



1ère Année Master Informatique, Semestre 1

Option : Systèmes informatiques Intelligents (SII)

Module : Conception et Complexité des Algorithmes

<p><i><u>MINI PROJET N°1</u></i></p> <p>ETUDE EXPERIMENTALE DES ALGORITHMES DE TRI</p>

- OUHOCINE Sarah

Professeur :

- M^{me} m.DJEFFAL

Table des matières

1 Introduction générale

1.1 Un algorithme de tri, c'est quoi ?	
1.2 Critères de classification d'un algorithme de tri.....	
1.3 Complexité algorithmique.....	
1.3.1 Complexité temporelle.....	
1.3.2 Complexité spatiale.....	
1.4 Quelques exemples des algorithmes de tri.....	
1.4.1 Tris par comparaison.....	
1.4.2 Tris utilisant la structure de données.....	
1.4.3 Tri externes.....	
1.5 Quelques Notions importantes.....	
1.5.1 Diviser pour régner !	
1.5.2 Qu'est-ce qu'un tas !.....	

2 Partie I : Etude des Algorithmes

2.1 Algorithme 1 : Tri par Insertion.....	
2.1.1 Description de la méthode de tri.....	
2.1.2 Illustration avec un exemple.....	
2.1.3 Ecriture de l'algorithme en pseudo langage.....	
2.1.4 Calcule de la complexité Temporelle.....	
2.1.5 Calcule de la complexité Spatiale.....	
2.1.6 Ecriture du programme en langage C.....	
2.1.7 Mesure des temps d'exécution.....	
2.1.8 Représentation Graphique.....	
2.1.9 Comparaison de la complexité théorique avec la complexité expérimentale.....	
2.2 Algorithme 2 : Tri à bulles.....	
2.2.1 Description de la méthode de tri.....	
2.2.2 Illustration avec un exemple.....	
2.2.3 Ecriture de l'algorithme en pseudo langage.....	
2.2.4 Calcule de la complexité Temporelle.....	
2.2.5 Calcule de la complexité Spatiale.....	
2.2.6 Ecriture du programme en langage C.....	
2.2.7 Mesure des temps d'exécution.....	
2.2.8 Représentation Graphique.....	
2.2.9 Comparaison de la complexité théorique avec la complexité expérimentale.....	
2.3 Algorithme 3 : Tri par fusion.....	
2.3.1 Description de la méthode de tri.....	
2.3.2 Illustration avec un exemple.....	
2.3.3 Ecriture de l'algorithme en pseudo langage.....	
2.3.4 Calcule de la complexité Temporelle.....	
2.3.5 Calcule de la complexité Spatiale.....	
2.3.6 Ecriture du programme en langage C.....	
2.3.7 Mesure des temps d'exécution.....	
2.3.8 Représentation Graphique.....	

2.3.9 Comparaison de la complexité théorique avec la complexité expérimentale.....	
2.4 Algorithme 4 : Tri Rapide (Quick Sort).....	
2.4.1 Description de la méthode de tri.....	
2.4.2 Illustration avec un exemple.....	
2.4.3 Ecriture de l'algorithme en pseudo langage.....	
2.4.4 Calcule de la complexité Temporelle.....	
2.4.5 Calcule de la complexité Spatiale.....	
2.4.6 Ecriture du programme en langage C.....	
2.4.7 Mesure des temps d'exécution.....	
2.4.8 Représentation Graphique.....	
2.4.9 Comparaison de la complexité théorique avec la complexité expérimentale.....	
2.5 Algorithme 5 : Tri par tas (Heap Sort).....	
2.5.1 Description de la méthode de tri.....	
2.5.2 Illustration avec un exemple.....	
2.5.3 Ecriture de l'algorithme en pseudo langage.....	
2.5.4 Calcule de la complexité Temporelle.....	
2.5.5 Calcule de la complexité Spatiale.....	
2.5.6 Ecriture du programme en langage C.....	
2.5.7 Mesure des temps d'exécution.....	
2.5.8 Représentation Graphique.....	
2.5.9 Comparaison de la complexité théorique avec la complexité expérimentale.....	
3 Partie II : Comparaison des Algorithmes	
3.1 Représentation des graphes des complexités des 5 algorithmes.....	
3.1.1 Complexité théorique.....	
3.1.2 Complexité expérimentale.....	
3.2 Représentation des tableaux comparatifs des complexités et des temps d'exécution des 5 algorithmes.....	
3.2.1 Complexité théorique.....	
3.2.2 Temps d'exécution.....	
3.3 Comparaison des 5 Algorithmes.....	
4 Annexe	
4.1 Fonctions et procédure communes à tous les programmes.....	
4.1.1 Fonction de création et de calcule du temps d'exécution de la fonction de tri pour un tableau comportant des données en ordre croissant (Bon Ordre).....	
4.1.2 Fonction de création et de calcule du temps d'exécution de la fonction de tri pour un tableau comportant des données en ordre décroissant (Ordre Inverse).....	
4.1.3 Fonction de création et de calcule du temps d'exécution de la fonction de tri pour un tableau comportant des données non triés (Ordre Aléatoire/ Quelconque).....	
5 Matériel utilisé	

1 Introduction générale

1.1 Un algorithme de tri, c'est quoi ?

Un [algorithme de tri](#) est un algorithme qui permet d'organiser une collection d'objets selon une relation d'ordre déterminée. Les objets à trier sont des éléments d'un ensemble muni d'un ordre total. Il est par exemple fréquent de trier des entiers selon la relation d'ordre usuelle « est inférieur ou égal à ». Les algorithmes de tri sont utilisés dans de très nombreuses situations. Ils sont en particulier utiles à de nombreux algorithmes plus complexes dont certains algorithmes de recherche, comme la recherche par dichotomie. Ils peuvent également servir pour mettre des données sous forme canonique* ou les rendre plus lisibles pour l'utilisateur.

La collection à trier est souvent donnée sous forme de tableau, afin de permettre l'accès direct aux différents éléments de la collection, ou sous forme de liste, ce qui peut se révéler être plus adapté à certains algorithmes et à l'usage de la programmation fonctionnelle.

1.2 Critères de classification d'un algorithme de tri :

La classification des algorithmes de tri est très importante, car elle permet de choisir l'algorithme le plus adapté au problème traité, tout en tenant compte des contraintes imposées par celui-ci. La principale caractéristique qui permet de différencier les algorithmes de tri, outre leur principe de fonctionnement, est la [complexité algorithmique](#).

1.3 Complexité algorithmique :

1.3.1 Complexité temporelle :

La [complexité temporelle](#) ([en moyenne](#) ou [dans le pire des cas](#)) mesure le nombre d'opérations élémentaires effectuées pour trier une collection d'éléments. C'est un critère majeur pour comparer les algorithmes de tri, puisque c'est une estimation directe du temps d'exécution de l'algorithme. Dans le cas des algorithmes de tri par comparaison, la complexité en temps est le plus souvent assimilable au nombre de comparaisons effectuées, la comparaison et l'échange éventuel de deux valeurs s'effectuant en temps constant.

La complexité en temps est souvent notée T et exprimée comme une fonction du nombre n d'éléments à trier à l'aide des notations de Landau O

1.3.2 Complexité spatiale :

La [complexité spatiale](#) ([en moyenne](#) ou [dans le pire des cas](#)) représente, quant à elle, la quantité de mémoire dont va avoir besoin l'algorithme pour s'exécuter. Celle-ci peut dépendre, comme le temps d'exécution, de la taille de l'entrée. Il est fréquent que les complexités spatiales en moyenne et dans le pire des cas soient identiques.

1.4 Quelques exemples des algorithmes de tri :

1.4.1 Tris par comparaison :

Les algorithmes procédant par comparaisons successives entre éléments, dits « tris par comparaisons », ils lisent les entrées uniquement au moyen d'une fonction de comparaison binaire ou ternaire (lorsque le cas d'égalité est traité différemment). Il existe encore différents principes de fonctionnement au sein de cette classe : certains algorithmes de tri par comparaison procèdent [par sélection](#), d'autres [par insertions successives](#), d'autre tri [à bulles](#), tris [cocktail](#), tri [pair-impair](#), tri [stupide](#), tri de [Shell](#), tri [par fusions](#), tri [rapide](#), tri [par tas](#), tri [introsort](#), tri [arborescent](#), tri [smoothsort](#), tri [à peigne](#) ...

1.4.2 Tris utilisant la structure de données :

C'est des algorithmes plus spécialisés faisant des hypothèses restrictives sur la structure des données à trier (par exemple, le tri [par comptage](#) ou [tri par dénombrement](#), applicable uniquement si les données sont prises dans un ensemble borné connu à l'avance), tri [par base](#), tri [par paquets](#)...

1.4.3 Tris externes :

Ces algorithmes sont souvent basés sur une approche assez voisine de celle du [tri fusion](#). Le principe est le suivant :

- découpage du volume de données à trier en sous-ensembles de taille inférieure.
- tri de chaque sous-ensemble pour former des « monotonies » (sous-ensembles triés).
- interclassement des monotonies (trie).

1.5 Quelques Notions importantes :

1.5.1 Diviser pour régner !

Les algorithmes qui suivent généralement l'approche dite "**diviser pour régner**" ce sont des [tris externes](#), ils séparent le problème en plusieurs sous-problèmes semblables au problème initial mais de taille moindre, résolvent les sous-problèmes de façon récursive, puis combinent toutes les solutions pour produire la solution du problème original. L'un des avantages des algorithmes "diviser-pour-régner" est que leurs temps d'exécution sont souvent faciles à déterminer. Le paradigme "diviser-pour-régner" implique trois étapes à chaque niveau de la récursivité :

- **Diviser** : le problème en un certain nombre de sous-problèmes.
- **Régner** : sur les sous-problèmes en les résolvant de manière récursive. Si la taille d'un sous-problème est suffisamment réduite, on peut toutefois le résoudre directement.
- **Combiner** : les solutions des sous-problèmes pour produire la solution du problème original.
-

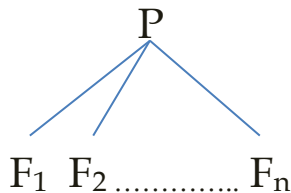
1.5.2 Qu'est-ce qu'un tas !

Un tas est une structure de données de type arbre, tel que ses clés sont ordonnées selon la propriété de tas :

- La clé d'un nœud parent à une plus haute priorité que les clés de ses enfants. La "priorité" signifie ici que les clés des enfants sont toutes inférieures à la clé de la racine (la clé maximale).

On dit qu'un arbre est ordonné en tas lorsque la propriété suivante est vérifiée :

- Pour tous nœuds P et F de l'arbre tels que F soit un fils de P : $\text{clé}(P) > \text{clé}(F)$



$$P.\text{clé} > F_i.\text{clé} \quad / \quad i = \{1, 2, \dots, n\}$$

2 Partie I : Etude des Algorithmes

2.1 Algorithme 1 : Tri par Insertion

2.1.1 Description de la méthode de tri

Le tri par insertion est un algorithme de tri classique. La plupart des personnes l'utilisent naturellement pour trier des cartes à joueurs.

Son principe est de parcourir la liste non triée (I_1, I_2, \dots, I_n) en la décomposant en deux parties :

- Une partie déjà triée (**PT**)
- Une partie non triée (**PNT**).

La méthode est identique à celle que l'on utilise pour ranger des cartes que l'on tient dans sa main : on insère dans le paquet de cartes déjà rangées une nouvelle carte au bon endroit.

L'opération de base consiste à prendre l'élément frontière (le plus à gauche) dans la PNT, puis à l'insérer à sa place dans la PT (place que l'on recherchera séquentiellement), puis à déplacer la frontière d'une position vers la droite.


Ces insertions s'effectuent tant qu'il reste un élément à ranger dans la PNT.

L'insertion de l'élément frontière est effectuée par décalages successifs d'une cellule.

2.1.2 Illustration avec un exemple

 Partie Non Trié (PNT)

 Partie Trié (PT)

 L'Elément à insérer (l'Elément le plus à gauche dans la PNT)

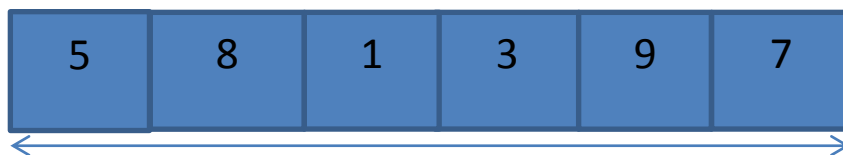


Tableau Non Trié (PNT)

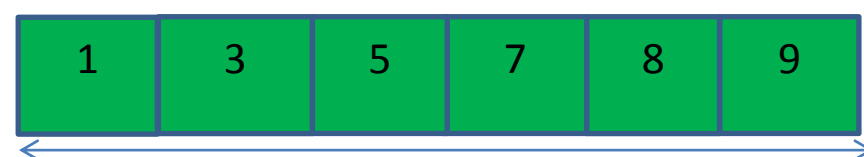
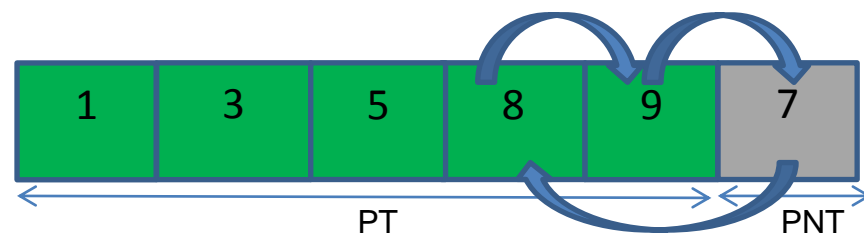
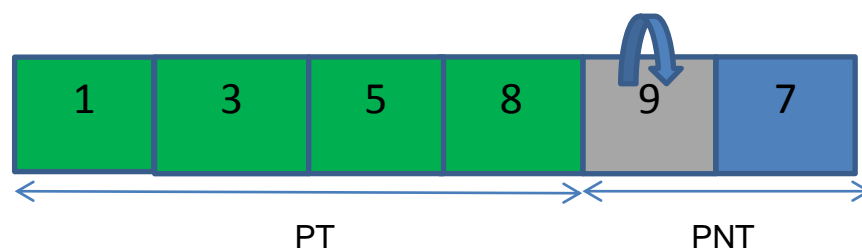
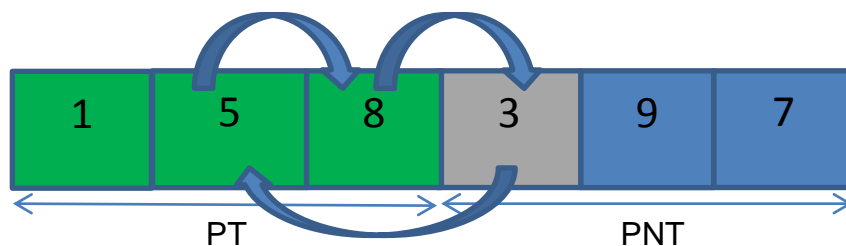
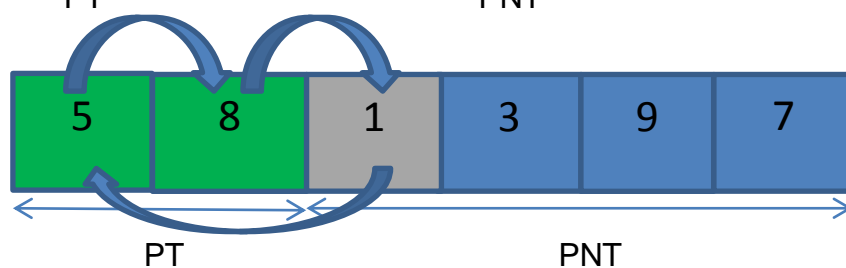
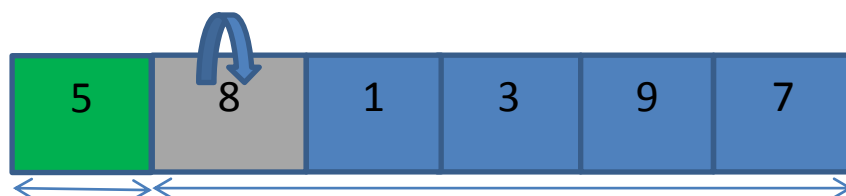


Tableau Trié (PT)

2.1.3 Ecriture de l'algorithme en pseudo langage

➤ Itératif

```
fonction tri_insertion_itératif (T, n) : Tableau d'entier ;
debut
    pour i de 1 a n - 1
    faire
        insert (T, n, i) ;
    fait ;
retourner T ;
fin;

procédure insert (T[]:d'entier , n:entier , i: entier )
VAR x: entier ;
DEBUT
    x = T[i] ;
    tant que (i > 0 et T[i - 1] > x)
    faire
        T[i] = T[i - 1] ;
        i = i - 1 ;
    fait ;
    T[i] = x ;
FIN .
```

➤ Récuratif

```
fonction tri_insertion_récuratif (T[]:d'entier , n:entier , i: entier ) : tableau d'entier :
VAR
DEBUT
    SI(i < n) ALORS
        T := insert (T, n, i + 1);
    FinSi ;
    retourner T;
FIN .

fonction insert (T, n, i) : tableau d'entier ;
VAR x: entier ;
DEBUT
    x = T[i] ;
    tant que (i > 0 et T[i - 1] > x)
    faire
        T[i] = T[i - 1] ;
        i = i - 1 ;
    fait ;
    T[i] = x ;
    retourner T;
FIN .
```

2.1.4 Calcule de la complexité Temporelle

>> Meilleur Cas :

(Il s'agit du cas où le tableau est déjà trié)

Dans ce cas l'algorithme ne rentre jamais dans la boucle tant que, ainsi il y aura **n-1** comparaisons et au plus **n** affectations. → $CT(n) = n + (n-1) = 2n - 1$.

Et donc la complexité au meilleur cas est linéaire c.-à-d. de l'ordre $n \rightarrow O(n)$.

>> Pire cas

(il s'agit du cas où le tableau est trié dans l'ordre inverse) Dans ce cas le nombre de comparaisons "**Tantque Tab[i-1] > x faire**" est une valeur qui ne dépend que de la longueur i de la partie (i_1, i_2, \dots, i_i) déjà rangée.

Il y a donc au pire i comparaisons pour chaque i variant de **2** à **n** : La complexité au pire cas en nombre de comparaison c'est la somme des n termes suivants ($i = 2, i = 3, \dots, i = n \rightarrow$ (c'est la somme des n premiers entiers moins 1).

$$CT(n) = \sum_{i=2}^n i = 2 + 3 + 4 + \dots + n = (n(n+1)/2) - 1 = ((n^2 + n - 2)/2)$$

Et donc la complexité au pire cas est quadratique c.-à-d. de l'ordre $n^2 \rightarrow O(n^2)$.

2.1.5 Calcul de la complexité Spatiale

L'algorithme de tri par insertion utilise :

- Un tableau d'entier en question (de taille n) à trier.
- 2 variables de type entier (i et n).
-

En considérant que la taille d'un entier est sur 4 Octets. Nous obtenons la complexité spatiale : → $CS(n) = 4(n + 2) \text{ octets} = 4n + 8 \rightarrow CS(n) = O(n)$.

2.1.6 Ecriture du programme en langage C

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

void tri_insertion( long int *array , long int n) {
    for(int i = 1; i < n; i++) {
        long int valeur_a_inserer = array [i];
        long int j = i;

        // Vérifier si le précédent est supérieur à la valeur à insérer

        while (j > 0 && array [j -1] > valeur_a_inserer ) {
            array [j] = array [j -1];
            j --;
        }
        array [i] = valeur_a_inserer ;
    }
}
```

2.1.7 Mesure des temps d'exécution

On a vu précédemment que l'obtention du temps d'exécution sera pour les trois cas trois (3) suivants :

1. Les données du tableau sont dans l'ordre.
2. Les données du tableau sont dans un ordre aléatoire.
3. Données du tableau dans l'ordre inverse.

Taille du Tableau (n)	50000	100000	200000	400000	800000	1600000
Bon Ordre	0.000338	0.000563	0.001089	0.001996	0.004169	0.008278
Ordre Aléatoire	0.002114	0.004120	0.009080	0.019512	0.043121	0.086129
Ordre Inverse	3.934113	17.157114	62.752112	267.088113	1116.756111	4690.376113
Taille du Tableau (n)	3200000	6400000	12800000	25600000	51200000	/
Bon Ordre	0.0160176	0.033724	0.073385	0.168638	0.370869	/
Ordre Aléatoire	0.168704	0.357526	0.757107	1.652573	3.795680	/
Ordre Inverse	19934.098113	86713.329114	346853.319112	1508811.938113	6789653.719111	/

2.1.8 Représentation Graphique (à fin de la partie 1)

2.1.9 Comparaison de la complexité théorique avec la complexité expérimentale

a) Ordre Inverse :

On remarque que les temps d'exécution sont approximativement multipliés par 4 lorsque n est doublé.

Exemples :

$$N0 = 50000 \rightarrow T0 = 3.934113$$

$$N1 = 100000 \rightarrow 2 * N0 \rightarrow T1 = 17.157114 \approx 4 * T0 = 2^2 * T0$$

$$N2 = 200000 \rightarrow 2 * N1 \rightarrow T2 = 62.752112 \approx 4 * T1 = 2^2 * T1$$

On en déduit que le temps d'exécution est proportionnel à n, donc on peut le représenter par la formule suivante :

$$T1(x * N) = x^2 * T1(N) \text{ pour tous } x * N \text{ qui appartient à } [50000 - 2048000000] \dots 1$$

(x étant la tangente d'un point sur le graphe).

b) Ordre Bon et Ordre Aléatoire :

On constate que lorsque la taille n du est doublé, le temps d'exécution est approximativement doublé aussi.

Exemples :

$$N1 = 50000 \rightarrow T1 = 0.000338$$

$$N2 = 100000 = 2 * N1 \rightarrow T2 = 0.000563 \approx 2 * T1$$

On en déduit que le temps d'exécution est proportionnel à N, donc on peut le représenter par la formule suivante :

$$T2(x * N) = x * T2(N) \text{ pour tous } x * N \text{ qui appartient à } [50000 - 2048000000] \dots 2$$

(x étant la tangente d'un point sur le graphe).

Ainsi les données expérimentales de la complexité suivent une des deux fonctions soit T1 au pire cas, soit T2 au meilleur cas et au cas moyen.

Et comme la complexité théorique au pire cas est de l'ordre de : $P(n) = n^2$

Et la complexité théorique au meilleur cas est de l'ordre de : $M(n) = n$

On conclue que :

$$M(n) < T1(n) < P(n) \text{ tel que } n < Cte^2 * n < n^2 \text{ avec } Cte=2 \text{ et } n \gg \gg \gg \dots 1$$

et

$$M(n) < T2(n) < P(n) \text{ tel que } n < Cte * n < n^2 \text{ avec } Cte = 2 \text{ et } n \gg \gg \gg \dots 2$$

→ CONCLUSION :

Les données obtenues sont bornées (majorées par les données du pire cas et minorées par les données au meilleur cas). Et donc :

" ☺ Le modèle théorique est conforme avec les données expérimentales ☺ "

→ PS : Nous ne pouvant pas généraliser car les tests faits n'englobent pas toutes les valeurs possibles ☹.

2.2 Algorithme 2 : Tri à bulles

2.2.1 Description de la méthode de tri :

Le tri à bulles consiste à parcourir un tableau à plusieurs reprises, en comparant les éléments du tableau deux à deux à chaque itération et en le permutant selon le tri (croissant ou décroissant).

→ Étudions le cas croissant.

Après un premier parcours du tableau, nous aurons l'élément le plus grand (maximum) du tableau à la dernière position c.-à-d. la $n^{\text{ième}}$ case.

> Nous répétons le travail une autre fois tout en excluant la dernière case du tableau (triée déjà au parcours précédent) et donc à la fin de ce nouveau parcours le nouvel élément le plus grand dans les $(n-1)$ cases du tableau se positionnera à l'avant dernière case du tableau c.-à-d. la $(n-1)^{\text{ième}}$ case.

Nous répétons l'opération $N-1$ fois (N étant la taille du tableau) c.-à-d. jusqu'à la case 2.

2.2.2 Illustration avec un exemple




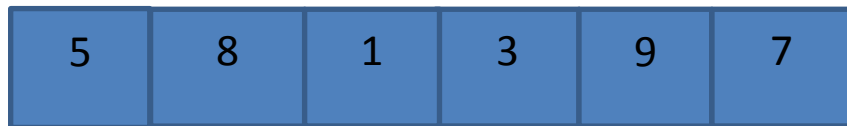
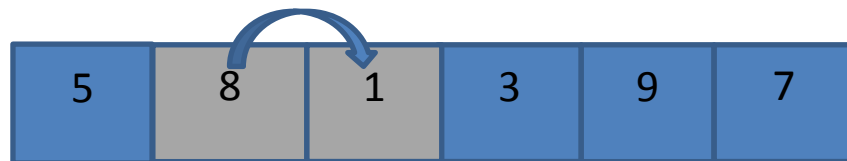
-  Partie Non Trié
-  Partie Trié
-  Les éléments à permuter

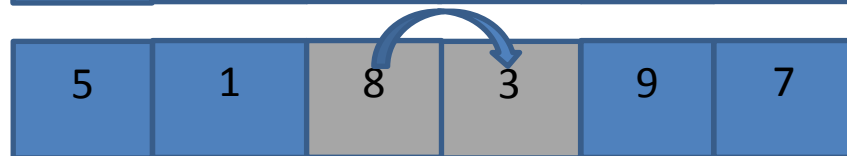
Tableau initial



5 < 8 pas de permutation
Passer à 8
8 > 1 permuter



8 > 3 permuter



8 < 9 pas de permutation
Passer à 9
9 > 7 permuter



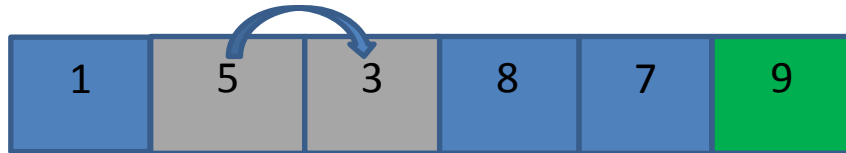
5 > 1 permuter



← Refaire le même travail avec les $n-1$ cases

→ trié

5>3 permuter



5<8 pas de permutation
Passer à 8
8>7 permuter



1<3 passer à 3
3<5 passer à 5
5<7 passer à 7



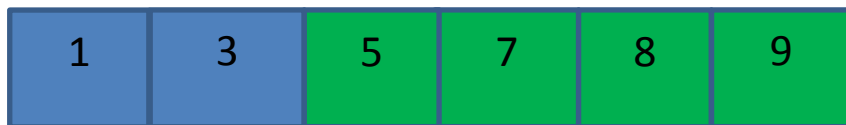
← Refaire le même travail avec les **n-2** cases → trié

1<3 passer à 3
3<5 passer à 5

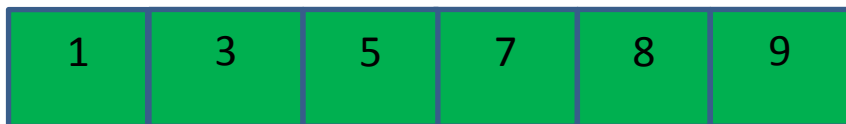


← Refaire le même travail avec les **n-3** cases → trié

1<3 passer à 3



← Refaire le même travail avec **n-4** cases → trié



trié

2.2.3 Ecriture de l'algorithme en pseudo langage

➤ Itératif :

```

procedure tri_a_bulles (tab []:d'entier , taille : entier )
VAR
i,j,n : entier ;
DEBUT
    Pour i de taille à 2
    faire
        Pour j = 1 à i -1
        faire
            SI (tab [j] > tab [j+1] ) ALORS

```

```

    permuter (tab[j], tab[j +1]) ;
  FinSi ;
fait ;
fait ;
FIN .

```

```

procedure permmuter (x:entier , y: entier )
VAR
temp : entier ;
DEBUT
  temp = x;
  x = y;
  y = temp ;
FIN .

```

2.2.4 Calcule de la complexité Temporelle

>> **Meilleur Cas :**

(Il s'agit du cas ou le tableau est déjà trié)

Dans ce cas l'algorithme effectue n^2 comparaisons et aucune permutation $CT(n) = n^2$
Et donc la complexité dans ce cas est quadratique c.-à-d. de l'ordre $n^2 \rightarrow O(n^2)$.

>> **Pire cas**

(il s'agit du cas ou le tableau est trié dans l'ordre inverse) Dans ce cas le programme effectue donc un nombre important de permutation, comme le montre l'illustration ci-dessus, sa complexité est :

$$CT(n) = (n-2) + 1 + ([n-1]-2) + 1 + \dots + 1 + 0 = (n-1) + (n-2) + \dots + 1$$

$$CT(n) = n(n-1)/2 = (n^2 - n)/2$$

Et donc la complexité au pire cas est quadratique c.-à-d. de l'ordre $n^2 \rightarrow O(n^2)$.

2.2.5 Calcule de la complexité Spatiale

L'algorithme de tri à bulles utilise :

- Un tableau d'entier en question (de taille n) à trier.
- 3 variables de type entier (i et j et n).
-

En considérant que la taille d'un entier est sur 4 Octets. Nous obtenant la complexité spatiale : $\rightarrow CS(n) = 4(n + 3) \text{ octets} = 4n+12 \rightarrow CS(n) = O(n)$.

2.2.6 Ecriture du programme en langage C

```

long long int * tri_bulles ( long long int *tab , long long int size )
{
  long long int i,j, temp =0,n;
  n = size -1;
  for(i=n; i >=0; i --)
  {

```

```

    for (j=1; j <= i; j++)
    {
        if ( tab [j] < tab [j -1])
        {
            temp = tab [j];
            tab [j] = tab[j -1];
            tab [j -1] = temp ;
        }
    }
    return tab ;
}

```

2.1.7 Mesure des temps d'exécution

On a vu précédemment que l'obtention du temps d'exécution sera pour les trois cas trois (3) suivants :

1. Les données du tableau sont dans l'ordre.
2. Les données du tableau sont dans un ordre aléatoirement.
3. Données du tableau dans l'ordre inverse.

Taille du Tableau (n)	50000	100000	200000	400000	800000	1600000
Bon Ordre	5.233	24.241	100.244	414.318	1702.147	7152.576
Ordre Aléatoire	13.518	59.582	246.927	997.615	4124.499	16913.89
Ordre Inverse	9.681	50.425	176.982	815.631	2750.186	11967.03
Taille du Tableau (n)	3200000	6400000	12800000	25600000	51200000	/
Bon Ordre	30115.912	120466.98	503555.528	2125007.906	7798781.981	/
Ordre Aléatoire	69012.093	279502.37	1118012.795	4349072.984	17396695.27	/
Ordre Inverse	47870.345	189092.75	869841.123	3192324.678	12769315.9	/

2.1.9 Comparaison de la complexité théorique avec la complexité expérimentale

Que ce soit dans l'Ordre Inverse, l'Ordre Bon, Ou l'Ordre Aléatoire :

Nous remarquons que les temps d'exécution évoluent de manière quadratique, ce qui signifie que si par exemple nous doublons le nombre de n en entrée, le temps d'exécution a son tour se multiplie par 4.

Le temps d'exécution est donc proportionnel à n , et nous pouvons le représenter par la formule suivante :

$$T(x * N) = x^2 * T(N) \text{ pour tous } x * N \text{ qui appartient a } [50000 - 2048000000]$$

(x étant la tangente d'un point sur le graphe).

Exemple :

$$N1 = 50000 \Rightarrow T1 = 13.518$$

$$N2 = 100000 = 2 * N1 \Rightarrow T2 = 59.582 \approx 4 * T1 = 2^2 * T1.$$

On relève que :

$$T(n) \leq P(n) \text{ tel que } Cte^2 * n \leq n^2 < Cte^2 * n < n^2 \text{ avec } Cte=2 \text{ et } n \gg \gg \gg$$

→CONCLUSION :

Les données obtenues sont majorées par les données du pire cas Et donc :

" 😊 Le modèle théorique est conforme avec les données expérimentales 😊 "

→PS : Nous ne pouvant pas généraliser car les tests faits n'englobent pas toutes les valeurs possibles 😊.

2.3 Algorithme 3 : Tri par fusion

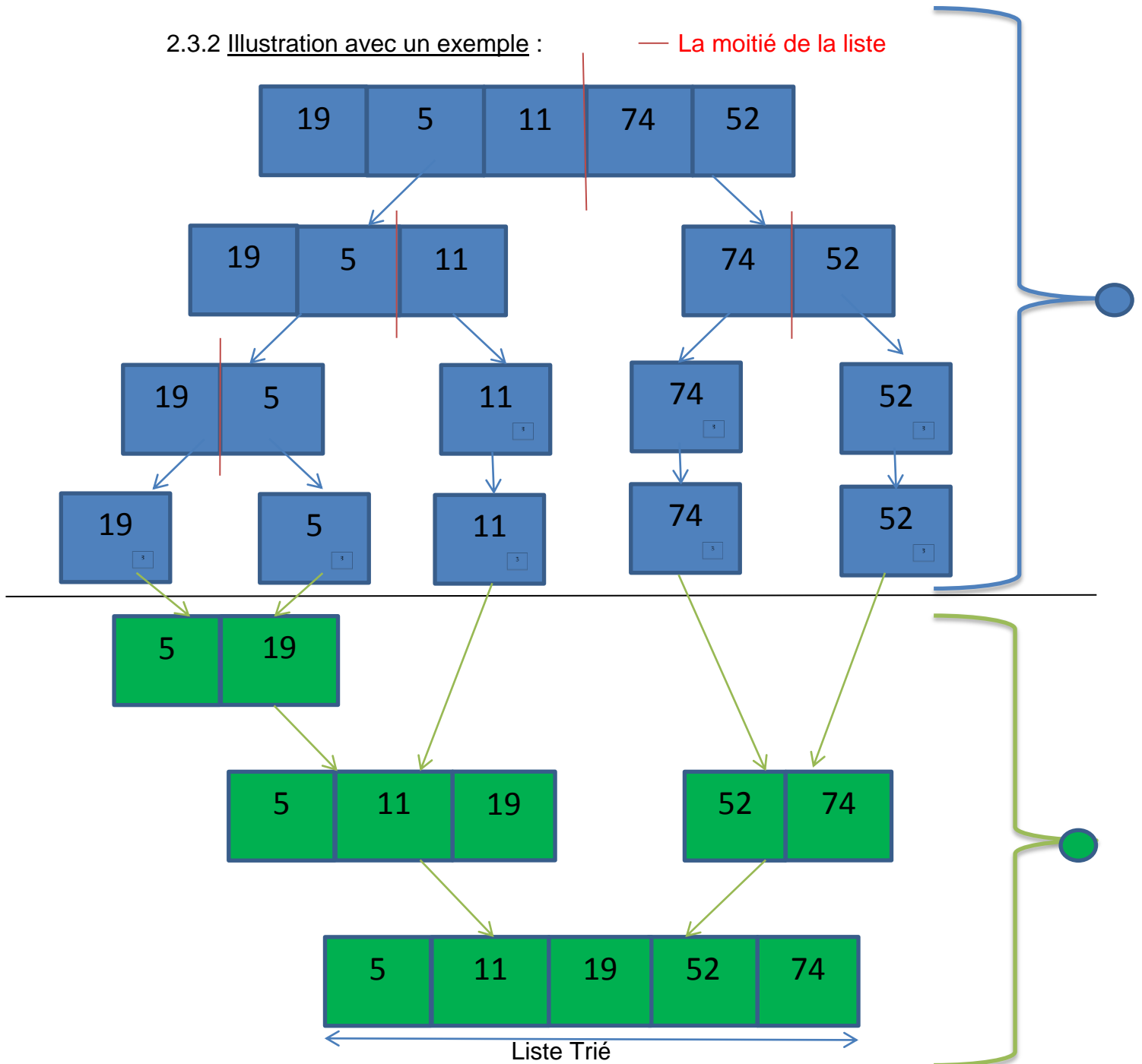
2.3.1 Description de la méthode de tri :

Ce tri est basé sur la technique algorithmique "diviser pour régner". L'opération principale de l'algorithme est la *fusion*, qui consiste à réunir deux listes triées en une seule. L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être. Dans cette méthode de tri, la liste non triée est divisée en N sous-listes, chacune ayant un élément, ensuite, il fusionne à plusieurs reprises ces sous-listes, pour obtenir des sous-listes triées, et à la fin on obtient une liste bien triée.

La fusion est considérée comme suit :

- Division de la liste (éléments du tableau) récursivement en deux moitiés jusqu'à ce qu'elle ne puisse plus être divisible.
- Fusion des listes les plus petites dans la nouvelle liste dans l'ordre croissant.

2.3.2 Illustration avec un exemple :



2.3.3 Ecriture de l'algorithme en pseudo langage :

➤ Récuratif :

Procedure Tri_Fusion (Liste []:d` entier ; debut , fin: entier)

VAR

i, j, milieu : entier

DEBUT

```
    SI( debut >= fin ) ALORS
        milieu = ( debut + fin ) / 2;
        Tri_Fusion (Liste ,debut , milieu );
        Tri_Fusion (Liste , milieu +1, fin);
        Fusion (debut ,milieu , milieu +1, fin );
    FinSi ;
```

FIN.

Procedure Fusion (Liste []:d` entier ; debut1 , fin1 , debut2 , fin2 : entier)

VAR

i, j, k : entier

DEBUT

```
    i = debut1 ;
    j = debut2 ;
    k = 0;

    Tantque (i <= fin1 et j <= fin2 )
        Faire
            SI( Liste [i] < Liste [j]) ALORS
                ListTmp [k] = Liste [i];
                i = i+1;
                k = k+1;

                SINON
                    ListTmp [k] = Liste [j];
                    j = j+1;
                    k = k+1;
            FinSi ;
        Fait ;
```

```
    Tantque (i <= fin1 )
```

```
        Faire
```

```
            [k] = Liste [i];
```

```
            i = i+1;
```

```
            k = k+1;
```

```
        Fait ;
```

```
    Tantque (j <= fin2 )
```

```
        Faire
```

```
            ListTmp [k] = Liste [j];
```

```
            j = j+1;
```

```
            k = k+1;
```

```
        Fait ;
```

```
    Pour j = 0 a k
```

```
        Faire
```

```
            Liste [first +j] = ListTmp [j];
```

```
        Fait ;
```

FIN.

2.3.4 Calcul de la complexité Temporelle :

>> **Pire** et **Meilleur** cas → c'est la même complexité, tel que :

$$CT(n) = \begin{cases} (2 * CT(n/2) + 2*n) & \text{si } n \leq 2 \\ 1 & \text{sinon} \end{cases}$$



$$CT(2^n) = \begin{cases} (2 * CT(2^n/2) + 2*2^n) & \text{si } n \leq 2^n \\ 1 & \text{sinon} \end{cases} \Rightarrow \begin{cases} 2 * CT(2^{n-1}) + 2*2^n & \text{si } n \leq 2^n \\ 1 & \text{sinon} \end{cases}$$

PREUVE PAR RECURENCE QUE $CT(2^n) = n * 2^{n+1}$ RESTE VRAIE POUR $n=n+1$

$$\begin{aligned} CT(2^n) &= 2 * 2^n + 2 * CT(2^{n-1}) \\ &= 2 * 2^n + 2 * (n-1) * 2^n \\ &= 2 * 2^n + (n-1) * 2^{n+1} \\ &= 2^{n+1} + (n-1) * 2^{n+1} \end{aligned}$$

$$\boxed{CT(2^n) = n * 2^{n+1}}$$

$CT(2^n) = \log_2(2^n) * 2^{n+1} \rightarrow$ on sait que : $CT(n) < CT(2^{\log_2(n)})$
 $CT(n) = \log_2(n) * n + 2 = O(n * \ln(n)).$

ET donc la complexité au pire et au meilleur cas : $\boxed{CT(n) = O(n * \ln(n))}$.

2.3.5 Calcul de la complexité Spatiale :

L'algorithme de tri par fusion utilise :

- Un tableau d'entier en question (de taille n) à trier.
- 6 variables de type entier.

En considérant que la taille d'un entier est sur 4 Octets. Nous obtenant la complexité spatiale : $\rightarrow CS(n) = 4(n + 6) \text{ octets} = 4n+24 \rightarrow CS(n) = O(n)$.

2.3.6 Ecriture du programme en langage C :

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

void merge ( long long int tab [], long long int first1 , long long int last1 , long long int first2 ,
long long int last2 )
{
    long long int i = first1 ;
    long long int j = first2 ;
    long long int k = 0;
```

```

while (i <= last1 && j <= last2 )
{
    if (tab [i] < tab [j])
    {
        Tmp [k] = tab [i];
        i++;
        k++;
    }
    else
    {
        Tmp [k] = tab [j];
        k++;
        j++;
    }
}

while (i <= last1 )
{
    Tmp [k] = tab [i];
    k++;
    i++;
}

while (j <= last2 )
{
    Tmp [k] = tab [j];
    k++;
    j++;
}

for (j=0; j<k; j++)
    tab [ first1 +j] = Tmp[j];
}

void MergeSort ( long long int tab [], long long int first , long long int last )
{
    long long int middle ,i,j;
    if (first >= last ) return ;
    middle = ( first + last ) /2;
    MergeSort (tab ,first , middle );
    MergeSort (tab , middle +1, last );
    merge (tab ,first ,middle , middle +1, last );
}

```

2.3.7 Mesure des temps d'exécution :

On a vu précédemment que l'obtention du temps d'exécution sera pour les trois cas trois (3) suivants :

1. Les données du tableau sont dans l'ordre.
2. Les données du tableau sont dans un ordre aléatoire.

3. Données du tableau dans l'ordre inverse.

Taille du Tableau (n)	50000	100000	200000	400000	800000	1600000
Bon Ordre	0.01	0.02	0.042	0.087	0.18	0.489
Ordre Aléatoire	0.028	0.041	0.069	0.119	0.25	0.569
Ordre Inverse	0.008	0.0108	0.0401	0.101	0.201	0.346
Taille du Tableau (n)	3200000	6400000	12800000	25600000	51200000	/
Bon Ordre	1.276	2.456	4.045	7.009	13.278	/
Ordre Aléatoire	1.123	2.389	5.398	10.061	26.354	/
Ordre Inverse	0.829	1.589	3.489	7.009	15.389	/

2.3.8 Représentation Graphique (à la fin de la partie 1)

2.3.9 Comparaison de la complexité théorique avec la complexité expérimentale

Que ce soit dans l'Ordre Inverse, l'Ordre Bon, Ou l'Ordre Aléatoire

On remarque que les temps d'exécution sont approximativement multipliés par 2 lorsque n est doublé.

Exemples :

$N1 = 50000 \Rightarrow T1 = 0.01$

$N2 = 100000 = 2 * N1 \Rightarrow T2 = 0.02 \approx 2 * T1$

On en déduit que le temps d'exécution est proportionnel à n , et nous pouvons le présenter par la formule suivante :

$T1(x * N) = x * T1(N)$ pour tous $x * N$ qui appartient a [50000 - 2048000000] (x étant la tangente d'un point sur le graphe).

→ CONCLUSION :

Et donc les données expérimentales de la complexité suivent la fonction $T1$.
Et la complexité théorique au pire et au meilleur cas est de l'ordre de :
 $M(n) = n * \log(n)$

Les données obtenues sont majorées par les données du pire cas Et donc :
" 😊 Le modèle théorique est conforme avec les données expérimentales 😊 "

→ PS : Nous ne pouvant pas généraliser car les tests faits n'englobent pas toutes les valeurs possibles 😊.

2.4 Algorithme 4

: Tri rapide

2.4.1 Description de la méthode de tri :

Cette méthode illustre le principe <<diviser pour régner>>, elle consiste à appliquer plusieurs partitionnements sur le tableau non trié.

Comment partitionner un tableau ! :

1 - Sélectionner au hasard un pivot.

Une fois le pivot sélectionné, on l'utilise pour partitionner les données en 2 ensembles :

- ❖ A gauche du pivot sont les nombres inférieurs au pivot.
- ❖ A droite du pivot sont les nombres supérieurs au pivot.

2 - On place 2 sélecteurs aux extrémités du tableau non trié.

3 -

- 1 ↑ ❖ Le 1^{er} sélecteur va se déplacer par avant (gauche → droite) jusqu'à ce qu'il rencontre le 1^{er} nombre plus grand que le pivot arbitrairement choisi.
- 2 ↑ ❖ Quand le 1^{er} sélecteur trouve une valeur qui dépasse le pivot, le 2^{ème} sélecteur, va entrer en action et se déplace par arrière (droite → gauche) jusqu'à ce qu'il rencontre le 1^{er} nombre plus petit que le pivot choisi.

4 - Les sélecteurs ont identifié des valeurs qui encadrent le pivot, la procédure de tri va inter changer (SWAP) ces 2 valeurs, la plus petite prend la place de la plus grande, et la plus grande occupera la position de la petite.

Puis on répète la procédure jusqu'à ce que les deux sélecteurs se rencontrent à la même position, qui sera celle du pivot.

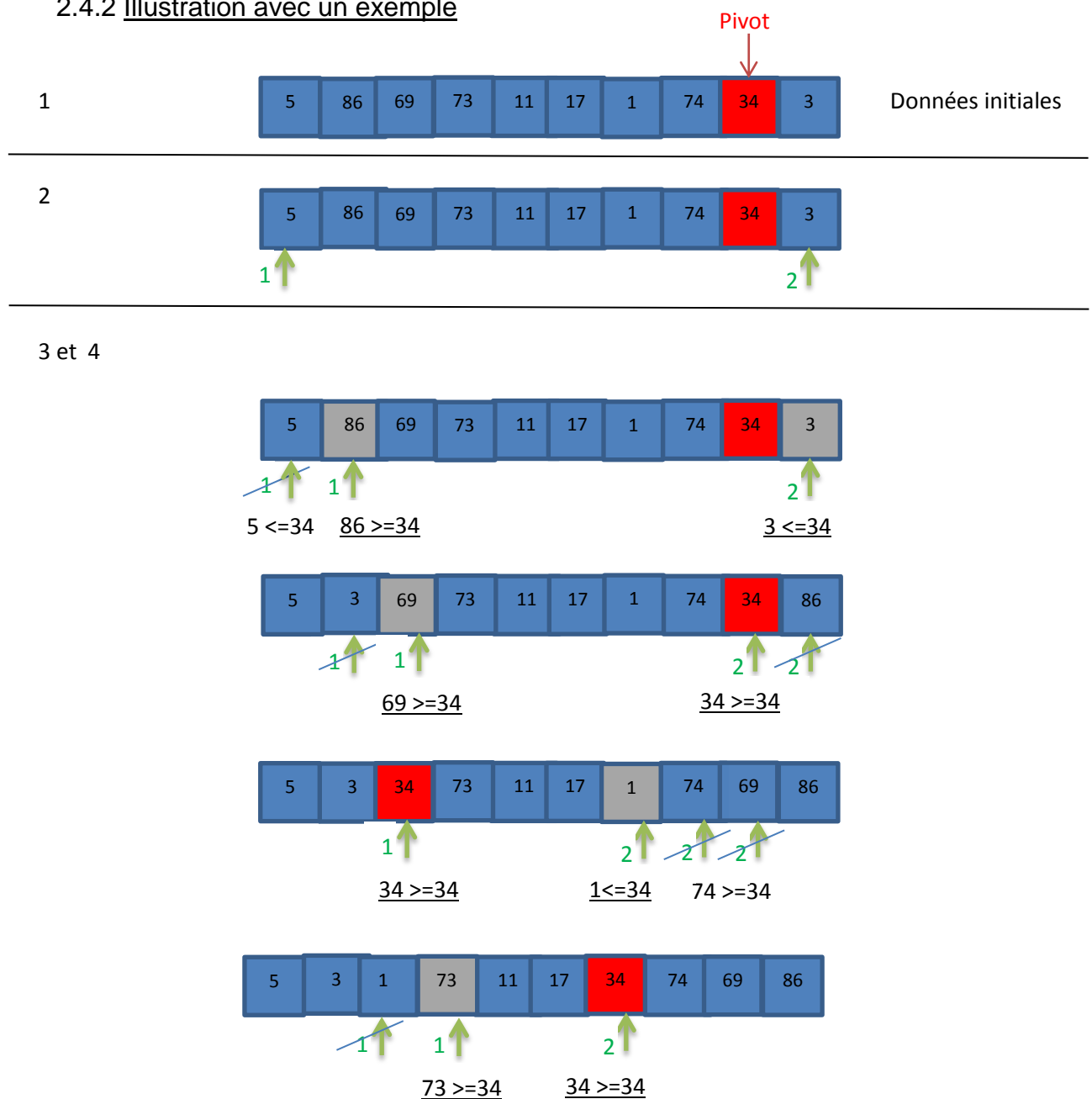




Tableau -Partitionné

$34 \geq 34$ $17 \leq 34$

-Non trié



Pivot : -Bien Placé

$11 \leq 34$

5

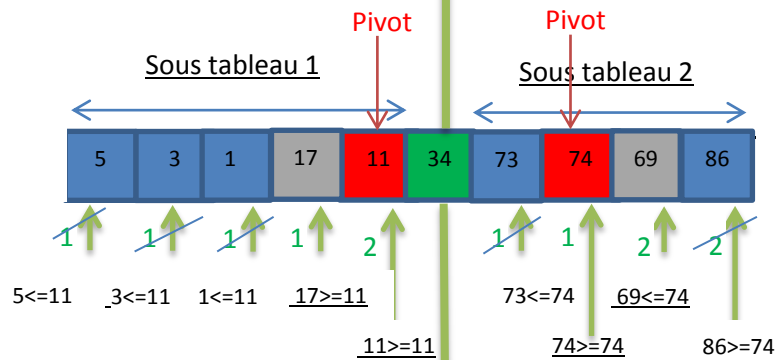
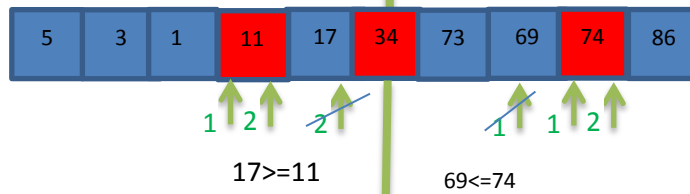


Tableau -Partitionné

-Non trié



Pivot : -Bien Placé

$17 \geq 11$

$69 \leq 74$

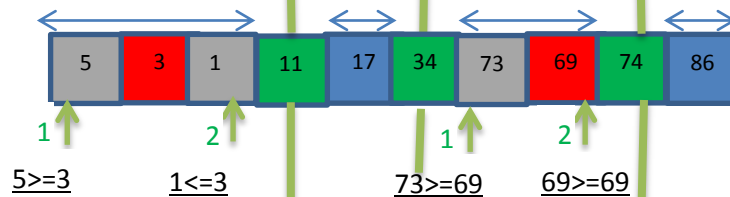
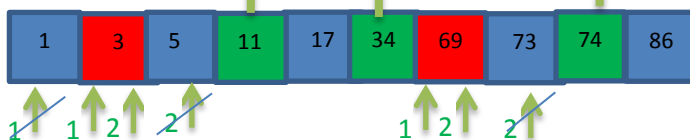


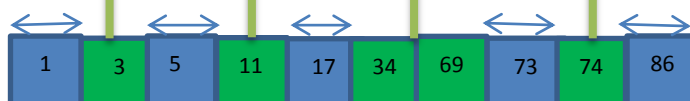
Tableau -Partitionné

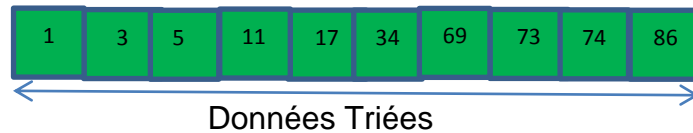
-Non trié



Pivot : -Bien Placé

Les sous Tableaux à un
élément sont déjà triés





2.4.3 Ecriture de l'algorithme en pseudo langage

➤ Itératif :

```

procedure permuter ( a* :entier , b* : entier )
VAR
temp : entier ;
DEBUT
    temp = *a;
    *a = *b;
    *b = temp ;
FIN.

fonction Partition ( data [1.. n]: entier , left :entier , right : entier ): entier
VAR
x,i,j: entier ;
DEBUT
    x = data [ right ];
    i = ( left - 1);
    POUR j de left a j right - 1 pas 1)
    FAIRE
        SI( data [j] <= x) ALORS
            ++i;
            permuter (& data [i], & data [j]);
        FIN SI;
    FAIT ;
    permuter (& data [i + 1], & data [ right ]);
    return (i + 1);
FIN.

Procedure Tri_Rapide_Iterative ( data [1.. n]: entier , count : entier )
VAR
startIndex , endIndex ,top: entier ;
stack * : entier ;
DEBUT
    startIndex = 0;
    endIndex = count - 1;
    top = -1;
    stack = ( entier *) Allouer ( Taille ( entier ) * count );
    stack [++ top] = startIndex ;
    stack [++ top] = endIndex ;
    TANT QUE (top >= 0)
    FAIRE
        endIndex = stack [top --];
        startIndex = stack [top --];
        p: entier ;
        p = Partition (data , startIndex , endIndex );
        SI(p - 1 > startIndex ) ALORS

```

```

        stack [++ top] = startIndex ;
        stack [++ top] = p - 1;
    FIN SI;

    SI(p + 1 < endIndex ) ALORS
        stack [++ top] = p + 1;
        stack [++ top] = endIndex ;
    FIN SI;

    FAIT ;
    Libérer ( stack );
FIN .

```

➤ Récurif :

```

procedure permuter ( tableau [] :entier ,a:entier , b: entier )
VAR
temp : entier ;
DEBUT
    temp = tableau [a];
    tableau [a] = tableau [b];
    tableau [b] = temp ;
FIN

procedure quickSort ( tableau []: entier , debut :entier , fin: entier )
VAR
gauche , droit : entier ;
pivot : entier ;
DEBUT
    gauche = debut -1;
    droite = fin +1;
    CONST pivot = tableau [ debut ];

    /* Si le tableau est de longueur nulle(tableau vide), il n'y a rien à faire . */
    SI( debut >= fin ) ALORS return ;
    FIN SI;

    /* Sinon , on parcourt le tableau , une fois de droite à gauche , et une autre fois de
    gauche à droite, à la recherche d'éléments mal placées, que l'on permute. Si les deux
    parcours se croisent, on s'arrête . */

    TANT QUE (1)
    FAIRE
        REPETER
            droite = droit - 1;
        JUSQU'À ( tableau [ droite ] > pivot );

        REPETER
            gauche = gauche + 1;
        JUSQU'À ( tableau [ gauche ] < pivot );

        SI( gauche < droite ) ALORS
            permuter ( tableau , gauche , droite );
            SINON break ;
    FIN FAIRE

```

```

        FIN SI;
    FAIT ;

    /* Maintenant, tous les éléments inférieurs au pivot sont avant ceux supérieurs au
    pivot . On a donc deux groupes de cases à trié, On utilise pour cela l'appel de la
    méthode quickSort ! */

    quickSort ( tableau , debut , droite );
    quickSort ( tableau , droite +1, fin);

    FIN .

```

2.4.4 Calcule de la complexité Temporelle

>> Meilleur cas :

S'il s'agit des tableaux dont le pivot est proche de la médiane)

Nous devons d'abord remarquer qu'à chaque étape du processus, nous doublons le nombre de sous-tableaux comme la montre l'illustration ci-dessus, . mais que la taille de chacun de ces sous-tableaux est divisée en deux. Donc, au final, chaque étape demandera N opérations.

Il nous faut maintenant compter le nombre d'étapes nécessaires pour trier un tableau. Pour cela, Le tri est bien évidemment terminé lorsque nous atteignons des sous-tableaux de taille 1 (les sous tableaux à 1 élément).

Donc : il y a autant d'étapes que de divisions par 2 nécessaires pour passer de n à 1.

La fonction qui nous donne «combien de fois diviser n par 2 avant d'arriver à 1 est la fonction logarithme de base 2 : "Log₂". → 2¹, 2², 2³, 2⁴2ⁿ

Du coup, le tri rapide a une complexité d'ordre: $O(N * \log_2(N))$.

>> Pire cas :

- Si le tableau est trié dans l'ordre inverse avec choix du 1^{er} élément du tableau comme pivot c.-à-d. l'élément le plus grand) :

Dans ce cas tous le tableau sera parcouru pour accéder au n^{ième} élément (le plus petit) afin de les permuter puis, ainsi le n^{ième} élément (le plus grand) sera bien placé, on vérifie pour le 1er élément (plus petit) qui se positionne aussi à sa place donc pas de permutation.

On se retrouve dans la situation : le (n-1)^{ième} élément (2eme élément le plus grand du tableau), a été permuté avec le 2eme...

Puis, comme toutes les autres valeurs sont inférieures au (n-1)^{ième} élément, nos deux recherches d'éléments mal placés se sont croisées, et on n'a rien permuter de plus.

De même, après cette étape, un autre passage déterminera que le 2eme est bien placé et qu'on peut l'ignorer après avoir comme même parcouru tous le tableau.

Ce schéma se répètera ainsi jusqu'à ce que tout le tableau soit trié. On se rend compte qu'à chaque _étape, on place correctement une valeur, pour ce faire, on parcourt tout l'ensemble tableau à trier.

Ainsi, le premier passage fait n tests, le second n-1, et ainsi de suite. Au final, on aura effectué → $n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1$ tests, et n permutations.

En calculant la somme $n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1$,

$$CT(n) = \sum_{i=1}^n i = 1 + 2 + 3 + 4 + \dots + n = (n(n+1)/2) = ((n^2 + n)/2)$$

Et donc la complexité au pire cas est quadratique c.-à-d. de l'ordre $n^2 \rightarrow O(n^2)$.

2.4.5 Calcul de la complexité Spatiale :

L'algorithme du tri rapide utilise :

- Un tableau d'entier à N éléments.
- 3 variables de type entier.

en considérant que la taille d'un entier est sur 4 Octets, nous obtenant la complexité spéciale : $CS(n) = 4(n + 3)\text{Octets} = 4n + 12 \text{ octets} \rightarrow CS(n) = O(n)$.

2.4.6 Ecriture du programme en langage C

```
# include < stdio .h>
# include < stdlib .h>
# include < time .h>
void permuter (int tableau [], int a, int b)
{
    int temp = tableau [a];
    tableau [a] = tableau [b];
    tableau [b] = temp ;
}

void quickSort (int tableau [], int debut , int fin )
{
    int gauche = debut -1;
    int droite = fin +1;
    const int pivot = tableau [ debut ];

    if( debut >= fin ) return ;

    while (1)
    {
        do droite --; while ( tableau [ droite ] > pivot );
        do gauche ++; while ( tableau [ gauche ] < pivot );

        if( gauche < droite ) permuter ( tableau , gauche , droite );
        else break ;
    }

    quickSort ( tableau , debut , droite );
    quickSort ( tableau , droite +1, fin);
}
```

2.4.7 Mesure des temps d'exécution

On a vu précédemment que l'obtention du temps d'exécution sera pour les trois cas trois (3) suivants :

1. Les données du tableau sont dans l'ordre.
2. Les données du tableau sont dans un ordre aléatoirement.
3. Données du tableau dans l'ordre inverse.

Taille du Tableau (n)	50000	100000	200000	400000	800000	1600000
Bon Ordre	14.059	58.986	242.596	960.603	3862.209	15376.899
Ordre Aléatoire	0.013	0.03	0.061	0.129	0.319	1.99
Ordre Inverse	9.938	36.974	153.987	625.932	2525.934	10450.592
Taille du Tableau (n)	3200000	6400000	12800000	25600000	51200000	/
Bon Ordre	61518.502	243616.821	979345.348	3917385.021	15638212.99	/
Ordre Aléatoire	8.91	30.998	123.924	455.999	1760.924	/
Ordre Inverse	43895.943	182171.509	769659.995	3087949.934	12870579.93	/

2.4.8 Représentation Graphique (à la partie 1)

2.4.9 Comparaison de la complexité théorique avec la complexité expérimentale

- Dans l'Ordre Inverse et l'Ordre Bon :

On remarque que les temps d'exécution sont approximativement multipliés par 4 lorsque n est doublé.

Exemples :

$$N1 = 50000 \rightarrow T1 = 9.938$$

$$N2 = 100000 = 2 * N1 \rightarrow T2 = 36.974 = 4 * T1 = 2^2 * T1$$

Aussi

$$N1 = 400000 \rightarrow T1 = 625.932$$

$$N2 = 800000 = 2 * N1 \rightarrow T2 = 2525.934 = 4 * T1 = 2^2 * T1$$

On en déduit que le temps d'exécution est proportionnel à N , ce que l'on peut représenter par la formule suivante :

$$T1(x * N) = x^2 * T1(N) \text{ pour tous } x * N \text{ qui appartient à } [50000 - 2048000000]$$

(x étant la tangente d'un point sur le graphe).

- Dans l'Ordre aléatoire :

Puis l'on peut aussi constater que lorsque la taille n est doublée, le temps d'exécution aussi à peu près doublé.

Exemples :

$$N1 = 50000 \rightarrow T1 = 0.0130$$

$$N2 = 100000 = 2 * N1 \rightarrow T2 = 0.03 = 2 * T1$$

Aussi

$$N1 = 400000 \rightarrow T1 = 0.129$$

$$N2 = 800000 = 2 * N1 \rightarrow T2 = 0.319 = 2 * T1$$

On en déduit que le temps d'exécution est proportionnel à N , ce que l'on peut représenter par la formule suivante :

$$T2(x * N) = x * T2(N) \text{ pour tous } x * N \text{ qui appartient à } [50000 - 2048000000]$$

(x étant la tangente d'un point sur le graphe).

Ainsi les données expérimentales de la complexité suivent une des deux fonctions $T1$ et $T2$.

Et la complexité théorique au pire cas est de l'ordre de : $P(n) = n^2$

Et la complexité théorique au meilleur cas est de l'ordre de : $M(n) = n * \log(n)$

On relève que :

$$M(n) < T1(n) < P(n) \text{ tel que } n * \log(n) < Cte^2 * n < n^2 \text{ avec } Cte = 2 \text{ et } n \gg \gg \gg$$

et

$$M(n) < T2(n) < P(n) \text{ tel que } n * \log(n) < Cte * n < n^2 \text{ avec } Cte = 2 \text{ et } n \gg \gg \gg$$

→ CONCLUSION :

Les données obtenues sont bornées (majorées par les données du pire cas et minorées par les données au meilleur cas). Et donc :

" ☺ Le modèle théorique est conforme avec les données expérimentales ☺ "

→PS : Nous ne pouvant pas généraliser car les tests faits n'englobent pas toutes les valeurs possibles ☹.

2.5 Algorithme 5

: Tri par tas

2.5.1 Description de la méthode de tri :



Dans cet algorithme on va d'abord :

- 1- Construire un arbre en utilisant les éléments du tableau.
- 2- Créer le Tas en Permutant les valeurs des nœuds jusqu'à avoir le nœud parent (racine) supérieure à tous les autres nœuds fils, (les permutations sont faites ainsi dans le tableau).
- 3- Une fois cela est fait, Permuter le premier nœud (racine) avec le dernier nœud dans le tas et dans le tableau, puis on ne touche plus à ce dernier élément car il est trié à ce stade-là.
- 4- Supprime le dernier nœud du tas.
- 5- Répéter le processus (2→4) avec le nouvel arbre obtenu jusqu'à ce qu'un seul élément reste dans le tas.

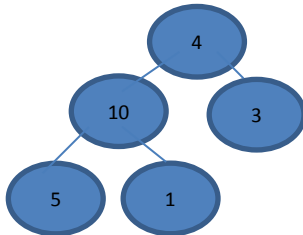
2.5.2 Illustration avec un exemple :

Données initiales

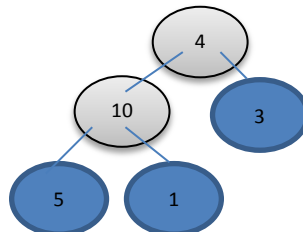
4	10	3	5	1
---	----	---	---	---

 élément déjà trié
 éléments à permuter

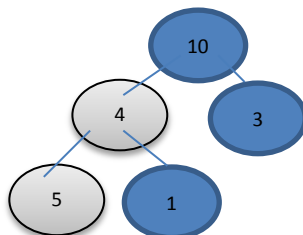
1



2

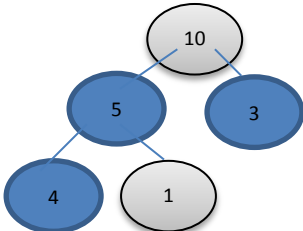


4	10	3	5	1
---	----	---	---	---



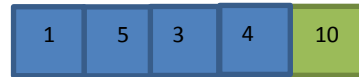
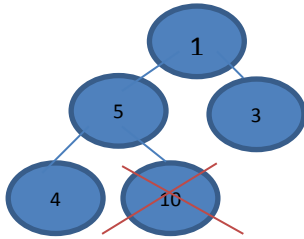
10	4	3	5	1
----	---	---	---	---

3

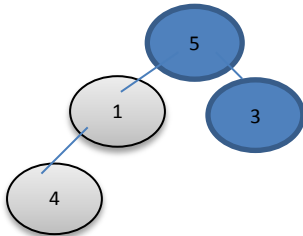
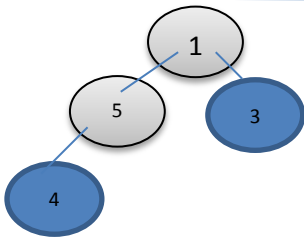


10	5	3	4	1
----	---	---	---	---

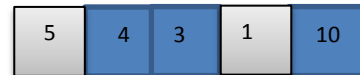
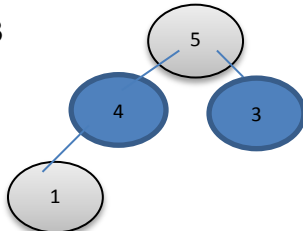
4



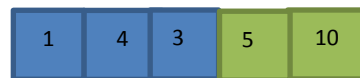
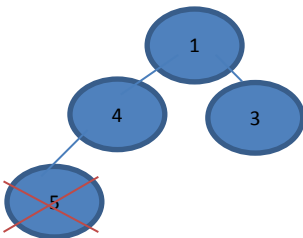
5-2



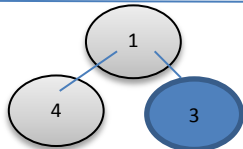
5-3



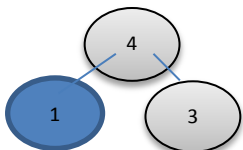
5-4



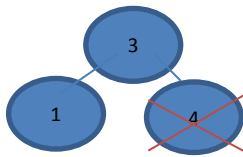
5-2



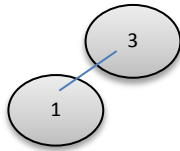
5-3



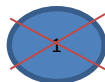
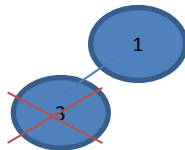
5-4



5-3



5-4



2.5.3 Ecriture de l'algorithme en pseudo langage

➤ Itératif :

Procédure tri_tas (arbre []:d'entier , longueur : entier)

VAR

i : entier ;

DEBUT

 Pour i := longueur / 2 a 1

 Faire

 tamiser (arbre , i, longueur) ;

 Fait ;

 Pour i := longueur a 2

 Faire

 permuter (arbre [1] , arbre [i]);

 tamiser (arbre , 1, i -1) ;

 Fait ;

FIN.

Procédure tamiser (arbre , noeud , n)

VAR

k := noeud

j := 2*k

DEBUT

 Tant que (j <= n

 SI (j < n et arbre [j] < arbre [j +1]) ALORS

 j := j+1;

 FinSi ;

```

    SI ( arbre [k] < arbre [j]) ALORS
        permuter ( arbre [k], arbre [j]);
        k := j;
        j := 2*k;
    SINON
        j := n+1
    FinSi ;
Fait ;
FIN.

```

2.5.4 Calcule de la complexité Temporelle

>> Au **pire cas** et **Meilleur cas** il s'agit de la même complexité, tel que

Dans cet algorithme nous faisons appel à la fonction tamiser : **n fois**.

Et à l'intérieur de la fonction tamiser le parcours du tableau (qui est à présent sous forme d'un arbre), se fait deux _éléments à la fois, puis pour chaque élément aussi 2 fils, ...ect. ce qui donne une complexité de l'ordre **log(n)**.

Et donc la complexité est de l'ordre **n*log(n)**. → **CT(n) = O(n log n)**

Le nombre de permutations dans le tas est majoré par le nombre de comparaisons et il est du même ordre de grandeur.

La complexité au pire en nombre de transferts du tri par tas est donc en **O(n log n)**.

2.5.5 Calcule de la complexité Spatiale

Dans la procédure Tri_Tas nous avons utilisé:

- Un tableau de taille n.
- La longueur n qui est de type entier.
- Une variable i de type entier.

Dans la fonction tamisser nous avons utilisé:

- Un tableau de taille n
- Un nœud de type entier.
- la taille du tableau qui est un entier.
- Deux variables j et k de type entier.

En considérant que la taille d'un entier est sur 4 Octets, nous obtenant :

CS(n) = 4(2n + 6)Octets = 8n + 24 octets → CS(n) = O(n).

2.5.6 Ecriture du programme en langage C

```

void makeheap ( long long int x[ ], long long int n ) //entasser ou tamisser
{
    long long int i, val , s, f ;

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

```

```

    val = x[i] ;
    s = i ;
    f = ( s - 1 ) / 2 ;
    while ( s > 0 && x[f] < val )
    {
        x[s] = x[f] ;
        s = f ;
        f = ( s - 1 ) / 2 ;
    }
    x[s] = val ;
}

void heapsort ( long long int x[ ], long long int n )
{
    long long int i, s, f, ivalue ;

    for ( i = n - 1 ; i > 0 ; i -- )
    {
        ivalue = x[i] ;
        x[i] = x[0] ;
        f = 0 ;

        if ( i == 1 ) s = -1 ;
        else s = 1 ;
        if ( i > 2 && x[2] > x[1] ) s = 2 ;
        while ( s >= 0 && ivalue < x[s] )
        {
            x[f] = x[s] ;
            f = s ;
            s = 2 * f + 1 ;
            if ( s + 1 <= i - 1 && x[s] < x[s + 1] ) s++ ;
            if ( s > i - 1 ) s = -1 ;
        }
        x[f] = ivalue ;
    }
}

```

2.5.7 Mesure des temps d'exécution

On a vu précédemment que l'obtention du temps d'exécution sera pour les trois cas trois (3) suivants :

1. Les données du tableau sont dans l'ordre.
2. Les données du tableau sont dans un ordre aléatoirement.
3. Données du tableau dans l'ordre inverse.

Taille du Tableau (n)	50000	100000	200000	400000	800000	1600000
Bon Ordre	0.000404	0.00092	0.001092	0.004455	0.008391	0.162938
Ordre Aléatoire	0.008498	0.010937	0.020995	0.044998	0.044984	0.091298
Ordre Inverse	0.012935	0.029839	0.055379	0.109373	0.249837	4.996609
Taille du Tableau (n)	3200000	6400000	12800000	25600000	51200000	/
Bon Ordre	0.329787	0.6593638	1.295228	2.29921	3.592823	/
Ordre Aléatoire	2.3996352	5.3429483	10.9431497	21.929619	45.97493	/
Ordre Inverse	11.069936	23.995368	49.991254	95.999215	202.991522	/

2.5.8 Représentation Graphique (à la fin de la partie 1)

2.5.9 Comparaison de la complexité théorique avec la complexité expérimentale

Que ce soit dans l'Ordre Inverse, l'Ordre Bon, Ou l'Ordre Aléatoire :

- Nous remarquons que les temps d'exécution évoluent de manière logarithmique, c.-à-d. quand le n est doublé, le temps d'exécution est doublé aussi. Le temps d'exécution est donc proportionnel à N, et nous pouvons le représenter par la formule suivante :

$T(x * N) = x * T(N) \text{ pour tous } x * N \text{ qui appartient à } [50000 - 2048000000]$ <p>(x étant la tangente d'un point sur le graphe).</p>
--

Exemple :

$N1 = 50000 \rightarrow T1 = 0.000404$

$N2 = 100000 = 2 * N1 \rightarrow T2 = 0.00092 = 2 * T1.$

la complexité est toujours quadratique et la formule citée plus haut est toujours valide. On relève que :

$T(n) = P(n)$ tel que $Cte * n = n * \log(n)$ avec $Cte = 2$ et $n \gg \gg \gg$

→ CONCLUSION :

Les données obtenues sont proche des données de la complexité théorique)

Et donc :

" 😊 Le modèle théorique est conforme avec les données expérimentales 😊 "

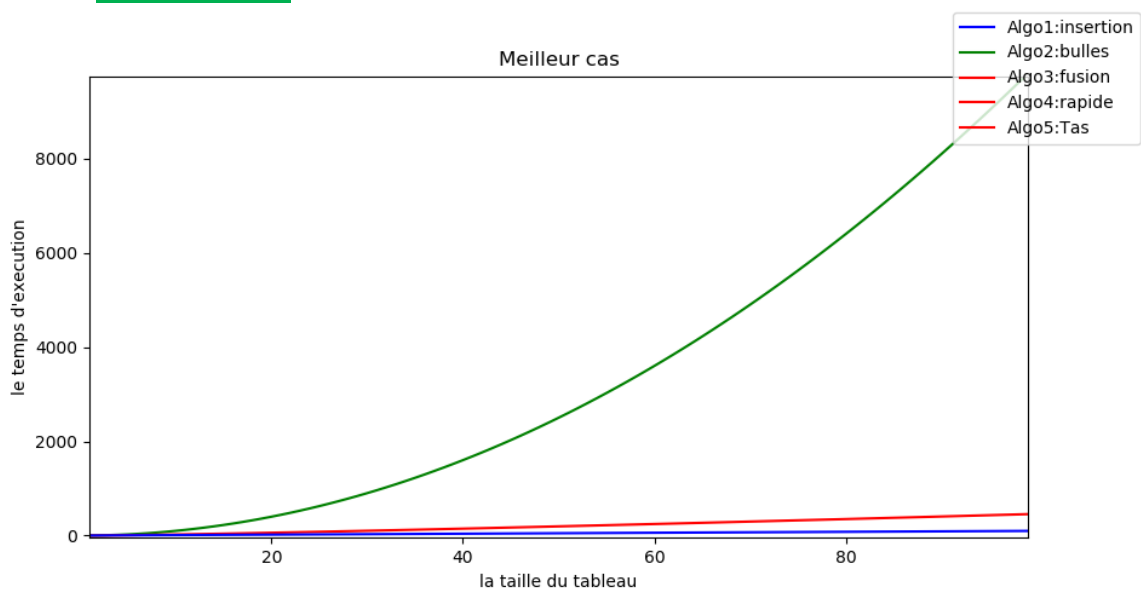
→ PS : Nous ne pouvant pas généraliser car les tests faits n'englobent pas toutes les valeurs possibles 😊.

3 Partie II : Comparaison des Algorithmes

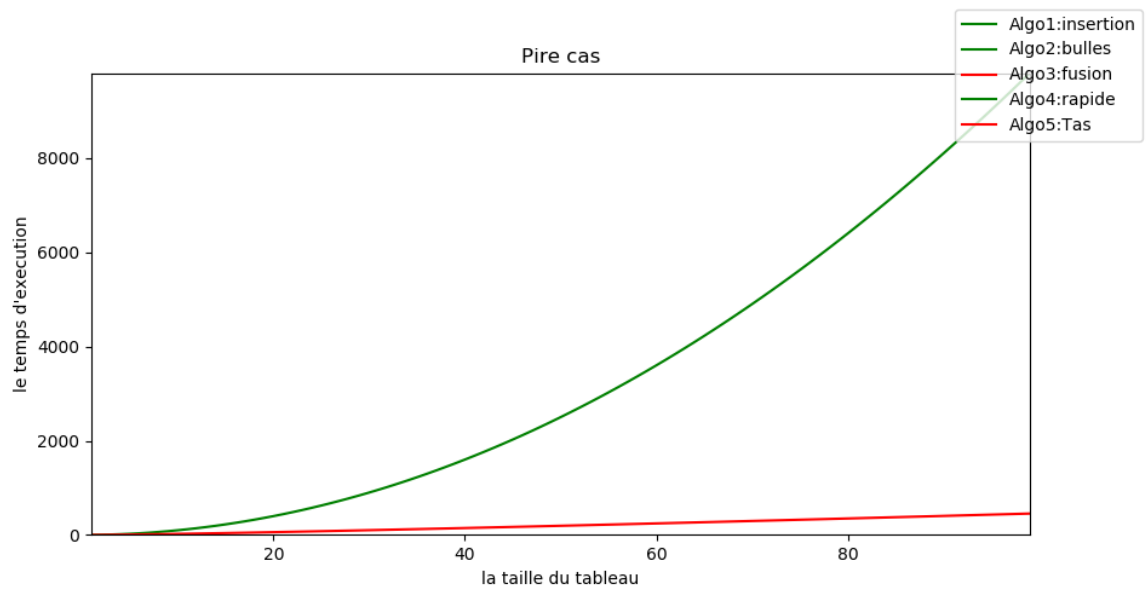
3.1 Représentation des graphes des complexités des 5 algorithmes

3.1.1 Complexité théorique :

➤ BON ORDRE

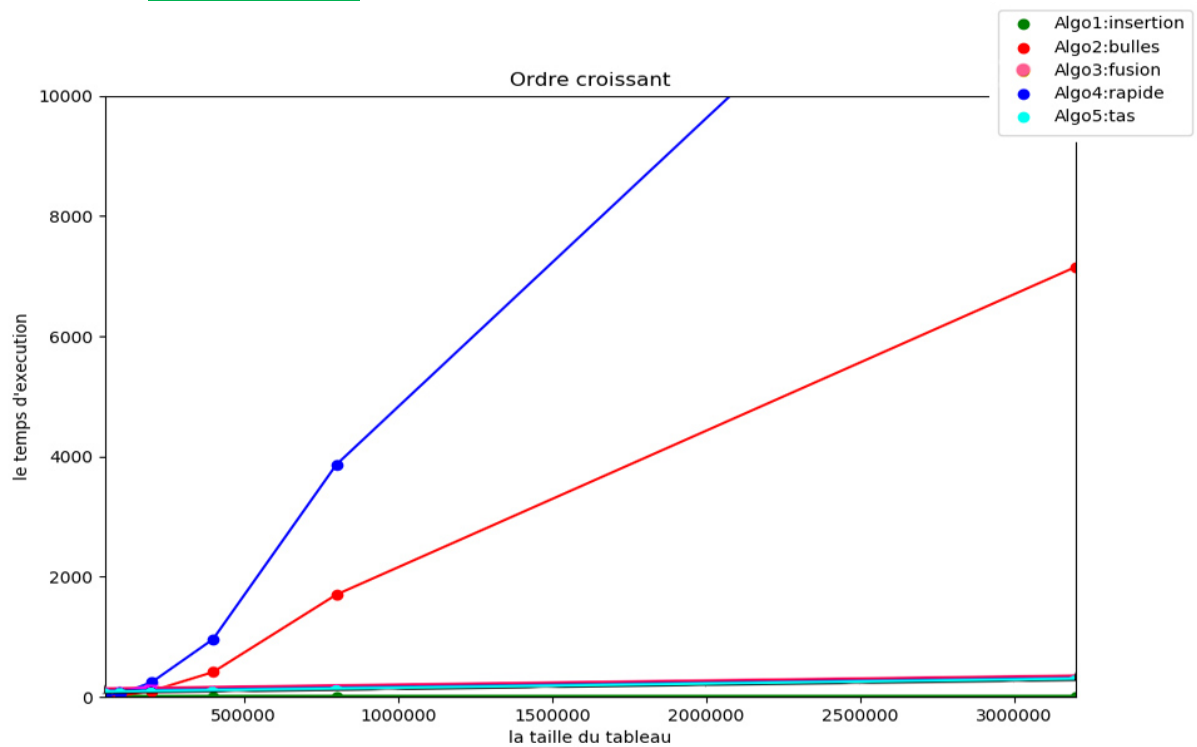


➤ ORDRE INVERSE :

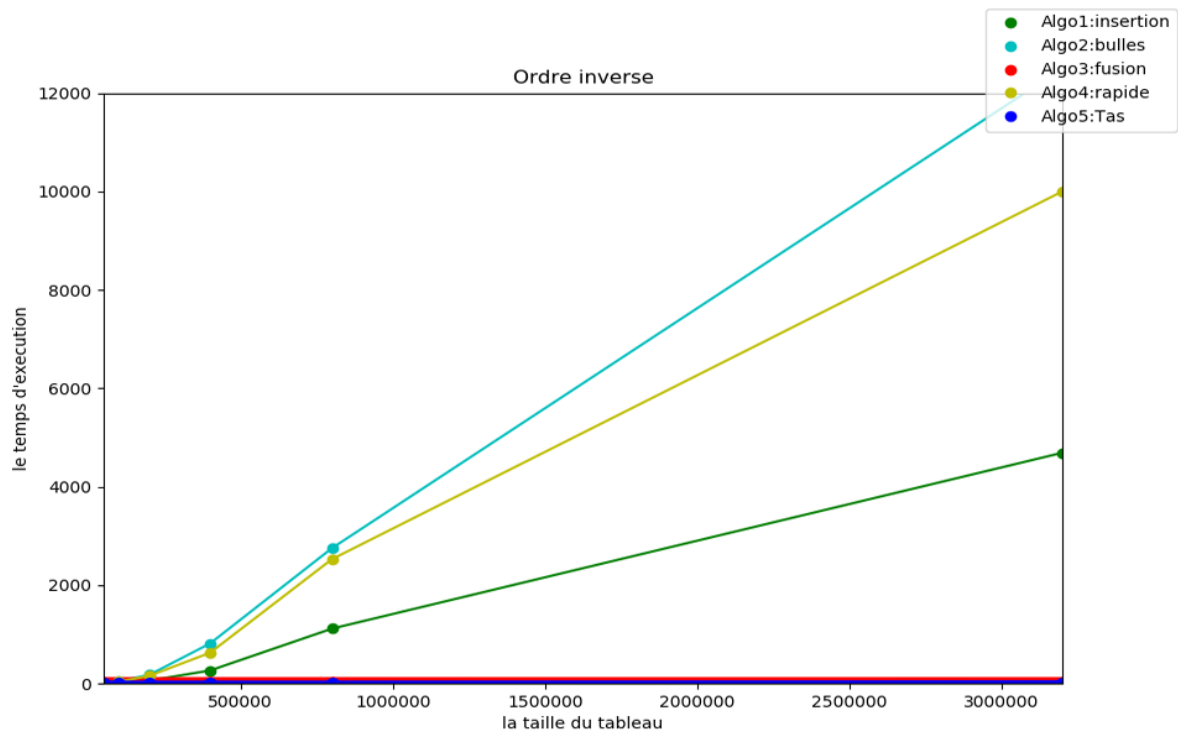


3.1.2 Complexité expérimentale

➤ BON ORDRE :



➤ ORDRE INVERSE :



3.2 Représentation des tableaux comparatifs des complexités et des temps d'exécution des 5 algorithmes

3.2.1 Complexité théorique

L'Algorithme	Meilleur cas	Pire cas
Tri par Insertion	$O(n)$	$O(n^2)$
Tri à bulles	$O(n^2)$	$O(n^2)$
Tri par Fusion	$O(n * \text{LOG}_2(n))$	$O(n * \text{LOG}_2(n))$
Tri Rapide	$O(n * \text{LOG}_2(n))$	$O(n^2)$
Tri Par Tas	$O(n * \text{LOG}_2(n))$	$O(n * \text{LOG}_2(n))$

3.2.2 Temps d'exécution

➤ Bon Ordre

Taille du Tableau (n)	50000	100000	200000	400000	800000	1600000
Tri Insertion	0.000338	0.000563	0.001089	0.001996	0.004169	0.008278
Tri A bulles	5.233	24.241	100.244	414.318	1702.147	7152.576
Tri Fusion	0.01	0.02	0.042	0.087	0.18	0.489
Tri Rapide	14.059	58.986	242.596	960.603	3862.209	15376.899
Tri Par Tas	0.000404	0.00092	0.001092	0.004455	0.008391	0.162938
Taille du Tableau (n)	3200000	6400000	12800000	25600000	51200000	/
Tri Insertion	0.0160176	0.033724	0.073385	0.168638	0.370869	
Tri A bulles	30115.912	120466.98	503555.528	2125007.906	7798781.981	
Tri Fusion	1.276	2.456	4.045	7.009	13.278	
Tri Rapide	61518.502	243616.82	979345.348	3917385.021	15638212.99	
Tri Par Tas	0.329787	0.6593638	1.295228	2.29921	3.592823	

➤ Ordre Inverse

Taille du Tableau (n)	50000	100000	200000	400000	800000	1600000
Tri Insertion	0.00211	0.00412	0.00908	0.019512	0.043121	0.086129
Tri A bulles	13.518	59.582	246.927	997.615	4124.499	16913.89
Tri Fusion	0.028	0.041	0.069	0.119	0.25	0.569
Tri Rapide	0.013	0.03	0.061	0.129	0.319	1.99
Tri Par Tas	0.008498	0.010937	0.020995	0.044998	0.044984	0.091298
Taille du Tableau (n)	3200000	6400000	12800000	25600000	51200000	/
Tri Insertion	0.168704	0.357526	0.757107	1.652573	3.79568	

Tri A bulles	69012.093	279502.37	1118012.795	4349072.984	17396695.27	
Tri Fusion	1.123	2.389	5.398	10.061	26.354	
Tri Rapide	8.91	30.998	123.924	455.999	1760.924	
Tri Par Tas	2.3996352	5.3429483	10.9431497	21.929619	45.97493	

➤ Ordre Aléatoire

Taille du Tableau (n)	50000	100000	200000	400000	800000	1600000
Tri Insertion	3.934113	17.15711	62.75211	267.0881	1116.756	4690.3761
Tri A bulles	9.681	50.425	176.982	815.631	2750.186	11967.03
Tri Fusion	0.008	0.0108	0.0401	0.101	0.201	0.346
Tri Rapide	9.938	36.974	153.987	625.932	2525.934	10450.592
Tri Par Tas	0.012935	0.029839	0.055379	0.109373	0.249837	4.996609
Taille du Tableau (n)	3200000	6400000	12800000	25600000	51200000	/
Tri Insertion	19934.098	86713.329	46853.3191	1508811.938	6789653.719	
Tri A bulles	47870.345	189092.75	869841.123	3192324.678	12769315.9	
Tri Fusion	0.829	1.589	3.489	7.009	15.389	
Tri Rapide	43895.943	182171.51	769659.995	3087949.934	12870579.93	
Tri Par Tas	11.069936	23.995368	49.991254	95.999215	202.991522	

3.3 Comparaison des Algorithmes décrits :

D'après les graphes et tableaux comparatifs nous pouvons maintenant comparer entre les 5 algorithmes de tri étudiés dans la partie | :

Nous remarquons que les méthodes de tri se différencient du coup chaque algorithme a des avantages et des inconvénients classifiés généralement selon :

1. La simplicité d'implémentation de l'algorithme :

Pour la simplicité d'implémentation il est clair que les algorithmes :

Tri à Insertion et Tri à bulles sont les plus involontaires selon leurs complexités et donc les plus simples à implémenter en les comparant aux trois autres tris restants qui sont un peu difficiles à implémenter.

2. Le temps d'exécution de l'algorithme.

En ce qui concerne la rapidité d'exécution, il est clair aussi que le tri par Insertion et tri à bulles ne sont pas très efficaces car leurs complexités temporelles très grandes.

Par contre le tri fusion vient en première place car est distinctement plus performant et donne des meilleurs résultats que les autres, La preuve que la plupart des SGBD tel que (ORACLE,... utilise le tri fusion pour les jointures entre tables.

Suivie du Tri par tas qui partage la 1^{ère} place aussi avec le tri par fusion ; qui est jusqu'au jour d'aujourd'hui considère comme le plus intéressant des algorithmes de tri , qui donne les meilleurs temps d'exécution !

Et enfin, le Tri rapide qui vient en deuxième place « comme son nom l'indique » est un tri assez rapide, son efficacité dépend du choix du pivot (c'est le problème principal de ce tri) , car si ce dernier est mal choisi on se retrouve avec un temps d'exécution identique à celui des deux premiers algorithmes (insertion, à bulles).

3. L'Espace mémoire occupé par l'algorithme.

Ceci n'est plus un problème de nos jours , car la plupart des machines disposent de suffisamment de stockage.

4 ANNEXE :

4.1 Fonctions et procédure communes à tous les programmes

4.1.1 Fonction de création et de calcul du temps d'exécution de la fonction de tri pour un tableau comportant des données en ordre croissant (Bon Ordre)

```
double sortedTime( long int n) {
long int * array = malloc (n* sizeof ( long int ));
long int i;
    for (i = 0; i < n; i++)
    {
        array [i] = i;
    }
    clock_t start = clock ();
    insertionSort (array , n) ; //remplacer par la fonction (bulles,fusion,rapide,tas)
    clock_t end = clock ();
    return ( double ) (end - start )/ CLOCKS_PER_SEC ;
}
```

4.1.2 Fonction de création et de calcul du temps d'exécution de la fonction de tri pour un tableau comportant des données en ordre décroissant (Ordre Inverse)

```
double inversedTime ( long int n) {  
long int * array = malloc (n* sizeof ( long int ));  
long int i;  
    for (i = 0; i < n; ++i)  
    {  
        array [i] = n - i;  
    }  
    clock_t start = clock ();  
    insertionSort (array , n); //remplacer par la fonction (bulles,fusion,rapide,tas)  
    clock_t end = clock ();  
    return ( double ) (end - start )/ CLOCKS_PER_SEC ;  
}
```

4.1.3 Fonction de création et de calcul du temps d'exécution de la fonction de tri pour un tableau comportant des données non triées (Ordre Aléatoire/ Quelconque)

```
double randomTime ( long int n) {  
long int * array = malloc (n* sizeof ( long int ));  
long int i;  
    for (i = 0; i < n; ++i)  
    {  
        array [i] = (int) rand ()%n;  
    }  
    clock_t start = clock ();  
    selectionSort (array , n); //remplacer par la fonction (bulles,fusion,rapide,tas)  
    clock_t end = clock ();  
    return ( double ) (end - start )/ CLOCKS_PER_SEC ;  
}
```

5 Matériel utilisé

PC UTILISE :	Lenovo-PC
PRECESSEUR :	INTEL(R) CELERON(R) CPU N2840 @2.16 GHz 2.16GHz
RAM :	4 Go
OS :	64 bits, PROCESSEUR *64

FIN

