|  |  |
| --- | --- |
|  | **COMPLEXITÉ** |
|  | **Master 1 IL**  **Groupe 2**  **2018** |

|  |
| --- |
| **[ Algorithmique avancÉe et complexitÉ ]** |
| E-mail : geronimotoutcourt@gmail.com |

Rapport de TP Mini Projet COMPLEXITÉ : Les TRI

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



*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**

**Boucle Interne**

**SI (T[I]>T[I+1]) ALORS**

**Boucle Externe**

**PERMUTER(T[I],T[I+1]);**

**CHANGEMENT=VRAI;**

**FIN SI;**

**FIN POUR;**

**FIN TANT QUE;**

**FIN;**

# 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 :

C(TriBulle)= = N²- N +1 ∼ O(N²)

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

Boucle externe : Une Itération

Boucle interne :

C(TriBulle)=1\* ∼ 𝛺(N)

# Implémentation : En langage 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.*

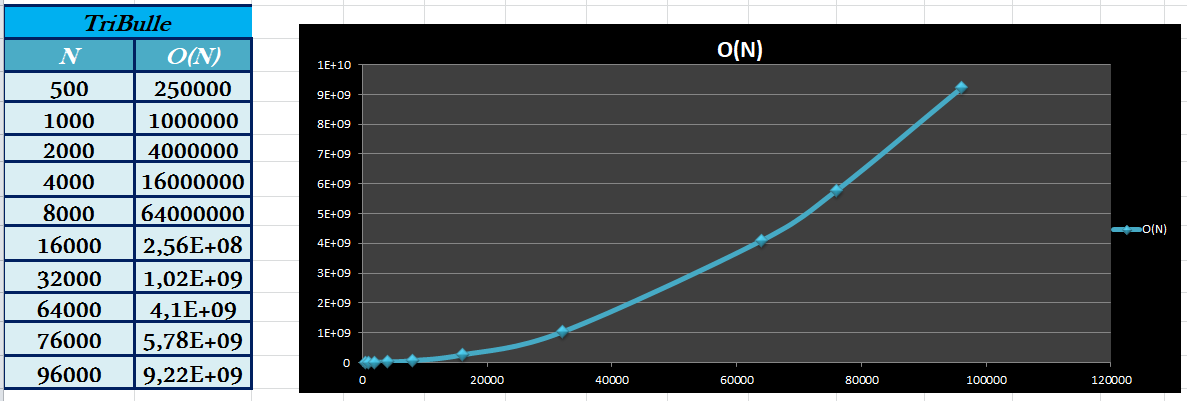
# *C:\Users\Moflawer\Desktop\Dol_Gul_Dur\WorkShop_Tree\C\Almost_Done\TP_COMPLEXITÉ\TP Mini Projet\bulle.PNG*

# Représentation Graphique :

# *Graphe du temps d’exécution en fonction de N.*

# *C:\Users\Moflawer\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tn1.png*

# *Graphe de la complexité théorique en fonction de N.*



# 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**

**Boucle Externe**

**SI (T[I]>T[I+1]) ALORS**

**Boucle Interne**

**PERMUTER(T[I],T[I+1]);**

**CHANGEMENT=VRAI;**

**FIN SI;**

**FIN POUR;**

**M = M -1;**

**FIN TANT QUE;**

**FIN;**

# Complexité :

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

C(TriBulleOpt)= = =∼ O(N²)

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

C(TriBulleOpt)= = ∼ 𝛺(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 :

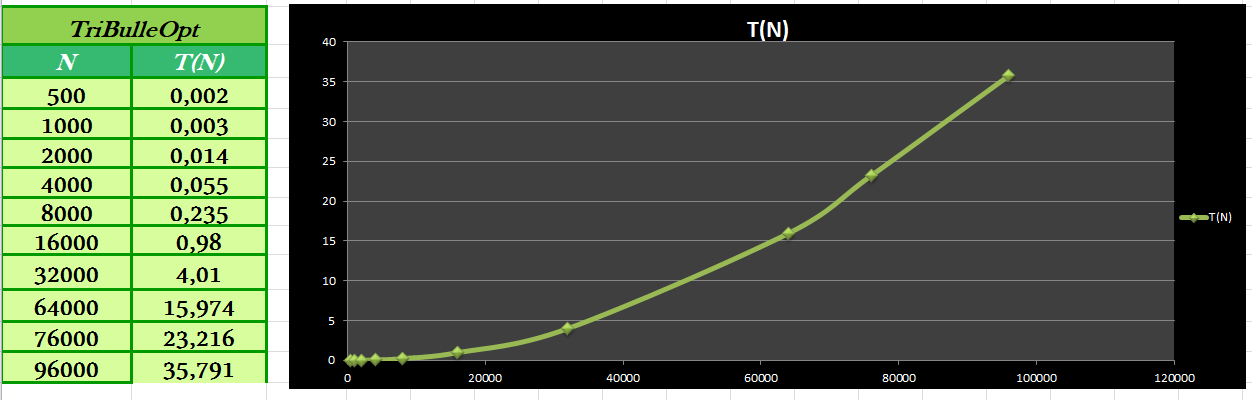
# *Affichage du temps d’exécution de l’algorithme pour chaque valeur de N (T = le temps d’exécution calculé pour chaque exécution de la fonction TriBulleOpt)*

# *Les tableaux testés sont tous initialement triés de manière décroissante.*

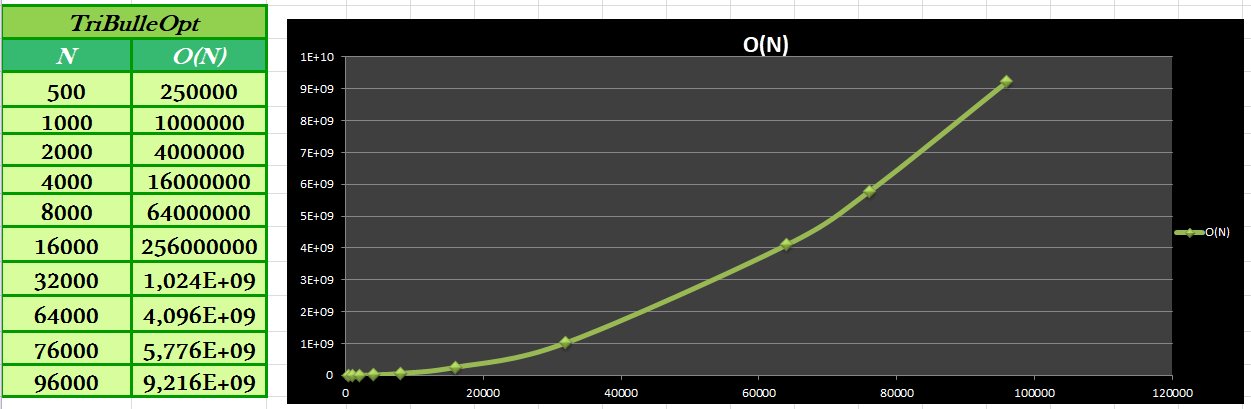
# *C:\Users\Moflawer\Desktop\Dol_Gul_Dur\WorkShop_Tree\C\Almost_Done\TP_COMPLEXITÉ\TP Mini Projet\bulleOpt.PNG*

# Représentation Graphique :

# *Graphe du temps d’exécution en fonction de N.*



# *Graphe de la complexité théorique en fonction de N.*



# 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**

**= Nbr(Permuation)**

**SINON**

**I=I-1;**

**FIN SI**

**SINON**

**I = I+1;**

**FIN SI;**

**FIN TANT QUE;**

**FIN;**

# 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(TriGnome)= = ∼ O(N²)

**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(TriGnome)= ∼ 𝛺(N)

# Implémentation : En langage 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++;**

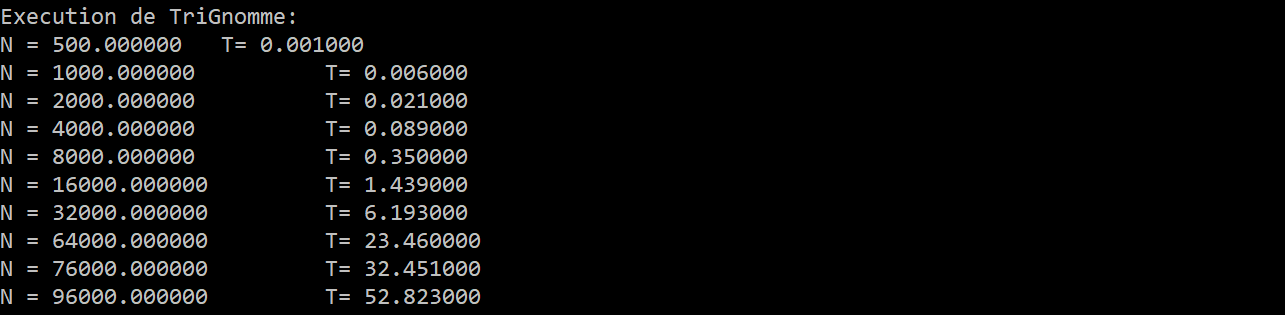
**}**

**}**

# 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)*

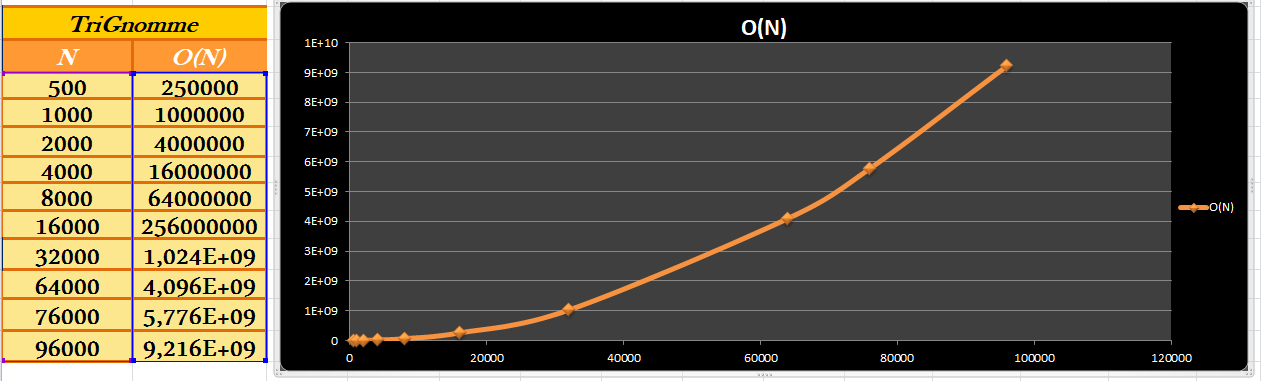
# *Les tableaux testés sont tous initialement triés de manière décroissante.*



# Représentation Graphique :

# *Graphe du temps d’exécution en fonction de N.* C:\Users\Moflawer\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tn3.png

# *Graphe de la complexité théorique en fonction de N.*



# 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;**

***O(i)***

**DEBUT**

**DEC1 = PUISSANCE(10,I);**

**DEC2 = DEC1\*10;**

**RETOURNER (X mod (DEC2) – X mod (DEC1))/DEC1;**

**FIN;**

# Complexité :

C(clé)=+1+C(X mod )= i +

# Implémentation : En langage 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**

**Boucle Externe**

**Boucle Interne**

**SI (CLE(T[J],I) = K) ALORS**

**TAB[H]=T[J];**

***O(i)***

**H = H+1;**

**FIN SI;**

**FIN POUR;**

**FIN POUR;**

**RETOURNER TAB;**

**FIN;**

# Complexité :

**Boucle Externe**

**Boucle Interne**

C(TriAux)=

=

=

=

# Implémentation : En langage 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;**

# Complexité :

C(TriBase)=

=

= = 10.n. ∼ (n.k²)

# 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 :

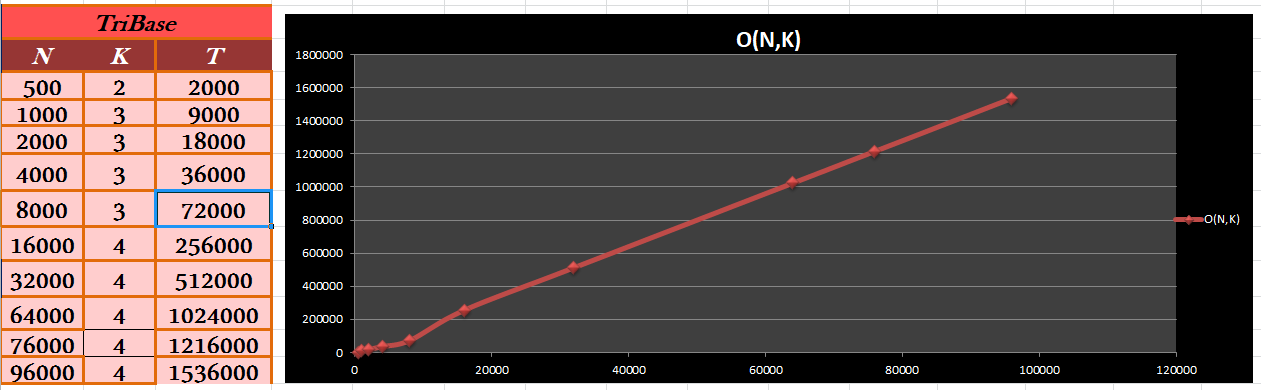
# *Affichage du temps d’exécution de l’algorithme pour chaque valeur de N (T = le temps d’exécution calculé pour chaque exécution de la fonction TriBase)*

# *Les tableaux testés sont tous initialement triés de manière décroissante.*

# C:\Users\Moflawer\Desktop\Dol_Gul_Dur\WorkShop_Tree\C\Almost_Done\TP_COMPLEXITÉ\TP Mini Projet\base.PNG

# Représentation Graphique :

# *Graphe du temps d’exécution en fonction de N.*



# *Graphe de la complexité théorique en fonction de N.*

# *C:\Users\Moflawer\AppData\Local\Microsoft\Windows\INetCache\Content.Word\yn4.png*

# 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);**

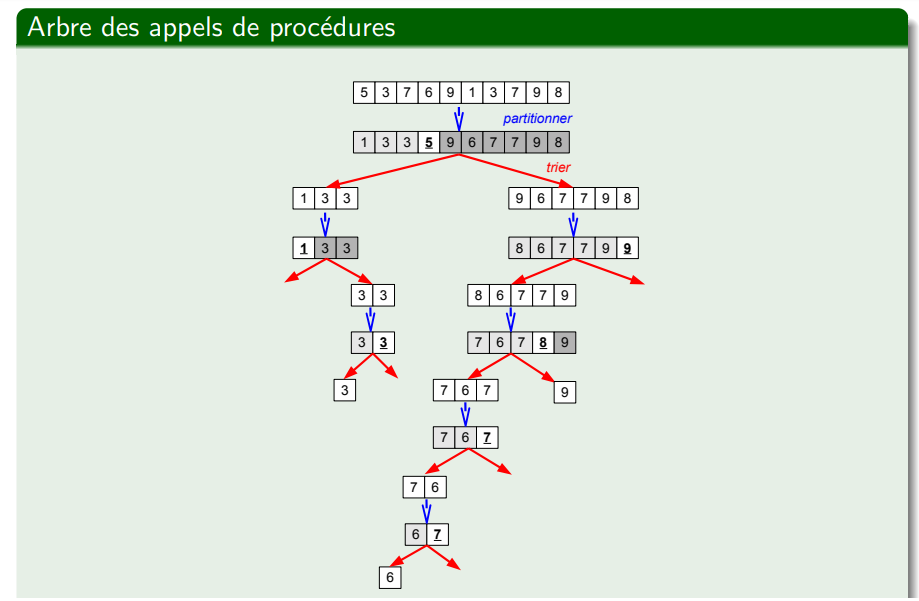
**TRIRAPIDE(T,Q+1,R);**

**FIN SI;**

**FIN;**

# Complexité :

# Arbre Récursif :



**H = Hauteur de l’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(N²)

**Au meilleur des cas :** Le partitionnement coupe le tableau en deux sous de même longueur. Ainsi ***N=*** ⤇ ***H=***

C(TriRapide)= Hauteur(Arbre Récursif)\*N= \*N ∼ 𝛺 (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);**

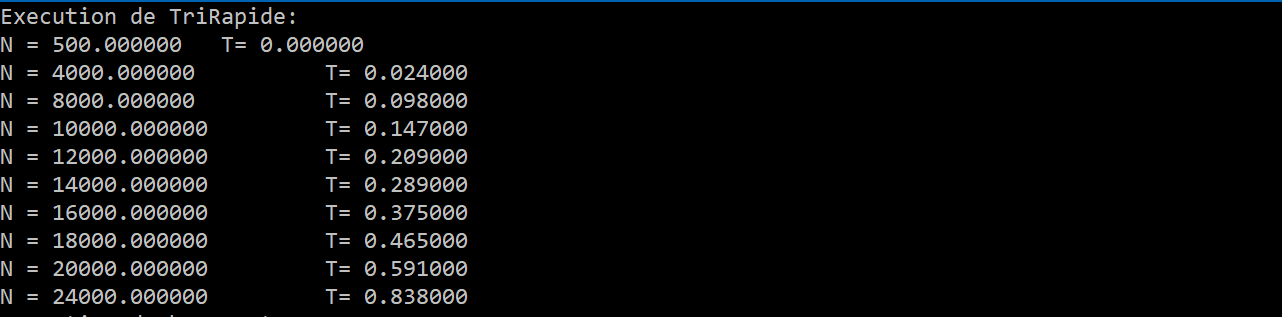
**}**

**}**

# 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.*

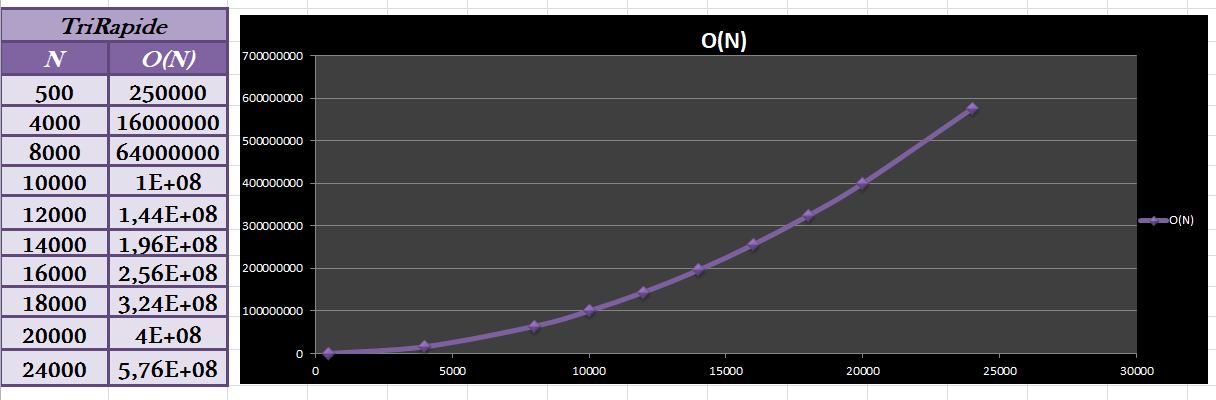


# Représentation Graphique :

# *Graphe du temps d’exécution en fonction de N.*

# C:\Users\Moflawer\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tn5.png

# *Graphe de la complexité théorique en fonction de N.*



# 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 :

# 

**H = Hauteur de l’arbre récursif**

# Étant un arbre binaire Alors : *N=* ⟺ *H=*

# Downheap : construction du tas (tamisage) :

C(downheap)= =

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

C(heapsort)= = = = ∼ 𝜃(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);**

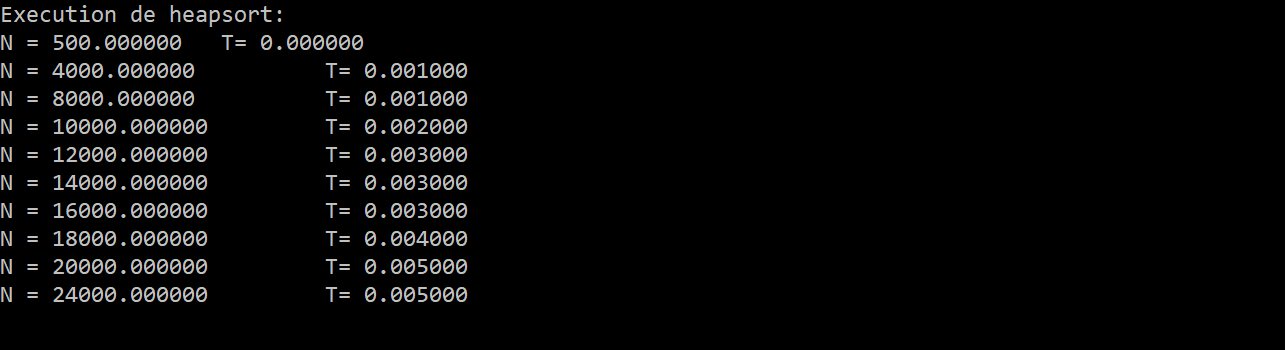
**}**

**}**

# 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.*

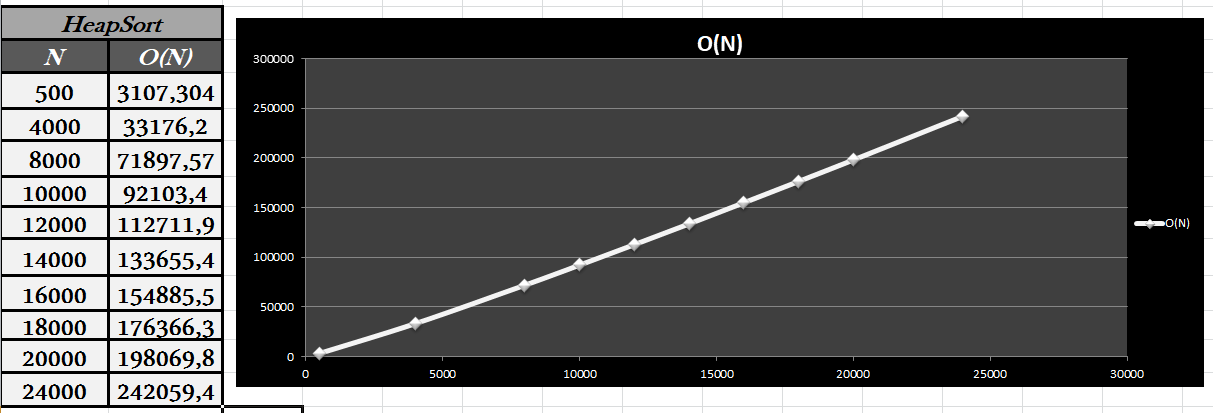
****

# Représentation Graphique :

# *Graphe du temps d’exécution en fonction de N.*

# C:\Users\Moflawer\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tn6.png

# *Graphe de la complexité théorique en fonction de N.*

****

(\*)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);**

**}**

**}**

**//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\_Valeurs(),1));\*/**

**return 0;**

**}**