République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumédiène

Faculté d'Informatique
Département Informatique

Master Systèmes Informatiques intelligents

Module : Conception et Complexité des Algorithmes

Rapport de projet de TP Tri

Réalisé par :
BENBACHIR Mohamed Amir, 191932021049
BOUCHOUL Bouchra, 191931081317
KHEMISSI Maroua, 191935007943
MEDJKOUNE Roumaissa, 191931081005

Année universitaire : 2022 / 2023

Table des matières

Chapitre 1

Introduction

Les algorithmes de tri ont une grande importance pratique. Ils sont fondamentaux dans certains domaines, comme l'informatique de gestion où l'on tri de manière quasi-systématique des données avant de les utiliser.
L'étude du tri est également intéressante en elle-même car il s'agit sans doute du domaine de l'algorithmique qui a été le plus étudié et qui a conduit à des résultats remarquables sur la construction d'algorithmes et l'étude de leur complexité.

Chapitre 2

Environment Experimentale

Pour notre environment experimentale on a utilise Repl.it qui est un IDE en ligne qui permet d'écrire et d'exécuter des programmes depuis un navigateur web. Il a l'avantage de ne pas nécessiter d'installation sur le poste de travail.

Repl.it permet de créer des espaces de stockage et d'exécution de programme, appelés repls qui sont caracterisees par : .

0.5 vCPUs

1 GiB de storage

1 GiB RAM

Chapitre 3

Tri par insertion

3.1 Fonctionnement de l'algorithme

Le tri par insertion est un algorithme de tri simple qui fonctionne de manière similaire à la façon dont nous trions les cartes à jouer entre mains.

Le tableau est virtuellement divisé en une partie triée et une partie non triée. Les valeurs de la partie non triée sont sélectionnées et placées à la bonne position dans la partie triée.

Les étapes sur la façon dont cet algorithme fonctionne se résume comme suit :

— Si c'est le premier élément, il est déjà trié.

— Choisissez l'élément suivant.

— Comparez avec tous les éléments de la sous-liste triée.

— Décalez tous les éléments de la sous-liste triée qui sont supérieurs à la valeur à trier vers la droite, puis insérez la valeur.

— Répétez jusqu'à ce que la liste soit triée.

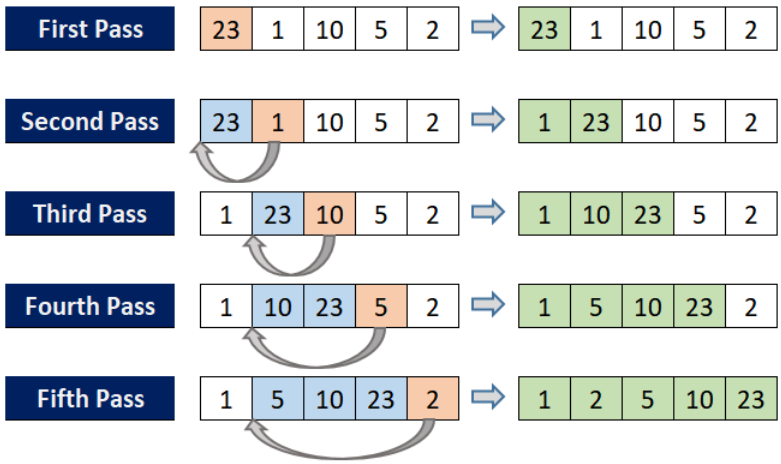


FIGURE 3.1 – Exemple graphique d'un tri par insertion

```
Fonction Insert(Entrée : tab : tableau d'entier ;)
  Variables :
  i,j : entier;
  tmp,longeur : entier;
  début
    longeur ← taille(T);
    pour i ← 1 à N-1 faire
      tmp ← tab[i];
      j ← i;
      tant que j > 0 et tab[j − 1] > tmp faire
        tab[j] ← tab[j − 1];
        j ← j − 1;
      fin tq
      tab[j] ← tmp;
    fin pour
  fin
```

3.2 Calcul de complexité

3.2.1 Complexité temporelle

La complexité d'un algorithme de tri par insertion dépend de la taille n du tableau et de sa nature : si le tableau est déjà trié (ou partiellement trié), la complexité est en effet beaucoup moins important que si le tableau est trié dans l'ordre inverse.

Meilleur Cas :

Ce type de complexité se produit souvent lorsque les éléments du tableau initial sont triés. Il faut donc comparer un à un (n-1) éléments.

* La boucle (pour) s'exécute un nombre de fois égal à N-1.

* la boucle (tant que) ne s'exécute pas.

Il y a donc N-1 comparaisons et au plus N affectations. C(n)=N+(N-1)=2N+1. La complexité du meilleur cas est d'ordre N. Il en résulte une complexité linéaire O(n).

Pire Cas :

Ce type de complexité se produit lorsque les éléments du tableau sont initialement triés dans l'ordre inverse.

A chaque étape, tous les éléments du sous-tableau trié doivent donc être décalés vers la droite pour que l'élément à trier qui est plus petit que tous les éléments déjà triés à chaque étape puisse être placé au début.

Donc on effectue l'échange a chaque comparaison.

Le nombre d'échange est alors :

C(n)=2+3+...+N-1= N(N-1)/2 .

La complexité dans le pire des cas est d'ordre N^2.$Ilenrsulteunecomplexitquadratique O(n^2)$.

Moyen Cas :

Ce type de complexité se produit généralement lorsque les éléments d'un tableau sont mélangés de sorte que seulement la moitié des éléments sont décalé, ce qui signifie que chaque élément est plus petit que la moitié des éléments à sa gauche. En considérant un décalage de (p-1)/2 éléments, la complexité moyenne du calcul est donc : C(n)=1/2+1+...+(N-1)/2= N(N-1)/4

La complexité sera donc la moitié de la complexité du pire cas, mais elle est toujours d'ordre $N^2, doncc'estunecomplexitquadratique O(n^2)$

3.2.2 Complexité spatiale

Le tri par insertion englobe une complexité spatiale de O(1) en raison de l'utilisation d'une variable supplémentaire tmp.

3.3 Experimentation

Dans cette partie nous allons voir les résultats des exécutions de cet algorithme sur différents taille de tableau et sur données qui se représentent en 3 configuration (triée en bon ordre , triée en ordre inverse , aléatoire)

Les données du tableau sont triées en bon ordre.

Taille	10000	50000	100000	500000	1000000	5000000	10000000	50000000
temps(s)	0.000014	0.000063	0.000136	0.000337	0.001291	0.003248	0.012637	0.064748

Les données du tableau sont triées en ordre inverse.

Taille	10000	50000	100000	500000	1000000	5000000	10000000	50000000
temps(s)	0.061094	1.500352	6.068683	101.321114	511.278107			

Les données du tableau sont positionnés aléatoirement.

Taille	10000	50000	100000	500000	1000000	5000000	10000000	50000000
temps(s)	0.017770	0.537932	1.994625	60.600109	223.830612			

Le graphe :

La figure suivante représente les résultats d'exécution de cet algorithme selon les différentes taille du tableau.

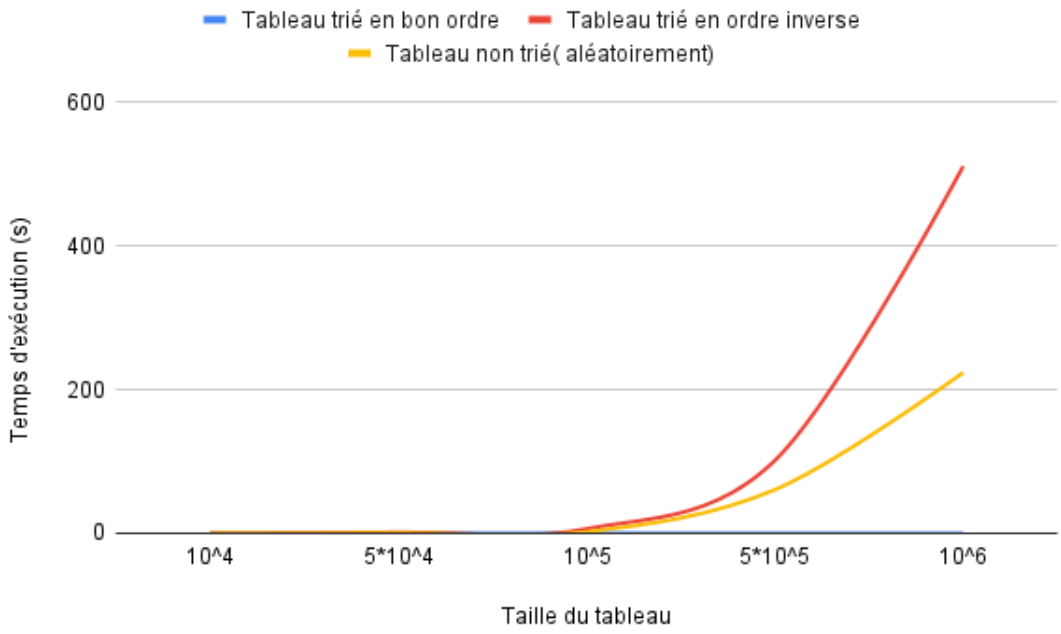


FIGURE 3.2 – Temps d'execution du tri par insertion sur les différents taille du tableau

D'après le graphique et les tableaux ci-dessus, nous remarquons que le tri par insertion prend beaucoup de temps lors du tri des éléments dans l'ordre inverse ou dans leur positionnement aléatoire, Les courbes s'évoluent d'une manière quadratique à chaque fois que la taille du tableau augmente. Cependant, si les éléments sont déjà triés, cela fonctionne très bien et ne prend pas beaucoup de temps.

On en conclut que la complexité théorique est cohérente avec les résultats expérimentaux.

3.4 Conclusion

A partir d'étude expérimentale et théorique de l'algorithme tri par insertion, On constate que malgré sa complexité en temps quadratique sur des données inversées ou triées aléatoirement, il est encore largement utilisé car il est capable de s'exécuter en temps linéaire sur des entrées déjà triées, et de manière très efficace sur de petites entrées.

Chapitre 4

Tri par Selection

4.1 Fonctionnement de l'algorithme

Le tri par sélection (ou tri par extraction) est un algorithme de tri par comparaison. Le principe du tri par selection est de :

— rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0.

— rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1.

— continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

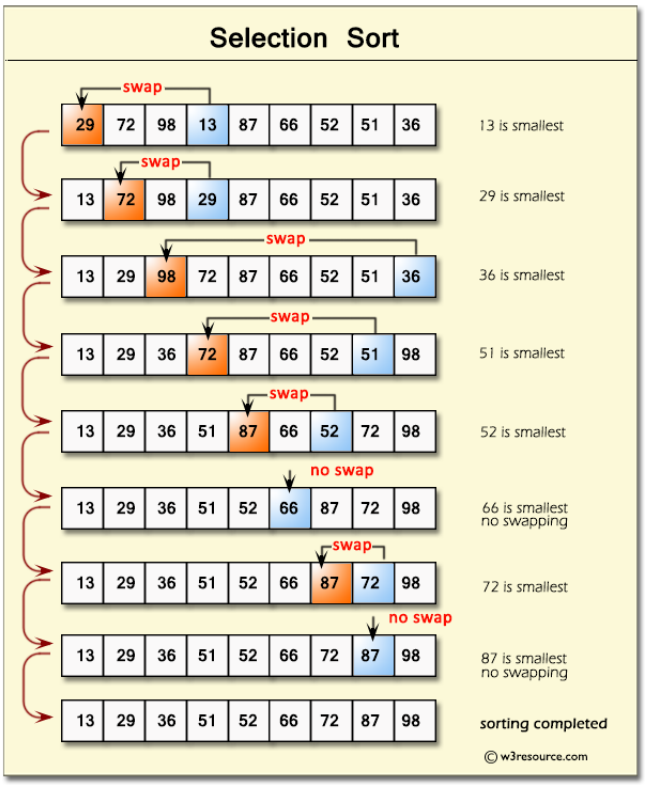


FIGURE 4.1 – Exemple graphique d'un tri par selection

Nous pouvons le représenter via le pseudo code suivant :

```
Fonction Selection(Entrée : tab : tableau d'entier ;)
  Variables :
  i,j : entier;
  tmp,min : entier;
  début
    pour i ← 1 à N-1 faire
      min ← tab[i];
      j ← i;
      pour j ← i + 1 à N-1 faire
        si tab[j] < tab[min] alors
          min ← j;
      fin pour
      tmp ← tab[min];
      tab[min] ← tab[i];
      tab[i] ← tmp;
    fin pour
  fin
```

4.2 Calcul de complexité

4.2.1 Complexité temporelle

Dans tous les cas, pour trier n éléments, le tri par sélection effectue au plus un nombre linéaire d'échanges :

Meilleur Cas : aucun si l'entrée est déjà triée.

Moyenne Cas : n-(1/2+...+1/n)= n-nln(n) c'est-à-dire si les éléments sont deux à deux distincts et que toutes leurs permutations sont équiprobables (en effet, l'espérance du nombre d'échanges à l'étape i est

Pire cas : n-1 échanges qui est atteint par exemple lorsqu'on trie la séquence 2,3,...,n,1 ;

Et donc le tri par sélection effectue n(n-1)2 comparaisons. Et sa complexité est donc O(n^2).

le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme selon la variation de la taille de l'expression :

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t(ns)	45	1225	4950	124750	499500	12497500	49995000	499995*10^4	4999995*10^5	49999995*10^6

La figure suivante représente l'évolution du temps d'exécution selon la longueur du tableau :

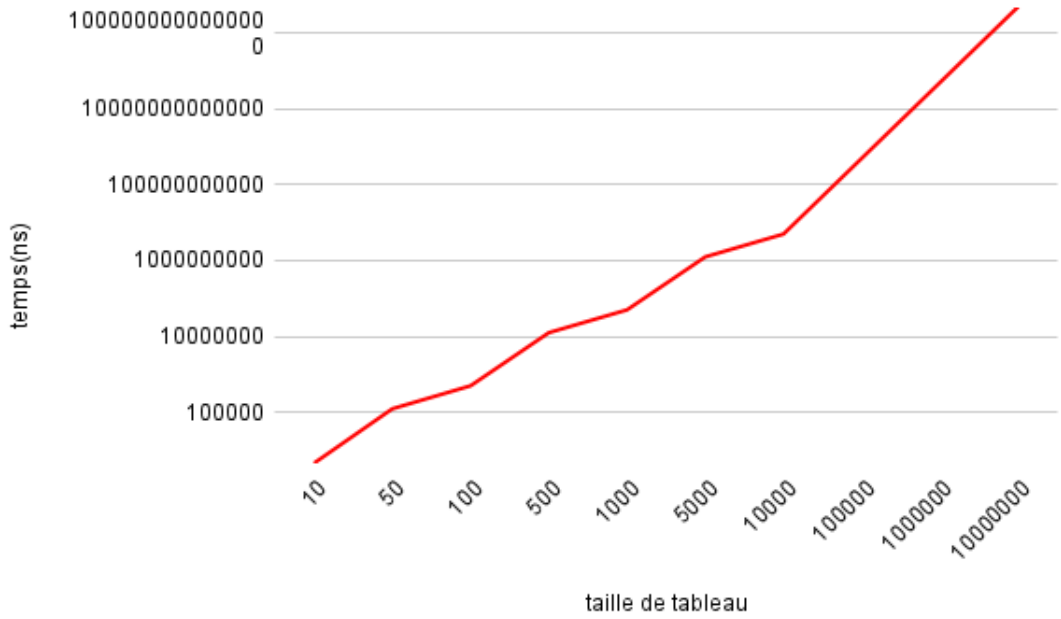


FIGURE 4.2 – Temps d'exécution théorique du Tri Selection selon la longueur du tableau

4.2.2 Complexité spatiale

Dans tous les cas O(1)

4.3 Experimentation

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme selon la variation de la taille et la configuration de l'entree .

Les données du tableau sont triées en ordre inverse.

N	10000	50000	100000	500000	1000000	5000000	10000000	50000000
Temp(s)	0.038229	1.092237	3.880469	89.18019	//	//	//	//

Les données du tableau sont triées en bon ordre.

N	10000	50000	100000	500000	1000000	5000000	10000000	50000000
Temp(s)	0.0324	0.6452	2.94	78.785	//	//	//	//

Les données du tableau sont aleatoires.

N	10000	50000	100000	500000	1000000	5000000	10000000	50000000
Temp(s)	0.026639	0.639662	3.268848	75.4	//	//	//	//

La figure suivante représente l'évolution du temps d'exécution en (s) selon la taille et la configuration du tableau :

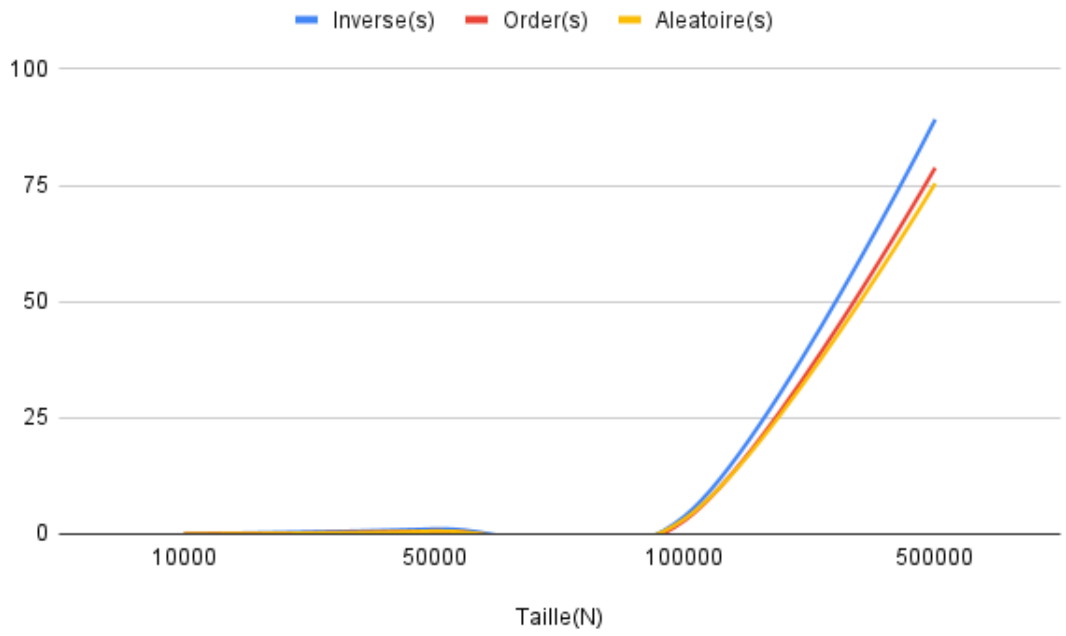


FIGURE 4.3 – Temps d'exécution du programme selon la taille du tableau

Depuis le graphe, on observe que le temps d'exécution évolue de manière quadratique avec l'augmentation de la taille du tableau quelque soit la configuration utilise, ce qui correspond bien à la complexité théorique calculée auparavant.

4.4 Conclusion

L'algorithme de tri par selection se caractérise par son fonctionnement inconditionnel sur n'importe quel tableau. Par ailleurs, elle est coûteuse en temps.

Chapitre 5

Tri rapide

5.1 Description

Le tri rapide, aussi appelé "tri de Hoare" du nom de son inventeur "Tony Hoare" ou "Quick Sort" en anglais, est considéré comme l'algorithme le plus performant des tris en table qui est certainement celui qui est le plus employé dans les programmes depuis son invention en 1960. Cette méthode illustre le principe dit « diviser pour régner », qui consiste à appliquer récursivement une méthode destinée à un problème de taille donnée à des sous-problèmes similaires, mais de taille inférieure. Ce principe général produit des algorithmes qui permettent souvent d'importantes réductions de complexité

5.2 Fonctionnement de l'algorithme

Le principe de ce tri est d'ordonner le vecteur T[n] en cherchant dans celui-ci une clé pivot autour de laquelle réorganiser ses éléments. Donc On considère un élément au hasard dans le tableau, le pivot et on procède à une partition du tableau en 2 zones : les éléments inférieurs ou égaux au pivot dans un coté et les éléments supérieurs ou égaux au pivot dans l'autre coté pui on place le pivot dans sa position appropriée. On répète récursivement la procédure sur chacune des partitions créées en considérant un pivot dans chaque partie jusqu'à ce qu'elle soit réduite à une liste à un seul élément. [3]

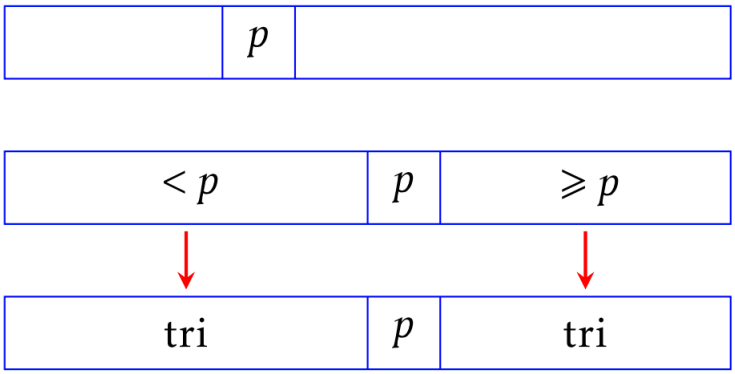


FIGURE 5.1 – Illustration Tri Rapide

Ils existent plusieurs methodes de choix du pivot qui est determinant pour la performance de l'algorithme :

1. Le pivot est le premier élément du tableau
2. Le pivot est le dernier élément du tableau
3. Le pivot est au millieu du tableau
4. Le pivot est choisi au hasard
5. pivot est trouvé en recherchant la médiane

Dans ce TP nous avons testé les trois premières méthodes comme il est demandé dans l'énoncé. Pour programmer le Tri Rapide nous allons utiliser deux procédures :

1. Partition :une fonction qui divise un tableau en entrée en deux sous listes et retourne le pivot. Il peut y avoir plusieurs façons de faire la partition, dans le cas de la première et la deuxième methodes on commençe le parcours du tableau par l'élément le plus à gauche ou à droite selon le pivot choisi et on utilise un compteur p qui garde la trace de l'indice des éléments plus grands que le pivot. Pendant le parcours, si on trouves un élément plus petit que le pivot, on incrémente p et l'élément actuel sera échangé avec Tab[p]. Sinon il sera ignoré. Dans la troisième méthode on parcourt le tableaux à partir des deux extrémités, jusqu'à rencontrer un élément supérieur au pivot dans la prtie droite du tableau ou le contraire dans la pertie gauche et on permute entre les deux.

Partition ainsi que la procédure TriRapide qui assure l'application du même processus sur les sous listes droites et gauches générées (assure la récursivité).

5.2.1 Méthode 1 : Le pivot est le premier élément du tableau

```
Fonction Partition1(Entrée : tab : tableau d'entier, deb :entier, fin :entier)
  Variables :
  i : entier;
  p,pivot : entier;
  début
    p ← deb;
    pivot ← tab[deb];
    pour i ← deb + 1 à fin faire
      si tab[i]< pivot alors
        p++ ;
        permuter(tab[i], p) ;
      fin si
    fin pour
    permuter(tab[p], deb);
    retourner p ;
  fin
```

```
Fonction TriRapide1(Entrée : tab : tableau d'entier, deb :entier, fin :entier)
  Variables :
  pivot : entier;
  début
    si deb< fin alors
      pivot← Partition1(tab,deb, fin) ;
      TriRapide1(tab, deb, pivot − 1) ;
      TriRapide1(tab, pivot+1, fin) ;
    fin si
  fin
```

5.2.2 Méthode 2 : Le pivot est le dernier élément du tableau

```
Fonction Partition2(Entrée : tab : tableau d'entier, deb :entier, fin :entier)
  Variables :
  i : entier;
  p,pivot : entier;
  début
      p ← fin;
      pivot ← tab[fin];
      pour i ← fin − 1 à deb faire
          si tab[i]> pivot alors
              p− ;
              permuter(tab[i], p) ;
          fin si
      fin pour
      permuter(tab[p], fin);
      retourner p ;
  fin
```

```
Fonction TriRapide2(Entrée : tab : tableau d'entier, deb :entier, fin :entier)
  Variables :
  pivot : entier;
  début
      si deb< fin alors
          pivot← Partition2(tab,deb, fin) ;
          TriRapide2(tab, deb, pivot − 1) ;
          TriRapide2(tab, pivot+1, fin) ;
      fin si
  fin
```

5.2.3 Méthode 3 : Le pivot est au millieu du tableau

```
Fonction Partition3(Entrée : tab : tableau d'entier, deb :entier, fin :entier)
  Variables :
  i,j,x : entier;
  pivot : entier;
  début
      x ← (fin-deb)/2 ;
      pivot ←tab[x] ;
      i ← deb − 1 ;
      j ← fin;
      tant que i<=j faire
          tant que tab[i] < pivot ; faire
              i ←i+1 ;
          fin tq
          tant que tab[j] > pivot ; faire
              j ←j+1 ;
          fin tq
          si i<= j alors
              Permuter(tab,i,j) ;
          fin si
      fin tq
      retourner i
  fin
```

```
Fonction TriRapide3(Entrée : tab : tableau d'entier, deb :entier, fin :entier)
  Variables :
  pivot : entier;
  début
      si deb< fin alors
          pivot← Partition3(tab,deb, fin) ;
          TriRapide3(tab, deb, pivot − 1) ;
          TriRapide3(tab, pivot+1, fin) ;
      fin si
  fin
```

5.3 Calcul de complexité

5.3.1 Complexité temporelle

Pour calculer la complexité des algorithmes récursives de Tri Rapide nous utilisons la formule suivante :
le nombre des niveaux de partition * temps nécessaire pour un niveau

sachant que la complexité temporelle pour réaliser la partition est de l'ordre O(n) car à chaque niveau de partitionnement,un total de n éléments sera divisé en partitions gauche et droite (1 × n au premier niveau, 2 × n/2 au deuxième, 4 × n/4 au troisième, etc.) L'effort total est donc le même à tous les niveaux de partitionnement.

1. Meilleur cas : le meilleur ca s'achève lorsque le tableau Tab de taille n se divise sur 2 partie égales de n/2 éléments et les sous tableaux a leurs tour se divisent en deux sous-listes de même taille.

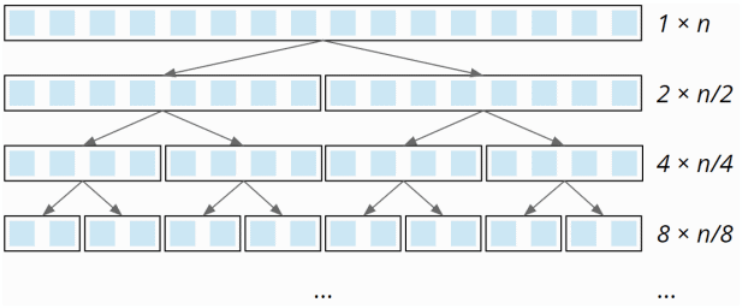


FIGURE 5.2 – Tri Rapide complexité meilleur cas

Le nombre des niveaux de partition :

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\log_2 n = k \log_2 2$$

$$k = \log_2 n$$

Donc Dans le meilleur des cas, la complexité temporelle est : O(n * log n)
Cette complexité est atteinte dans le cas suivant :
le pivot choisi est le premier élément dans le tableau ou le dernier et les éléments sont aléatoires.
le pivot choisi est toujours au millieu du tableau

2. Pire cas : le pire cas correspond au cas où le tableau ne serait pas divisé en deux partitions de tailles approximativement égales, mais une de longueur 0 et une de longueur n-1 (tous les éléments sauf l'élément pivot) alors l'effort de partitionnement décroît linéairement de n à 0 avec la liste droite ou gauche vide selon le pivot choisi. Ainsi nous calculons la comlexité comme suit :

$$n + n - 2 + n - 3 + n - 4... + 2 = \frac{n(n+1)}{2} - 1 = n^2 + n = O(n^2)$$

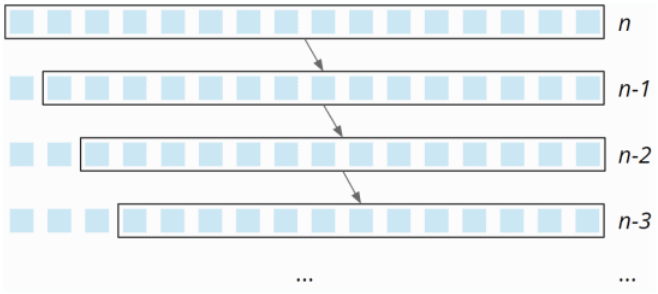


FIGURE 5.3 – Tri Rapide complexité pire cas

Cette complexité est atteinte lorsque le pivot choisi est a la tête ou a la fin du tableau et ce dernier est triée d'une manière descendante ou ascendante.

3. Moyen cas : est atteint lorsque le tableau serait divisé successivement en deux sous-listes de taille à peu près équivalente par exemple n/10 et 9n/10. En utilisons la formule au dessus nous aboutissons à une complexité similaire au meilleur cas O(n * log n).

5.3.2 Complexité spatiale

La complexité spatiale de Tri Rapide est dans tout les cas O(log n) car pour chaque niveau de récursion, nous avons besoin de mémoire supplémentaire sur la pile. Come nous avons vu quand nous avons calculé la complexité temporelle dans le cas moyen et le meilleur cas, la profondeur maximale de récursion est limitée par O(log n) . Dans le pire des cas, la profondeur maximale de récursion est de n

5.4 Experimentation

Dans cette partie nous allons voir les résultats des exécutions des trois méthodes de l'algorithme de Tri Rapide sur différentes taille de tableau qui se représentent en 3 configuration (triée en bon ordre , triée en ordre inverse , aléatoire)

5.4.1 Les données de tableau sont triées en bon ordre

Longueur	Temps d'execution(s) Pivot Premier	Temps d'execution(s) Pivot dernier	Temps d'execution(s) Pivot au millieu
10000	0,029577	0,043917	0,000195
50000	0,823045	1,740042	0,000884
100000	3,04634	6,597959	0,001466
500000	74,614464	164,69075	0,010226
1000000	296,226501	510,830078	0,019255
5000000	/	/	0,112097
10000000	/	/	0,251001
50000000	/	/	1,238545
100000000	/	/	2,378429

FIGURE 5.4 – Tableau représentant le temps d'execution en s des trois méthodes de tri rapide selon les différentes tailles des données d'entrée triées en bon ordre

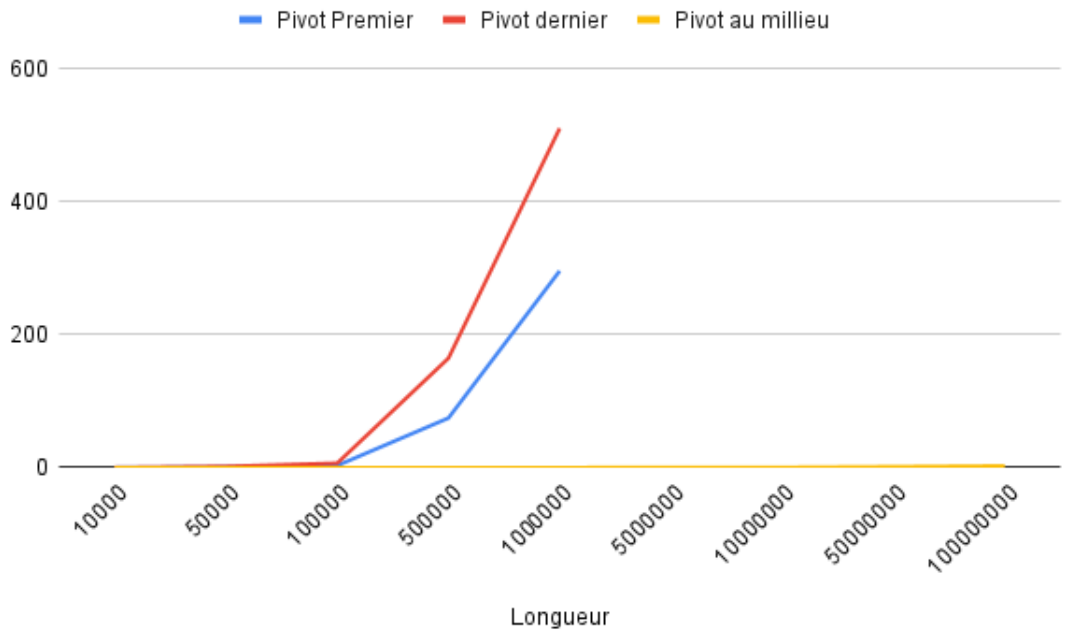


FIGURE 5.5 – graphique représentant le temps d'execution en s des trois méthodes de tri rapide selon les différentes tailles de tableau

5.4.2 Les données de tableau sont triées en ordre inverse

Longueur	Temps d'execution(s) Pivot Premier	Temps d'execution(s) Pivot dernier	Temps d'execution(s) Pivot au millieu
10000	0,07197	0,024216	0,000126
50000	1,617155	0,694654	0,000862
100000	5,861941	3,14895	0,001388
500000	140,326385	80,10989	0,008361
1000000	292,001967	310,336578	0,013707
5000000	/	/	0,099575
10000000	/	/	0,211869
50000000	/	/	1,155541
100000000	/	/	2,172875

FIGURE 5.6 – Tableau représentant le temps d'execution en s des trois méthodes de tri rapide selon les différentes tailles des données d'entrée triées en ordre inverse

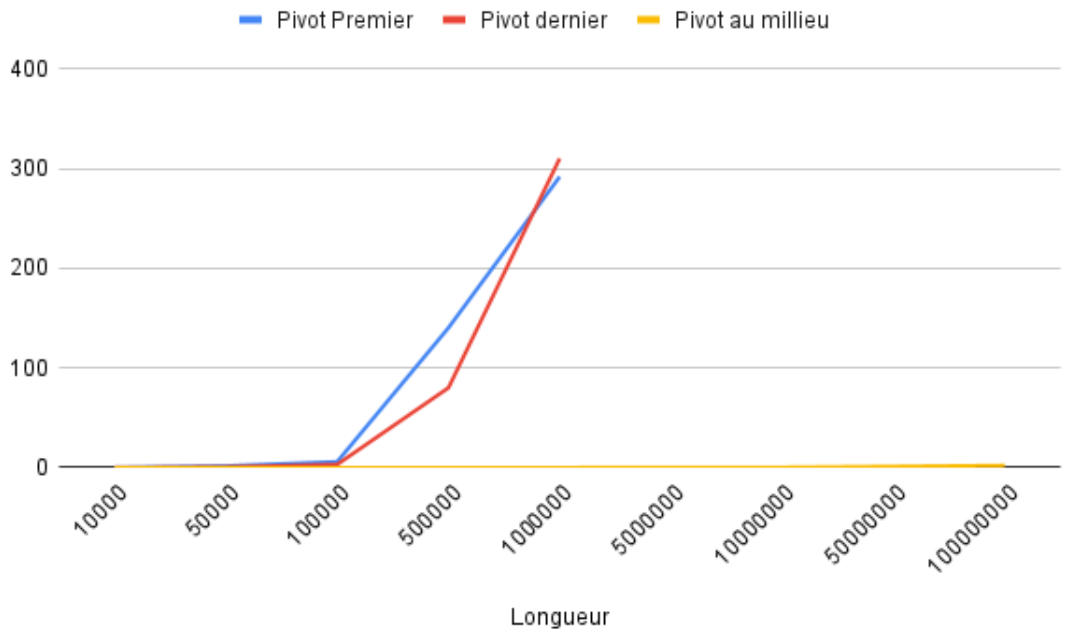


FIGURE 5.7 – graphique représentant le temps d'execution en s des trois méthodes de tri rapide selon les différentes tailles de tableau

5.4.3 Les données de tableau sont positionnés aléatoirement

Longueur	Temps d'execution(s) Pivot Premier	Temps d'execution(s) Pivot dernier	Temps d'execution(s) Pivot au millieu
10 000,00	0,00082	0,000610	0,00061
50 000,00	0,00364	0,003436	0,003348
100 000,00	0.007577	0,009621	0,007963
500 000,00	0,046222	0,047442	0,040366
1 000 000,00	0,096678	0,094327	0,089004
5 000 000,00	0,539326	0,538301	0,485072
10 000 000,00	1,169976	1,136208	1,039578
50 000 000,00	6,207915	6,148364	5,115126
100 000 000,00	13,207039	12,383818	11,578823

FIGURE 5.8 – Tableau représentant le temps d'execution en s des trois méthodes de tri rapide selon les différentes tailles des données d'entrée sont aléatoires

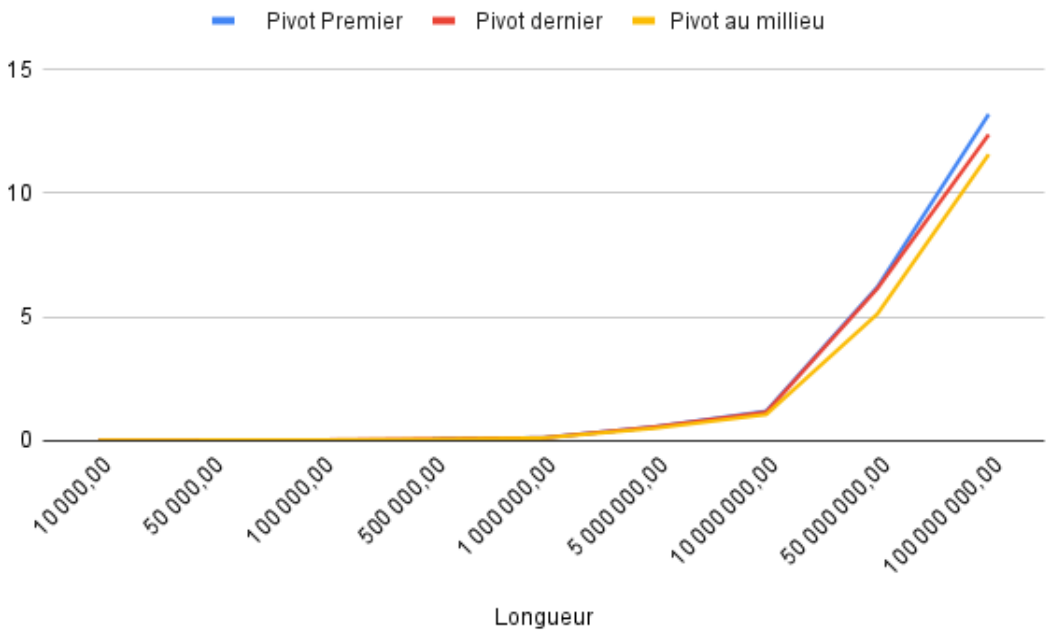


FIGURE 5.9 – graphique représentant le temps d'execution en s des trois méthodes de tri rapide selon les différentes tailles de tableau

5.5 Analyses Des Résultats

Cas Pivot au début : Pour des données d'entrée distribuées de manière aléatoire, le temps nécessaire est légèrement supérieur au double si la taille du tableau est doublée. Cela correspond au temps d'exécution quasi-linéaire attendu O(n log n).

Pour les données d'entrée triées dans l'ordre croissant ou décroissant, le temps requis quadruple lorsque la taille de l'entrée est doublée, nous avons donc un temps quadratique $O(n^2)$.

Le tri des données dans l'ordre décroissant ne prend qu'un peu plus de temps que le tri des données dans l'ordre croissant et l'execution s'arrete si la taille de tableau dépasse 10^6.

Cas Pivot au dérnier : Dans ce cas de choix de pivot les résultas sont approximativement similaires aux ceux de choix de pivot au début : une complexité O(n log n) pour des données de tableau aléatoires et O(n^2) si les données sont triées avec une légère amélioration sur les tableaux triées dans l'ordre décroissant.

cas Pivot au millieu : Pour des données d'entrée triées et non triées l'étude de l'évolution de temps d'execution correspond au temps d'exécution quasi-linéaire attendu O(n log n).
L'algorithme est nettement plus rapide pour les données d'entrée prétriées que pour les données aléatoires.

5.6 Conclusion

Le Tri rapide est l'algorithme de tri le plus utilisé en raison de sa complexité temporelle optimale en moyenne de O(n log n) et sa complexité spatiale O(log n) ce qui en fait un excellent choix pour les situations où l'espace est limité. Bien qu'il offre ces avantages il est considéré comme instable à cause des permutations des éléments et très improbable avec choix du pivot qui affecte considérablement sa complexité où elle atteint O(n^2) dans les pires cas mais il reste le meilleur choix partout où la stabilité n'est pas nécessaire.

Chapitre 6

Tri par fusion

6.1 Description de l'objectif de l'algorithme

En informatique, un tableau est une structure de données représentant une séquence finie d'éléments définis par un index représentant leurs positions au sein du tableau. C'est un type de conteneur que l'on retrouve dans un grand nombre de langages de programmation et est l'un des plus utilisés dû à sa simplicité. Les données du tableau étant accessible individuellement il est nécessaire de faire une recherche lorsque l'on souhaite accéder a une valeur spécifique du tableau. Cependant, lorsque la taille de la structure est grande il devient difficile d'y accéder efficacement.

6.2 Fonctionnement de l'algorithme

Le tri par fusion aussi appeler tri dichotomique est un exemple classique d'algorithme de division pour régner. L'opération principale de l'algorithme est la fusion, qui consiste à réunir deux listes triées en une seule. L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire . On peut résumer son fonctionnement en deux étapes :

1. Divisez la liste non triée en sous-listes jusqu'à ce qu'il y ait N sous-listes avec un élément dans chacune (N est le nombre d'éléments dans la liste non triée).
2. Fusionnez les sous-listes deux à la fois pour produire une sous-liste triée, répétez cette opération jusqu'à ce que tous les éléments soient inclus dans une seule liste.

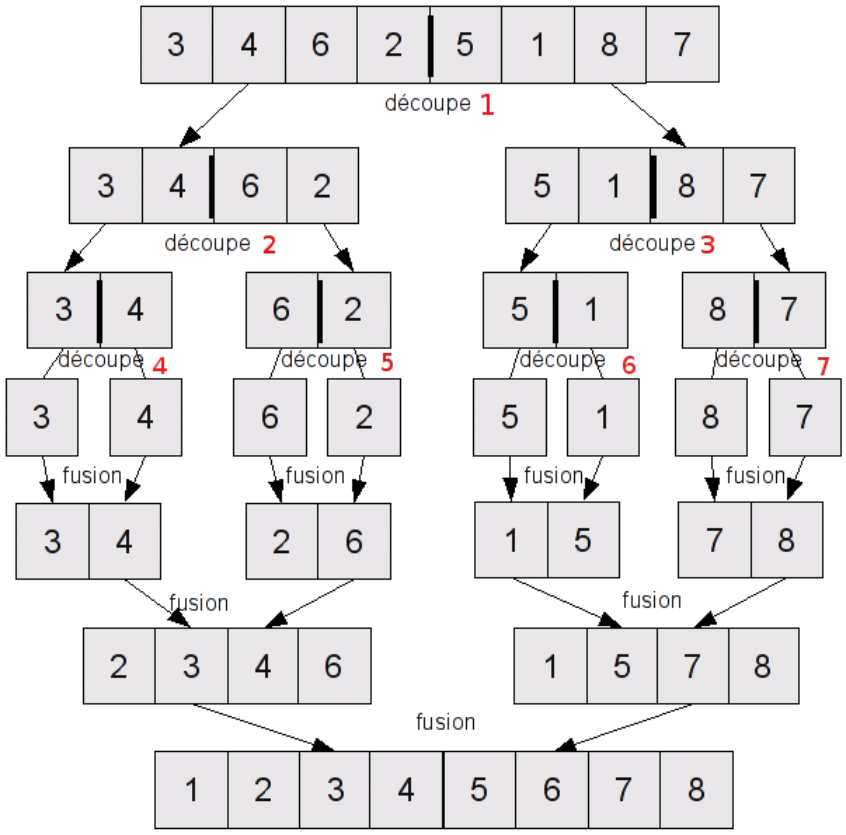


FIGURE 6.1 – Exemple graphique d'un tri par fusion

Afin d'optimiser le déroulement du tri, nous utilisons 2 fonctions distinctes. Une fonction Tri-Fusion qui divise le tableau récursivement et une fonction Fusion qui elle trie les sous tableaux avant de les fusionner à nouveau. Nous pouvons le représenter via le pseudo code suivant :

Fonction Fusion(Entrée : tab : tableau d'entier ; droite, gauche : entier ;)

Variables :

TabG, TabD : tableau d'entier;

SousTab1, SousTab2, SousTab1Index, SousTab2Index, SousTabFusionIndex : entier;

début

$milieu \leftarrow \frac{droite+gauche}{2}$;

$SousTab1 \leftarrow milieu - gauche + 1$;

$SousTab2 \leftarrow droite - milieu$;

pour $i \leftarrow 1$ **à** $SousTab1$ **faire**

$tabG[i] \leftarrow tab[gauche + i]$;

fin pour

pour $j \leftarrow 1$ **à** $SousTab2$ **faire**

$tabG[j] \leftarrow tab[mid + 1 + j]$;

fin pour

$SousTab1 \leftarrow 0$;

$SousTab2 \leftarrow 0$;

$SousTabFusionIndex \leftarrow gauche$;

tant que $SousTab1Index < SousTab1$ *et* $SousTab2Index < SousTab2$ **faire**

si *tabG[SousTab1Index] <= tabD[SousTab2Index]* **alors**

$tab[SousTabFusionIndex] \leftarrow tabG[SousTab2Index]$;

$SousTab1Index + +$;

sinon

$tab[SousTabFusionIndex] \leftarrow tabD[SousTab2Index]$;

$SousTab2Index + +$;

fin si

$SousTabFusionIndex + +$;

fin tq

tant que $SousTab1Index < SousTab1$ **faire**

$tab[SousTabFusionIndex] \leftarrow tabG[SousTab1Index]$;

$SousTab1Index + +$;

$SousTabFusionIndex + +$;

fin tq

tant que $SousTab2Index < SousTab2$ **faire**

$tab[SousTabFusionIndex] \leftarrow tabG[SousTab2Index]$;

$SousTab1Index + +$;

$SousTabFusionIndex + +$;

fin tq

fin

Fonction TriFusion(Entrée : tab : tableau d'entier ; debut, fin : entier ;)

Variables :

milieu : entier;

début

```
// On divise le tableau en 2 de manière recursive puis on les tri avant
   de les fusionner
```

si $debut >= fin$ **alors**

retour;

fin si

```
// On calcule l'index du milieu du tableau
```

$milieu \leftarrow debut + (fin - debut)/2$;

TriFusion(tab, debut, milieu);

TriFusion(tab, milieu + 1, fin);

```
// On utilise la fonction fusion pour trier puis fusioner les sous
   tableaux en un seul tableau trié
```

Fusion(tab, debut, mid, fin);

fin

6.3 Calcul de complexité

6.3.1 Complexité temporelle

la fonction merge consiste a decouper une liste de taille N en N sous-listes avec un élément dans chacune donc on doit parcourir tout le tableau qui est donc de complexite O(n).

Complexité d'une fonction récursive

Pour calculer la complexité d'une fonction récursive, il faut souvent utiliser une formule de récurrence. apres la division , on aura deux problemes a résoudre de taille N/2. On les résout récursivement on exprime donc la complexité au pire cas de merge sort par n(log n + 1) quelque soit l'order des elements de tableau , le tri fusion consiste toujour a le decouper en sous listes jusqu'a ce qu'il ya que des feuilles et les fusionner . **et donc La complexité de tri par fusion est toujours(meilleur , moyenne et pire cas) égale à : O(nln(n)).**

le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme selon la variation de la taille de l'expression :

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t(ns)	10	84.95	200	1349.48	3000	18494.85	40000	500000	6000000	70000000

La figure suivante représente l'évolution du temps d'exécution selon la longueur du tableau :

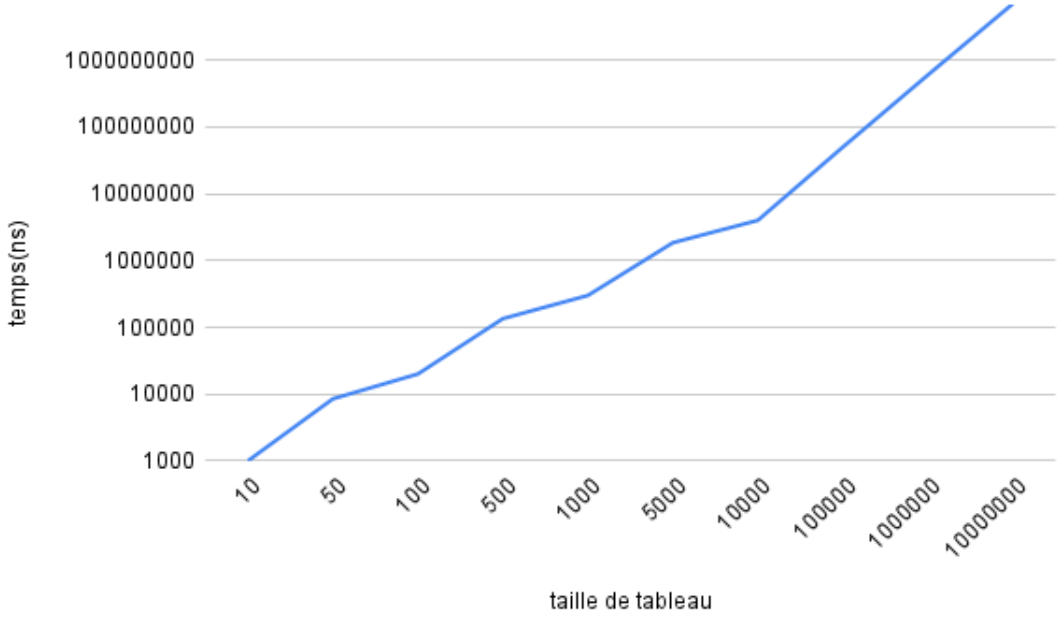


FIGURE 6.2 – Temps d'exécution théorique du programme selon la longueur du tableau

6.3.2 Complexité spatiale

la complexite spatiale de tri fusion est O(n).

6.4 Experimentation

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme selon la variation de la taille et la configuration de l'entree .

Les données du tableau sont triées en ordre inverse.

N	10000	50000	100000	500000	1000000	5000000	10000000	50000000
t1(s)	0.00042	0.002448	0.00494	0.025386	0.055339	0.304839	0.603316	3.768904
t2(s)	0.000808	0.004676	0.009942	0.052479	0.113005	0.624881	1.28433	7.613348
t3(s)	0.001207	0.00691	0.014647	0.081096	0.169904	0.944664	1.942545	11.444372
t4(s)	0.001619	0.009289	0.019483	0.10869	0.227738	1.259653	2.613055	15.351673
t5(s)	0.001993	0.011645	0.024382	0.133484	0.284621	1.568324	3.303382	19.305843
t6(s)	0.00238	0.013891	0.029357	0.160016	0.342119	1.905382	3.922201	23.20739
t7(s)	0.002792	0.016027	0.034042	0.186831	0.400959	2.220602	4.579432	27.165472
t8(s)	0.003176	0.017979	0.038603	0.213543	0.459195	2.531605	5.232867	31.079348
t9(s)	0.003569	0.020067	0.043068	0.241344	0.512189	2.844363	5.811679	34.919087
t10(s)	0.003952	0.022506	0.047414	0.268108	0.569763	3.172519	6.390016	38.880528
Moyenne(s)	0.002192	0.012544	0.026588	0.147098	0.313483	1.737683	3.568282	21.273597

Les données du tableau sont triées en bon ordre.

N	10000	50000	100000	500000	1000000	5000000	10000000	50000000
t1(s)	0.000569	0.002628	0.005863	0.030151	0.061241	0.360749	0.709762	4.124928
t2(s)	0.001068	0.005192	0.011261	0.059449	0.121092	0.682264	1.400675	8.280286
t3(s)	0.001561	0.007656	0.016423	0.088011	0.176175	1.001632	2.058852	12.254911
t4(s)	0.00202	0.010261	0.021879	0.11565	0.228058	1.310229	2.746449	15.998439
t5(s)	0.002476	0.012919	0.027025	0.144914	0.283809	1.661196	3.447879	19.919397
t6(s)	0.002887	0.015353	0.032353	0.1737	0.341154	2.013703	4.154876	23.882657
t7(s)	0.003342	0.017781	0.037866	0.202648	0.398567	2.376409	4.836434	27.617182
t8(s)	0.003832	0.02029	0.043166	0.232642	0.459596	2.74455	5.524018	31.593124
t9(s)	0.004247	0.022741	0.048409	0.262612	0.511865	3.105877	6.199981	35.648717
t10(s)	0.004753	0.025202	0.053627	0.292089	0.564445	3.421129	6.906015	39.616554
Moyenne(s)	0.002675	0.014002	0.029787	0.160187	0.3146	1.867774	3.798494	21.89362

Les données du tableau sont aleatoires.

N	10000	50000	100000	500000	1000000	5000000	10000000	50000000
t1(s)	0.000659	0.003734	0.007724	0.044176	0.082721	0.449917	1.040069	5.768796
t2(s)	0.001049	0.006208	0.012345	0.074345	0.140179	0.734552	1.777799	9.790986
t3(s)	0.001447	0.00852	0.01707	0.103247	0.197034	1.049007	2.509943	13.84349
t4(s)	0.001784	0.011054	0.02166	0.132922	0.252066	1.37088	3.239306	17.834287
t5(s)	0.002294	0.013381	0.026351	0.16232	0.307356	1.704408	3.973541	21.750978
t6(s)	0.002644	0.015765	0.030871	0.192797	0.365797	2.039695	4.707078	25.682164
t7(s)	0.00303	0.018165	0.035004	0.222074	0.420898	2.385981	5.453311	29.653989
t8(s)	0.003439	0.020634	0.039404	0.249832	0.480518	2.729575	6.194547	33.541621
t9(s)	0.003828	0.022892	0.043711	0.278123	0.542099	3.078519	6.927452	37.484699
t10(s)	0.004228	0.02522	0.048058	0.307679	0.596437	3.417302	7.677789	41.405992
Moyenne(s)	0.00244	0.014557	0.02822	0.176752	0.338511	1.895984	4.350083	23.6757

La figure suivante représente l'évolution du temps d'exécution en (s) selon la taille et la configuration du tableau :

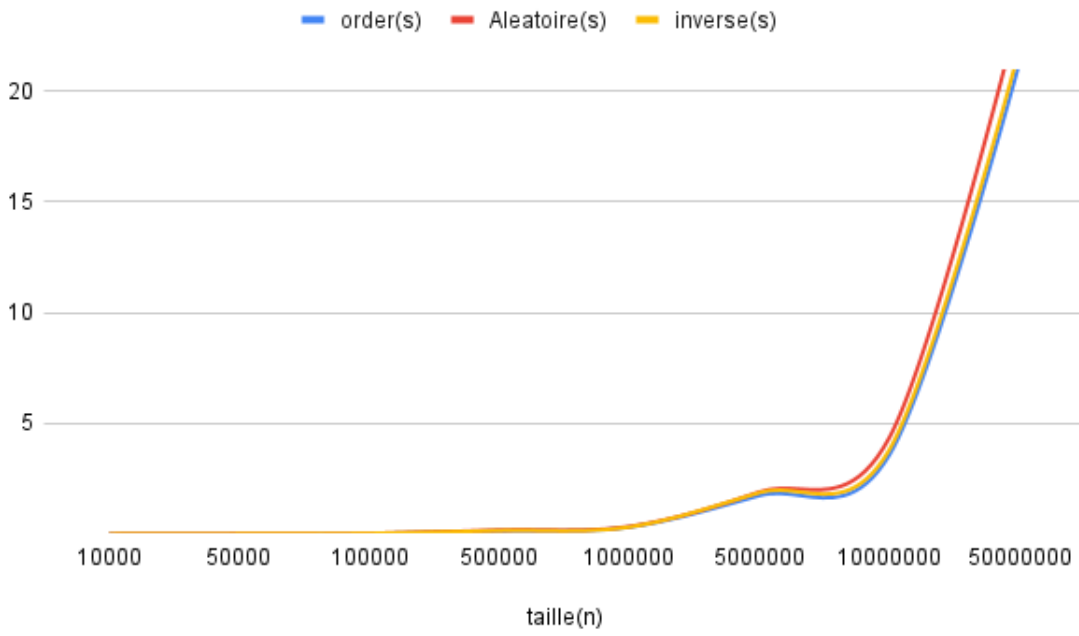


FIGURE 6.3 – Temps d'exécution du programme selon la taille du tableau

Depuis les graphes, on observe que le temps d'exécution évolue de manière linéairethemtique avec l'augmentation de la taille du tableau quelque soit l'ordre, ce qui correspond bien à la complexité théorique calculée auparavant.

6.5 Conclusion

D'apres l'etude theorique et experimentale de la fonction tri fusion on observe que le temps d'execution evolue de maniere lineairethemtique avec l'augmentation de la taille du tableau quelque soit la configuration utilisee , O(nlog(n)) est pratiquement une complexite optimale pour les configuration inverse et aleatoire mais il y'a meilleur fonctions de tri avec une complexite plus optimale lorsque on parle de la configuration en bon ordre.

Chapitre 7

Tri par bulle

7.1 Fonctionnement de l'algorithme

Le tri à bulles ou tri par propagation1 est un algorithme de tri. Il consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés.

Le principe du tri à bulles (bubble sort ou sinking sort) est de comparer deux à deux les éléments x1 et x2 consécutifs d'un tableau et d'effecteur une permutation si x1 > x2. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

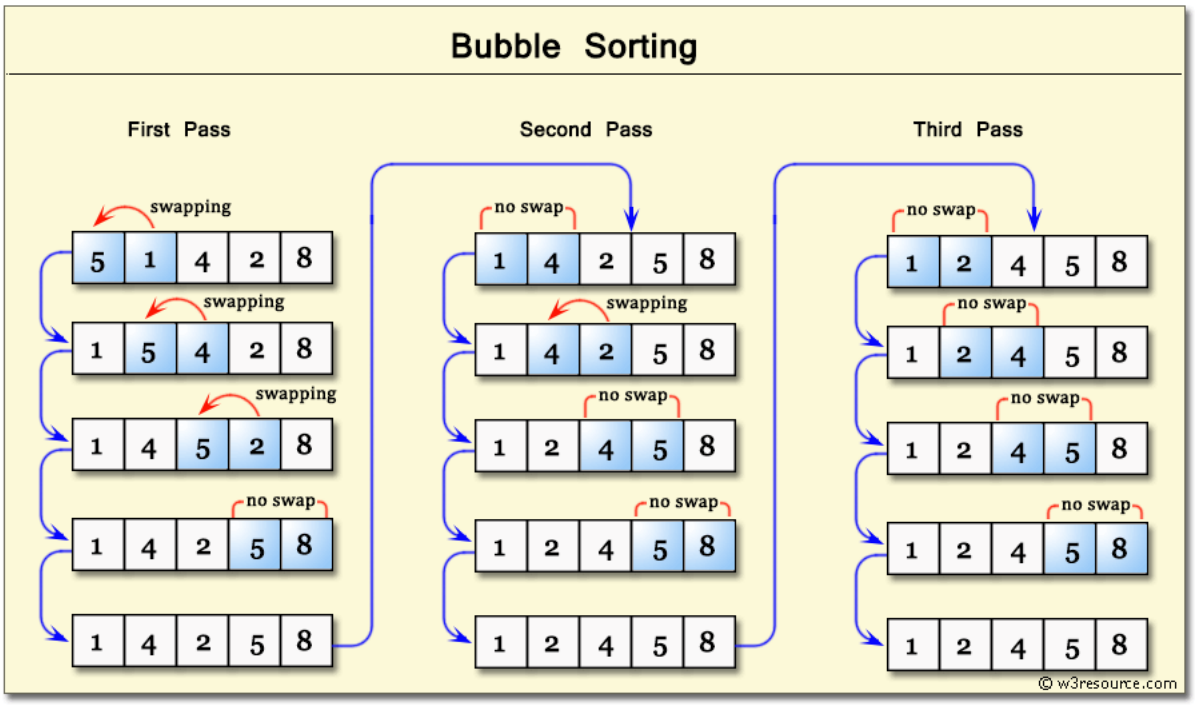


FIGURE 7.1 – Exemple graphique d'un tri par bulle

Nous pouvons le représenter via le pseudo code suivant :

```
Fonction Bulle(Entrée : tab : tableau d'entier ;)
  Variables :
  i,j : entier;
  tmp,b : entier;
  début
    pour i ← 1 à N-1 faire
      b ← true;
      tant que b faire
        pour j ← i + 1 à N-1 faire
          si tab[j] > tab[j+1] alors
            b ← false;
            tmp ← tab[j];
            tab[j] ← tab[j + 1];
            tab[j + 1] ← tmp;
        fin pour
      fin tq
    fin pour
  fin
```

7.2 Calcul de complexité

7.2.1 Complexité temporelle

Pour le tri a bulle le nombre d'itérations de la boucle externe est compris entre 1 et n. il y a exactement n-1 comparaisons et au pire cas n-1 permutations.

Moyenne et Pire Cas : Le nombre de comparaisons "si Tab[j-1] > Tab[j] alors" est une valeur qui ne dépend que de la longueur n de la liste (n est le nombre d'éléments du tableau), ce nombre est égal au nombre de fois que les itérations s'exécutent, le comptage montre que la boucle "pour i de n jusquà 1 faire" s'exécute n fois (donc une somme de n termes) et qu'à chaque fois la boucle "pour j de 2 jusquà i faire" exécute (i-2)+1 fois la comparaison "si Tab[j-1] > Tab[j] alors".

La complexité en nombre de comparaisons est égale à la somme des n termes suivants (i = n, i = n-1,....)

C = (n-2)+1 + ([n-1]-2)+1 +.....+1+0 = (n-1)+(n-2)+...+1 = n(n-1)/2 (c'est la somme des n-1 premiers entiers).

La complexité en nombre de comparaison est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

La Meilleur Cas : (une seule itération) est atteint quand le tableau est déjà trié. Dans ce cas, la complexité est linéaire. O(n)

7.2.2 Complexité spatiale

O(1)

7.3 Experimentation

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme selon la variation de la taille et la configuration de l'entree .

Les données du tableau sont triées en ordre inverse.

N	10000	50000	100000	500000	1000000	5000000	10000000	50000000
Temp(s)	0.034025	0.976482	4.519502	115.577318	//	//	//	//

Les données du tableau sont triées en bon ordre.

N	10000	50000	100000	500000	1000000	5000000	10000000	50000000
Temp(s)	0.000009	0.000034	0.000084	0.000274	0.000704	0.00361	//	//

Les données du tableau sont aleatoires.

N	10000	50000	100000	500000	1000000	5000000	10000000	50000000
Moyenne(s)	0.09877	4.081052	18.926856	493.184478	//	//	//	//

La figure suivante représente l'évolution du temps d'exécution en (s) selon la taille et la configuration du tableau :

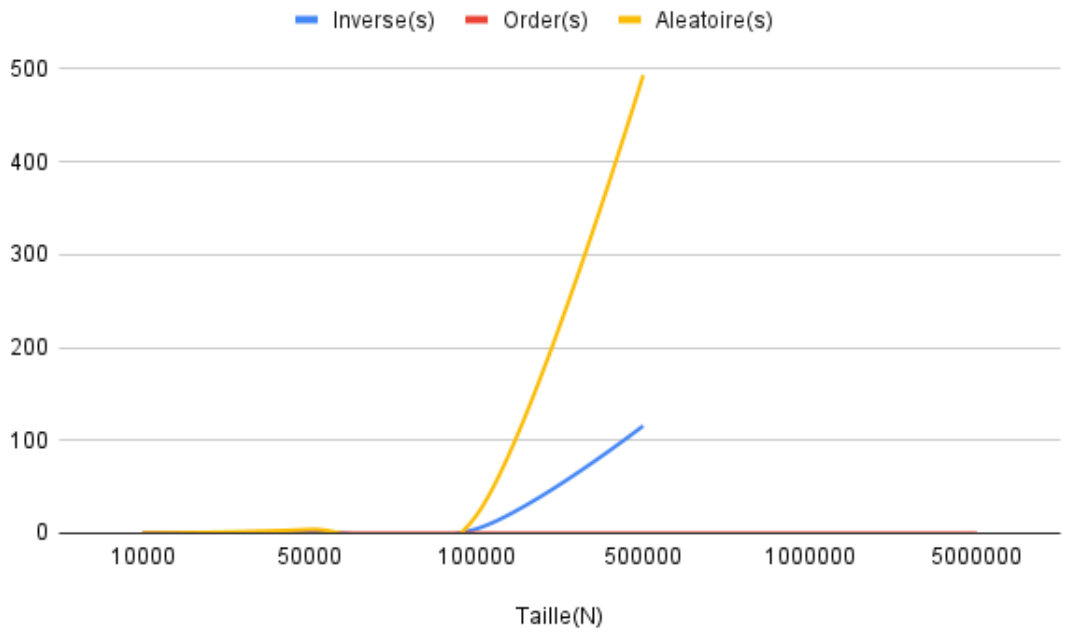


FIGURE 7.2 – Temps d'exécution du programme selon la taille du tableau

Depuis les graphes, on observe que le temps d'exécution évolue de manière quadratique avec l'augmentation de la taille du tableau pour le tri aleatoire et inverse , par contre il s'evolue de maniere linear dans la meilleur cas (tableau tri en bon ordre) , ce qui correspond bien à la complexité théorique calculée auparavant.

7.4 Conclusion

A partir d'étude expérimentale et théorique de l'algorithme tri a bulle, On constate que malgré sa complexité en temps quadratique sur des données inversées ou triées aléatoirement, il est encore largement utilisé car il est capable de s'exécuter en temps linéaire sur des entrées déjà triées.

Chapitre 8

Tri par TAS

8.1 Description

Il existe plusieurs types de méthodes de tri utilisées pour trier différents types de structures de données. L'une des méthodes de tri les plus populaires et les plus efficaces en informatique est le tri par tas.
Le tri par tas est une technique de tri basée sur la comparaison et basée sur la structure de données du tas.
Tas (Heap en anglais) est une structure de données arborescente dans laquelle tous les nœuds de l'arbre sont dans un ordre spécifique. Tas est toujours un arbre binaire complet.
L'élément racine du Tas doit être à l'indice 0 dans le tableau.
Pour le noeud à l'indice i :

- A l'indice (i-1)/2 se trouve le noeud parent.
- A l'indice (2*i)+1, le nœud enfant gauche.
- A l'index (2*i)+2, le noeud enfant de droite.

8.2 Fonctionnement de l'algorithme

L'algorithme du tri par tas fonctionne selon les étapes suivantes :

- A partir des éléments du tableau donné, nous devons faire un tas maximum. Nous commençons à empiler chaque sous-arbre de bas en haut et obtenir un max-heap après avoir appliqué la fonction à tous les éléments, y compris l'élément racine.
- Retirer l'élément racine et le mettre à la fin du tableau (nième position) Mettre le dernier élément de l'arbre (tas) à la place vacante.
- Réduire la taille du tas de 1.
- Heapifier à nouveau l'élément racine de façon à avoir.
- l'élément le plus haut à la racine.

— Répétez ce processus encore et encore jusqu'à ce que tous les éléments de la liste soient triés dans le tableau avec succès.

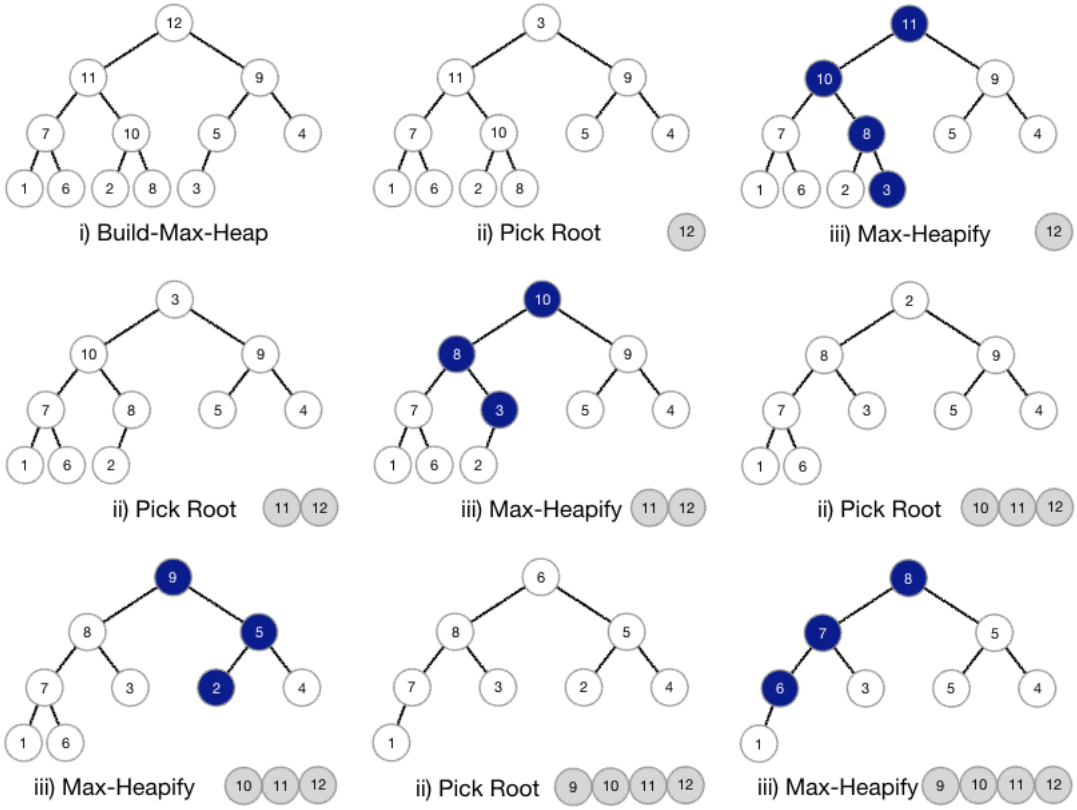


FIGURE 8.1 – Exemple graphique d'un tri par tas

Nous pouvons représenter cet algorithme via le pseudo code suivant :

Fonction heapify(Entrée : tab : tableau d'entier ; k :entier ;)

Variables :

left,right : entier;

tmp,swapwith : entier;

début

$left \leftarrow k * 2 - 1$;

$right \leftarrow k * 2$;

$swapwith \leftarrow k - 1$;

si *left<N et tab[left]>tab[swapwith]* **alors**

$swapwith \leftarrow left$;

fin si

si *right<N et tab[right]>tab[swapwith]* **alors**

$swapwith \leftarrow right$;

fin si

si *swapwith<>k-1* **alors**

$temp \leftarrow tab[swapwith]$;

$tab[swapwith] \leftarrow tab[k-1]$;

$tab[k-1] \leftarrow temp$;

heapify(tab,swapwith);

fin si

fin

Fonction heapSort(Entrée : tab : tableau d'entier ;)

Variables :

temp,wn ,k : entier;

début

pour $k \leftarrow n/2$ **à** *k>0* **faire**

heapify(tab,k);

$k \leftarrow k - 1$;

fin pour

pour $wn \leftarrow n - 1$ **à** *wn>0* **faire**

$temp \leftarrow tab[0]$;

$tab[0] \leftarrow tab[wn]$;

$tab[wn] \leftarrow temp$;

heapify(tab,1);

fin pour

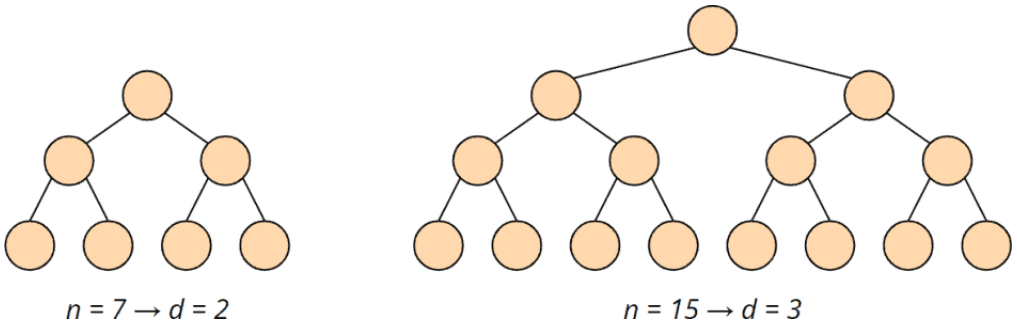
fin

8.3 Calcul de complexité

8.3.1 Complexité temporelle

Heap Sort a des complexités temporelles O(nlog n) pour tous les cas (meilleur cas, cas moyen et pire cas).

Dans la fonction heapify(), nous parcourons l'arbre de haut en bas. La hauteur d'un arbre binaire (la racine n'étant pas comptée) de taille n est log2 n au plus, c'est-à-dire que si le nombre d'éléments double, l'arbre ne devient plus profond que d'un niveau :



La complexité de la fonction heapify() est donc O(log n).

Complexité temporelle de la méthode HeapSort() :

Pour construire initialement le tas, la méthode heapify () est appelée pour chaque nœud parent - en arrière, en commençant par le dernier nœud et en terminant à la racine de l'arbre.

Un tas de taille n a n/2 nœuds parents (arrondis à l'inférieur)

La méthode heapify() est appelée n-1 fois. Ainsi, la complexité totale pour réparer le tas est également O(n log n).

8.3.2 Complexité spatiale

Étant donné que le tri en tas est un algorithme de tri conçu sur place, l'espace requis est constant, donc O (1). En effet, dans le cas de l'entrée :

1. Nous utilisons la structure de tas pour organiser tous les éléments de la liste.
2. Après avoir supprimé le plus grand nœud du tas max, nous plaçons l'élément supprimé à la fin de la même liste.

Par conséquent, nous n'utilisons aucun espace supplémentaire lors de l'implémentation de cet algorithme. Cela donne à l'algorithme une complexité spatiale de O(1).

8.4 Experimentation

Dans cette partie nous allons voir les résultats des exécutions de cet algorithme sur différents taille de tableau et sur données qui se représentent en 3 configuration (triée en bon ordre , triée en ordre inverse , aléatoire)

Les données du tableau sont triées en bon ordre.

Taille	10000	50000	100000	500000	1000000	5000000	10000000	50000000
temps(s)	0,000066	0,000303	0,000682	0,003029	0,00469	0,027033	0,062347	0,315974

Les données du tableau sont triées en ordre inverse.

Taille	10000	50000	100000	500000	1000000	5000000	10000000	50000000
temps(s)	0,000052	0,00028	0,000643	0,002413	0,005757	0,02379	0,049275	0,269919

Les données du tableau sont positionnés aléatoirement.

Taille	10000	50000	100000	500000	1000000	5000000	10000000	50000000
temps(s)	0,000078	0,000384	0,000725	0,002596	0,004857	0,035832	0,057231	0,257452

Le graphe :

La figure suivante représente les résultats d'exécution de cet algorithme selon les différentes taille du tableau.

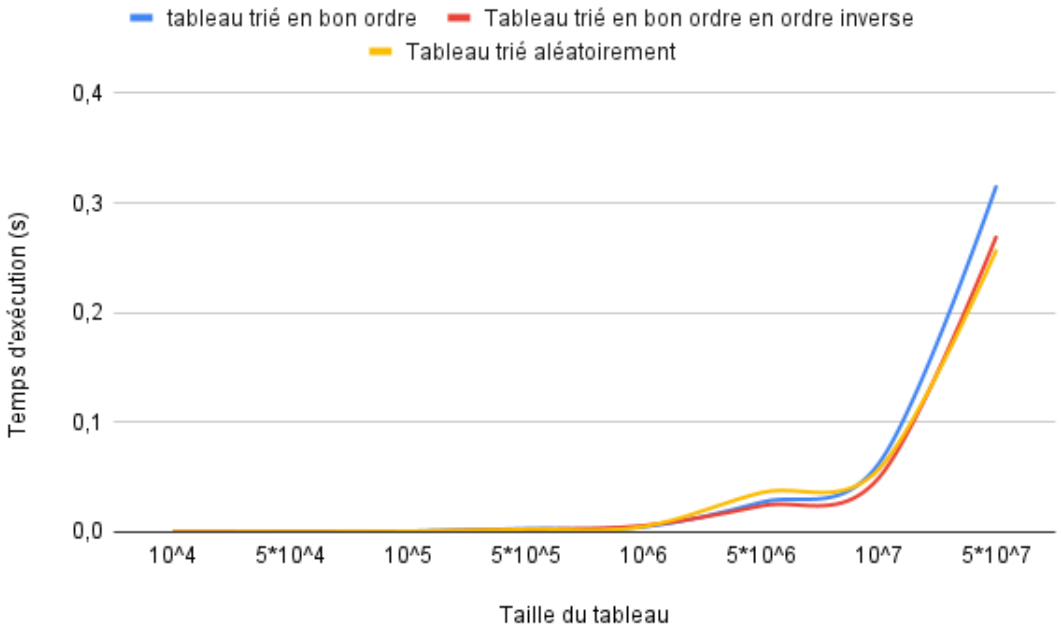


FIGURE 8.2 – Temps d'execution du tri par tas sur les différents taille du tableau

D'après les graphes et les tableaux ci-dessus, Nous remarquons que le temps d'exécution évolue de manière quasi linéaire avec l'augmentation de la taille du tableau quel que soit sa configuration (les 3 courbes sont presque similaires) . On en conclut que la complexité théorique est cohérente avec les résultats expérimentaux.

8.5 Conclusion

A partir d'étude expérimentale et théorique de l'algorithme tri par tas, On remarque que avec sa complexité temporelle de O(n log(n)) le tri en tas est toujours optimal. Il a fonctionné avec succès sur les données que nous avions prises et a obtenu des résultats satisfaisants pour les 3 configurations (table trié en ordre, table trié en inverse , table trié aleatoirement).

Chapitre 9

Nombre Comparaisons

Les tableau suivant représente les nombre de comparaison des algorithmes de tri selon la configuration du tableau.

9.1 Tableau trie en bon ordre

N	10000	50000	10^5	$5*10^5$
Tri par selection	49995000	1249975000	4999950000	124999750000
Tri par insertion	9999	49999	99999	499999
Tri par fusion	1336310	7844810	16689460	94757320
Tri a bulle	16588	56783	107007	508329
Tri tas	244460	1455438	3112517	17837785
Tri rapide (Début)	25005000	625025000	12500050000	62500250000
Tri rapide (Fin)	49995001	1249975001	4999950001	124999750001
Tri rapide (Millieu)	180011	1003409	2106802	11927156

La figure suivante représente l'évolution des nombres de comparaisons selon la taille du tableau 100000

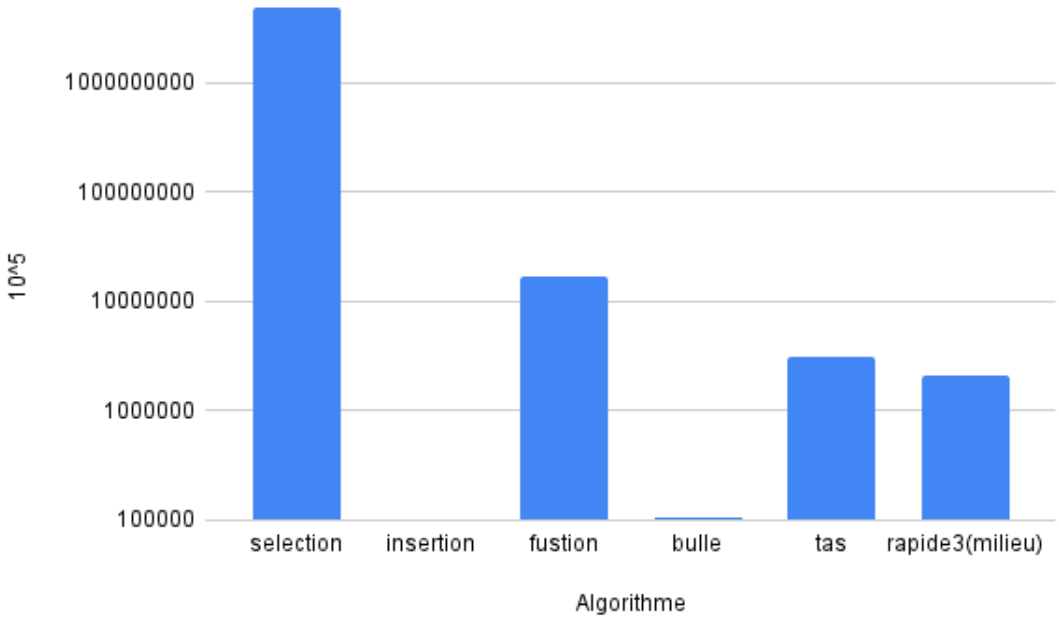


FIGURE 9.1 – nombre des comparaisons selon l'algorithme

Depuis le graphe, on observe que le nombre de comparaison dans la meilleur cas des algorithmes fusion , selection , tas et rapide est tres eleves par rapport aux algorithmes insertion et bulle.

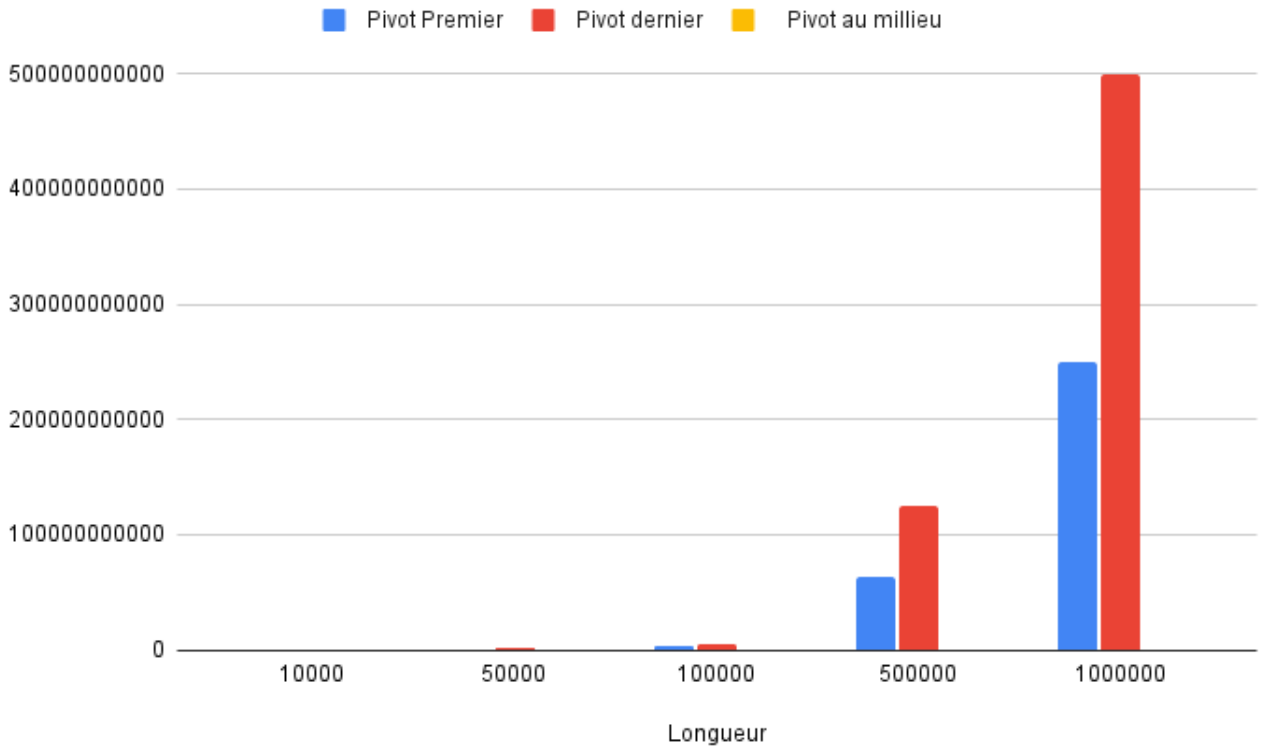


FIGURE 9.2 – Nombre des comparaisons effectuées par les algorithmes de Tri Rapide selon la taille du tableau

9.2 Tableau trie en ordre inverse

N	10000	50000	10^5	$5*10^5$
Tri par selection	49995000	1249975000	4999950000	124999750000
Tri par insertion	49995000	1249975000	4999950000	12499750000
Tri par fusion	1336310	7844810	16689460	94757320
Tri a bulle	50001245	1250026893	5001098891	5001098891
Tri tas	226682	1366047	2926640	16977997
Tri rapide (Début)	50005000	1250025000	5000050000	125000250000
Tri rapide (Fin)	50005000	1250025000	5000050000	/
Tri rapide (Millieu)	143169	848331	1796637	9786465

La figure suivante représente l'évolution du nombre de comparaisons selon l'algorithme utilisé avec une taille du tableau 100000

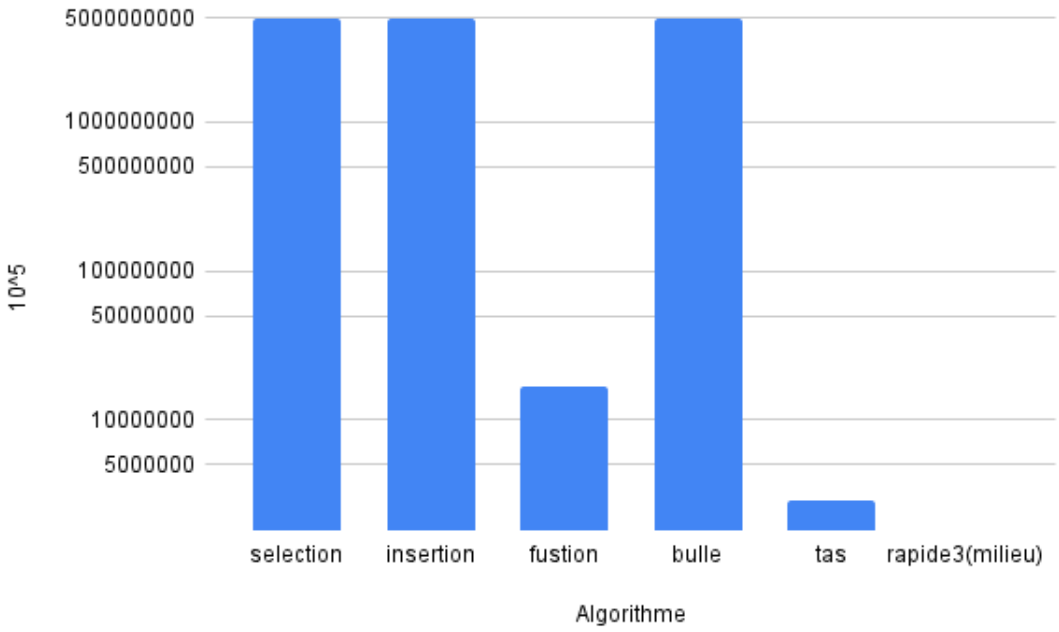


FIGURE 9.3 – nombre des comparaisons selon l'algorithme

Depuis le graphe, on observe que le nombre de comparaison des algorithmes selection ,insertion et bulle est tres eleves par rapport aux autres algorithmes fusion , tas , et le tri rapide donnent des meilleurs resultats.

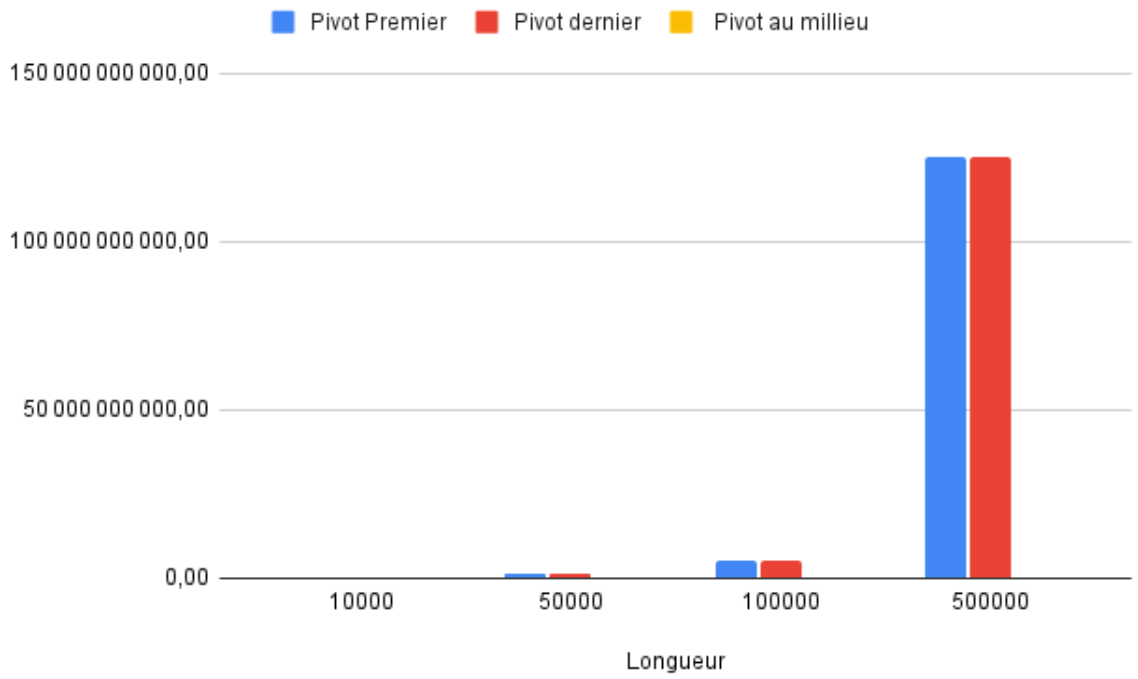


FIGURE 9.4 – Nombre des comparaisons effectuées par les algorithmes de Tri Rapide selon la taille du tableau

9.3 Tableau trie aleatoirement

N	10000	50000	10^5	$5*10^5$
Tri par selection	49995000	1249975000	4999950000	124999750000
Tri par insertion	25234406	623931087	2492077689	62470481149
Tri par fusion	1336310	7844810	16689460	94757320
Tri a bulle	50002183	1250010338	5003278417	//
Tri tas	235334	1409925	3019611	17397152
Tri rapide (Début)	161015	978870	12056258	11881598
Tri rapide (Fin)	168562	993500	2141419	12216452
Tri rapide (Millieu)	262726	1474652	3184007	18730250

La figure suivante représente l'évolution du nombre de comparaisons selon l'algorithme utilisé avec une taille du tableau 100000

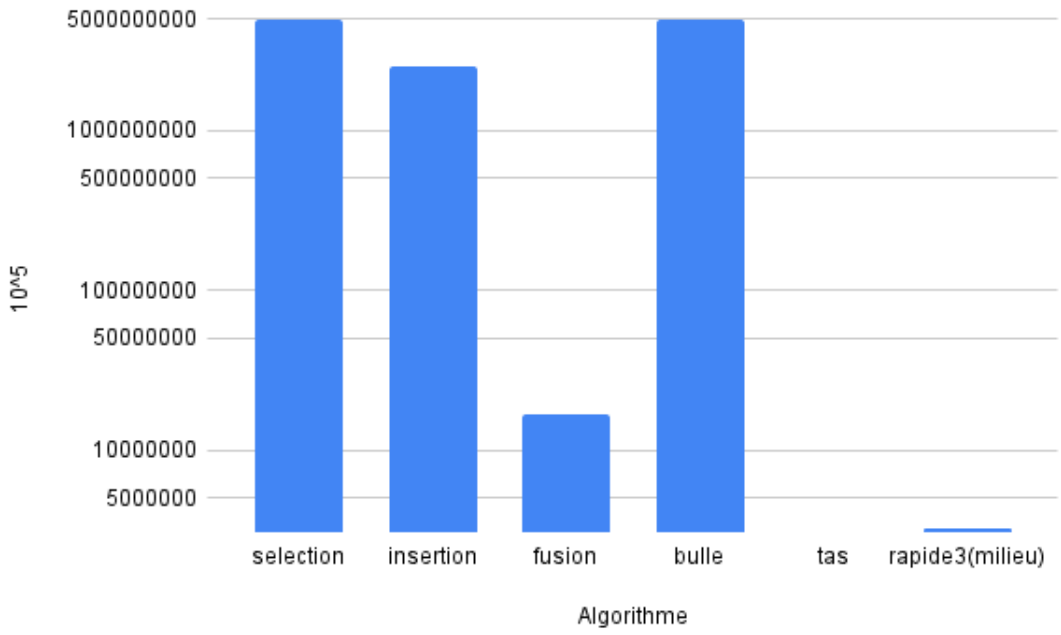


FIGURE 9.5 – nombre des comparaisons selon l'algorithme

Depuis le graphe, on observe que le nombre de comparaison des algorithmes selection ,insertion et bulle est très elevés par rapport aux autres algorithmes fusion , rapide et tas avec meilleur complexite.

9.4 Conclusion

depuis les graphes , on observe que le nombre des comparaisons represénte quasiment la complexité de l'algorithme ainsi qu'ils existent des algorithmes de comparaisons qui ne sont pas couteux en nombres de comparaisons effectuées et ils sont privilégiés dans la pratique dans le cas où la comparaison est dispendieuse.

Chapitre 10

Conclusion

Beaucoup d'algorithmes existent, mais certains sont bien plus utilisés que d'autres en pratique. Le tri par insertion est souvent plébiscité pour des données de petite taille, tandis que des algorithmes asymptotiquement efficaces, comme le tri fusion, le tri par tas ou quicksort, seront utilisés pour des données de plus grande taille.
La comparaison empirique d'algorithmes n'est pas aisée limitée au complexité de l'algorithme, il y'a beaucoup de paramètres entrent en compte : taille de données, ordre des données, matériel utilisé, taille de la mémoire vive, etc.

Chapitre 11

Annexe

11.1 Repartition des taches

Member	Tache
BEBNBACHIR Mohamed Amir	Implimentation des algorithmes et l'etude expiremental selection et a bulle
BOUCHOUL Bouchra	L'étude théorique et expérimentale et implémentation de Tri TAS et insertion
KHEMISSI Maroua	L'étude théorique du du tri par selection ,fusion et bulle ,implimentation et etude experimentale de tri fusion
MEDJKOUNE Roumaissa	L'étude théorique et expérimentale et implémentation de Tri Rapide

11.2 files.c

Le programme c qui genere les fichiers des donnees

```c
#include <stdio.h>
#include <stdlib.h>
#define Length 50000000
int main()
{
    long i;
    FILE *f;
    f=fopen("tablerand.txt","w");

    /*Initialization of the array  */
    for(i=0;i<Length;i++){
       fprintf(f, "%d ", rand());
       //fprintf(f, "%d ", i); for ordered
       //fprintf(f, "%d ", Length-i); for inverse
    }
    fclose(f);

}
```

11.3 main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define Length 50000000

typedef long *array;
array A[Length];
array Tmp[Length];

/*********** Procedure Merge ***********/
void merge(array a, long first,long last){
   long i = first;
   long med = (first+last)/2;
   long j = med+1;
```

```
15      long k = 0;
16      while (i<=med && j<=last){
17              if (a[i] < a[j]){
18              Tmp[k] = a[i];
19              i++;
20              k++;
21              }
22          else{
23              Tmp[k] = a[j];
24              k++;
25              j++;}
26       }
27      while (i<=med) {Tmp[k] = a[i]; k++;i++;}
28      while (j<=last) {Tmp[k] = a[j]; k++; j++;}
29
30      for (j=0; j<k; j++)
31           a[first+j] = Tmp[j];
32  }
33  /*********** Procedure merge sort ************/
34  void mergeSort(array a, long first, long last)
35  {
36      long middle,i,j;
37      if (first>=last) return;
38      middle = (first+last)/2;
39      mergeSort(a,first,middle);
40      mergeSort(a,middle+1,last);
41      merge(a,first,last);
42  }
43  /********* bubblesort*******/
44  void bubsort(int* T,int n){
45       int  temp;
46       int b;
47       for (int i =1;i<n;i++){
48           b=1;
49           for (int j=0;j<n-i;j++){
50               if (T[j]>T[j+1]){
51                   b=0;
52               temp=T[j];
53               T[j]=T[j+1];
54               T[j+1]=temp;}
```

```
55          }
56          if(b)break;
57
58      }
59  }
60  /********** selection ********/
61  void selection(int* T,int n){
62      int  min,temp;
63      for (int i =0;i<n;i++){
64          min =i;
65          for (int j=i+1;j<n;j++){
66              if (T[j]<T[min]){
67                  min=j;
68              }
69          }
70          temp=T[min];
71          T[min]=T[i];
72          T[i]=temp;
73
74      }
75  }
76
77  /*********** tri insertion ********/
78  void insertion(long *T, long n) {
79    long j, temp;
80    for (long i = 1; i < n; i++) {
81      temp=T[i];
82      j=i-1;
83      while ((T[j]>temp) && (j>=0)) {
84              T[j+1]=T[j];
85              j=j-1;
86              }
87        T[j+1]=temp;
88    }
89  }
90  /************ tri tas **********/
91  void swap(long* a, long* b)
92  {
93
94      long temp = *a;
```

```
95
96      *a = *b;
97
98      *b = temp;
99  }
100
101 // To heapify a subtree rooted with node i
102 // which is an index in arr[].
103 // n is size of heap
104 void heapify(long arr[], long N, long i)
105 {
106     // Find largest among root, left child and right child
107
108     // Initialize largest as root
109     long largest = i;
110
111     // left = 2*i + 1
112     long left = 2 * i + 1;
113
114     // right = 2*i + 2
115     long right = 2 * i + 2;
116
117     // If left child is larger than root
118     if (left < N && arr[left] > arr[largest])
119
120         largest = left;
121
122     // If right child is larger than largest
123     // so far
124     if (right < N && arr[right] > arr[largest])
125
126         largest = right;
127
128     // Swap and continue heapifying if root is not largest
129     // If largest is not root
130     if (largest != i) {
131
132         swap(&arr[i], &arr[largest]);
133
134         // Recursively heapify the affected
```

```
135         // sub-tree
136         heapify(arr, N, largest);
137     }
138 }
139 void heapsort(long *T, long n) {
140   long temp;
141   for (long k = n / 2-1; k >= 0; k--) {
142     heapify(T, n, k);
143   }
144   for (long wn = n - 1; wn >= 0; wn--) {
145     temp = T[0];
146     T[0] = T[wn];
147     T[wn] = temp;
148     heapify(T, wn, 0);
149   }
150 }
151 /***************Tri Rapide***************/
152   void permuter(array t, long a, long b) {
153   long temp;
154   temp = t[a];
155   t[a] = t[b];
156   t[b] = temp;
157 }
158 //pivot au millieu
159 long partition_1(array tab, long deb, long fin) {
160   long p = deb;
161   long pivot = tab[deb];
162   for (long i = deb + 1; i <=fin; i++) {
163     if (tab[i] < pivot) {
164       p++;
165       permuter(tab, i, p);
166     }
167   }
168   permuter(tab, p, deb);
169   return (p);
170 }
171
172 void tri_rapide_1(array tab, long deb, long fin) {
173   if (deb < fin) {
174     long piv = partition_1(tab, deb, fin);
```

```
175     tri_rapide_1(tab, deb, piv - 1);
176     tri_rapide_1(tab, piv + 1, fin);
177   }
178 }
179 //pivot a la fin
180 long partition_2(array tab, long deb, long fin) {
181   long p = fin;
182   long pivot = tab[fin];
183   for (long i = fin - 1; i >=deb; i--) {
184     if (tab[i] > pivot) {
185       p--;
186       permuter(tab, i, p);
187     }
188   }
189   permuter(tab, p, fin);
190   return (p);
191 }
192
193 void tri_rapide_2(array tab, long deb, long fin) {
194   if (deb < fin) {
195     long piv = partition_2(tab, deb, fin);
196     tri_rapide_2(tab, deb, piv - 1);
197     tri_rapide_2(tab, piv + 1, fin);
198   }
199 }
200
201
202 //pivot au millieu
203 void tri_rapide_3(long list[], long left, long right)
204 {
205     if (left >= right) return;
206     long mid = (left + right) / 2;
207     long i = left, j = right;
208     long pivot = list[mid];
209     long k = 0;
210
211     while (left <= right) {
212         while (list[left] < pivot) left++;
213         while (list[right] > pivot) right--;
214         if (left <= right) {
```

```
            permuter(list,left,right);
            left++; right--;
        }
    }
    tri_rapide_3(list, i, right);
    tri_rapide_3(list, left, j);
}
int main(void) {

  long Ns[8] = {10000,   50000,   100000,   500000,
                1000000, 5000000};
  FILE *f, *fptr;
  clock_t t1, t2;
  double delta;
  fptr = fopen("log rand bullr.txt", "w");
  for (int i = 0; i < 4; i++) {
    f = fopen("tablerand.txt", "r");
    for (long j = 0; j < Ns[i]; j++) {
      //T[j] = j;
      fscanf(f, "%li", &T[j]);
    }
    t1 = clock();
    printf("%li com\n",bubsort(T, Ns[i]));
    t2 = clock();
    delta = (double)(t2 - t1) / CLOCKS_PER_SEC;
    fprintf(fptr, "%f ,", delta);
    printf("%li items %f s\n ",Ns[i],delta);
  }
  fclose(fptr);
  for (long i = 0; i < 100; i++) {
    printf("%li\n", T[i]);
  }
  return 0;
}
```

Chapitre 12

Bibliographie

[1] Article wikipedia : tri fusion https ://fr.wikipedia.org/wiki/Tri$_{f}usion$

[2] Video ALGO1 - Chapitre 4 : Récursivité - Partie 2 : Compter la complexité, Diviser pour Régner : https ://youtu.be/h1Wto5KtZBc [3]Thomas H. Cormen, Algorithmique. Dunod, 3è édition, 2010