# 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 |  |  |  |  |  |  |  |  |  |