جامعة هواري بومدين
للعلوم والتكنولوجيا
USTHB

# Les algorithmes de tri

Master 01 Ingénierie Des Logiciels 04/01/2023

Complexité des Algorithmes


| | | |
|---|---|---|
| BELHARRAT Yanis Zidane | 191931085131 | Groupe 01 |
| DOUDOU Amir | 191931081313 | Groupe 01 |

# Environnement de travail :

## Matériel :

L'implémentation des différents algorithmes étudiés ici ont été réalisés sur une machine ayant les spécifications suivantes :

- Processeur : AMD Ryzen 5 5850U
- Ram : 16gb.
- Système d'exploitation : Windows 11 64 bit.

## Logiciel :

L'implémentation et tests des algorithmes ont été faits avec :

- Editeur de texte et environnement de travail VS CODE.
- Langage C comme langage de programmation.

# Introduction :

Selon le dictionnaire, « trier » signifie « répartir en plusieurs classes selon certains critères ». De manière plus restrictive, le terme de « tri » en algorithmique est très souvent attaché au processus de classement d'un ensemble d'éléments dans un ordre donné. Par exemple, trier N entiers dans l'ordre croissant, ou N noms dans l'ordre alphabétique.

Cependant, certaines questions se posent, quel méthode de tri est la plus efficace ? La plus rapide ? La moins couteuse ? C'est pour tenter d'apporter une réponse à ces questions-là que les chercheurs continuent de découvrir de nouvelles méthodes de tri et de perfectionner celles existantes.

# Les tris qui seront étudiés sont les suivants :

- Tri à bulles
- Tri gnome
- Tri par distribution
- Tri rapide
- Tri par tas

- **Tri à bulles :**

Le tri à bulles consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés, ce processus est répété pour chaque élément du tableau et continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

Exemple :

Nous avons un tableau T :

| 8 | 15 | 5 | 11 | 6 |
|---|---|---|---|---|

Itération 01 : compare 8 et 15, rien ne change

| 8 | 15 | 5 | 11 | 6 |
|---|---|---|---|---|

Compare 15 avec 5 et permute

| 8 | 5 | 15 | 11 | 6 |
|---|---|---|---|---|

Compare 15 avec 11 et permute

| 8 | 5 | 11 | 15 | 6 |
|---|---|---|---|---|

Compare 15 avec 6 et permute

| 8 | 5 | 11 | 6 | 15 |
|---|---|---|---|---|

Itération 02 : Compare 8 avec 5 et permute

| 5 | 8 | 11 | 6 | 15 |
|---|---|---|---|---|

Compare 8 et 11, rien ne change, passe a 11

| 5 | 8 | 11 | 6 | 15 |
|---|---|---|---|---|

Compare 11 avec 6 et permute

| 5 | 8 | 6 | 11 | 15 |
|---|---|---|---|---|

Compare 11 avec 15, rien ne change

| 5 | 8 | 6 | 11 | 15 |
|---|---|---|---|---|

Itération 03 : Compare 5 avec 8, rien ne change, passe à 8

| 5 | 8 | 6 | 11 | 15 |
|---|---|---|---|---|

Compare 8 et 6, permute.

| 5 | 6 | 8 | 11 | 15 |
|---|---|---|---|---|

Le reste des itérations ne sont que des comparaisons et il n’y aura aucune permutation puisque le tableau est déjà trie.

**Procédure du TriBulle (E/S: un tableau T[n] d'entiers ; E/n :entier)**

```
void TriBulle (int T; int n) {
Int Changement = 1; //(variable booléenne)
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;
                                                        }
                                            }
                      }
}
```

**Procédure du TriBulleOpt (E/S: un tableau T[n] d'entiers ; E/n ;entier)**

```
 void TriBulleOpt (int T; int n) {
int m=n-1;
int Changement = 1; //(variable booléenne)
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=m-1;
                       }
}
```

## Complexité temporelle :

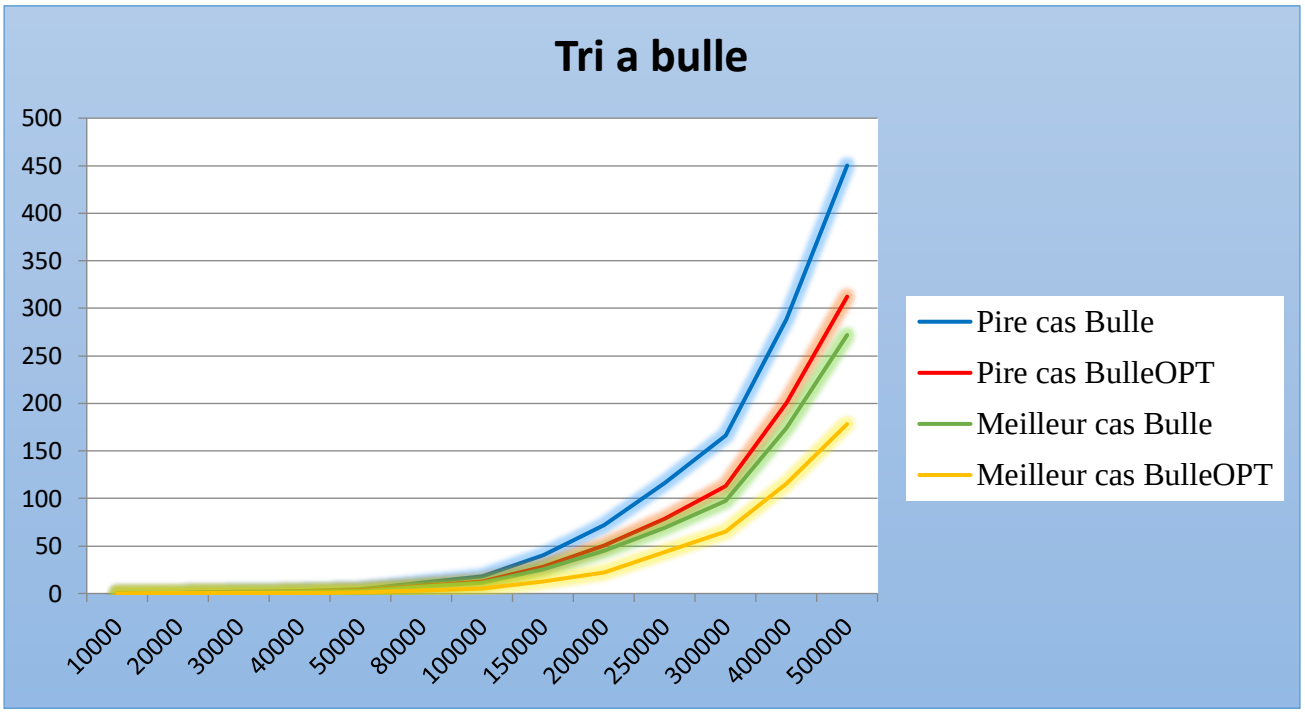
- Pire cas : La complexité temporelle de cet algorithme au pire cas est en $O(n^2)$, dans le cas où les valeurs du tableau sont triées dans le sens inverse, ce qui résulte en  comparaisons, donc $n(n-1)/2 = (n^2 - n)/2$ comparaisons et permutations.

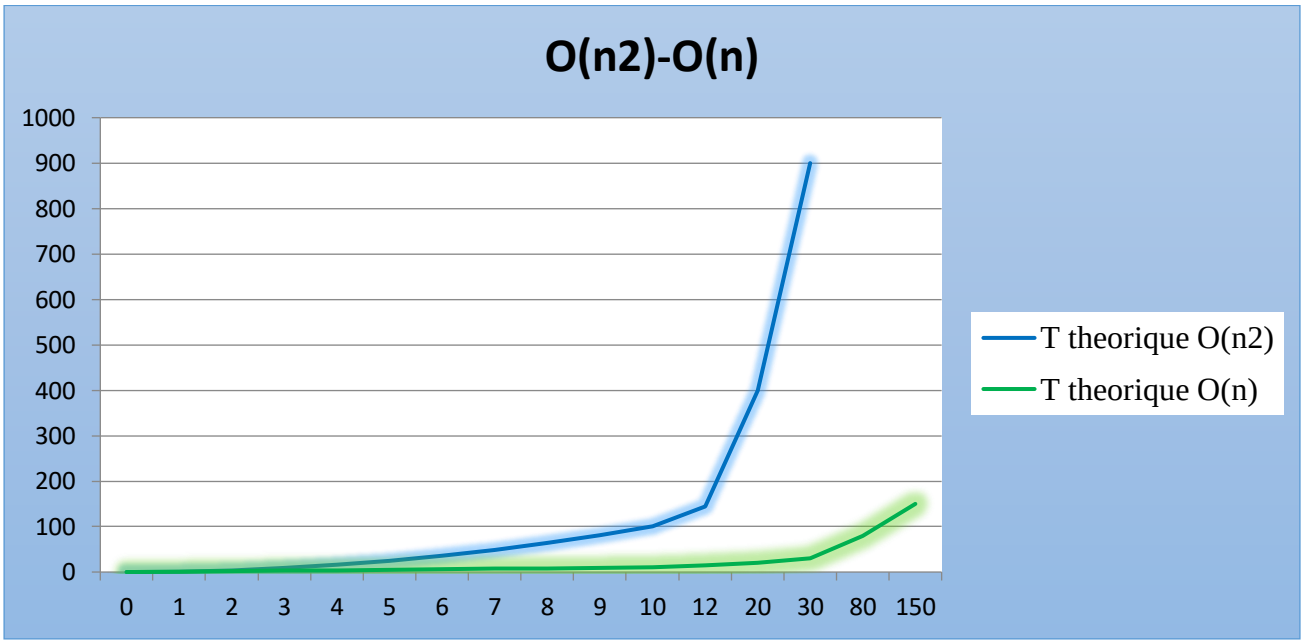
- Meilleur cas : Dans le meilleur cas, le tableau est déjà trie, donc on aura juste n-1 comparaisons, et aucune permutation ce qui fait que la complexité est en O(n-1) dans le cas de base et le cas optimisé.

## Résultats des exécutions (temps exécutions expérimental) :

| Taille N | Pire Cas Bulle | Pire Cas BulleOpt | Meilleur Cas Bulle | Meilleur Cas BulleOpt |
|---|---|---|---|---|
| 10000 | 0.17100 | 0.121000 | 0.110000 | 0.054000 |
| 20000 | 0.71400 | 0.523000 | 0.430000 | 0.216000 |
| 30000 | 1.59800 | 1.151000 | 0.966000 | 0.492000 |
| 40000 | 2.83300 | 2.027000 | 1.716000 | 0.862000 |
| 50000 | 4.49600 | 3.173000 | 2.728000 | 1.344000 |
| 80000 | 11.63000 | 8.039000 | 7.067000 | 3.497000 |
| 100000 | 17.89500 | 12.51900 | 11.162000 | 5.519000 |
| 150000 | 40.43800 | 28.08100 | 25.235000 | 12.907000 |
| 200000 | 72.11600 | 50.10300 | 44.925000 | 22.313000 |
| 250000 | 116.1460 | 78.63100 | 69.363000 | 43.427000 |
| 300000 | 166.5480 | 113.0610 | 97.541000 | 65.118000 |
| 400000 | 288.8380 | 200.5210 | 173.976000 | 115.81000 |
| 500000 | 450.1860 | 312.2580 | 271.906000 | 178.24700 |

## Comparaison des graphes des temps d'exécutions:




## Analyse et conclusion:

On peut voir depuis les graphes précédents que la procédure optimise de tri bulle et effectivement plus rapide que la procédure de base et ceci dans les deux cas. Pour ce qui est du pire cas, ou le tableau initial est trie à l'envers, on a un temps d'exécution qui est presque deux fois plus long que le temps d'exécution au meilleur cas ou le tableau initial est déjà trié.
On peut aussi conclure qu'il y a une relation entre le temps d'exécution expérimental et le temps d'exécution théorique [$O(n^2)$ au pire cas, $O(n)$ au meilleur cas], puisqu'il y a une similarité entre les deux graphes.

- **Tri gnome :**

Le tri gnome, aussi appelé tri stupide est basé sur le concept d'un nain de jardin (gnome) triant ses pots de fleurs. Un nain de jardin trie les pots de fleurs par la méthode suivante :

Il regarde le pot de fleurs à côté de lui et le précédent ; s'ils sont dans le bon ordre, il avance d'un pot, sinon il les échange et recule d'un pot.

S'il n'y a pas de pot précédent (il est au début de la ligne du pot), il avance ; s'il n'y a pas de pot à côté de lui (il est à la fin de la ligne de pot), il a terminé.

Exemple :

Nous avons un tableau T :

| 8 | 15 | 5 | 11 | 6 |
|---|---|---|---|---|

Itération 01 :

| 8 | 5 | 15 | 11 | 6 |
|---|---|---|---|---|

Itération 02 :

| 5 | 8 | 15 | 11 | 6 |
|---|---|---|---|---|

Itération 03 :

| 5 | 8 | 11 | 15 | 6 |
|---|---|---|---|---|

Itération 04 :

| 5 | 8 | 11 | 6 | 15 |
|---|---|---|---|---|

Itération 05 :

| 5 | 8 | 6 | 11 | 15 |
|---|---|---|---|---|

Itération 06 : (résultat final de T)

| 5 | 6 | 8 | 11 | 15 |
|---|---|---|---|---|

## Complexité temporelle :

- Pire cas : La complexité temporelle de cet algorithme au pire cas est en $O(n^2)$, dans le cas où les valeurs du tableau sont triées dans le sens inverse, ce qui résulte en n*n comparaisons.
- Meilleur cas : Dans le meilleur cas, le tableau est déjà trie, donc on aura juste n comparaisons, ce qui fait que la complexité est en O(n).

## Procedure du tri-gnome en pseudo-code :

```
Procedure tri-gnome(tab[])
pos ← 2
TANT QUE (pos < longueur(tab) )
    SI (tab[pos] >= tab[pos-1])
        pos ← pos + 1
    SINON
        Permuter tab[pos] ET tab[pos-1]
        SI (pos > 2)
            pos ← pos - 1
        FIN SI
    FIN SI
FIN TANT QUE
    FIN PROCEDURE
```

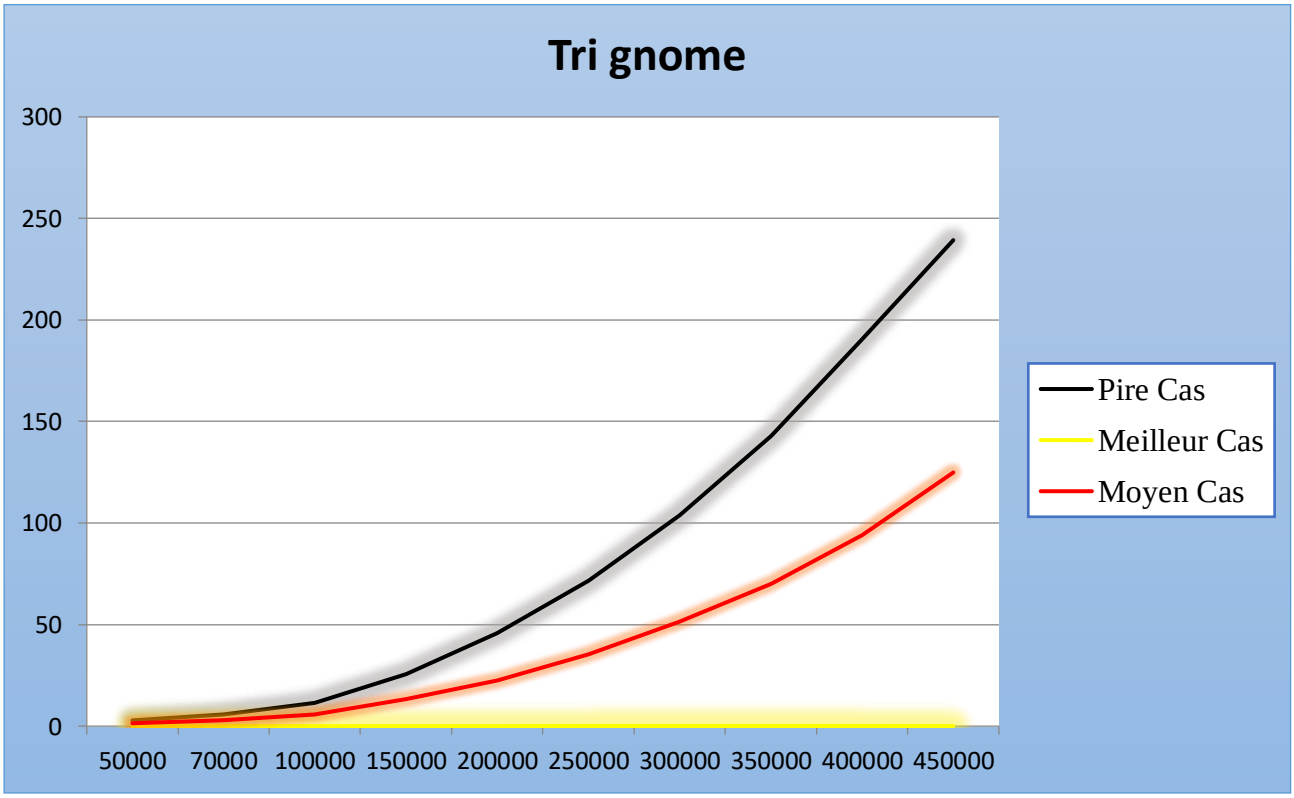
## Procedure du tri-gnome en C :

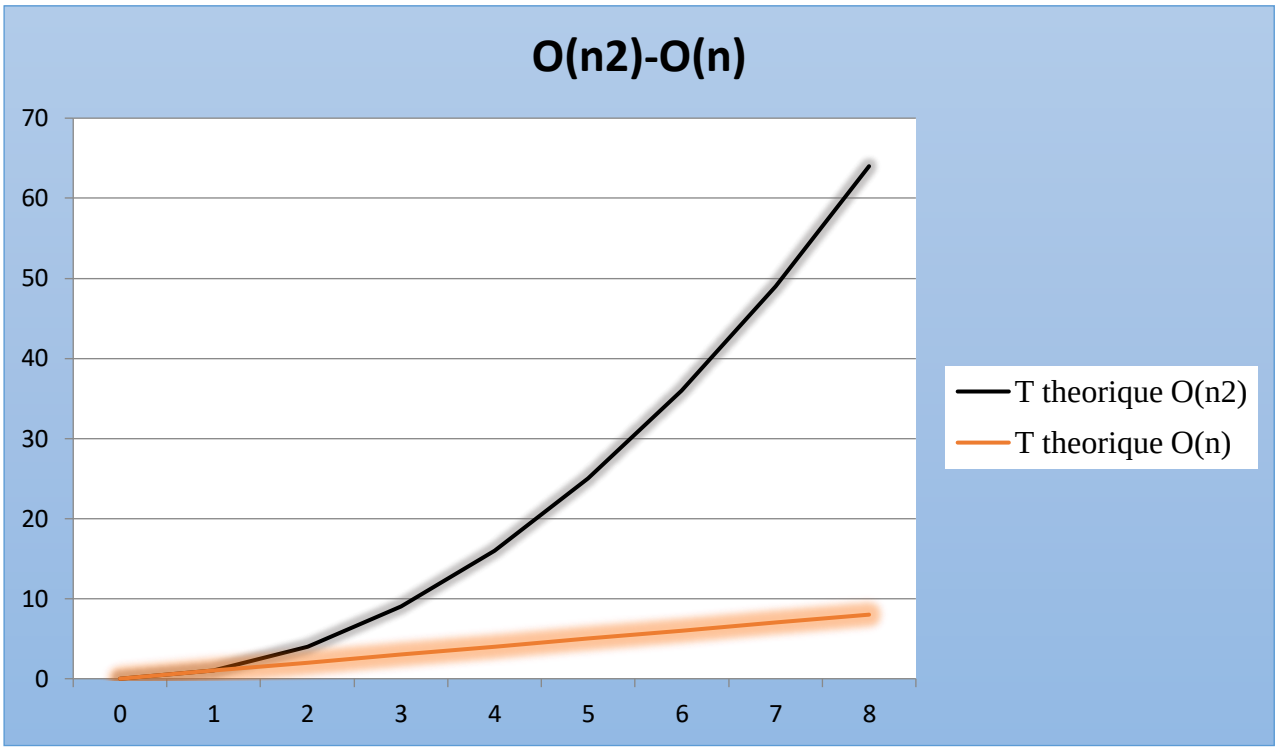
```
void tri_gnome (long int* tab, long b) {
  long pos=1;
  while(pos<b){
if(tab[pos] >= tab[pos-1])
 pos++;
  else{
long t = tab[pos-1];
      tab[pos-1]= tab[pos];
      tab[pos] = t;
if (pos > 1){pos--;}
}           }                    }
```

## Résultats des exécutions (temps exécutions expérimental) :

| Taille N | Pire Cas | Meilleur Cas | Moyen Cas |
|---|---|---|---|
| 50000 | 2.806s | 0.000s | 1.401s |
| 70000 | 5.592s | 0.000s | 2.765s |
| 100000 | 11.475s | 0.000s | 5.749s |
| 150000 | 25.731s | 0.000s | 13.470s |
| 200000 | 45.846s | 0.000s | 22.572s |
| 250000 | 71.757s | 0.001s | 35.429s |
| 300000 | 103.693s | 0.001s | 51.395s |
| 350000 | 142.954s | 0.001s | 70.218s |
| 400000 | 190.500s | 0.001s | 93.965s |
| 450000 | 239.30s | 0.00143s | 124.916s |
| 700000 | | 0.00142s | 285.744s |
| 800000 | - | 0.00143s | 375.694s |

## Comparaison des graphes des temps d'exécutions:






## Analyse et conclusion:

On peut voir depuis les graphes précédents que pour le pire cas, ou le tableau initial est trie à l'envers, un temps d'exécution qui est presque deux fois plus long que le temps d'exécution au meilleur cas ou le tableau initial est déjà trié.

On peut aussi conclure qu'il y a une relation entre le temps d'exécution expérimental et le temps d'exécution théorique [$O(n^2)$ au pire cas, $O(n)$ au meilleur cas], puisqu'il y a une similarité entre les deux graphes.

## Remarque :

On a remarqué dans les comparaisons précédentes que la complexité théorique est plus élevé, ceci est dû au fait que la complexité expérimental dépend de nombreuses choses, tels que la machine utilisée (processeur, mémoire), ainsi que les langages de programmation.

- **Tri par distribution :**

Le tri par distribution, aussi appelé tri par base est un tri non comparatif.

Le principe de l'algorithme est le suivant :

- On commence par trier les valeurs en regardant seulement chiffre du nombre.
- On passe ensuite au chiffre suivant et on continue de trier jusqu'au dernier chiffre de la plus grande valeur.

On peut voir le mécanisme dans l'exemple suivant :

Nous avons un tableau T :

| | | | | |
|---|---|---|---|---|
| 88 | 15 | 52 | 12 | 60 |

Itération 01 :

| | | | | |
|---|---|---|---|---|
| 60 | 52 | 12 | 15 | 88 |

Itération 02 : (résultat final de T)

| | | | | |
|---|---|---|---|---|
| 12 | 15 | 52 | 60 | 88 |

## Fonction clé(E/ x, i : entier) : entier ;

```
int cle(int x, int i){
  int x1=pow(10,i);   // exemple si i=2, le nbr sera divisé sur 100 (pow est la fonction puissance dans C, eq a 10^i.
  return (x/x1)%10;   //on a besoin du modulo pour retourner le chiffre
 }
```

## Fonction TriAux(T, n, i) entier;

```
void TriAux(int *tab,int nb,int i){
int k,j,val;
for(k=0;k<nb;k++)
{  val=tab[k];
   j=k;
   while((j>0) && (cle(val,i)<cle(tab[j-1],i)))
   {      tab[j]=tab[j-1];      j--;   }
   tab[j]=val;    }   }
```

## Fonction TriBase(T, n, k) entier;

```
void TriBase(int *tab,int nb,int k){ //tab=tableau en entree, nb=taille
                                     //du tableau, k=  le nombre de
                                       chiffre à trier, si k=0 on trie seulement
                                       pour les premiers chiffres etc.
  for(int j=0;j<=k;j++){        triAux(tab,nb,j);                    }
                                  }
```

## Complexité temporelle :

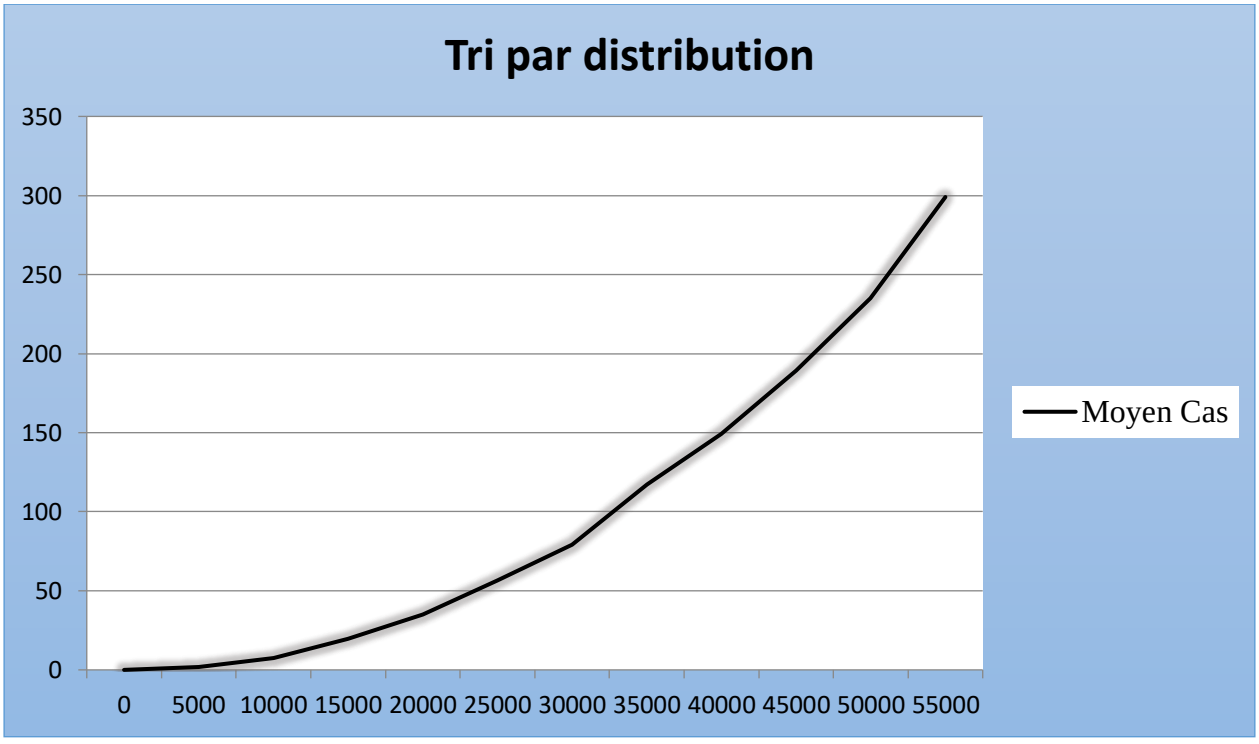
La complexité de ce tri dépend forcément de l'algorithme de tri utilisé dans notre boucle principale, et puisqu'il nous a été demande que la fonction TriAux doit s'exécuter en un temps linéaire en fonction de la taille n du tableau, on a utilisé une variante du tri insertion puisque sa complexité est linéaire au moyen cas.

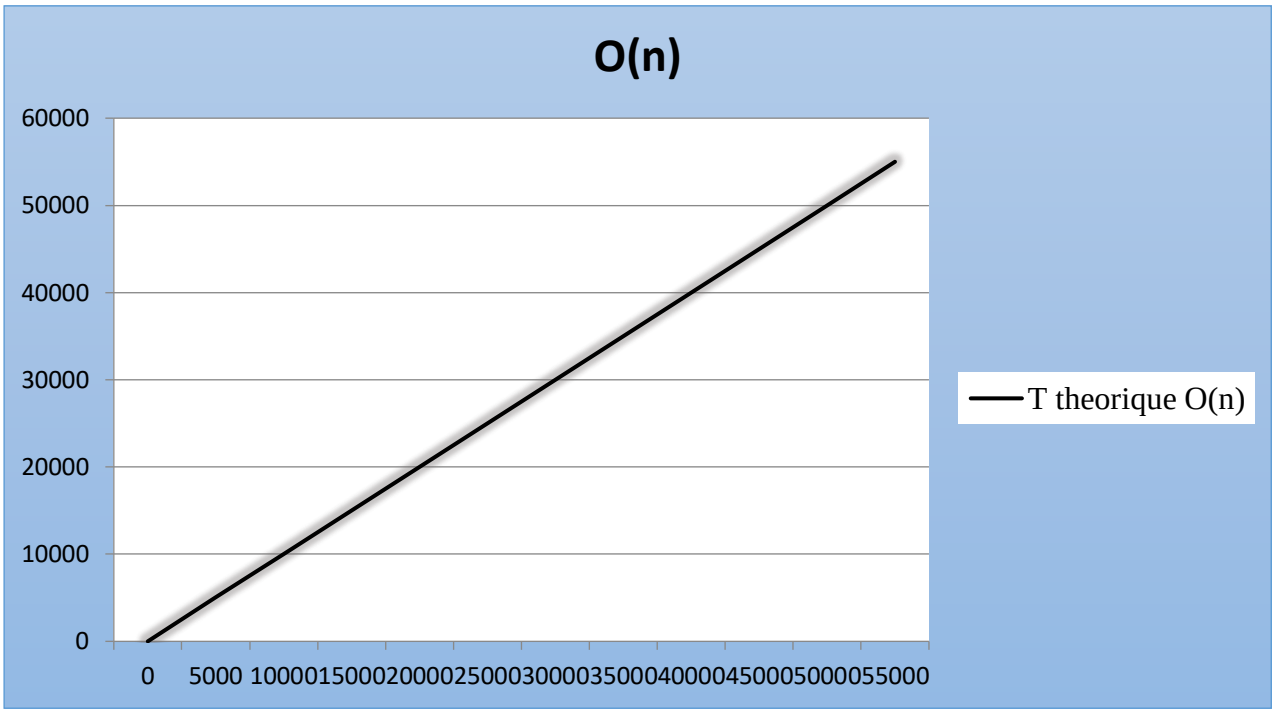
Donc Pour un tableau de taille n d'éléments codés sur d chiffres pouvant prendre k valeurs possibles, le temps d'exécution est en O(d*n), dans notre cas on a $0 \leq d \leq 9$, donc la complexité est en O(n).

## Résultats des exécutions (temps exécutions expérimental) :

| Taille N | Moyen Cas |
|---|---|
| 2000 | 0.483000s |
| 5000 | 1.828000s |
| 10000 | 7.477000s |
| 15000 | 19.536000s |
| 20000 | 35.209000s |
| 25000 | 56.330000s |
| 30000 | 79.347000s |
| 35000 | 117.223000s |
| 40000 | 149.153000s |
| 45000 | 189.166000s |
| 50000 | 234.897000s |
| 55000 | 299.021000s |

## Comparaison des graphes des temps d'exécutions:






## Analyse et conclusion:

On peut voir depuis les graphes précédents le temps d'exécution de tri par distribution au pire cas, ainsi que le temps d'exécution théorique.
On remarque une ressemblance entre les deux, ce qui fait qu'y a une relation entre le temps expérimental et théorique O(n).

Remarque : la complexité de la boucle principale qui trie est linéaire.

- **Tri rapide:**

Le tri rapide est l'algorithme de tri interne le plus efficace, Son principe est de parcourir le tableau = (1, 2, ... , n) en le divisant systématiquement en deux sous-tableaux T1 et T2. L'un est tel que tous ses éléments sont inférieurs à tous ceux de l'autre tableau et en travaillant séparément sur chacun des deux sous-tableaux en réappliquant la même division à chacun des deux sous-tableaux jusqu'à obtenir uniquement des sous-tableaux à un seul élément.

C'est un algorithme dichotomique qui divise donc le problème en deux sous-problèmes dont les résultats sont réutilisés par recombinaison,

Dans le tri rapide, le choix du pivot est essentiel. Dans l'implémentation ci-dessus, on se contente de prendre le dernier élément du tableau. Mais d'autres stratégies sont possibles.

Exemple :

Nous avons un tableau T :

| 4 | 11 | 1 | 18 | 8 |
|---|---|---|---|---|

Itération 01 : on divise en deux sous tableaux, le pivot=8 (élément à droite)

T1:

| 4 | 1 |
|---|---|

T2:

| 11 | 18 |
|---|---|

T= T1 + pivot + T2 =

| 4<8 | 1<8 | 8 | 11>8 | 18>8 |
|---|---|---|---|---|

2eme division : pivot =1

T11= [] vide

T12= [4]

T1=T11 + pivot + T22 =

| 1 | 4 |
|---|---|

3eme division : pivot = 18

T21= [11]

T22= [] vide

T2=T21 + pivot + T22

| 11 | 18 |
|---|---|

Tableau final :

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 8 | 11 | 18 |

Dans un tableau plus grand, on aura beaucoup plus de sous-tableaux, on continue jusqu'à ce que le tableau initial soit trié.

## Code C de la procedure Trirapide et Partitionner :

```
void tri_rapide (int T[], int g, int d)
{
int piv;
if(g < d)
{
piv= partition (T, g, d);
tri_rapide(T,g,piv-1);
tri_rapide(T,piv+1,d);
}
}
```

```
int partition(int T[], int g, int d)
{
int ipiv;
int i,j,temp;
ipiv=T[d];
i=g-1;
j=d;
while(i<=j)
{ 1 3 12 13 7 11 9
do{i++;}while(T[i]<ipiv && i<=d)
do{j--;}while (T[j]>ipiv && j>=g)
if(i<j)
```

```
        {
        temp=T[i];          //permutation des cases
        T[i]=T[j];
         T[j]=temp;
         }
}
temp=T[i];          //derniere permutation avec le pivot
T[i]=T[d];
T[d]=temp;
return i;
}
```

## Complexité temporelle :

- Pire cas : La complexité temporelle de cet algorithme au pire cas est en $O(n^2)$, dans le cas où le pivot choisi est le plus grand ou le plus petit élément a chaque fois.
- Moyen cas : La complexité temporelle de cet algorithme au moyen cas est en O(nlogn) dans le cas où les tableaux divisés sont de tailles différentes.

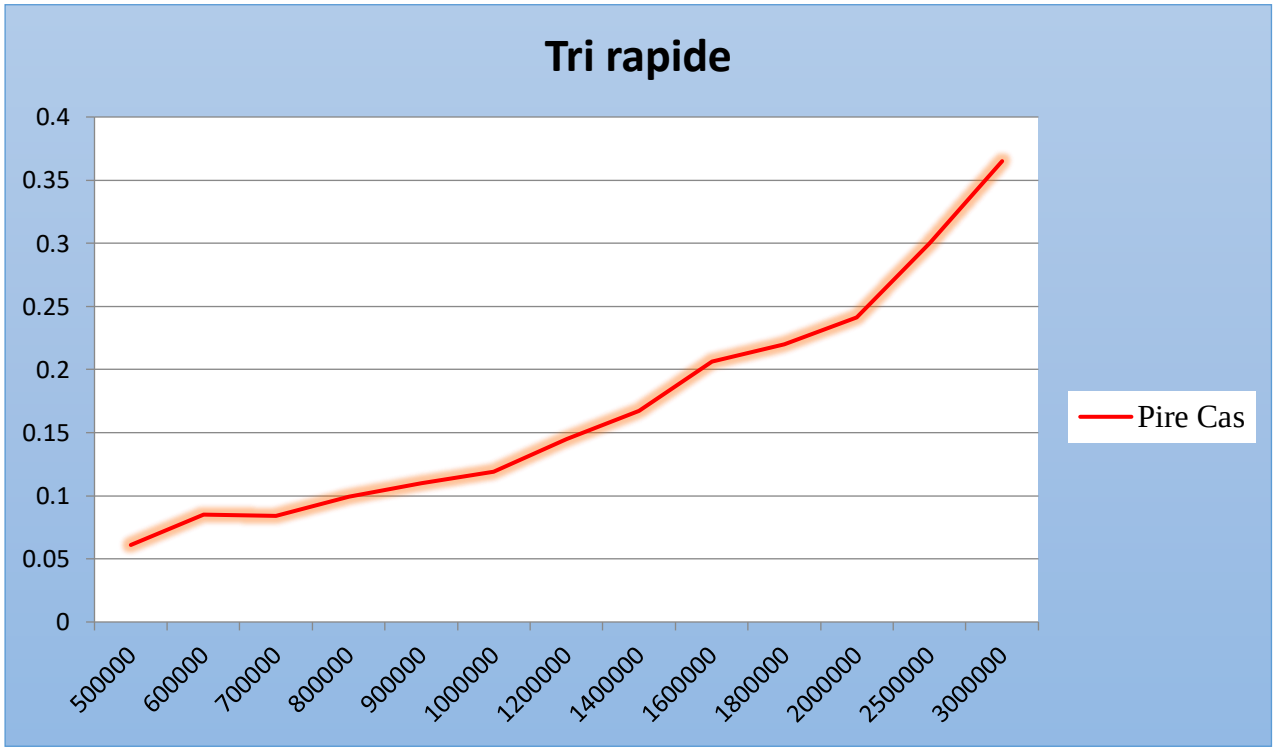
- Meilleur cas : Dans le meilleur cas, le tableau est toujours divise en 2 parties égales, ce qui fait que la complexité est en O( nlogn).

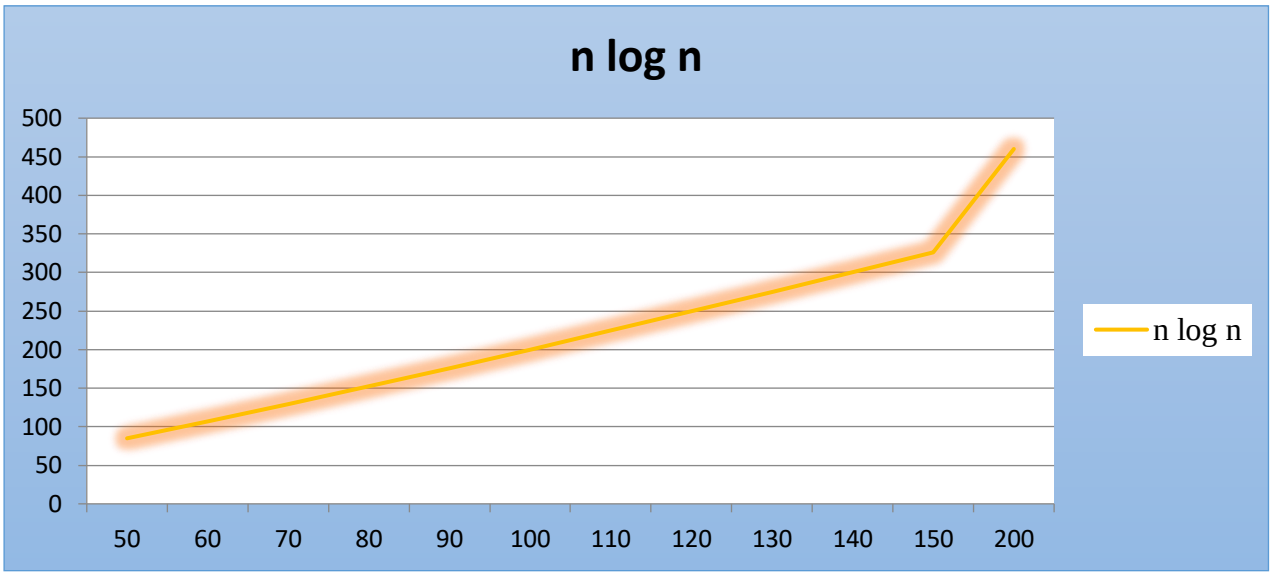
## Résultats des exécutions (temps exécutions expérimental) :

L'algorithme tri rapide s'exécute en un temps extrêmement rapide, donc nous avons pris des valeurs assez grandes.

| Taille N | Pire Cas |
|---|---|
| 500000 | 0.061000s |
| 600000 | 0.085000 |
| 700000 | 0.084000 |
| 800000 | 0.099000 |
| 900000 | 0.110000 |
| 1000000 | 0.119000 |
| 1200000 | 0.145000 |
| 1400000 | 0.167000 |
| 1600000 | 0.206000 |
| 1800000 | 0.220000 |
| 2000000 | 0.241000 |
| 2500000 | 0.300000 |
| 3000000 | 0.365000 |

## Comparaison des graphes des temps d'exécutions:






## Analyse et conclusion:

On peut voir depuis les graphes précédents le temps d'exécution de tri rapide au moyen cas, ainsi que le temps d'exécution théorique O(nlogn).
On remarque que le tri rapide a donné des temps d'exécutions extrêmement rapides même pour des valeurs n très grandes, ce qui fait de lui le tri le plus efficace étudié dans notre projet, et même compare à tous les tris existants.
On peut aussi voir une corrélation entre les deux graphes des temps d'exécutions expérimental et théorique O(nlogn).

## Equation de récurrence du tri rapide :

Nous allons calculer l'équation de récurrence dans le cas où les tableaux partitionnés sont toujours égales à n/2, on a aussi deux appels récursifs.

On obtient l'équation suivante : $\mathbf{T(n)=2\,T(\frac{n}{2})+n}$ / $\mathbf{T(1)=C}$ (constante)

$\mathbf{T(n)=2[2\,T(\frac{n}{4})+\frac{n}{2}]+n}$ // on démontre que le tri est en O(nlogn)

$\mathbf{T(n)=2^2\,T(\frac{n}{4})+\frac{2n}{2}+n}$

$\mathbf{T(n)=2^2[2\,T(\frac{n}{8})+\frac{2n}{4}]+\frac{2n}{2}+n = 2^3\,T(\frac{n}{8})+\frac{2^2n}{4}+\frac{2n}{2}+n}$

$\mathbf{T(n)= 2^3\,T(\frac{n}{8})+\frac{4n}{4}+\frac{2n}{2}+n = 2^3\,T(\frac{n}{8})+n+n+n}$

On continuant, on obtient : $\mathbf{T(n)=2^k\,T(\frac{n}{2^k})+k\,n}$ // On pose $\mathbf{n=2^k}$

$\mathbf{k=\log_2 n}$ ➔ $\mathbf{T(n)=2^{\log n}\,T(\frac{n}{2^{log_2 n}})+\log_2 n * n}$ ➔ **T(n)=nC+nlogn** CQFD

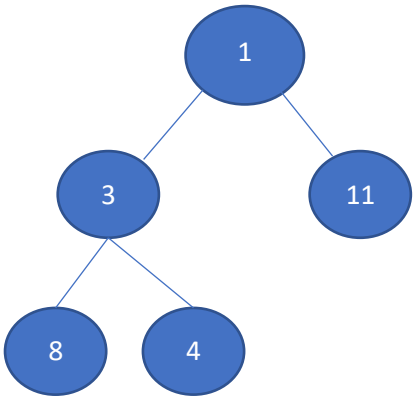
- **Tri par tas:**

L'algorithme du tri par tas repose sur un élément fondamental : le tas (d'où son nom). En effet, ce tri crée un tas min du tableau donné en entrée, et le parcourt afin de reconstituer les valeurs triées dans notre tableau.

**Exemple :**

On prend la suite de nombres suivante que l'on va trier dans l'ordre croissant avec le tri par tas : 8 4 11 3 1

*1ère phase :* créer le tas (le min dans notre exemple est 1)




2ème étape : parcourir le tas pour trier les éléments

Pour trier les éléments grâce à notre tas, on retire la racine à chaque tour (l'élément le plus petit de notre tas, et donc de notre tableau), on le range à sa place définitive, et on entasse le dernier nœud du tas pour combler le trou de la racine mais aussi respecter les règles d'un tas.

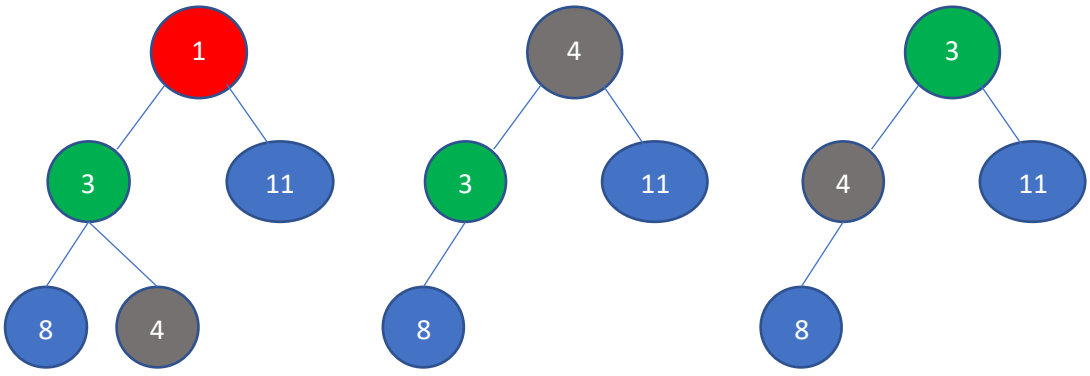



Tableau:

| 1 |
|---|

La racine du tas (en rouge) est placée dans le tableau (1) et le dernier noeud (en gris) va remplacer la racine (4), mais il ne faut pas oublier de l'entasser pour respecter les règles du tas min (3).

On continue ces opérations tant que le tas contient des éléments :

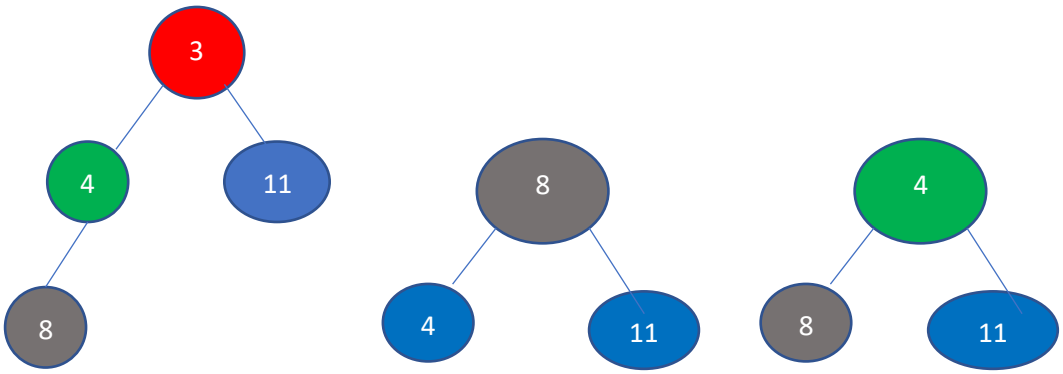



Tableau :

| 1 | 3 |
|---|---|

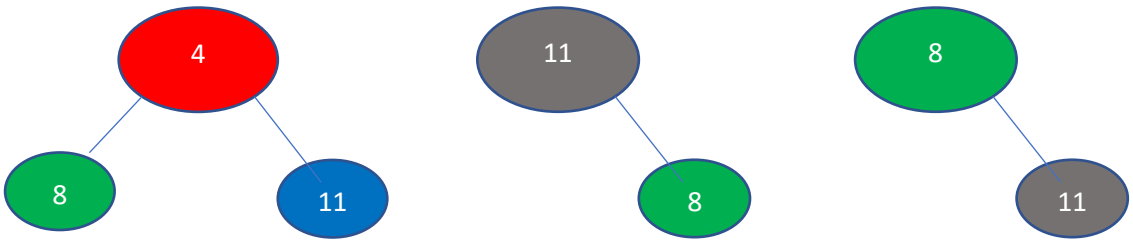



Tableau :

| 1 | 3 | 4 |
|---|---|---|

…

Tableau final :

| 1 | 3 | 4 | 8 | 11 |
|---|---|---|---|---|

## Complexité temporelle :

Le tri par tas est divisé en 2 phases :

- La transformation du tableau de N éléments en un tas, cette phase est effectuée en temps linéaire en O(n).
- Tandis que la phase la plus coûteuse de l'algorithme est la seconde boucle, c'est-à-dire l'extraction des éléments du tas:
- Extraire le minimum du tas en O(log n):

- Échanger le dernier élément avec le premier; O(1)
- Diminuer la taille du tas de 1; O(1)
- Entasser le premier élément; O(log n)

Donc la complexité est en O(n log n).

## Procedure du tri-tas en C :

```
void permuter (int* tab [], int i, int j)
{ int r;
        r=tab[i];
        tab[i]=tab[j];
        tab[j]=r;
}
```

```
void remonter (int* tab[], int n, int i)
void remonter ( int tab[], long n, long i)
{
        if (i==0) return;
        if (tab[i]<tab[(i-1)/2]) {
                permuter (tab, i, (i-1)/2);
                remonter (tab, n, (i-1)/2); }
}
```

```
void redescendre ( int tab[], const int n, int i)
{
   for (int rs=0;rs<n;rs++){printf("T[%d]=%ld \t",rs,tab[rs]);}
   printf("\n \n \n");
   long min;
   if (2i+1>=n) return;
      if(i==0 && booli==1){
       min=1; }
      else if(i==0 && booli==0){
       if(tab[1]<tab[2])
       min=1;
       else    min=2;
                                        }
    else{    if(2i+2>=n && 2i+1<n)   min=2i+1;
                  else if (tab[2i+2]<tab[2i+1])
                        min=2i+2;
                        else
                        min=2i+1;
        }
     if (tab[min]<tab[i])
       {      permuter (tab, min, i);
              printf(" permute %ld avec %ld \n",tab[min],tab[i]);
              redescendre (tab, n, min);
       }
}
```

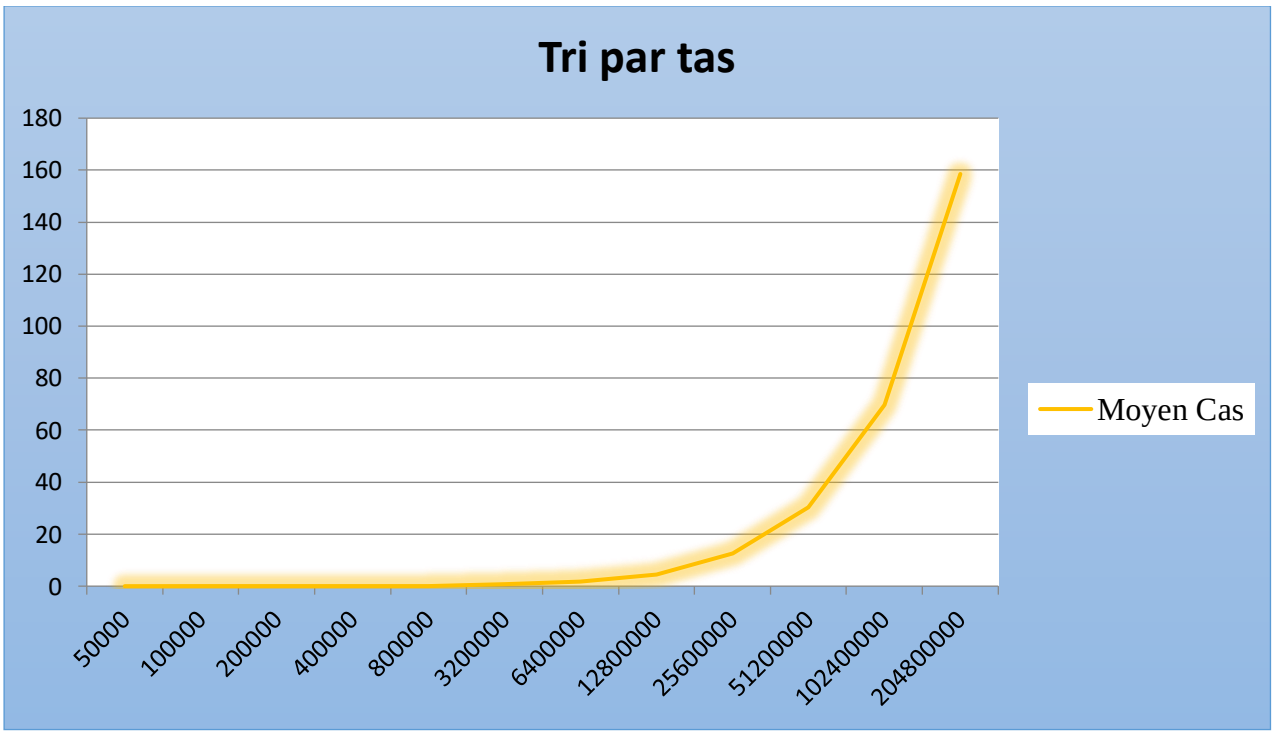
```
void organiser( int tab[], int n )
{       long i;
        for(i = 1 ; i < n ; i++)
        remonter(tab, n, i);
}
```

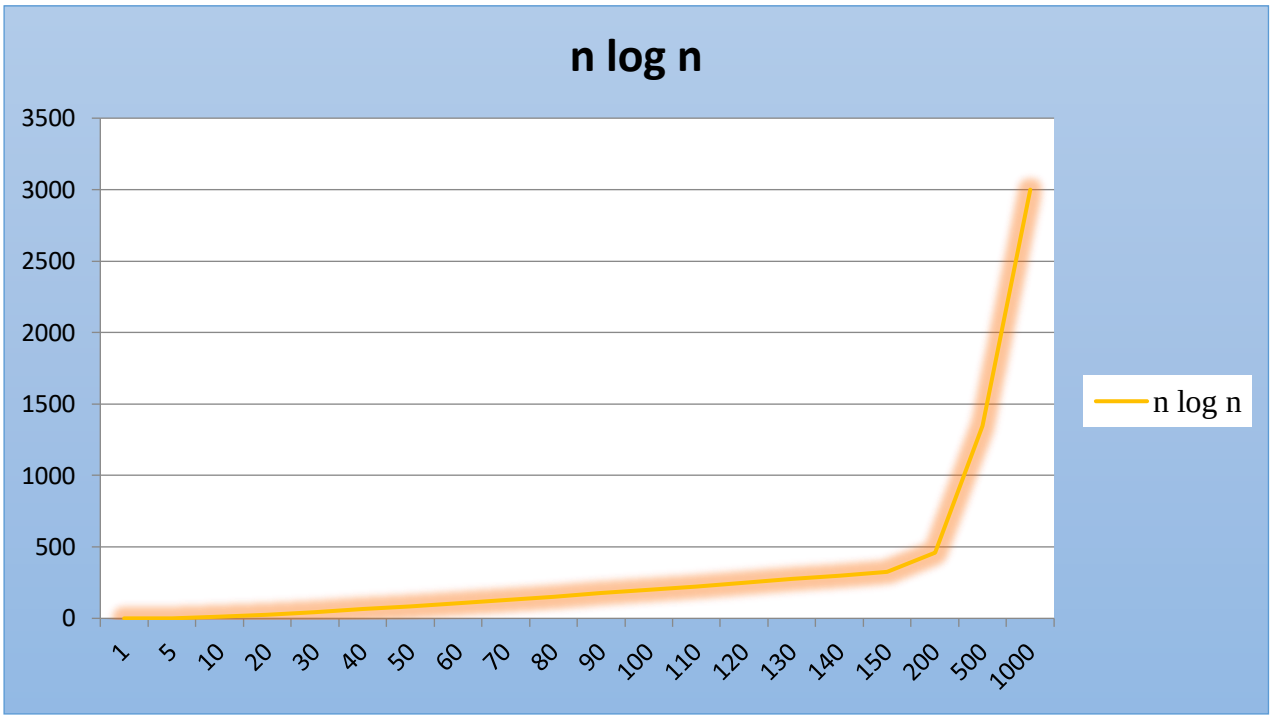
```
long int tri_tas(int* tab[], long n )
{       long i=n-1,j=0;
        long int* tab2=( long int )malloc(n sizeof(long int*));
        organiser(tab, n);
                while (i>=0)
            {if (i==2){  booli=1; }
              permuter(tab, 0, i);
              tab2[j]=tab[i];
              redescendre(tab, i, 0);
              i--;j++;
            } return tab2;
}}
```

## Résultats des exécutions (temps exécutions expérimental) :

| Taille N | Moyen Cas |
|---|---|
| 50000 | 0.007000s |
| 100000 | 0.015000s |
| 200000 | 0.033000s |
| 400000 | 0.070000s |
| 800000 | 0.152000s |
| 3200000 | 0.706000s |
| 6400000 | 1.75000s |
| 12800000 | 4.479000s |
| 25600000 | 12.512000s |
| 51200000 | 30.328000s |
| 102400000 | 69.605000s |
| 204800000 | 158.537000s |

## Comparaison des graphes des temps d'exécutions:

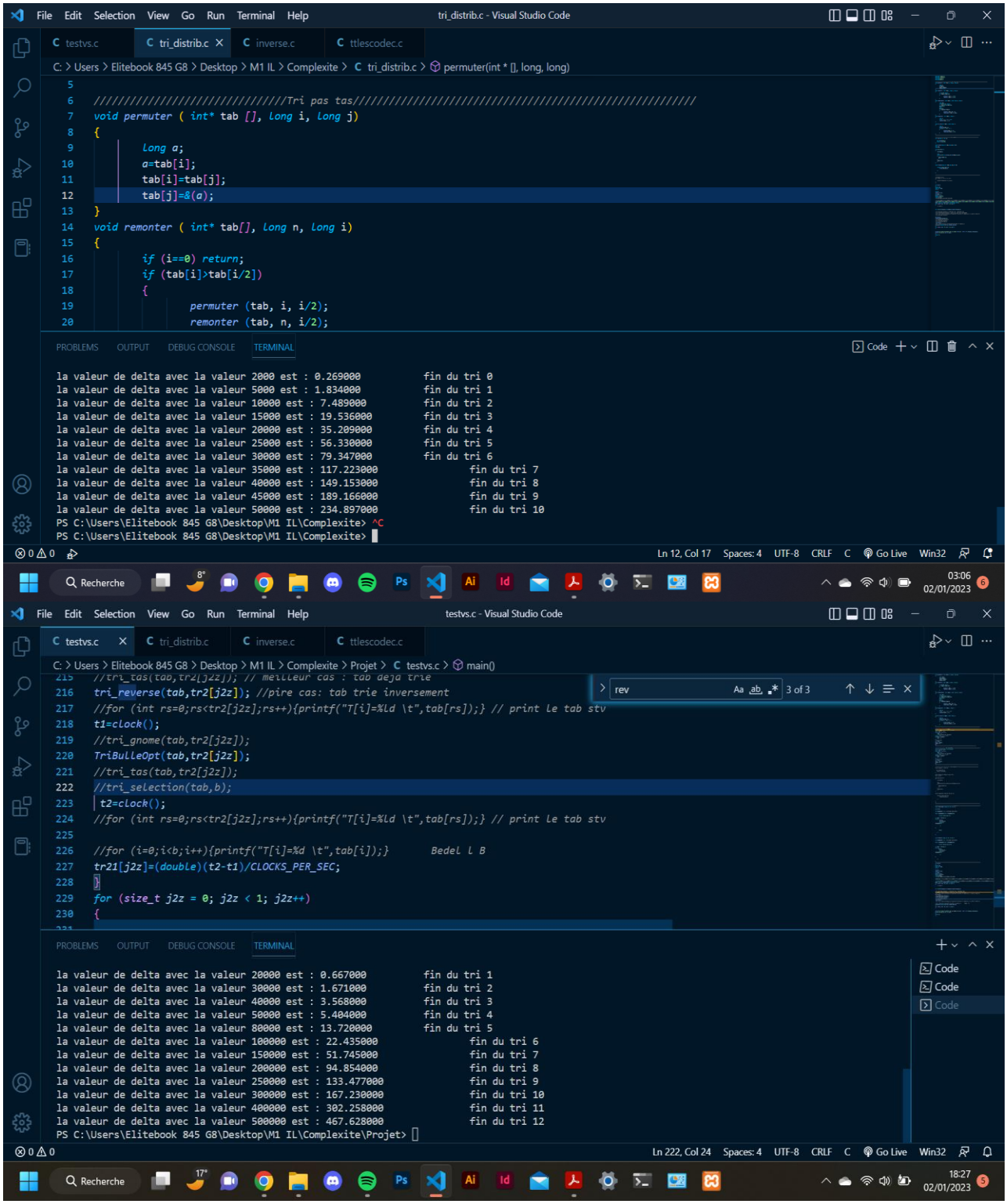





## Analyse et conclusion:

On peut voir depuis les graphes précédents le temps d'exécution de tri par tas au moyen cas, ainsi que le temps d'exécution théorique O(nlogn).
On remarque qu'il y a une ressemblance entre les deux graphes des temps d'exécutions expérimental et théorique O(nlogn), donc on conclut qu'il y a une relation entre les deux.

# Démonstration de quelques tests effectués:



# Conclusion générale

Les algorithmes de tri ont une grande importance et sont fondamentaux dans certains domaines, c'est pour cela que les informaticiens tentent toujours de trouver des solutions plus efficaces et plus rapide.

Pour ce qui est des tris étudié ici, on peut distinguer deux catégories : les tris simples ( tri gnome, tri à bulle, tri par distribution) et les tris complexes(tri rapide, tri par tas).

Les tris simples ont en général une complexité quadratique, tandis que les tris complexes sont un peu plus difficiles à programmer, mais ont une efficacité accrue avec une complexité se rapprochant du (nlogn).

Apres avoir fait subir une multitude de tests aux algorithmes vue précédemment, on peut conclure que le tri rapide est de loin le meilleur choix et le plus efficace parmi les autres tris.