contente pas d'afficher des résultats textuels. Il génère automatiquement trois scripts GNUPlot (`.gnu`) pour créer des graphiques PNG. Cette automatisation est un atout majeur pour l'analyse visuelle des résultats :

- `plot.gnu` : Un graphique général comparant tous les algorithmes.
- `plot_efficient.gnu` : Un zoom sur les algorithmes efficaces ($O(n \log n)$).
- `plot_complexity.gnu` : Une comparaison directe entre un algorithme $O(n^2)$ (Bubble Sort) et un algorithme $O(n \log n)$ (Quick Sort).

3. Analyse Détaillée des Algorithmes de Tri

3.1. Algorithmes de Complexité Quadratique ($O(n^2)$)

Ces algorithmes sont simples à comprendre et à implémenter, mais leur performance se dégrade rapidement avec l'augmentation de la taille des données.

- **3.1.1. Tri à Bulles (Bubble Sort)** : Il parcourt répétitivement la liste, compare les éléments adjacents et les échange s'ils sont dans le mauvais ordre. Les plus grands éléments "remontent" comme des bulles. Il est connu pour son inefficacité, même pour des ensembles de données de taille modeste.
- **3.1.2. Tri par Insertion (Insertion Sort)** : Il construit le tableau trié final un élément à la fois. Il prend chaque élément non trié et l'insère à sa place correcte dans la partie déjà triée du tableau. Il est très efficace pour les petites listes ou les listes qui sont déjà presque triées.
- **3.1.3. Tri par Sélection (Selection Sort)** : Il divise le tableau en deux parties : une partie triée à gauche et une partie non triée à droite. Il répétitivement trouve le plus petit élément de la partie non triée et le déplace à la fin de la partie triée.

3.2. Algorithmes de Complexité Quasi-Linéaire ($O(n \log n)$)

Ces algorithmes sont beaucoup plus performants sur de grands ensembles de données et sont généralement basés sur le paradigme "diviser pour régner".

- **3.2.1. Tri Rapide (Quick Sort)** : C'est l'un des algorithmes de tri les plus rapides en pratique. Il choisit un élément comme "pivot" et partitionne le tableau autour de ce pivot, de sorte que tous les éléments plus petits soient avant le pivot et tous les éléments plus grands soient après. Il applique ensuite récursivement cette partition aux sous-tableaux.
- **3.2.2. Tri Fusion (Merge Sort)** : C'est un algorithme "diviser pour régner" exemplaire. Il divise le tableau en deux moitiés, trie récursivement chaque moitié, puis fusionne les deux moitiés triées. Sa performance est garantie en $O(n \log n)$, mais il nécessite de la mémoire supplémentaire pour stocker les sous-tableaux temporaires.

- **3.2.3. Tri de Shell (Shell Sort) :** C'est une généralisation du tri par insertion. Il commence par trier des paires d'éléments éloignés d'un certain "gap" (écart), puis progressivement réduit cet écart jusqu'à ce qu'il soit de 1, à quel point le tableau est trié. C'est un tri de comparaison efficace pour les tailles moyennes de données.
- **3.2.4. TimSort :** C'est un algorithme de tri hybride, dérivé de Tri Fusion et Tri par Insertion. Il est conçu pour performer très bien sur de nombreux types de données du monde réel. Il fonctionne en divisant le tableau en petits blocs (appelés "runs"), en les triant avec Tri par Insertion, puis en fusionnant ces runs avec une stratégie de fusion optimisée. C'est l'algorithme de tri par défaut en Python et en Java.

4. Analyse du Code C et Optimisations

4.1. Gestion de la Mémoire

Le code gère la mémoire de manière responsable. Chaque allocation avec `malloc` (dans `measureTime*` et `merge`) est accompagnée d'une vérification pour s'assurer que l'allocation a réussi. De plus, chaque bloc alloué est libéré avec `free` pour éviter les fuites de mémoire.

4.2. Immutabilité des Données de Test

Comme mentionné précédemment, l'utilisation de `memcpy` pour créer une copie temporaire du tableau avant chaque tri est une décision de conception cruciale. Elle garantit que chaque algorithme est testé dans des conditions identiques, ce qui est essentiel pour une comparaison juste.

4.3. Gestion des Cas Extrêmes ($O(n^2)$)

Le code intègre une optimisation pratique : les algorithmes $O(n^2)$ ne sont testés que pour des tailles de tableau jusqu'à 16 000. Pour des tailles supérieures, leur temps d'exécution serait prohibitif (plusieurs minutes ou heures), rendant l'expérience utilisateur déplaisante. Le programme enregistre `-1.0` pour ces tests, ce qui permet à GNUPlot d'ignorer ces points de données grâce à la condition `($2>0? $2:1/0)`.

4.4. Portabilité et Automatisation

Le code n'utilise que des bibliothèques C standard (`stdio.h`, `stdlib.h`, `time.h`, `string.h`), ce qui le rend hautement portable sur différents systèmes d'exploitation. La génération de scripts GNUPlot est une excellente fonctionnalité qui automatise la phase de visualisation, faisant du programme un outil d'analyse complet.

5. Résultats Attendus et Interprétation

5.1. Performance des Algorithmes $O(n^2)$

Pour les petites tailles ($n < 1000$), les différences seront minimales. Cependant, à mesure que n augmente, le temps d'exécution des algorithmes $O(n^2)$ augmentera de manière quadratique. On s'attend à ce que Bubble Sort soit le plus lent, suivi de Selection Sort, et Insertion soit un peu plus rapide.

5.2. Performance des Algorithmes $O(n \log n)$

Ces algorithmes montreront une croissance beaucoup plus lente du temps d'exécution. Sur un graphique en échelle logarithmique, leurs courbes seront presque des lignes droites.

- **Quick Sort** est souvent le plus rapide en moyenne, mais sa performance peut se dégrader en $O(n^2)$ dans le pire des cas (si le pivot est mal choisi à chaque fois).
- **Merge Sort** offrira des performances très stables et prévisibles, garantissant toujours une complexité $O(n \log n)$, au prix d'une utilisation mémoire plus élevée.
- **TimSort** devrait être extrêmement compétitif, souvent surclassant les autres sur des données réalistes, car il est optimisé pour les données partiellement triées.
- **Shell Sort** offrira de bonnes performances, se situant généralement entre les algorithmes $O(n^2)$ et les meilleurs algorithmes $O(n \log n)$.

5.3. Analyse Comparative

Le graphique général montrera clairement deux "familles" de courbes : les courbes très pentues des algorithmes $O(n^2)$ et les courbes beaucoup plus plates des algorithmes $O(n \log n)$. Le graphique [complexity_comparison.png](#) illustrera de manière spectaculaire l'avantage de la complexité $O(n \log n)$ sur $O(n^2)$ pour de grandes tailles de données.

5.4. Impact de la Constante Masquée

La notation Big-O ignore les constantes multiplicatives. En pratique, pour de petites tailles de données, un algorithme $O(n^2)$ avec une petite constante peut être plus rapide qu'un algorithme $O(n \log n)$ avec une grande constante (due à une plus grande complexité de mise en œuvre, des appels récurrents, etc.). Les résultats pour les petites tailles de tableau ($n < 4000$) illustreront cet effet.

6. Conclusion

Le programme C analysé est un excellent outil pédagogique et pratique pour l'étude des algorithmes de tri. Il démontre avec succès comment implémenter une variété d'algorithmes, mesurer leurs performances de manière rigoureuse et visualiser les résultats de manière efficace.

L'analyse théorique et les résultats empiriques attendus convergent vers une conclusion claire : le choix d'un algorithme de tri est critique et dépend fortement de la taille et de la nature des données. Pour les petites listes, des algorithmes simples comme le Tri par Insertion peuvent être suffisants. Cependant,

pour des applications traitant de grands volumes de données, l'utilisation d'algorithmes efficaces comme Quick Sort, Merge Sort ou TimSort est indispensable pour garantir des performances acceptables. Ce projet illustre parfaitement le passage de la théorie de la complexité algorithmique à la pratique de la mesure de la performance.