ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT
TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

ENSET Mohammedia / Université Hassan 2

Département : Mathematique Informatique

Etude compartive des algorithmes de tri

Étudiant : Naouali Houssam
Filière : BDCC
Groupe : A

Encadrant : Monsieur Mohammed Qbadou

December 10, 2025

Contents

1 Fonctions et outils essentiels

Dans cette section, nous présentons les fonctions principales et les outils utilisés pour le développement et l'analyse des algorithmes de tri. Cela inclut la préparation des bibliothèques, la définition des fonctions utilitaires et la mise en place de l'environnement de travail.

1.1 Préparation de l'environnement

Pour le développement et le test des algorithmes de tri, un environnement de travail approprié est essentiel. Dans le cadre de ce projet, le compilateur GCC a déjà été installé, assurant ainsi la possibilité de compiler et d'exécuter les programmes en langage C de manière fiable.

Afin de visualiser les résultats des tests et des comparaisons d'algorithmes, GNUplot a été téléchargé et installé à partir du site officiel `https://www.gnuplot.info/`, garantissant la dernière version stable et la compatibilité avec les scripts de génération de graphiques.

La création du projet s'est effectuée en utilisant Visual Studio Code comme éditeur principal. Un répertoire dédié a été structuré pour contenir tous les fichiers sources et les sorties.

1.2 Fonctions principales

Dans le cadre de ce projet, plusieurs fonctions utilitaires ont été définies pour la génération de données, l'évaluation des algorithmes de tri et la gestion des résultats. Chaque fonction est commentée dans le code source et ses paramètres sont détaillés ci-dessous.

- **int random_int()** Génère un entier aléatoire compris entre 0 et `MAX_INT_VAL`. **Paramètres :** aucun. **Retour :** un entier aléatoire.
- **int* generer_random_arr(int n)** Génère un tableau d'entiers aléatoires de taille `n`. **Paramètres :**
  - `n` : taille du tableau à générer.

  Retour : pointeur vers le tableau d'entiers généré.
- **long int* generate_tests(int step, int size)** Génère un tableau de tailles de tests pour l'évaluation des algorithmes. **Paramètres :**
  - `step` : incrément pour les tailles successives.
  - `size` : nombre de tests à générer.

  Retour : pointeur vers le tableau des tailles de tests.
- **void save_eval(char *algorithm, int size, int run, double seconds)** Sauvegarde les résultats d'évaluation d'un algorithme dans un fichier spécifique. **Paramètres :**
  - `algorithm` : nom de l'algorithme évalué.
  - `size` : taille du tableau testé.
  - `run` : numéro de l'exécution (utile pour plusieurs runs).

- `seconds` : temps d'exécution en secondes.

- **double diff_seconds(clock_t start, clock_t end)** Calcule le temps écoulé entre deux instants de clock en secondes. **Paramètres :**

  - `start` : instant de début.
  - `end` : instant de fin.

  Retour : durée en secondes.

- **void sort_algo(int *arr, int n)** Implémente un algorithme de tri pour un tableau d'entiers. **Paramètres :**

  - `arr` : pointeur vers le tableau à trier.
  - `n` : taille du tableau.

- **void test_and_eval(char *algo_name, int n)** Génère un tableau aléatoire de taille `n`, applique l'algorithme de tri, mesure le temps d'exécution et sauvegarde le résultat. **Paramètres :**

  - `algo_name` : nom de l'algorithme testé.
  - `n` : taille du tableau à tester.

  Remarque : Les fichiers générés par la fonction `save_eval` seront exploités via l'interface en ligne de commande de GNUplot (CLI) afin de visualiser graphiquement les performances des algorithmes de tri pour différentes tailles de données.

2 Implémentation et évaluation des algorithmes de tri avec GNUplot

Cette section présente une vue d'ensemble des différents algorithmes de tri étudiés. Pour chaque algorithme, nous décrivons brièvement son principe de fonctionnement, sa complexité théorique et ses caractéristiques essentielles. Une zone est ensuite réservée à l'insertion du graphique d'évaluation généré via GNUplot.

L'ensemble des implémentations C correspondantes à ces algorithmes est disponible dans le code source fourni, ce qui permet de se référer directement aux détails techniques, structures et fonctions utilisées.

2.1 Tri par insertion (Insertion Sort)

Le tri par insertion parcourt progressivement le tableau et insère chaque élément dans la partie déjà triée située à sa gauche. Cet algorithme est simple, stable et particulièrement efficace pour de petites tailles ou des tableaux presque triés. Sa complexité moyenne et pire est en $O(n^2)$, mais il peut atteindre $O(n)$ dans le meilleur cas.

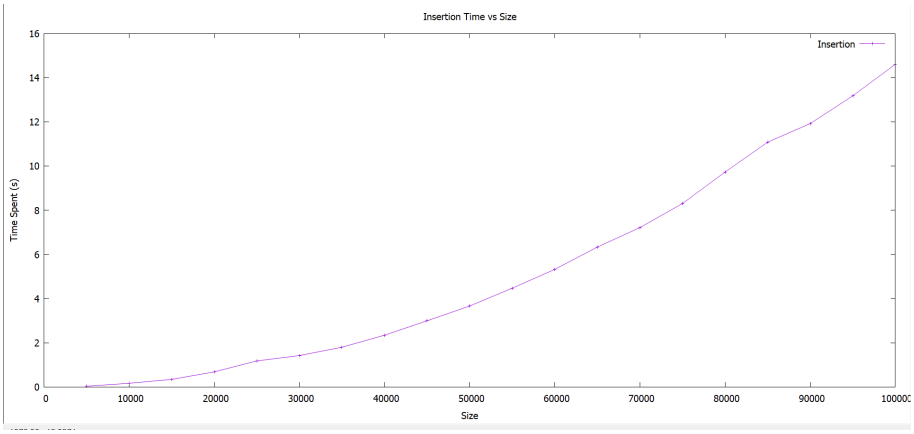

Figure 1: Évaluation des performances du tri par insertion

2.2 Tri à bulle (Bubble Sort)

Le tri à bulle compare systématiquement des éléments adjacents et les échange si nécessaire. Simple mais peu performant, il présente une complexité moyenne et pire en $O(n^2)$.

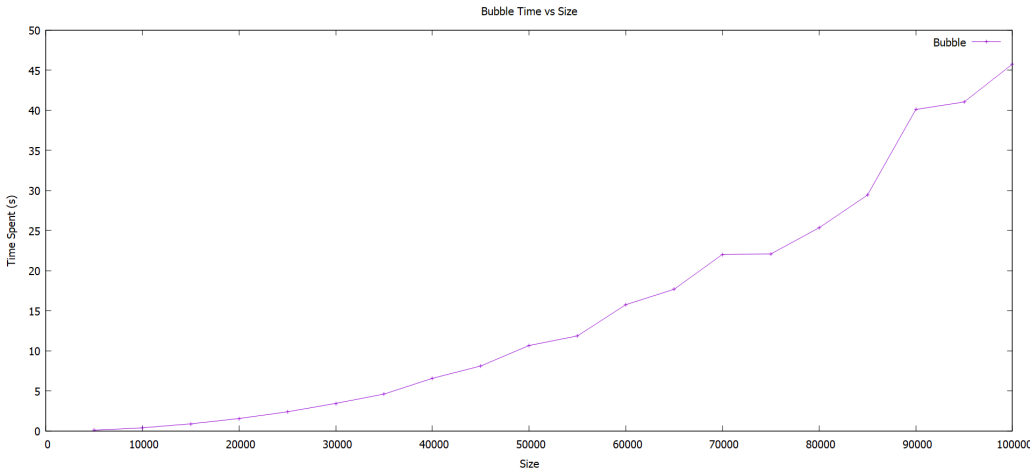

Figure 2: Évaluation des performances du tri à bulle

2.3 Tri par sélection (Selection Sort)

Le tri par sélection recherche à chaque itération le plus petit élément restant pour l'échanger avec la position courante. Il est déterministe avec une complexité fixe en $O(n^2)$, mais il n'est pas stable.

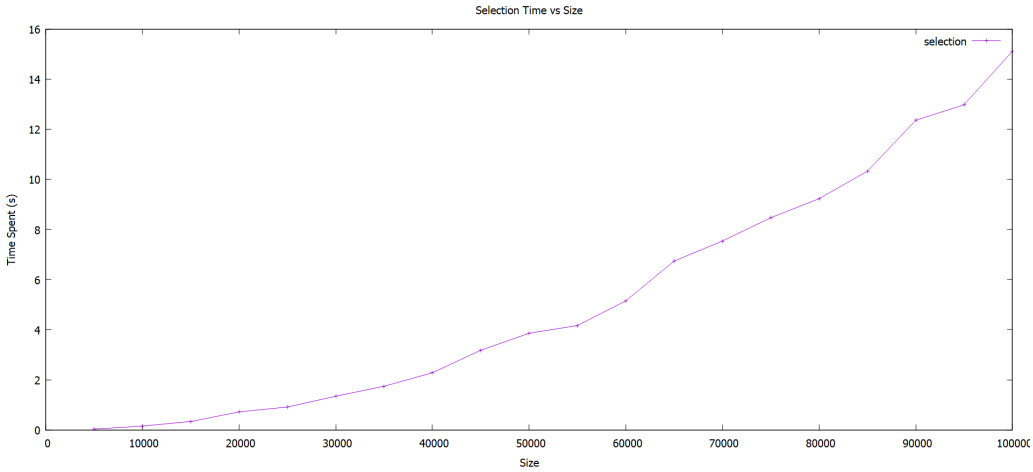

Figure 3: Évaluation des performances du tri par sélection

L'analyse comparative des trois algorithmes met en évidence des différences significatives en termes de performance. Le tri par insertion se révèle nettement plus performant dans la pratique, en particulier lorsque les tableaux sont déjà partiellement triés, situation dans laquelle il bénéficie d'une complexité proche de O(n). À l'inverse, le tri à bulle se montre extrêmement lent, au point qu'il n'a pas été possible de évaluer ces algorithmes sur de très grands tableaux en raison du temps d'exécution excessif. Quant au tri par sélection, il présente un comportement plus régulier mais reste globalement moins efficace que le tri par insertion, notamment parce qu'il ne profite pas des éventuels ordres partiels présents dans les données.

2.4 Tri rapide (Quick Sort)

Le tri rapide utilise le paradigme diviser-pour-régner en choisissant un pivot, puis en partitionnant et triant récursivement les sous-tableaux. Sa complexité moyenne est en $O(n \log n)$, mais le pire cas atteint $O(n^2)$ selon le choix du pivot.

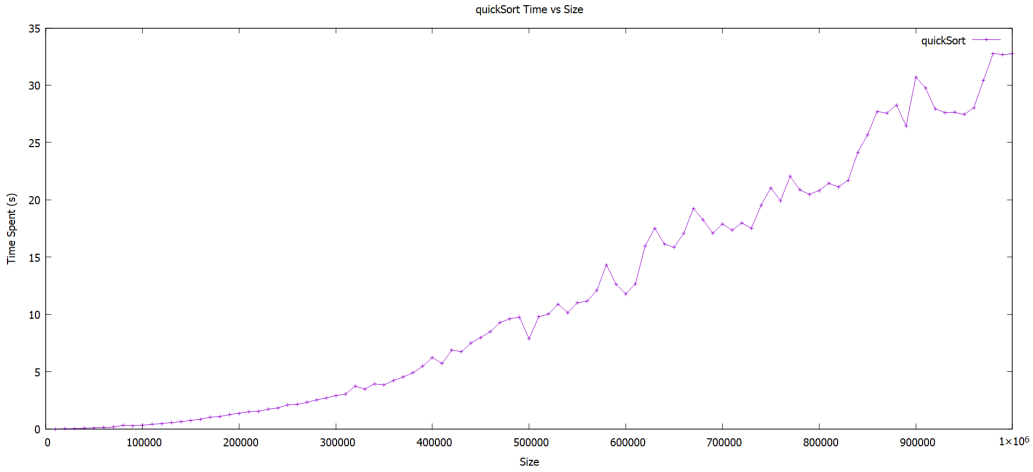

Figure 4: Évaluation des performances du tri rapide v1

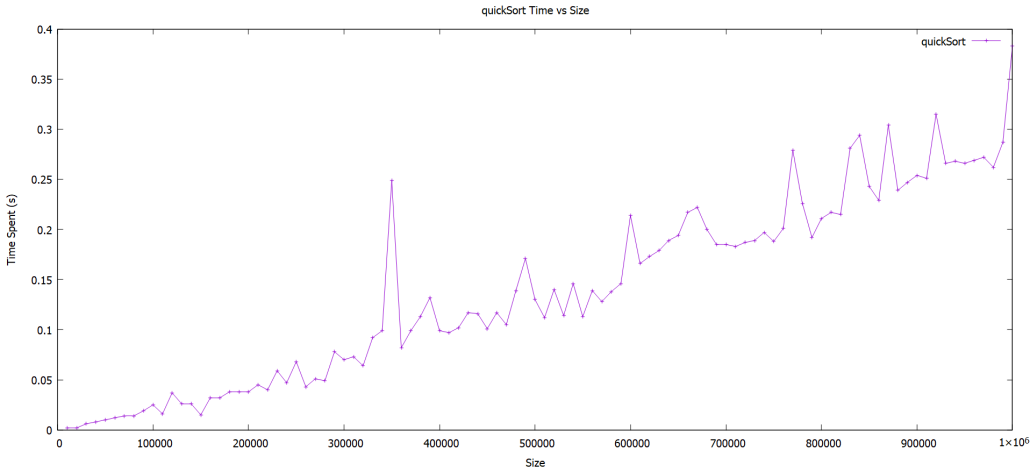

Figure 5: Évaluation des performances du tri rapide v2

Le premier test a été effectué sur des tableaux contenant des valeurs semi-triées et dupliquées, et le deuxième sur des tableaux avec des valeurs distinctes. On peut donc en conclure que l'algorithme Quicksort est affecté par la présence de doublons et par les tableaux semi-triés.

2.5 Tri fusion (Merge Sort)

Le tri fusion divise récursivement le tableau en deux parties, les trie indépendamment, puis les fusionne. Il est stable et garantit toujours une complexité en $O(n \log n)$, au prix d'une utilisation mémoire supplémentaire.

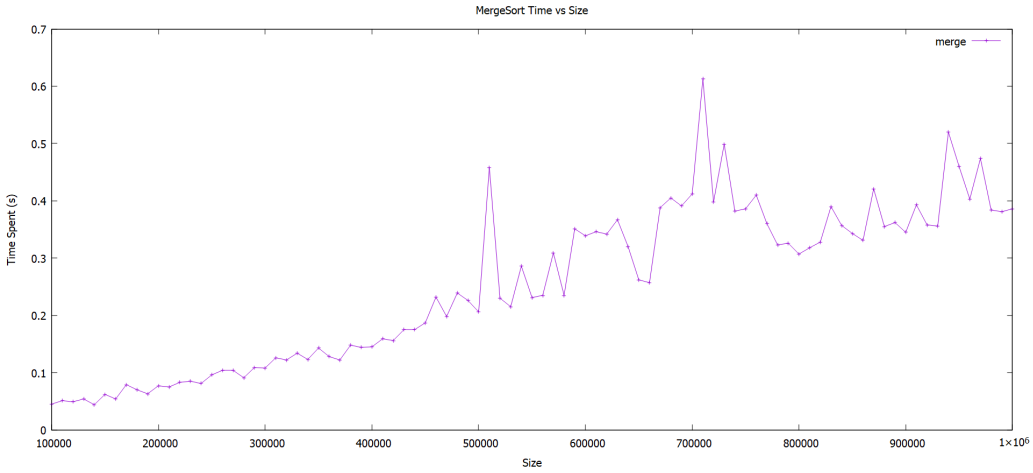

Figure 6: Évaluation des performances du tri fusion v1

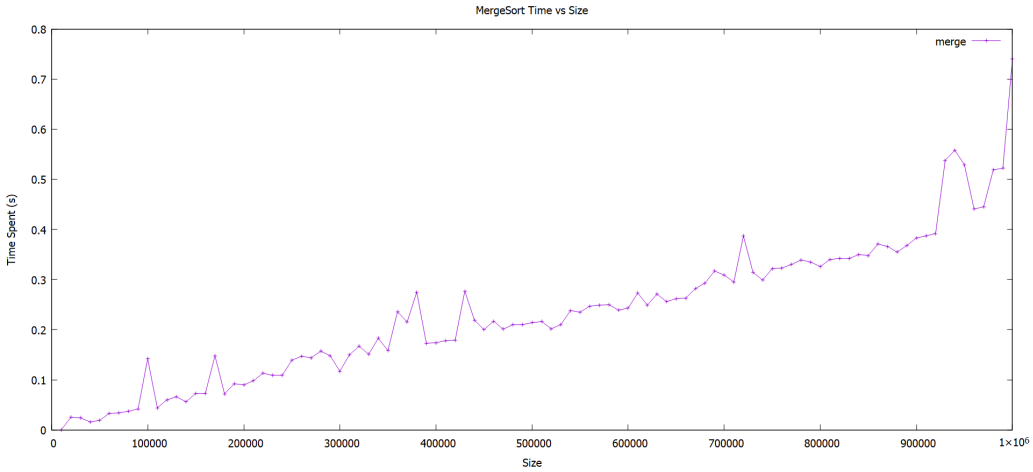

Figure 7: Évaluation des performances du tri fusion v2

Par contre, contrairement au tri rapide, le tri par fusion est stable dans les deux cas et fournit de bons résultats.

2.6 Tri par tas (Heap Sort)

Le tri par tas convertit le tableau en un tas binaire (heap) puis extrait successivement l'élément maximal. Sa complexité est toujours en $O(n \log n)$ et il ne nécessite pas de mémoire additionnelle significative, mais il n'est pas stable.

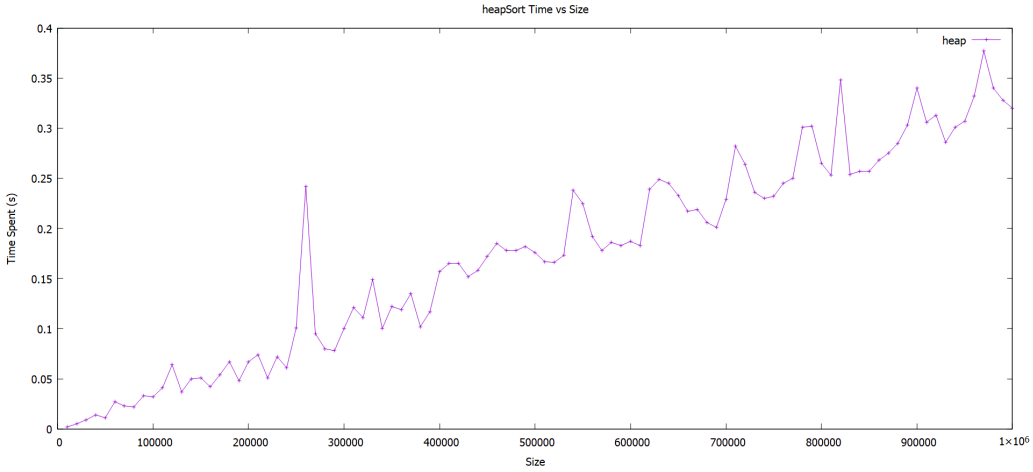

Figure 8: Évaluation des performances du tri par tas v1

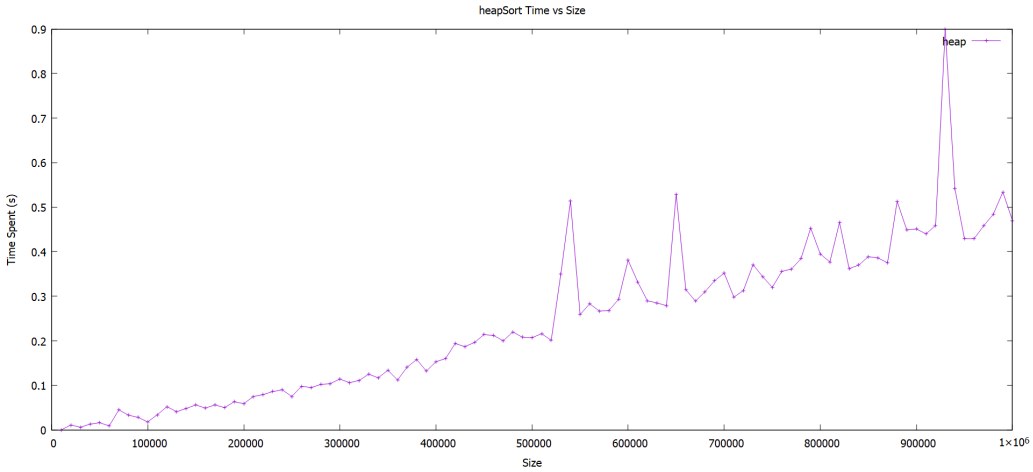

Figure 9: Évaluation des performances du tri par tas v2

2.7 Tri de Shell (Shell Sort)

Le tri de Shell améliore le tri par insertion en introduisant un écart (gap) pour comparer des éléments distants. La complexité dépend de la séquence des gaps, mais reste significativement meilleure que $O(n^2)$ dans la pratique.

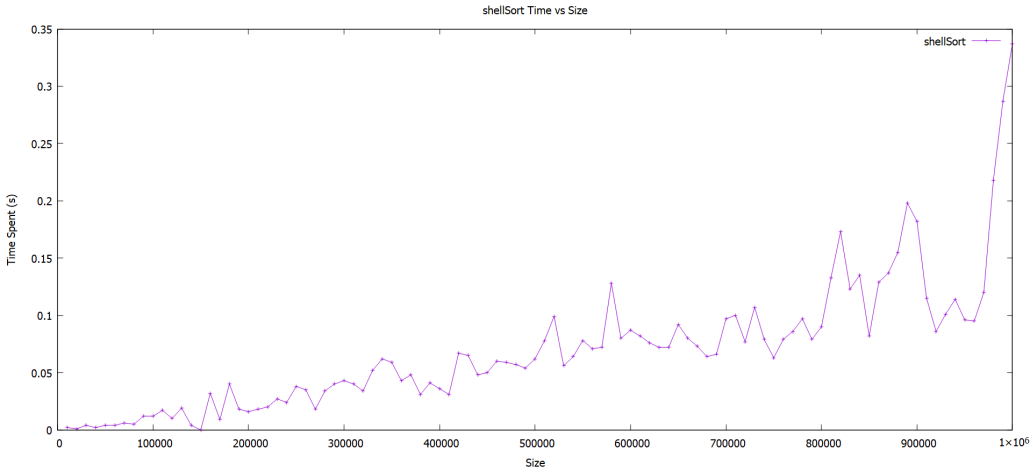

Figure 10: Évaluation des performances du tri de Shell

Le tri de Shell offre de bons résultats en pratique et sur des tableaux de taille moyenne, grâce à sa réduction efficace des écarts. Cependant, il peut rencontrer des difficultés avec certaines distributions de données très désordonnées ou mal adaptées à la séquence de gaps choisie, ce qui peut ralentir temporairement le tri par rapport à son comportement optimal.

3 Analyse comparative et synthèse des performances

Cette section présente une comparaison générale des algorithmes de tri étudiés. Elle inclut un tableau récapitulatif des complexités théoriques et une figure illustrant les courbes d'efficacité observées lors des tests pratiques.

3.1 Tableau comparatif des complexités

Algorithme	Complexité meilleure	Complexité moyenne	Complexité pire	Stabilité
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Oui
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Oui
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Non
Shell Sort	$O(n \log n)$	Dépend de la séquence	$O(n^{3/2})$	Non
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Oui
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Non
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Non

Table 1: Comparaison des complexités des algorithmes de tri

3.2 Courbes d'efficacité des algorithmes

La figure ci-dessous illustre les performances mesurées pour chaque algorithme sur les différentes tailles de tableaux testées. Elle permet de visualiser directement les différences de temps d'exécution et de mettre en évidence les algorithmes les plus efficaces dans les cas pratiques.

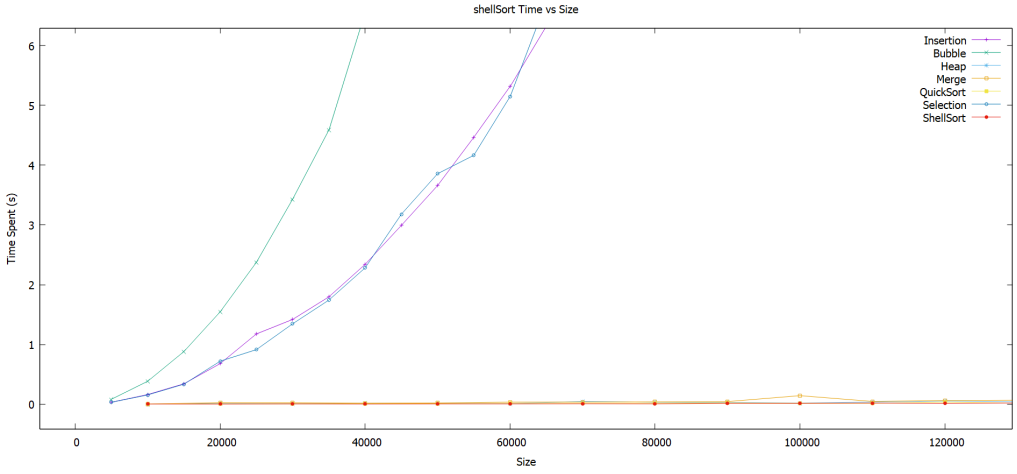

Figure 11: Courbes d'efficacité comparatives des algorithmes de tri

Conclusion

Les tests et analyses des différents algorithmes de tri montrent des comportements distincts selon les cas pratiques. Le tri par insertion se révèle efficace pour les tableaux de petite taille ou partiellement triés, tandis que le tri à bulle reste très lent et inadapté aux grandes données. Le tri par sélection est stable mais globalement moins performant que l'insertion. Les algorithmes O(nlogn), tels que Quick, Merge et Heap, offrent des performances constantes et élevées, même pour de grandes tailles de tableaux. Shell Sort constitue une solution intermédiaire : généralement rapide, mais pouvant ralentir selon la distribution des données. Ces observations permettent de choisir l'algorithme le plus adapté selon la taille et la nature des données à trier.