République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département d'intelligence artificielle

Rapport du Mini projet n°1

Module

Algorithmes Avancé et Complexité

Spécialité

Informatique Visuelle

Rapport réalisé par :

- **Etudiant :** SAYOUD Lynda

  Matricule : 191931031844

  Groupe : 01

- **Etudiant :** DRAI Said

  Matricule : 191931029353

  Groupe : 01

Partie I

Etude des méthodes de tri

I. Tri par Insertion :

1) Description

- ***Définition :***

Le tri par insertion (*insertion sort* en anglais) est un algorithme de tri par comparaison simple, et intuitif. C'est un algorithme de tri stable, en place, et le plus rapide en pratique sur une **entrée de petite taille** (autrement dit C'est un algorithme efficace quand il s'agit de trier un petit nombre d'éléments).

- ***Principe :***

Le principe du tri par insertion est de trier les éléments du tableau comme avec des cartes :

- On prend nos cartes mélangées dans notre main.
- On crée deux ensembles de carte, l'un correspond à l'ensemble de carte triée, l'autre contient l'ensemble des cartes restantes (non triées).
- On prend au fur et à mesure, une carte dans l'ensemble non trié et on l'insère à sa bonne place dans l'ensemble de carte triée.
- On répète cette opération tant qu'il y a des cartes dans l'ensemble non trié.

2) Algorithme itératif

```
Algorithme Tri_insertion ;
VARIABLE
t : tableau [0..N-1]d'entiers ;
i,j,k,N: entier;
DEBUT
j←1 ;
tant que (j<N)
faire
        i←j
        k←t[j]
        tant que (i>0 et t[i-1]>k)
        faire
```

```
                t[i]←t[i-1] ;
                 i←i-1 ;
        fait ;
        t[i]←k ;
        j←j+1 ;
fait ;
FIN
```

3) Complexité temporelle

- ***Meilleur cas :***

Dans le meilleur des cas, avec des données déjà triées, l'algorithme effectuera seulement *n* comparaisons car l'algorithme ne rentre jamais dans la 2 -ème boucle tant que, ainsi il y a N-1 comparaisons et au plus N affectations. Sa complexité dans le meilleur des cas est donc en $\Theta(n)$.

$$\mathbf{CT(n)= \Theta(n).}$$

- ***Pire des cas :***

Dans le pire des cas, avec des données triées en ordres inverse, les parcours successifs du tableau imposent d'effectuer *(n-1)+(n-2)+(n-3)..+1* comparaisons et échanges. On a donc une complexité dans le pire des cas du tri par insertion en $\Theta(n^2)$.

$$\mathbf{CT(n)= \Theta(n^2).}$$

- ***Complexité Moyenne : (Cas exacte)***

Si tous les éléments de la série à trier sont distincts et que toutes leurs permutations sont équiprobables, la complexité en moyenne de l'algorithme est de l'ordre de $(n^2-n)/4$ comparaisons et échanges. La complexité en moyenne du tri par insertion est donc également en $\Theta(n^2)$

$$\mathbf{CT(n)= \Theta(n^2)}$$

4) Complexité Spatiale :

L'algorithme de tri par insertion utilise le tableau en question (de taille N) à trier ainsi que 5 variables entiers et il n'alloue aucune autre case mémoire. On suppose que la taille d'une donnée de type scalaire (entier, réel, logique ou caractère) occupe un mot mémoire sur 4 Octets.

On a : 5 variables scalaires i, j, k, N et t de type entier.

Donc : $\boldsymbol{D = 5\ données}$.

En conséquence, la complexité spatiale $CS(n)$ de l'algorithme Tri_insertion est :

1- En notation exacte : $CS(n) = 4(N + 5)$ mots mémoires.
2- En notation asymptotique : $CS(n)= \Theta(N)$.

5) L'algorithme en C :

```
double InsertionSort(int T[], int N)
{
     int j,i,x;
     double temps_exe;

     clock_t start=clock();
        j=1;
     while(j<N)
      {
       x=T[j];
       i=j;

       while(i>0 && T[i-1]>x)
       {
        T[i]=T[i-1];
              i=i-1;
             }
             T[i]=x;
             j++;
         }

    clock_t end= clock();
    temps_exe = (double)(end-start)/CLOCKS_PER_SEC;

    return temps_exe;
}
```

6) Les temps d'exécution T :

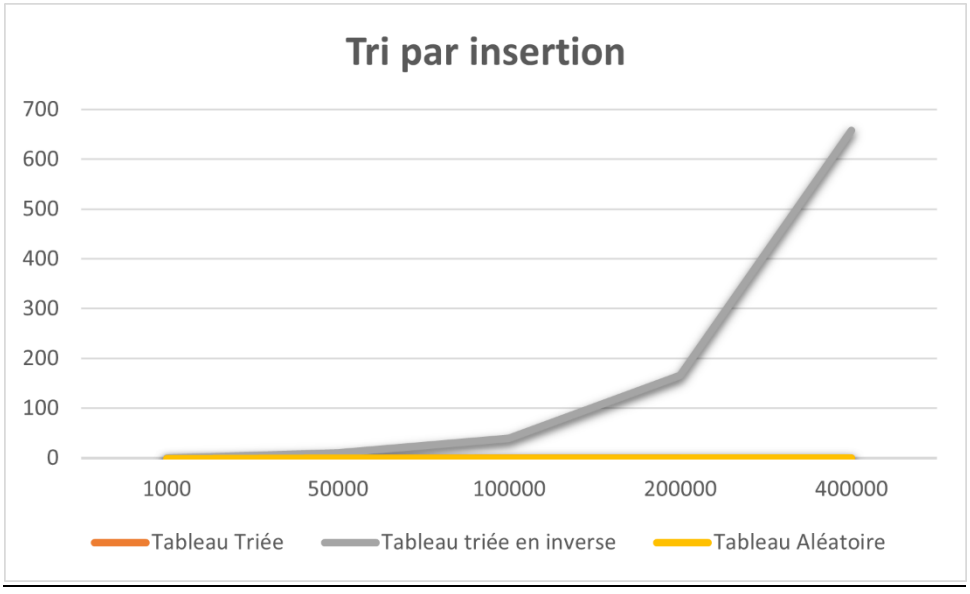
On complète le tableau suivant :

n	10^3	$5*10^4$	10^5	$2*10^5$	$4*10^5$	$8*10^5$	$1.6*10^6$	$3.2*10^6$	$6.4*10^6$
T (triée)	0.000000	0.000000	0.000000	0.001000	0.001000	0.002000	0.009000	0.012184	0.033611
T (Triée inversement)	0.001000	2.401000	9.651000	39.569000	165.401000	657.298000	10123.366000	22003.910645	86713.329
T (aléatoires)	0.002001	0.004507	0.008967	0.019399	0.043008	0.043008	0.086016	0.168591	0.357413

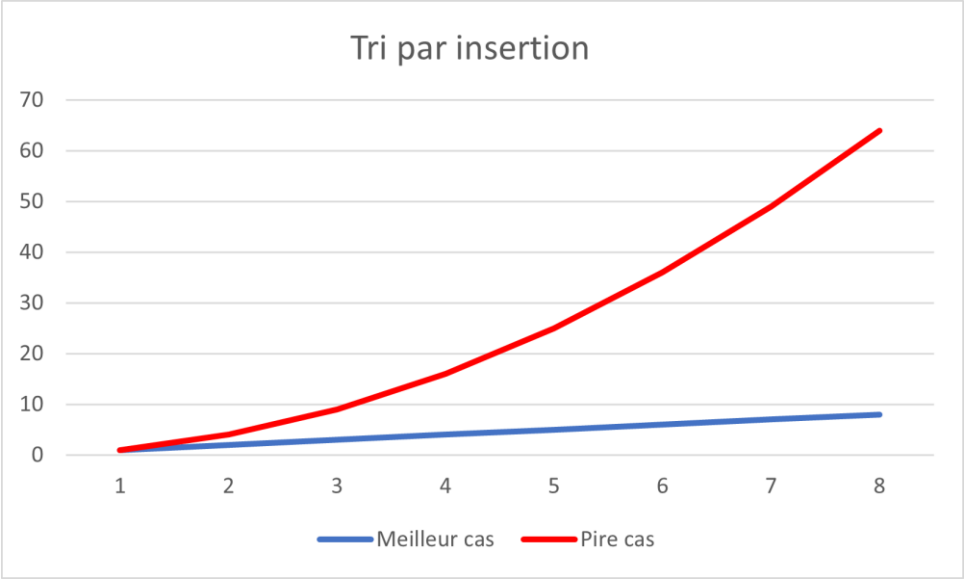
n	$12.8*10^6$	$25.6*10^6$	$51.2*10^6$	$1024*10^6$	$2048*10^6$
T(triée)	0.073272	0.168525	0.370756	7,4	152,46
T(Triée inversement)	346853.319	1508811.938	6789653.719	18000000	18000000
T(aléatoires)	0.756994	1.65246	3.795567	76,21	152,46

7) Représentation graphique des variations du temps d'exécution mesuré T(n) et les variations des complexités théoriques :

1. Graphe des variations des temps d'exécution selon la taille N du tableau :



2. Graphe des fonctions de la complexité théorique au ***Pire*** et ***Meilleur Cas*** :



8) Comparaison entre la complexité théorique & la complexité expérimentale

On remarque que les temps d'exécution sont approximativement multipliés par 4 lorsque N est doublé pour :

Les tableaux en ordre inverse.

Exemples :

N1 = 200000 → T1 =39.569000

N2 = 400000 = 2 * N1 → T2 = 165.401000≈ 4 * T1 = 2^2 * T1

On en déduit que le temps d'exécution est proportionnel à N, ce que l'on peut représenter par la formule suivante :

T1(x * N) = x^2 * T1(N)

x :étant la tangente d'un point sur le graphe.

On constate aussi que lorsque la taille N du tableau lorsque les données sont triées est double, le temps d'exécution est lui aussi à peu près doublé.

Exemples :

N1 = 400000 → T1 =0.001000

N2 = 800000 = 2 * N1 → T2 = 0.002000≈ 2 * T1

On en déduit que le temps d'exécution est proportionnel à N, ce que l'on peut représenter par la formule suivante :

T2(x * N) = x * T2(N)

Ainsi nos données expérimentales de la complexité suivent une des deux fonctions T1 et T2.

La complexité théorique au pire cas est de l'ordre de :

$$CTP(n)= \Theta(n^2).$$

La complexité théorique au meilleur cas est de l'ordre de :

$$CTM(n) = \Theta(n).$$

Et donc :

$$CTM(n) \leq T1(n) \leq CTP(n) \Leftrightarrow n \leq 2* n \leq n^2$$

On en conclue que : Le modèle théorique est conforme avec les mesures expérimentales .

II. Tri à Bulles :

1) Description :

- ***Principe :***

Le principe du tri à bulles (bubble sort ou sinking sort) est de comparer deux à deux les éléments e1 et e2 consécutifs d'un tableau et d'effecteur une permutation si e1 > e2. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

2) Algorithme itératif :

```
Algorithme Tri_a_Bulles ;
i , j , n, Temp :  Entier ;
T:  Tableau [0 ..N-1] de entiers ;
TRI : Boolean ;
Début
TRI←Faux ;
Tant que(TRI=Faux)
Faire
      TRI←Vrai ;
      Pour i de 0 à N-2
      Faire
            Si(T[i] > T[i+1]) alors
                  Temp←T[i] ;
                  T[i]←T[i+1] ;
                  T[i+1]←Temp ;
```

```
                    TRI ←Faux ;
               Fsi ;
        Fait ;
        N←N-1 ;
Fait ;
Fin.
```

3) Complexité temporelle :

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

- ***Meilleur cas :***

C'est lorsque les données d'un tableau sont déjà triées, le programme effectue N^2 comparaison et aucune permutation, ainsi sa complexité est : $CT(n) = n^2$ tests. Et Donc :

$$\mathbf{CT(n)=\Theta(n^2)}$$

- ***Pire des cas :***

C'est dans le cas où les données sont en ordre inverse, les parcours successifs du tableau imposent d'effectuer $(n^2-n)/2$ comparaisons et échanges. Et donc :

$$\mathbf{CT(n)=\Theta(n^2)}$$

- ***Complexité Moyenne (Exacte) :***

Si tous les éléments de la série à trier sont distincts et que toutes leurs permutations sont équiprobables, la complexité en moyenne de l'algorithme est de l'ordre de $(n^2-n)/4$ comparaisons et échanges.

$$\mathbf{CT(n)}=\Theta(n^2)$$

4) Complexité Spatiale :

On a :

Un tableau de N éléments et 4 variables entières et 1 booléenne.

On suppose que la taille d'une donnée scalaire est sur 4 Octets nous obtenant : CS(n) = 4(N +

Octets = 4N + 20 octets.

En notation asymptotique :

$$\mathbf{CS(n)}=\Theta(n)$$

5) Algorithme en C :

```
double bubble_sorted(int T[],int N)
{int j,i,x,TRI;
   double temps_exe;
   clock_t start=clock();
        TRI=0;
        while(TRI==0)
        {
                TRI=1;
                for(i=1;i<=N-1;i++)
                {
                        if(T[i]>T[i+1])
                        {
                                x=T[i];
                                T[i]=T[i+1];
                                T[i+1]=x;
                                TRI=0;
                        }
                }
                N=N-1;
        }
  clock_t end= clock();
   temps_exe = (double)(end-start)/CLOCKS_PER_SEC;

   return temps_exe;
}
```

6) Les temps d'exécution T :

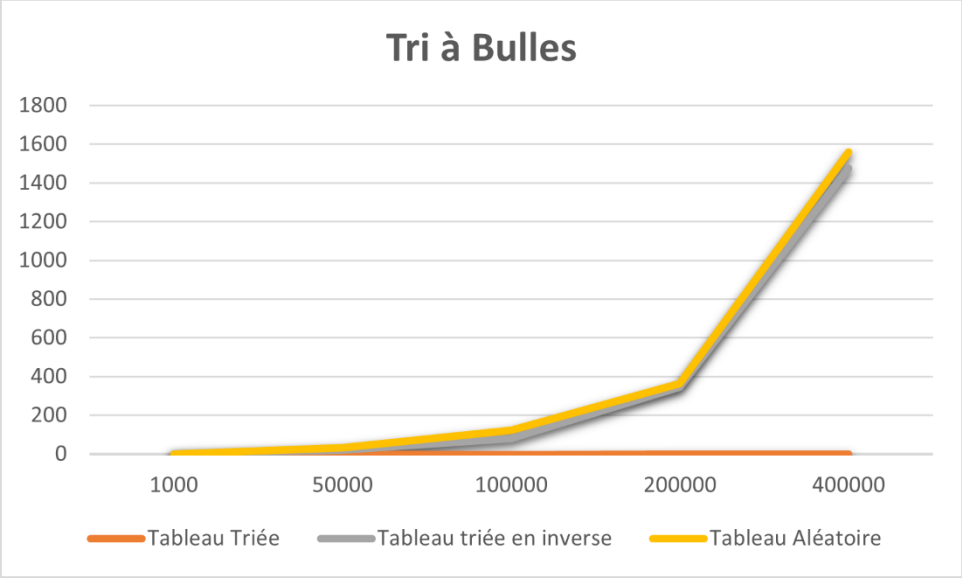
On complète le tableau suivant :

n	10^3	$5*10^4$	10^5	$2*10^5$	$4*10^5$	$8*10^5$	$1.6*10^6$	$3.2*10^6$	$6.4*10^6$
T (triée)	0.000000	0.000000	0.001000	0.004000	0.016000	0.065	0.257	1.024	4.097
T (Triée inversement)	0.002000	5.058000	20.004000	80.463000	344.38164	1473.954	6308.5206	27000.468	115563.88
T (aléatoires)	0.002000	7.568000	30.641000	123.209000	364.559000	1560.31252	6678.137	28582.43	122332.79

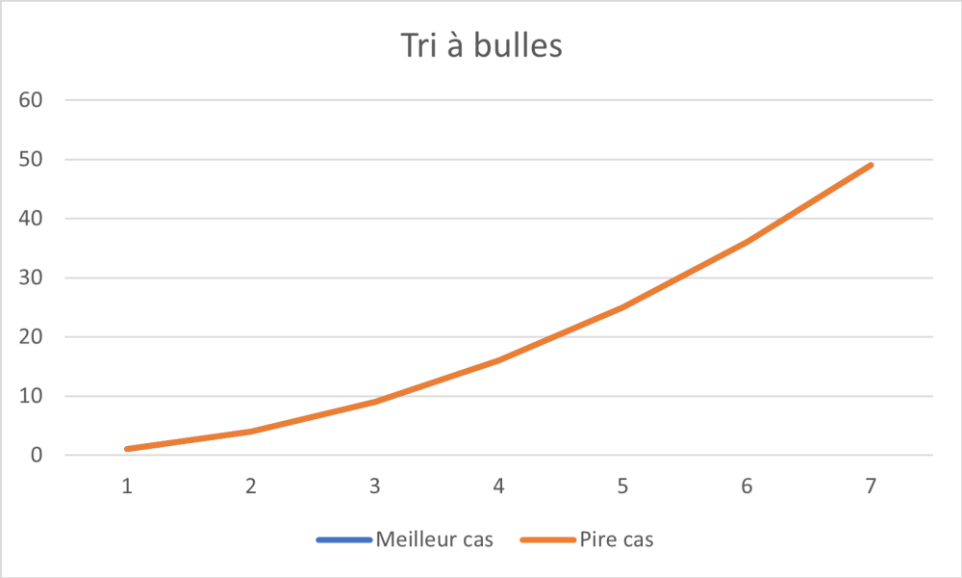
n	$12.8*10^6$	$25.6*10^6$	$51.2*10^6$	$1.024*10^6$	$2.048*10^6$
T(triée)	16.389	65.536	262.144	1048.576	4195.13
T(Triée inversement)	495760.9988	2126814.685	2126816.685	38959842.18	165631336.9
T(aléatoires)	518691.0531	2219997.707	9501590.188	40666806	174053939.7

7) Représentation graphique des variations du temps d'exécution mesuré T(n) et les variations des complexités théoriques :

1. Graphe des variations des temps d'exécution selon la taille N du tableau :



2. Graphe des fonctions de la complexité théorique au Pire et Meilleur Cas :



8) Comparaison:

On remarque que les temps d'exécution évoluent de manière quadratique, ce qui signifie que si par exemple nous doublons le nombre de N en entrée, le temps d'exécution va être multiplié par 4.

Le temps d'exécution est donc proportionnel à N, et nous pouvons le représenter par la formule suivante :

$$\mathbf{T(x * N) = x^2 * T(N)}$$

Exemple : N1 = 50000 →T1 = 7.568000

N2 = 100000 = 2 * N1 → T2 = 30.641000 ≈ 4 * T1 = 2^2 * T1.

Remarque : On constate que quel que soit l'ordre des éléments du tableau (ordonnées, inverses ou aléatoires), la complexité est toujours quadratique .

III. Tri Fusion:

1) Description :

- ***Définition :***

Le tri par fusion est l'un des algorithmes de tri les plus populaires et les plus efficaces. Et il est basé sur le paradigme Diviser pour régner. Ce dernier repose sur 3 étapes :

- **DIVISER** : le problème d'origine est divisé en un certain nombre de sous-problèmes
- **RÉGNER** : on résout les sous-problèmes (les sous-problèmes sont plus faciles à résoudre que le problème d'origine)
- **COMBINER** : les solutions des sous-problèmes sont combinées afin d'obtenir la solution du problème d'origine.
- ***Principe :***

Son principe s'appuie sur la méthode deviser pour régner définit ci-dessus. L'avantage de tri à fusion est que les deux listes sont fusionnées en même temps. L'algorithme est connu pour son efficacité en complexité (temps et mémoire) et pour trier les listes :

- On coupe les données en deux parties égales.
- On trie les données de chaque partie (on découpe chaque partie, l'algorithme devient récursif).
- On fusionne les deux parties.

On devise récursivement le tableau de départ en deux sous tableaux jusqu'à ce que le tableau ne contient qu'un seul élément. Une fois que les éléments triés indépendamment les uns des autres. A partir de ce moment le backtracking (Retour sur trace) commence et on va fusionner les sous tableau en un seul jusqu'à obtenir le tableau de départ trié. La fusion consiste en des comparaisons successives.

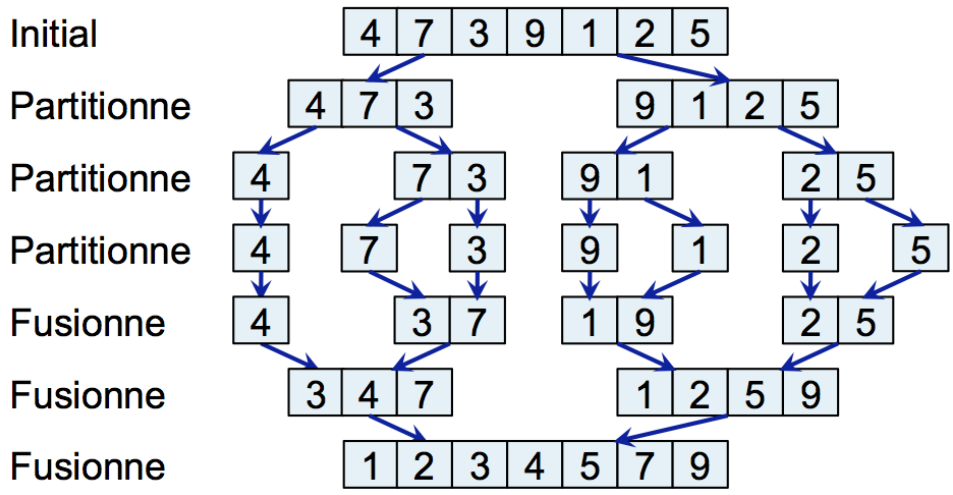


Figure-1:Exemple Tri_Fusion

2) Algorithme itératif :

```
ALGORITHME Tri_Fusion ;
Variables
 T: tableau [0..N-1] de entiers ;
 gauche,droite: entier;
 centre: entier;
DEBUT
   SI (gauche < droite) ALORS
      //Trouvons le milieu(centre) d'abord
       centre ← (gauche + droite) / 2;

      TriFusion (T, gauche, centre);

       TriFusion (T, centre + 1, droite);
       FUSIONNER (T, gauche, centre, droite);//Fusionner est une fct
   FSI
FIN.
```

3) Complexité Temporelle:

Le tri par fusion est un algorithme récursif et la complexité temporelle peut être exprimée comme une relation de récurrence.

$CT(n)=2CT(n/2)+\Theta(n)$

Et donc :

$$\boldsymbol{CT(n)=\Theta(n.log_2\,(n))}$$

La complexité temporelle du tri par fusion est Θ(n.log(n)) dans les **3 cas (pire, moyen et meilleur)** car le tri par fusion divise toujours le tableau en deux moitiés et prend un temps linéaire pour fusionner deux moitiés.

4) Complexité Spatiale :

L'algorithme de tri fusion n'utilise que le tableau de N éléments (à trier) ainsi que 3 variables entiers. on suppose que la taille d'un entier est sur 4 Octets. Nous obtenant la complexité spatiale :

CS(N) = 4(N + 3) octets = 4N + 12 octets

Ainsi :

$$\boldsymbol{CS(n) = \Theta\,(n)}$$

5) L’algorithme en C:

```
void fusionner(int tab[],int deb1,int fin1,int fin2)
{
   int *table1;
   int deb2=fin1+1;
   int compt1=deb1;
   int compt2=deb2;
   int i;
   int A=fin1-deb1+1;
   table1=(int*)malloc(A*sizeof(int));

       for(i=deb1;i<=fin1;i++)
           {
           table1[i-deb1]=tab[i];
           }

       for(i=deb1;i<=fin2;i++)
           {
           if (compt1==deb2)
               {
               break;
               }
           else if (compt2==(fin2+1))
               {
               tab[i]=table1[compt1-deb1];
               compt1++;
               }
           else if (table1[compt1-deb1]<tab[compt2])
               {
               tab[i]=table1[compt1-deb1];
               compt1++;
               }
           else
               {
               tab[i]=tab[compt2];
               compt2++;
               }
           }
       free(table1);
}


void fusionsort_1(int tableau[],int deb,int fin)
       {
       if (deb!=fin)
           {
           int milieu=(fin+deb)/2;
           fusionsort_1(tableau,deb,milieu);
           fusionsort_1(tableau,milieu+1,fin);
           fusionner(tableau,deb,milieu,fin);
           }
       }

double FusionSort(int tab[],int N)
```

```
{
   double temps_exe;
     clock_t start=clock();
   if (N>0)
       {
       fusionsort_1(tab,0,N-1);
       }
  clock_t end= clock();
    temps_exe = (double)(end-start)/CLOCKS_PER_SEC;

    return temps_exe;
}
```

6) Les temps d'exécution T:

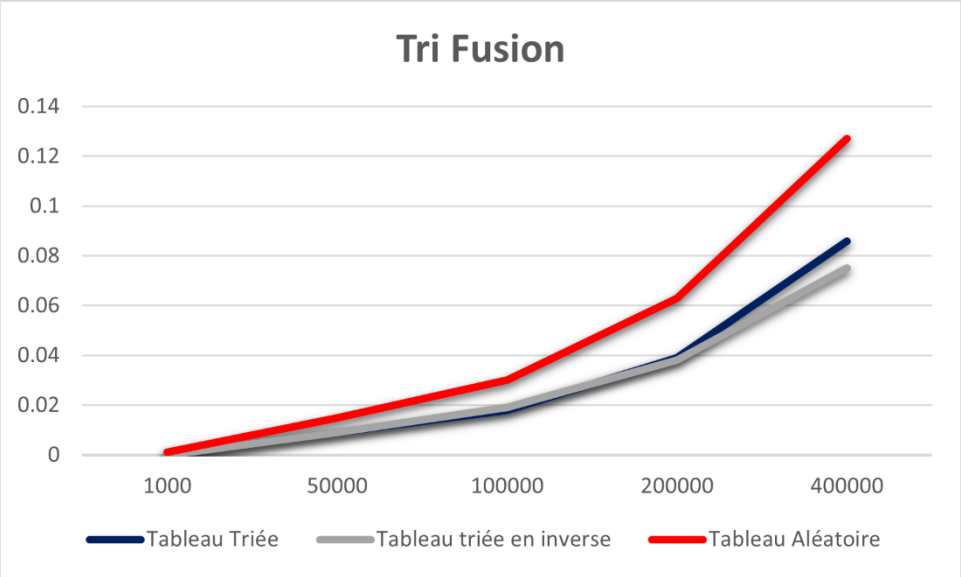
On complète le tableau :

n	10^3	$5*10^4$	10^5	$2*10^5$	$4*10^5$	$8*10^5$	$1.6*10^6$	$3.2*10^6$	$6.4*10^6$
T (triée)	0.000000	0.005000	0.009000	0.018000	0.039000	0.0858	0.18876	0.41527	0.91359
T (Triée inversement)	0.000000	0.005000	0.009000	0.019000	0.038000	0.07500	0.15200	0.30400	0.609
T (aléatoires)	0.001000	0.007000	0.015000	0.030000	0.063000	0.127	0.252	0.506	1.008

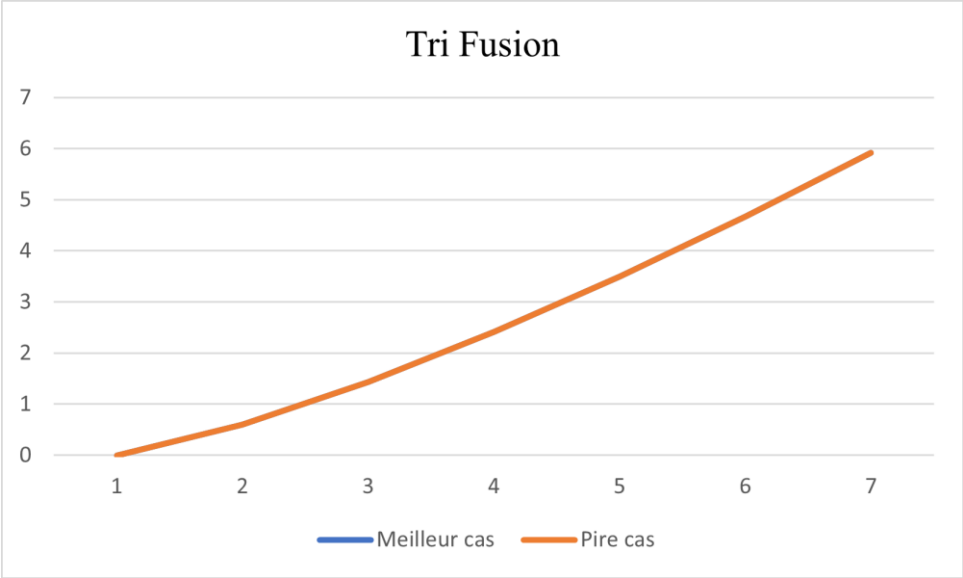
n	$12.8*10^6$	$25.6*10^6$	$51.2*10^6$	$1.024*10^6$	$2.048*10^6$
T(triée)	2.009	4.4218	8.843	21.29	42.4494
T(Triée inversement)	1.218	2.437	4.864	9.788	19.655
T(aléatoires)	2.019	4.038	8.76	16.152	32.420

7) Représentation graphique :

1. Graphe des variations des temps d'exécution selon la taille N du tableau :



2. Graphe des fonctions de la complexité théorique au Pire et Meilleur Cas :



8) Comparaison:

On remarque que les temps d'exécution sont approximativement multipliés par 2 lorsque N est doublé pour : les tableaux triée ,en ordre inverse et même aléatoires.
Exemples :
N1 = 50000 → T1 = 0.005000
N2 = 100000 = 2 * N1 → T2 = 0.009000 ≈ 2 * T1

IV. Tri Rapide(Quicksort):

1) Description :

- ***Définition :***

Le tri rapide - aussi appelé "tri de Hoare" (du nom de son inventeur Tony Hoare) ou "tri par segmentation", en anglais "**quicksort**", - est certainement l'algorithme de tri interne le plus efficace. Il est un algorithme de tri par comparaison, son fonctionnement est plutôt simple à comprendre et il est très utilisé sur de **grandes entrées**.

- ***Principe :***

Le tri rapide utilise le principe de *diviser pour régner*, c'est-à-dire que l'on va choisir un élément du tableau (qu'on appelle **pivot**), puis l'on réorganise le tableau initial en deux sous tableaux :

- L'un contenant les éléments inférieurs au pivot.
- L'autre contenant les éléments supérieurs au pivot.

On continue ce procédé (qu'on appelle **partitionnement**, c'est-à-dire choisir un pivot et réorganiser le tableau) jusqu'à se retrouver avec un tableau découpé en NN sous tableaux (N étant la taille du tableau), qui est donc trié.

2) Algorithme récursif :

```
Global:Tab :Tableau[min..max] tableau d'entier ;
```

```
Fonction Partition
Entrée : Gauche, Droite : Entier ;
Résultat : entier
Local: i, j, pivot, temp: Entier;
Sortie: Résultat :entier ;
début
     pivot←Tab[Droite];
     i← Gauche-1;
     j ←Droite;
     Répeter
        Répeter  i ← i+1
        Jusqu'à (Tab[i] >= pivot) ;

        Répéter j ← j-1
       Jusqu'à (Tab[j] <= pivot);
        Temp←Tab[i];
```

```
        Tab[i] ←Tab[j];
        Tab[j] ←temp
     Jusqu'à (j <= i);
     Tab[j] ← Tab[i];
     Tab[i] ← Tab[Droite];
     Tab[Droite] ← temp;
     Résultat←i
Fin ;

Algorithme TriRapide
Gauche,Droite :Entier ;
i : entier ;
Début
      si Droite > Gauche alors
            i ← Partition( Gauche , Droite );
            TriRapide(Gauche , i-1 );
            TriRapide( i+1 , Droite );
      Fsi
Fin.
```

3) Complexité Temporelle :

- ***Meilleur cas :***

  Le calcul de la complexité du tri rapide est très semblable à celui du tri par fusion, mais au lieu de fusionner nos sous tableaux, on les réorganise (cette opération est de nouveau en temps linéaire, comme pour la fusion de deux sous tableaux), on retrouve aussi les deux appels récursifs qui divisent par deux notre tableau actuel. La complexité est donc calculée de la même façon, et on se retrouve avec un résultat en :

$$\mathbf{CT(n)=\Theta(n.\log_2(n))}$$

- ***Pire des cas :***

  Lorsque à chaque niveau de la récursivité le découpage conduit à trier un sous-tableau de 1 élément et un sous-tableau contenant tout le reste.
  Et donc :

$$\mathbf{CT(n)= \Theta(}n^2\mathbf{).}$$

- ***Complexité Moyenne : (Cas exacte)***

$$\mathbf{CT(n)= \Theta(n.\log_2(n))}$$

4) Complexité Spatiale :

$CS(n) = \Theta(n)$

5) Algorithme en C :

```
void permuter (int* a, int* b)
    {
        int tmp;
        tmp=*a;
        *a=*b;
        *b=tmp;
    }

double quickSort(int T[], int N)
{
   double temps;
   clock_t start, stop;

   int min,max,i,pivot,p_pivot;
   Pile P;

   start = clock();
   CreerPile (&P);
   Empiler(&P,0);
   Empiler (&P,N-1);

   while (!PileVide(P))
   {
       Depiler (&P, &max);
       Depiler (&P, &min);
       pivot=T[max];
       p_pivot=min;
       for (i=min; i<=max-1; i++)
       {
           if (T[i]<=pivot)
           {
               permuter(&T[i],&T[p_pivot]);
               p_pivot++;
           }
       }
       permuter(&T[p_pivot], &T[max]);
       if (p_pivot+1 <max)
       {
           Empiler(&P,p_pivot+1);
           Empiler(&P,max);
       }
       if (p_pivot-1>min)
       {
           Empiler(&P,min);
           Empiler(&P,p_pivot-1);
       }
   }
   stop = clock();
```

```
temps=(double)(stop-start)/CLOCKS_PER_SEC;
return temps;
 }
```

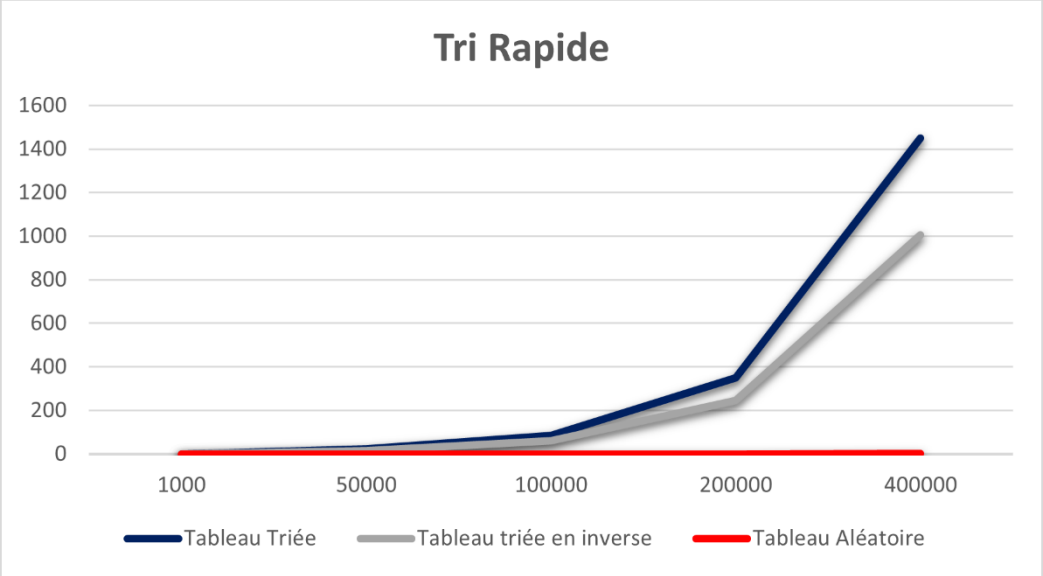
6) Les temps d'exécution T :

n	10^3	$5*10^4$	10^5	$2*10^5$	$4*10^5$	$8*10^5$	$1.6*10^6$	$3.2*10^6$	$6.4*10^6$
T (triée)	0.000000	5.299000	21.911000	84.723000	348.391000	1449.306	6000.129	24960.53	103336.62
T (Triée inversement)	0.000000	3.697000	15.101000	60.631000	245.286000	1005.67	4123.25766	16905.35641	67621.42562
T (aléatoires)	0.000000	0.009000	0.021000	0.049000	0.109000	0.2398	0.5323	1.1819	2.672

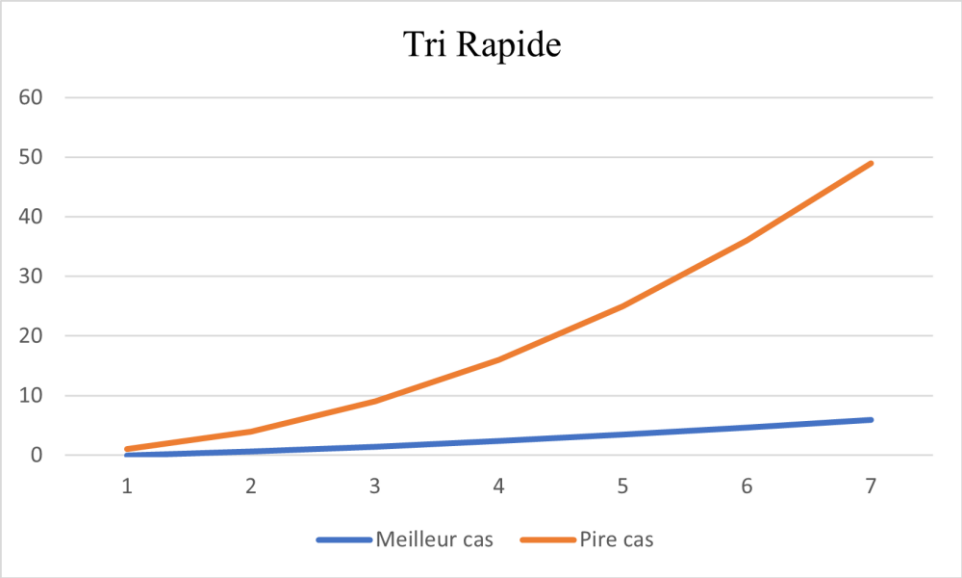
n	$12.8*10^6$	$25.6*10^6$	$51.2*10^6$	$1.024*10^6$	$2.048*10^6$
T(triée)	427813.6251	1771148.408	7332554.409	……	…..
T(Triée inversement)	270485.7	1081945.9	4544159.80	……	…..
T(aléatoires)	5.85507	12.2866	28.2534	…….	……

7) Les graphes :

1. Graphe des variations des temps d'exécution selon la taille N du tableau :



2. Graphe des fonctions de la complexité théorique au Pire et Meilleur Cas :



8) Comparaison :

On remarque que les temps d'exécution sont approximativement multipliés par 4 lorsque N est doublé pour : les tableaux triées et triées en ordre inverse.

Exemples :

N1 = 50000 → T1 = 5.299000

N2 = 100000 = 2 * N1 → T2 = 21.911000≈ 4 * T1 = 2^2 * T1

On constate aussi que lorsque la taille N du tableau en ordre aléatoires est doublée, le temps d'exécution est lui aussi à peu près doublé.

Exemples : N1 = 50000 → T1 = 0.009000

N2 = 100000 = 2 * N1 → T2 = 0.021000≈ 2 * T1

V. Tri par Tas :

1) Description :

- ***Définition :***

Le tri par tas (***heapsort*** en anglais) est un des tris optimaux par comparaison. Le tri par tas utilise une structure de données pour trier en place et de manière optimale. C'est un exemple de la conception d'un algorithme à l'aide (et autours) d'une structure de données.

- ***Principe :***

1- Construisez un tas maximum à partir des données de telle sorte que la racine soit l'élément le plus élevé du tas.
2- Supprimez la racine, c'est-à-dire l'élément le plus élevé du tas et remplacez-le ou échangez-le avec le dernier élément du tas.
3- Ajustez ensuite le tas max, afin de ne pas violer les propriétés max heap (heapify). L'étape ci-dessus réduit la taille du tas de 1.
4- Répétez les trois étapes ci-dessus jusqu'à ce que la taille du tas soit réduite à 1.

2) Algorithme itératif :

Procédure tri_tas (arbre []: d ' entier , longueur : entier)

VAR

i : entier ;

DEBUT

Pour i ←longueur /2 **à** 1

Faire

tamiser (arbre , i , longueur) ;

Fait ;

Pour i ← longueur **à** 2

Faire

Permuter (arbre [1] , arbre [i]) ;

tamiser (arbre , 1 , i -1) ;

Fait ;

FIN.

Procédure tamiser (arbre , noeud , n)

```
VAR
 k := noeud
 j := 2* k
DEBUT
        Tant que j <= n
        Faire   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 .
```

3) Complexité Temporelle :

- ***Meilleur cas :***

  CT(n)=Θ(n.log(n))

- ***Pire des cas :***

  CT(n)=Θ(n.log(n))

4) Complexité Spatiale :

Nous avons :

-Le tableau de taille N.

-Sa longueur n qui est un entier.

-Une variable i entière.

Dans la fonction tamiser nous avons :

-Notre tableau de taille N

-Un nœud qui est un entier. -la taille du tableau qui est un entier. -Deux variables de type entier j et k. on suppose que la taille d'un entier est sur 4 Octets, nous obtenant :

CS(n) = 4(2N + 6) Octets = 8N + 24 octets.

Ainsi : **CS(n) = Θ (n)**

5) L'algorithme en C:

```
void MakeHeap(int arr[], int n, int i)
{
   // Find largest among root, left child and right child
   int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
     largest = left;
    if (right < n && arr[right] > arr[largest])
     largest = right;
    // Swap and continue heapifying if root is not largest
   if (largest != i) {
     permuter(&arr[i], &arr[largest]);
     MakeHeap(arr, n, largest);
   }
 }

double HeapSort(int arr[], int n)
 {
   double temps;
   clock_t start, stop;
   start = clock();
   // Build max heap
   for (int i = n / 2 - 1; i >= 0; i--)
     MakeHeap(arr, n, i);
    // Heap sort
   for (int i = n - 1; i >= 0; i--)
    {
       permuter(&arr[0], &arr[i]);
      // Heapify root element to get highest element at root again
     MakeHeap(arr, i, 0);
```

```
  }
  stop =clock();
  temps = (double)(stop-start)/CLOCKS_PER_SEC;
  return temps;
}
```

6) Temps d'exécution :

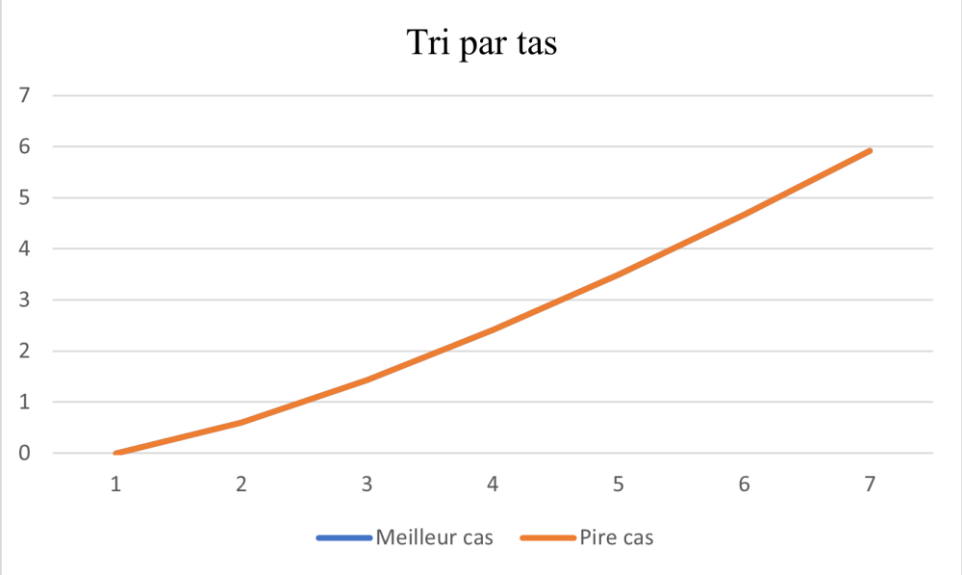
n	10^3	$5*10^4$	10^5	$2*10^5$	$4*10^5$	$8*10^5$	$1.6*10^6$	$3.2*10^6$	$6.4*10^6$
T (triée)	0.000000	0.007000	0.016000	0.033000	0.0621	0.12482	0.2546	0.5118	1.02336
T (Triée inversement)	0.000000	0.006000	0.015000	0.031000	0.06386	0.1315	0.2709	0.558	1.149
T (aléatoires)	0.000000	0.009000	0.021000	0.044000	0.0919	0.1921	0.4016	0.8395	1.754

n	$12.8*10^6$	$25.6*10^6$	$51.2*10^6$	$1024*10^6$	$2048*10^6$
T(triée)	2.0770	4.1979	8.4638	19.335	44.47
T(Triée inversement)	2.368	4.878	10.04	20.7	42.646
T(aléatoires)	3.6671	7.6643	16.0186	33.987	71.03281

7) Les graphes :

1. Graphe des variations des temps d'exécution selon la taille N du tableau :

2. Graphe des fonctions de la complexité théorique au Pire et Meilleur Cas :



8) Comparaison :

Nous remarquons que les temps d'exécution évoluent de manière logarithmique, ce qui signifie que si par exemple nous doublons le nombre de N en entrée, le temps d'exécution se voit aussi doublé.
Exemple :
N1 = 50000 → T1 = 0.007000
N2 = 100000 = 2 * N1 → T2 = 0.016000≈ 2 * T1.

Et on constate que cela est quel que soit l'ordre des éléments du tableau (ordonnées, inverses ou aléatoires), la complexité est toujours 2*

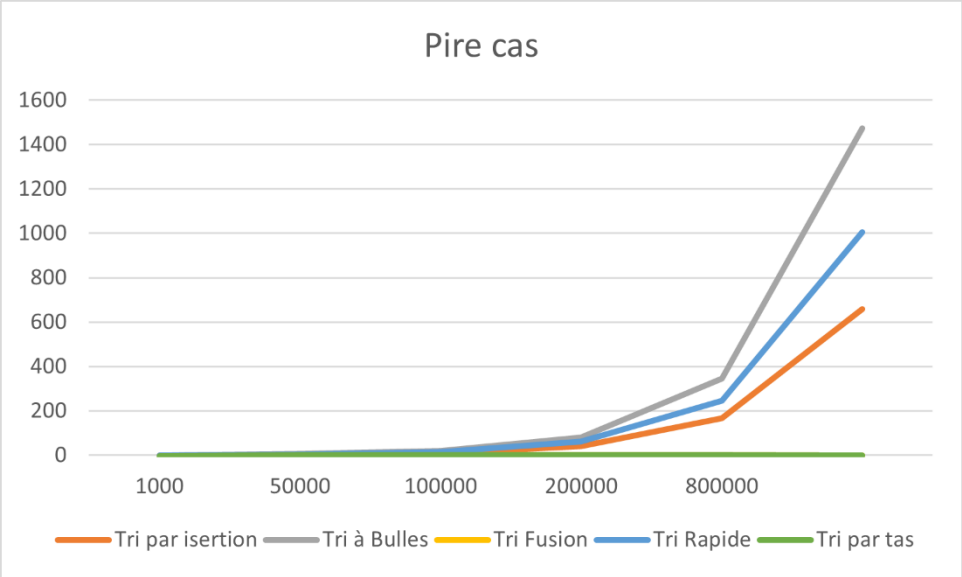
Partie II

Comparaison des méthodes de tri

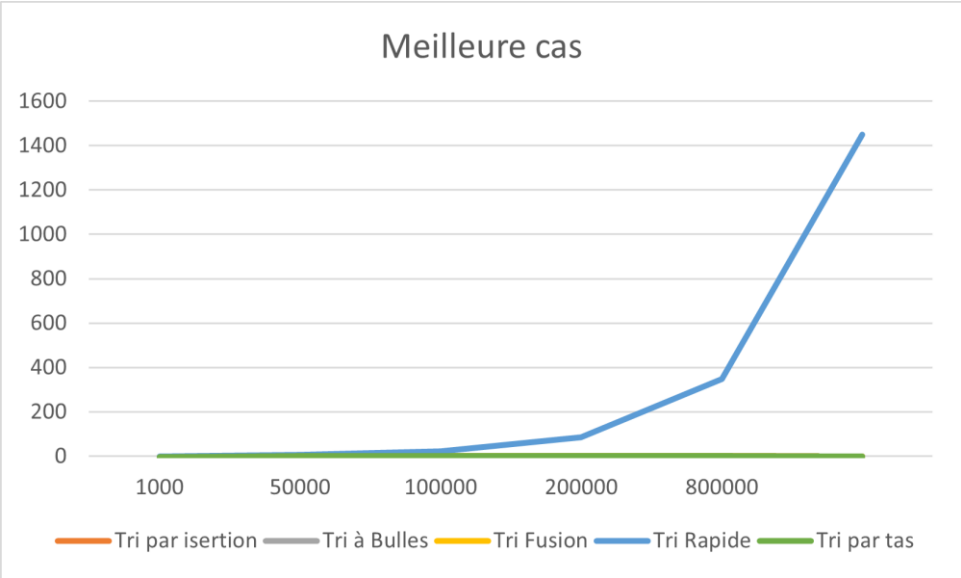
1.Les graphes des complexités théoriques et expérimentales des 5 algorithmes de tri.

1 -Complexité théorique :

-Comparaison de la Complexité théorique au pire cas pour les cinq (5) algorithmes



-Comparaison de la Complexité théorique au meilleur cas pour les cinq (5) algorithmes



2 -Complexité expérimentale :

2-Tableau comparatif des complexités théoriques pour les 5 algorithmes:

1 -Complexité théorique :

Algorithme	Complexité au pire cas	Complexité au meilleur cas
Tri par insertion	N^2	N
Tri à Bulles	N^2	N^2
Tri Fusion	$N.\log_2(N)$	$N.\log_2(N)$
Tri rapide	N^2	$N.\log_2(N)$
Tri par tas	$N.\log_2(N)$	$N.\log_2(N)$

2-Temps d'exécution :

1)Meilleur cas :

N	10^3	$5*10^4$	10^5	$2*10^5$	$4*10^5$	$8*10^5$
Tri par insertion	0	0	0	0.001	0.001	0.002
Tri à Bulles	0	0	0.001	0.004	0.016	0.065
Tri Fusion	0	0.005	0.009	0.018	0.039	0.0858
Tri rapide	0	5.299	21.911	84.723	348.391	1449.306
Tri par tas	0	0.007	0.016	0.033	0.0621	0.12482

2) Pire cas :

N	10^3	$5*10^4$	10^5	$2*10^5$	$4*10^5$	$8*10^5$
Tri par insertion	0.001	2.401	9.651	39.569	165.401	657.298
Tri à Bulles	0.002	5.058	20.004	80.463	344.38164	1473.954
Tri Fusion	0	0.005	0.009	0.019	0.038	0.075
Tri rapide	0	3.697	15.101	60.631	245.286	1005.67
Tri par tas	0	0.006	0.015	0.031	0.06386	0.1315

3-Comparaison entre les 5 algorithmes de tri :

D'après, l'étude comparative des graphes et tableaux des algorithmes de tris, On a déduit que chaque algorithme de ces 5 algorithmes a des avantages et des inconvénients catégorisés en 3 parties :

1.Temps d’exécution :

Pour la rapidité d’exécution le tri par Insertion et tri à bulles ne sont pas très efficaces vu leurs complexités temporelles très élevées alors qu’on trouve que le tri fusion il est nettement plus performant et donne de meilleurs résultats que les précédents algorithmes. De plus, le Tri rapide qui est un tri assez rapide mais plusieurs variantes existent ainsi son efficacité dépend du choix du pivot, car si ce dernier est mal choisi on se retrouve avec un temps d’exécution similaire à celui des deux premiers algorithmes traités (insertion & bulles). Puis viens le tri par tas qui est considéré comme le plus intéressant des algorithmes de tri, qui donne les meilleurs temps d’exécution.

2. Espace mémoire utilisé : pour ce point, on ne se soucie pas vraiment de cet aspect, car de nos jours la plupart des machines disposent suffisamment d’espace mémoire.

3.L’implémentation : Les algorithmes Tri pas Insertion et Tri à bulles sont les plus simples à implémenter comparer aux trois autres qui demandent un peu plus de maitrise.