Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département Intelligence Artificielle et Sciences des données

Spécialité: Système Informatique Intelligent

Module: Complexité des algorithmes

Rapport du TP 02 : Les Algorithmes de Tri

Année universitaire : 2022/2023

Ce rapport a été réalisé par :

“TEAM 28”

NOM:	PRÉNOM:	MATRICULE:
BEHLOUL	HALA LYNA	191931068962
HENDEL	LYNA MARIA	191931068959
NAAR	SARAH	181831087978
NADIR	SOMIA	181833020729

Table des matières

problématique :

Comment ranger des données afin de faciliter leur accès futur ?

C'est par exemple l'ordre alphabétique du dictionnaire, où les mots sont rangés dans un ordre logique qui permet de ne pas devoir parcourir tout l'ouvrage pour retrouver une définition,Cette problématique permet d'introduire la notion de tri (avec plusieurs sens distincts : séparer, ordonner, choisir), puis d'étudier différents algorithmes de tri. Le tri permet essentiellement d'accélérer les recherches, grâce à différentes méthodes .

objectif de ce tp :

L'objectif de ce TP est de mesurer le temps d'exécution de différents algorithmes de tri en C. En utilisant les fonctions de gestion du temps () de ce langage,les algorithmes de tri auxquels nous allons nous intéresser existent pour la plupart Nous nous intéresserons donc plutôt à l'efficacité de ces algorithmes afin de pouvoir les comparer entre eux.

calcul de complexité:

Afin d'évaluer la complexité des différents algorithmes de tri présentés, on comptera le nombre de comparaisons et d'échanges de valeur entre deux éléments du tableau sans prendre en compte les affectations et comparaisons sur des variables de comptage de boucles.

Les méthodes présentées sont de deux types :

- des méthodes qui trient les éléments deux à deux, de manière plus ou moins efficace, mais qui nécessitent toujours de comparer chacun des N éléments avec chacun des N-1 autres éléments, donc le nombre de comparaisons sera de l'ordre de N^2— on note cet ordre de grandeur O(N^2). Par exemple, pour N=1000, N^2=10^6, pour N=10^6, N^2=10^8. Les algorithmes de ce type sont :

    1. une méthode de tri élémentaire, le tri par sélection ;
    2. et sa variante, le tri par propagation ou tri bulle ;
    3. une méthode qui s'apparente à celle utilisée pour trier ses cartes dans un jeu,le tri par insertion ;

- des méthodes qui sont plus rapides, car elles trient des sous-ensembles de ces N éléments puis regroupent les éléments triés, elles illustrent le principe « diviser pour régner ». Le nombre de comparaisons est alors de l’ordre de N (log(N)). Par exemple, pour N=1000, N(log(N))=10000 environ, pour $N=10^6$, $N(log(N))=20\times10^6$ environ. Les algorithmes de ce type sont :

1. le fameux tri rapide ou *Quicksort* ;
2. et enfin, le tri par fusion

Environnement matériel

Les expérimentations pour chaque algorithme ont été effectuées sous la machine ayant les caractéristiques suivantes Ordinateur portable

Dell Intel’R) cORE(™) i7-7500U CPU @ 2.70GHz , 2901MHz 2 coeurs , 8Go RAM

et

Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz 2.70 GHz,8,00 Go (7,90 Go utilisable)

i7-8565U 1.80GHz 1.99 GHz 8ram

Partie 1 & 2

Les Algorithmes de trie , leur complexités , les tableaux des tests et les graphes

Tous les algorithmes de tri utilisent une procédure qui permet d'échanger (de permuter) la valeur de deux variables Dans le cas où les variables sont entières,

la procédure échanger est la suivante :

procédure échanger (E/S a,b : Entier)
Déclaration temp : Entier **<u>début</u>** **temp ← a** **a ← b** **b ← temp** **<u>fin</u>**

Algorithme 1 : Tri par sélection

1.1. Description de l'algorithme:

Le tri par sélection (ou tri par extraction) est un algorithme de tri par comparaison. Cet algorithme est simple, mais considéré comme inefficace car il s'exécute en temps quadratique en le nombre d'éléments à trier, et non en temps pseudo linéaire.

Le principe du tri par sélection est le suivant :

- 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.

1.2. pseudo code:

```
procédure tri_selection(tableau t)

  n ← longueur(t)
   pour i de 0 à n - 2
       min ← i
       pour j de i + 1 à n - 1
           si t[j] < t[min], alors min ← j
       fin pour
       si min ≠ i, alors échanger t[i] et t[min]
   fin pour
 fin procédure
```

1.3. Calcul de la complexité

Dans tous les cas, pour trier *n* éléments, le tri par sélection effectue (n*(n-1))/2 comparaisons. **Sa complexité est donc Θ(n^2)**. De ce point de vue, il est inefficace puisque les meilleurs algorithmes s'exécutent en temps O(n log n). Il est même moins bon que le tri par insertion ou le tri à bulles, qui sont aussi quadratiques dans le pire cas mais peuvent être plus rapides sur certaines entrées particulières.

Complexité Temporelle(n)=Θ(n^2)

- **Le pire cas** est quand le tableau est dans l'ordre inverse car il devra faire les permutations plus les comparaisons pour tous les éléments

- **Le meilleur cas** est quand le tableau est trié dans ce cas l'algo doit faire les comparaisons seulement

- dans le **cas moyen** l'algorithme doit faire les comparaisons dans tout le tableau et faire quelques permutations

1.4. Mesure du temps d’exécution :

Taille du tableau	10^4	5*10^4	10^5	5*10^5	10^6	5*10^6	10^7	5*10^7	10^8
Aléatoire	0.155	3.155	12.085	316.35	1177.06	3674.3451	5779.26	7061.32	10007.038
Dans le bon ordre	0.121	2.944	11.824	314.883	1167.772	3651.921	5770.12	6795.12	9514.054
Ordre inverse	0.126	5.376	18.491	335.388	1218.851	3680.21	6217.17	7216.12	10048.294

1.5. Le graphe

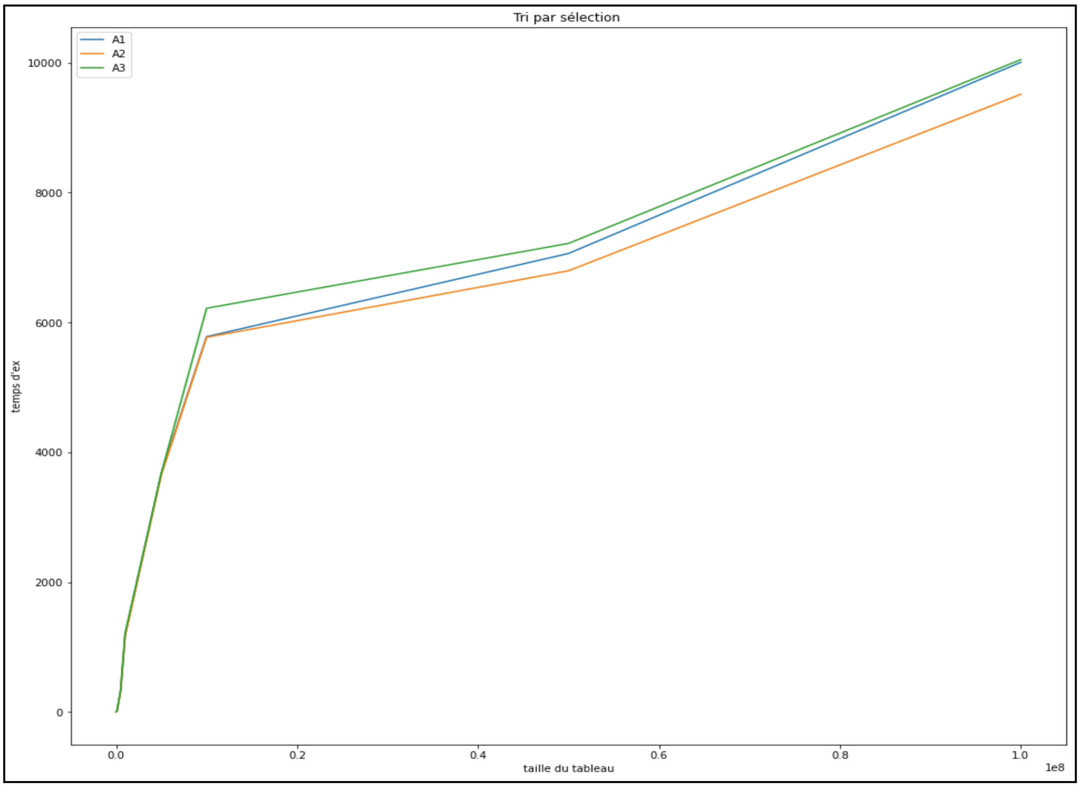


Figure 1: Graphe du temps d’exécution de l’algorithme du tri par sélection

Selon le graphe, on remarque que les temps d'exécution augmente presque de la même manière pour de petites taille du tableau ensuite plus la taille augmente considérablement plus le temps d'exécution augmente, le plus rapide étant celui des données triées, suivi de l'aléatoire en finissant par ceux d'ordre inverse.

2 . Algorithme 2: Tri par insertion :

2.1. Description de la méthode:

Le tri par insertion est l'un des algorithmes les plus classiques et son utilisation est très répandue vu sa facilité.
Le principe de l'algorithme de tri est de parcourir une liste non triée, on la décompose en deux, une partie triée et l'autre non. On prend l'élément le plus à gauche dans la partie non triée et on l'insère dans sa place dans le côté trié
Tant qu'il nous reste un élément à ranger dans le côté non trié on continue de faire des insertions et les insertions de l'élément le plus à gauche se fait par décalage consécutifs d'une cellule.

2.2. le pseudo code:

```
fonction tir_insertion_itératif(T, n)
debut
    pour i de 1 a n - 1
    faire
        insert(T, n, i)
    fait;
fin;



procedure insert(T[]:d'entier, n:entier, i:entier)
    VAR
        x: entier;
    DEBUT
        x = T[i]
        tant que (i > 0 et T[i - 1] > x)
        faire
            T[i] = T[i - 1]
            i = i - 1
        fait;
        T[j] = x
    FIN.
```

2.3.1 Calcul de la complexité Temporelle:

➔ **Nous allons traiter trois cas: Le pire et le meilleur et le moyen**

❖ **Pire cas**: Le pire cas que nous puissions rencontrer est lorsque le tableau est trié de l'ordre inverse. Et donc dans le pire ds cas on a i comparaisons pour chaque i allant de 2 jusqu'à n la complexité est donc égale à la somme des n termes (i=2, i=3, i=4, …, i=n)

Complexité temporelle(n) = 2+3+4+...+N = $\frac{N(N+1)}{2} - 1$ Comparaison dans le pire des cas

Complexité Temporelle (n)= $\frac{N^2+N-2}{2}$ comparaisons ou tests dans le pire des cas

La complexité dans ce cas est de l'ordre de O(N^2)

Meilleur Cas:

Le meilleur cas correspond au cas ou notre tableau est trié, donc l'algorithme ne rentre jamais dans la boucle , donc on a N-1 comparaisons
Complexité(n)= N+(N+1)=2N+ 1 Comparaisons.

La complexité dans le meilleur cas est de l'ordre de O(N).

2.3.2 Calcul de la complexité Spatiale:

L'algorithme de tri par insertion utilise le tableau de taille N afin de trier les deux variables entières et il n'alloue donc pas de case mémoire.

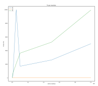
La taille d'un nombre entier est sur 4 octets la complexité spatiale est calculée de la sorte:

Complexité Spatiale(n)= 4(N+2) octets

❖ **le pire cas** correspond au cas ou le tableau est trié dans l'ordre inverse et dans ce cas la on aura a faire n-1 comparaison et jusqu'à n permutations
❖ **le meilleur cas** correspond au cas ou le tableau est trié dans le bon ordre dans ce cas on aura que les comparaisons

❖ dans le **cas moyen** c'est à dire le tableau est aléatoirement ordonné on aura les comparaisons plus quelques permutations

2.4. Mesure du temps d'exécution

Taille	**10^4**	**5*10^4**	**10^5**	**5*10^5**	**10^6**	**5*10^6**	**10^7**	**5*10^7**	**10^8**
aléatoire	0.11	2.52	9.999	243.748	582.145	10004.	1704.654	2621.840	5006.001
Trié	0.000	0.00	0	0.002	0.003	0.01	0.03	0.05	0.03
Inverse	0.217	4.835	19.239 	475.048	761.831	2074.563	3627.320	5237.012	9943.732

2.7. Le graphe :

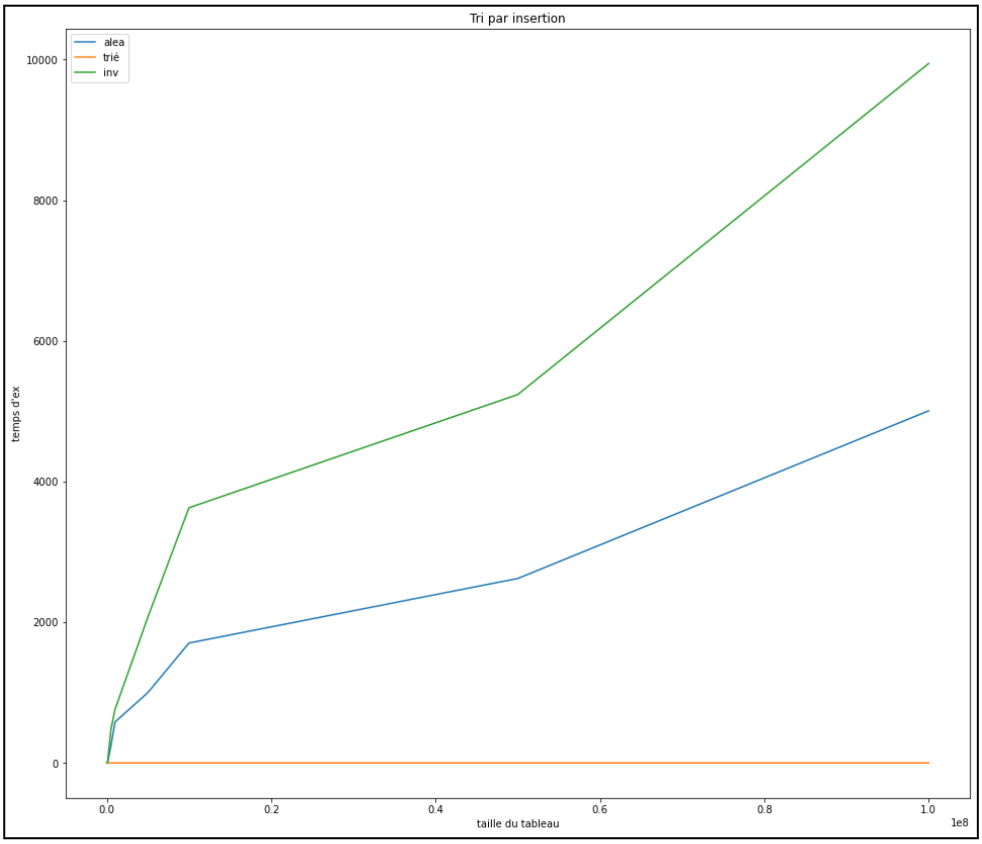


Figure 2: Graphe du temps d'exécution de l'algorithme du tri par insertion

On peut remarquer que les temps d’exécution ont pratiquement été multipliés par 4 lorsque les données sont triées dans l’ordre inverse, pour le tableau des données triées aléatoirement le temps d'exécution a doublé contrairement au tableau des données triées dans le bon ordre.

3-Algorithme 3: Tri a Bulles

3.1. Description de la méthode de tri

Le principe de cet algorithme est de parcourir le tableau element par element et si ils ne sont pas dans l'ordre il permute entre ces derniers

3.2. Pseudo code :

tri_à_bulles(Tableau T)

pour i allant de (taille de T)-1 à 1
    pour j allant de 0 à i-1
        si T[j+1] < T[j]
            (T[j+1], T[j]) = (T[j], T[j+1])

3.3. Calcul de complexité:

Le tri à bulles est un algorithme de tri lent, on peut donc s'attendre à une complexité importante.

Le pire cas correspond à quand le tableau est trié dans l'ordre inverse,
Le programme effectue donc un nombre important de permutation, tel que sa complexité est de : **CT(n) = (n-2)+1 + ([n-1]-2)+1 +.....+1+0 = (n-1)+(n-2)+...+1 tests.**

CT(n) = N(N−1) 2 tests.

sa complexité est donc O(n^n)

Le meilleur cas est quand le tableau est trié dans l'ordre souhaité mais il effectuera quand même n^2 comparaison

- le **pire cas** est quand le tableau est trié dans l'ordre inverse dans ce cas on aura n^2+n itérations entre les comparaisons et les permutations

- le **meilleur cas** est presque pareil que le cas moyen car même quand le tableau est trié dans le bon ordre on aura quand même n^2 comparaisons

❖ le **cas moyen** l'algo doit quand même parcourir tout le tableau et faire les comparaisons et les permutations

3.4. Mesure du temps d'exécution

Taille du tableau	**10^4**	**5*10^4**	**10^5**	**5*10^5**	**10^6**	**5*10^6**	**10^7**	**5*10^7**	**10^8**
Ordre aleatoire	0.36	7.344	29.68	726.868	4024.31	60913.204	110091.906	3249837.831	4754371.023
Par ordre	0.11	2.837	11.267	276.036	1821.341	34143.021	50127.047	2125419.014	3108430.492
Ordre inverse	0.197	5.079	20.415	502.96	2804.379	5109.421	899104.556	2843791.271	3920074.102

3.5. Le graphe

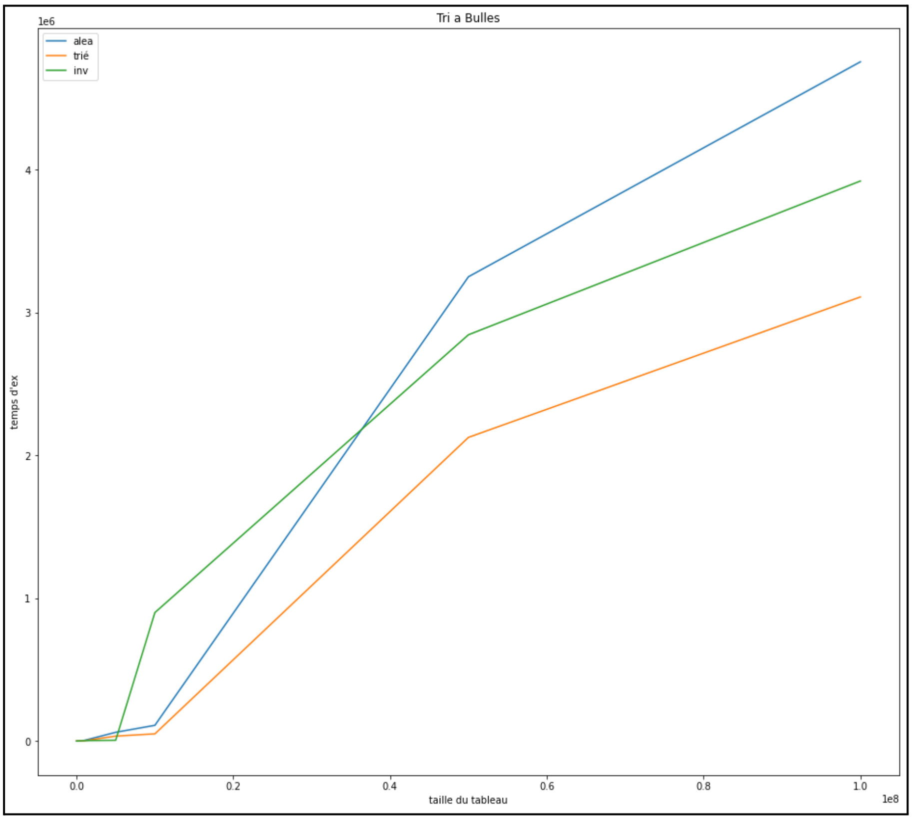


Figure 3: Graphe du temps d'exécution de l'algorithme du tri à bulles

Pour ce qui est du tri à bulles, on peut constater selon le graphe que le temps d'exécution pour le tableau trié est le meilleur, suivi des données triées de manière aléatoire et finissant par ceux d’ordre inverse sauf que ces deux derniers s’inversent à une taille bien précise laissant le tableau des données aléatoirement avec le temps d’exécution le plus élevé.

4.Algorithme 4 : Tri rapide:

4.1. Description de la méthode de tri:

En informatique, le tri rapide ou tri pivot (en anglais quicksort) est un algorithme de tri fondé sur une méthode qui illustre le principe dit diviser pour régner .

Ce dernier consiste à parcourir un tableau Tab en le divisant systématiquement en deux sous-tableaux Tab1 et Tab2 . L'un est tel que tous ses éléments sont inférieurs à tous ceux de l'autre tableau et en travaillant séparément sur chacune des deux sous-listes en ré-appliquant la même division à chacune des deux sous-listes jusqu'à obtenir uniquement des sous-tableaux qui contiennent un seul élément.

C'est un algorithme dichotomique qui divise donc le problème en deux sous-problèmes dont les résultats sont réutilisés par recombinaison, il est donc de complexité O(n.log(n)).

- ➔ **les étapes à suivre au cours de l'algorithme**

  soit un tableau A[p..r] ;

- ➢ diviser : choisir une valeur quelconque d'indice q dans le tableau A[p..r] A[q], que l'on dénomme pivot, Partitionner en 3 ce sous tableaux :

  - ☐ A[p..q-1] contient les clés inférieures à A[q], cela va constituer le sous tableau 1
  - ☐ A[q] le pivot
  - ☐ A[q+1..r] contient les clés supérieures à A[q].

  remarque : Les deux sous-tableaux gauche et droite peuvent éventuellement être vides.

- ➢ régner : les sous-tableaux A[p..q-1] et A[q+1..r] sont traités en appelant récursivement le tri rapide.

- ➢ combiner : cette phase est instantanée. Puisque les sous-tableaux sont triés sur place, aucun travail n'est nécessaire pour les combiner.

Le choix du pivot est déterminant pour l'efficacité de ce tri. Plusieurs options sont possibles :

1. Choisir le premier élément du tableau
2. Choisir le dernier élément du tableau
3. Choisir un élément au hasard
4. Choisir l'élément au milieu du tableau
5. Trouver le pivot optimal en recherchant la médiane

4.3. Pseudo code itératif:

```
procedure permuter ( a* :entier , b* : entier )

VAR
        temp : entier ;
DEBUT
        temp = *a;
       *a = *b;
       *b = temp ;
FIN .
```

```
fonction Partition ( data [1.. n]: entier , left :entier , right : entier): entier

VAR
      x,i,j: entier ;
DEBUT

          x = data [ right ];
          i = ( left - 1);


POUR j de left a j right - 1 pas 1)

FAIRE
         SI( data [j] <= x)
            ALORS
               ++i;
```

```
    permuter (& data [i], & data [j]);
  FIN SI;
FAIT ;
     permuter (& data [i + 1], & data [ right ]);
     return (i + 1);
FIN .
```

```
Procédure QuickSortIterative ( data [1.. n]: entier , count : entier )

VAR
    startIndex , endIndex ,top: entier ;
    stack * : entier ;

DEBUT
   startIndex = 0;
   endIndex = count - 1;
   top = -1;
   stack = ( entier *) Allouer ( Taille ( entier ) * count );
   stack [++ top] = startIndex ;
   stack [++ top] = endIndex ;
   TANT QUE (top >= 0)
       FAIRE
          endIndex = stack [top --];
          startIndex = stack [top --];
          p: entier ;
          p = Partition (data , startIndex , endIndex );

          SI(p - 1 > startIndex )
            ALORS
                DEBUT
                   stack [++ top] = startIndex ;
                   stack [++ top] = p - 1;
                FIN ;
         FIN SI;

         SI(p + 1 < endIndex )
         ALORS
              DEBUT
```

```
                                   stack [++ top] = p + 1;
                                   stack [++ top] = endIndex ;
                         FIN
                  FIN SI;
FAIT ;
Liberer ( stack );
FIN .
```

4.4. Pseudo code récursive avec les trois pivots :

remarque : durant cette question nous avons fait plusieurs code afin de comprendre ou est ce qu'il se trouve le problème qu'on on utilisé le pivot comme le premier et le dernier élément du tableau et aussi afin de trouver l'algorithme le plus optimal en comparant leurs complexité .

1. algorithme récursif tri rapide

```
void tri_rapide (int *tableau, int taille)

{
    int mur, courant, pivot, tmp;
    if (taille < 2) return;
    // On prend comme pivot l'élément qui se trouve au milieu
    pivot = tableau[taille/2];
    mur  = courant = 0;
    while (courant<taille) {
        if (tableau[courant] <= pivot) {
            if (mur != courant) {
                tmp=tableau[courant];
                tableau[courant]=tableau[mur];
                tableau[mur]=tmp;
            }
            mur ++;
        }
        courant ++;
    }
    tri_rapide(tableau, mur - 1);
    tri_rapide(tableau + mur - 1, taille - mur + 1);
}

int* ordre_inv(int *t,unsigned long long n){
     int i,temp;
            clock_t t1,t2;
    double delta;
    t1 = clock();
```

```
    for(i = 0; i<n/2; i++){
        temp = t[i];
        t[i] = t[n-i-1];
        t[n-i-1] = temp;
    }
      t2 = clock();
    delta = (double) (t2-t1)/CLOCKS_PER_SEC ;

   printf("Le temps d'exécution est (ordre inv) : %lf\n",delta);
    return t;
}
int main()
iint taille_1=500000; //on prends la taille comme 50000
int  i;
    int*t,*tmp;
    t=creatTab(taille_1);// creation du tableau t
    tmp =(int *)malloc(taille_1 * sizeof(int));
  //aleatoire

 clock_t t1,t2,t3,  t4;
 double delta ;
 printf("bonjour; Voici un programme de tri de %d nombres \n", taille_1);

 //l'inverse du tableau

 /* Saisie par l'opérateur des valeurs entières dans le tableau */
 t1 = clock();
 /* Tri du tableau */
 tri_rapide (t,taille_1);
 t2 = clock();
    delta = (double) (t2-t1)/CLOCKS_PER_SEC ;
printf("Le temps d'execution est : %lf\n",delta);

  t1 = clock();
 /* Tri du tableau */

 t= ordre_inv(t,taille_1);
 t1  = clock();
tri_rapide (t,taille_1);
 t2 = clock();
    delta = (double) (t2-t1)/CLOCKS_PER_SEC ;
printf("Le temps d'exécution  d'un tableau  inverse  est : %lf\n",delta);
```

2. algorithme récursif tri rapide

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <time.h>


/* fonction qui permet à l'opérateur de saisir sa liste de nombres
 */
int* creatTab(unsigned long long n){
    int i;


    int *t =(int *)malloc(n * sizeof(int));

    for(i=0; i<n; i++)
    {
        t[i]=rand();

    }

        return t;

}

/* fonction de tri RÉCURSIF
 */

void tri_tab_recursif(int * tab,int deb,int fin)
{
 const int pivot = tab[deb];
 int pos=deb;
 int i;

 if (deb>=fin)
 return;

/* cette boucle permet de placer le pivot (début du tableau à trier)
 au bon endroit dans le tableau
 avec toutes les valeurs plus petites avant
 et les valeurs plus grandes après
 à la fin, la valeur pivot se trouve dans le tableau à tab[pos]
 */
 for (i=deb; i<fin ; i++)
 {
 if (tab[i]<pivot)
```

```
 {
 tab[pos]=tab[i];
 pos++;
 tab[i]=tab[pos];
 tab[pos]=pivot;
 }
 }
/* Il ne reste plus qu'à rappeler la procédure de tri
 sur le début du tableau jusquà pos (exclu) : tab[pos-1]
 */
 tri_tab_recursif(tab,deb,pos);
/* et de rappeler la procédure de tri
 sur la fin du tableau à partir de la première valeur après le pivot
 tab[pos+1]
 */
 tri_tab_recursif(tab,pos+1,fin);


}
int* ordre_inv(int *t,unsigned long long n){
     int i,temp;
          clock_t t1,t2;
    double delta;
    t1 = clock();
    for(i = 0; i<n/2; i++){
        temp = t[i];
        t[i] = t[n-i-1];
        t[n-i-1] = temp;
    }
      t2 = clock();
    delta = (double) (t2-t1)/CLOCKS_PER_SEC ;

   printf("Le temps d'execution est (ordre inv) : %lf\n",delta);
    return t;
}
int main()
{
int taille_1=500000;
int  i;
    int*t,*tmp;
    t=creatTab(taille_1);// creation du tableau t
    tmp =(int *)malloc(taille_1 * sizeof(int));
  //aleatoire
 clock_t t1,t2,t3,  t4;
 double delta ;
 printf("bonjour; Voici un programme de tri de %d nombres \n", taille_1);

 //l'inverse du tableau

 /* Saisie par l'opérateur des valeurs entières dans le tableau */
```

```
 t1 = clock();
 /* Tri du tableau */
 tri_tab_recursif(t,0,taille_1);
 t2 = clock();
    delta = (double) (t2-t1)/CLOCKS_PER_SEC ;
printf("Le temps d'execution est : %lf\n",delta);

  t1 = clock();
 /* Tri du tableau */

 t= ordre_inv(t,taille_1);
 t1  = clock();
tri_tab_recursif(t,0,taille_1);
 t2 = clock();
    delta = (double) (t2-t1)/CLOCKS_PER_SEC ;
printf("Le temps d'execution  d'un tableau  inverse  est : %lf\n",delta);

}
```

4.5. Calcul de la complexité temporelle:

Les performances du tri rapide dépendent de la manière dont la procédure PARTITION parvient à créer deux sous-tableaux équilibrés ou non.

Dans le pire cas :

le tableau est trié dans l'ordre inverse et le 1er élément sera le pivot dans ce cas on aura n tests pour le premier élément n-1 pour le 2eme il donne la récurrence suivante :
le 1er ´el´ement (plus grand) du tableau sera pris en pivot tout le tableau sera parcouru pour atteindre de n i`eme ´élément (plus petit) afin de les permuter puis, ainsi le n i`eme ´élément (plus grand) sera `à sa place, on vérifie pour le 1er ´élément (plus petit) qui s'avère aussi a sa place donc pas de permutation (mais tous le tableau est parcouru pour s'en rendre compte).
On se retrouve dans la situation : le n-1 i`eme ´élément (2eme plus grande valeur du tableau), a été  permutée avec le 2eme, puis, comme toutes les autres valeurs sont inférieures au n-1 i`eme ´el´ement, nos deux recherches d'´élément mal placés se sont croisées , et on n'a rien permuté de plus. D'après cette étape, un autre passage déterminera que le 2eme est bien placé et qu'on peut l'ignorer après avoir comme même parcouru tout le tableau.
Ce schéma se répétera ainsi jusqu'à ce que tout le tableau soit trié. On se rend compte que, à chaque étape, on place correctement une valeur et que, pour que cela se produise, on parcourt tout l'ensemble du tableau à trier.

Ainsi, le premier passage fait n tests, le second n-1, et ainsi de suite. Au final, on aura effectué N + (N - 1) + (N - 2) + ... + 3 + 2 + 1 tests,

- ➔ ce qui nous donne une somme égale à $\frac{(n^2+n)}{2}$

soit la notation landau **O(N^2)**

Le meilleur cas correspond à choisir le pivot le plus proche de la médiane, **pire cas** est quand le tableau est trié

- ➔ le **pire cas** correspond au cas ou le tableau est trié dans l'ordre inverse et le premier élément est le pivot dans ce cas on aura plus de n^2 tests
- ➔ le **meilleur cas** est quand le tableau est trié dans l'ordre souhaité et le pivot choisi est proche de la médiane
- ➔ le **cas moyen** est quand le tableau est tiré aléatoirement et le pivot est choisi au hasard

4.6. Mesure du temps d'exécution

Taille du tableau	**10^4**	**5*10^4**	**10^5**	**5*10^5**	**10^6**	**5*10^6**	**10^7**	**5*10^7**	**10^8**
Ordre aléatoire	0.005	0.034	0.048	0.218	0.253	2.958	10.037	204.505	789.04
Ordre inverse	0.002	0.014	0.011	0.061	0.162	2.548	9.15	202.657	794.699
Ordre trie	0.001	0.014	0.013	0.077	0.202	3.599	13.727	307.737	1057.21

4.7. Mesure du temps d'exécution en fonction des pivot en utilisant l'algorithme récursif :

1.1.1. Choix du pivot 01 : Premier élément du tableau avec l'algorithme récursif pivot = data [deb]

du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Ordre aléatoire	0.003000	0.007000	0.023000	-	-	-	-	-	-

Ordre inverse	0.14600 0	3.30700 0	-	-	-	-	-	-	-
Ordre inverse	0.19700 0	2.80300 0	-	-	-		–	-	-

1.1.2. Choix du pivot 02 :l'élément du milieu de du tableau avec l'algorithme récursif

soit pivot =data [deb+(fin-deb)/2)]

du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Ordre aléatoire	0.00100 0	0.01300 0	0.01700 0	0.08600 0	0.19800 0	2.06500 0	5.89400 0	115.011 000	437.405 000
Ordre inverse	0.00100 0	0.00400 0	0.00700 0	0.04700 0	0.12500 0	1.55700 0	5.24800 0	110.181 000	408.955 000
Ordre inverse	0.00100 0	0.00400 0	0.01100 0	0.05200 0	0.13100 0	1.76600 0	6.60400 0	143.469 000	509.989 000

1.1.3. Choix du pivot 03 :l'élément un eléments aléatoire de du tableau avec l'algorithme récursif avec le pivot =data[fin-100]

du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Ordre aléatoire	0.09800 0	1.19000 0	-	-	-	-	-	-	-
Ordre inverse	0.07900 0	1.18500 0	-	-	-	-	-	-	-
Ordre inverse	0.16700 0	3.29600 0	-	-	-	-	-	-	-

4.7. Mesure du temps d’exécution en fonction des pivot en utilisant l’algorithme itératif:

1. **Choix du pivot 03 : un élément aléatoire de du tableau avec l’algorithme récursif avec le pivot =** data[left+100]

Taille du tableau	**10^4**	**5*10^4**	**10^5**	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Ordre aléatoire	0.023	0.142	0.214	-	-	-	-	-	-
Ordre inverse	0.026	0.107	0.22	-	-	-	-	-	-
Ordre trie	0.024	0.115	0.227	-	-	-	-	-	-

2. **Choix du pivot 03 :dernier élément du tableau avec l’algorithme récursif avec le pivot** =data[FIN]

	10*4	5*10**4	10**5	5*10**5					
alea	0.001	0.021	0.034	0.099	-	-	-	-	-
trie	0.285	6.842	27.972	-	-	-	-	-	-
inv	0.182	3.143	10.221	-	-	-	-	-	-

4.8. analyse des tableaux en fonction des pivots choisit :

nous remarquons qu’au cours de certaines exécution l’algorithme crash et ne termine pas son exécution et cela revient au choix du pivot et de la taille du tableau . Cela est dû à cause de la surcharge de la pile récursive .

L’implémentation de la récursivité nécessite l’utilisation d’une pile. À chaque appel récursif, il faut en effet mémoriser les valeurs des variables locales de la fonction. Voici ce qu’il se passe lorsque la ligne tri_rapide(tableau, mur - 1); est exécutée

La pile nécessaire au stockage des valeurs des variables locales a une taille par défaut de 1000. Si l’on souhaite faire plus de 1000 appels récursifs, il faut modifier la limite ce qui est cas dans notre situation quand la taille d'un tableau n dépasse les $5*10^5$
 L’appel récursif peut conduire à une légère surcharge de travail comparé à la boucle itérative, c’est pourquoi la seconde option est probablement moins rapide que la première

(d’un facteur constant indépendant de n). Par ailleurs, l’implémentation récursive utilise plus de mémoire (à cause de la pile).

4.9. Le graphe

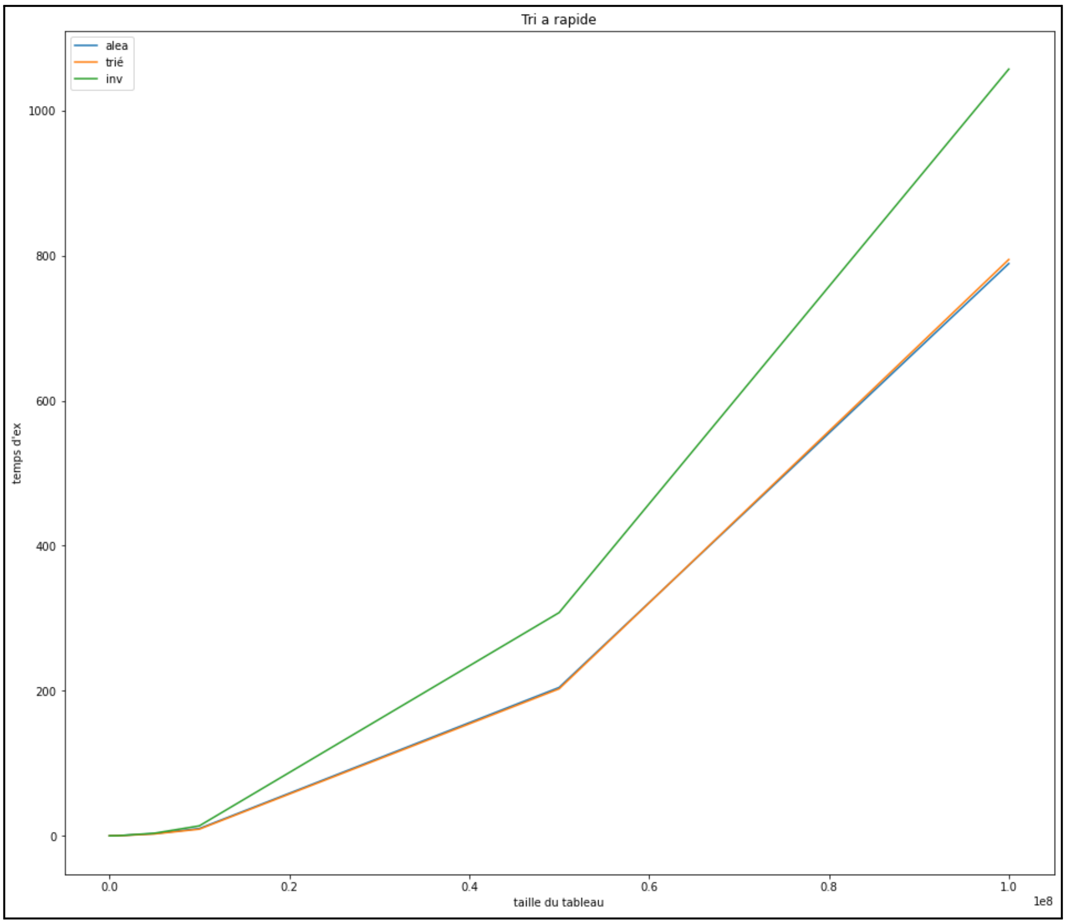


Figure 4: Graphe du temps d’exécution de l’algorithme du tri rapide

5. **Algorithme 05: Tri par fusion:**

5.1.Description de l'algorithme

Le tri fusion se décrit naturellement sur des listes et c'est sur de telles structures qu'il est à la fois le plus simple et le plus rapide. Cependant, il fonctionne aussi sur des tableaux. La version la plus simple du tri fusion sur les tableaux a une efficacité comparable au tri rapide, mais elle n'opère pas en place : une zone temporaire de données supplémentaire de taille égale à celle de l'entrée est nécessaire (des versions plus complexes peuvent être effectuées sur place mais sont moins rapides). Sur les listes, sa complexité est optimale, il s'implémente très simplement et ne requiert pas de copie en mémoire temporaire.

5.2.Pseudo Algorithme:

```
fonction triFusion( int i, int j, int tab[], int tmp[])
```

```
VAR
      m,i,j,pg(pointeur gauche),pd,c;
DEBUT
           m = (i + j) / 2;
           triFusion(i, m, tab, tmp);
           triFusion(m + 1, j, tab, tmp);
           pg = i;    pd = m + 1;

POUR c de i a j  pas 1  FAIRE
         Si pg == m + 1  Alors   tmp[c] = tab[pd];  pd++;
           Sinon  Si pd == j + 1 Alors  tmp[c] = tab[pg];   pg++;
                     Sinon Si tab[pg] < tab[pd]  Alors tmp[c] = tab[pg];  pg++;
         Sinon   tmp[c] = tab[pd];   pd++;
FAIT

Pour c de i a j pas 1  FAIRE     tab[c] = tmp[c];  FAIT

FIN .
```

5.3. Mesure du temps d'exécution:

Taille tableau	**10^4**	**5*10^4**	**10^5**	**5*10^5**	**10^6**	**5*10^6**	**10^7**	**5*10^7**	**10^8**
Aléatoire	0.007	0.046	0.037	0.111	0.234	1.025	2.131	11.178	23.056
Par ordre	0.002	0.014	0.012	0.06	0.109	0.567	1.188	6.434	13.401
Ordre inverse	0.004	0.017	0.010	0.073	0.101	0.558	1.181	6.385	13.364

5.3.1 Calcul de la complexité temporelle:

Au pire et meilleur cas il s'agit de la même complexité, tel que :

CT(N) =2 *CT($\frac{N}{2}$) + 2 * N si N ≤ 2

sinon 1

Pour les puissances de 2, on a CT(2^n) = 2 * CT(2^{n-1}) + 2 *2^n
et il suffit de le montrer par récurrence

- **le meilleur cas** est quand le tableau est trié dans le bon ordre ou dans l'ordre inverse
- **le pire cas** est quand le tableau est aléatoirement trié

5.4. Le graphe

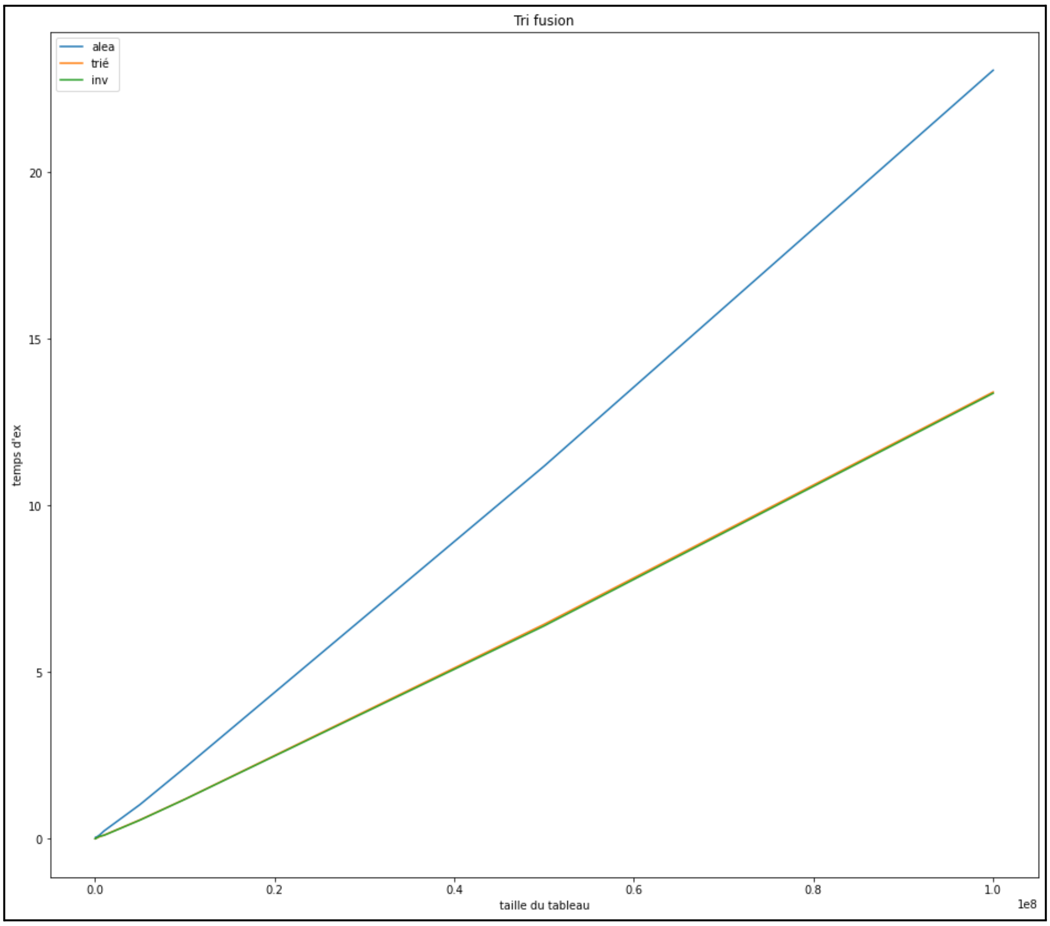


Figure 5: Graphe du temps d'exécution de l'algorithme du tri par fusion

Le graphe suivant démontre que le meilleur temps d'exécution est celui du tableau ayant les données triées dans le bon ordre ou l'ordre inverse, laissant dans ce cas le pire temps d'exécution au tableau des données triées de manières aléatoires.

6. Algorithme 6: Tri par tas

L'idée qui sous-tend cet algorithme consiste à voir le tableau comme un arbre binaire. Le premier élément est la racine, le deuxième et le troisième sont les deux descendants du premier élément, etc. Ainsi le n élément a pour enfants les éléments 2n et 2n+1 si l'indexation se fait à partir de 1 (2n+1 et 2n+2) si l'indexation se fait à partir de 0). Si le tableau n'est pas de taille (2^{n}-1), les branches ne se finissent pas toutes à la même profondeur.

Dans l'algorithme, on cherche à obtenir un tas, c'est-à-dire un arbre binaire vérifiant les propriétés suivantes (les deux premières propriétés découlent de la manière dont on considère les éléments du tableau) :

- la différence maximale de profondeur entre deux feuilles est de 1 (*i.e.* toutes les feuilles se trouvent sur la dernière ou sur l'avant-dernière ligne) ;
- Les feuilles de profondeur maximale sont « tassées » sur la gauche.
- chaque nœud est de valeur supérieure (resp. inférieure) à celles de ses deux fils, pour un tri ascendant (resp. descendant).

6.2. Pseudo algorithme:

```
Procedure tri_tas ( arbre []: d ‘ entier , longueur : entier )
{ VAR i : entier ;
DEBUT
 Pour i := longueur /2 a 1
          tamiser ( arbre , i , longueur ) ;
                    Fait ;
 Pour i := longueur a 2
          permuter ( arbre [1] , arbre [ i ]) ;
          tamiser ( arbre , 1 , i -1) ;
                    Fait ;
 FIN . }
Procedure tamiser ( arbre , noeud , n )
{VAR k := noeud;
 j := 2* k;
 DEBUT
 Tant que j <= n
          SI ( j < n et arbre [ j ] < arbre [ j +1])
                    ALORS j := j +1;
          FinSi ;
           SI ( arbre [ k ] < arbre [ j ])
                    ALORS permuter ( arbre [ k ] , arbre [ j ]) ;
                     k := j ; j := 2* k ;
```

```
                SINON j := n +1
                FinSi ;
        Fait ;
FIN .
```

6.3. Complexité temporelle

La complexité est la même dans le pire et le meilleur cas.

Dans cet algorithme on fait appel à la fonction tamiser n fois et c'est à l'intérieur de cette fonction qu'on parcourt le tableau qui est sous forme d'un arbre, le parcours se fait avec deux éléments à la fois, chaque éléments a deux fils et ainsi de suite, ceci nous donne une complexité de l'ordre de log(n).

Complexité temporelle(n) = O(n log n);

Le nombre d'échanges dans le tas est majoré par le nombre de comparaisons et il est du même ordre de grandeur.
La complexité au pire en nombre de transferts du tri par tas est donc en O(n log n).

Calcul de la complexité Spatiale:

Le contenu du programme principal est le suivant:
-Un tableau de taille N
-Une longueur N qui est une valeur entière
-Une variable i en entrée.

Pour la fonction tamiser nous avons:
-Un tableau de taille N
-Un noeud (entier)
-Deux variables entières.
-La taille du tableau qui est une valeur entière.

Si on suppose que la taille d'un octet est sur 4 octets on obtient la formule suivante:

Complexité spatiale(n)= 4(2N+6)= 8N+24 octets

La complexité spatiale est donc d'ordre N
Complexité spatiale(n)= O(N)

- **pire cas** correspond au cas ou le tableau est aléatoirement trie car dans ce cas il aura beaucoup de permutations
- **meilleur cas** correspond au cas ou le tableau est trié dans le bon ordre
- le **cas moyen** on a le tableau trié dans l'ordre inverse dans ce cas l'algo doit faire les tests et inverser

6.4. Mesure du temps d'exécution:

Taille du tableau	**10^4**	**5*10^4**	**10^5**	**5*10^5**	**10^6**	**5*10^6**	**10^7**	**5*10^7**	**10^8**
aléatoire	0.001	0.015	0.03	0.189	0.388	2.573	5.316	32.431	70.106
Par ordre	0.002	0.01	0.02	0.116	0.233	1.255	2.472	13.151	27.1
Ordre inverse	0.001	0.009	0.019	0.116	0.227	1.491	2.635	14.644	30.274

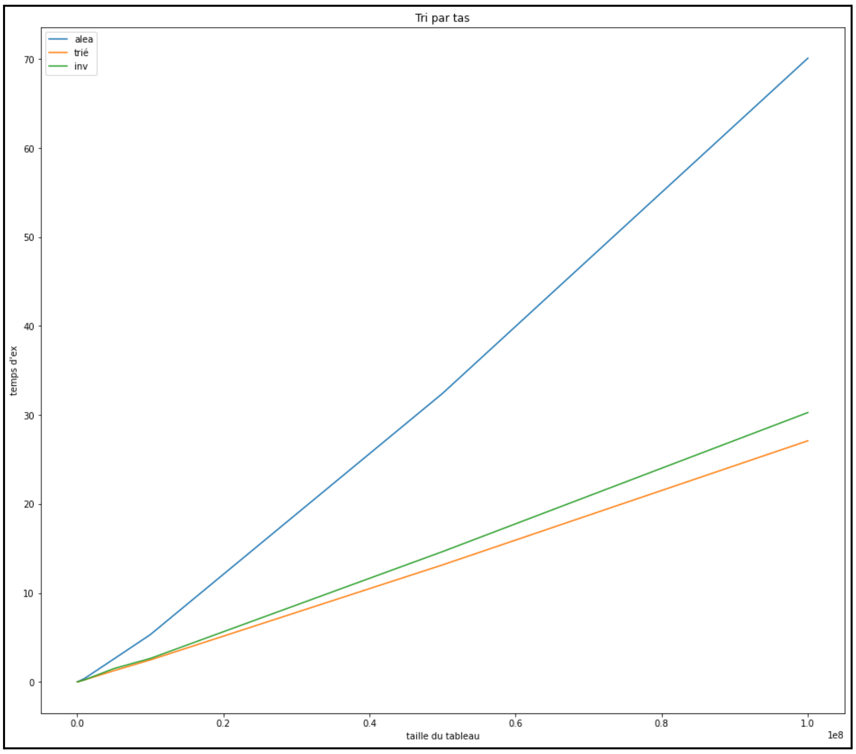


Figure 6: Graphe du temps d'exécution de l'algorithme du tri par tas

Pour ce qui est du dernier algorithme de tri qui est le tri par tas, le meilleur temps d'exécution correspond à celui du tableau des données triées dans le bon ordre suivi de près par le tableau des données inversé laissant ainsi le tableau des données aléatoire avec le plus grand temps d'exécution.

7. Le nombre de comparaisons d'éléments du tableau :

7.1 Tri par sélection

7.1.1 : Données triées en ordre croissant

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	49995000	1249975000	4999950000	12499975000	499999500000				

7.1.2 : Données non triées (aléatoire)

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	49995000	1249975000	4999950000	12499975000	499999500000				

7.2 Tri par insertion

7.2.1 : Données triées en ordre croissant

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	99995444	4999919981	9999942262	499999468621	999999				

7.2.3 : Données non triées (aléatoire)

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	25411228	681825154	2630446592	22146381140	46741112024				

7.3 Tri à bulle

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	49995000	1249975000	4999950000	124999750000					

7.4 Tri rapide

7.4.1 Choix du pivot 01 : dernier élément du tableau

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	13392	69012	148994	934464	1934464	19934464			

7.4.2 Choix du pivot 02 :Médiane du tableau

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	13512	72072	154666	934512	1934464	934500	19934464		

7.4.3 Choix du pivot 03 : Élément aléatoire du tableau

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	13512	99876	199902	999872					

7.5 Tri fusion

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	29998	149996	299999	1499986	2999978	14999920			

7.6 Tri par tas

7.6.1 : Données triées en ordre croissant

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	273914	1596610	3401710	19211402	40575586	226997834	473763418	2605377614	5404909606

7.6.2: Données non triées (aléatoire)

Taille du tableau	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7	$5*10^7$	10^8
Nombre de comparaison	258364	1526004	3276712	19105884	40401272	226789700	474062516	2606441880	5413700048

Conclusion:

D’après les tableaux contenant les différents résultats, ainsi que les graphes tracés à partir de ces derniers, nous pouvons établir les conclusions suivantes, soulignant que chacun des 6 algorithmes étudiés au long de ce tp ont leurs avantages ainsi que leurs inconvénients qui peuvent être listés de la sorte:

-Espace mémoire occupé
-Temps d'exécution
-Facilité d’implémentation

L’espace mémoire ne pose plus vraiment de problème ces temps-ci vu la capacité de stockage mémoire des nouvelles machines.

Pour la facilité d’implémentation les algorithmes d’insertion,de sélection et de tri à bulles sont relativement plus simples à implémenter par rapport aux autres qui demandent plus de connaissances et de maîtrise.

Pour ce qui est de la rapidité du temps d’exécution, les tri par insertion et par bulles ne sont pas très performants et ceci est justifié par leur complexités temporelles qui est élevée. Le tri par fusion est performant et donne de meilleurs résultats. Le tri rapide est comme son nom l’indique est rapide mais cela dépend du pivot choisi, si on se trompe de pivot on peut se retrouver avec un complexité temporelle aussi élevée que les algorithmes précédents.

Et pour finir l’algorithme du tri par tas est l’un des plus intéressant vu ses résultats remarquables de temps d’exécution.

Répartition des tâches:

BEHLOUL Hala Lyna: Réalisation de l’algorithme 6, Calcul des complexités, analyse des graphes et rédaction du rapport.

HENDEL Lyna Maria: Réalisation de l’algorithme 4, calcul des complexités,réalisation de temps d'exécution, rédaction du rapport.

NAAR Sarah : Réalisation de l’algorithme 1,2,3, calcul des complexités, réalisation de temps d'exécution et rédaction du rapport.

NADIR Somia: Réalisation de l’algorithme 5 et réalisation des graphes, nombre de comparaisons et rédaction du rapport