République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des **S**ciences et de la **T**echnologie **H**ouari **B**oumediene
Faculté d'Electronique et d'Informatique
Département Informatique

Algorithme Avancé et Complexité
Rapport du Mini-Projet
LES TRIS

Rédigé par :

GUEDOUDJ Lamia
161631048387

MOHAMMEDI Ahlem
201500008953

Chargée de TP :
Dr. B.HEDJAZI

Niveau : 1ere Année Master
Spécialité : ingénierie des logicielles

Année universitaire : 2020/2021

Les Tris

Objectif

L'objectif de ce document est de présenter plusieurs algorithmes classiques de tri. On commence par présenter chaque méthode de manière intuitive, puis on détaille un exemple d'exécution de l'algorithme ainsi que son algorithme et pour finir on donnera des valeurs expérimentales pour chaque algorithme

Dans ce document nous étudierons cinq méthodes de tri différentes : Le Tri à bulles (Classique et Optimisé), le Tri Gnome le Tri par Distribution (Par Base ou Radi-Sort en Anglais), le Tri rapide et en fin le Tri par Tas (heap-Sort en Anglais).

Remarque :

Pour le calcul des temps d'exécution nous avons utilisé les fonctions : QueryPerformanceFrequency et QueryPerformanceCounter de la bibliothèque windows.h, les résultats sont donc en millisecondes.

I. Le Tri à bulles

Principe : Le principe du tri à bulles est de parcourir le tableau en échangeant lors du parcours deux éléments consécutifs s'ils sont rangés dans le mauvais ordre et de répéter ce processus jusqu'à ce qu'il n'y ait plus d'échanges lors d'un parcours.

1- Tri bulle

a- Programme C de la procédure TriBulle :

```
/************************  1 . Tri à bulles : ************************/

void Permuter(long* x,long* y) //permuter deux valeurs
  {
            int z=*x;
            *x=*y;
            *y=z;
  }

void TriBulle (long* T,long n)
  {long i;
   int changement=1;
        while(changement==1)
    {
            changement=0;
          for(i=0;i<n-1;i++)
                                if (T[i]>T[i+1])
          {
                                    Permuter(&T[i],&T[i+1]);
                                    changement=1;
          }
    }
}
```

b- Complexité théorique :

```
void TriBulle (long* T,long n)
 {      long i;
        int changement=1;  .....1
        while (changement==1) .....1 / n
       {      changement =0;.....1
              for (i=0;i<n-1;i++) .....n-1 /  3(n-1)
                    if (T[i]>T [i+1]).....0 / n-1
                   {
                         Permuter (&T[i],&T[i+1]); .....0 / 3(n-1)
                         changement =1 .....0 / n-1
                    }
         }
}
```

→ Meilleur cas

Dans le meilleur cas (lorsque le tableau est trié), la boucle externe est exécutée une seule fois et donc la boucle interne est exécutée : n-1-0=n-1 fois.

T(n)= 1+1+1+ (n-1)=n+2

D'où la complexité de cette fonction au meilleur cas est de l'ordre ~$\Omega(n)$.

→ Pire cas :

Dans le pire cas (lorsque le tableau est trié dans l'ordre inverse), la boucle externe est exécutée : (n-1) fois pour trier les n valeurs et 1 dernière fois lorsque tout le tableau est trié. Donc, la boucle interne est exécutée : $n \times (n-1) = n(n-1) = n^2 - n$ fois

T(n)= 1+n (3(n-1) + (n-1) +3(n-1) + (n-1))=1+8n²-8n

D'où la complexité de cette fonction au pire cas est de l'ordre ~ $O(n^2)$.

c- Etude du coût réel de l'algorithme :

	la procédure TriBulle	
N	T(N) au Meilleur cas	T(N) au Pire cas
1000	0	7,99
2000	0,01	30,44
3000	0,01	56,6
4000	0,01	96,09
5000	0,01	152,32
6000	0,02	212,27
7000	0,02	310,28
8000	0,02	392,92
9000	0,02	495,49
10000	0,03	608,78
20000	0,06	2537,58
30000	0,08	5605,53
40000	0,11	10058,98
50000	0,14	15728,48
100000	0,3	62773,07

d- Comparaison de complexité théorique avec les mesures expérimentales :

→ Meilleur cas :

Calcule de Δt : On a $T(n) = (n+2) \times \Delta t \leftrightarrow 0{,}01 = (1000+2)\,\Delta t$

$$\Delta t = \frac{0{,}01}{1002} = 4{,}99 * 10^{-6}$$

→ Pire cas :

Calcule de Δt : On a $T(n) = 1+8n^2-8n \times \Delta t \leftrightarrow 7{,}99 = 1+8(1000)^2-8(1000)\,\Delta t$

$$\Delta t = \frac{7{,}99}{1+8(1000)^2-8(1000)} = 9{,}99 * 10^{-7}$$

→ D'où : le tableau ci-dessous, représente les variations théoriques du temps d'exécution en fonction de n : $T(n) = F(n) \times \Delta t$ dans le meilleur et le pire cas :

	la procédure TriBulle (Δt)	
N	T(N) au Meilleur cas	T(N) au Pire cas
1000	0,005	7,912
2000	0,010	31,664
3000	0,015	71,256
4000	0,020	126,688
5000	0,025	197,960
6000	0,030	285,072
7000	0,035	388,025
8000	0,040	506,817
9000	0,045	641,449
10000	0,050	791,921
20000	0,100	3167,842
30000	0,150	7127,762
40000	0,200	12671,683
50000	0,250	19799,604
100000	0,499	79199,208

On trouve que la complexité théorique à une variation asymptotique par rapport aux mesures expérimentales. Donc la complexité théorique est un proche des résultats obtenus en mesure expérimentaux $(The \sim Te)$

Donc les résultats expérimentaux sont presque compatibles avec la complexité théorique

2- Tri bulle optimisé

a- Programme C de la procédure TriBulleOpt :

```
/************************  1 . Tri à bulles : ************************/
void Permuter(long* x,long* y) //permuter deux valeurs
  {
            int z=*x;
            *x=*y;
            *y=z;
  }
 void TriBulleOpt (long* T,long n)
{int i;
            long m=n-1;
            int changement=1;
            while(changement==1)
    {
                    changement=0;
                    for( i=0;i<m;i++)
                                    if (T[i]>T[i+1])
                {
Permuter(&T[i],&T[i+1]);
                                                changement=1;
                }
                    m--;
    }
}
```

b- Complexité théorique :

```
void TriBulleOpt (long* T,long n)
{       int i;
        long m=n-1;.....2
        int changement=1;.....1
        while(changement==1) .....n-1
       {        changement=0; .....1 /n-1
                for( i=0;i<m;i++)..... 3 / 8*(n − 1) + (n − 2) + (n − 3) + ⋯ + (n − n)
                        if (T[i]>T[i+1]).....0
                        {     Permuter (&T[i],&T[i+1]); .....0
                              changement =1; .....0
                        }
                      m--;.....n-1
   }
}
```

→ Meilleur cas

Dans le meilleur cas (lorsque le tableau est trié), la boucle externe est exécutée une seule fois et donc la boucle interne est exécutée : n-1-0=n-1 fois.

T(n)=6+n-1=n+5

D'où la complexité de cette fonction au meilleur cas est de l'ordre ~$\Omega(n)$.

→ Pire cas :

Dans le pire cas (lorsque le tableau est trié dans l'ordre inverse), la boucle externe est exécutée : (n- 1) fois pour trier les n valeurs et 1 dernière fois lorsque tout le tableau est trié.

La boucle interne est exécutée :

$(n-1)$ fois pour trier la première valeur du tableau.

$(n-2)$ fois pour trier la deuxième valeur du tableau.

$(n-3)$ fois pour trier la troisième valeur du tableau.

:::

$(n-n)$ fois pour trier la dernière valeur. Donc, la boucle interne est exécutée :

$(n-1)+(n-2)+(n-3)+\cdots+(n-n)$ =

$$\sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

$$T(n) = 3 + n - 1 + n - 1 + 8 * (1/2n^2 - 1/2n) = 1 + 2n + 4n^2 - 4n = 4n^2 - 2n + 1$$

D'où la complexité de cette fonction au pire cas est de l'ordre ~ $O(n^2)$

c- Etude du coût réel de l'algorithme :

Le tableau suivant représente les variations de temps d'exécution expérimentale au meilleur et pire cas

	la procédure TriBulleOpt	
N	T(N) au Meilleur cas	T(N) au Pire cas
1000	0	5,48
2000	0,01	21,87
3000	0,01	49,76
4000	0,02	89,55
5000	0,02	113,49
6000	0,03	160,17
7000	0,03	215,12
8000	0,03	287,18
9000	0,03	362,35
10000	0,04	457,34
20000	0,08	1772,74
30000	0,12	4102,93
40000	0,16	7244,61
50000	0,19	11513,43
100000	0,4	46216,97

d- Comparaison de complexité théorique avec les mesures expérimentales :

→ Meilleur cas :

Calcule de Δt : On a $T(n) = (n+5) \times \Delta t \leftrightarrow 0{,}04 = 1005\ \Delta t$

$$\Delta t = \frac{0{,}04}{1005} = 4{,}98 * 10^{-6}$$

→ Pire cas :

Calcule de Δt : On a $T(n) = 4n^2\text{-}2n+1 \times \Delta t \leftrightarrow 5{,}48 = 4(1000)^2\text{-}2(1000) +1\ \ \Delta t$

$$\Delta t = \frac{5{,}48}{4n^2-2n+1} = 1{,}37 * 10^{-6}$$

→ D'où : le tableau ci-dessous, représente les variations théoriques du temps d'exécution en fonction de n : $T(n) = F(n) \times \Delta t$ dans le meilleur et le pire cas :

	la procédure TriBulleOpt (Δt)	
N	T(N) au Meilleur cas	T(N) au Pire cas
1000	0,005	5,477
2000	0,010	21,915
3000	0,015	49,312
4000	0,020	87,669
5000	0,025	136,986
6000	0,030	197,264
7000	0,035	268,501
8000	0,040	350,698
9000	0,045	443,855
10000	0,050	547,973
20000	0,100	2191,945
30000	0,149	4931,918
40000	0,199	8767,890
50000	0,249	13699,863
100000	0,498	54799,726

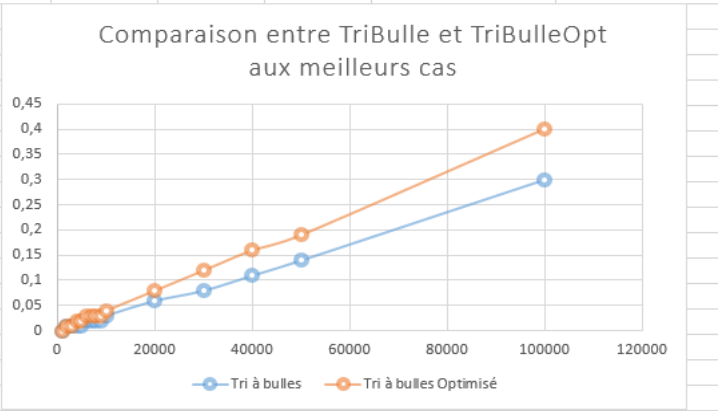
On trouve que la complexité théorique à une variation asymptotique par rapport aux mesures expérimentales. Donc la complexité théorique est un proche des résultats obtenus en mesure expérimentaux ($The \sim Te$)

Donc les résultats expérimentaux sont presque compatibles avec la complexité théorique

3- Comparaison entre les deux algorithmes :

→ Comparaison aux meilleurs cas :

N	Tri à bulles	Tri à bulles Optimisé
1000	0	0
2000	0,01	0,01
3000	0,01	0,01
4000	0,01	0,02
5000	0,01	0,02
6000	0,02	0,03
7000	0,02	0,03
8000	0,02	0,03
9000	0,02	0,03
10000	0,03	0,04
20000	0,06	0,08
30000	0,08	0,12
40000	0,11	0,16
50000	0,14	0,19
100000	0,3	0,4

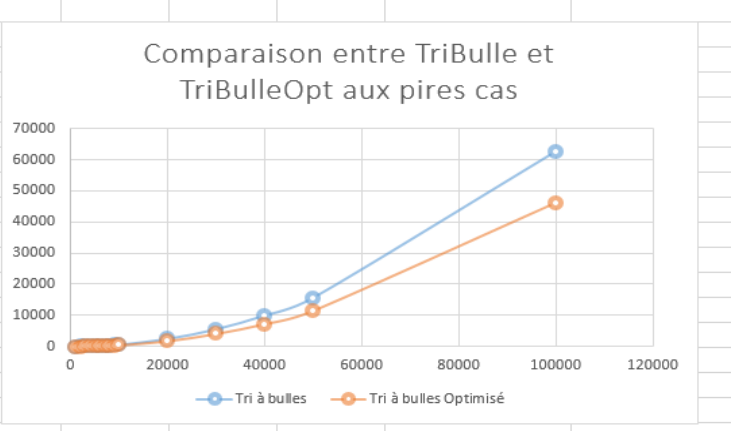


D'après les graphes ci-dessus, on remarque que :

- ✓ La complexité des deux fonctions est linéaire.
- ✓ Les deux fonctions donnent presque le même résultat dans le meilleur cas.

→ Comparaison aux pires cas :

N	Tri à bulles	Tri à bulles Optimisé
1000	7,99	5,48
2000	30,44	21,87
3000	56,6	49,76
4000	96,09	89,55
5000	152,32	113,49
6000	212,27	160,17
7000	310,28	215,12
8000	392,92	287,18
9000	495,49	362,35
10000	608,78	457,34
20000	2537,58	1772,74
30000	5605,53	4102,93
40000	10058,98	7244,61
50000	15728,48	11513,43
100000	62773,07	46216,97



D'après les deux graphes ci-dessus, on remarque que :

- ✓ La complexité des deux fonctions au pire cas est quadratique.
- ✓ Le temps d'exécution de TriBulleOpt est inférieur au temps d'exécution de TriBulle dans le pire cas parce que le nombre d'itération dans la boucle interne diminue de 1 après chaque itération de la boucle externe.

II. Le Tri Gnome

Principe : Le tri de gnome est un algorithme de tri qui s'apparente au tri par insertion, car il fonctionne avec un élément à la fois mais le met au bon endroit par une série de permutation, similaire à un tri à bulle. C'est un concept simple.

1- Tri gnome

a- Programme C de la procédure TriGnome :

```
/*************************     2 . Tri Gnome :     ******************************/

void Permuter(long* x,long* y) //permuter deux valeurs
  {
          int z=*x;
          *x=*y;
          *y=z;
  }

void TriGnome (long* T,long n)
 {
          long i=0;
          while(i<n-1)
    {
                  if (T[i]<=T[i+1])
          i++;
                  else
       {
                                                  Permuter(&T[i],&T[i+1]);
                                                  if(i!=0)
                          i--;
                  else
                          i++;
       }
    }
}
```

b- Complexité théorique :

void TriGnome (long* T,long n)

{ long i=0; 1

while (i<n-1) n-1

{if (T[i] <=T [i+1]) { 1

i++; 2

}

else {

Permuter (&T[i], &T [i+1]); 2n

if (i!=0) 2n

i--;.....n

else

i++;.....n

}

}

}

→ Meilleur cas

Dans le meilleur cas (lorsque le tableau est trié), on ne fait aucune permutation. Donc la boucle est exécutée : n-2-0+1=n-1 fois. Le nombre d'échanges=0.

T(n)= 1+(n-1) (1+ 2) =$3n-2$

D'où la complexité de cette fonction au meilleur cas est de l'ordre ~$\Omega(n)$.

→ Pire cas :

Dans le pire cas (lorsque le tableau est trié dans l'ordre inverse), le nombre d'échange varie entre 2n et n^2 donc la boucle est exécutée entre 2n et n^2 fois (selon la valeur de n) pour faire les permutations et n-1 fois après que le tableau soit trié. Le nombre d'échanges=2n.

D'où la complexité de cette fonction au pire cas est de l'ordre ~ $O\ (n^2)$.

c- Etude du coût réel de l'algorithme :

Le tableau suivant représente les variations de temps d'exécution expérimentale au meilleur et pire cas

	la procédure TriGnome	
N	T(N) au Meilleur cas	T(N) au Pire cas
1000	0	7,88
2000	0,01	32,52
3000	0,01	69,02
4000	0,01	112,88
5000	0,02	154,34
6000	0,02	226,97
7000	0,02	302,51
8000	0,03	393,15
9000	0,03	501,53
10000	0,03	611,86
20000	0,06	2541,79
30000	0,1	5745,57
40000	0,13	10445,01
50000	0,16	16297,03
100000	0,32	65440,03

d- Comparaison de complexité théorique avec les mesures expérimentales :

→ Meilleur cas :

Calcule de Δt : On a $T\ (n) = (3n - 2) \times \Delta t \leftrightarrow 0{,}02 = ((3* 1000)\text{-}2)\ \Delta t$

$$\Delta t = \frac{0{,}02}{((3* 1000)-2)} = 1{,}66 * 10^{-6}$$

→ Pire cas :

Calcule de Δt : On a $T\ (n) = (n + 8n^2 - 8n) \times \Delta t \leftrightarrow 7{,}88 = (1000\ + 8(1000\ )^2 - 8(1000\ ))\ \Delta t$

$$\Delta t = \frac{7{,}88}{(1000\ )^2} = 0{,}0000078$$

→ D'où : le tableau ci-dessous, représente les variations théoriques de t en fonction de n : $T\ (n)$ = F $(n) \times \Delta t$ dans le meilleur et le pire cas :

→ Dans le meilleur cas On trouve que la complexité théorique à une variation asymptotique par rapport aux mesures expérimentales. Donc la complexité théorique est un peu proche des résultats obtenus en mesure expérimentaux Donc les résultats expérimentaux sont presque compatibles avec la complexité théorique en revanche, au pire cas la complexité théorique ne donne pas les mêmes résultats que les mesures expérimentales

	la procédure TriGnome (Δt)	
N	T(N) au Meilleur cas	T(N) au Pire cas
1000	0,005	7,800
2000	0,010	31,200
3000	0,015	70,200
4000	0,020	124,800
5000	0,025	195,000
6000	0,030	280,800
7000	0,035	382,200
8000	0,040	499,200
9000	0,045	631,800
10000	0,050	780,000
20000	0,100	3120,000
30000	0,149	7020,000
40000	0,199	12480,000
50000	0,249	19500,000
100000	0,498	78000,000

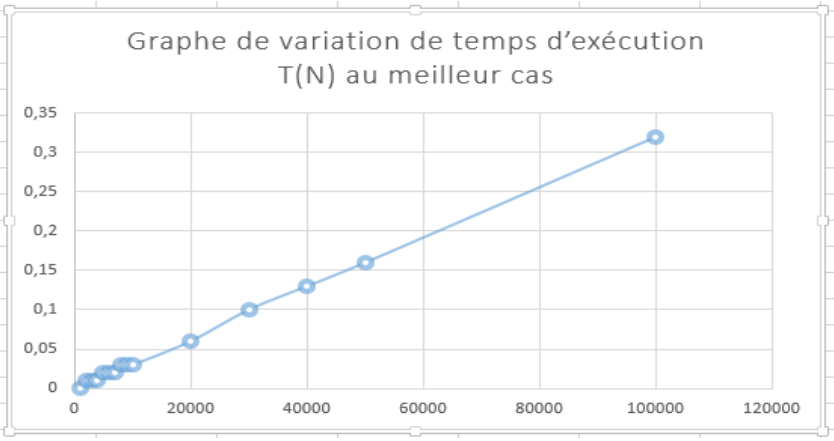
On trouve que la complexité théorique à une variation asymptotique par rapport aux mesures expérimentales. Donc la complexité théorique est un peu proche des résultats obtenus en mesure expérimentaux ($The \sim Te$)

Donc les résultats expérimentaux sont presque compatibles avec la complexité théorique

e- Graphe de variation de temps d'exécution T(N) :

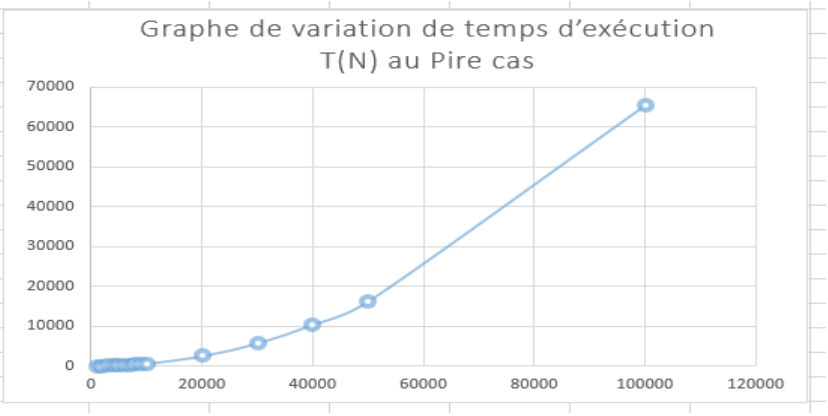
→ Meilleur cas

N	T(N) au Meilleur cas
1000	0
2000	0,01
3000	0,01
4000	0,01
5000	0,02
6000	0,02
7000	0,02
8000	0,03
9000	0,03
10000	0,03
20000	0,06
30000	0,1
40000	0,13
50000	0,16
100000	0,32

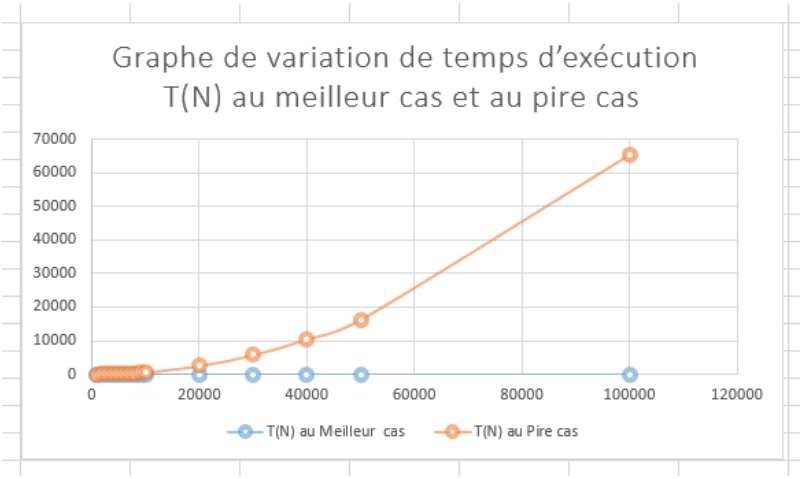


→ Pire cas

N	T(N) au Pire cas
1000	7,88
2000	32,52
3000	69,02
4000	112,88
5000	154,34
6000	226,97
7000	302,51
8000	393,15
9000	501,53
10000	611,86
20000	2541,79
30000	5745,57
40000	10445,01
50000	16297,03
100000	65440,03



f- Comparaison entre le meilleur et le pire cas :



D'après le graphe ci-dessus, on remarque que :

- ✓ La complexité de fonction au pire cas est linéaire.
- ✓ La complexité de fonction au meilleur cas est quadratique.
- ✓ Approche adoptée au meilleur cas est plus performante

III. Le Tri par distribution

1- Tri par distribution

a- Programme C de la procédure TriBase :

→ La fonction cle

```
/************     Fonction cle     ************/
   int cle(long x,long i) //retourner la clé i d'une valeur x
 {long j;
            for (j=1; j<=i; j++)
                    x=x/10;
            return x%10 ;
 }
```

→ La procédure TriAux

```
/************ Procedure TriAux ************/
void TriAux (long *T,long n,long i)    //tri d'un tableau par la ième clé
 {
 long j, m,indice,l;
   long *tab=(long *)malloc((n+1)*sizeof(long));
     long *k=(long *)malloc((n+1)*sizeof(long));
    indice=0;
   for(j=0;j<n;j++){
       k[j]=cle(T[j],i);
   }
       for(j=0;j<10;j++){

         for(m=0;m<n;m++){
             if(k[m]==j){

                tab[indice]=T[m];
                indice=indice+1;
             }
         }
       }
         for(m=0;m<n;m++){
         T[m]=tab[m];

         }
}
```

→ La procédure TriBase

```
/************ Procedure TriBase ************/
 void TriBase(long* T,long n,long k)
{ long i;
            for (i=1; i<=k; i++) {
                TriAux(T, n, i);
                                  }
}
```

b- Complexité théorique :

→ Meilleur \ Pire cas

→ Calcule de la complexité de la Fonction cle :

long cle(long x,long i) //retourner la clé i d'une valeur x
{long j,r;
for (j=1; j<=i; j++) { 3*i
r=x%10; 2i
x=x/10; 2i
}
return r ;.....1
}

→ Calcule de la complexité de la procédure TriAux :

void TriAux (long *T,long n,long i) //tri d'un tableau par la ième clé
{long j, m, indice, l ;
long *tab = (long *) malloc ((n+1)*sizeof(long));
long *k = (long *) malloc ((n+1)*sizeof(long));
indice =0; 1
for (j=0;j<n;j++) { 3n
k[j]=cle(T[j], i);..... n (7i+1)+n
}
for (j=0;j<10;j++) {..... 10
for (m=0;m<n;m++) {..... 10*3n
if (k[m] ==j) { 10n
tab [indice]=T[m];..... 10n
indice =indice+1; 11n
}
}
}
for (m=0;m<n; m++) {..... 3n
T[m]=tab[m]; ..... n
}
}

La fonction cle a un temps constant car au pire cas le nombre de chiffres d'un entier est égal à k (la boucle de la fonction cle s'exécute k fois) et dans l'énoncé k est considéré comme une constante entière, donc la fonction cle est de l'ordre de $O(1)$.

$T(n) = 36$

Pour réaliser TriAux en temps linéaire : nous avons utilisé deux tableaux supplémentaires l'un contient les chiffres soit des unités ou des dizaines ... et le deuxième tableau contient les éléments triés par le chiffre des unités puis par les dizaines ...

La première boucle de la procédure TriAux s'exécute n fois donc elle est de l'ordre $\Theta(n)$.

La deuxième boucle c'est une boucle imbriqué (la boucle externe s'exécute 10 fois et la boucle interne s'exécute n fois) donc elle est de l'ordre$\Theta\ (10 * n) \sim \Theta(n)$.

La troisième boucle s'exécute n fois donc elle est de l'ordre $\Theta(n)$.

D'où, la complexité de la procédure TriAux est de l'ordre $\Theta\ (n + 10 \times n + n) \sim \Theta(n)$. (Car $i \leq 5$).

$$T(n) = 3n + n(7i + 1) + n + 10 + 10 * 3n + 10n + 10n + 11n + 3n + n$$
$$= 3n + 36n + n + 10 + 30n + 20n + 14n + n =$$
$$95n + 10$$

La complexité au pire et meilleur cas de la procédure TriBase est de l'ordre de $\Theta\ (k \times n) \sim \Theta(n)$ car (k paramètre d'entrée = 5)

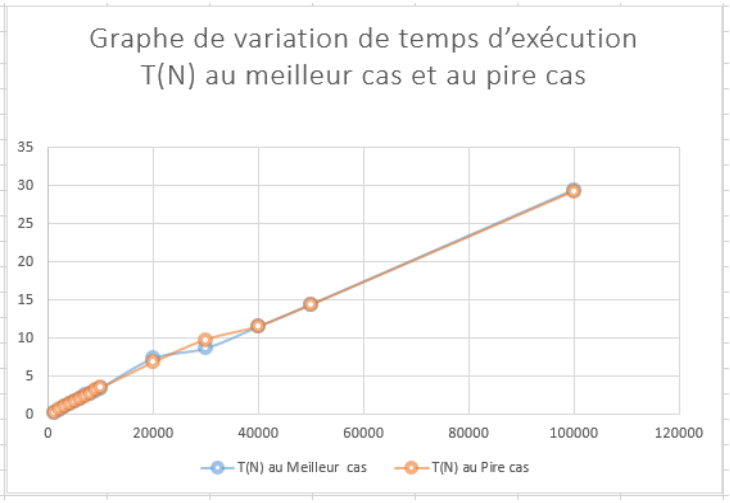
$$T(n) = 5(95n + 10) = 475n + 50$$

c- Etude du coût réel de l'algorithme :

Le tableau suivant représente les variations de temps d'exécution expérimentale au meilleur et pire cas

	la procédure TriBase	
N	T(N) au Meilleur cas	T(N) au Pire cas
1000	0,33	0,33
2000	0,67	0,69
3000	1,02	1,09
4000	1,37	1,46
5000	1,76	1,74
6000	2,16	2,09
7000	2,58	2,43
8000	2,82	2,84
9000	3,16	3,19
10000	3,48	3,54
20000	7,38	6,89
30000	8,69	9,85
40000	11,54	11,58
50000	14,45	14,39
100000	29,43	29,22

d- Graphe de variation de temps d'exécution T(N) :



D'après le graphe ci-dessus, on remarque que :

✓ La complexité de fonction au pire et meilleur cas est linéaire

e- Comparaison de complexité théorique avec les mesures expérimentales :

→ Meilleur \ Pire cas :

Calcule de Δt : On a $T(n) = 475n + 50 \times \Delta t \leftrightarrow 0{,}33 = 475(1000) + 50\ \Delta t$

$$\Delta t = \frac{0{,}33}{475(1000)+50} = 6{,}94 * 10^{-7}$$

→ D'où : le tableau ci-dessous, représente les variations théoriques de t en fonction de n : $T(n) = F(n) \times \Delta t$ dans le meilleur et le pire cas :

	TriBase (Δt)
N	T(N) M \ P cas
1000	0,330
2000	0,659
3000	0,989
4000	1,319
5000	1,648
6000	1,978
7000	2,308
8000	2,637
9000	2,967
10000	3,297
20000	6,593
30000	9,890
40000	13,186
50000	16,483
100000	32,965

On trouve que la complexité théorique à une variation asymptotique par rapport aux mesures expérimentales. Donc la complexité théorique est un proche des résultats obtenus en mesure expérimentaux ($The \sim Te$)

Donc les résultats expérimentaux sont presque compatibles avec la complexité théorique

IV. Le Tri rapide

1- Tri rapide

a- Programme C de la procédure Trirapide :

```
/****************************** 4 . Tri rapide :    ******************************/
void triRapide (long* tab,long p,long r)
 {
        long q;
        if(p<r)
     {
                q=partitionner(tab,p,r);
                triRapide(tab,p,q-1);
                triRapide(tab,q+1,r);
     }
}
/************ Fonction partitionner ************/
 int partitionner(long* tab,long d,long f)
{ //mettre chaque élément à gauche ou à droite de pivot
        long eltPivot;
        int i,j,x;
        eltPivot=tab[(d+f)/2];
        i=d;
        j=f+1;
        do
    {
                do{
                    i++;
                }while(tab[i]<eltPivot && i<=f);

                do{
                    j--;
                }while(tab[j]>eltPivot);

                if(i<j)
        {
         x=tab[i];
                    tab[i]=tab[j];
                    tab[j]=x;
        }
          }while(j>i);

          tab[d]=tab[j];
          tab[j]=eltPivot;

           return j;

           }
```

b- Complexité théorique :

La complexité de cet algorithme est égale à la complexité pour trier les éléments de p à q-1 + la complexité pour trier les éléments de q+1 à r (q indice est l'indice de pivot).

→ Meilleur cas : Dans le meilleur cas (choix de bon pivot), le pivot est la valeur qui permet d'avoir le même nombre de valeurs à gauche et à droite de pivot. Donc, à chaque itération la fonction triRapide fait deux appels récursifs à elle-même avec la moitié des éléments du tableau en entrée pour chaque appel et donc la complexité est de l'ordre $\Omega\ (n\ log\ n)$.

→ Pire cas : Dans le pire cas (choix de mauvais pivot), le pivot est la valeur qui permet d'avoir n-1 éléments à gauche de pivot et aucun élément à droite (ou inversement). Donc, à chaque itération la fonction triRapide fait deux appels récursifs à elle-même avec n-1 éléments en entré dans un appel et aucun élément dans l'autre et donc la complexité est de l'ordre $\Omega\ (n^2)$.

→ L'équation de récurrence du tri rapide :

→ Meilleur cas

Dans le partitionnement le plus équilibré possible, la fonction PARTITIONNER produit deux sous- problèmes de taille non supérieure à $\frac{n}{2}$ vu que l'un est de taille $\frac{n}{2}$ et l'autre de taille $\frac{n}{2} - 1$.

On a : $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

$= 2*T\left(\frac{n}{2}\right) + n$

= 2*(2* T ($\frac{n}{2^2}$) +$\frac{n}{2}$) + n

= 2^2 * T ($\frac{n}{2^2}$) + 2n

=2^2 * (2* $T\left(\frac{n}{2^3}\right)$+ $\frac{n}{2^2}$+2n

= 2^3 * $T\left(\frac{n}{2^3}\right)$+3n

...

=2^k* $T\left(\frac{n}{2^k}\right)$+ k*n

On pose $n = 2^k \rightarrow k = log\ n$

$$T(n) = n * T(1) + n * log\ n = n + nlogn$$

Donc : $T(n) = O(n\ log\ n)$

→ Pire cas :

Dans le pire cas, pour le tri rapide quand la procédure de partitionnement produit un sous-problème à n − 1 éléments et une autre avec 0 élément. Supposons que ce partitionnement survienne à chaque appel récursif. Le partitionnement coûte Θ(n). Comme l'appel récursif sur un tableau de taille 0 rend la main sans rien faire, T(0) = Θ(1) et la récurrence pour le temps d'exécution est

$$\mathbf{T(n)=}\begin{cases} \boldsymbol{T(0) = \Theta(1) = 1} & \textbf{si n = 0} \\ \boldsymbol{T(n-1) + T(0) + \Theta(n)} & \textbf{si n>1} \end{cases}$$

On a : $T(n) = T(n-1) + T(0) + \Theta(n)$

T (n-1)=T (n-2) + (n-1)

T (n-2)=T (n-3) + (n-2) T (n-3)=T (n-4) + (n-3)

…

T(2)=T(1) + 2

T(1) =1

Donc : T(n) = T (n-1) +n

=T (n-2) + (n-1) + n

= T (n-3) + (n-2) + (n-1) + n

= T (n-4) + (n-3) + (n-2) + (n-1) + n

…

T(n) = T(1) + 2 + 3+ … + (n-3) + (n-2) + (n-1) + n

= 1 + 2 + 3+ … + (n-3) + (n-2) + (n-1) + n

$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ est de l'ordre de $\Theta\ (n^2)$

c- Etude du coût réel de l'algorithme :

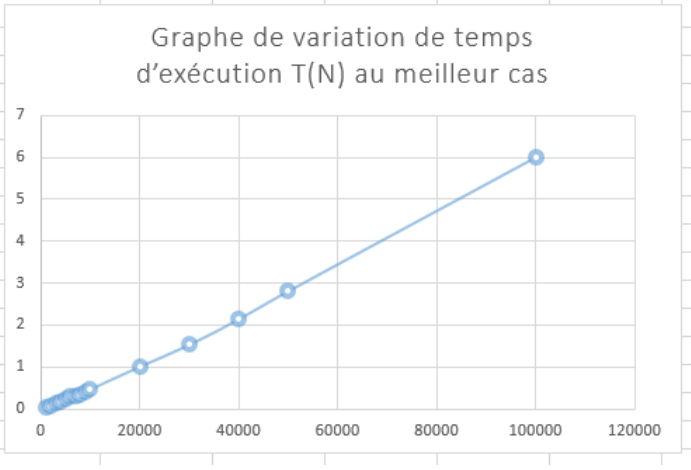
Le tableau suivant représente les variations de temps d'exécution en fonction de n valeurs données au meilleur et pire cas

	la procédure TriRapide	
N	Meilleur cas	pire cas
1000	0,04	0,06
2000	0,08	0,12
3000	0,13	0,18
4000	0,17	0,23
5000	0,23	0,29
6000	0,29	0,35
7000	0,32	0,42
8000	0,35	0,53
9000	0,41	0,55
10000	0,47	0,61
20000	1,01	1,05
30000	1,53	1,58
40000	2,14	2,19
50000	2,81	2,83
100000	6	6,6

d- Graphe de variation de temps d'exécution T(N) :

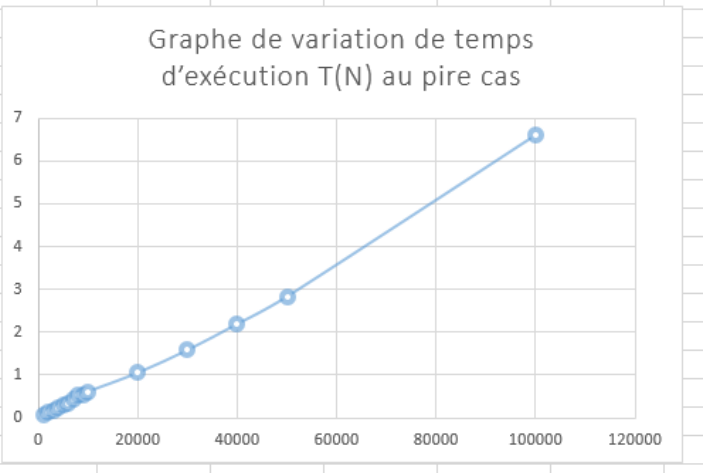
→ Meilleur cas

N	T(N) meilleur cas
1000	0,04
2000	0,08
3000	0,13
4000	0,17
5000	0,23
6000	0,29
7000	0,32
8000	0,35
9000	0,41
10000	0,47
20000	1,01
30000	1,53
40000	2,14
50000	2,81
100000	6

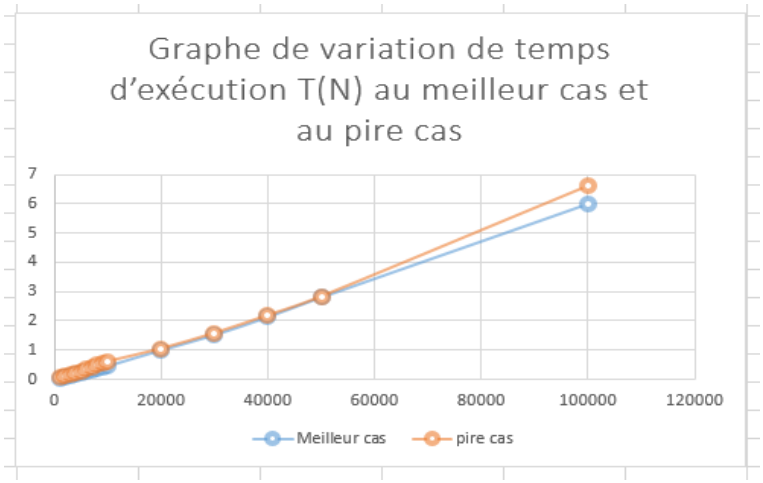


→ Pire cas

N	T(N) au pire cas
1000	0,06
2000	0,12
3000	0,18
4000	0,23
5000	0,29
6000	0,35
7000	0,42
8000	0,53
9000	0,55
10000	0,61
20000	1,05
30000	1,58
40000	2,19
50000	2,83
100000	6,6



e- Comparaison entre le meilleur et le pire cas :



D'après le graphe ci-dessus, on remarque que :

- ✓ La complexité de fonction au pire cas est linéarithmique.
- ✓ La complexité de fonction au meilleur cas est quadratique.
- ✓ Approche adoptée au meilleur cas est plus performante

f- Comparaison de complexité théorique avec les mesures expérimentales :

→ Meilleur cas :

Calcule de Δt : On a $T(n) = n \times \Delta t \leftrightarrow 0{,}05 = 1000\ \Delta t$

$$\Delta t = \frac{0{,}04}{1000 \log 1000} = 1{,}33 * 10^{-5}$$

→ Pire cas :

Calcule de Δt : On a $T(n) = n \times \Delta t \leftrightarrow 0{,}06 = 1000\ \Delta t$

$$\Delta t = \frac{0{,}06}{1000^2} = 0{,}00000006$$

→ D'où : le tableau ci-dessous, représente les variations théoriques du temps d'exécution en fonction de n : $T(n) = F(n) \times \Delta t$ dans le meilleur et le pire cas :

	la procédure TriRapide(Δt)	
N	Meilleur cas	pire cas
1000	0,040	0,060
2000	0,088	0,240
3000	0,139	0,540
4000	0,192	0,960
5000	0,246	1,500
6000	0,301	2,160
7000	0,358	2,940
8000	0,415	3,840
9000	0,473	4,860
10000	0,532	6,000
20000	1,144	24,000
30000	1,786	54,000
40000	2,448	96,000
50000	3,125	150,000
100000	6,650	600,000

On trouve que la complexité théorique à une variation asymptotique par rapport aux mesures expérimentales. Donc la complexité théorique est un peu proche des résultats obtenus en mesure expérimentaux ($The \sim Te$) Donc les résultats expérimentaux sont presque compatibles avec la complexité théorique

V. Le Tri par tas

1- Tri par tas

a- Programme C de la procédure Tritas :

```c
/*********************   5 . Tri par tas :   ******************************/
/*********** Procedure inserer_dans_tas  ************/
  void inserer_dans_tas(long *arr, long n, long i) {
    // Find largest among root, left child and right child
    long largest = i;
    long left = 2 * i + 1;
    long 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]);
      inserer_dans_tas(arr, n, largest);
    }
  }
/*********** Procedure supprime_min  ************/
void supprime_min (long *arr, long n) {
   // Heap sort
   long i;
    for (i = n - 1; i >= 0; i--) {
      Permuter(&arr[0], &arr[i]);

      // Heapify root element to get highest element at root again
      inserer_dans_tas(arr, i, 0);
    }
  }
/*********** Procedure Tri_Tas ************/
  void Tri_Tas(long *arr, long n) {
    long i;
    // Build max heap
    for (i = n / 2 - 1; i >= 0; i--)
      inserer_dans_tas(arr, n, i);
   supprime_min ( arr, n);
  }
```

b- Complexité théorique :

void inserer_dans_tas(long *arr, long n, long i)

{long largest = i; 1logn

long left = 2 * i + 1; 3logn

long right = 2 * i + 2; 3logn

if (left < n && arr[left] < arr[largest]).....1logn

largest = left; 1logn

if (right < n && arr[right] < arr[largest])..... 1logn

largest = right; 1logn

if (largest != i) { 1logn

Permuter(&arr[i], &arr[largest]); 3logn

inserer_dans_tas(arr, n, largest); 1logn

}

}

$$T(n) = 16\, logn$$

void supprime_min (long *arr, long n)

```
{ long i;
   for (i = n - 1; i >= 0; i--) { ..... n
            Permuter(&arr[0], &arr[i]);.....3n
           inserer_dans_tas(arr, i, 0);.....16nlogn
      }
   }
```

$$T(n) = 16nlogn + 4n$$

```
      void Tri_Tas(long *arr, long n)
     { long i;
       for (i = n / 2 - 1; i >= 0; i--)..... 3n/2
       {inserer_dans_tas(arr, n, i) ..... 16logn*n/2=8nlogn
        } supprime_min ( arr, n); ..... 16nlogn + 4n
       }
```

$$T(n) = 3n/2 + 8nlogn + 16nlogn + 4n$$

→ Meilleur \ Pire cas :

Dans la procédure **inserer_dans_tas()** , au pire des cas, l'élément sera inséré au niveau d'une feuille de l'arbre. Le nombre d'itérations de la boucle de l'algorithme est égal au nombre de nœud se trouvant sur la chaine allant de la racine vers la feuille. Ce nombre correspond à la profondeur de l'arbre. Il s'agit donc de mesurer la profondeur par rapport au nombre total de nœud. Si l'arbre est complet, le nombre de nœud dans un niveau de l'arbre est une puissance de 2.

Au niveau 0 (la racine), il y a 2^0 nœud

Au niveau 1, il y a 2^1nœuds

Au niveau 2, il y a 2^2 nœuds

Au niveau p, il y a 2^P nœuds (p étant la profondeur de l'arbre)

Le nombre total de nœuds de l'arbre est donc égal à $\sum_{i=0}^{i=p} 2^i = \frac{1-2^{p+1}}{1-2} = 2^{p+1}-1$

$2^{p+1}-1 = n \rightarrow 2^{p+1} = n+1 \rightarrow p+1 = \log(n+1) \rightarrow p = \log(n+1) - 1$

Donc la complexité est en $\Theta(log\ n)$

La procédure supprime_min() fait appel à la procédure inserer_dans_tas() qui a une complexité de l'ordre $\Theta(log\ n)$ et un appel à la procédure Permuter() qui a une complexité constante, n fois.

Donc la complexité est en $\Theta(n\ log\ n)$.

La procédure Tri_Tas () fait appel à inserer_dans_tas() $\frac{n}{2}$ fois. Donc le pire cas de complexité cette étape est : $\frac{n}{2} \times log\ n\ = \Theta(nlog\ n)$

Tri_Tas() fait appel à supprime_min() , qui a une complexité égale à $\Theta(nlog\ n)$, une seule fois.

Donc La complexité de la procédure Tri_Tas () est de l'ordre $\Theta(nlog\ n)$ pour tous les cas (meilleur cas et pire cas).

c- Etude du coût réel de l'algorithme :

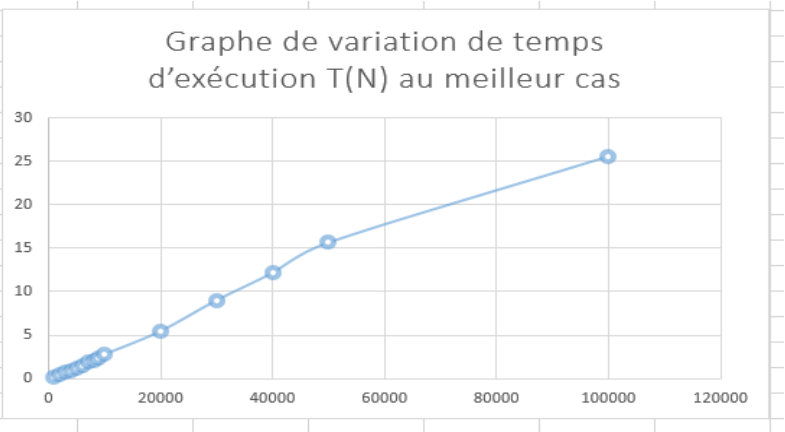
Le tableau suivant représente les variations de temps d'exécution en fonction de n valeurs données au meilleur et pire cas

	la procédure TriTas	
N	T(N) au Meilleur cas	T(N) au Pire cas
1000	0,21	0,26
2000	0,45	0,46
3000	0,69	0,87
4000	0,95	0,99
5000	1,21	1,26
6000	1,46	1,54
7000	1,85	1,86
8000	2,09	2,25
9000	2,36	2,51
10000	2,8	3,31
20000	5,47	5,72
30000	9,01	9,47
40000	12,15	14,39
50000	15,65	15,98
100000	25,54	40,54

d- Graphe de variation de temps d'exécution T(N) :

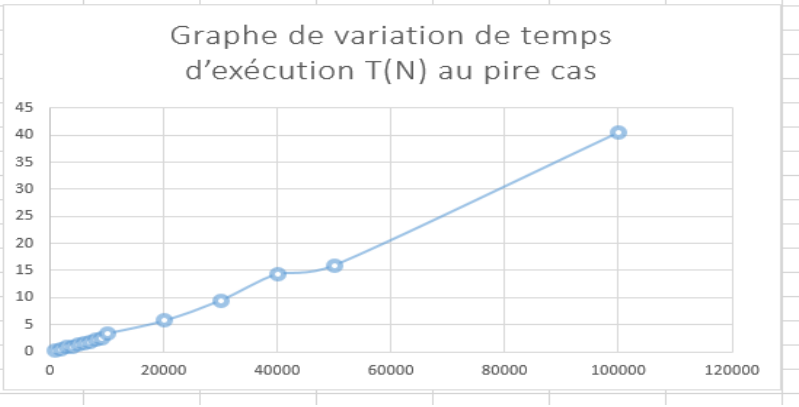
→ Meilleur cas

N	T(N) au Meilleur cas
1000	0,21
2000	0,45
3000	0,69
4000	0,95
5000	1,21
6000	1,46
7000	1,85
8000	2,09
9000	2,36
10000	2,8
20000	5,47
30000	9,01
40000	12,15
50000	15,65
100000	25,54

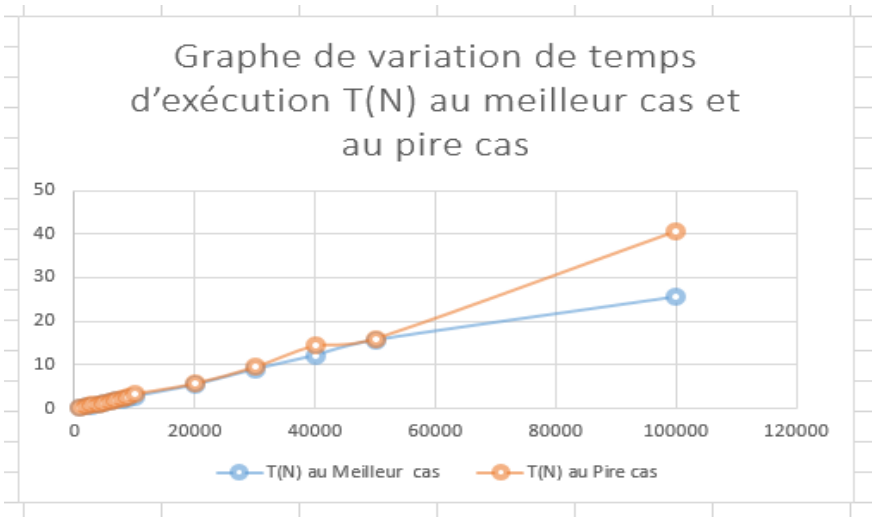


→ Pire cas

N	T(N) au Pire cas
1000	0,26
2000	0,46
3000	0,87
4000	0,99
5000	1,26
6000	1,54
7000	1,86
8000	2,25
9000	2,51
10000	3,31
20000	5,72
30000	9,47
40000	14,39
50000	15,98
100000	40,54



e- Comparaison entre le meilleur et le pire cas :



D'après le graphe ci-dessus, on remarque que :

- ✓ La complexité de fonction au pire et au meilleur cas est linéarithmique.
- ✓ Approche adoptée au meilleur cas est plus performante car le temps d'exécution est plus petit.

f- Comparaison de complexité théorique avec les mesures expérimentales :

→ Meilleur \ Pire cas :

Calcule de Δt : On a $T(n) = 3n/2 + 8nlogn + 16nlogn + 4n \times \Delta t \leftrightarrow$

$0{,}2 = 3(1000)/2 + 8(1000)log(1000) + 16(1000)log(1000) + 4(1000)\,\Delta t$

$$\Delta t = \frac{0{,}2}{3(1000)/2+8(1000)log(1000)+16(1000)log(1000)+4(1000)} = \frac{0{,}2}{77500} = 2{,}58 * 10^{-6}$$

→ D'où : le tableau ci-dessous, représente les variations théoriques de t en fonction de n : $T(n) = n \times \Delta t$ dans le meilleur et le pire cas :

	TriTas (Δt)
N	T(N) M \ P cas
1000	0,200
2000	0,437
3000	0,688
4000	0,949
5000	1,216
6000	1,489
7000	1,766
8000	2,047
9000	2,331
10000	2,619
20000	5,610
30000	8,742
40000	11,966
50000	15,258
100000	32,379

On trouve que la complexité théorique à une variation asymptotique par rapport aux mesures expérimentales. Donc la complexité théorique est un proche des résultats obtenus en mesure expérimentaux $(The \sim Te)$

Donc les résultats expérimentaux sont presque compatibles avec la complexité théorique

VI. Programme principale

```c
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#define MILLI_S 1000.0

typedef long *array;
array A;
array Tmp;

/*********** Procedure afficher ************/
void afficher_tab(long *tab,int n){
    long i;
      for (i = 0; i < n; i++){
        printf("%ld \t", tab[i]);
      }

}
```

{On trouve les procédures des tris précédentes ici}

```c
int Menu()
{
    int Choix;
    do
    {
        printf("                                    É»\n");
        printf("                                    º\n");
        printf("                                    º\n");
        printf("                                    º\n");
        printf("                                    È¼\n");
        printf("\n                          Veuillez choisir un Algorithme de tri, Tapez :\n");
        printf("\n                               1- Tri a bulles");
        printf("\n                               2- Tri a bulles Optimal");
        printf("\n                               3- Tri gnome");
        printf("\n                               4- Tri par distribution");
        printf("\n                               5- Tri rapide");
        printf("\n                               6- Tri par Tas");
        printf("\n                               7- Quitter\n");
        printf("\n\n\n\nChoix :");
        scanf("%d",&Choix);
    } while (Choix <1 || Choix >7);
//  system("cls");
    return Choix;
}
```

```c
int main()
 {      printf("\n\n");
        printf(">>>>>>>>>>>>>>>>>>>>>>>>      Ce Projet est realise par:   MOHAMMEDI AHLEM     &    GUEDOUDJ LAMIA      <<<<<<<<<<<<<<<<<<\n");
        printf("  \n");
        printf("\n\n\n");
        LARGE_INTEGER deb, fin, freq;
            int cas;
            long var;
            long i,j;
            int Choix;
            long tab_val[]={ pow(10,3),2*pow(10,3),3*pow(10,3),4*pow(10,3),5*pow(10,3),6*pow(10,3),7* pow(10,3),8*pow(10,3),
            9*pow(10,3),10*pow(10,3) ,20*pow(10,3) ,30*pow(10,3) ,40*pow(10,3) ,50*pow(10,3),100*pow(10,3)};
            //long tab_val[]={100};
        Choix = Menu();
            while (Choix!=7)
    {           printf("\n        Veuillez choisir un cas de tri, Tapez :\n");
                printf("\n                1- Meilleur cas");
                printf("\n                2- Pire cas");
                printf("\n\n\nCas :");
                scanf("%d",&cas);
          for (i=0;i<15;i++)
        {       long *tab=(long *)malloc((tab_val[i]+1)*sizeof(long));

                  switch(cas)
        {
                 case 1:
                        var=1;
                        for (j=0;j<tab_val[i];j++)
                        {tab[j]=var; var++;}
                        break;
                  case 2:
                        var=tab_val[i];
                        for (j=0;j<tab_val[i];j++)
                        {tab[j]=var; var--;}
                        break;
        }
```

```c
        }
                    QueryPerformanceFrequency(&freq);
                    QueryPerformanceCounter(&deb);

                       switch(Choix)
        {
                    case 1:
                            TriBulle(tab,tab_val[i]);
                            break;
                    case 2:
                            TriBulleOpt(tab,tab_val[i]);
                            break;
                    case 3:
                            TriGnome(tab,tab_val[i]); // afficher_tab(tab,tab_val[i]);
                            break;
                      case 4:
                            TriBase(tab,tab_val[i],5); // afficher_tab(tab,tab_val[i]);
                            break;
                    case 5:
                            triRapide(tab,0,tab_val[i]-1);
                            break;
                    case 6:
                            Tri_Tas(tab, tab_val[i]);
                            //afficher_tab(tab,tab_val[i]);
                            break;
        }
           QueryPerformanceCounter(&fin);
          double delta=(MILLI_S * (fin.QuadPart - deb.QuadPart)) / freq.QuadPart;
                    printf("T[%ld]\t=\t%3.2lf ms \n",tab_val[i],delta);
    }
        printf("\n\n\n\n\n\n Appuyez sur une touche pour retourner au menu principal  ");
                    getch();
      system("cls"); //efface l'écran
                Choix=Menu();
    }     return 0;
}
```

VII. Comparaison entre les cinq méthodes de tris

1- Complexité théorique

Regrouper dans un tableau comparatif les complexités théoriques et dans un autre tableau les mesures du temps obtenues pour les 5 algorithmes :

La complexité théorique / les Tris	Meilleur cas	Pire cas
Le tri à bulles	n	n^2
Le tri à bulles optimisé	n	n^2
Le tri gnome	n	n^2
Le tri par distribution	n	n
Le tri rapide	$n\ log\ n$	n^2
Le tri par tas	$n\ log\ n$	$n\ log\ n$

2- Etude du coût réel des algorithmes

→ Meilleur cas :

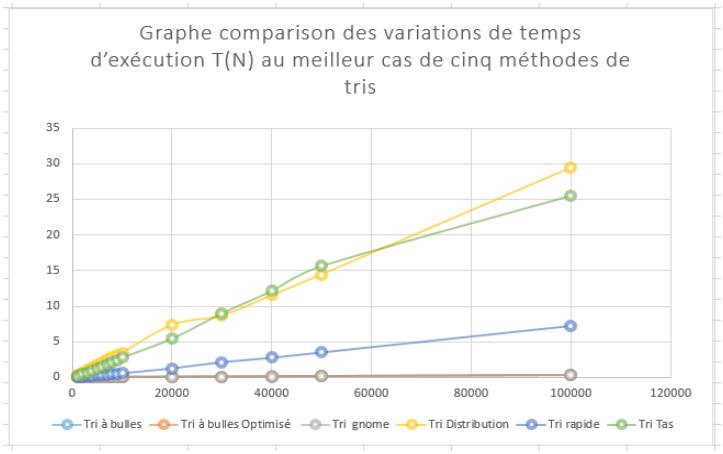
N	Tri à bulles	Tri à bulles Optimisé	Tri gnome	Tri Distribution	Tri rapide	Tri Tas
1000	0	0	0	0,33	0,04	0,21
2000	0,01	0,01	0,01	0,67	0,08	0,45
3000	0,01	0,01	0,01	1,02	0,13	0,69
4000	0,01	0,02	0,01	1,37	0,17	0,95
5000	0,01	0,02	0,02	1,76	0,23	1,21
6000	0,02	0,03	0,02	2,16	0,29	1,46
7000	0,02	0,03	0,02	2,58	0,32	1,85
8000	0,02	0,03	0,03	2,82	0,35	2,09
9000	0,02	0,03	0,03	3,16	0,41	2,36
10000	0,03	0,04	0,03	3,48	0,47	2,8
20000	0,06	0,08	0,06	7,38	1,01	5,47
30000	0,08	0,12	0,1	8,69	1,53	9,01
40000	0,11	0,16	0,13	11,54	2,14	12,15
50000	0,14	0,19	0,16	14,45	2,81	15,65
100000	0,3	0,4	0,32	29,43	6	25,54

→ Pire cas :

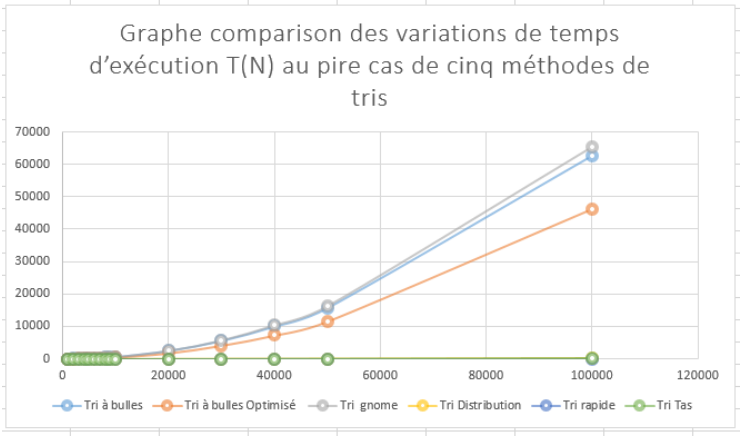
N	Tri à bulles	Tri à bulles Optimisé	Tri gnome	Tri Distribution	Tri rapide	Tri Tas
1000	7,990	5,480	7,880	0,330	0,060	0,26
2000	30,440	21,870	32,520	0,690	0,120	0,46
3000	56,600	49,760	69,020	1,090	0,180	0,87
4000	96,090	89,550	112,880	1,460	0,230	0,99
5000	152,320	113,490	154,340	1,740	0,290	1,26
6000	212,270	160,170	226,970	2,090	0,350	1,54
7000	310,280	215,120	302,510	2,430	0,420	1,86
8000	392,920	287,180	393,150	2,840	0,530	2,25
9000	495,490	362,350	501,530	3,190	0,550	2,51
10000	608,780	457,340	611,860	3,540	0,610	3,31
20000	2537,580	1772,740	2541,790	6,890	1,050	5,72
30000	5605,530	4102,930	5745,570	9,850	1,580	9,47
40000	10058,980	7244,610	10445,010	11,580	2,190	14,39
50000	15728,480	11513,430	16297,030	14,390	2,830	15,98
100000	62773,070	46216,970	65440,030	29,220	6,600	40,54

3- Représentation des graphes de variation du temps d'exécution T(N)

→ Meilleur cas :



→ Pire cas :



D'après le graphe ci-dessus, on remarque que :

- ✓ Approches adoptées au meilleur cas des algorithmes tri bulle, tri bulle optimisé et tri rapide sont plus performantes
- ✓ Approches adoptées au pire cas des algorithmes tri tas, tri rapide et tri distribution sont plus performantes

4- Etude du cout des algorithmes en utilisant Δt

→ Meilleur cas :

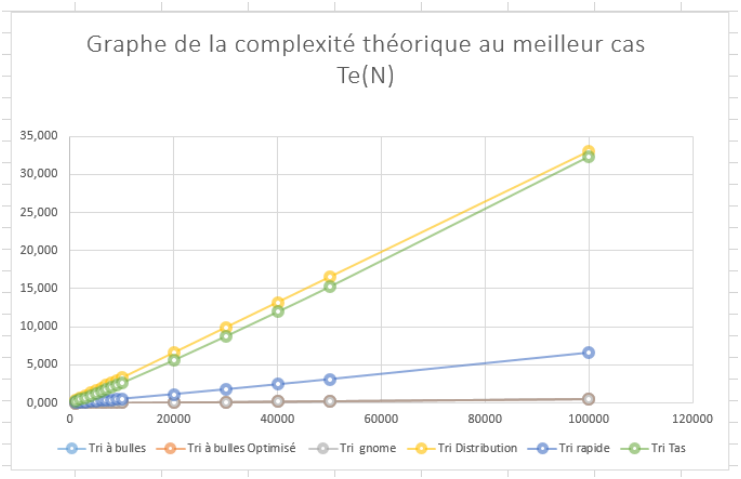
N	Tri à bulles	Tri à bulles Optimisé	Tri gnome	Tri Distribution	Tri rapide	Tri Tas
1000	0,005	0,005	0,005	0,330	0,040	0,200
2000	0,010	0,010	0,010	0,659	0,088	0,437
3000	0,015	0,015	0,015	0,989	0,139	0,688
4000	0,020	0,020	0,020	1,319	0,192	0,949
5000	0,025	0,025	0,025	1,648	0,246	1,216
6000	0,030	0,030	0,030	1,978	0,301	1,489
7000	0,035	0,035	0,035	2,308	0,358	1,766
8000	0,040	0,040	0,040	2,637	0,415	2,047
9000	0,045	0,045	0,045	2,967	0,473	2,331
10000	0,050	0,050	0,050	3,297	0,532	2,619
20000	0,100	0,100	0,100	6,593	1,144	5,610
30000	0,150	0,149	0,149	9,890	1,786	8,742
40000	0,200	0,199	0,199	13,186	2,448	11,966
50000	0,250	0,249	0,249	16,483	3,125	15,258
100000	0,499	0,498	0,498	32,965	6,650	32,379

→ Pire cas :

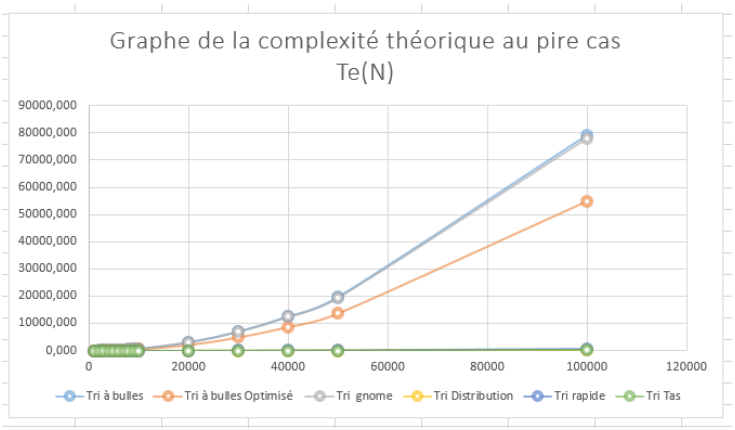
N	Tri à bulles	Tri à bulles Optimisé	Tri gnome	Tri Distribution	Tri rapide	Tri Tas
1000	7,912	5,477	7,800	0,330	0,060	0,200
2000	31,664	21,915	31,200	0,659	0,240	0,437
3000	71,256	49,312	70,200	0,989	0,540	0,688
4000	126,688	87,669	124,800	1,319	0,960	0,949
5000	197,960	136,986	195,000	1,648	1,500	1,216
6000	285,072	197,264	280,800	1,978	2,160	1,489
7000	388,025	268,501	382,200	2,308	2,940	1,766
8000	506,817	350,698	499,200	2,637	3,840	2,047
9000	641,449	443,855	631,800	2,967	4,860	2,331
10000	791,921	547,973	780,000	3,297	6,000	2,619
20000	3167,842	2191,945	3120,000	6,593	24,000	5,610
30000	7127,762	4931,918	7020,000	9,890	54,000	8,742
40000	12671,683	8767,890	12480,000	13,186	96,000	11,966
50000	19799,604	13699,863	19500,000	16,483	150,000	15,258
100000	79199,208	54799,726	78000,000	32,965	600,000	32,379

5- Représentation des graphes de variation du temps théoriques

→ Meilleur cas :



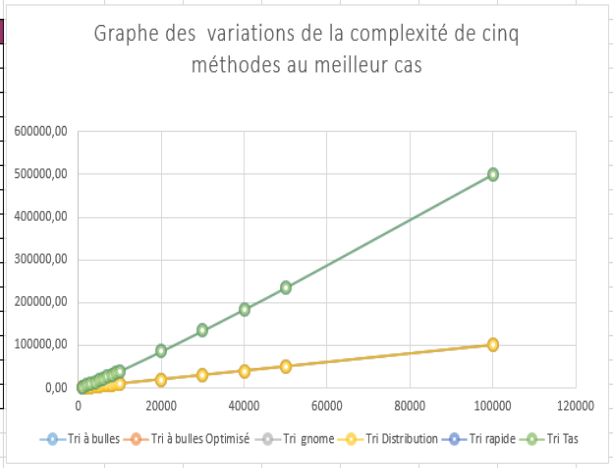
→ Pire cas :



6- Représentation des variations de la complexité :

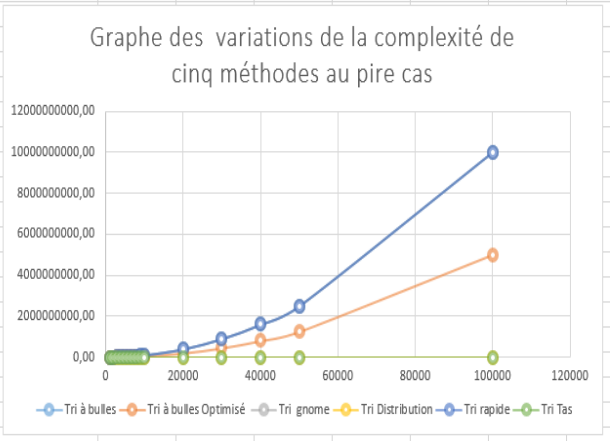
→ Meilleur cas :

N	Tri à bulles	Tri à bulles Optimisé	Tri gnome	Tri Distribution	Tri rapide	Tri Tas
1000	1000,00	1000,00	1000,00	1000,00	3000,00	3000,00
2000	2000,00	2000,00	2000,00	2000,00	6602,06	6602,06
3000	3000,00	3000,00	3000,00	3000,00	10431,36	10431,36
4000	4000,00	4000,00	4000,00	4000,00	14408,24	14408,24
5000	5000,00	5000,00	5000,00	5000,00	18494,85	18494,85
6000	6000,00	6000,00	6000,00	6000,00	22668,91	22668,91
7000	7000,00	7000,00	7000,00	7000,00	26915,69	26915,69
8000	8000,00	8000,00	8000,00	8000,00	31224,72	31224,72
9000	9000,00	9000,00	9000,00	9000,00	35588,18	35588,18
10000	10000,00	10000,00	10000,00	10000,00	40000,00	40000,00
20000	20000,00	20000,00	20000,00	20000,00	86020,60	86020,60
30000	30000,00	30000,00	30000,00	30000,00	134313,64	134313,64
40000	40000,00	40000,00	40000,00	40000,00	184082,40	184082,40
50000	50000,00	50000,00	50000,00	50000,00	234948,50	234948,50
100000	100000,00	100000,00	100000,00	100000,00	500000,00	500000,00



→ Pire cas :

N	Tri à bulles	Tri à bulles Optimisé	Tri gnome	Tri Distribution	Tri rapide	Tri Tas
1000	1000000,00	499500,00	1000000,00	1000,00	1000000,00	3000,00
2000	4000000,00	1999000,00	4000000,00	2000,00	4000000,00	6602,06
3000	9000000,00	4498500,00	9000000,00	3000,00	9000000,00	10431,36
4000	16000000,00	7998000,00	16000000,00	4000,00	16000000,00	14408,24
5000	25000000,00	12497500,00	25000000,00	5000,00	25000000,00	18494,85
6000	36000000,00	17997000,00	36000000,00	6000,00	36000000,00	22668,91
7000	49000000,00	24496500,00	49000000,00	7000,00	49000000,00	26915,69
8000	64000000,00	31996000,00	64000000,00	8000,00	64000000,00	31224,72
9000	81000000,00	40495500,00	81000000,00	9000,00	81000000,00	35588,18
10000	100000000,00	49995000,00	100000000,00	10000,00	100000000,00	40000,00
20000	400000000,00	199990000,00	400000000,00	20000,00	400000000,00	86020,60
30000	900000000,00	449985000,00	900000000,00	30000,00	900000000,00	134313,64
40000	1600000000,00	799980000,00	1600000000,00	40000,00	1600000000,00	184082,40
50000	2500000000,00	1249975000,00	2500000000,00	50000,00	2500000000,00	234948,50
100000	10000000000,00	4999950000,00	10000000000,00	100000,00	10000000000,00	500000,00



7- Analyse des résultats :

Dans le graphe ci-dessus, on remarque que :

→ Le Tri par tas et le tri rapide sont des algorithmes de tri rapides.

→ Le Tri à bulles, le tri par distribution et le tri gnome sont des algorithmes de tri lents.

Algorithmes de tri rapides :

Le tri par tas (complexité$O\ (n\ log\ n)$) est plus rapide que le tri rapide (complexité $O\ (n^2)$) lorsqu'on choisit le mauvais pivot dans le tri rapide, le. Mais lorsqu'on choisit le bon pivot dans le tri rapide, la complexité de ce dernier sera de l'ordre $\Omega\ (log\ n)$ donc il sera plus rapide que le tri par tas.

Algorithmes de tri lents :

Au pire cas, le tri par distribution (complexité$O\ (n\ log\ n)$) est plus rapide que le tri à bulles optimisé qui est à son tour plus rapide que le tri gnome et le tri à bulles.

VIII. Conclusion

L’importance des algorithmes de tri dans notre quotidien n’est plus à démontrer, la moindre requête à votre site préféré nécessite de trier les résultats pour les afficher.

Les algorithmes de tri peuvent être subdivisés en deux grandes classes : les algorithmes de tri rapides et les algorithmes de tri lents. Pour choisir un algorithme de tri plusieurs critères sont pris en considération tels que :

- ✓ La taille de la structure à trier.
- ✓ Le type de la structure à trier.
- ✓ L’état de l’entrée de la structure