algo lma
laboratoire d'algorithmique
laboratoire de mathematiques algorithmique

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Sorting Algorithms

Implementation of a variety of sorting algorithms, and look at their behavior and running times on different inputs.

Ramatou ABIBOU
Semestre 5, Systèmes de Communication
Faculté I&C, EPFL
Email : **ramatou.abibou@epfl.ch**

Assistant Responsable : Andrew BROWN

Professeur : Amin SHOKROLLAHI

Semestre d'Hiver 2004

Résumé

Un sujet majeur d'algorithmique est le tri d'objets tels que les tableaux. Le choix du meilleur algorithme pour une tache particulière peut être un processus compliqué, qui exige souvent des analyses mathématiques sophistiquées.

La plupart des algorithmes de tri consistent simplement à ranger des nombres d'un tableau dans l'ordre numérique ou des caractères par ordre alphabétique : il est donc important de comprendre que les algorithmes sont très largement dépendant du type des éléments à trier, et qu'en fin de compte il n'est pas difficile de passer à une généralisation.

Les algorithmes de tri présentés dans ce document sont soit :

- ***élémentaires*** : tri à bulles, tri par sélection, tri par insertion et tri par shell
- ***complexes*** : tri rapide (quicksort), tri trois-médiane, tri par tas (heapsort)

Il est aussi important de préciser qu'il s'agit ici d'**algorithmes de tri internes** qui supposent que l'accès aléatoire à chaque élément est possible avec le même coût, c'est par exemple le cas si les données sont stockées dans la mémoire vive.

Ces algorithmes sont étudiés sous l'axe d'analyse et de la performance. Cette performance a été mesurée sur les mêmes outils et dans les mêmes conditions (même ordinateur, même compilateur) pour chaque méthode afin de ne péjorer l'une d'entre elles et avoir une base de comparaison fiable. La performance dans notre cas se définit par le temps que prend l'algorithme pour exécuter le tri d'un tableau et le nombre total de comparaisons et de mouvements (un échange correspond à trois mouvements) que l'algorithme effectue pour trier l'objet soumis (dans notre cas un tableau)
Pour analyser les différents algorithmes de tri, nous avons implémenté en langage de programmation C et calculé le nombre d'opérations fondamentales effectué par chaque méthode de tri ci-dessus cité en fonction des éléments à trier. Cette méthode nous a permis d'analyser le comportement de chaque algorithme de tri appliqué à un tableau presque trié, non trié, et trié dans l'ordre inverse. Cette méthodologie nous a permis d'arriver aux conclusions suivantes :

- de manière générale, les tris complexes sont plus performants que les méthodes de tri élémentaires en regard d'un volume de données important
- Le choix de la méthode de tri dépend fortement du volume de données à trier. L'effort d'implémentation des méthodes de tri complexes doit être mis dans la balance avec leur performance afin d'effectuer le choix le plus adéquat et le plus optimal
- Le tri Shell qui requiert peu de code est un bon compromis performance Versus Volume de données si les incréments sont bien définis.

Table des Matières

RESUME -- 2

TABLE DES MATIÈRES-- 3

1. INTRODUCTION -- 5

1.1 Historique des Algorithmes de Tri.--6

1.2 Situation actuelle et perspectives des Algorithmes de Tri. --6

2. TRIS ELEMENTAIRES -- 8

2.1 Tri à Bulles (Bubble Sort)--8
2.1.1 Méthode et Implémentation--8
2.1.2 Analyse et Performance Théoriques du Tri à bulles -- 10
2.1.3 Résultats des Tests Effectués-- 12
2.1.4 Courbes Obtenus-- 12
2.1.5 Observations et Conclusions sur l'Algorithme du Tri à Bulles------------------------------------ 15

2.2 Tri par Sélection (Selection Sort)--- 15
2.2.1 Méthode et Implémentation du Tri par Sélection-- 15
2.2.2 Analyse et Performance Théoriques du Tri par Sélection -- 17
2.2.3 Résultats des Tests Effectués-- 18
2.2.4 Courbes Obtenus-- 18
2.2.5 Observations et Conclusions sur le Tri par Sélection --- 20

2.3 Tri par Insertion (InsertionSort)--- 21
2.3.1 Méthode et Implémentation -- 21
2.3.2 Analyse et Performance Théoriques du Tri par Insertion --- 22
2.3.3 Résultats des Tests Effectués-- 24
2.3.4 Courbes Obtenus-- 24
2.3.5 Observations et Conclusions sur l'Algorithme du Tri par Insertion------------------------------- 25

2.4 Tri par Shell (Shell Sort) --- 26
2.4.1 Méthode et Implémentation -- 26
2.4.2 Analyse et performance Théoriques du Tri -- 28
2.4.3 Résultats des Tests Effectués-- 28
2.4.4 Courbes Obtenus-- 28
2.4.5 Observations et Conclusions sur l'Algorithme du Tri Shell-- 28

2.5 Comparaisons des Tris Elémentaires -- 29

3. TRIS SOPHISITIQUES OU COMPLEXES---31

3.1 Tri Rapide (quiksort)--- 31
3.1.1 Méthode et Implémentation-- 31
3.2.2 Analyse et performance du Théoriques du Quicksort-- 33
3.2.3 Résultats des Tests Effectués-- 35
3.1.4 Courbes Obtenus-- 35
3.1.5 Observations et Conclusions sur l'Algorithme du Quicksort--------------------------------------- 36

3.2 Tri Trois-Mediane : Amélioration du tri rapide par la stratégie du choix du pivot ------------------ 36
3.2.1 Méthode et Implémentation --- 37
3.2.2 Analyse et performance Théoriques du Tri --- 38
3.2.3 Résultats des Tests Effectués --- 39
3.2.4 Courbes Obtenus --- 39
3.2.5 Observations et Conclusions sur l'Algorithme du Tri 3-mediane --- 40

3.3 Tri par Tas (Heapsort) -- 40
3.3.1 Méthode et Implémentation --- 41
3.3.2 Analyse et Performance Théoriques du Heapsort -- 45
3.3.3 Résultats des Tests Effectués --- 46
3.3.4 Courbes Obtenus --- 46
3.3.5 Observations et Conclusions sur l'Algorithme du Heapsort -- 47

3.4 Comparaisons des Tris Sophistiqués --- 48

4. CONCLUSION GENERALE -- 48

5. ANNEXE DES RESULTATS DES TESTS -- 53

1. INTRODUCTION

Un **algorithme** décrit un traitement sur un certain nombre fini de données. Il est fait d'un ensemble fini d'étapes.

Qu'est-ce qu'un tri ? On suppose qu'on se donne un suite de N nombres entiers, et on veut les ranger en ordre croissant (ou décroissant) au sens large. Ainsi, pour n = 7, la suite (5, 2, 3, 0, 6, 1, 1) devra devenir (0, 1, 1, 2, 3, 5, 6).

C'est ainsi que plusieurs **algorithmes de tri** sont développés. Ils permettent de « trier » ou ranger une série de données dans un ordre précis (Par exemple dans un ordre alphabétique). Les algorithmes de tri sont très importants en pratique, en particulier sur les graphes (tri topologique ou TopSort en Anglais, Kushukal, etc), dans les systèmes d'exploitation et aussi dans le domaine de l'informatique de gestion où beaucoup d'applications consistent à trier des fichiers.
De plus on estime que plus que 25% du temps de calculs utilisé commercialement est consacré aux opérations de tri (*sorting*). Ceci illustre l'importance du développement des méthodes de tri puissantes. Mais ces dernières peuvent être lentes ou rapides selon la taille des données à trier ou si le tableau est presque trié, non trié et trié dans l'ordre inverse.

Afin de contribuer en partie à la réduction de temps de calcul (25%), le but de ce projet est d'analyser et de comparer des algorithmes pour prédire quelle méthode de tri conviendrait le mieux aux données à trier : **d'où le problème du choix optimal des méthodes de tris**.

Dans tout ce qui suit, on suppose que l'on **trie des nombres entiers et que ceux-ci se trouvent dans un tableau.**

Complexité d'un algorithme :
Analyser un algorithme revient à prévoir les ressources (i.e. la quantité de mémoire) nécessaires à cet algorithme et à mesurer son temps d'exécution. En général, quand on analyse plusieurs algorithmes candidats pour un problème donné, on arrive aisément à identifier le candidat le plus efficace. Ce type d'analyse peut révéler plusieurs candidats valables et permet d'éliminer les autres.

Comme la plupart des méthodes de tri interne, les algorithmes effectuent deux types d'opérations : des comparaisons entre clés et des échanges d'éléments.
On compte le nombre de ces opérations fondamentales pour un tableau de N éléments, ce qui donne une idée de la complexité.

Si on s'intéresse à l'application des tris dans la réalité, souvent on a des données de grandes tailles à trier, c'est donc nécessaire de voir comment ces méthodes de tri se comportent quand la taille du tableau devient grande. D'où le nom de la complexité : bonne inférieure et supérieure. Plusieurs chercheurs ont travaillé sur la complexité des algorithmes de tri et ont confirmé les résultas ci-dessus que nous avons tester au cous de ce projet et ça s'avère vrai
L'analyse d'un algorithme, même simple, peut s'avérer difficile. Il est donc nécessaire de se donner des outils mathématiques pour parvenir à nos fins.

Notation asymptotique
Les fonctions que l'on considère dans cette section sont des fonctions de N dans N. Soient f et g deux fonctions.
Définition 1 : On dira que f est O(g) si et seulement si il existe $c > 0$ et n0 tel que $f(n) <= c \cdot g(n)$
Définition 2 : On dira que f est Ω(g) si et seulement s' il existe $c > 0$ et n0 tel que $f(n) > c \cdot g(n)$
Définition 3 : On dira que f est o(g) si et seulement si lim (f(n)/g(n))= 0,si n tend vers l'infini.
Définition 4 : On dira que f est ɵ (g) si et seulement s'il existe c1, c2 > 0 et n0 tel que c1g<=f<=c2g, on a également ɵ (g) =O(f(n))∩Ω(g(n)
On pourra étudier plus tard dans ce rapport, grâce à ces notions ci-dessus, la complexité de différents algorithmes implémentés et tester avec divers tableaux.

1.1 Historique des Algorithmes de Tri.

Le terme "Algorithme" vient du mathématicien perse al-khowarizmi, traduit en latin par algorismus.

Vers l'an 400 avant Jésus-Christ : premier algorithme non trivial, Euclide et son fameux algorithme de calcul du PGCD

1.2 Situation actuelle et perspectives des Algorithmes de Tri.

L'algorithmique est plus qu'une branche de l'informatique (Informatique vue ici comme la science des ordinateurs). Partout où l'objectif de"mécanisation" (pas toujours réalisable) est requis, l'algorithme s'avère très utile et efficace.

Les perspectives d'avenir sont modestes en considérant que les ordinateurs sont aujourd'hui toujours incapables de rivaliser avec des fonctions cognitives pourtant basiques chez l'animal et l'homme. Au delà du service offert par les machines, les scientifiques informaticiens apportent néanmoins des modèles et des algorithmes impactant fortement d'autres disciplines (Informatique de Gestion, biologie, chimie, …)

On peut aussi noter qu'un programme de tri est généralement une seule fois, voire peu de fois. Lorsqu'un algorithme de tri est résolu pour un ensemble de données, il n'est plus nécessaire dans l'application qui manipule celle-ci. Si un algorithme de tri élémentaire est aussi rapide que par exemple la saisie des données ou l'impression, alors il n'est pas nécessaire de chercher un moyen plus efficace d'opérer.

Dans la pratique, les fichiers à trier sont soit de petite taille mais très nombreux, soit alors partiellement triés ou même déjà, ou alors contiennent un grand nombre de clés dupliqués : pour ces fichiers, les algorithmes élémentaires s'avèrent mieux adaptés, tandis que les algorithmes sophistiqués entraînent des surcharges les rendant moins efficaces.

De nos jours, on continue à optimiser les algorithmes de tris en améliorent leur implémentation mais sans pour pouvoir changer radicalement leur complexité, c'est-à-dire les bornes supérieures et inférieures.
Au lieu de penser à développer de nouveaux algorithmes de tris hyper-sophistiqués dans le futur, il serait sûrement intéressant de contourner ou régler les failles ou carences des algorithmes élémentaires, ceci afin d'espérer éviter les éventuels problèmes de surcharge tout en bénéficiant d'une bonne complexité.

2. TRIS ELEMENTAIRES

Ils sont au nombre de quatre étudiés dans ce projet, leur implémentation est facile et requiert très peu de code.

Les tableaux à trier sont représentés comme suit : a[0],a[1],…a[i]…a[N-1] avec N la taille du tableau où i est l'indice du ième élément du tableau.

Par convention dans ce rapport, on note par :

- **MoyC**, **MaxC** et **MinC** respectivement le nombre de comparaisons moyen (cas moyen), maximal (pire des cas) et minimal (meilleur des cas).
- **MoyE**, **MaxE** et **MinE** respectivement le nombre d'échanges moyen, maximal et minimal.
- **MoyM**, **MaxM** et .**MinM** respectivement le nombre de mouvements moyen, maximal et minimal.

Un échange correspond à trois mouvements.

2.1 Tri à Bulles (Bubble Sort)

Le nom de ce tri vient de ce que les éléments les plus grands (lourds) remontent vers la fin du tableau comme les bulles vers le haut d'un tube à essai.
C'est le tri le plus simple que nous allons étudier.

2.1.1 Méthode et Implémentation

Le tri à Bulle est une méthode de tri qui consiste à comparer successivement tous les éléments adjacents d'un tableau et à les échanger si le premier élément est supérieur au second. On recommence cette opération tant que tous les éléments ne sont pas triés. A chaque étape de l'algorithme l'élément maximal est déplacé à la fin de la suite.

Voici un exemple d'application de cette méthode sur un tableau de taille N=6

Données :

a[0]	a[1]	a[2]		a[3]	a[5]
115	101	30	63	20	47

1ère passage :

Echanges : 101↔115

30↔115

63↔115

20↔115
47↔115

Résultat du 1er passage

101 30 63 20 47	115

Au 1er passage l'élément (115) ,le plus grand du tableau est déplacé en N-1 (ici 5ème) position

2ème passage :
Echanges : 30↔101
63↔101
20↔101
47↔101

Résultat du 2ème passage

30 63 20 47	101	115

Au 2ème passage l'élément (101),deuxième plus grand élément du tableau est déplacé en N-2(ici 4ème) position

3ème passage :
pas d'echanges : 30 < 63

Echanges: 20↔6
47↔63

Résultat du 3ème passage

30 20 47	63	101	115

Au 3ème passage l'élément (63) , 3ème plus grand élément du tableau est déplacé en N-3 (ici 3ème) position

4ème passage :
Echanges:

20↔30
pas d'echanges : 30 < 47

Résultat du 4ème passage

20 30	47	63	101	115

Au 4ème passage l'élément (47),4ème plus grand élément du tableau est déplacé en N-4 (ici 2ème) position

5ème passage :
Pas d'echanges : 20 < 30

Résultat du 5ème passage

20	30	47	63	101	115

Au 5ème le tableau est trié et l'algirithme s'arrête et on s'aperçoit qu'il y a N-1 (5 passages)

Code Source de tri à bulle en Langage C

```
#include <iostream>
```

```
void BubbleSort(int *a ,int N,long long  *echanges,long long*comparaison)
            {bool condition =false;
                  int r = N-1;
                  while(!condition){
                        condition = true;
                        for (int i=1;i<=r;i++){
                              (*comparaison)++;
                               if(a[i]<a[i-1]){
                               swap(a,i,i-1);
                               (*echanges)++;
                               condition= false;
                              }
                        }
                        r = r-1;
                  }
}
```

2.1.2 Analyse et Performance Théoriques du Tri à bulles

Nombre de comparaisons :

Le nombre de comparaisons maximal (MaxC) :
Si le tableau est trié en ordre inverse, l'algorithme implémenté ci-dessus fait N-1 comparaisons entre toutes les paires d'éléments du tableau pour déplacer le premier plus grand élément à la bonne position, N-2 comparaisons pour déplacer le deuxième plus grand élément. Alors il faut N-i comparaisons pour déplacer le ième élément plus grand à la bonne place. Remarquons ici qu'on ne fait aucune comparaison pour avoir le plus petit (Nème plus grand) élément du tableau à la bonne position.
Ainsi on calcule le nombre total de comparaisons en sommant de i=1(indice du N-1ème plus grand élément) à N-1 (indice du Nième plus grand élément) les N-i comparaisons. D'où là formule suivante :

$$\mathbf{MaxC(N) = \sum_{i=1}^{N-1}(N-i) = N(N-1)/2}$$

Le nombre de comparaisons minimal (MinC) :
Si le tableau est déjà trié, dans l'algorithme BubbleSort implémenté ci-dessus la boucle externe « while» est exécutée une fois et la boucle interne for est exécutée N-1 fois. Par conséquent on a :

MinC(N)=N-1

Les complexités en nombre de comparaisons pour le tri à bulle sont donc :

MaxC(N)= N(N-1)/2)=O(N^2) MoyC(N)=O(N^2) MinC(N) = N-1=O(N)

Nombre de Mouvements :

Le nombre d'échanges maximal (MaxE) :
Le nombre d'échanges peut être au maximal N-1 au premier parcours de la « while » de la méthode BubbleSort implémentée ci-dessus, N-2 au deuxième parcours, ainsi de suite jusqu'à 1 échange au N-1 parcours. C'est le cas si le tableau est trié en ordre décroissant. On a donc:

$$MaxE(N) = \sum_{i=1}^{N-1}(N-i) = N(N-1)/2 \qquad MaxM(N) = 3 * N(N-1)/2$$

La complexité au pire en nombre de mouvements du tri bulle est en **O(N^2) .** Notons que si le tableau est trié, on ne fait aucun mouvement.

MinM(N)=0

Calcul par dénombrements du nombre d'échanges moyen (tableau non trié).
Soit t un tableau de taille N et t' son miroir (t'[1]=t[N],t'[2]=t[N-1],….t'[N]=t[1])

Si on exécute l'algorithme sur t et t, chaque paire d'éléments est échangée, soit dans t, soit dans t', jamais dans les deux. Donc pour le tri de t et t' on a en tout autant d'échanges qu'il y a de paires d'éléments d'indices différents, soit N(N-1)/2 échanges, donc 3*N(N-1)/2 mouvements.
Considérons l'ensemble T de tous les tableaux de taille N, ne comportant pas d'éléments égaux. Supposons qu'ils sont tous équiprobables. D'après la définition de la complexité en moyenne, si coût E(t) est le nombre d'échanges effectués pour trier le tableau t, et si p(t) est la probabilité du tableau t ,on a :

$$MoyE(n) = \sum_{t \in T} p(t).coûtE(t)$$

On peut partitionner T en Tc, ensembles des tableaux de taille N tels que t[1]< t[N] et Td , ensemble des tableaux de taille N tels que t[1]>t[N] . Si t € Tc, alors t' $\in$ Td et réciproquement. D'où :

$$MoyE(N) = \sum_{t \in Tc} p(t).coûtE(t) + \sum_{t' \in Td} p(t').coûtE(t')$$

Tous les tableaux étant équiprobable, posons pour tout t, p(t) =p'. On obtient :

$$MoyE(n) = p'*(\sum_{t \in Tc} coûtE(t) + \sum_{t' \in Td} coûtE(t'))$$

$$= p'*\sum_{t \in Tc} (coûtE(t) + coûtE(t'))$$

$$= p' *\sum N(N-1)/2 = p' *card(Tc)* N(N-1)/2$$

Il existe une bijection entre Tc et Td, ils sont donc de même cardinalité, d'où :

P' *card(Tc)=1/2

Par conséquent :

$$MoyE(N) = 1/2 * N(N-1)/2 = N(N-1)/4$$

Dans le cas où les éléments sont distincts, et où les listes ont de même probabilité, on a en nombre de mouvements :

$$MoyM(N) = 3* N(N-1)/4 = O(N^2)$$

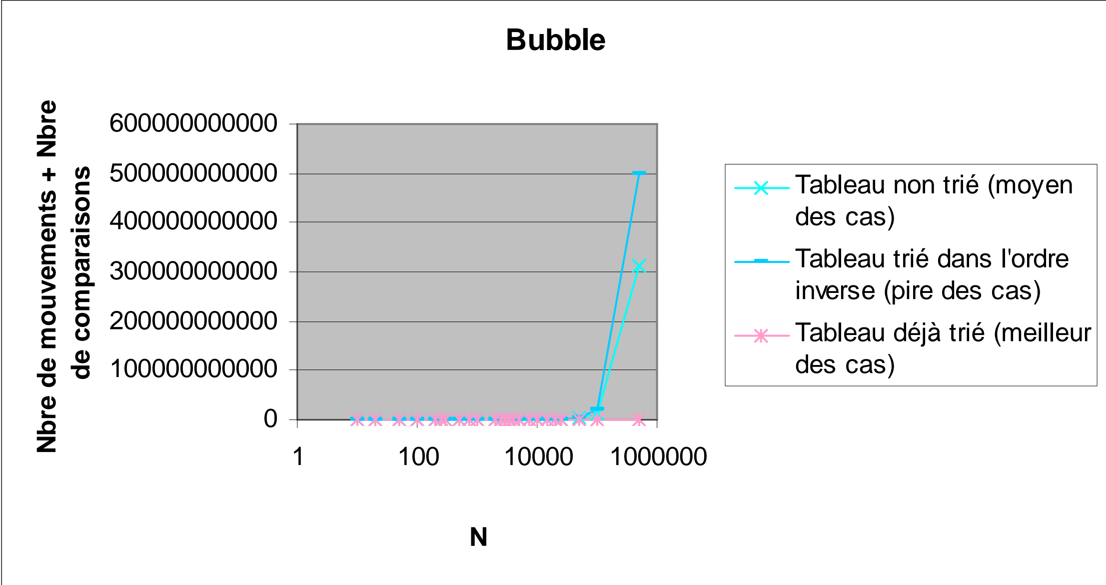
Par conséquent, la complexité en moyenne en nombre de mouvements du tri bulles est en O(N^2).

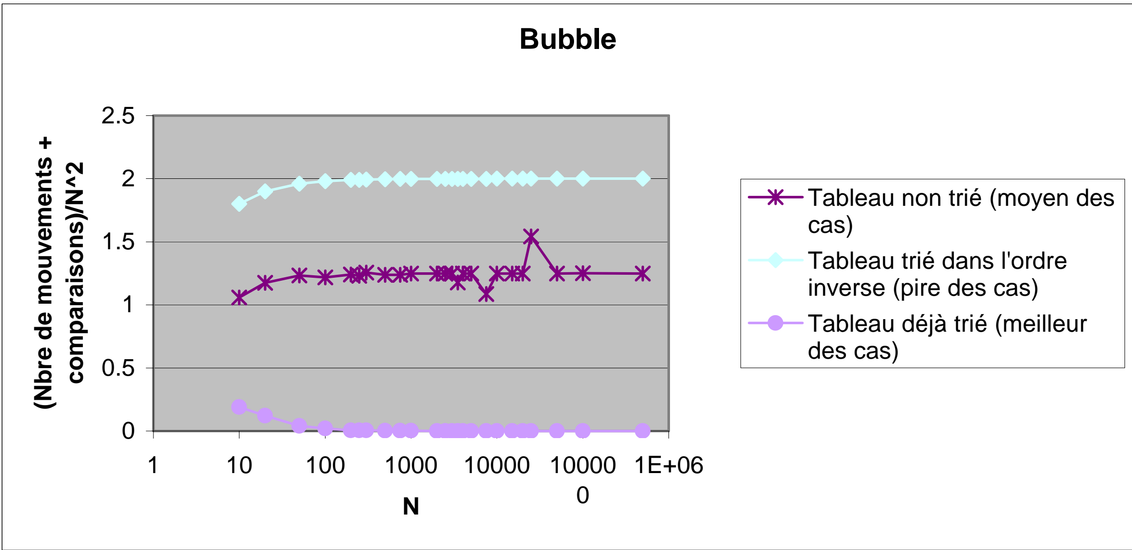
Pour le tri à bulle, le comportement pire des cas, moyen des cas et meilleur des cas s'obtiennent respectivement avec un tableau trié dans l'ordre inverse, non trié et presque trié

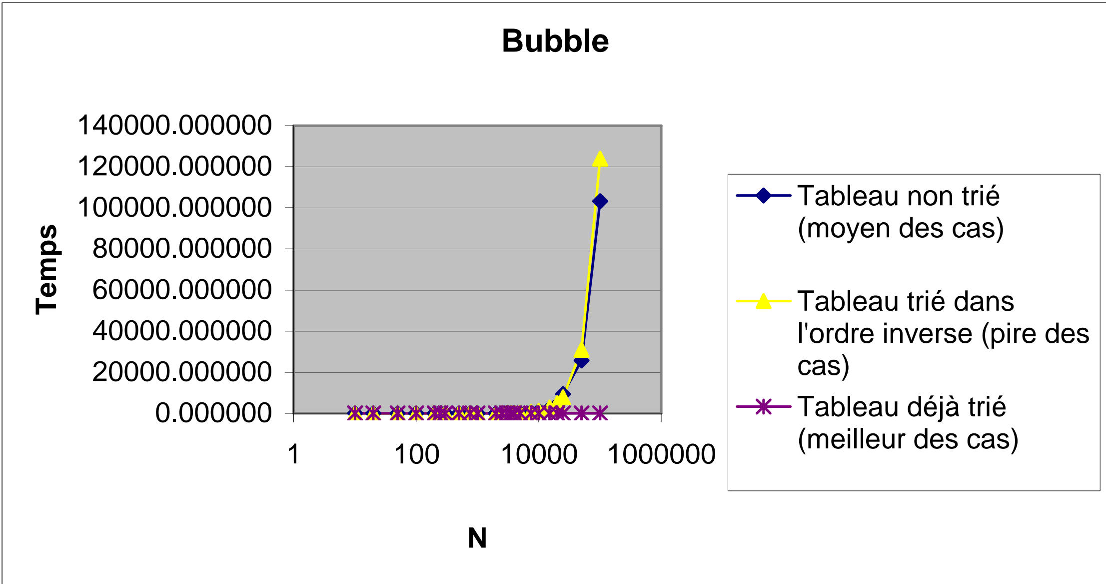
2.1.3 Résultats des Tests Effectués

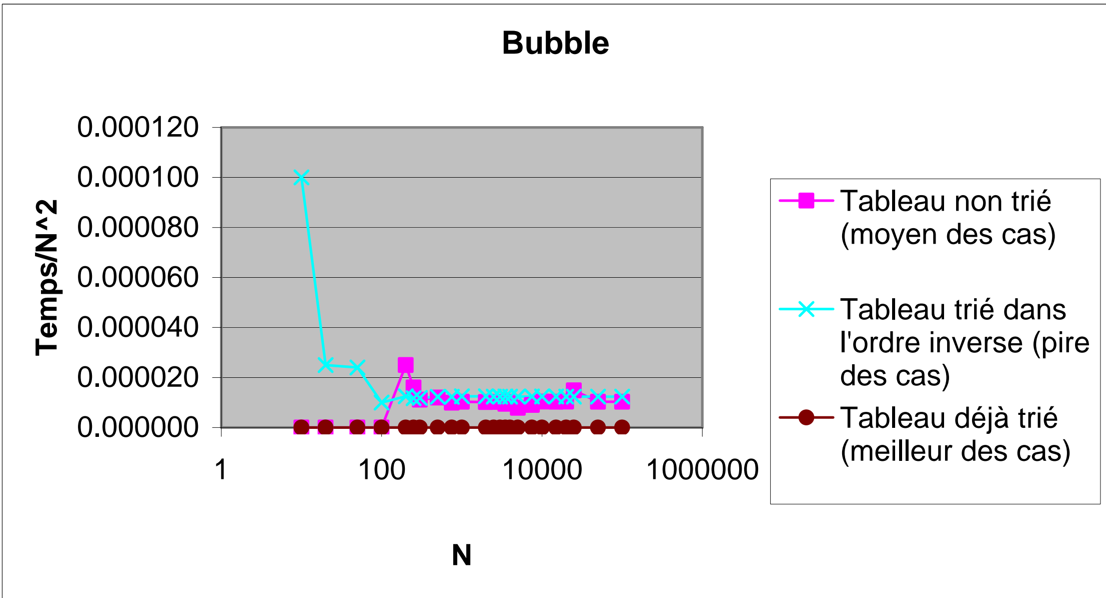
-Voir annexe1

2.1.4 Courbes Obtenus


Bubble
Nbre de mouvements + Nbre de comparaisons
600000000000
500000000000
400000000000
300000000000
200000000000
100000000000
0
1
100
10000
1000000
N
Tableau non trié (moyen des cas)
Tableau trié dans l'ordre inverse (pire des cas)
Tableau déjà trié (meilleur des cas)


Bubble
(Nbre de mouvements + comparaisons)/N^2
2.5
2
1.5
1
0.5
0
1
10
100
1000
10000
10000
0
1E+06
N
Tableau non trié (moyen des cas)
Tableau trié dans l'ordre inverse (pire des cas)
Tableau déjà trié (meilleur des cas)


Bubble
Temps
140000.000000
120000.000000
100000.000000
80000.000000
60000.000000
40000.000000
20000.000000
0.000000
1
100
10000
1000000
N
Tableau non trié (moyen des cas)
Tableau trié dans l'ordre inverse (pire des cas)
Tableau déjà trié (meilleur des cas)


Bubble
Temps/N^2
0.000120
0.000100
0.000080
0.000060
0.000040
0.000020
0.000000
1
100
10000
1000000
N
Tableau non trié (moyen des cas)
Tableau trié dans l'ordre inverse (pire des cas)
Tableau déjà trié (meilleur des cas)

2.1.5 Observations et Conclusions sur l'Algorithme du Tri à Bulles

Etant donné que sa complexité en nombre moyen et maximal de mouvements est assez élevée, le tri à bulles a peu d'intérêt en pratique.

Pour un tableau presque trié, théoriquement cette méthode semble intéressante et bonne car elle moins coûteuse en comparaisons. Mais si on regarde loin dans la réalité et dans les cas pratiques, on ignore ce bon comportement de la méthode de tri à bulle car on cherche toujours à mettre dans l'ordre une suite d'objets disposées en désordre.
Selon la théorie, la complexité du tri à bulles est **quadratique** : il trie un tableau en un temps $0(N$^$2)$, c'est-à-dire que le temps pour trier un tableau de taille N (N éléments) est proportionnel au carré de la taille. Ce résultat est parfaitement illustré et confirmé par les tests effectués ci-dessus.

Une piste d'amélioration du tri à bulles serait, lors du parcours de la boucle interne FOR, de vérifier que l'élément i se trouve à la bonne position (a[i]).

Dans les paragraphes suivants, on analysera d'autres algorithmes de tri qui tiennent compte de l'ordre des éléments dans le tableau initial.

2.2 Tri par Sélection (Selection Sort)

2.2.1 Méthode et Implémentation du Tri par Sélection

L'idée est de trier un tableau en déterminant son plus petit, son deuxième plus petit, troisième plus petit, etc. élément. C'est à dire trouver la position du plus petit élément dans le tableau et ensuite échanger a[0] et a[i1]. Ensuite, déterminer la position i2 de l'élément avec le plus petit des a[1],………..a[N-1] et échanger a[1] et a[i2]. On continue de cette manière jusqu'à ce que tous les éléments soient dans la position correcte.
Un avantage de tri par sélection est qu'il est progressif, car à l'étape i de l'algorithme, le tableau est trié de a[0] jusqu'à a[i-1]

Données :	115 101 30 63 20 47
Sélection 20	
Placement	20 | 101 30 63 115 47
Sélection 30	
Placement	20 30 | 101 63 115 47
Sélection 47	
Placement	20 30 47 | 63 115 101

Sélection 63 : Pas de changement sur le tableau, car 63 est lui-même le 4ième plus petit élément.
Placement

20 30 47 63	115 101

Sélection 101
Placement

20 30 47 63 101 115

Dans l'exemple de tri par sélection ci-dessus, les éléments placés dans le bon ordre se trouvent à gauche de la barre verticale

Commentaires : Comme on le voit dans l'exemple ci-dessus, le dernier élément (115) n'est pas sélectionné. En effet, comme cet algorithme se fait de façon progressive et le dernier élément du tableau étant le plus grand, il est déjà à la bonne position.
Si tel n'était pas le cas, il aurait été sélectionné et échangé avec d'autres éléments du tableau par conséquent il ne serait au dernier placement.

Code Source de tri Sélection en Langage C

```
#include <iostream>
#include <time.h>
#include "swap.cpp"
void selectionSort(int* a, int N, long long* echanges, long long * comparaisons)
{
for (int i = 0; i < N; i++)
        {
int min = a[i];
                int minIndex = i;
                for (int j = i; j < N; j++)
                {
                        (*comparaisons)++;
                        if (a[j] < min)
                        {
                                min = a[j];
                                minIndex = j;
                        }
                }
                if (i != minIndex)
                        (*echanges)++;
                swap(a,i,minIndex);
        }
}
```

2.2.2 Analyse et Performance Théoriques du Tri par Sélection

Nombre de comparaisons :
Pour tout tableau de taille N, on effectue exactement N-1 comparaisons pour trouver le 1er minimum, N-2 comparaisons pour trouver le $2^{ème}$ minimum, ainsi de suite. Plus précisément, on fait (N-1-i) comparaisons pour trouver le ième minimum du tableau.

Avec le tri par sélection, le nombre de comparaisons de clés ne dépendent pas de la nature du tableau (presque trié, non trié et trié dans l'ordre inverse). Ainsi on a:

$$MaxC\ N)=MinC\ (N)=MoyC(N)= \sum_{i=1}^{N-i}(N-1-i) = N(N-1)/2$$

Les complexités en nombre de comparaisons pour le tri par sélection sont donc :

$$MaxC(N)=MinC\ (N)=MoyC(N) =N(N-1)/2 = O(N^2)$$

Nombre de Mouvements :
Aussi pour les échanges, la complexité au pire des cas est égale à la complexité en moyenne. Puisqu'on fait au maximum **un seul échange** à chaque parcours de la boucle FOR interne de l'algorithme implémenté en paragraphe 2.21. Comme on a N éléments dans le tableau. On a donc :

MaxE(N)= N-1

MaxM(N)= 3*(N-1)

Avec le calcul **par dénombrement du nombre d'échanges moyen**, comme ça été fait pour le tri par bulle au paragraphe 2.12, on peut montrer que le nombre moyen d'échanges est :

MoyE(N)=N/N-1 + (N-1)/(N-2)+ ……. +2/1 = N- ln(N) + ө(1)

Le nombre d'échanges moyen doit être multiplié par trois pour obtenir le nombre de mouvements moyen (MoyM(N)).

Les complexités au pire et en moyenne en nombre de mouvements de la sélection sont en O(N).

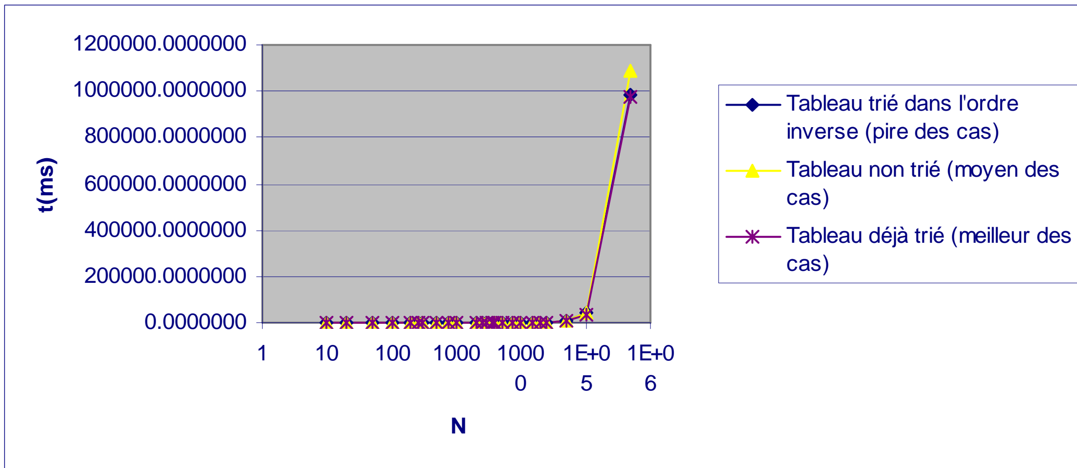
Pour un tableau presque trié, l'échange ne se fait jamais car ce n'est pas nécessaire : d'où

MinE(N) = 0

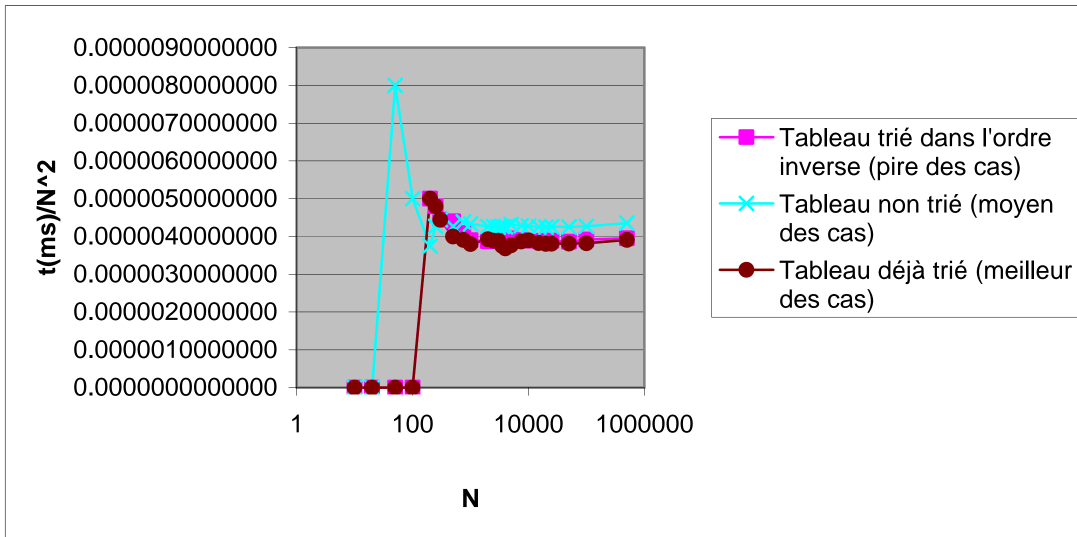
2.2.3 Résultats des Tests Effectués

-Voir annexe2

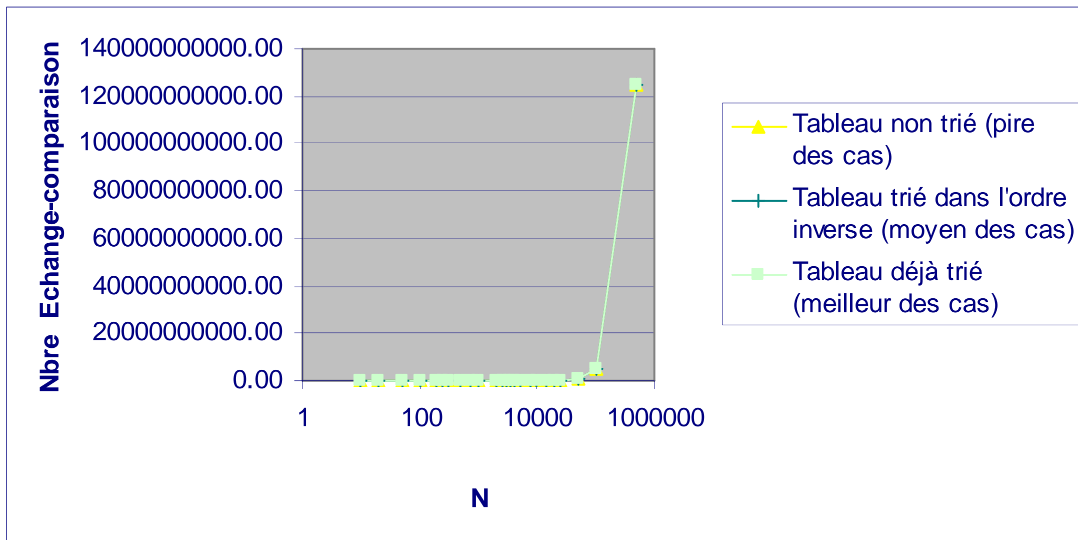
2.2.4 Courbes Obtenues

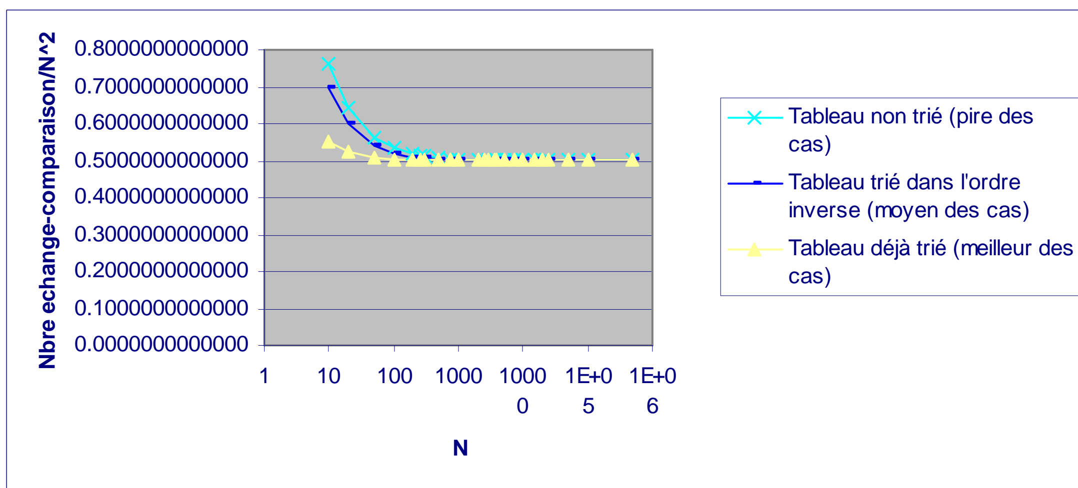


t(ms)
1200000.0000000
1000000.0000000
800000.0000000
600000.0000000
400000.0000000
200000.0000000
0.0000000
1
10
100
1000
1000
0
1E+0
5
1E+0
6
N
Tableau trié dans l'ordre inverse (pire des cas)
Tableau non trié (moyen des cas)
Tableau déjà trié (meilleur des cas)



t(ms)/N^2
0.0000090000000
0.0000080000000
0.0000070000000
0.0000060000000
0.0000050000000
0.0000040000000
0.0000030000000
0.0000020000000
0.0000010000000
0.0000000000000
1
100
10000
1000000
N
Tableau trié dans l'ordre inverse (pire des cas)
Tableau non trié (moyen des cas)
Tableau déjà trié (meilleur des cas)





2.2.5 Observations et Conclusions sur le Tri par Sélection

- En observant les courbes ci-dessus, on se rend compte que les résultats théoriques de complexité démontrés et confirmés par plusieurs chercheurs dans le domaine de l'algorithme reste aussi vrai en pratique, heureusement pour l'histoire des tris.

-Le premier graphe du haut illustrent les résultats du temps mis par le tri Sélection pour trier un tableau de taille N. Comme ça été démontré théoriquement ci-dessus, les graphes en temps ont une allure quadratique pour les types de tableaux utilisés dans ce projet (en N^2), ce qui est vrai seulement à partir d'une certaine valeur très grande, on voit sur le graphe que cette valeur correspond à N= 10^5. Par ailleurs on s'aperçoit facilement sur le graphe qu'il existe une constante k =temps/N^2 ≅ 39*10^-7 à partir de N=10^4.

Les deux derniers graphes illustrent bien que le nombre de comparaisons plus le nombre de mouvement suivent bien N^2 et la constante K≅0.5 à partir de N=10^4

b) Avantage par rapport à d'autres tris :

Aucun autre algorithme ne peut trier un tableau avec aussi peu de mouvements que le tri par Sélection. En effet comme le nombre d'échanges en pire des cas et en moyen des cas est linéaire, c'est-à-dire en O(N), il peut être intéressant pour des entrées qui nécessitent beaucoup plus d'échanges et très peu de comparaisons. Par exemple le tri par Sélection surpasse beaucoup de méthodes plus sophistiquées dans un type important d'application :

Dans la pratique, c'est la méthode à choisir pour trier un nombre important de petits fichiers (équivalent à un tableau de petite taille associé à de grandes clés.
-Désavantage par rapport à d'autres tris :
L'un des inconvénients de ce tri est que son temps d'exécution ne dépend pas de la nature de l'entrée à trier. IL passe quasiment le même temps pour trier un tableau déjà trié, un tableau généré de façon aléatoire et un tableau trié dans l'ordre inverse.

Remarquons aussi que la méthode par Sélection est peu pratique pour les tableaux partiellement trié et déjà trié contrairement à tri à bulle qui a une complexité O(n) pour le meilleur des cas. On verra au paragraphe suivant une méthode qui trie tout en accordant une attention particulière aux types de tableau partiellement trié.

2.3 Tri par Insertion (InsertionSort)

2.3.1 Méthode et Implémentation

L'idée est de trier successivement les premiers éléments du tableau. A la ième étape on insère le ième élément à son rang parmi les i-1 éléments précédents qui sont déjà triés entre eux. L'algorithme commence par s'exécuter à partir du 2ème élément du tableau. Cette méthode s'appelle aussi la méthode du joueur de cartes. Les étapes successives sont décrites ci-dessous.

Voici un exemple d'application de cette méthode sur un tableau de taille N=6

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
Données:	115	101	30	63	20	47
1ère insertion :	101	115	30	63	20	47
2ème insertion:	30	101	115	63	20	47
3ère insertion:	30	63	101	115	20	47
4re insertion:	20	30	63	101	115	47

5ème insertion:

20	30	47	63	101	105

Le tableau est trié **jusqu'à la barre verticale** ; l'élément à insérer est encerclé

```
#include <iostream>


void Insertion ( int *a, int N,long long *echanges ,long long * comparaisons,
                long long *mouvements)
{

  for (int i= 1;i<=N-1;i++)
  {
    int t= a[i];
    (*mouvements)++;
    int j=i-1;
    (*comparaisons)++;
    while(a[j]>t && j>=0)
    {
      a[j+1]=a[j];
      (*mouvements)++;
      j=j-1;
      (*comparaisons)++;
    }

    a[j+1]=t;
    (*mouvements)++;
  }
}
```

2.3.2 Analyse et Performance Théoriques du Tri par Insertion

Nombre de comparaisons

Pour insérer le ième élément, nous avons au moins 1 et au plus i-1 comparaisons à faire.
On a donc:

$$MaxC(N) = \sum_{i=2}^{N} (i-1) = O(N^2)$$

$MinC(N)=N-1$

On peut montrer comme ça été démontré pour le tri bulle que :

$MoyC(N)= O(N^2)$

Nombre de Mouvements

Contrairement aux autres algorithmes où on compte le nombre d'échanges, la méthode de tri par insertion fait uniquement des mouvements.

Cet algorithme est facile à analyser. Ainsi :
On compte i-1 mouvements dans le meilleur des cas (le comportement avec le tableau déjà trié). D'où :

$MinM(N)=0$

Avec un tableau trié dans l'ordre inverse, ce qui, pour ce tri, est le pire des cas :
La boucle interne WHILE qui se trouve dans l'implémentation ci-dessus fait au maximum N-i itérations si on est à la ième insertion, comme on commence le tri à partir de i=2 (deuxième élément du tableau.) en plus on a N éléments au total dans le tableau.
Le nombre de mouvements maximal se calcule comme suit :

$$MaxM(N)=\sum_{i=2}^{N}(i-2) = O(N^2)$$

La complexité du nombre de mouvements maximal est : $MoyM(N) == O(N^2)$

Avec le calcul **par dénombrement du nombre d'échanges moyen** , comme ça été fait pour le tri par bulle au paragraphe 2.12 , on peut montrer que le nombre moyens d'échanges est :

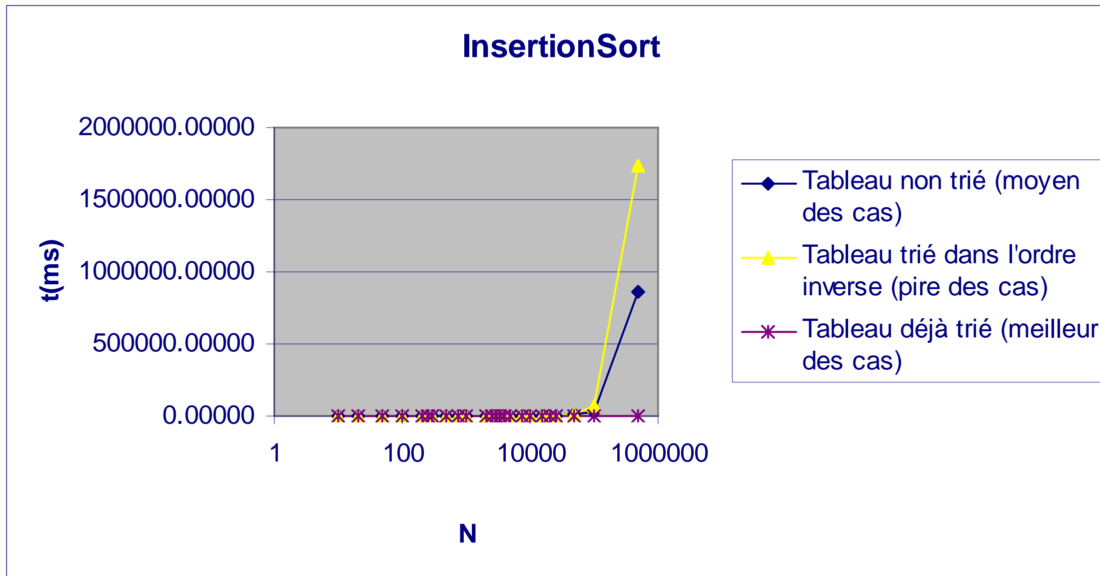
$MoyM(N)= N*(N+3)/4$

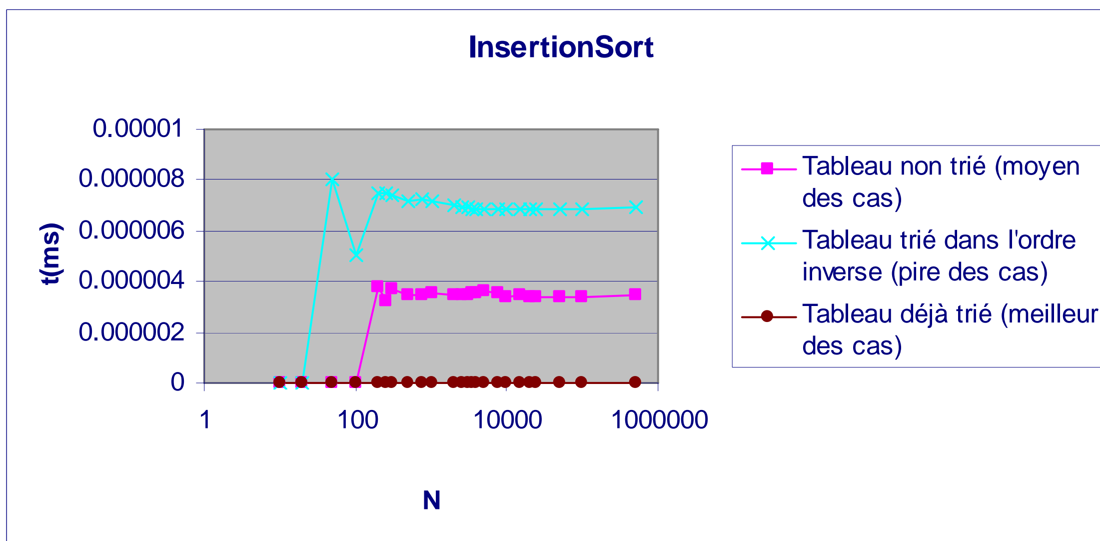
La complexité du nombre moyen de mouvements est : $MoyM(N) == O(N^2)$
La complexité en O(n^2) qui est prouver théoriquement seront justifiés par les résultats des tests et les graphes ci-dessous..

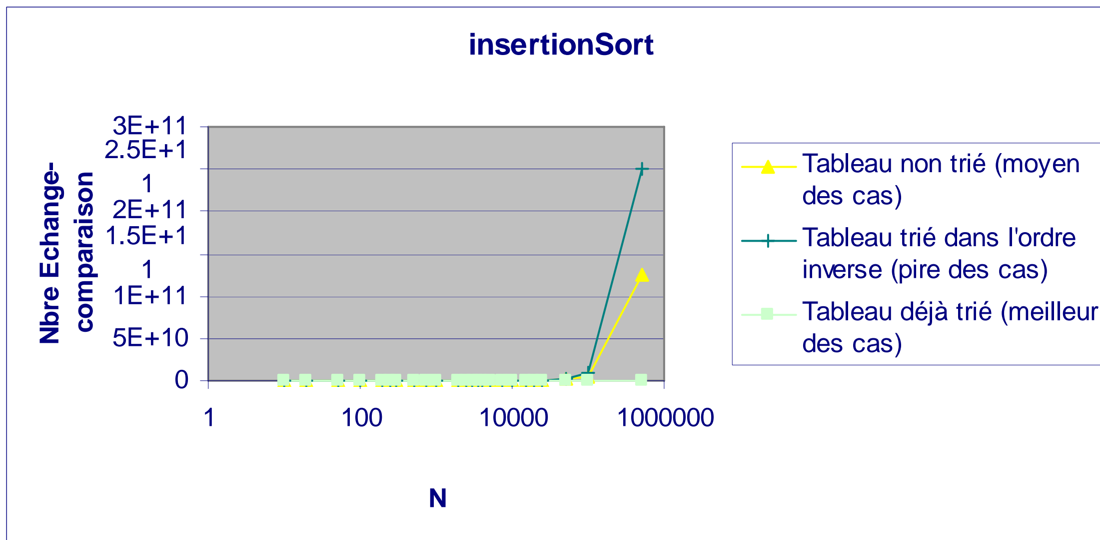
2.3.3 Résultats des Tests Effectués

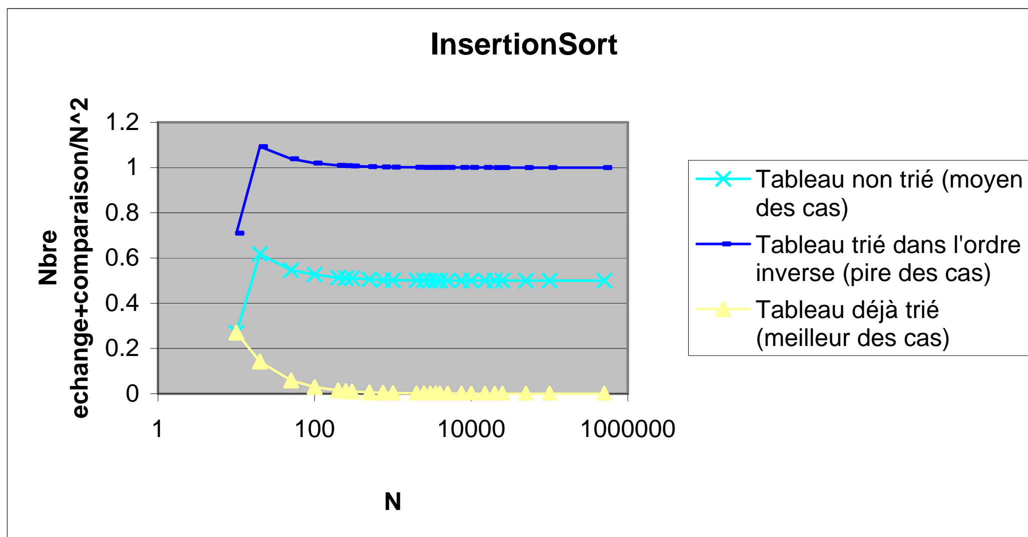
-Voir annexe 3

2.3.4 Courbes Obtenus









2.3.5 Observations et Conclusions sur l'Algorithme du Tri par Insertion

Avantages du tri Insertion par rapport à d'autres tris :
A la différence du tri par Sélection, le temps d'exécution du tri par Insertion dépend essentiellement de l'ordre initial des clés dans le tableau. Par exemple si le tableau est grand et que les clés sont déjà ordonnées (ou presque ordonnées), le tri par insertion est rapide, tandis que le tri par sélection est lent.

Désavantages du tri Insertion par rapport à d'autres tris :
Pour un tableau trié dans l'ordre inverse ; le fait que le tri par Sélection fait au maximum N-1 échanges (=3*(N-1) mouvements) fait que quand N devient très grand, par exemple on le voit sur les graphes que pour N= 5*10^5, le tri Par Sélection est plus rapide que le tri par insertion.

En conclusion, il est à noter qu'on pourrait améliorer le temps de calcul de tri par insertion pour améliorer son comportement dans le meilleur des cas en évitant les transferts inutiles pour un tableau déjà trié .
J'ai remarqué ceci lors des tests pratiques, j'ai réfléchi à ce problème et j'ai abouti à la conclusion que seul le tableau déjà trié en bénéficie, le cas moyen très peu car les nombres utilisés pour les tris dans ce projet sont générés aléatoirement. Ainsi on ne gagne pas beaucoup en optimisant de cette manière et surtout parce que c'est en faveur des tableaux déjà triés qui ne reflètent pas les problèmes de tri dans la réalité.

2.4 Tri par Shell (Shell Sort)

Cet algorithme de tri a été développé en 1950 par D. L. Shell

2.4.1 Méthode et Implémentation

Rappel : La lenteur du tri par insertion est due au fait que les seules permutations réalisées le sont avec des éléments adjacents. Les éléments peuvent donc parcourir tout le tableau mais en ne se déplaçant que d'une place à la fois. Par exemple, si l'élément de plus petite clé est à la fin du tableau, N étapes sont nécessaires pour le placer correctement.

Le tri Shell est une extension du tri par insertion qui entraîne un gain de temps en permutations d'éléments éventuellement très éloignés.
Il faut réorganiser le tableau de manière à obtenir un sous tableau ordonné en sélectionnant tous les h-ièmes éléments (à partir de n'importe quelle position initiale)
Un tel tableau est dit h-ordonné et il est constitué de h sous tableaux ordonnés indépendants, entrelacés. En h-ordonnant le tableau pour une grande valeur de h, on échange les éléments très distants dans le tableau, afin de faciliter la même opération pour les valeurs inférieures de h. L'utilisation d'une telle méthode avec une série décroissante de h terminée par 1 donne un tableau trié : C'est le principe du tri Shell.

Une manière d'implémenter le tri Shell serait d'utiliser, pour chaque valeur de h un tri par insertion de manière indépendante sur chacun des h tableaux. En dépit de l'apparente simplicité de ce processus, il est possible d'utiliser une approche encore plus simple précisément parce que les sous tableaux sont indépendants.

Comment décider de l'incrément à utiliser ?

En général, il est assez difficile de donner une réponse. Les propriétés de nombreuses séquences d'incréments ont été étudiées et beaucoup d'entre elles fonctionnent très bien dans la pratique, sans qu'il soit toutefois possible de trouver la meilleure d'entre elles. En pratique ,on utilise des séquences qui décroissent de manière de manière géométrique ,ce qui fait que le nombre d'incréments est fonction du logarithme de la taille du tableau .Par exemple ,si chaque incrément est environ la moitié du précédent, on a seulement besoin de 20 incréments pour trier un tableau d'un million d'éléments et si le rapport est de 1 à 4 seulement de 10 incréments pour le même tableau..
L'utilisation d'aussi peu d'incréments que possible est une caractéristique importante qui est facile à respecter. Il est également nécessaire de considérer les interactions

arithmétiques entre ces incréments comme la taille de leurs diviseurs communs et d'autres propriétés.
L'utilisation d'un bon incrément à pour effet pratique d'augmenter la vitesse d'environ 25%, mais ce problème est un véritable casse-tête qui donne un bon exemple de la complexité d'un algorithme apparemment simple.
La séquence d'incréments 1, 4, 13, 40, 121, 364, 1093, 3280, 984... utilisée dans l'implémentation du tri Shell dans ce projet, dont le ratio entre les incréments est d'environ un tiers , a été proposée par Knuth en 1969 .Celle -ci est facile à générer (on démarre avec 1 ,on calcule le prochain incrément en multipliant le précédent par trois et en ajoutant1) et donne un tri relativement efficace ,même pour des fichiers de taille plus petite.

Peut on encore faire mieux ?

Beaucoup d'autres séquences d'incréments donnent un tri plus efficace, mais il est difficile d'avoir un gain de plus de 20% du résultat obtenu par la séquence des incréments de Knuth, même pour les valeurs de N très grandes.
Notons que la séquence 4^(i+1)+ 3*2^i +1 pour i>0 , qui donne 1, 8 (pour i=0),23(pour i=1), 77(pour i=2),281,1076,4193,16577…, est vraiment la plus rapide dans le pire des cas.
Les séquences d'incréments encore meilleures peuvent très bien exister. Il faut juste se baser sur les séquences de bases et essayer de les améliorer. D'un autre côté, il existe de mauvaises séquences : par exemple la suite géométrique de raison 2 (1,2,4….) qui est la séquence originale proposée par Shell en 1959 donne de mauvaises performances parce que les éléments de position impaire ne sont pas comparés avec ceux de position paire jusqu'à la passe finale.

Code en C du tri Shell

```
#include "methods.h"
void Shellsort (int *a,int l,int r,long   *comparaisons,long
*mouvements){

    for (int h = 1; h <= int((r-l+1)/9) ; h = 3*h+1) ;
     printf("%d\n",h);
          for(;h>1;h /=3)
          printf("%d\n",h);
                 for(int i=l+h;i<=r;i++){

                        int j=i;
                        int v=a[i];
                        (*mouvements)++;
                        (*comparaisons)++;
                        while((j>=h+l)&&(v<a[j-h])){
                               a[j]=a[j-h];
                               (*mouvements)++;
                               j=j-h;
                               (*comparaisons)++;
                        }
                        a[j]=v;
                        (*mouvements)++;
                 }

 }
```

2.4.2 Analyse et performance Théoriques du Tri Shell

La description de l'efficacité du tri Shell est forcément imprécise parce que personne n'a jamais réussi à analyser cet algorithme. Cette lacune dans la connaissance rend difficile non seulement l'évaluation de différentes séquences d'incréments mais aussi la comparaison analytique de ce tri avec les autres méthodes. Bien que la modélisation formelle du temps d'exécution de ce tri Shell ne soit pas connue (il dépend en effet de la séquence utilisée). Knuth à trouver que la modélisation formelle donnée par N(log(N))^2 et N^1.25 donne de bons résultats.
Complexité de l'algorithme tri Shell implémenté dans ce projet est :
-Pour le nombre de comparaisons
MoyC(N)=O(N^3/2)
MaxC(N)=O(N^3/2)
MinC(N)=O(N^3/2)

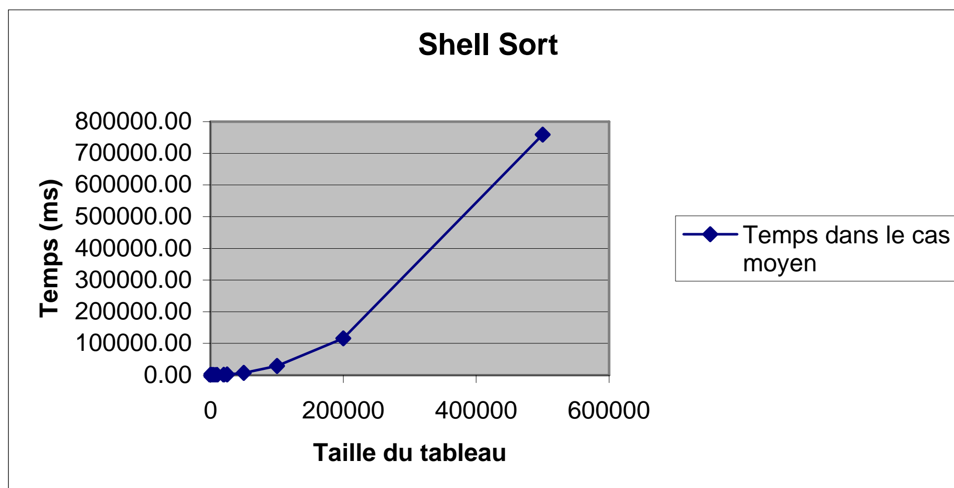
-Pour le nombre de mouvements

MoyM(N)=O(N^3/2)
MaxM(N)=O(N^3/2)
MinM(N)=O(N^3/2)

2.4.3 Résultats des Tests Effectués

-Voir annexe 4

2.4.4 Courbes Obtenues



2.4.5 Observations et Conclusions sur l'Algorithme du Tri Shell

La courbe ci-dessus suit bien l'allure de N^3/2. On peut donc dire que son temps de tri croit moins vite que les algorithmes de complexité O(N^2). On a également vu que le tri Shell peut devenir très mauvais si le choix de l'incrément n'est pas judicieux. L'algorithme est

très facile à coder, mais le choix de l'incrément optimal est plus complexe, ce qui rend difficile une généralisation de sa performance.

Pour conclure, évoquons certains aspects connus de l'analyse de ce tri. L'objectif ici est de montrer que même des algorithmes apparemment simples peuvent avoir des performances comparables à celles des tris complexes.

2.5 Comparaisons des Tris Elémentaires

Voici un tableau récapitulatif de leurs complexités

Algorithme	Nombre de comparaisons			Nombre de mouvements		
	Min	Moy	Max	Min	Moy	Max
Tri bulle	N-1	N(N-1)/2	N(N-1)/2	0	3*N(N_1)/4	N(N-3)/2
Tri Sélection	N(N-1)/2	N(N-1)/2	N(N-1)/2	0	N –ln(N) + ɵ(1)	3*(N-1)
Tri Insertion	N-1	N*(N+3)/4	N(N-3)/2	0	N*(N+3)/4	N(N-5)/2

<u>Tableau contenant le nombre de comparaisons et de mouvements des tris élémentaires</u>

Algorithme	Nombre de comparaisons			Nombre de mouvements		
	Min	Moy	Max		Moy	Max
Tri bulle	O(N)	O(N^2)	O(N^2)		O(N^2)	O(N^2)
Tri Sélection	O(N^2)	O(N^2)	O(N^2)		O(N)	O(N)
Tri Insertion	O(N)	O(N^2)	O(N^2)		O(N^2)	O(N^2)
Tri Shell			O(N(log(N))^2)			

<u>Tableau contenant les performances des tris élémentaires</u>

On sait que la plus part des tris élémentaires (Bulle, Insertion, Sélection) ont une complexité quadratique en O(N^2) excepté le tri par Shell.
Mais Le tri à Bulle est généralement plus lent que les deux autres (Insertion et Sélection) on peut améliorer cet algorithme de manière à éviter les parcours sans échanges.
Jusqu'ici, mon rapport a présenté quatre méthodes de tri simples ; Un élément de comparaison entre ces méthodes est que les méthodes de sélection donnent à l'étape i, le début du futur tableau à trier. On dit que ce tri est **progressif.** Cela permet de commencer éventuellement un traitement en parallèle sur ce tableau. Cela n'est pas possible si on utilise une méthode par Insertion.

Pour un tableau aléatoire (non trié): Le temps d'exécution de tri bulle est environ 2 fois celui de tri par Sélection dont le temps est environ une fois et demi celui de Insertion.
On peut donc conclure que le tri par insertion est le meilleur, suivi par le tri Sélection et en dernier le tri par Bulle qui est vraiment lent pour les entrées aléatoires.

Par contre pour les entrées spéciales, par exemple :

Pour un tableau presque trié : le tri bulle est 10 fois mieux que Sélection à partir de N =10000 mais Insertion reste toujours le meilleur car le temps total d'exécution est linéaire (O(N))

Pour un tableau trié dans l'ordre inverse: Pour les N très grand, le tri est sélection est le plus rapide, suivi du tri par bulles à cause du fait qu'il fait les échanges entre deux éléments adjacents tandis que le tri par Insertion fait des mouvements sur tout le tableau chaque fois si un élément n'est pas à la bonne place.

Parmi ces quatre tris élémentaires, le tri Shell est le choix idéal à faire pour beaucoup d'applications de tri parce qu'il a un temps d'exécution raisonnable quelque soit le type de tableau à trier.

Vu l'importance des tris dans plusieurs domaines, il existe des méthodes de tri plus performantes que les tris élémentaires, leur fonctionnement fera le sujet des paragraphes suivants.

3. TRIS SOPHISITIQUES OU COMPLEXES

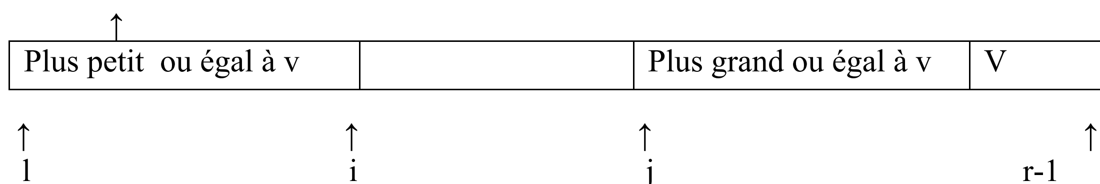
Quand les éléments du tableau associés aux clés sont grands, les tris élémentaires deviennent mauvais. Mais il existe des tris plus évolués qui présentent des complexités en O(NlogN) en moyenne et même dans le pire des cas en ce qui concerne le tri par tas (heapsort en anglais) : ce sont les ***tris sophistiqués***.

3.1 Tri Rapide (quiksort)

La version initiale de l'algorithme quicksort a été inventée en 1962 par C. A. R. Hoare. Il doit son nom au fait qu'il est l'un des algorithmes de tri connus les plus rapides. Il fonctionne bien pour de nombreux types de données en entrée.
L'algorithme du tri rapide a pour avantage de se faire « sur place » (il transforme une structure de données en utilisant une quantité de mémoire supplémentaire minimale et constante, les entrées sont écrasées par les sorties)

3.1.1 Méthode et Implémentation

On choisit un élément arbitraire b appelé le pivot parmi les éléments du tableau. On subdivise le tableau en deux sous tableaux, S1 contient les éléments du tableau qui sont plus petits ou égaux au pivot et dans S2, les éléments du tableau qui sont plus grands ou égaux au pivot. L'algorithme de tri rapide est ensuite appliqué récursivement à ces deux sous suites. Si un sous tableau contient un élément, on ne fait rien.
Un sous tableau du tableau original est déterminé de façon unique par deux indices i et j (voir l'implémentation ci-dessus). A chaque appel récursif de la méthode quicksort, nous appelons la routine en spécifiant l (de l'élément de plus à gauche) et r (de l'élément de plus à droit).
Comment créer les sous tableaux:
Pour le tri rapide implémenté dans ce projet, nous avons utilisé la stratégie générale suivante pour la **méthode de partition.** On choisit arbitrairement a[r] pour qu'il soit le pivot de la partition. Il ira donc à sa place finale. On parcourt ensuite le tableau en partant de la gauche jusqu'à ce qu'on trouve un élément plus grand que le pivot, de même on parcourt le tableau en partant de la droite jusqu'à ce qu'on trouve un élément plus petit que le pivot. Ces deux éléments n'étant pas à leur place, on les permute. En continuant de cette manière, on s'assure qu'aucun élément du tableau à gauche de l'indice de gauche n'est plus grand que le pivot et qu'aucun élément du tableau à droite de l'indice droit n'est plus petit que le pivot, comme décrit dans le diagramme suivant.



Lorsque les indices se croisent, on complète le processus de partition en échangeant a[r-1] avec l'élément le plus à gauche de la partie de la partie droite du sous tableau (c'est-à-dire l'élément dont le rang est égal à l'indice de gauche: **a[i] et a[r-1]).**

Code en Langage C de la méthode quicksort

```
#include "partition.cpp"
 void quickSort(int* a, int l, int r,long long  *echanges,  long long
*comparaisons){
        if (r <= l+1)

                return;
//if there are 2 elements
        if (r == l+2)
        {
                (*comparaisons)++;
                if (a[l] > a[r-1])
                {
                        swap(a,l,r-1);
                        (*echanges)++;

        }

                return;
        }

        if (r >= l+3)
        int k= partition(a,l,r,echanges,comparaisons);
        quickSort(a,l, k,echanges,comparaisons);
        quickSort(a,k+1,r,echanges,comparaisons);

        }

}
```

Code en Langage C de la méthode partition

```
//#include "swap.cpp"
int partition(int* a, int l, int r , long long  *echanges,  long long
*comparaisons){
        int i = l;
        int j = r-1;
        int pivot = a[r-1];
        while (i < j)
        {
                (*comparaisons)++;
        while(a[j]>=pivot && j > l){
```

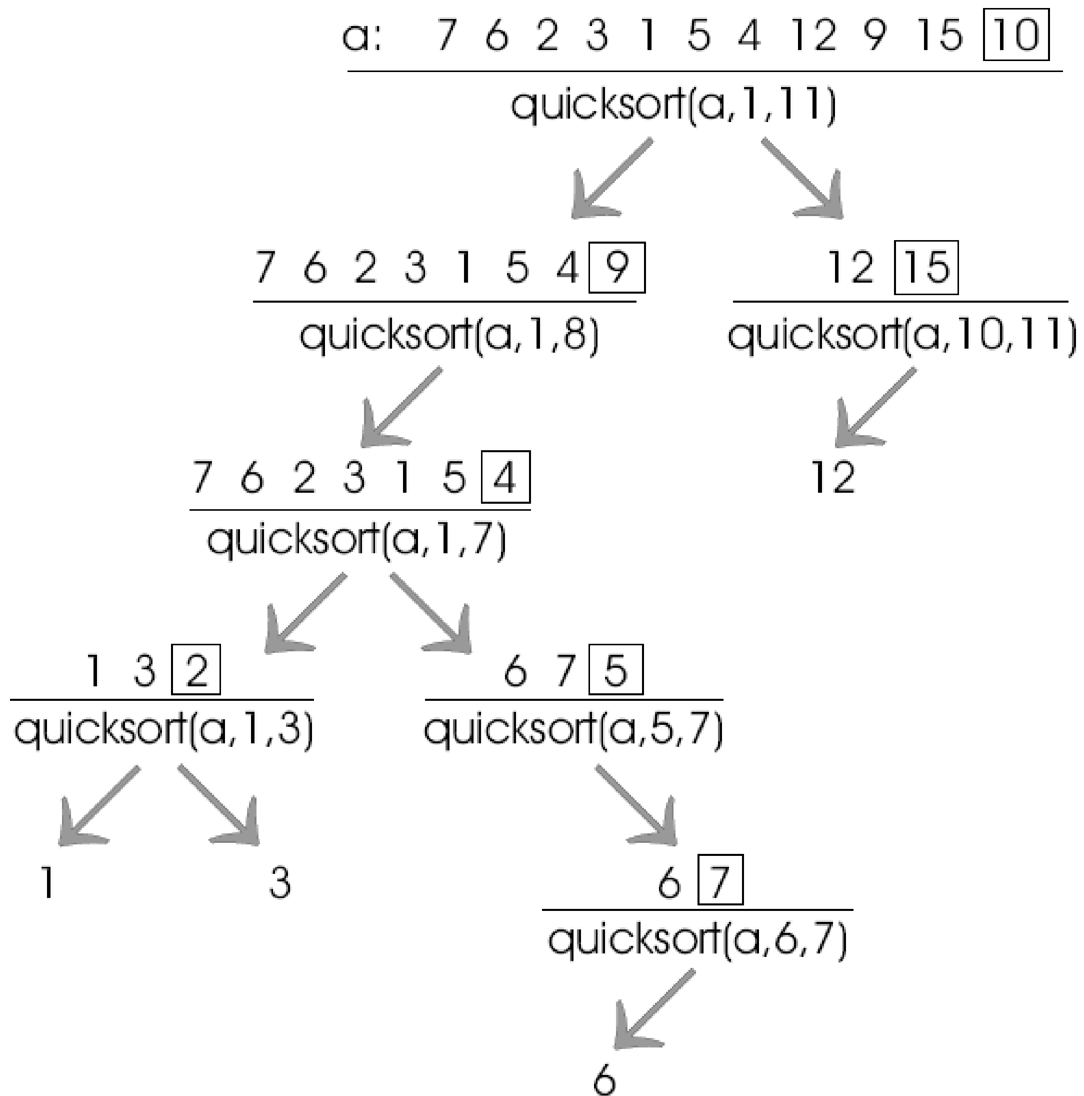
```
                        j--;
                        (*comparaisons)++;
                }
                (*comparaisons)++;

        while (a[i] <= pivot && i < r-1){

                        i++;
                        (*comparaisons)++;
                }

                if (i < j){
                        swap(a,j,i);

                        (*echanges)++;


                        }


        }

                swap(a,i,r-1);
                (*echanges)++;


        return i;
```

Notons que la méthode partition doit être implémentée avec prudence. Afin d'être sûr que le programme récursif se termine, on veille à ce que la méthode ne s'appelle pas elle-même lorsqu'elle traite des sous tableaux de taille 1 ou moins, et d'autre part à ce qu'elle ne s'appelle que lorsqu'elle traite des sous tableaux dont la taille est plus petit que celle du tableau en entrée. Ces stratégies peuvent sembler superflues, mais il est facile de donner une propriété qui peut aboutir à un dysfonctionnement spectaculaire. Une erreur courante dans l'implémentation du tri rapide est, par exemple, de ne pas s'assurer qu'un élément est toujours rangé à sa place, ce qui provoque une boucle récursive infinie quand le pivot est soit le plus petit élément du tableau, soit le plus grand.

L'efficacité du tri est liée à la manière dont la partition scinde le tableau, celle-ci étant elle-même dépendante de la valeur du pivot.

L'exemple ci-dessus montre que la partition découpe un tableau quelconque en deux sous tableaux quelconques plus petit, mais ce découpage peut être fait n'importe où dans le tableau initial. **On préfère choisir un élément qui sépare le tableau en son milieu,** mais on n'a pas l'information nécessaire pour faire cela. Si le tableau est aléatoire, choisir a[r-1] comme pivot est équivalent au choix d'un élément quelconque et donne, en moyenne, un découpage proche du milieu. C'est la stratégie qu'utilise le tri rapide implémenté dans ce projet.

Voici un exemple de tri rapide (quicksort)



3.2.2 Analyse et performance du Théoriques du Quicksort

Malgré ses nombreux atouts, le programme du tri rapide à tendance à être inefficace pour certains tableaux.

Nombre de comparaisons et Nombre de mouvements

Si on l'exécute, par exemple, sur un tableau trié dans l'ordre inverse ou presque trié, de taille N, toutes les partitions sont dégénérées et le programme s'appelle N fois, en enlevant seulement un élément à chaque appel (le pivot). Ainsi le nombre de comparaisons dans le pire des cas est:

MaxC(N)=N+ (N-1)+(N-2)+…….+2+1 =(N+1)N/2

La complexité en comparaisons et en mouvement dans le pire des cas est :

MaxC(N)= MaxM(N)=Ω(N^2)

Le cas le plus favorable au tri rapide est celui dans lequel l'étape de partition scinde le tableau exactement en deux. Cela permet au nombre de comparaisons effectuées de vérifier l'équation de récurrence propre au principe « diviser pour régner » :
MinC(N)=2*MinC(N/2) + N
Le terme 2*MinC(N/2) correspond au coût du tri des deux sous tableaux et N à celui de l'examen de chaque élément à cause de l'utilisation de l'indice de partition. Cette équation de récurrence admet comme solution :
MinC(N) ~N/log(N)
Le nombre de mouvements en meilleur des cas est 0 :
MinM(N)=0
Dans le meilleur des cas, On en déduit que la complexité est :

MinC(N)=O(NlogN)

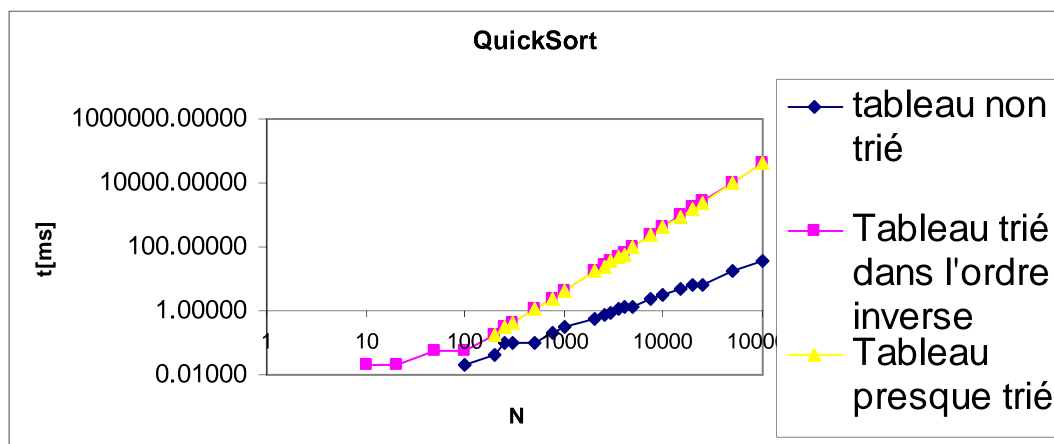
Dans le cas moyen, le nombre de comparaisons et mouvements effectués par le tri rapide est de l'ordre de 2Nln(N).

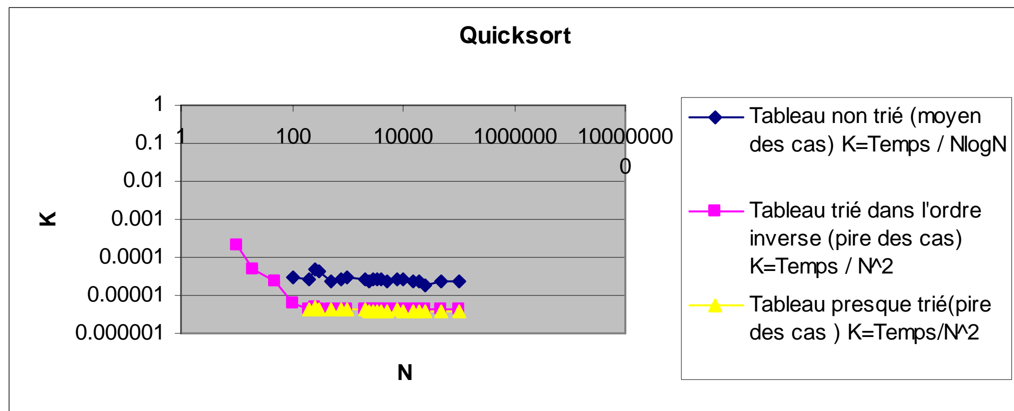
Il est à signaler que les complexités en nombre de mouvements sont du même ordre ou d'ordre inférieur, car on échange très peu dans l'algorithme mais la comparaison reste très coûteuse pour ce tri.
Bien que la réalité ne corresponde pas toujours à la théorie, il est vrai que les pivots partagent les tableaux au centre en moyenne. La prise en compte de la probabilité exacte de chaque de chaque position du pivot complique la relation de récurrence, mais le résultat final est similaire.

3.2.3 Résultats des Tests Effectués

-Voir annexe 5

3.1.4 Courbes Obtenus





3.1.5 Observations et Conclusions sur l'Algorithme du Quicksort

Comportement du tri rapide en face de différentes entrées possibles :
On peut aussi remarquer qu'en dessous d'une certaine taille des tableaux, le coût en temps et en place des appels récursifs devient significatif par rapport aux nombres de comparaisons. Au-dessous d'un certain seuil, il devient plus avantageux d'utiliser la version itérative d'une méthode de tri simple (tri par Sélection par exemple). Expérimentalement on s'aperçoit que la valeur de ce seuil est de l'ordre de 10.

Notons que le tri rapide est quasiment 3 fois plus rapides que le tri shell.

Contrairement au tri par tas, le tri rapide nécessite une pile auxiliaire de (mémoire) de taille maximum log (N).
La performance du tri rapide dépend du choix du pivot. Durant le projet, Nous avons essayé quelques stratégies de pivots tel que trois-médiane et neuf-médiane.
Pendant le test d'exécution, on s'est rendu compte que la stratégie des neufs n'a aucun intérêt, au lieu d'améliorer les performances du tri rapide, elle la détériore. Ceci se justifie par le calcul des neufs médianes qui est compliqué par conséquent la méthode prend du temps. Cependant la stratégie trois –médiane qu'on abordera au paragraphe suivant est d'une grande importance.

3.2 Tri Trois-Mediane : Amélioration du tri rapide par la stratégie du choix du pivot

Pour éviter les cas limites pour lesquels la complexité au pire du nombre de comparaisons est en O(N^2).On peut, par exemple chercher le trois- médianes du tableau (C'est-à-dire ,le premier élément du tableau, l'élément milieu et le dernier élément .On tri ces trois éléments et on prend comme pivot l'élément milieu des trois.

3.2.1 Méthode et Implémentation

Ce tri rapide amélioré à la même implémentation que quicksort, sauf que le pivot ici est trois-mediane. Le principe de diviser le tableau en 2 sous tableaux à chaque étape reste le même.
Les expériences que nous avons faites sur différentes entrées possibles montrent que la stratégie trois-mediane et la limite sur les petits sous tableaux réduisent encore plus le temps d'exécution du tri rapide.

Code en langage C de la méthode partition

```
#include "getmedian.cpp"
int  partition(int* a, int l, int r , long long  *echanges,  long long  *comparaisons){

                    int p = r-1;
                    int q = l;
                    int pivot;
                    int medIndex = getMedian(a[l], a[(r-1+l)/2], a[r-1]);
                    if (medIndex == 1)
                        medIndex = l;
                        if (medIndex == 2)
                        medIndex = (l+r-1)/2;
                        if (medIndex == 3)
                         medIndex = r-1;
                        swap(a,medIndex,r-1);
                    (*echanges)++;
                    pivot = a[r-1];

                    while (q < p)
                    {

                     //(*comparaisons)++;
                    while (a[p] >= pivot && p > l){
                              p--;
                        (*comparaisons)++;
                        }

                    //(*comparaisons)++;

                    while (a[q] <= pivot && q < r-1){
                              q++;
                              (*comparaisons)++;
                        }

                              if (q < p){
                              swap(a,p,q);

                              (*echanges)++;
```

```
                        }

                }
                swap(a,q,r-1);
               (*echanges)++;
                return q;
}
```

Code en Langage C de la méthode partition

```
#include "partition.cpp"
void quickSort3Median(int* a, int l, int r ,long long  *echanges, long long  *comparaisons)
{

  if (r <= l+1)

                return;



        //if there are 2 elements
        if (r == l+2)
        {
          //(*comparaisons)++;
                if (a[l] > a[r-1])
                {
                        swap(a,l,r-1);
                        (*echanges)++;
                }

                return;
        }

                if (r >= l+3)
        {

          int k= partition(a,l,r,echanges,comparaisons);

quickSort3Median(a,l,k,echanges,comparaisons);

quickSort3Median(a,k+1,r,echanges,comparaisons);
}
}
```

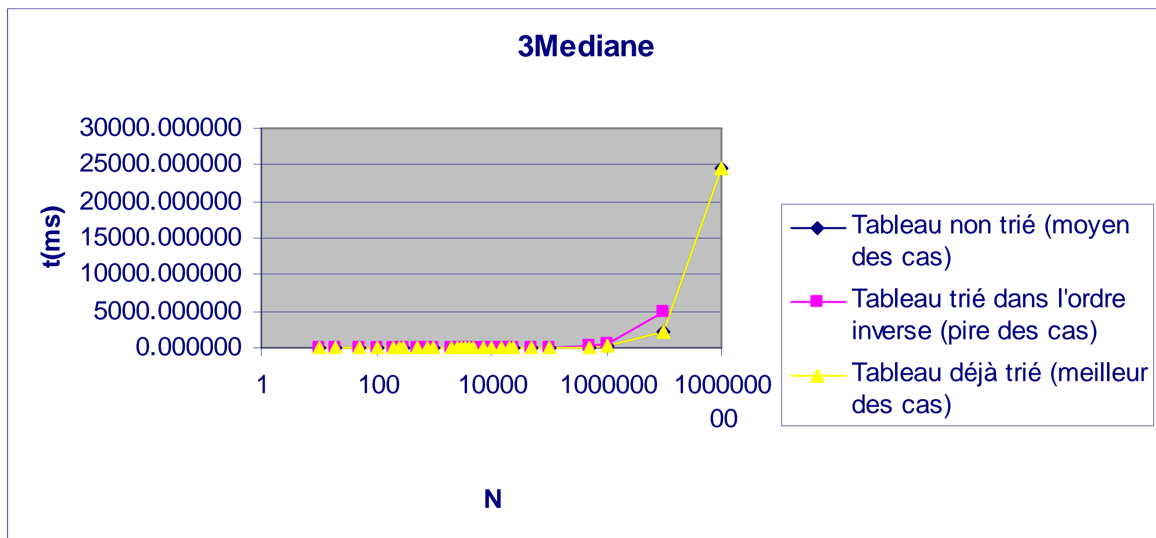
3.2.2 Analyse et performance Théoriques du Tri Trois-Mediane

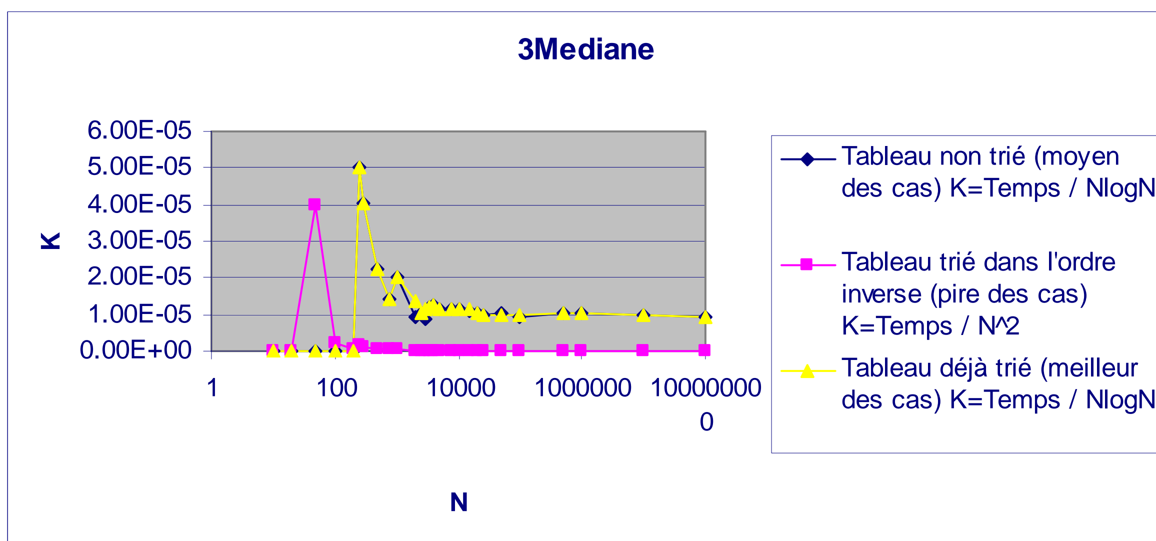
C'est quasiment, le même analyse que pour le tri rapide (quickSort). Ils ont pas conséquent les mêmes complexités. Certes on a remarqué lors de nos différents tests qu'il est plus rapide que quicksort car il fait moins de comparaisons et de mouvements. Mais ce n'est pas une raison pour qu'ils aient des complexités différentes.

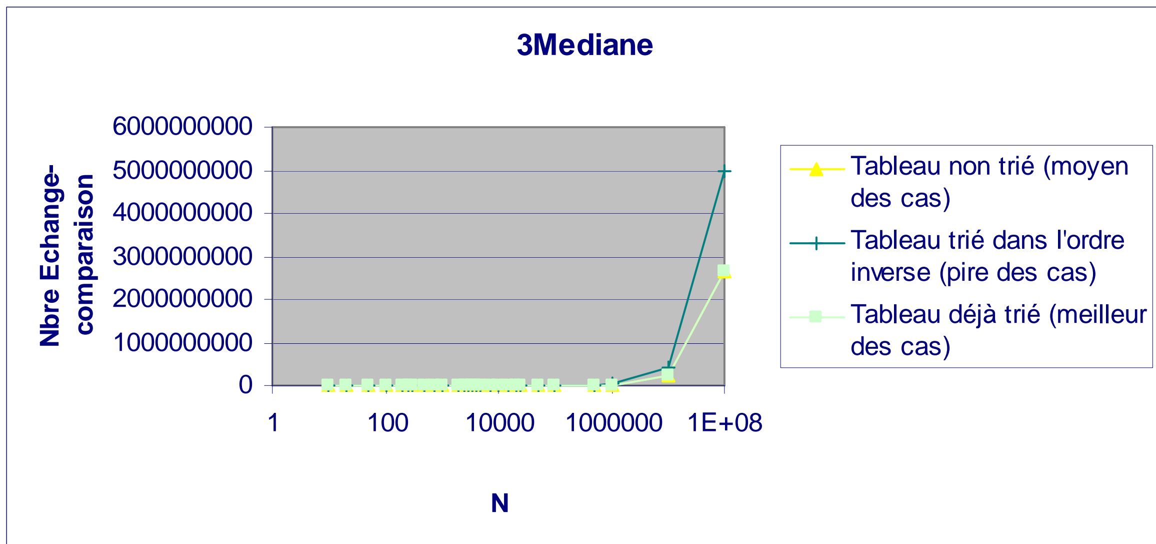
3.2.3 Résultats des Tests Effectués

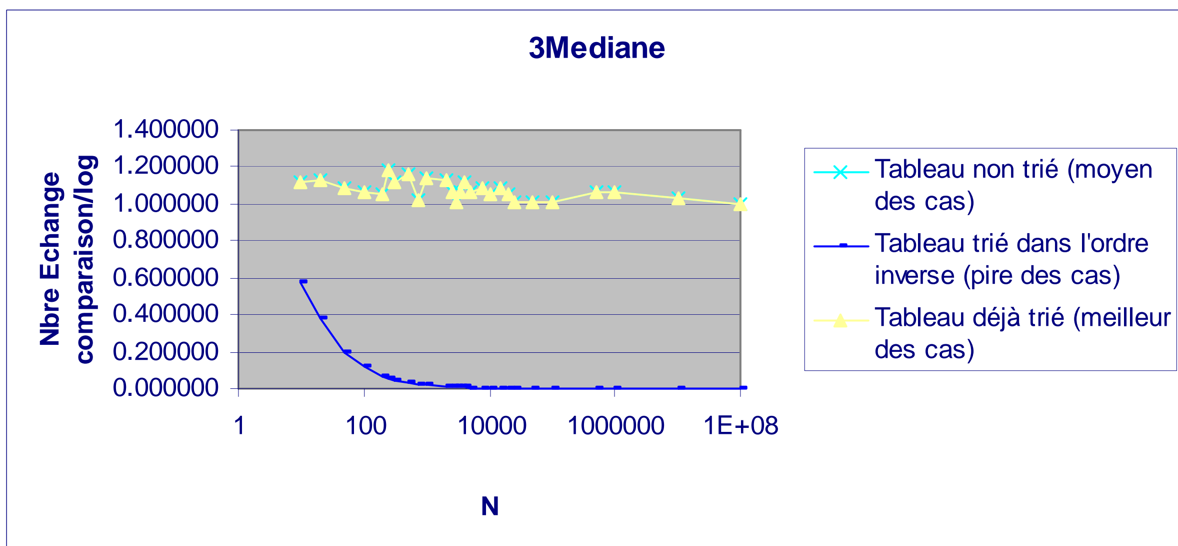
-Voir annexe 6

3.2.4 Courbes Obtenus









3.2.5 Observations et Conclusions sur l'Algorithme du Tri 3-mediane

Les tests effectués sur le tri 3-mediane nous a prouvé qu'il est deux fois plus rapide que quicksort. En plus il améliore nettement la dégénérescence de quicksort dans le pire des cas (avec un tableau presque trié ou trié dans l'ordre inverse)

3.3 Tri par Tas (Heapsort)

C'est une structure de données simples, appelée tas (heap en Anglais), qui permet de gérer efficacement les opérations de bases des files à priorité.
Les enregistrements d'un tas sont stockés dans un tableau dans lequel chaque clé est garantie supérieure à celle situées à deux autres positions spécifiques. Ces dernières sont à

leur tour chacune supérieure à deux autres clés, etc. Ce rangement est visualisable sous forme d'un arbre binaire ayant des arêtes (liens) menant de chaque nœud aux deux nœuds inférieurs.

3.3.1 Méthode et Implémentation

L'implémentation de cet algorithme utilise comme méthodes :

-Le **SiftDown** :
La procédure de sifts est une procédure efficace qui peut être utilisée pour insérer des éléments dans un heap, ou pour enlever des éléments. Il existe deux techniques de sifting:
le « sift up » et le « sift down »
Mais dans ce projet on utilise le sift down qui fonctionne de la manière suivante :
Etant donnée un heap dans sa représentation implicite dans la forme d'un tableau a avec N éléments, nous faisons un sift down de l'élément avec indice i comme suit :
Si le sommet i est une feuille (c'est-à-dire qu'il n'a pas de fils) ou s'il est plus grand que ses deux fils on ne fait rien. Sinon on échange le sommet i avec le plus grand de ses fils et on recommence ce procédé de façon récursive

-Le **BottomUpHeapCreate** :
Elle permet de créer un heap(tas) à partir du tableau de longueur N en utilisant la méthode SifDown

-HeapSort
Comme on a un tas, on est sûr que l'élément a[0] est le plus grand élément du tableau. Ainsi à chaque étape i>=1,nous remplaçons la racine de l'arbre courant (i.e.,a[0])avec a[N-1], et nous faisons un siftdown de l'élément a[0] dans l'arbre donné par les éléments a[0],……….,a[N-i-1].

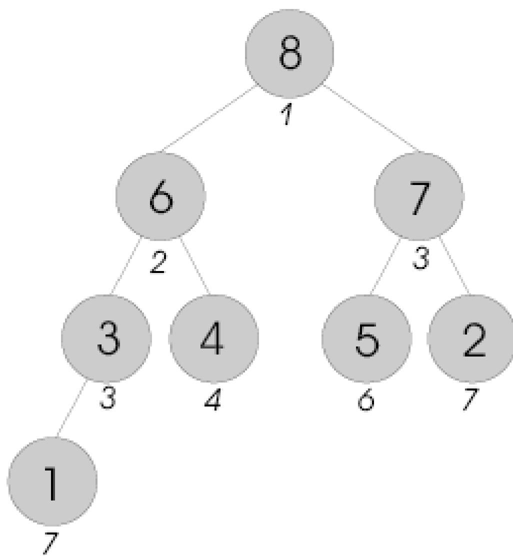
La méthode implique trois phases :

Un tableau a de longueur N peut être considéré comme un arbre binaire avec racine de la façon suivante (représentation implicite) : les sommets de l'arbre correspondent aux positions 0, 1,2,…………….N-1
La racine est à la position 0, et pour tout sommet i, ses fils sont en positions 2i+1 et 2i+2, à condition que ces valeurs soient plus petites que N.

- Construction du tas (heap) à partir de l'arbre binaire

  Ce tableau est un tas(heap) si :
  a[i]>=a[2i+1] et a[i]>=a[2i+2] avec 2i , 2i+1<N

Voici l'exemple de représentation d'un arbre binaire sous forme d'un tas (heap) en utilisant la méthode « **BottomUpHeapCreate »** implémentée ci-dessus.

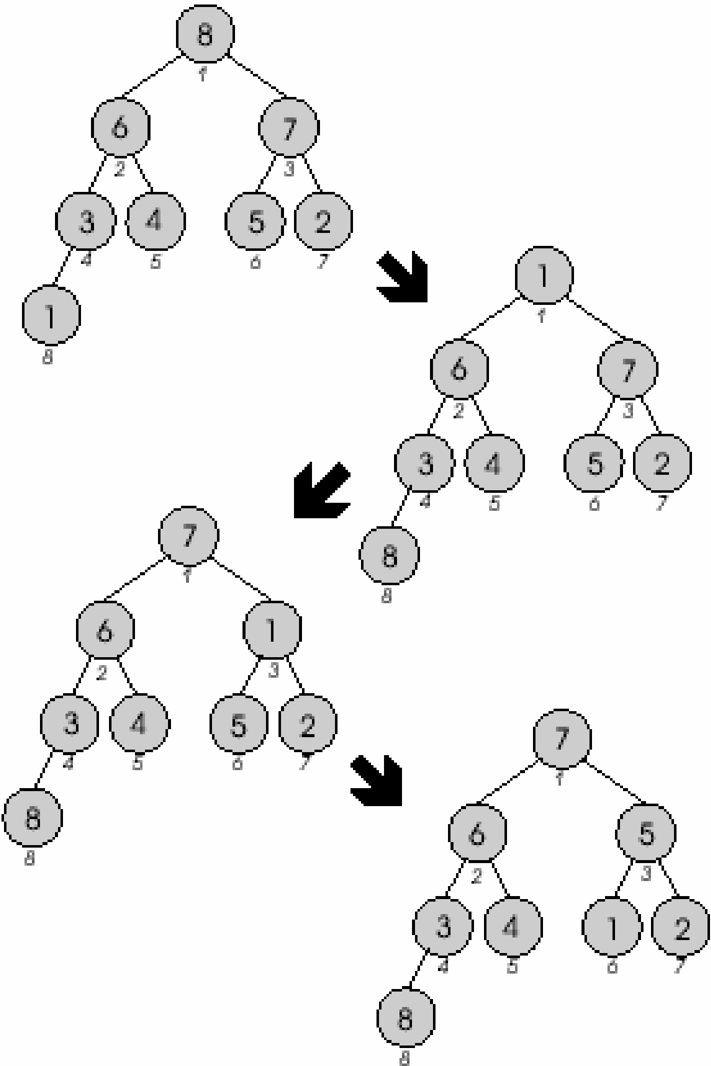


Le tableau [8,6,7,3,4,5,2,1] vérifie la propriété de heap, parce que

$$8 \geq 6, 8 \geq 7$$
$$6 \geq 3, 6 \geq 4; \qquad 7 \geq 5, 7 \geq 2$$
$$3 \geq 1.$$

- Une fois qu'on a un tas, on peut appliquer maintenant la méthode qui consiste qui consiste à échanger la racine avec le dernier élément du tas. Cette manière de faire est d'un grand intérêt car on sait que l'élément maximal du tableau se trouve à la racine. Dès lors on a plus un tas, on le reconstruit de nouveau en siftant depuis la racine jusqu'en l'avant dernier élément qu'on vient de déplacer en bas de l'arbre. (c'est-à-dire on descend l'élément de la racine tout en vérifiant à chaque noeud, la propriété d'un heap. On reprend le même principe jusqu'à ce qu'il reste un seul élément dans l'arbre. Ce n'est pas toujours facile de définir un méthode très clairement, on voit mieux sur l'exemple ci-dessous.

Voici une étape d'application de heapsort (tri par tas) décrit ci-dessus



Code en langage C de la méthode HeapSort

```
#include "SiftDown.cpp"
```

```
#include "TopDownHeapCreate.cpp"
void HeapSort(int *a, int N,long long *comparaisons,long long  *echanges) {
	TopDownHeapCreate(a,N,comparaisons,echanges);
	for (int i = N-1 ;i>=1 ; i--){
		swap(a,0,i,echanges);
		SiftDown(a,0,i-1,comparaisons,echanges);


	}
}
```

Code en langage C de la méthode BottomUpHeapCreate

```
void BottomUpHeapCreate(int*a,int N){
	int i;
	for (i =floor((N-1)/2);i<=0;i--)
		SiftDown(a,i,i);
}
```

Code en langage C de la méthode SiftDown

```
#include <math.h>

void SiftDown(int *a,int i, int ind_max,long long *comparaisons,long long  *echanges){
	int j=0;//fils
	bool Swapped ;
	Swapped = true;
	(*comparaisons)++;
	if(ind_max==0){
		return;
	 (*comparaisons)++;
	}
	 else

		if(ind_max==1){
			if (a[i]<a[ind_max])

			swap(a,i,ind_max,echanges);
		else return;


		}





		else{
```

```
		while (Swapped && i>=0 && i<=(int)floor(ind_max/2)){
			Swapped = false;

			if(a[2*i+1]>a[2*i+2]){
				(*comparaisons)++;
				j= 2*i+1;
			}
			else
				j= 2*i+2;

			if(2*i+2 >= ind_max){
				(*comparaisons)++;
				j=2*i+1;
			}

			if(2*i+1 >= ind_max){
				(*comparaisons)++;
				j=i;
			}
			if (a[i]<a[j]){

				(*comparaisons)++;
				swap(a,i,j,echanges);
				i= j;
				Swapped = true;
			}
		}
	}
}
```

3.3.2 Analyse et Performance Théoriques du Heapsort

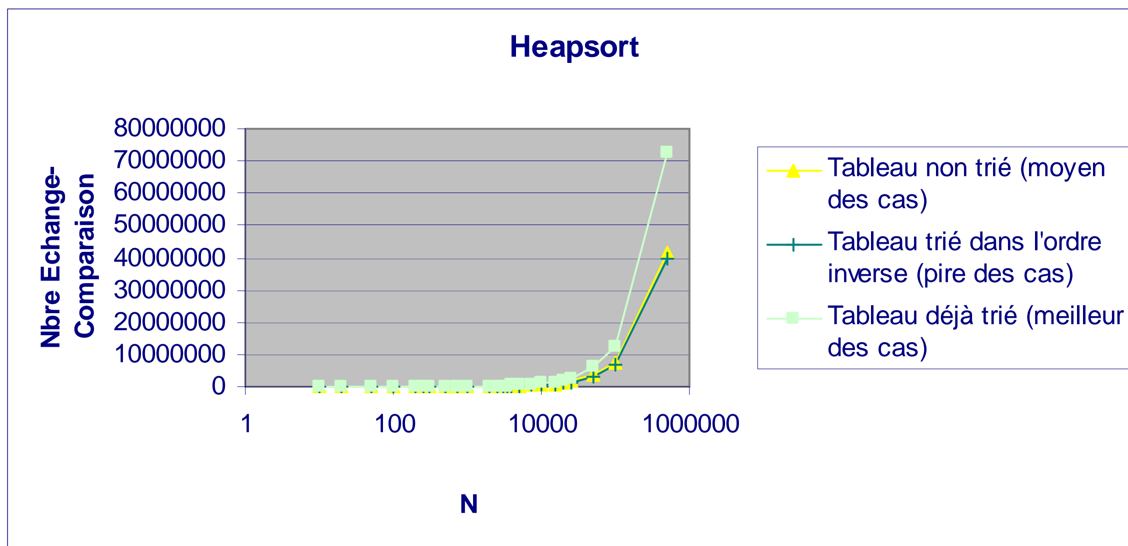
La performance de Heapsort est facile à analyser
Le temps d'exécution du tri par tas est indépendant de la nature du tableau à trier. Le heap s'exécute en O(N) et les N-1 sifts. Chaque opération de sifting lors de son appel utilise au plus O(log(N/i)) = O(log(i)) opérations. Donc le temps de parcours total est :

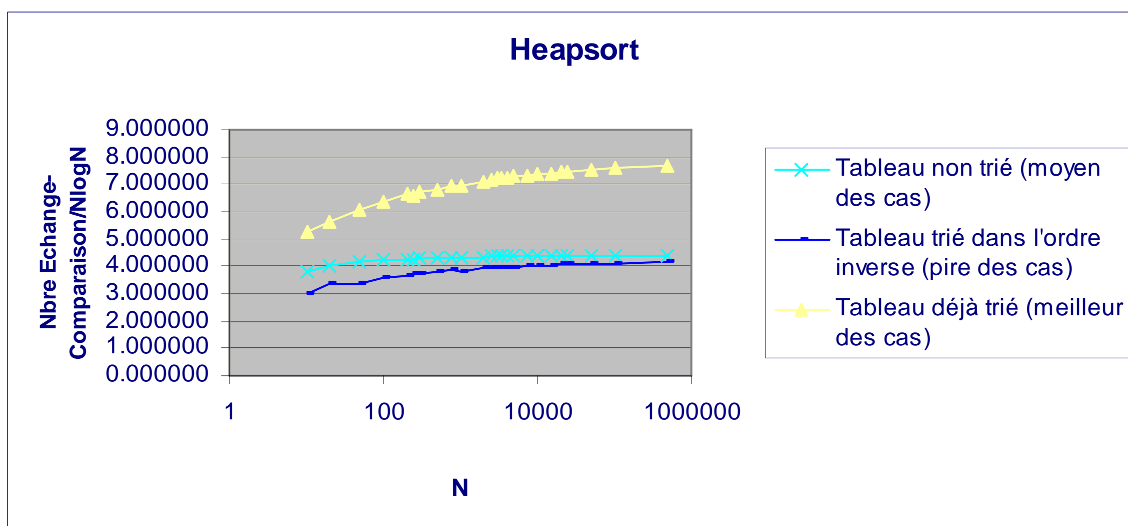
$$\sum_{i=2}^{N} O(\log(i)) = O(N) + O(\log N\,!) = O(N\log N)$$

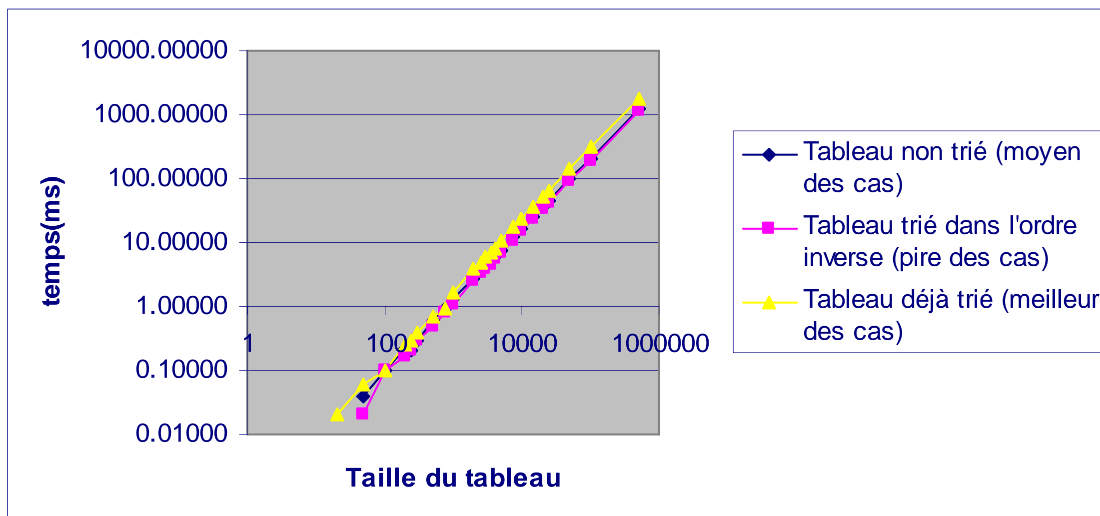
3.3.3 Résultats des Tests Effectués

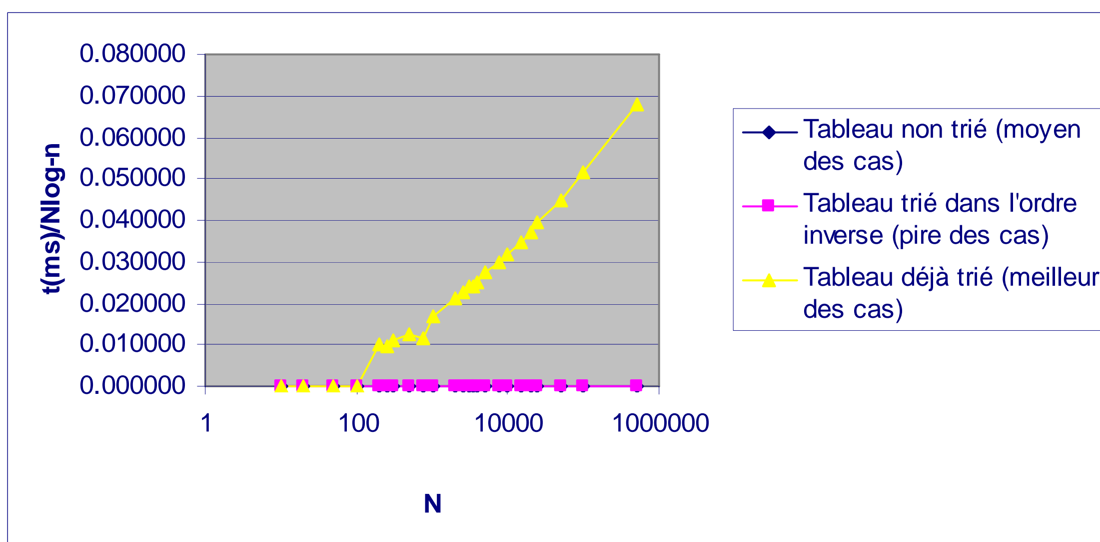
-Voir annexe 7

3.3.4 Courbes Obtenus





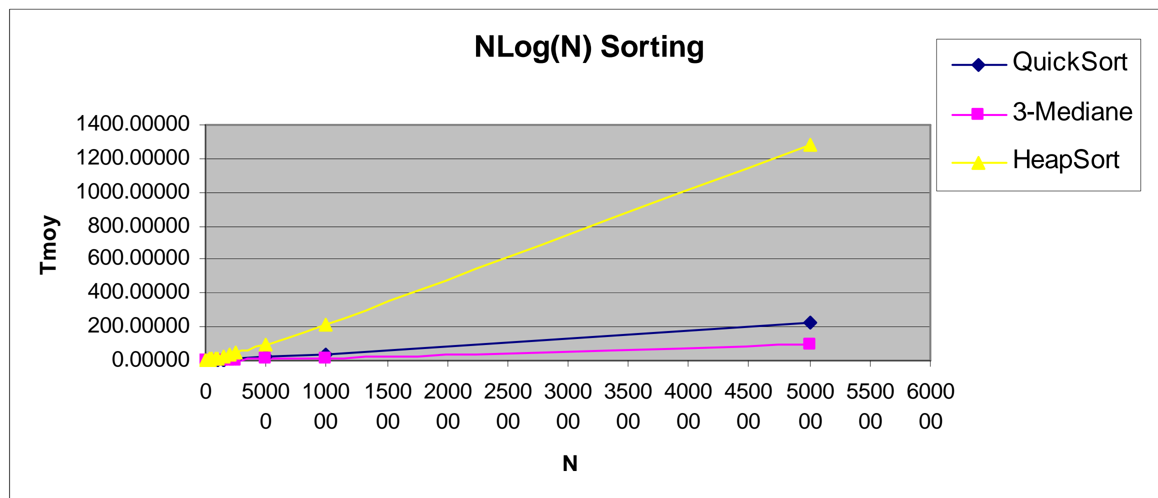




3.3.5 Observations et Conclusions sur l'Algorithme du Heapsort

Le tri par tas est une technique de tri qui est comparable au tri rapide (quicksort) pour le temps moyen. L'utilisation d'un tas permet de trier N éléments en temps de l'ordre de Nlog(N), indépendamment de la nature des données (ici des tableaux) O(NlogN), ceci sans mémoire auxiliaire car on peut utiliser le tableau à trier pour représenter le tas.
Les tests effectués sur le heapsort nous permet de conclure qu'il est devient mauvais sur des tableaux presque trié, ce se justifie par son fonctionnement. On pourrait donc penser à une amélioration de cet algorithme en tenant compte si certains sommets sont bien placés.

3.4 Comparaisons des Tris Sophistiqués



Tout ce qui suit se résume sur la courbe de NlogNSorting ci-dessus.
Bien entendu, il s'agit de savoir quand choisir le tri par tas, le tri rapide ou le tri 3-médiane pour telle ou telle application. Le choix entre heapsort et tri rapide (quicksort) revient à privilégier les performances dans le cas moyen ou dans le pire des cas. En général, rendre heapsort plus rapide que quicksort n'est pas gagné d'avance car la version améliorer de quicksort (3-médiane) relève ce défi.
Les résultats des tests empiriques en annexe montrent que heapsort est plus lent que quicksort dans le moyen des cas mais il est mieux de choisir heapsort pour les types de tableaux triés dans l'ordre inverse où quicksort dégénère en O(N^2).

4. CONCLUSION GENERALE

Durant le projet, différents algorithmes de tri ont été étudiés et comparés. La question qui se pose maintenant est de déterminer si l'on peut espérer trouver une méthode de tri « meilleure» que toutes les autres. Ce n'est pas aussi simple. Il faut évidemment préciser le (ou les) critère(s) selon le(s)quel(s) on compare les méthodes et pour quelles méthodes ce critère est significatif.
Pour obtenir des résultats intéressants, on va se restreindre à la classe Tc des algorithmes de tri qui opèrent uniquement par comparaisons deux à deux des clés des éléments à trier. De plus, on fait l'hypothèse que toutes les clés du tableau à trier sont différentes.
Dans ces conditions, comme le confirme les différents tests expérimentaux,on en déduit que :
La complexité, en nombre de comparaisons, aussi bien en moyenne qu'au pire, de tout algorithme de la classe Tc est d'ordre de grandeur supérieur ou égal à **Nlog(N) .**
On peut en déduire que les algorithmes de tri rapide(quicksort) et tri par tas(heapsort) sont optimaux en moyenne; le tri par tas est de plus optimal dans le cas le pire(une entrée de tableaux triés dans l'ordre inverse). Mais ceci est vrai seulement à partir d'une certaine

valeur de la taille du tableau. Par conséquent, il est important de savoir qu'on ne gagne pas grande chose à utiliser les tris compliqués (rapides) pour trier les tableaux qui ont moins d'une centaine d'éléments.
Pour analyser les algorithmes de tri, on s'est basé sur deux opérations fondamentales: les comparaisons et les mouvements. Le résultat énoncé ci-dessus porte seulement sur les comparaisons. Le nombre de mouvements ne doit pas être négligé pour autant : Si on considère par exemple le tri par Insertion, il est optimal dans le pire des cas en nombre de comparaisons mais le nombre de transfert est en O(N^2). Pratiquement le temps total d'exécution de l'algorithme est proportionnel à N^2.
Le tri par Shell requiert peu de code et se montre compétitif pour des tableaux de grandes tailles en face des tris performants (quicksort, tri par tas) qui sont deux fois plus rapides, excepté pour de grands N mais qui sont beaucoup plus complexes à implémenter.

En résumé si vous avez besoin d'une solution rapide pour un problème de tri et que vous ne voulez pas vous occuper de l'interfaçage d'un système de tri, vous pouvez utiliser le tri shell et reporter à plus tard le travail supplémentaire pour une méthode plus sophistiquée.

Pour mieux voir le comportement des divers algorithmes étudiés dans ce projet dans le pire des cas, moyen des cas et meilleur des cas, on a procédé à une représentation sous forme d'histogramme ci-dessous.

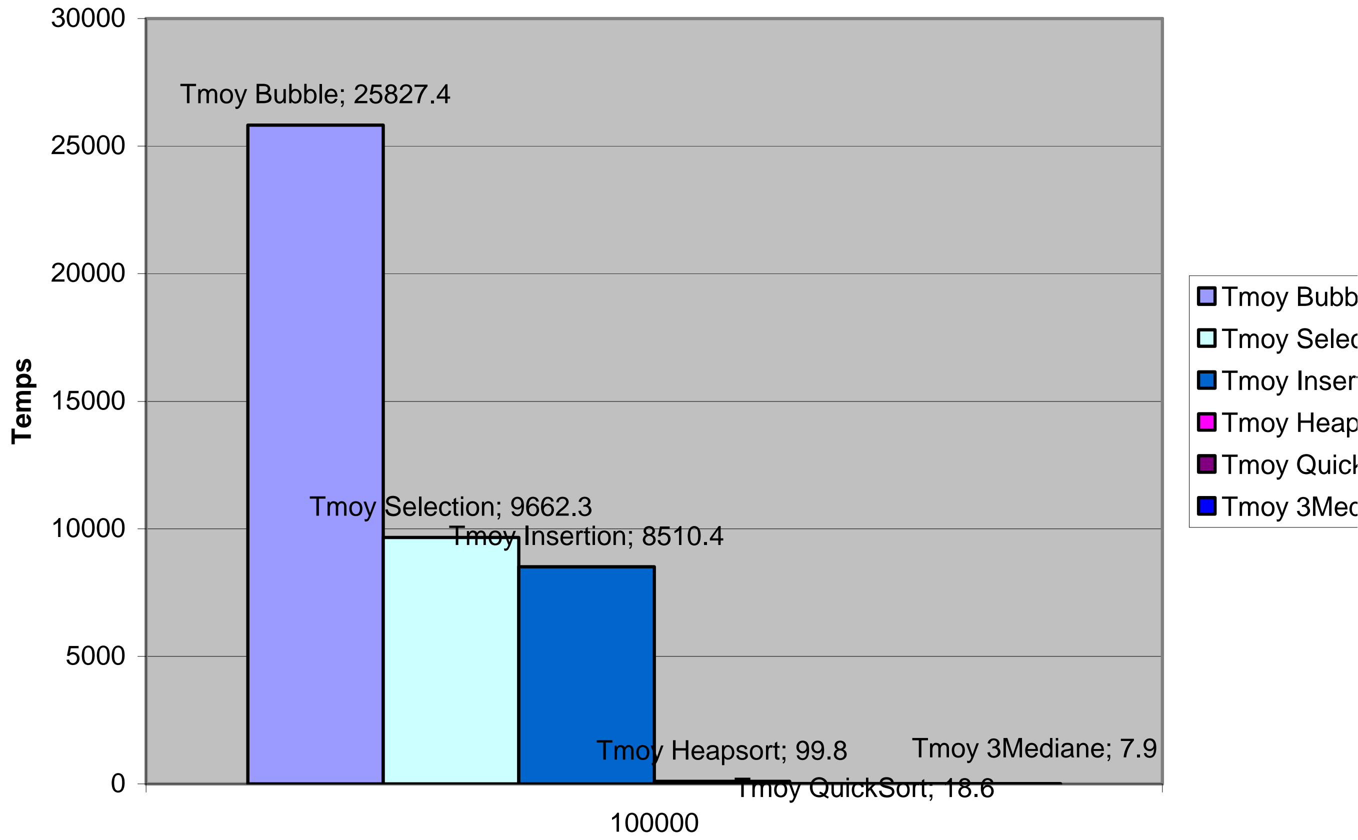


Tableau non trié pour N=100000
Temps
30000
25000
20000
15000
10000
5000
0
Tmoy Bubble; 25827.4
Tmoy Selection; 9662.3
Tmoy Insertion; 8510.4
Tmoy Heapsort; 99.8
Tmoy QuickSort; 18.6
Tmoy 3Mediane; 7.9
100000
Tmoy Bubb
Tmoy Selec
Tmoy Inser
Tmoy Heap
Tmoy Quick
Tmoy 3Med

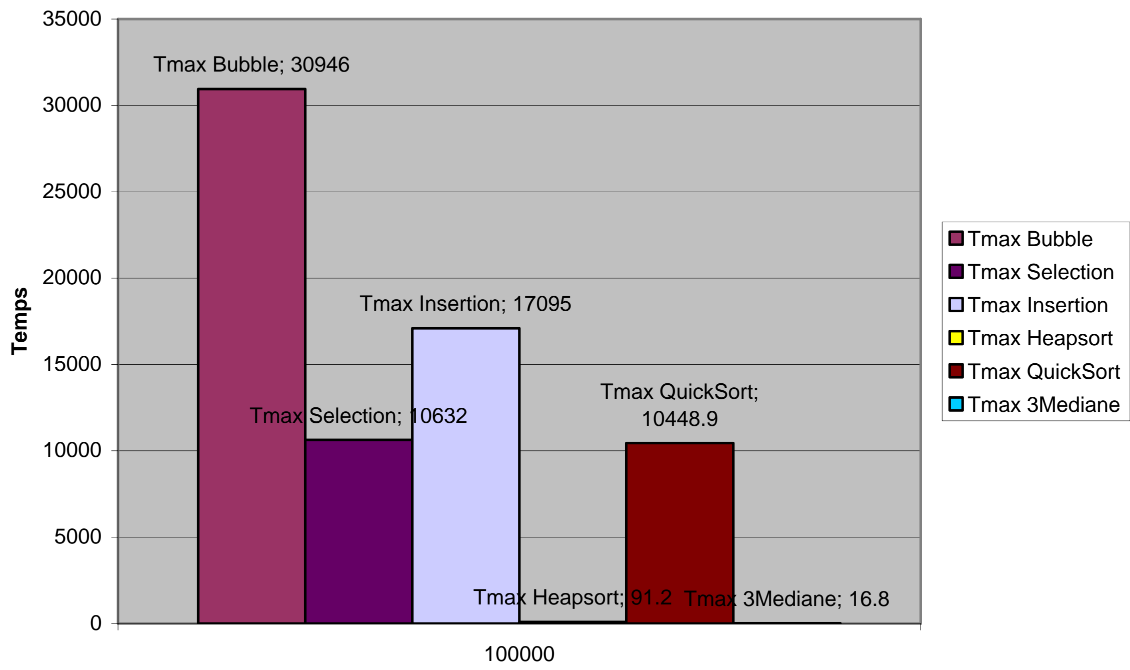


Tableau trié en sens inverse pour N=100000
Temps
35000
30000
25000
20000
15000
10000
5000
0
Tmax Bubble; 30946
Tmax Insertion; 17095
Tmax Selection; 10632
Tmax QuickSort; 10448.9
Tmax Heapsort; 91.2
Tmax 3Mediane; 16.8
100000
Tmax Bubble
Tmax Selection
Tmax Insertion
Tmax Heapsort
Tmax QuickSort
Tmax 3Mediane

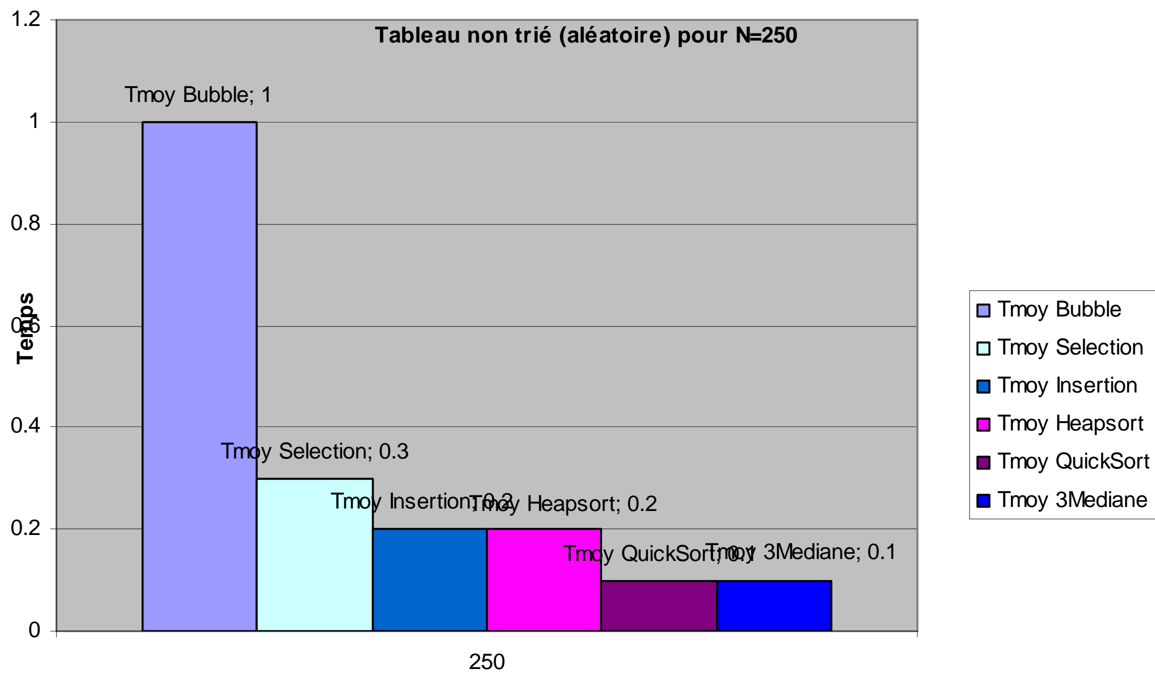


Tableau non trié (aléatoire) pour N=250
Temps
1.2
1
0.8
0.6
0.4
0.2
0
250
Tmoy Bubble; 1
Tmoy Selection; 0.3
Tmoy Insertion; 0.2
Tmoy Heapsort; 0.2
Tmoy QuickSort; 0.1
Tmoy 3Mediane; 0.1
Tmoy Bubble
Tmoy Selection
Tmoy Insertion
Tmoy Heapsort
Tmoy QuickSort
Tmoy 3Mediane

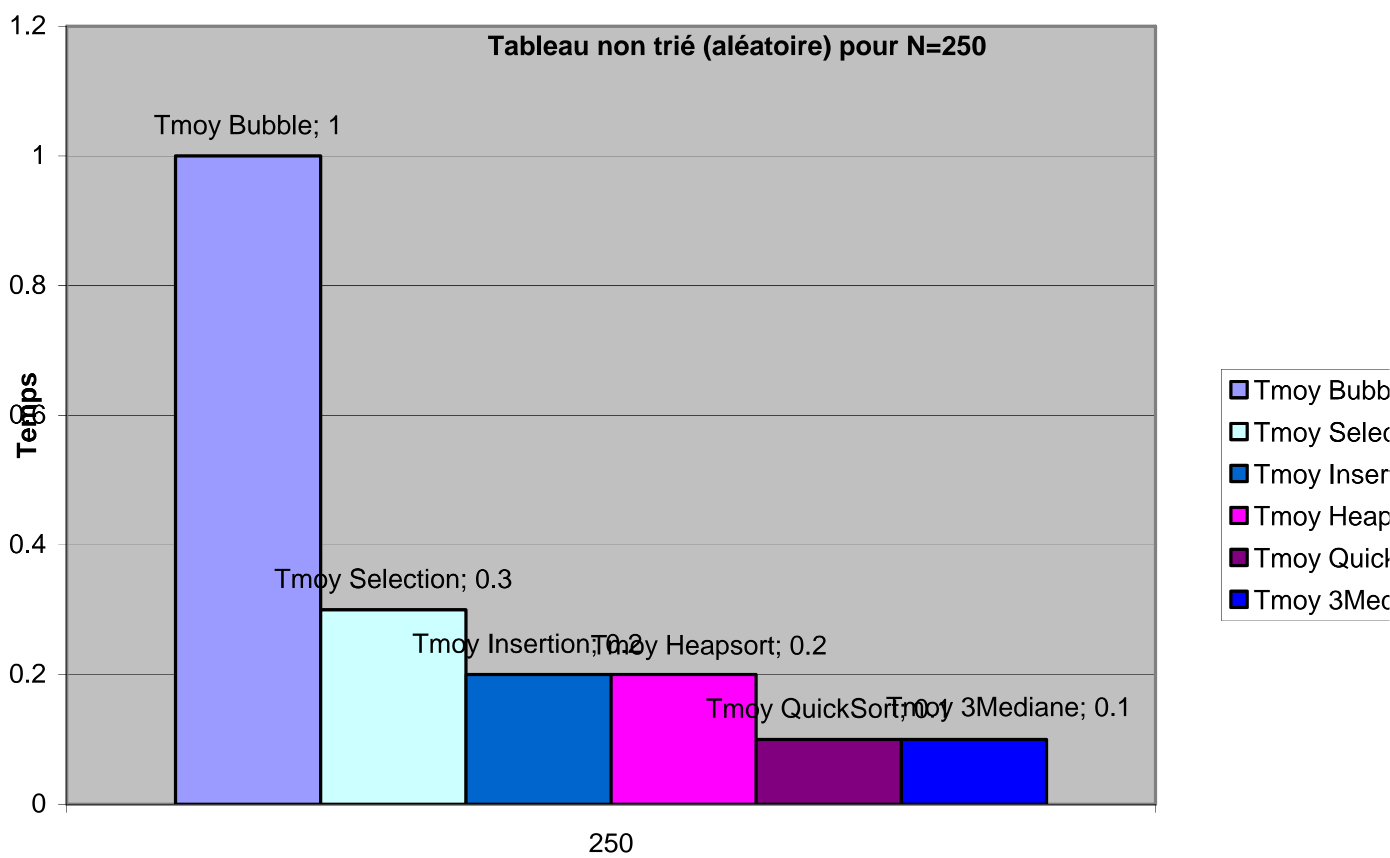


Tableau non trié (aléatoire) pour N=250
Temps
1.2
1
0.8
0.6
0.4
0.2
0
250
Tmoy Bubble; 1
Tmoy Selection; 0.3
Tmoy Insertion; 0.2
Tmoy Heapsort; 0.2
Tmoy QuickSort; 0.1
Tmoy 3Mediane; 0.1

5. ANNEXE DES RESULTATS DES TESTS

- Annexe1 : Résultats des tests de Tri à Bulles(BubbleSort)
- Annexe2 : Résultats des tests de Tri par Sélection(SelectionSort)
- Annexe1 : Résultats des tests de Tri par Insertion(InsertionSort)
- Annexe1 : Résultats des tests de Tri par Shell (ShellSort)
- Annexe1 : Résultats des tests de Tri rapide(quickSort)
- Annexe1 : Résultats des tests de Tris trois-médiane(three-median)
- Annexe1 : Résultats des tests de Tri par tas(heapSort)