École Centrale de Lille

Algorithmique avancée et programmation

Implémentation d'algorithmes de tri

Compte-rendu

Nom	**Prénom**
AQUILO	Axel
GOSMANN	Gabriel
DOS SANTOS SILVA	Vitor
STEFFENS WONTROBA	Vinícius

Villeneuve-d'Ascq
2022

École Centrale de Lille

Implémentation d'algorithmes de tri

Compte-rendu

Compte-rendu du projet présenté aux évaluateurs du sujet Algorithmique avancée et programmation de l'École Centrale de Lille.

Villeneuve-d'Ascq
2022

Table des matières

Table des figures

Divide et impera.

Julius Cesar

1. Introduction

1.1 Compréhension du problème

Le problème présenté demande au groupe d'adapter et d'implémenter des algorithmes qui effectuent le tri des listes et des tableaux. Plus que cela, il cherche à faire une comparaison entre les méthodes abordées, en analysant leurs niveaux de complexité, de performance, d'avantages et d'inconvénients. Les algorithmes proposés sont :

1. Tri Fusion (Merge Sort) ;
2. Tri Rapide (Quick Sort) ;
3. Tri Fusion de listes.

1.2 L'organisation du programme et du groupe

1.2.1 Le programme

Le programme a été divisé de la manière la plus naturelle que l'on puisse imaginer : un fichier *.c* et sa bibliothèque respective *.h* pour chaque algorithme implémenté.

1.2.2 Le groupe

L'équipe a été divisée pour que la mise en œuvre se fasse de manière optimale.

— AQUILO Axel : Détermination des niveaux de complexité de chaque algorithme ;

— GOSMANN Gabriel : Implémentation et adaptation de l'algorithme de Tri Fusion de Listes ;

— DOS SANTOS SILVA Vitor : Implémentation et adaptation de l'algorithme de Tri Rapide ;

— STEFFENS WONTROBA Vinícius : Implémentation et adaptation de l'algorithme de Tri Fusion.

2. Développement

2.1 Tri fusion

2.1.1 Implémentation pour un tableau

L'implémentation de l'algorithme *Tri Fusion* est relativement plus simple puisqu'il n'y a pas de différence si le vecteur a été rempli aléatoirement, inversement ou dans l'ordre. Cela dit, nous avons choisi de remplir le vecteur au hasard.

Le fichier qui effectue ce remplissage s'intitule *teste.c*. Cela appelle la fonction *tri_fusion*, présente dans le fichier *tri_fusion.c*, qui effectue en fait le tri de la table insérée. Le nombre d'opérations et de comparaisons effectuées lors de l'exécution est compté afin de déterminer l'efficacité et la complexité de l'algorithme. Ce processus est répété 100 fois jusqu'à ce qu'un vecteur de taille prédéfinie *MAX_ELT* soit atteint. Le résultat est envoyé dans un fichier .txt, intitulé *tea.txt*, qui est ensuite tracé à l'aide d'un programme externe. Les résultats sont présentés ci-dessous :

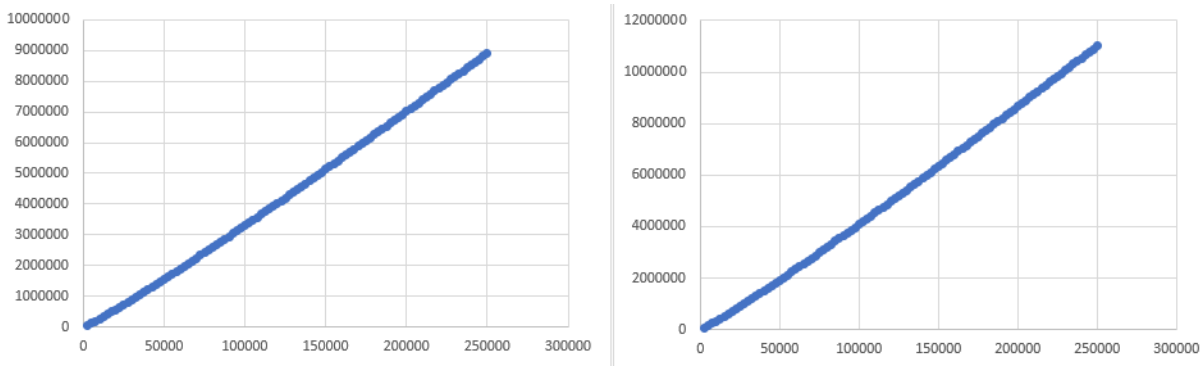

FIGURE 2.1 – Opérations x taille et comparaisons x taille

En analysant les images présentées dans la figure 2.1, il est clair qu'il s'agit d'un algorithme de complexité *n log(n)*.

2.1.2 Compatibilité avec fonction standard

Une fonction compatible avec la déclaration donné sur l'énoncé a été développé et est présentée sur le fichier "fusion_sort.c"

2.1.3 Complexité théorique du tri fusion

Nous abordons le problème de calcul de complexité de l'algorithme en considérant la complexité du pire des cas. Prenons pour cela une liste de n éléments à ordonner. Le programme va tout d'abord diviser cette liste en 2 listes de taille n/2, puis redivise chaque liste etc. jusqu'à obtenir des listes de unitaires. Cette première opération comprend au maximum 2^{n+1}, soit donc une complexité en $log(n)$. Ensuite, le programme va fusionner les listes entre elles, en les ordonnant de façon a ce que la liste produite soit trié. Pour cela le programme dépile les 2 listes et ajoute les éléments dans l'ordre de tri souhaité. De cette façon toute les listes utilisés sont correctement ordonnées. La fusion de deux listes de taille a et b est en $O(a+b)$, et cette opération est à faire un nombre de fois qui est en $O(log(n))$ Ainsi, la complexité totale de l'algorithme de tri fusion est en $O(nlog(n) + log(n)$, soit $O(nlog(n))$.

2.2 Tri rapide

2.2.1 Implémentation pour un tableau

La première tache sur le tri rapide était de l'implémenter sur un tableau. Ce qui a été facilement fait en adaptant le code en *teste.c* pour le tri rapide.

2.2.2 Analyse de la dégénérescence

Les données produits par le tri rapide on été analysé. On voit sur la Figure 2.2 que le numéro de comparaisons grandi beaucoup en fonction de N.

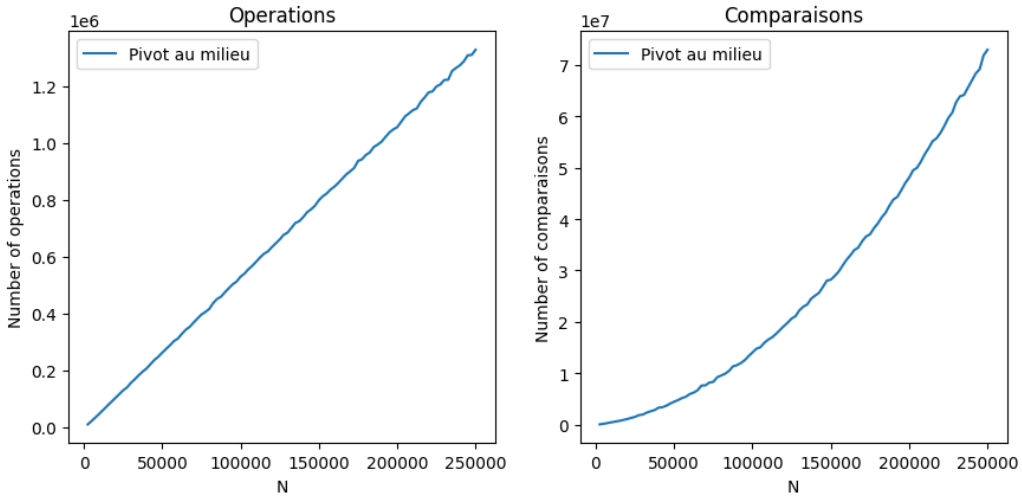

FIGURE 2.2 – Pivot au milieu

Toutefois, c'est possible de le rendre plus efficace, en utilisant un pivot aléatoire chaque fois. Le résultat est présenté sur la Figure 2.3.

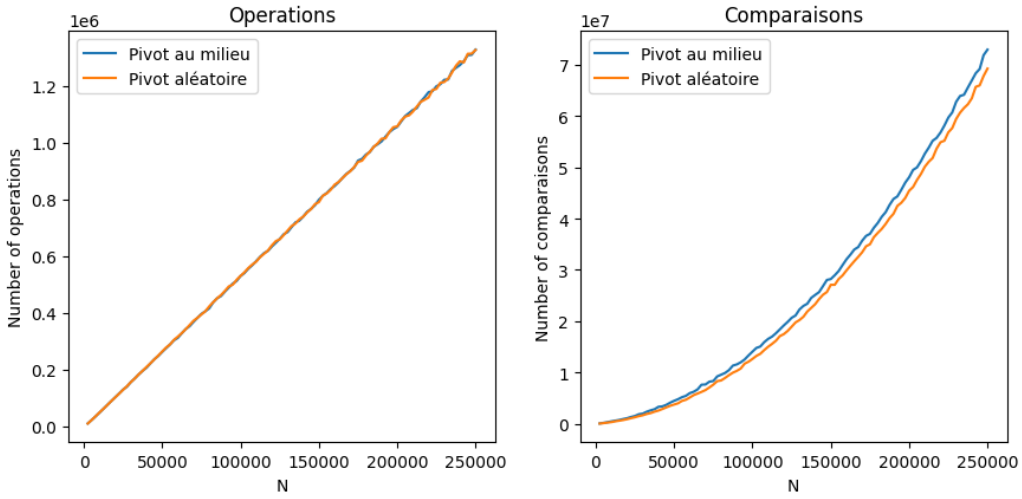

FIGURE 2.3 – Pivot aléatoire

2.2.3 Compatibilité avec fonction standard

Une fonction compatible avec la déclaration donné sur l'énoncer a été développé et est présentée sur le fichier "quick_sort.c"

2.2.4 Complexité théorique du tri rapide

Nous considérons encore une fois la complexité du pire des cas, pour une liste de n éléments à ordonner. Pour cet algorithme, on divise la liste mère en 2 liste, en mettant d'un côté des valeurs inférieures à un pivot, et dans l'autre les valeurs supérieures, et on recommence l'opération jusqu'à l'obtention de listes unitaires, mais jamais obtenues simultanément car les listes restent attachées à leur liste mère. Cette fois ci, on passe en revu chaque valeur lors de la division des liste. Cette première étape est donc de complexité $O(n)$, et réalisée de l'ordre de $O(log(n)$ fois. On procède ensuite à une concaténation des listes, en comparant seulement deux éléments des listes car d'après le principe utilisé, l'une des deux liste est strictement supérieure à l'autre lors de la concaténation. Cette deuxième étape est donc rapide, et de complexité $log(n)$. On se retrouve dans le même cas que pour le tri fusion, avec une complexité totale en $O(nlog(n))$.

2.3 Tri fusion des listes

Nous avons implémenté un programme capable de trier des linked-lists basé sur le algorithme de tri fusion. Le tri fusion est un algorithme de tri efficace (complexité $O(n \log n)$) créé en 1945 par John Von Neumann. Le tri par fusion est basé sur une stratégie diviser pour régner, ce qui le rend particulièrement adapté aux implémentations récursives. L'image 2.4 montre un diagramme visuel du fonctionnement de l'algorithme.

Dans cet algorithme, il y a deux opérations de base (différencié en couleur dans l'image) et qui sont responsables des itérations récursives :

— la division des linked lists (rouge) ;

— la jonction de listes liées (vert).

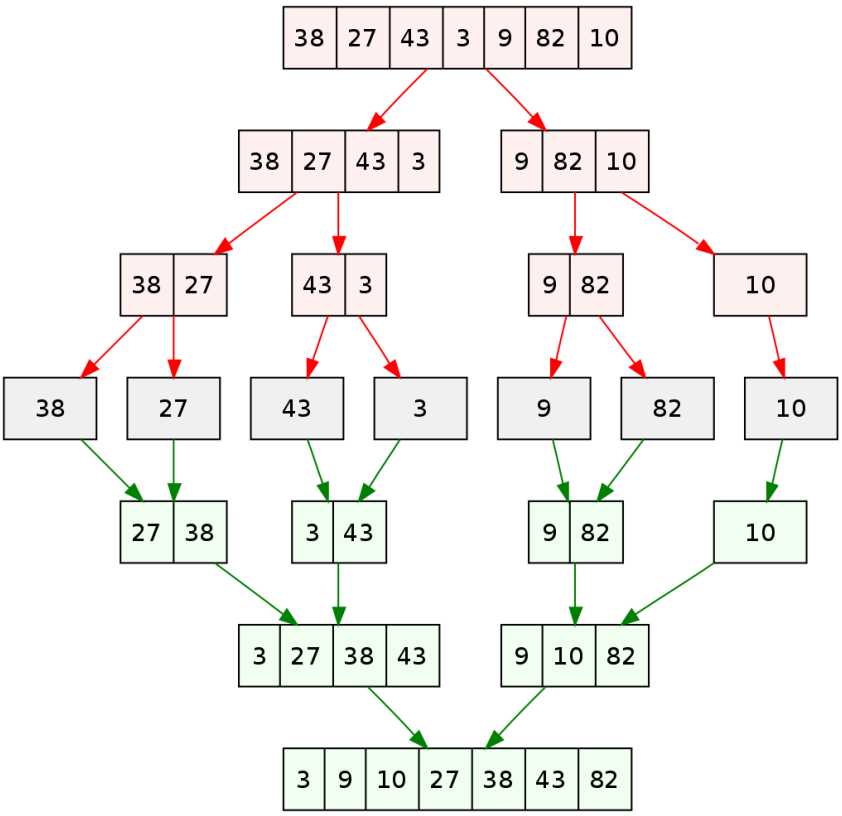

FIGURE 2.4 – Tri Fusion Algorithmique Diagram

L'idée est de décomposer une seule linked list en plusieurs listes d'éléments uniques et de s'assurer que ce processus de jonction est effectué de manière ordonnée. Il est facile de penser qu'une liste d'un seul élément est déjà triée. Sachant cela, lors du tri de listes d'un seul élément, une seule comparaison est nécessaire. En progressant, lors du tri des listes de 2 éléments, on peut supposer que les listes sont déjà triées, ce qui permet de ne faire la comparaison qu'entre les premiers (plus petits) termes de chaque liste pour associer les éléments de la liste suivante. La priorité des algorithmes de tri est de minimiser le nombre de comparaisons et d'opérations et cette idée de restreindre la comparaison uniquement aux premiers éléments de chaque liste permet justement cela.

Pour y parvenir, nous avons développé deux fonctions pour résoudre récursivement ces problèmes de séparation et de jonction :

— Fonction `join` ;

— Fonction `fusion`.

2.3.1 Fonction `join`

Le prototype de cette fonction peut être vu ci-dessous.

```
t_node * join(t_node * left, t_node * right);
```

Cette fonction est en fait l'endroit où se déroule la majeure partie du processus d'inscription. Il reçoit deux linked lists précédemment triées et renvoie une seule liste liée issue de la combinaison des deux linked lists de manière ordonnée. Le code source de cette fonction peut être vu ci-dessous.

```
t_node * join(t_node * left, t_node * right){

    t_node * result = NULL;

    if( left == NULL )return right;

    else if( right == NULL )return left;

    if(left->data <= right-> data){
        result = left;
        result->next = join(left->next, right);
    }
    else{
        result = right;
        result->next = join(left, right->next);
    }

    return result;
}
```

L'utilisation de récursions rend la syntaxe de cette fonction très compacte et facile à suivre. Une fois que la fonction est appelée, un nouveau pointeur vers linked-lists pointant vers NULL est créé, ce pointeur est censé pointer vers la liste chaînée résultante. L'idée est que le premier élément de chaque liste chaînée sera comparé et le plus petit sera enregistré dans le pointeur `result`. L'élément qui vient d'être ajouté au pointeur est virtuellement supprimé de sa liste d'origine en appelant de manière récursive la fonction `join` et en lui passant non pas la même liste mais un pointeur vers son élément suivant. Cela crée effectivement une nouvelle liste sans le premier élément (celui stocké maintenant sur `result`) du point de vue de la fonction appelée. Ce processus est répété itérativement jusqu'à ce qu'une des listes soit vide. Lorsque cela se produit, naturellement tous les

éléments doivent provenir de l'autre list, qui était auparavant trié. Lorsque les deux listes sont vides, le processus est arrêté et la liste triée `result` est renvoyée.

2.3.2 Fonction `fusion`

Le prototype de cette fonction peut être vu ci-dessous.

```
t_node * fusion(t_node * list, int length);
```

C'est la fonction principale du programme. Passant un pointeur vers le premier élément d'une linked list et le nombre d'elements de la linked list, la fonction renvoie un pointeur vers le premier élément de la même linked list triée. Le code source de cette fonction peut être vu ci-dessous.

```
t_node * fusion(t_node * list, int length){

    t_node * list_right = list;
    int i;

        // if it is larger than 1 it can be divided
    if(length > 1){
        // the idea is to divide the initial list into 2 smaller lists
        int half = length/2;

        for(i = 0; i < half-1; i++)
            list_right = list_right->next;

        t_node * aux = list_right->next;
        list_right->next = NULL;
        list_right = aux;

        list = fusion(list, half);
        list_right = fusion(list_right, length-half);

        // make fusion here
        list = join(list, list_right);
    }
    return list;
}
```

Comme c'était le cas avec la fonction `join`, l'aspect récursif de celle-ci rend l'implémentation trés courte et propre. Fondamentalement, la fonction a utilisé le paramètre `length` pour subdiviser la liste d'arguments en deux, ce qui est l'un des operations de base nécessaires à l'algorithme. Ce processus est répété tant que la liste est composée de plus

d'un élément, d'où la condition. Une fois les listes divisées, il y a maintenant le pointeur à `list` sur le côté gauche de la liste d'origine et le pointeur `list_right` pointant sur le côté droit de la liste d'origine. Au fur et à mesure que cela est fait, la même fusion s'appelle, mais en passant maintenant les listes avec la moitié de la taille d'origine. La récursivité fait en sorte que ce processus continue de manière itérative jusqu'à ce que seules des listes d'éléments uniques soient atteintes. Une fois que cela se produit, l'algorithme procède à la jonction récursive de toutes les listes précédemment divisées. Les processus ne se terminent que lorsqu'une seule liste est récupérée et donc le pointeur vers son premier élément est ensuite renvoyé par la fonction.

2.3.3 Représentation Graphique

Comme demandé, nous avons développé une fonction appelée generatePNG qui prend une liste comme argument et génère un fichier .dot qui la décrit graphiquement. Le prototype de cette fonction peut être vu ci-dessous :

```
void generatePNG(const t_node * const_list, char * filename);
```

La fonction est basée dans un fichier `base.dot` et n'ajoute que les lignes qui décrivent les nœuds de la liste donnée. Au fur et à mesure qu'une liste est triée et que toutes les listes pertinentes ont été enregistrées dans leurs formats `.dot` respectifs dans un dossier `dots` spécifique, un script que nous avons écrit (`convert_png.sh`) exécute ce dossier et convertit chaque fichier `.dot` dans le fichier `.png` correspondant et l'enregistre dans le dossier `images`. A titre d'exemple, l'image ci-dessous montre la représentation graphique d'une structure $3 \rightarrow 9 \rightarrow 10 \rightarrow 82 \rightarrow NULL$:

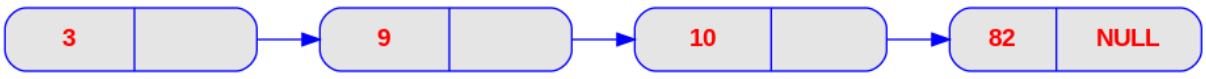

FIGURE 2.5 – Exemple représentation graphique

Ayant cet outil en main, nous avons décidé de nous faire une représentation graphique de l'algorithme de tri par fusion avec des listes liées similaires à celle vue sur la figure 2.4. Notre représentation est visible sur la figure 2.6.

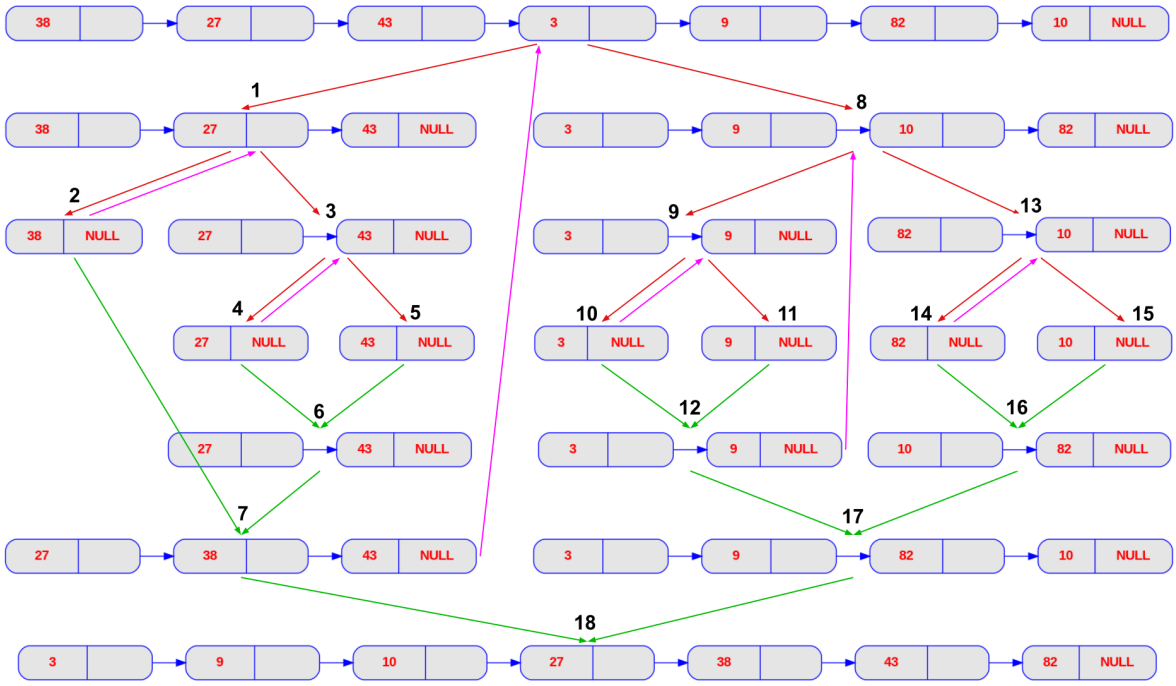

FIGURE 2.6 – Fusion sort avec notre représentation graphique

Les flèches rouge et verte ont la même signification que le schéma précédent comme étant l'appel de la fonction `fusion` et `join` respectivement. La flèche rose est un ajout intéressant car elle indique quand une fonction est terminée et revient à l'endroit où elle a été appelée. Les nombres sont également un ajout et ils montrent chaque étape intermédiaire du processus dans l'ordre réellement créé. Compte tenu de cela, nous pensons que cette représentation représente beaucoup plus clairement ce qui se passe réellement lorsque l'algorithme est en cours d'exécution - principalement comment les effets récursifs se produisent.

2.3.4 Préférence du tri par fusion pour les linked lists

Une caractéristique importante qui rend le tri par fusion particulièrement adapté au tri des linked lists est le fait que les algorithmes qui nécessitent un accès aléatoire sont beaucoup plus complexes dans les listes que dans les tableaux. Comme les données ne sont pas continuellement espacées en mémoire dans les listes, accéder à un élément d'une liste nécessite de parcourir la liste jusqu'à cet élément, ce qui est une opération de complexité $O(n)$, par opposition à la même chose dans un tableau qui a complexité $O(1)$. Cette caractéristique rend les algorithmes qui nécessitent cet accès aléatoire mal adaptés aux listes chaînées. Sur le tri par fusion, d'autre part, seuls les premiers éléments des listes

sont comparés, et c'est une opération qui a la complexité $O(1)$ pour les listes chaînées également.

3. Conclusion et analyse des résultats

Finalement, on peut comparer le résultat des algoritmes avec la fonction déjà implémenté en C, la fonction *qsort*. La comparaison a été faite en utilisant le temps, comment c'est la seul métrique qu'on peut avoir sur la fonction. Le résultat est visible sur la figure 3.1.

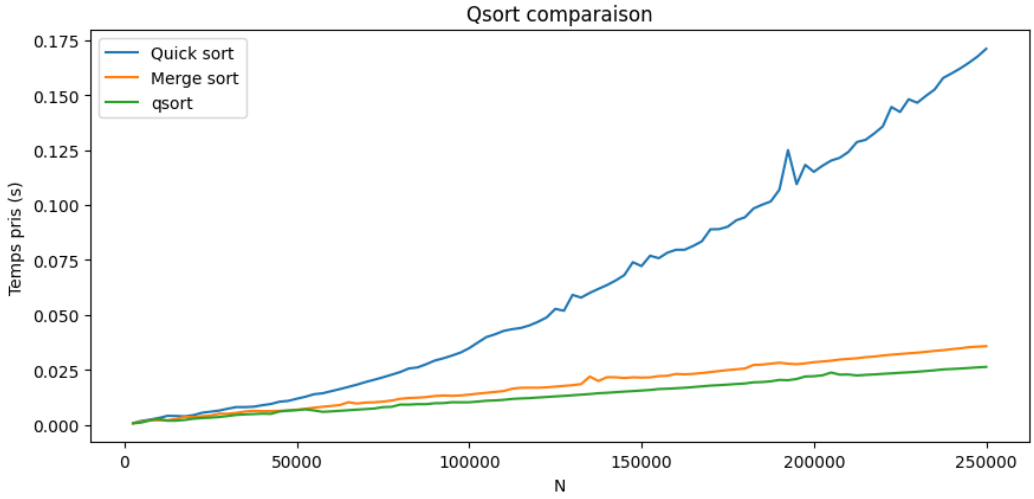

FIGURE 3.1 – Comparaison avec qsort

On peut conclure que la fonction standard de C est encore plus efficace, mais c'est quand même plus proche du tri fusion que du tri rapide.

4. Références bibliographiques

[1] D.M. Ritchie et B.W. Kernighan. *The C programming language*. Bell Laboratories, 1988.