# Partie 1

## 1 – tri par insertion

### Description :

L’algorithme consiste en effet à diviser la liste en deux parties : une partie triée (qui commence par le premier élément) et une partie non triée (qui contient les éléments restants). Il prend ensuite un élément de la partie non triée, le compare aux éléments de la partie triée, et l'insère à la bonne position de manière à ce que la partie triée reste triée. Ce processus est répété jusqu'à ce que la liste entière soit triée.

### Algorithme

Procédure : Tri par insertion

Entrée : T : tableau d’entiers ;

n : taille du tableau ;

Var : i, j, courant: entier ;

Début :

Pour i := 1 a n faire :

courant := T[i] ;

j := i -1 ;

tantque (j <= 0 et T[j] > courant) faire :

T[j+1] := T[j] ;

j-- ;

fait ;

T[j+1] := courant ;

Fait ;

Fin ;

Procédure : triParInsertionRecursive

Entrée : T : tableau d’entiers ;

n : entier ;

j : entier ;

Var : courant : entier ;

Début :

si T[ j ] < T[ j+1 ] alors :

courant := t[j] ;

T[j] := T[j+1] ;

T[j+1] := courant ;

si j > 0 alors : triParInsertionRecursive(T,n,J-1) ;

sinon : triParInsertionRecursive(T,n,J+1)

sinon :

si j < n alors :

triParInsertionRecursive(T, n, j+1) ;

fsi ;

fsi ;

Fin ;

### Calcule de la complexité temporelle :

#### Meilleur cas :

Dans le cas où le tableau est déjà trié, la boucle interne qui compare et déplace les éléments n'a pas besoin de faire de déplacement, car tous les éléments sont déjà dans l'ordre. La complexité temporelle du meilleur cas est donc O(n). L’algorithme effectue simplement une comparaison pour chaque élément du tableau.

#### Pure cas :

Dans le cas où le tableau est trié dans l’ordre inverse, la boucle interne s’exécute i fois pour chaque element. La première valeur nécessite 0 déplacements (car il n'y a pas d'élément à gauche), la deuxième nécessite 1 déplacement, la troisième nécessite 2 déplacements, et ainsi de suite.

Cela signifie que le nombre total de déplacements nécessaires dans le pire cas est la somme des premiers n-1 entiers, ce qui est donné par la formule :

Nombre total de déplacements = 1 + 2 + 3 + ... + (n-1) = (n-1) \* n / 2

Cette formule représente le nombre total de déplacements nécessaires dans la boucle interne de la boucle externe. Cela montre que la complexité temporelle dans le pire cas du tri par insertion est quadratique, soit O(n\*\*2)en raison de ces déplacements

### Calcule de la complexité spatiale :

La complexité spatiale dépend souvent de la manière dont l'algorithme manipule la mémoire, mais pour le tri par insertion, le tri se fait généralement sur place, ce qui signifie que l'algorithme n'a besoin que d'une mémoire constante, indépendamment de la taille du tableau. Cela s'explique par le fait que le tri par insertion effectue les déplacements d'éléments dans le tableau lui-même, sans avoir besoin d'une structure de données externe.

Ainsi, la complexité spatiale du tri par insertion est généralement considérée comme O(1), ce qui signifie qu'elle nécessite une quantité constante de mémoire supplémentaire, indépendamment de la taille du tableau.

### Le programme en C :

void triParInsertion(int \*T, int n) {

int i, j, courant;

for (i = 1; i < n; i++) {

courant = T[i];

j = i - 1;

// Déplacer les éléments plus grands que courant vers la droite

while (j >= 0 && T[j] > courant) {

T[j + 1] = T[j];

j--;

}

// Insérer courant à la bonne position

T[j + 1] = courant;

}

}

### Calcule du temps d’exécution pour différents échantillons :

#### triée :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **5\*104** | **105** | **2\*105** | **4\*105** | **8\*105** | **1.6\*2\*106** | **6.4\*106** | **12.8\*106** | **25.6\*106** |
| T | 0.000136 | 0.000288 | 0.000729 | 0.001385 | 0.002398 | 0.010145 | 0.030693 |  |  |
| **N** | **51.2\*106** | **1.024\*106** | **2.048\*106** |  | | | | | |
| T |  |  |  |

#### Non Triée (aléatoire) :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **5\*104** | **105** | **2\*105** | **4\*105** | **8\*105** | **1.6\*2\*106** | **6.4\*106** | **12.8\*106** | **25.6\*106** |
| T | 1.705814 | 7.158441 | 26.173725 | 107.608450 | 429.011678 | 527.906000 |  |  |  |
| **N** | **51.2\*106** | **1.024\*106** | **2.048\*106** |  | | | | | |
| T |  |  |  |

#### Triée (inverse) :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **5\*104** | **105** | **2\*105** | **4\*105** | **8\*105** | **1.6\*2\*106** | **6.4\*106** | **12.8\*106** | **25.6\*106** |
| T | 3.388014 | 14.099594 | 52.980701 | 216.402197 | 847.758600 |  |  |  |  |
| **N** | **51.2\*106** | **1.024\*106** | **2.048\*106** |  | | | | | |
| T |  |  |  |

## 2 – tri à bulles

### Description :

Le tri à bulles est un algorithme de tri simple qui fonctionne en comparant les éléments adjacents d'une liste et en les échangeant s'ils sont dans le mauvais ordre. L'algorithme parcourt la liste plusieurs fois, itérant à travers les éléments et déplaçant progressivement les éléments plus grands vers la fin de la liste.

### Algorithme :

Procédure : Tri à bulles

Entrée : T : tableau d’entiers ;

n : taille du tableau ;

Var : i, courant: entier ;

Permute : booléen ;

Début :

Permute = vrai ;

Tantque (permute = vrai) faire :

Permute := faux ;

Pour i := 0 a n -2 faire :

si ( T[i] > T[i+1] ) alors :

courant := T[i] ;

T[i] := T[i+1] ;

T[i+1] := courant ;

Permute := vrai ;

Fsi ;

Fait ;

Fait ;

Fin ;

Procédure : triABullesRecursive

Entrée : T : tableau d’entiers ;

n : entier ;

Var : courant : entier ;

Début :

Si n > 1 alors :

Pour i := 0 a n -2 faire :

si ( T[i] > T[i+1] ) alors :

courant := T[i] ;

T[i] := T[i+1] ;

T[i+1] := courant ;

Fsi ;

Fait ;

triABullesRecursive(T,n-1) ;

fsi ;

Fin ;

### Calcule de la complexité temporelle :

#### Meilleur cas :

Dans le meilleur des cas où le tableau est déjà trié, le tri à bulles effectuera une seule passe à travers la liste en comparant les éléments adjacents de 0 à n-2. Étant donné qu’aucune permutation ne sera nécessaire, la condition de permutation restera toujours fausse, ce qui conduira à la sortie anticipée de la boucle tantque. Ainsi, dans le scénario optimal d'une liste triée, la complexité temporelle du tri à bulles est linéaire, O(n)

#### Pure cas :

L'algorithme compare et échange le premier élément avec le deuxième, puis le deuxième avec le troisième, et ainsi de suite. À la fin de cette première passe, le plus grand élément est maintenant à la fin de la liste. Les itérations se poursuivent, déplaçant progressivement les éléments non triés vers leur position correcte. Cependant, à chaque passe, un seul élément est correctement placé à la fin de la liste

Il faut n-1 passes pour placer le deuxième plus grand élément, n-2 passes pour placer le troisième plus grand, et ainsi de suite. Le nombre total de passes est donc la somme de 1 à n-1, ce qui est équivalent à (n \* (n-1)) / 2.

La complexité temporelle du pire des cas pour le tri à bulles est donc O(n^2).

### Calcule de la complexité spatiale :

Le tri à bulles n'a généralement pas besoin d'une mémoire supplémentaire proportionnelle à la taille de la liste pour effectuer ses opérations de tri. Les échanges d'éléments se font généralement en utilisant des variables temporaires, ce qui n'ajoute pas de mémoire proportionnelle à la taille de la liste. Ainsi, la complexité spatiale du tri à bulles est souvent considérée comme constante O(1)

### Programme en C :

Void TriABulles(int T[], int n) {

int i, permute = 1 ;

while (permute == 1){

permute = 0;

for (i = 0; i < n - 1; i++) {

if (T[i] > T[i + 1]) {

int temp = T[i];

T[i] = T[i+1];

T[i+1]= temp;

permute = 1;

}

}

}

### Calcule du temps d’exécution pour différents échantillons :

#### triée :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 5\*104 | 105 | 2\*105 | 4\*105 | 8\*105 | 1.6\*2\*106 | 6.4\*106 | 12.8\*106 | 25.6\*106 |
| T |  |  |  |  |  |  |  |  |  |
| N | 51.2\*106 | 1.024\*106 | 2.048\*106 |  |  |  |  |  |  |
| T |  |  |  |  |  |  |  |  |  |

#### Non Triée (aléatoire) :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 5\*104 | 105 | 2\*105 | 4\*105 | 8\*105 | 1.6\*2\*106 | 6.4\*106 | 12.8\*106 | 25.6\*106 |
| T |  |  |  |  |  |  |  |  |  |
| N | 51.2\*106 | 1.024\*106 | 2.048\*106 |  |  |  |  |  |  |
| T |  |  |  |  |  |  |  |  |  |

#### Triée (inverse) :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 5\*104 | 105 | 2\*105 | 4\*105 | 8\*105 | 1.6\*2\*106 | 6.4\*106 | 12.8\*106 | 25.6\*106 |
| T |  |  |  |  |  |  |  |  |  |
| N | 51.2\*106 | 1.024\*106 | 2.048\*106 |  |  |  |  |  |  |
| T |  |  |  |  |  |  |  |  |  |

## 3 – tri fusion :

### Description :

Le tri fusion, ou "merge sort" en anglais, est un algorithme de tri efficace basé sur la stratégie diviser pour régner. Il divise la liste non triée en sous-listes de plus en plus petites jusqu'à ce que chaque sous-liste ne contienne qu'un seul élément. Ensuite, ces sous-listes sont fusionnées de manière récursive en comparant et en ordonnant les éléments, créant ainsi des sous-listes triées de plus en plus grandes jusqu'à ce que la liste entière soit triée.

### Algorithme :

Procédure : Fusion

Entrée : tab1, tab2 : tableaux d'entiers ;

n1, n2 : tailles des tableaux tab1 et tab2 respectivement ;

Sortie : T : tableau d'entiers triés résultant de la fusion de tab1 et tab2 ;

Var : i, j, k : entiers ;

T: tableau d’entiers;

Début :

i := 0 ; j := 0 ; k := 0 ;

// Fusion des deux tableaux triés

T := tableau de taille n1 + n2 ;

Tant que (i < n1 et j < n2) faire :

Si tab1[i] <= tab2[j] alors :

T[k] := tab1[i] ;

i := i + 1 ;

Sinon :

T[k] := tab2[j] ;

j := j + 1 ;

Fin Si ;

k := k + 1 ;

Fin Tant que ;

Tant que (i < n1) faire : // Ajout des éléments restants de tab1

T[k] := tab1[i] ;

i := i + 1 ;

k := k + 1 ;

Fin Tant que ;

Tant que (j < n2) faire : // Ajout des éléments restants de tab2

T[k] := tab2[j] ;

j := j + 1 ;

k := k + 1 ;

Fin Tant que ;

Fin.

Procédure : TriFusion

Entrée : T : tableau d'entiers ;

n : taille du tableau ;

Var : tab1, tab2 : tableau d'entiers ;

Début :

Si (n > 1) alors :

// Initialiser et copier la première moitié du tableau dans tab1

Pour i de 0 à n/2-1 faire :

tab1[i] := T[i] ;

fin pour ;

// Initialiser et copier la deuxième moitié du tableau dans tab2

Pour i de n/2 à n-1 faire :

tab2[i - n/2] := T[i] ;

fin pour ;

// Appeler récursivement TriFusion pour chaque moitié

TriFusion(tab1, n/2) ;

TriFusion(tab2, n - n/2) ;

Fusion(tab1, n/2, tab2, n - n/2, T) ;

Fin Si ;

Fin.

### Calcule de la complexité temporelle :

Dans le cas du tri fusion, le meilleur cas, le pire cas et le cas moyen ont tous la même complexité, Cela se produit lorsque l'algorithme effectue le même nombre d'opérations, peu importe la disposition initiale des éléments dans le tableau.

La complexité temporelle de l'algorithme de tri fusion dépend de deux parties principales : la division récursive et la fusion.

**Division récursive :**

L'algorithme divise récursivement la liste en deux parties jusqu'à ce que chaque partie ne contienne qu'un seul élément.

Il y a log(n) niveaux de division,

**Fusion :**

À chaque niveau de la récursion, toutes les paires de sous-listes sont fusionnées.

Chaque fusion nécessite de comparer et de déplacer chaque élément.

Il y a au plus n comparaisons et déplacements à chaque niveau.

La complexité totale de l'algorithme de tri fusion est donc le produit de la complexité de la division et de la complexité de la fusion. La complexité totale est exprimée comme O(n log n), où "n" est la taille de la liste à trier.

### Calcule de la complexité spatiale :

Dans une implémentation classique du tri fusion, où des copies temporaires sont créées pour stocker les sous-listes pendant la fusion, la complexité spatiale est généralement O(n), où n est la taille du tableau à trier. Cela signifie que l'algorithme nécessite un espace supplémentaire proportionnel à la taille du tableau initial pour stocker les sous-listes et effectuer les opérations de tri.

### Programme en C :

void triFusion(int T[], int n) {

if (n > 1) {

int i;

int n1 = n / 2;

int n2 = n - n1;

int tab1[n1];

int tab2[n2];

// Initialiser et copier la première moitié du tableau dans tab1

for (i = 0; i < n1; i++) {

tab1[i] = T[i];

}

// Initialiser et copier la deuxième moitié du tableau dans tab2

for (i = 0; i < n2; i++) {

tab2[i] = T[i + n1];

}

// Appeler récursivement triFusion pour chaque moitié

triFusion(tab1, n1);

triFusion(tab2, n2);

// Appeler la fonction de fusion

fusion(tab1, n1, tab2, n2, T);

}

}

void fusion(int tab1[], int n1, int tab2[], int n2, int T[]) {

int i = 0, j = 0, k = 0;

while (i < n1 && j < n2) {

if (tab1[i] <= tab2[j]) {

T[k] = tab1[i];

i++;

} else {

T[k] = tab2[j];

j++;

}

k++;

}

while (i < n1) {

T[k] = tab1[i];

i++;

k++;

}

while (j < n2) {

T[k] = tab2[j];

j++;

k++;

}

}

### Calcule du temps d’exécution pour différents échantillons :

#### triée :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **5\*104** | **105** | **2\*105** | **4\*105** | **8\*105** | **1.6\*2\*106** | **6.4\*106** | **12.8\*106** | **25.6\*106** |
| T | 0.009000 | 0.015000 | 0.031000 | 0.062000 | 0.12500 | 0.514000 | 1.087000 | 2.220000 | 4.678000 |
| **N** | **51.2\*106** | **1.024\*106** | **2.048\*106** |  | | | | | |
| T | 9.360000 | 19.502000 | 105.634000 |

#### Non Triée (aléatoire) :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **5\*104** | **105** | **2\*105** | **4\*105** | **8\*105** | **1.6\*2\*106** | **6.4\*106** | **12.8\*106** | **25.6\*106** |
| T | 0.009000 | 0.016000 | 0.031000 | 0.078000 | 0.18800 | 0.782000 | 1.562000 | 3.202000 | 6.759000 |
| **N** | **51.2\*106** | **1.024\*106** | **2.048\*106** |  | | | | | |
| T | 13.733000 | 27.899000 | 137.502000 |

#### Triée (inverse) :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **5\*104** | **105** | **2\*105** | **4\*105** | **8\*105** | **1.6\*2\*106** | **6.4\*106** | **12.8\*106** | **25.6\*106** |
| T | 0.005000 | 0.016000 | 0.016000 | 0.047000 | 0.125000 | 0.532000 | 1.077000 | 2.221000 | 4.909000 |
| **N** | **51.2\*106** | **1.024\*106** | **2.048\*106** |  | | | | | |
| T | 9.598000 | 20.139000 | 44.633000 |

Remarque : La variance des résultats obtenus peut être influencée par plusieurs facteurs, tels que le l'état initial de la mémoire, ou d'autres conditions d'exécution spécifiques à la machine.

## 4 – Tri rapide

### Description :

Un algorithme de tri basé sur la stratégie diviser pour régner. Son principe fondamental consiste à choisir un élément du tableau appelé "pivot" et à partitionner le tableau de telle sorte que les éléments plus petits que le pivot se trouve à sa gauche, et les éléments plus grands se trouvent à sa droite. Cette opération de partitionnement est effectuée récursivement sur les sous-tableaux avant et après le pivot jusqu'à ce que l'ensemble entier soit trié.

### Algorithme :

Procédure : TriRapid

Entrée : T : tableau d'entiers ;

deb : indice du début du tableau ;

fin : indice du fin du tableau ;

Var :

Début :

Si (deb < fin) alors :

// Partitionnement et choix du pivot

pivot := partitionner(tableau, début, fin) ;

// Tri récursif des sous-tableaux à gauche et à droite du pivot

TriRapid (tableau, début, pivot - 1) ;

TriRapid (tableau, pivot + 1, fin) ;

Fin.

Fonction choisirPivot(tableau, début, milieu, fin) :entier

// Comparaisons pour trouver le médian de trois

Si tableau[début] < tableau[milieu] alors

Si tableau[milieu] < tableau[fin] alors

Retourner milieu

Sinon Si tableau[début] < tableau[fin] alors

Retourner fin

Sinon

Retourner début

Sinon

Si tableau[début] < tableau[fin] alors

Retourner début

Sinon Si tableau[milieu] < tableau[fin] alors

Retourner fin

Sinon

Retourner milieu

Fin ;

Fonction : partitionner

Entrée : T : tableau d'entiers ;

deb : indice du début du tableau ;

fin : indice du fin du tableau ;

sortie : entier l’indice du pivot

Var : i, pivot, milieu, temp, pivotIndice: entier ;

Début :

milieu := (début + fin) / 2

// Sélection du médian de trois comme pivot

pivotIndice := choisirPivot(tableau, début, milieu, fin) ;

pivot := T[pivotIndice] ;

// permuter le pivot avec le dernier élément pour le placer temporairement à la fin

temp := T[pivotIndice] ;

T[pivotIndice] := T[fin] ;

T[fin] := temp ;

i := début – 1 ;

Pour j de début à fin - 1 faire

Si T[j] <= pivot alors

i := i + 1

temp := T[i] ;

T[i] := T[j] ;

T[j] := temp ;

// Remettre le pivot à sa position correcte

temp := T[i+1] ;

T[i+1] := T[fin] ;

T[fin] := temp ;

Retourner i + 1 ;

Fin ;

### Calcule de la complexité temporelle :

La complexité temporelle du tri rapide dépend du choix du pivot. On a choisi le médian 3 comme pivot.

#### Meilleur cas :

Dans le meilleur des cas, le choix du pivot est optimal à chaque niveau de la récursion, conduisant à une division équilibrée du tableau à chaque étape. Cela se produit lorsque le pivot est toujours le médian des trois éléments (le premier, le milieu et le dernier) dans le sous-tableau considéré.

La complexité temporelle dans le meilleur des cas est déterminée par la hauteur de l'arbre de récursion, et chaque niveau de l'arbre nécessite un temps linéaire pour effectuer la partition. La hauteur de l'arbre de récursion dans le meilleur des cas est Log(n)

Par conséquent, la complexité temporelle dans le meilleur des cas est O(nlogn).

#### Moyenner cas :

Division du Tableau : À chaque niveau de l'arbre de récursion, le tableau est divisé en deux parties grâce à la fonction de partitionnement. En moyenne, le pivot est choisi de manière à diviser le tableau en deux parties équilibrées.

Nombre de Niveaux : La hauteur totale de l'arbre de récursion dépend du nombre de fois où le tableau peut être divisé avant d'atteindre des sous-tableaux de taille 1. En moyenne, chaque niveau de l'arbre divise le tableau par 2.

Complexité Temporelle Moyenne : En moyenne, chaque niveau a un coût linéaire, et le nombre total de niveaux est log(n) Par conséquent, la complexité temporelle moyenne est O(nlogn).

#### Pire cas :

Le pire cas se produit lorsque le pivot est toujours choisi de manière à diviser le tableau de manière très inéquitable, (Si, à chaque étape, le premier élément est choisi comme pivot, et si le tableau est déjà trié, alors le pivot sera toujours la plus petite valeur du sous-tableau. Cela entraîne une division inéquitable à chaque étape.) conduisant à une hauteur d'arbre de récursion maximale. Dans ce scénario, chaque niveau de la récursion nécessite un temps linéaire pour effectuer la partition, et le nombre total de niveaux est égal à la taille du tableau n. donc sera O(n2).

Avec le choix du médian de trois comme pivot, la complexité temporelle moyenne reste O(nlogn). Cependant, le choix du médian de trois réduit le risque d'obtenir le pire cas dans des situations particulières, ce qui peut être fréquent avec d'autres choix de pivot.

### Calcule de la complexité spatiale :

La complexité spatiale du tri rapide est souvent de O(logn) en moyenne, mais elle peut atteindre O(n) dans le pire cas, en fonction de la hauteur de l'arbre de récursion.

**En moyenne O(logn))** : Dans la plupart des cas, le tri rapide utilise une approche "quicksort en queue" qui réduit la profondeur de la pile de récursion, et donc la complexité spatiale moyenne est de l'ordre de O(logn). Cela signifie que l'utilisation de la mémoire augmente logarithmiquement avec la taille de l'entrée.

**Dans le pire cas O(n))** : Dans le pire cas, la hauteur de l'arbre de récursion peut atteindre la taille du tableau (n), et chaque appel récursif ajoute un nouveau frame à la pile. Cela peut conduire à une complexité spatiale de l'ordre de O(n), car chaque élément du tableau nécessitera son propre frame sur la pile.

### Algorithme en c :

int partition(int arr[], int deb, int fin) {

int pivotIndex = medianOfThree(arr, deb, fin);

int pivot = arr[pivotIndex];

// permute le pivot avec le dernier elment tempo

swap(&arr[pivotIndex], &arr[fin]);

int i = deb - 1;

for (int j = deb; j < fin; j++) {

if (arr[j] <= pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

// place le pivot dans la position just

swap(&arr[i + 1], &arr[fin]);

return i + 1;

}

void triRapide(int arr[], int deb, int fin) {

if (deb < fin) {

int pivotIndex = partition(arr, deb, fin);

// tri recursive des deux sous-tableau

triRapide (arr, deb, pivotIndex - 1);

triRapide (arr, pivotIndex + 1, fin);

}

}

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

### Calcule du temps d’exécution pour différents échantillons :

#### triée :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **5\*104** | **105** | **2\*105** | **4\*105** | **8\*105** | **1.6\*2\*106** | **6.4\*106** | **12.8\*106** | **25.6\*106** |
| T | 0.003000 | 0.007000 | 0.014000 | 0.027000 | 0.05700 | 0.248000 | 0.499000 | 1.028000 | 2.137000 |
| **N** | **51.2\*106** | **1.024\*106** | **2.048\*106** |  | | | | | |
| T | 4.462000 | 9.311000 | 19.427000 |

#### Non Triée (aléatoire) :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **5\*104** | **105** | **2\*105** | **4\*105** | **8\*105** | **1.6\*2\*106** | **6.4\*106** | **12.8\*106** | **25.6\*106** |
| T | 0.008000 | 0.016000 | 0.032000 | 0.066000 | 0.13200 | 0.716000 | 1.973000 | 5.931000 | 19.972000 |
| **N** | **51.2\*106** | **1.024\*106** | **2.048\*106** |  | | | | | |
| T | 71.514000 | 258.331000 | 1041.597000 |

#### Triée (inverse) :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **5\*104** | **105** | **2\*105** | **4\*105** | **8\*105** | **1.6\*2\*106** | **6.4\*106** | **12.8\*106** | **25.6\*106** |
| T | 0.002000 | 0.013000 | 0.025000 | 0.052000 | 0.12400 | 0.473000 | 1.049000 | 2.124000 | 4.745000 |
| **N** | **51.2\*106** | **1.024\*106** | **2.048\*106** |  | | | | | |
| T | 10.029000 | 22.495000 | 40.421000 |