COMPLEXITÉ

Master 1 IL
Groupe 2
2018

Rapport de TP Mini Projet COMPLEXITÉ : Les TRI

BOUDOUR Mehdi – HAICHEUR Zakaria/ 201500008386/ TP: Les TRI

[ALGORITHMIQUE AVANCÉE ET COMPLEXITÉ]

E-mail : geronimotoutcourt@gmail.com

Ce document présent les solutions en 5 étapes : (1) les algorithmes écris en pseudo-code. (2) le calcul de la complexité au pire des cas. (3) Implémentation de l'algorithme en langage C. (4) capture de l'exécution de l'algorithme. (5) représentation graphique de l'évolution du temps d'exécution en fonction de N. Le programme C complet contenant les détails (affichage, calcul du temps d'exécution,...) d'implémentation est présenté à la fin du document.

I. Algorithme *Tri à bulles*:

Parcourir le tableau T de taille N du dernier au premier élément (presque . . .), avec un indice i. A chaque étape, la partie du tableau située à droite de i est considérée comme triée. On parcourt alors la partie de gauche (partie non triée) avec un indice j. Pour chaque j, si T[j − 1] > T[j], on les permute.

Principe:

Comparer 2 à 2 les éléments adjacents
Les échanger s'ils ne sont pas ordonnés
Comme les bulles, les plus grands éléments remontent en fin de liste.

2, 56, 4, -7, 0, 78, -45, 10
2, 4, -7, 0, 56, -45, 10, 78
2, -7, 0, 4, -45, 10, 56, 78
-7, 0, 2, -45, 4, 10, 56, 78
-7, 0, -45, 2, 4, 10, 56, 78
-7, -45, 0, 2, 4, 10, 56, 78
-45, -7, 0, 2, 4, 10, 56, 78

Algorithme :

```
PROCDURE TRIBULLE(E/S T[N]: TABLEAU D' ENTIER ;E/ N:ENTIER)
    CHANGEMENT : BOOLEEN;
DEBUT
    CHANGEMENT = VRAI;
    TANT QUE (CHANGEMENT=VAIR) FAIRE
        CHANGEMENT = FAUX
        POUR I = 1 JUSQU'A N-1 FAIRE
            SI (T[I]>T[I+1]) ALORS
              PERMUTER(T[I],T[I+1]);
              CHANGEMENT=VRAI;
            FIN SI;
        FIN POUR;
    FIN TANT QUE;
FIN;
```

Boucle Interne

Boucle Externe

Complexité :

Au pire des cas : T est initialement trié de façon décroissante :

Boucle externe : Nombre(itérations) = n (autant de parcours que d'éléments)

Boucle interne : $\sum_{1}^{N-1} 1 = (n - 1)$

$$C(TriBulle)=N(N-1) = N^2 - N + 1 \sim O(N^2)$$

Au meilleur des cas : T est initialement trié de façon croissante :

Boucle externe : Une Itération

Boucle interne : $\sum_{1}^{N-1} 1 = (n - 1)$

$$C(TriBulle)=1*\sum_{1}^{N-1} 1 = (N-1) \sim \Omega(N)$$

Implémentation : En langage C

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

Exécution :

Affichage du temps d'exécution de l'algorithme pour chaque valeur de N (T = le temps d'exécution calculé pour chaque exécution de la fonction ***TriBulle****)*

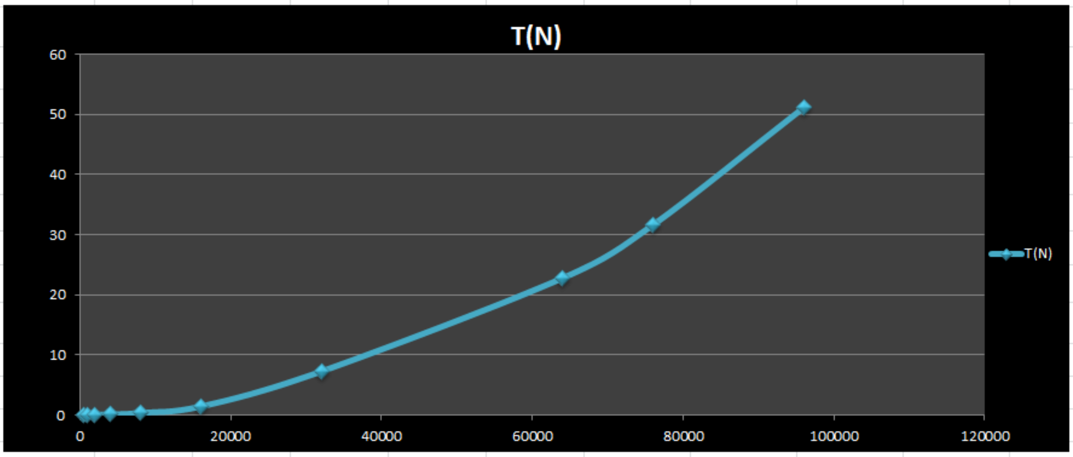
Les tableaux testés sont tous initialement triés de manière décroissante.

```
Execution de TriBulle :
N = 500.000000   T= 0.000000
N = 1000.000000          T= 0.005000
N = 2000.000000          T= 0.024000
N = 4000.000000          T= 0.082000
N = 8000.000000          T= 0.342000
N = 16000.000000         T= 1.407000
N = 32000.000000         T= 7.245000
N = 64000.000000         T= 22.769000
N = 76000.000000         T= 31.639000
N = 96000.000000         T= 51.153000
```

Représentation Graphique :

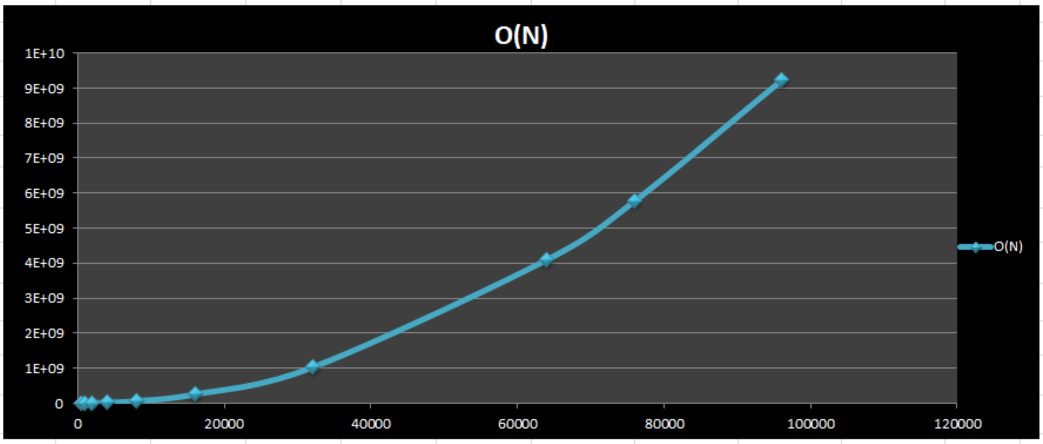
Graphe du temps d'exécution en fonction de N.

TriBulle	
N	***T(N)***
500	0
1000	0,005
2000	0,024
4000	0,082
8000	0,342
16000	1,407
32000	7,245
64000	22,769
76000	31,639
96000	51,153



Graphe de la complexité théorique en fonction de N.

TriBulle	
N	***O(N)***
500	250000
1000	1000000
2000	4000000
4000	16000000
8000	64000000
16000	2,56E+08
32000	1,02E+09
64000	4,1E+09
76000	5,78E+09
96000	9,22E+09



II. Algorithme *TriBulleOpt*:

Après le ième parcours du tableau, tous les i derniers éléments sont à leurs places définitives. Donc à chaque parcours de tableau, le parcours pourra s'arrêter un indice avant le précédent. L'algorithme devient :

Algorithme :

```
PROCEDURE TRIBULLEOPT(E/S T[N]: TABLEAU D' ENTIER ;E/
N:ENTIER)
    CHANGEMENT : BOOLEEN;
    M : ENTIER;
DEBUT
    M = N;
    CHANGEMENT = VRAI;
    TANT QUE (CHANGEMENT=VAIR) FAIRE
        CHANGEMENT = FAUX
        POUR I = 1 JUSQU'A M FAIRE
            SI (T[I]>T[I+1]) ALORS
              PERMUTER(T[I],T[I+1]);
              CHANGEMENT=VRAI;
            FIN SI;
        FIN POUR;
        M = M -1;
    FIN TANT QUE;
FIN;
```

Boucle Interne

Boucle Externe

Complexité :

Au pire des cas : T est initialement trié de façon décroissante :

$$C(\text{TriBulleOpt})=\sum_{m=1}^{N-1}\sum_{i=1}^{m} 1 = \sum_{m=1}^{N-1} m=\frac{n(n-1)}{2}\sim O(N^2)$$

Au meilleur des cas : T est initialement trié de façon croissante :

$$C(\text{TriBulleOpt})= \sum_{m=1}^{N-1}\sum_{i=1}^{m} 1 =1 * (n-1) \sim \Omega(N)$$

Implémentation : En langage C

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

Exécution :

Affichage du temps d'exécution de l'algorithme pour chaque valeur de N (T = le temps d'exécution calculé pour chaque exécution de la fonction ***TriBulleOpt****)*

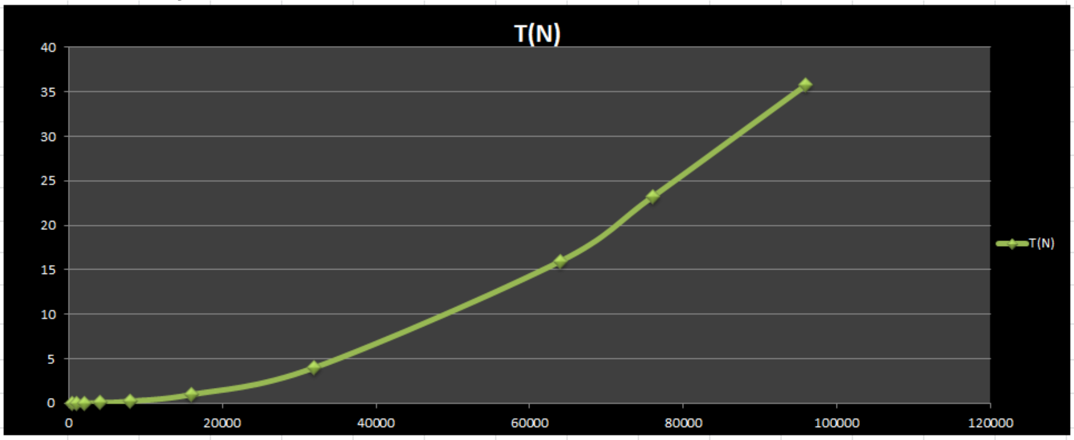
Les tableaux testés sont tous initialement triés de manière décroissante.

```
Execution de TriBulleOpt :
N = 500.000000   T= 0.002000
N = 1000.000000          T= 0.003000
N = 2000.000000          T= 0.014000
N = 4000.000000          T= 0.055000
N = 8000.000000          T= 0.235000
N = 16000.000000         T= 0.980000
N = 32000.000000         T= 4.018000
N = 64000.000000         T= 15.974000
N = 76000.000000         T= 23.216000
N = 96000.000000         T= 35.791000
```

Représentation Graphique :

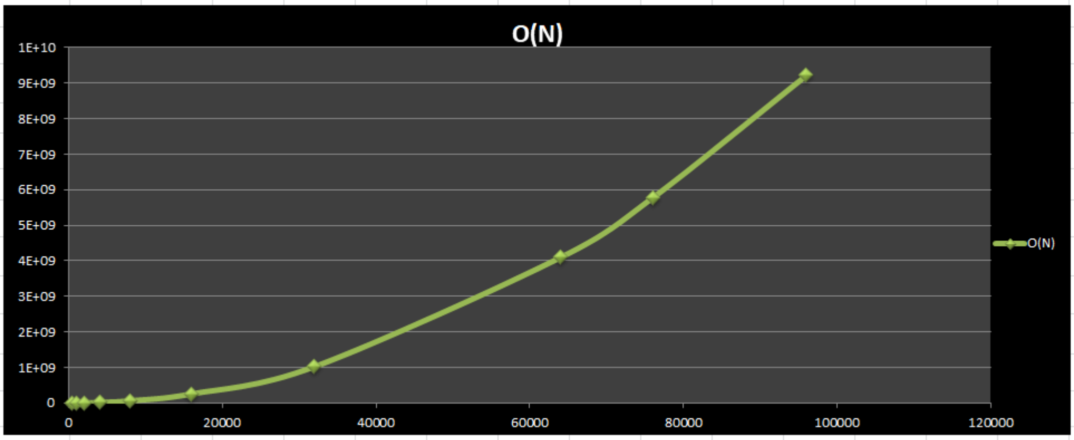
Graphe du temps d'exécution en fonction de N.

TriBulleOpt	
N	***T(N)***
500	0,002
1000	0,003
2000	0,014
4000	0,055
8000	0,235
16000	0,98
32000	4,01
64000	15,974
76000	23,216
96000	35,791



Graphe de la complexité théorique en fonction de N.

TriBulleOpt	
N	***O(N)***
500	250000
1000	1000000
2000	4000000
4000	16000000
8000	64000000
16000	256000000
32000	1,024E+09
64000	4,096E+09
76000	5,776E+09
96000	9,216E+09



III. Algorithme *Tri Gnome*:

Principe: Dans le tri gnome, on commence par le début du tableau, on compare deux éléments consécutifs (i, i+1) : s'ils sont dans l'ordre on se déplace d'un cran vers la fin du tableau (incrémente) ou on s'arrête si la fin est atteinte ; sinon, on les permute et on se déplace d'un cran vers le début du tableau (décrémente) ou si on est au début du tableau alors on se déplace d'un cran vers la fin (incrémente). (Ex4 TD4) Ecrire le programme C et donner sa complexité théorique au meilleur et pire cas.

T	pos	Condition	Action
[5, 3, 2, 4]	0	pos == 0	incrementer pos
[5, 3, 2, 4]	1	T[pos] < T [pos-1]	permuter, decrement pos
[3, 5, 2, 4]	0	pos == 0	incrementer pos
[3, 5, 2, 4]	1	T [pos] ≥ T [pos-1]	incrementer pos
[3, 5, 2, 4]	2	T [pos] < T [pos-1]	permuter, decrementer pos
[3, 2, 5, 4]	1	T [pos] < T [pos-1]	permuter, decrementer pos
[2, 3, 5, 4]	0	pos == 0	incrementer pos
[2, 3, 5, 4]	1	T [pos] ≥ T [pos-1]	incrementer pos
[2, 3, 5, 4]	2	T [pos] ≥ T [pos-1]	incrementer pos:
[2, 3, 5, 4]	3	T [pos] < T [pos-1]	permuter, decrementer pos
[2, 3, 4, 5]	2	T [pos] ≥ T [pos-1]	incrementer pos
[2, 3, 4, 5]	3	T [pos] ≥ T [pos-1]	incrementer pos
[2, 3, 4, 5]	4	pos == length(a)	terminé

Algorithme :

```
PROCDURE TRIGNOMME(E/S T[N]: TABLEAU D' ENTIER ;E/ N:ENTIER)
```

```
    I : ENTIER;
DEBUT
    I= 1;
    TANT QUE(I<N) FAIRE
        SI (T[I]>T[I+1]) ALORS
            PERMUTER(T+I,T+(I+1));
            SI (I=0) ALORS
                I = I+1
            SINON
                I=I-1;
            FIN SI
        SINON
            I = I+1;
        FIN SI;
    FIN TANT QUE;
FIN;
```

$$\sum_{i=1}^{N-1} i = \text{Nbr(Permuation)}$$

Complexité :

Au pire des cas : T trié ordre décroissant Chaque élément de position i aura i permutation à faire avant d'atteindre sa position légitime :

$$C(\text{TriGnome})=\sum_{i=1}^{N-1} i = \frac{n(n-1)}{2} \sim O(N^2)$$

Au meilleur des cas : T est initialement trié de façon croissante : il n'y a aucune permutation, et autant de comparaison et d'incrémentation que d'éléments

$$C(\text{TriGnome})=\sum_{i=0}^{N-1} 1 = (N-1) \sim \Omega(N)$$

Implémentation : En langage C

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

```
    }
}
```

Exécution :

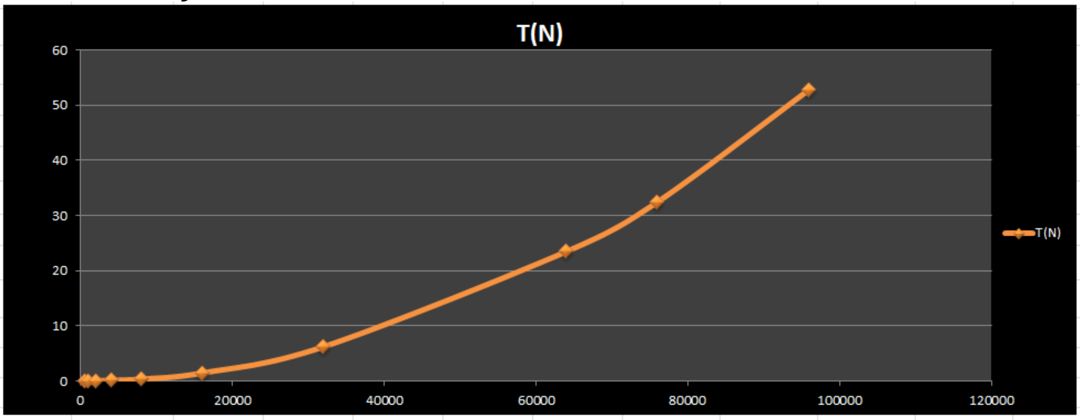
Affichage du temps d'exécution de l'algorithme pour chaque valeur de N (T = le temps d'exécution calculé pour chaque exécution de la fonction ***TriGnomme****)*
Les tableaux testés sont tous initialement triés de manière décroissante.

```
Execution de TriGnomme:
N = 500.000000   T= 0.001000
N = 1000.000000         T= 0.006000
N = 2000.000000         T= 0.021000
N = 4000.000000         T= 0.089000
N = 8000.000000         T= 0.350000
N = 16000.000000        T= 1.439000
N = 32000.000000        T= 6.193000
N = 64000.000000        T= 23.460000
N = 76000.000000        T= 32.451000
N = 96000.000000        T= 52.823000
```

Représentation Graphique :

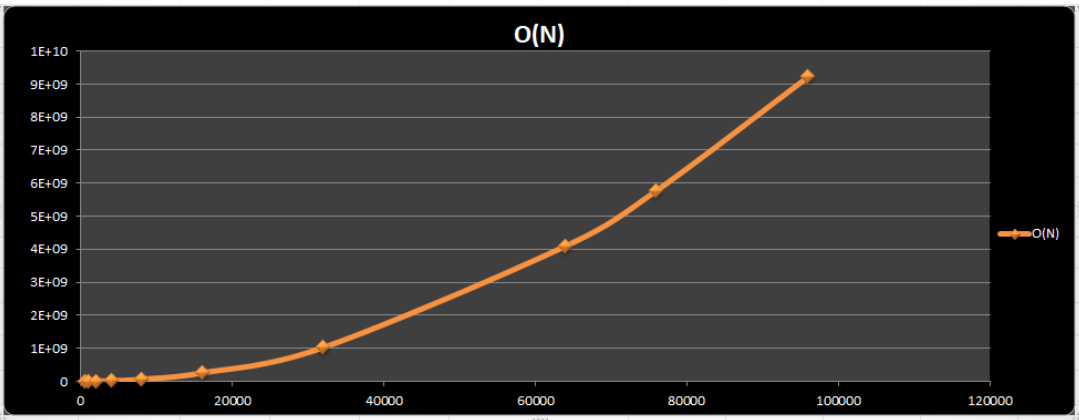
Graphe du temps d'exécution en fonction de N.

TriGnomme	
N	*T(N)*
500	0,001
1000	0,006
2000	0,021
4000	0,089
8000	0,35
16000	1,439
32000	6,193
64000	23,46
76000	32,451
96000	52,823



Graphe de la complexité théorique en fonction de N.

TriGnomme	
N	*O(N)*
500	250000
1000	1000000
2000	4000000
4000	16000000
8000	64000000
16000	256000000
32000	1,024E+09
64000	4,096E+09
76000	5,776E+09
96000	9,216E+09



IV. Algorithme *Tri par distribution*:

Principe: On utilise un tri par distribution (appelé aussi tri par base) pour trier des entiers selon leur chiffre le moins significatif (chiffre des unités), puis pour trier la liste obtenue selon le chiffre des dizaines puis selon le chiffre des centaines ...ect. La liste des entiers 141, 232, 045, 112, 143 va être triée selon le chiffre des unités, on obtient la liste 141, 232, 112, 143, 045 qui à son tour va être triée selon le chiffre des dizaines, on obtient la liste 112, 232, 141, 143, 045 puis va être triée selon le chiffre des centaines et on obtient la liste des entiers triée selon l'ordre croissant 045, 112, 141, 143, 232 .

IV.a) Algorithme *clé*:

Ecrire la fonction clé(E/ x, i : entier) : entier ; qui retourne soit le chiffre des unités, soit le chiffre des dizaines, soit le chiffre des centaines ...
Exemple : clé (143, 0)=3, clé (143, 1)=4, clé (143, 2)=1

Algorithme :

```
FONCTION CLE(E/ X,I:ENTIER): TABLEAU D' ENTIER
    DEC1,DEC2 : ENTIER;
DEBUT                                          O(i)
    DEC1 = PUISSANCE(10,I);
    DEC2 = DEC1*10;
    RETOURNER (X mod (DEC2) - X mod (DEC1))/DEC1;
FIN;
```

Complexité :

$$C(\text{clé})=C(\text{puissance}(10,i))+1+C(X \bmod 10^{i-1})= i+\frac{X}{10^i}$$

Implémentation : En langage C

```c
long cle(long X,long i) // O(i + 2*X)
{
    long dec1 =(long) pow(10,i);
    long dec2 = dec1*10;
    return (X%(dec2) - X%(dec1))/dec1;
}
```

IV.b) Algorithme *TriAux*:

Ecrire la fonction TriAux(T, n, i) qui réordonne les éléments de T tels que : clé(T[1], i) ≤ clé(T[2], i) ≤ . . . ≤ clé(T[n],i). TriAux doit s'exécuter en un temps linéaire en fonction de la taille n du tableau.

Algorithme :

```
FONCTION TRIAUX(E/ T[N]: TABLEAU D' ENTIER ;E/ N,I :ENTIER):
TABLEAU D' ENTIER
    K,I,H:ENTIER;
    TAB: TABLEAU [1..N] D' ENTIER;
DEBUT
    H=0;
    POUR K=0 JUSQU'A 9 FAIRE
        POUR J=1 JUSQU'A N FAIRE
            SI (CLE(T[J],I) = K) ALORS
                TAB[H]=T[J];
                H = H+1;
            FIN SI;
        FIN POUR;
    FIN POUR;
    RETOURNER TAB;
FIN;
```

O(i)

Boucle Interne

Boucle Externe

Complexité :

Boucle Interne

Boucle Externe

$$C(\text{TriAux}) = \sum_{k=0}^{9} \sum_{j=0}^{n-1} C\big(\text{clé}(T[i], i)\big)$$

$$= \sum_{k=0}^{9} \sum_{j=0}^{n-1} \left(i + \frac{T[i]}{10^{i}}\right)$$

$$= \sum_{k=0}^{9} \left(n.i + \frac{1}{10^{i}} \sum_{j=0}^{n-1} (T[i])\right)$$

$$= 10.n.i + \frac{1}{10^{i-1}} \sum_{j=0}^{n-1} (T[i])$$

Implémentation : En langage C

```c
long *TriAux(long *T,long n,long i)//
{
    long k,j,*Tab=(long *)malloc(n*sizeof(long)),h=0;
    for(k=0;k<=9;k++)
        for(j=0;j<n;j++)
            if(cle(T[j],i)==k)// O(i + 2*T[j]) +2
            {
                Tab[h]=T[j];
                h++;
            }
    return Tab;
}
```

IV.c) Algorithme *TriBase*:

En utilisant la procédure TriAux, écrire la fonction TriBase(T, n, k) du tri par base du tableau T.

Algorithme :

```
FONCTION TRIBASE((E/ T[N]: TABLEAU D' ENTIER ;E/ N,K :ENTIER)
: TABLEAU D' ENTIER
    I : ENTIER;
DEBUT
    POUR I=0 JUSQU'A K FAIRE
        T=TRIAUX(T,N,I);
    FIN POUR;
    RETOURNER T;
FIN;
```

$$\sum_{i=0}^{k} C\big(\mathrm{TriAux}(T, n, i)\big)$$

Complexité :

$$C(\mathrm{TriBase}) = \sum_{i=0}^{k} C\big(\mathrm{TriAux}(T, n, i)\big)$$

$$= \sum_{i=0}^{k}\Big(10.n.i + \frac{1}{10^{i-1}} \sum_{j=0}^{n-1} (T[i])\Big)$$

$$= 10.n.i + \frac{1}{10^{i-1}} \sum_{j=0}^{n-1} (T[i]) = 10.n.\frac{k(k-1)}{2} \sim (n.k^2)$$

Implémentation : En langage C

```
long *TriBase(long  *T,long n , long k)
{
    long i;
    for(i=0;i<=k;i++)
        T=TriAux(T,n,i);
    return T;
}
```

Exécution :

*Affichage du temps d'exécution de l'algorithme pour chaque valeur de N (T = le temps d'exécution calculé pour chaque exécution de la fonction **TriBase**)*

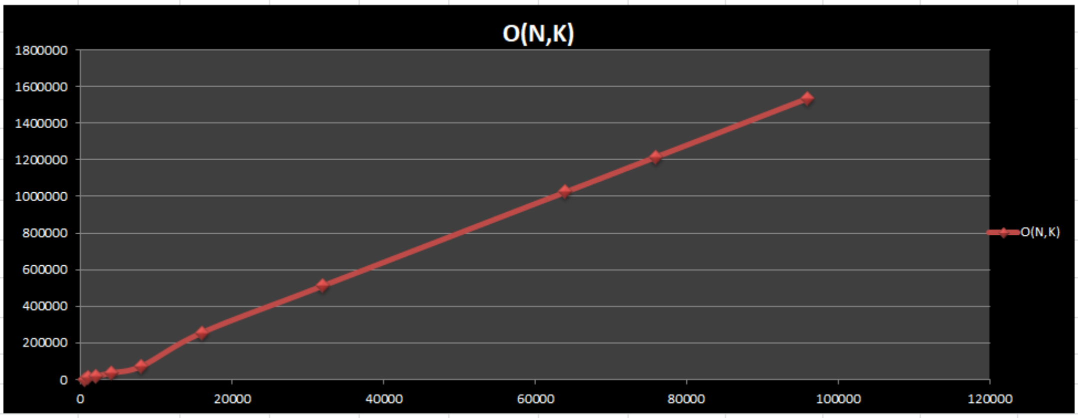
Les tableaux testés sont tous initialement triés de manière décroissante.

```
N = 500.000000   T= 0.002000
N = 1000.000000         T= 0.002000
N = 2000.000000         T= 0.004000
N = 4000.000000         T= 0.014000
N = 8000.000000         T= 0.043000
N = 16000.000000        T= 0.044000
N = 32000.000000        T= 0.058000
N = 64000.000000        T= 0.123000
N = 76000.000000        T= 0.147000
N = 96000.000000        T= 0.193000
```

Représentation Graphique :

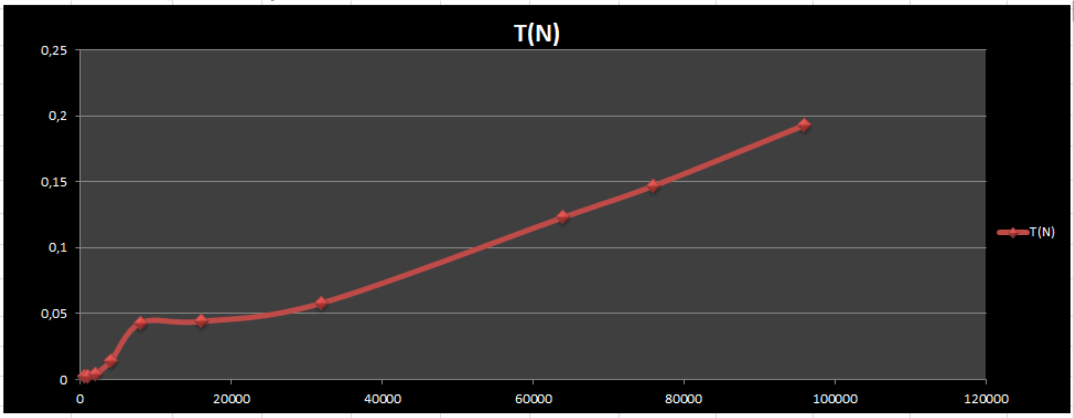
Graphe du temps d'exécution en fonction de N.

TriBase		
N	***K***	***T***
500	2	2000
1000	3	9000
2000	3	18000
4000	3	36000
8000	3	72000
16000	4	256000
32000	4	512000
64000	4	1024000
76000	4	1216000
96000	4	1536000



Graphe de la complexité théorique en fonction de N.

TriBase		
N	***K***	***T***
500	2	0,002
1000	3	0,002
2000	3	0,004
4000	3	0,014
8000	3	0,043
16000	4	0,044
32000	4	0,058
64000	4	0,123
76000	4	0,147
96000	4	0,193



V. Algorithme *Tri rapide*:

Principe: Le tri rapide est fondé sur une approche "diviser pour régner" que l'on peut décomposer en 3 étapes. On considère que nous avons un tableau Tab de taille n. On notera un "sous-tableau" de Tab, Tab[p…r], p étant l'indice du début du sous-tableau et r l'indice de la fin du sous tableau.

Diviser : le sous-tableau Tab[p…r] est partitionné (c-à-d réarrangé) en 2 sous- tableaux non vides A[p..q] et A[q+1..r] de telle sorte que chaque élément du tableau A[p..q] soit inférieur ou égal à chaque élément de A[q+1…r]. L'indice q est calculé pendant la procédure de partitionnement.

-Régner : Les 2 sous-tableaux A[p..q] et A[q+1..r] sont triés par des appels récursifs à la méthode principale de tri-rapide.

-Combiner : le tri rapide effectue le tri sur place. Cela implique qu'il n'y a aucun travail supplémentaire pour les fusionner : le sous-tableau Tab[p..r] tout entier est maintenant trié.

Algorithme : ***PARTITIONNER***

```
FONTION PARTITIONNER(E/ T : TABLEAU D' ENTIER,E/ D,F :ENTIER)
: ENTIER
    I,J,ELTPIVOT:ENTIER;
DEBUT;
        ELTPIVOT = T[D];
        I=D;  J=F;
        REPETER
            TANT QUE (T[I]<=ELTPIVOT ET I<=J)
                FAIRE I = I+1;
            FIN TANT QUE;
            TANT QUE (T[J]>ELTPIVOT ET I<=J)
                FAIRE J = J-1;
            FIN TANT QUE;
            SI (I<=J) ALORS
               PERMUTER(T+I,T+J);
            FIN SI;
        JUSQU'A (I>J);
        PERMUTER(T+D,T+J);
        RETOURNER J;
FIN;
```

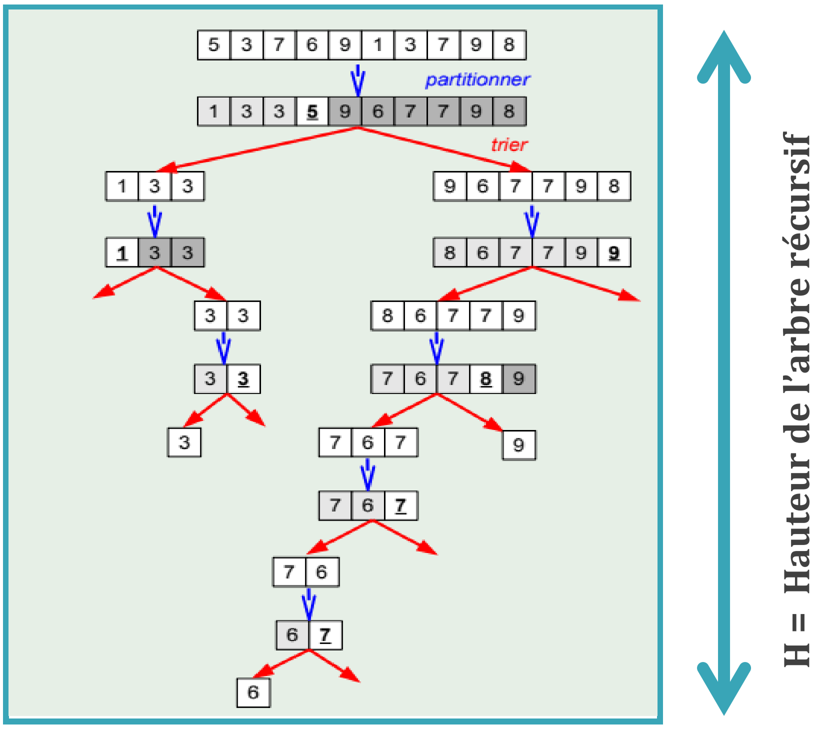
Algorithme : ***TriRapide***

```
PROCDURE TRIRAPIDE(E/S T[N]: TABLEAU D' ENTIER ;E/ P,R:ENTIER)
```

```
    Q:ENTIER;
DEBUT
     SI(P<R) ALORS
          Q=PARTITIONNER(T,P,R);
          TRIRAPIDE(T,P,Q-1);
          TRIRAPIDE(T,Q+1,R);
     FIN SI;
FIN;
```

Complexité :

Arbre Récursif :



Au pire des cas : Le partitionnement coupe le tableau en deux sous tableaux. 1 de longueur ***1*** rt un de longueur ***n-1***. Ainsi ***H= n***

C(TriRapide)= Hauteur(Arbre Récursif)*N= H*N = N*N ~ $O(N^2)$

Au meilleur des cas : Le partitionnement coupe le tableau en deux sous de même longueur. Ainsi $\boldsymbol{N}= 2^H \Rightarrow \boldsymbol{H}= \frac{\boldsymbol{log}(\boldsymbol{n})}{\boldsymbol{log}(\boldsymbol{2})}$

C(TriRapide)= Hauteur(Arbre Récursif)*N= $\frac{\boldsymbol{log}(\boldsymbol{n})}{\boldsymbol{log}(\boldsymbol{2})}$*N ~ $\Omega\ (Nlog(n))$

Implémentation : de *PARTITIONNER* En langage C

```
long partitionner(long *T,long d,long f)
{
    long i,j;
    long eltPivot = T[d];
    i=d;  j=f;
    do
    {
        while(T[i]<=eltPivot && i<=j) {i++;}
        while(T[j]>eltPivot && i<=j)  {j--;}
        if(i<=j){Permuter(T+i,T+j);}
    }
    while(i<=j);
    Permuter(T+d,T+j);
    return j;
}
```

Implémentation : de *TriRapide* En langage C

```
void TriRapide(long *T,long p, long r)
{
    long q;
    if(p<r)
    {
        q=partitionner(T,p,r);
        TriRapide(T,p,q-1);
        TriRapide(T,q+1,r);
    }
}
```

Exécution :

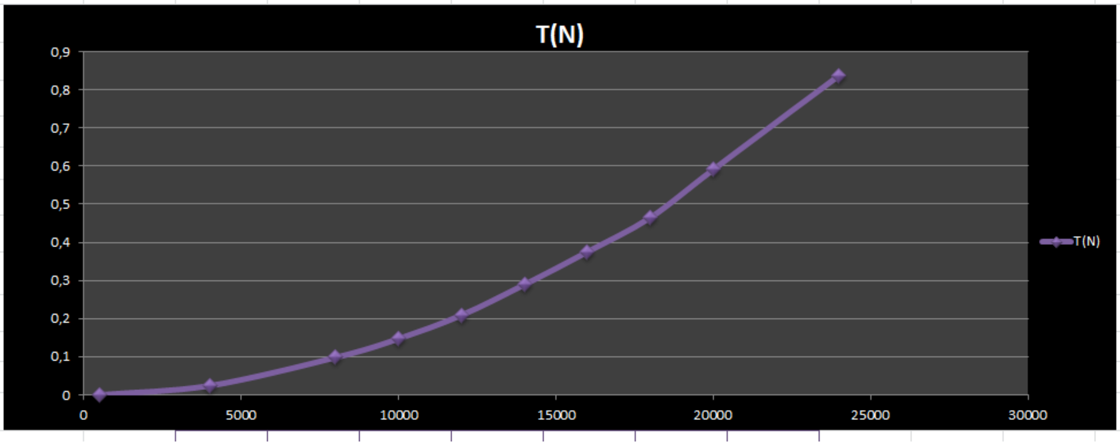
Affichage du temps d'exécution de l'algorithme pour chaque valeur de N (T = le temps d'exécution calculé pour chaque exécution de la fonction ***TriRapide****)*
Les tableaux testés sont tous initialement triés de manière décroissante.

```
Execution de TriRapide:
N = 500.000000   T= 0.000000
N = 4000.000000         T= 0.024000
N = 8000.000000         T= 0.098000
N = 10000.000000        T= 0.147000
N = 12000.000000        T= 0.209000
N = 14000.000000        T= 0.289000
N = 16000.000000        T= 0.375000
N = 18000.000000        T= 0.465000
N = 20000.000000        T= 0.591000
N = 24000.000000        T= 0.838000
```

Représentation Graphique :

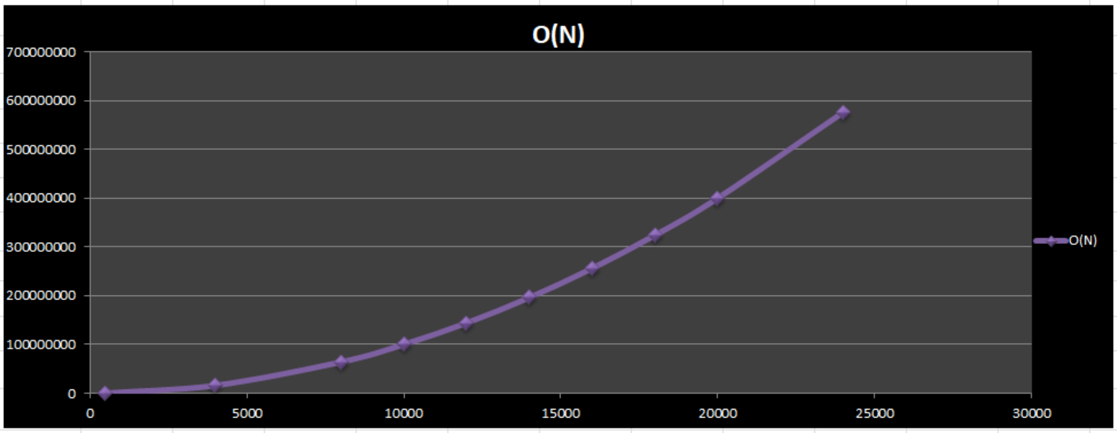
Graphe du temps d'exécution en fonction de N.

TriRapide	
N	*T(N)*
500	0
4000	0,024
8000	0,098
10000	0,147
12000	0,209
14000	0,289
16000	0,375
18000	0,465
20000	0,591
24000	0,838



Graphe de la complexité théorique en fonction de N.

TriRapide	
N	*O(N)*
500	250000
4000	16000000
8000	64000000
10000	1E+08
12000	1,44E+08
14000	1,96E+08
16000	2,56E+08
18000	3,24E+08
20000	4E+08
24000	5,76E+08



VI. Algorithme *Tri par tas*:

Principe: Le tri par tas se base sur une structure de données particulière : le tas. Il s'agit d'une représentation d'un arbre binaire sous forme de tableau. L'arbre est presque complet : il est rempli à tous les niveaux, sauf potentiellement le dernier. Le parcours de l'arbre se fait par un calcul d'indice sur le tableau. Pour un nœud d'indice i, on a le père à l'indice $[i/2]$, le fils gauche à l'indice 2i et le fils droit à l'indice 2i+1.

On va appliquer les procédures d'insertion et de suppression de la racine d'un tas à l'exemple de façon à, dans un premier temps, construire un tas puis en supprimant le minimum atteindre la solution où tous les éléments sont dans l'ordre croissant. Nous allons d'abord construire le tas en insérant dans ce tas les

éléments de l'exemple les uns après les autres. Voici la représentation des différentes étapes pour la liste 16-10-8-11-5-6-9-1. A chaque étape, l'élément ajouté est indiqué en bleu.

Algorithme : *Max*

```
FONCTION MAX(E/ A : TABLEAU D' ENTIER,E/ N,I,J,K :ENTIER):
ENTIER
    M:ENTIER;
DEBUT
    M = I;
    SI (J < N ET A[J] > A[M]) ALORS
        M = J;
    FIN SI;
    SI (K < N ET A[K] > A[M]) ALORS
        M = K;
    FIN SI;
    RETOURNER M;
FIN;
```

Algorithme : *Downheap*

```
PROCDURE DOWNHEAP (E/S A : TABLEAU D' ENTIER,E/ N,I :ENTIER)
    B:BOOLEEN;
    J,T:ENTIER;
DEBUT
    B = VRAI;
    TANT QUE (B=VRAI) FAIRE
        J = MAX(A, N, I, 2 * I + 1, 2 * I + 2);
        SI (J == I) ALORS
            B = FAUX;
        SINON
            T = A[I];
```

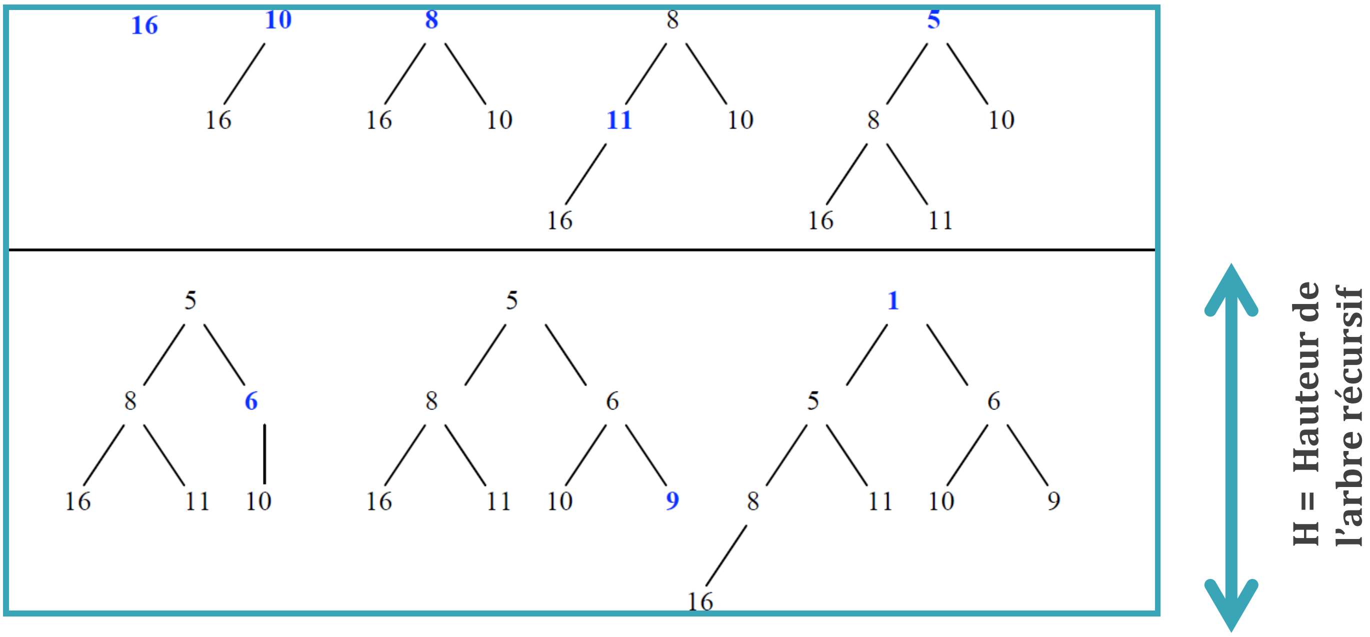
```
            A[I] = A[J];
            A[J] = T;
            I = J;
         FIN SI;
    FIN TANT QUE;
FIN;
```

Algorithme : ***Heapsort***

```
PROCDURE HEAPSORT ( (E/S A : TABLEAU D' ENTIER,E/ N :ENTIER)
    I,T:ENTIER;
DEBUT
    POUR I= (N - 2) /2 JUSQU'A I=0 PAS -1 FAIRE
        DOWNHEAP(A, N, I);
    FIN POUR;
    POUR I = 1  JUSQU'A N FAIRE
        T = A[N - I - 1];
        A[N - I - 1] = A[0];
        A[0] = T;
        DOWNHEAP(A, N - I - 1, 0);
    FIN POUR;
FIN;
```

Complexité :

Exemple : [16-10-8-11-5-6-9-1] Construisons l'**Arbre récursif :**



Étant un arbre binaire Alors : $N = \mathbf{2}^{\boldsymbol{H}} \Leftrightarrow H = \dfrac{\boldsymbol{log(n)}}{\boldsymbol{log(2)}}$

Downheap : construction du tas (tamisage) :

$$\text{C(downheap)} = \text{Hauteur(Arbre Récursif)} = H = \frac{\boldsymbol{log(n)}}{\boldsymbol{log(2)}}$$

Heapsort : contient une boucle qui exécute ***downheap*** à chaque itération ainsi :

$$\text{C(heapsort)} = \sum_{i=0}^{n} \text{C(downheap)} = \sum_{i=0}^{n-1} \frac{\boldsymbol{log(n)}}{\boldsymbol{log(2)}} = \frac{\boldsymbol{log(n)}}{\boldsymbol{log(2)}} \sum_{i=0}^{n-1} \mathbf{1}$$

$$= \frac{\boldsymbol{log(n)}}{\boldsymbol{log(2)}} (n - 1 - 0 + 1) \sim \theta(\text{n.log(n)})$$

Implémentation : de *Max* En langage C

```
long max (long *a, long n, long i, long j, long k) {
    long m = i;
    if (j < n && a[j] > a[m]) {
        m = j;
    }
    if (k < n && a[k] > a[m]) {
        m = k;
    }
    return m;
}
```

Implémentation : de *Downheap* En langage C

```
void downheap (long *a, long n, long i) {
    while (1) {
        long j = max(a, n, i, 2 * i + 1, 2 * i + 2);
        if (j == i) {
            break;
        }
        long t = a[i];
        a[i] = a[j];
        a[j] = t;
        i = j;
    }
}
```

Implémentation : de *Heapsort* En langage C

```
void heapsort (long *a, long n) {
    long i;
    for (i = (n - 2) / 2; i >= 0; i--) {
        downheap(a, n, i);
    }
    for (i = 0; i < n; i++) {
        long t = a[n - i - 1];
        a[n - i - 1] = a[0];
        a[0] = t;
        downheap(a, n - i - 1, 0);
    }
```

```
}
```

Exécution :

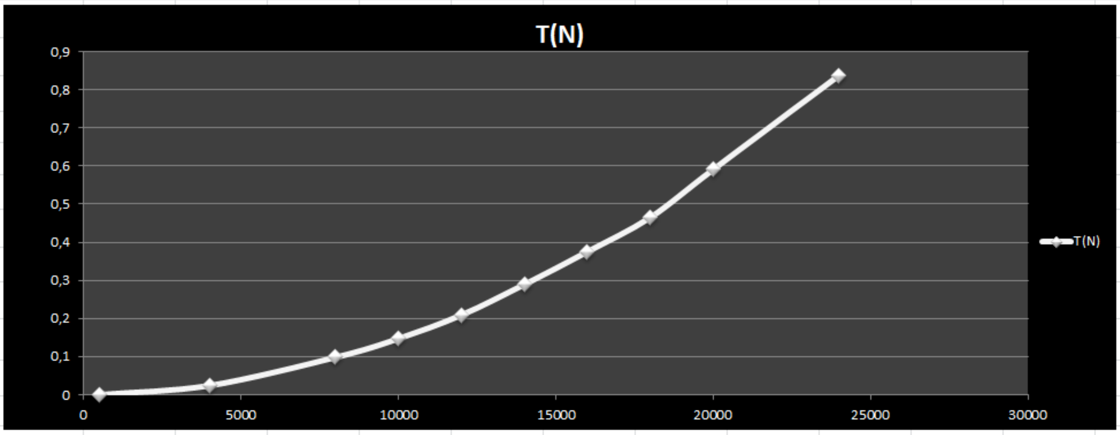
Affichage du temps d'exécution de l'algorithme pour chaque valeur de N (T = le temps d'exécution calculé pour chaque exécution de la fonction ***heapsort****)*
Les tableaux testés sont tous initialement triés de manière décroissante.

```
Execution de heapsort:
N = 500.000000   T= 0.000000
N = 4000.000000          T= 0.001000
N = 8000.000000          T= 0.001000
N = 10000.000000         T= 0.002000
N = 12000.000000         T= 0.003000
N = 14000.000000         T= 0.003000
N = 16000.000000         T= 0.003000
N = 18000.000000         T= 0.004000
N = 20000.000000         T= 0.005000
N = 24000.000000         T= 0.005000
```

Représentation Graphique :

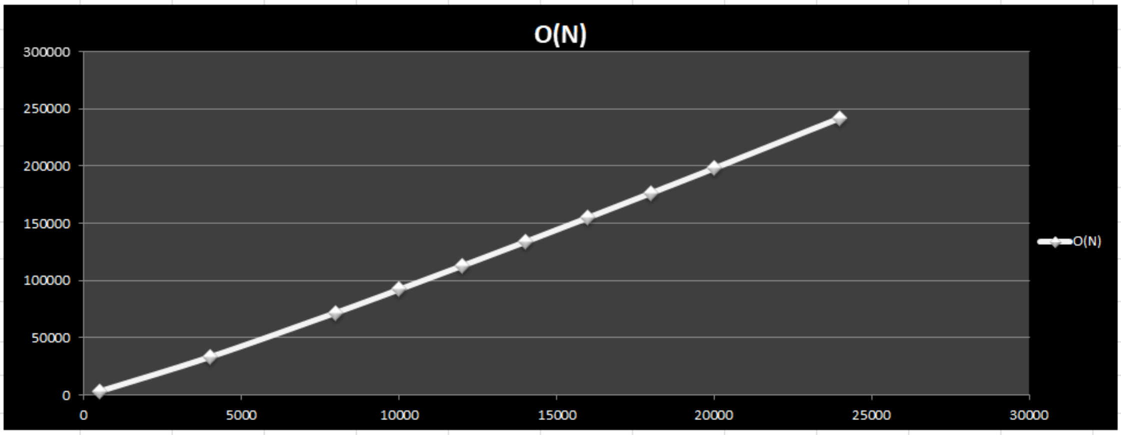
Graphe du temps d'exécution en fonction de N.

HeapSort	
N	***T(N)***
500	0
4000	0,024
8000	0,098
10000	0,147
12000	0,209
14000	0,289
16000	0,375
18000	0,465
20000	0,591
24000	0,838



Graphe de la complexité théorique en fonction de N.

HeapSort	
N	***O(N)***
500	3107,304
4000	33176,2
8000	71897,57
10000	92103,4
12000	112711,9
14000	133655,4
16000	154885,5
18000	176366,3
20000	198069,8
24000	242059,4



(*)Code Source du Programme complet :

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

void Permuter(long *x , long *y)
{
    long temp = *x;
    *x=*y;
    *y=temp;
}

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

//TriBulleOpt
void TriBulleOpt(long *T , long n)
{
    long m = n-1;
    long changement = 1;
    while(changement)
    {
        changement = 0;
        long i=0;
        for(i=0;i<m;i++)
```

```
        {
            if(T[i]>T[i+1])
            {
                Permuter(T+i,T+(i+1));
                changement = 1;
            }
        }
        m=m-1;
    }
}

//TriGnomme
void TriGnomme(long *T,long n)
{
    long i= 0;
    while(i<n-1)
    {
        if(T[i]>T[i+1])
        {
            Permuter(T+i,T+(i+1));
            if(i==0) i++;
            else i--;
        }
        else
            i++;
    }
}

//TriDistribution
//cl�
long cle(long X,long i) // O(i + 2*X)
{
    long dec1 =(long) pow(10,i);
    long dec2 = dec1*10;
    return (X%(dec2) - X%(dec1))/dec1;
}
//TriAux
long *TriAux(long *T,long n,long i)//
{
    long k,j,*Tab=(long *)malloc(n*sizeof(long)),h=0;
    for(k=0;k<=9;k++)
        for(j=0;j<n;j++)
            if(cle(T[j],i)==k)// O(i + 2*T[j]) +2
            {
```

```
                Tab[h]=T[j];
                h++;
            }
    return Tab;
}
//TriBase
long *TriBase(long  *T,long n , long k)
{
    long i;
    for(i=0;i<=k;i++)
        T=TriAux(T,n,i);
    return T;
}

//Quick Sort
//partitionner
long partitionner(long *T,long d,long f)
{
    long i,j;
    long eltPivot = T[d];
    i=d;  j=f;
    do
    {
        while(T[i]<=eltPivot && i<=j) {i++;}
        while(T[j]>eltPivot && i<=j)  {j--;}
        if(i<=j){Permuter(T+i,T+j);}
    }
    while(i<=j);
    Permuter(T+d,T+j);
    return j;
}
//TriRapide
void TriRapide(long *T,long p, long r)
{
    long q;
    if(p<r)
    {
        q=partitionner(T,p,r);
        TriRapide(T,p,q-1);
        TriRapide(T,q+1,r);
    }
}
```

```c
//TriparTas
long max (long *a, long n, long i, long j, long k) {
    long m = i;
    if (j < n && a[j] > a[m]) {
        m = j;
    }
    if (k < n && a[k] > a[m]) {
        m = k;
    }
    return m;
}

void downheap (long *a, long n, long i) {
    while (1) {
        long j = max(a, n, i, 2 * i + 1, 2 * i + 2);
        if (j == i) {
            break;
        }
        long t = a[i];
        a[i] = a[j];
        a[j] = t;
        i = j;
    }
}

void heapsort (long *a, long n) {
    long i;
    for (i = (n - 2) / 2; i >= 0; i--) {
        downheap(a, n, i);
    }
    for (i = 0; i < n; i++) {
        long t = a[n - i - 1];
        a[n - i - 1] = a[0];
        a[0] = t;
        downheap(a, n - i - 1, 0);
    }
}

void Affiche(long T[],long N)
{
    long i=0;
    printf("[");
    for(i=0;i<N-1;i++)
        printf("%d,",T[i]);
```

```
    printf("%d]\n",T[N-1]);
}

long Max(long *T,long n)
{
    long i,max=T[0];
    for(i=1;i<n;i++)
        if(T[i]>max) max=T[i];
    return max;
}


//Tableau Tri� Ordre d�croissant
long *PireCas(long n)
{
    long i,*T=(long *)malloc(n*sizeof(long));
    for(i=0;i<n;i++) T[i]=n-i;
    return T;
}
//Tableau Tri� Ordre Croissant
long *MeilleurCas(long n)
{
    long i,*T=(long *)malloc(n*sizeof(long));
    for(i=0;i<n;i++) T[i]=i;
    return T;
}


double **Calcul_des_Temps(double **tab , long algorithme)
{
        long j,position,min,max;
        for(j=0 ; j<10 ; j++)
        {
            long k;
            long *TAB = PireCas(tab[0][j]);
            if(algorithme==4){
            k = (long)log10((double)Max(TAB,tab[0][j]));
            printf("k = %d \n",k);}
            clock_t begin = clock();
            switch(algorithme)
            {
                case 1: TriBulle(TAB,tab[0][j]); break;
                case 2: TriBulleOpt(TAB,tab[0][j]); break;
                case 3: TriGnomme(TAB,tab[0][j]); break;
```

```
                case 4: TAB = TriBase(TAB,tab[0][j],k); break;
                case 5: TriRapide(TAB,0,tab[0][j]-1); break;
                case 6: heapsort(TAB,tab[0][j]); break;
            }
            clock_t end = clock();
            tab[1][j] = (double)(end - begin) / CLOCKS_PER_SEC;
        }
        return tab;
}

double **Tableau_de_Valeurs(void)
{
    long i ;
    double **tab;
    tab = (double **)malloc(4*sizeof(double *));
    for(i=0 ; i<4 ; i++) tab[i] = (double
*)malloc(10*sizeof(double));
    tab[0][0]=5*100;
    tab[0][1]=4000;
    tab[0][2]=8000;
    tab[0][3]=10000;
    tab[0][4]=18000;
    tab[0][5]=25000;
    tab[0][6]=36000;
    tab[0][7]=48000;
    tab[0][8]=56000;
    tab[0][9]=64000;
    for(i=0 ; i<10 ; i++)tab[1][i] = 0 ;
    return tab;
}

void Afficher_Tableau_de_Valeurs(double **tab)
{
    long j;
        for(j=0 ; j<10 ; j++)
        {
            printf("N = %f \t T= %f \t
\n",tab[0][j],tab[1][j]);
        }
}


long main(long argc, char *argv[])
{
```

```
  /*    printf("Execution de TriBulle :\n");
    Afficher_Tableau_de_Valeurs(Calcul_des_Temps(Tableau_de_Valeurs
(),1));
    printf("Execution de TriBulleOpt :\n");
    Afficher_Tableau_de_Valeurs(Calcul_des_Temps(Tableau_de_Valeurs
(),2));
    printf("Execution de TriGnomme:\n");
    Afficher_Tableau_de_Valeurs(Calcul_des_Temps(Tableau_de_Valeurs
(),3));
*/  printf("Execution de TriBase:\n");
    Afficher_Tableau_de_Valeurs(Calcul_des_Temps(Tableau_de_Valeurs
(),4));
/*  printf("Execution de TriRapide:\n");
    Afficher_Tableau_de_Valeurs(Calcul_des_Temps(Tableau_de_Valeurs
(),5));
    printf("Execution de heapsort:\n");
    Afficher_Tableau_de_Valeurs(Calcul_des_Temps(Tableau_de_Valeurs
(),6));
*/
/*
    long i;
long *T = PireCas(10);

printf("TriBulle : \n");
Affiche(T,10);
TriBulle(T,10);
printf("=> Traitement ...");
getchar();
Affiche(T,10);
printf("\n-----------------------------------------------------
\n");


T = PireCas(10);
getchar();
printf("TriBulleOpt : \n");
Affiche(T,10);
TriBulleOpt(T,10);
printf("=> Traitement ...");
getchar();
Affiche(T,10);
```

```
printf("\n--------------------------------------------------------
\n");


T = PireCas(10);
getchar();
printf("TriGnome : \n");
Affiche(T,10);
TriGnomme(T,10);
printf("=> Traitement ...");
getchar();
Affiche(T,10);
printf("\n--------------------------------------------------------
\n");



T = PireCas(100);
getchar();
printf("TriBase : \n");
Affiche(T,100);
T=TriBase(T,100,2   );
printf("=> Traitement ...");
getchar();
Affiche(T,100);
printf("\n--------------------------------------------------------
\n");


T = PireCas(10);
getchar();
printf("QuickSort : \n");
Affiche(T,10);
TriRapide(T,0,9);
printf("=> Traitement ...");
getchar();
Affiche(T,10);
printf("\n--------------------------------------------------------
\n");

T = PireCas(10);
getchar();
printf("Tri par Tas : \n");
Affiche(T,10);
heapsort(T,10);
```

```
printf("=> Traitement ...");
getchar();
Affiche(T,10);

//Affiche(T,50000);
//TriRapide(T,0,39999);
//printf("\n\n\n--------------------------------------------------------
-------------\n\n\n");
//Affiche(T,50000);
/*  for(i=5 ; i<6 ; i++)
    {
        long *T = PireCas(50000);
        printf("Temps d'Execution de l'Algo %d = %lf
.\n",i,Temps_Execution(T,50000,i));

    }
*/

//  printf("cle(143,2) = %d",cle(143,1));
    /*  printf("Execution de l'Algorithme 1 :\n");
        Afficher_Tableau_de_Valeurs(Calcul_des_Temps(Tableau_de_Val
eurs(),1));*/

    return 0;
}
```