Université IBN TOFAIL
****
Faculté des sciences
Département d'Informatique
****

STRUCTURES DE DONNEES

SMI – S4

Année universitaire : 2021/2022

Pr. Salma AZZOUZI

Table des matières

Chapitre 1 : Complexité des Algorithmes et Tri

I. Introduction

Jusqu'à présent la représentation d'un groupe de données de même nature (même type) n'est possible qu'au moyen des tableaux. Or leur facilité d'utilisation peut être fortement réduite lorsque :

a) La structure séquentielle d'un tableau ne reflète pas l'organisation globale des données comme par exemple :

   - Les liaisons ferroviaires/routières entre les différentes gares / villes d'une région ;
   - Tout arbre de calcul.

b) Le nombre maximal d'éléments du tableau n'est pas connu à la compilation (ne peut pas être fixé par le programmeur), à savoir :

   - Si ce nombre est choisi trop petit le programme ne pourra représenter et traiter toutes les données ;
   - S'il est pris (beaucoup) trop grand le gaspillage de mémoire peut pénaliser non seulement le fonctionnement du programme mais aussi celui du système informatique entier.

En effet un tableau peut difficilement refléter les successeurs ou prédécesseurs d'une donnée et les relations existantes entre certaines données. On a besoin d'autres structures plus souples pour manipuler les données. Ce cours présente plusieurs structures de données. Une structure de données est un moyen de stocker et organiser des données pour faciliter l'accès à ces données et leur modification : Les structures de données linéaires, structures arborescentes, graphes,…

Avant de détailler ces différentes structures dans les chapitres qui suivent, nous allons dans les sections de ce chapitre présenter les fondamentaux qui doivent vous guider pour concevoir et analyser des algorithmes. Il a pour objectif de rappeler et d'exposer certaines stratégies d'algorithmique qui nous serviront tout au long de ce cours : complexité des algorithmes, algorithmes de tri,….. Les parties suivantes de ce livre s'appuieront sur toutes ces fondations.

II. Algorithmique

Un algorithme est une procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs, et qui donne en sortie une valeur, ou un ensemble de valeurs. Un algorithme est donc une séquence d'opérations de calcul qui transforment l'entrée en sortie :

- Opérations exécutées en séquence ⇒ algorithme séquentiel
- Opérations exécutées en parallèle ⇒ algorithme parallèle
- Opérations exécutées sur un réseau de processeurs ⇒ algorithme réparti ou distribué

La mise en œuvre de l'un algorithme est l'écriture de ces opérations dans un langage de programmation donne un programme. Un programme (code) est la réalisation (l'implémentation)

d'un algorithme au moyen d'un langage donné sur une architecture donnée. Il s'agit de la mise en œuvre du principe

Il existe beaucoup de problèmes susceptibles d'être résolus par des algorithmes :

- **Chemin dans un graphe** : GPS, tout ce qui est acheminement (la Poste) ;
- **Flots** : affectations, emploi du temps, bagages (aéroport), portes d'embarquement ;
- **Compression/Décompression** : MP3, xvid, divx, codecs audio et vidéo, tar, zip ;
- **Internet** : Routage des informations, moteurs de recherche ;
- **Cryptage** : RSA (achat électronique) ;
- **Simulation** : Prévision météo, Réalité virtuelle ;
- **Ordonnancement** : fabrication de matériel dans les usines, construction automobile ;
- **Traitement d'images** (médecine, reconnaissance) , **effets spéciaux** (cinéma).

Tous les algorithmes ne sont pas équivalents, on les différencie selon 2 critères majeurs :

- **Temps de calcul :** lents vs rapides ;
- **Mémoire utilisée** : peu vs beaucoup.

On parle de complexité en temps (vitesse) ou en espace (mémoire utilisée). La théorie de la complexité étudie l'efficacité des algorithmes.

III. Complexité algorithmique

III.1. La théorie de la complexité

La théorie de la complexité est une branche de l'informatique théorique qui cherche à calculer, formellement, la complexité algorithmique nécessaire pour résoudre un problème au moyen de l'exécution d'un algorithme. Elle vise à répondre aux besoins d'efficacité des algorithmes (programmes et permet de :

- Classer les problèmes selon leur difficulté ;
- Classer les algorithmes selon leur efficacité ;
- Comparer les algorithmes résolvant un problème donné afin de faire un choix sans devoir les implémenter.

On s'intéresse dans ce cours, essentiellement, à l'efficacité en **termes de temps d'exécution.** Comme le temps exact de l'exécution d'un algorithme dépend de (1) la puissance de la machine et (2) la nature des données (les variables), si on change alors le type des données, le temps change.

Dans ce cours nous nous concentrons sur la complexité des algorithmes tenant **compte des unités de temps abstraites** proportionnelles au nombre d'opérations effectuées en suivant les règles suivantes :

- Chaque instruction basique consomme une unité de temps (affectation d'une variable, lecture, écriture, comparaison, …).
- Chaque itération d'une boucle rajoute le nombre d'unités de temps consommés dans le corps de cette boucle.

- Chaque appel de fonction rajoute le nombre d'unités de temps consommées dans cette fonction.
- Pour avoir le nombre d'opération effectuées par l'algorithme, on additionne toutes ces unités de temps consommées.

III.3 La complexité de l'algorithme

La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données. Elle est exprimée comme une fonction de la taille du jeu de données. Il existe trois types de complexité :

- **Complexité au meilleur des cas.** C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille n ;
- **Complexité au pire des cas.** C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille n ;
- **Complexité en moyenne**. C'est la moyenne des complexités de l'algorithme sur des jeux de données de taille n ;

On étudie ici, la complexité d'un algorithme qui mesure sa performance **asymptotique** dans **le pire des cas**.

- **Asymptotique.** On s'intéresse à des données très grandes parce que les petites valeurs ne sont pas assez informatives.
- **Pire des cas**. On s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre parce qu'on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé.

Un algorithme est dit « optimal » si sa complexité est la complexité minimale parmi les algorithmes de sa classe.

Quand on calcule la complexité d'un algorithme, on ne calcule généralement pas sa complexité exacte, mais **son ordre de grandeur**. Pour ce faire, on fait recours à **la notation asymptotique O(.).**

III.4. O-Notation

A. Définition

La O-notation (Notation de Landau) exprime la limite supérieure d'une fonction dans un facteur constant.

Définition :

Soit n la taille des données à traiter, on dit qu'une fonction f(n) et en O(g(n)) si :

$$\exists n_0 \in N,\ \exists k \in R,\ \forall n \geq n_0 \ : \quad |f(n)| \leq k\,|g(n)|$$

f(n) **et en** O(g(n))s'il existe un seuil à partir duquel la fonction f(.) est toujours dominée par g(.), à une constante multiplicative fixée près.

Si T(n) est la complexité temporelle d'un algorithme alors :

$$\text{T(n)} = O(f(n)) \quad \text{SSI} \quad \exists n_0 \in N,\ \exists k \in R,\ \forall n \geq n_0 \ : \quad T(n) \leq k \times f(n)$$

La figure 1 donne une représentation graphique montrant le comportement asymptotique de T(n) quand $n \to \infty$.

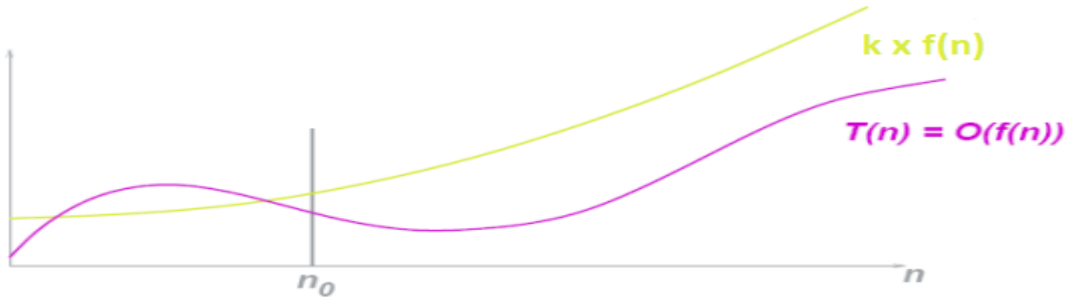


Figure 1 : Comportement asymptotique de T(n)

Exemple : On veut calculer la complexité $T(n) = k_1 \times n^2 + k_2 \times n$.

On remarque que :

Pour tout n>= 1, on a $k_1 \times n^2 + k_2 \times n \leq k_1 \times n^2 + k_2 \times n^2 = (k_1 + k_2) \times n^2$

Alors $T(n) \leq k \times n^2$ avec $k = k_1 + k_2$ et $n_0 = 1$

Donc $T(n)$ est en $O(n^2)$

B. *Algorithmes de complexité constante O(1)*

Soit l'algorithme suivant qui calcule la surface d'un cercle :

```
1 Fonction Air_cercle(Entier rayon)
2 {
3   PI <- 3.14159;                // (0)
4   Double volume;
5   volume <- PI * rayon * rayon // (1)
6   RENVOYER volume;             // (2)
7 }
```

- **L'instruction (0)** : l'affectation d'une valeur à une constante. On peut compter celle-ci comme étant une opération (certains l'omettront, ce qui importe peu)
- **L'instruction (1)** : est composée de deux opérations arithmétiques et d'une affectation. On peut la compter comme 3 opérations ou comme une seule (ce qui importe peu aussi).
- **L'instruction (2)** : la production de la valeur résultante de l'exécution de la fonction, est aussi une opération.

Si on additionne toutes les opérations, on arrive à :

- **2 opérations** : si on ne compte pas l'affectation d'une valeur à la constante (opération (0)) et si on compte l'instruction (1) comme une seule opération,
- **5 opérations** : si on compte les instructions de manière plus rigide.

L'algorithme sera donc **O(2)** ou **O(5),** tout dépendant de la manière de compter les opérations. L'important ici n'est pas la valeur exacte entre les parenthèses suivant le **O**, mais le fait que cette valeur

soit constante. Lorsqu'un algorithme est **O(c)** où *c* est une constante, on dit qu'il s'agit alors d'un algorithme en temps constant.

Une complexité constante est la complexité algorithmique idéale, puisque peu importe la taille de l'échantillon à traiter, l'algorithme prendra toujours un nombre fixé à l'avance d'opérations pour réaliser sa tâche. Tous les algorithmes en temps constant font partie d'une classe nommée **O(1)**.

C. *Calcul de complexité*

Afin de calculer la complexité d'un algorithme nous suivront les règles suivantes :

- O(c) = O(1) (C est une constante)
- O(cT) = cO(T)=O(T)
- O(T1)+O(T2)= O(T1+T2)=max(O(T1); O(T2)) ;
- O(T1) O(T2)= O(T1*T2)

Pour calculer la complexité d'un algorithme, on calcule la complexité de chaque partie de l'algorithme et on les combine en utilisant les règles précédentes avec pour :

- **Cas d'une instruction simple** : Les instructions de base (lecture, écriture, affectation comparaison, …) prennent un temps constant, noté O(1).
- **Cas d'une suite d'instructions ou d'une branche conditionnelle :** le temps d'exécution d'une séquence est déterminé par la règle de la somme.
- **Cas d'une boucle :** On multiplie la complexité du corps de la boucle par le nombre d'itérations.

Exemple : soit un algorithme qui renvoie la somme de tous éléments d'un ensemble représenté dans un tableau avec une longueur «n ».

```
1  Fonction Somme(Tableau T)
2  {
3    S <- 0;                                   O(1)
4
5    POUR i ALLANT_DE 0 A T.longeur-1   n x
6       S <-S+T[i];                            O(1)
7    FIN_POUR
8
9   RENVOYER S                                 O(1)
10 }
```

Au global, la complexité de cet algorithme est : O(1) + n * O(1) +O(1)=**O(n) (Complexité linéaire)**

D. *Calcul de complexité*

Le but de l'analyse de complexité est de pouvoir comparer plus facilement différents algorithmes qui effectuent la même tâche. On préférera par exemple l'algorithme qui s'exécute en **O(n)** par rapport à celui en **$O(n^2)$.**

Il devient également possible de **prédire le temps** que prendra un algorithme à trouver la solution sur une instance très grande en envisageant une mesure de temps d'exécution faite sur une plus petite instance.

En effet, nous nous intéressons par exemple à observer combien de fois plus de temps est nécessaire à un algorithme quand on double la taille des données d'entrée.

Le tableau suivant donne quelques complexités dans l'ordre du plus courant au moins courant. Il donne également une idée de la taille maximale que peuvent avoir les données (colonne « Max n ») avant que l'algorithme devienne impraticable (cela prendrait trop de temps pour résoudre le problème).

Tableau 1 : Tableau comparatif de quelques complexitésfréquentes [Réf]

Complexité	Nom courant	Temps quand on double la taille de l'entrée	Max n
$O(n)$	linéaire	prend 2 fois plus de temps	10^{12}
$O(1)$	constant	prend le même temps	pas de limite
$O(n^2)$	quadratique	prend 4 fois plus de temps	10^6
$O(n^3)$	cubique	prend 8 fois plus de temps	10 000
$O(\log n)$	logarithmique	prend seulement une étape de plus	$10^{10^{12}}$
$O(n \log n)$	linearithmique	prend deux fois plus de temps + un petit peu	10^{11}
$O(2^n)$	exponentiel	prend tellement de temps que c'est inconcevable	30

On observe une séparation nette entre la complexité exponentielle et les autres, qu'on appelle polynomiales. Les algorithmes exponentiels doivent être évités à tout prix, sauf si l'on sait que les instances sont très petites. La Figure 2 montre Les classes de complexité les plus fréquentes (par ordre croissant selon O(.).

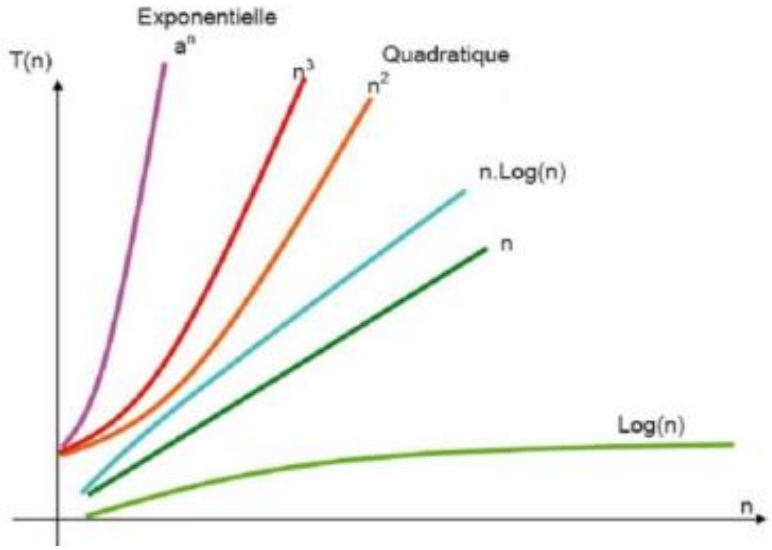


Figure 2 : Classes de complexité par ordre croissant de O(.)

Dans la section suivante, nous allons appliquer le calcul de complexité sur un domaine de l'algorithmique qui a été le plus étudié et qui a conduit à des résultats remarquables sur la construction d'algorithmes qui est la recherche et le tri d'une séquence. Dans ce chapitre, nous optons pour les tableaux comme séquence avant de reprendre ces opérations sur d'autres structures de données dans les chapitres qui suivent.

IV. Les algorithmes de recherche

Plusieurs stratégies existent pour trier et rechercher des éléments dans des tableaux. On peut déterminer leur efficacité grâce au calcul de la complexité. Cela correspond à la quantité de ressources nécessaires à l'exécution du programme.

IV.I. Recherche séquentielle

La recherche séquentielle (ou linéaire) consiste à comparer la valeur recherchée à toutes les valeurs présentes dans le tableau :

A. Recherche séquentielle itérative

L'algorithme suivant permet de faire une recherche séquentielle itérative de *valeur* dans un tableau *tab* de taille *n*.

```
recherche(tab, valeur)                                  
   {                                                    
     i ⟵ 0                                  O(1)       
     i_val ⟵ -1                             O(1)       
     n⟵ Longueur(tab)                       O(1)       
                                            n *  (Pire des cas)
      TantQue i < n   Faire                    O(1)     
            Si tab[i] = valeur Alors             O(1)   
                  i_val ⟵ i                      O(1)   
            FinSi                                       
            i ⟵ i + 1                              O(1) 
      FinTantQue                                        
      Retourner  i_val                             O(1) 
  }
```

Le paramètre de complexité est la longueur du tableau, qu'on appelle **n**. Avec : n = la taille du tableau, a = affectation, c = comparaison et o = opération, la Complexité au pire des cas (valeur n'est pas dans le tableau) est : 5*a + n*(2*c+2*a) +o=$\boldsymbol{\alpha * n + \beta = O(n).}$

B. Recherche séquentielle récursive

La récursion se fait sur la taille (= nombre d'éléments) du sous-tableau :

```
recherche(tab, valeur, i)
{
     i⟵  0
      b ⟵ tab.longeur
      Si i = n Alors
            Renvoyer -1
      Sinon
            Si tab[i] = valeur Alors
                  Retouner i
            Sinon
                  Retourner recherche(tab, valeur, i + 1)
            FinSi
      FinSi
}
```

- Si la taille est 0 : l'élément cherché n'y est pas ;
- Sinon on compare l'élément cherché avec le premier élément :
    - S'il y a égalité : on renvoie cette position ;
    - Sinon on recommence sur le sous-tableau privé de son premier élément.

IV.II. Recherche dichotomique

Le préalable à la méthode de recherche dichotomique est de disposer **d'un ensemble trié de données**, car la détermination du sous-ensemble dans lequel se poursuit la recherche se fait par comparaison entre la valeur recherchée et les valeurs de début et de fin du sous-ensemble.

A. Recherche séquentielle itérative et récursive

Le programme suivant permet de faire la recherche de *valeur* dans un tableau *tab* d'une façon itérative

```
RechercheDichotomie(tab, valeur)
{
    trouvé ⟵ FAUX
    résultat ⟵ -1
    gauche ⟵ 0
    droite ⟵ tab.Longueur - 1
    TantQue (gauche <= droite) ET (NON trouvre) Faire
        milieu ⟵ (droite + gauche) / 2
        Si tab[milieu] = valeur Alors
            resultat ⟵ milieu
            trouve ⟵ VRAI
        SinonSi tab[milieu] < valeur Alors
            gauche ⟵ milieu + 1
        Sinon
            droite ⟵ milieu - 1
        FinSi
    FinTantQue
    Retourner résultat
}
```

La version récursive est présentée dans le programme suivant :

```
RechDichoRecu(tab, valeur, gauche, droite)
{
    Si gauche > droite Alors
        Retourner -1
    Sinon
        milieu ⟵ (droite + gauche) // 2
        Si tab[milieu] = valeur Alors
            Renvoyer milieu
        SinonSi tab[milieu] < valeur
            Retourner RechDichoRecu(tab, valeur, milieu + 1, droite)
        Sinon
            Retourner RechDichoRecu(tab, valeur, gauche, milieu - 1)
        FinSi
    FinSi
}
```

B. *Complexité de recherche dichotomique*

Pour calculer la complexité de la recherche dichotomique, il faut chercher combien doit-on effectuer d'itérations pour un tableau de taille n dans le cas le plus défavorable (l'entier valeur n'est pas dans le tableau tab) ? Sachant qu'à chaque itération de la boucle on divise le tableau en 2, cela revient donc à se demander combien de fois faut-il diviser le tableau en 2 pour obtenir, à la fin, un tableau comportant un seul entier ? Autrement dit, combien de fois faut-il diviser n par 2 pour obtenir 1 ?

Mathématiquement cela se traduit par l'équation :

$$\frac{n}{2 \times 2 \times 2 \ldots \ldots \times 2 \times 2} = \frac{n}{2^a} = 1$$

Avec a le nombre de fois qu'il faut diviser n par 2 pour obtenir 1. Pour avoir la complexité, il faut donc trouver **a** ! Nous allons ainsi utiliser la notion mathématique : le **"logarithme base 2"** noté **$\log_2$** définit par: $Log2(2^x) = x$. Nous avons donc :

$$\frac{n}{2^a} = 1 \;\Rightarrow\; n = 2^a \;\Rightarrow \log_2(n) = \log_2(2^a) = a$$

Donc $\boldsymbol{a = log_2(n)}$

Nous pouvons donc dire que la complexité en temps dans le pire des cas de l'algorithme de recherche dichotomique est **O($\log_2$(n)).**

Nous pouvons constater que l'algorithme de recherche dichotomique est plus efficace que l'algorithme de recherche séquentielle car **n>$\log_2$(n**) quel que soit (voir Figure 2).

Cependant, il ne faut pas perdre de vu que dans le cas de la recherche dichotomique, il est nécessaire d'avoir un tableau trié, si au départ le tableau n'est pas trié, il faut rajouter la durée du **tri.**

Nous allons détailler dans ce qui suit quelques **algorithmes de tri**.

V. Les algorithmes de tri

Plusieurs stratégies existent pour trier des tableaux. On peut déterminer leur efficacité grâce au calcul de la complexité. Cela correspond à la quantité de ressources nécessaires à l'exécution du programme : Le tri par sélection, Le tri à bulle, Le tri par insertion, Le tri rapide, Le tri fusion, ……

Nous allons étudier dans ce qui suit quelques un de ces mes méthodes de tris et d'autres algorithmes vont être détaillés dans les séances de TD(s).

V.I. Le tri par sélection

Cet algorithme est intuitif. D'abord, il cherche l'élément le plus petit du tableau. Puis, il échange cet élément avec l'élément en première place du tableau. Enfin, il réitère ces actions jusqu'à ce que le tableau soit entièrement trié.

Exemple :

6	1	9	3
1	6	9	3
1	6	9	3
1	3	9	6
1	3	9	6
1	3	6	9
1	3	6	9
1	3	6	9

Le tri itératif par sélection est présenté dans l'algorithme suivant :

```
prodcedure Tri_Selection_iteratif (Tableau a[n] )
{
  POUR i de 0 A n-2 FAIRE
    indice_min <- i;
    POUR j DE i+1 A n-1 FAIRE
      SI a[j]<min ALORS indice_min <- j;
    FIN POUR
      echanger(a,i,indice_min);///echanger a[i] et a[indice_max];
  FIN POUR
}
```

L'algorithme qui permet de faire le tri par sélection d'une manière récursive est le suivant :

```
Procedure Tri_Selection_recursif (Tableau a[n],Entier IndiceR )
{
   Si ( IndiceR <> n-1)  ALORS
      DEBUT_SI
   //(Recherche l'indice du minimun dans le tableau a à partir de IndiceR+1
      indice_min=RechercheMin(a,IndiceR+1);
      echanger(a,IndiceR,indice_min);///echanger a[i] et a[indice_min];
      Tri_Selection_recursif (a,IndiceR+1);
   FIN_SI
}
```

La complexité du tri par sélection se calcule de la manière suivante : Pour rechercher le premier minimum on effectue n-1 comparaisons, pour le second n-2, ... le nombre total de comparaisons est de :

$$(n - 1) + (n - 2) + \ldots + 1 = \frac{n(n+1)}{2}$$ donc en **$O(n^2)$**

V.II. Le tri à bulles

Le principe du tri à bulles (bubble sort) est de comparer deux à deux les éléments e1 et e2 consécutifs d'un tableau et d'effecteur une permutation si e1 > e2. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation. Au cours d'une passe du tableau, les plus grands éléments remontent de proche en proche vers la droite **comme des bulles vers la surface.**

On s'arrête dès que l'on détecte que le tableau est trié : si aucune permutation n'a été faite au cours d'une passe.

Exemple :

Passe1

10	50	20	40	15	17	32
10	50	20	40	15	17	32
10	50	20	40	15	17	32
10	20	50	40	15	17	32
10	20	50	40	15	17	32
10	20	40	50	15	17	32
10	20	40	50	15	17	32
10	20	40	15	50	17	32
10	20	40	15	50	17	32
10	20	40	15	17	50	32
10	20	40	15	17	50	32
10	20	40	15	17	32	50

Borne

Passe2

10	20	40	15	17	32	50
10	20	40	15	17	32	50
10	20	40	15	17	32	50
10	20	15	40	17	32	50
10	20	15	40	17	32	50
10	20	15	17	40	32	50
10	20	15	17	40	32	50
10	20	15	17	32	40	50
10	20	15	17	32	40	50
10	20	15	17	32	40	50

Borne

Le tri à bulles : Algorithme itératif

```
PRODECURE Tri_bulle (Tableau a[n])
{
  Booleen permut  ;

  REPETER
        permut = FAUX
        POUR i  DE 1 à n-1 FAIRE
            SI a[i] > a[i+1] ALORS
               echanger (a,i,i+1)///Echanger a[i] et a[i+1]
               permut = VRAI
            FIN SI
        FIN POUR
  TANT QUE permut = VRAI
}
```

Le tri à bulles : Algorithme récursif

```
PRODECURE Tri_bulle_recursif (Tableau a[n],Entier indice)
{
  indice = n;

SI (n <> 1) ALORS
   POUR i ALLANT_DE 0 A n-1

        Si (arr[i] > arr[i + 1]) alors
            echanger(a[i], a[i + 1]);
        Fin SI
   Fin Pour
   Tri_bulle_recursif(a, n - 1);
FIN_SI

}
```

Le tri à bulles : Complexité

Le cas le plus défavorable se produit lorsque le tableau est trié à l'envers, et que le nombre maximum de comparaisons et d'échanges doit être effectué,

Donc, s'il y a n passes, alors la complexité temporelle moyenne peut-être donnée par :

$$(n - 1) + (n - 2) + \ldots + 1 = \frac{n(n+1)}{2}$$ Donc en **O(n^2)**

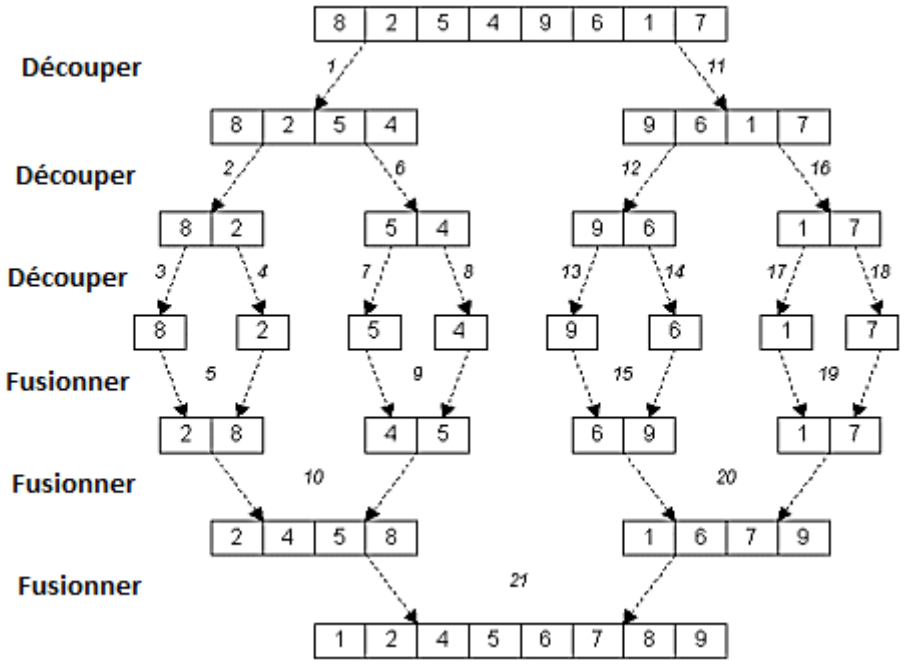
V.III. Le tri de Fusion

Le tri fusion est l'un des algorithmes de tri les plus populaires et les plus efficaces. Il est basé sur le principe de l'algorithme diviser pour mieux régner.

Il fonctionne en divisant le tableau en deux moitiés de manière répétée jusqu'à ce que nous obtenions un tableau divisé en éléments individuels. Un élément individuel est un tableau trié en soi.

Le tri fusion fusionne de manière répétée ces petits tableaux triés pour produire de plus grands sous-réseaux triés jusqu'à ce que nous obtenions un tableau trié final.

Exemple :



Le tri à bulles : Algorithme

```
1  Tri(Tableau T)
2  {
3  si (T.longueur=1) alors
4     retourner;
5  Fin Si
6  m <- T.longueur/2;
7  T1<-T[1:m] ;
8  T2<-T[m+1:T.longueur] ;
9  Tri(T1);
10 Tri(T2);
11 Fusion(T, T1, T2);
12 }
```

La procédure Tri fait appel à la procédure Fusion :

```
1  Fusion(T, T1, T2) {
2  i<-0; i1 <-0; i2 <- 0;
3  TANT_QUE (i1 < T1.longueur et i2 < T2.longueur)
4        si (T1[i1] < T2[i2]) alors
5             T[i] <- T1[i1];
6              i1  <-  i1+ 1;
7        sinon
8             T[i] <- T2[i2];
9              i2<- i2 + 1;
10     i <- i + 1;
11 FIN_TANT_QUE
12 si ( i1 =T1.longueur)alors
13    pour j de i2 à T2.longueur-1 faire
14          T[i] <- T2[j];
15          i <- i + 1;
16    FIN_POUR
17 FIN_SI
18 si ( i2 =T2.longueur)alors
19    pour j de i1 à T1.longueur-1 faire
20          T[i] <- T1[j];
21          i <- i + 1;
22    FIN_POUR
23 FIN_SI }
```

La fonction Tri utilise la fonction Fusion, nous allons donc d'abord analyser cette dernière :

- Il y a deux boucles : un « tant que » et un « pour » (juste l'une des Pour va s'exécuter car soit T1 finit avant T2 soit l'inverse) , et on ne sait pas à l'avance combien d'opération chacune d'elle va effectuer;
- En revanche, on sait que le nombre total d'itérations combiné des deux boucles est exactement n=T1.longueur+T2.longueur;
- Puisque les deux boucles font un nombre constant d'opérations, la complexité de la fonction est O(n). En effet, nous copions ici tous les éléments de T1 et T2 vers T exactement une fois.

Voyons maintenant la fonction Tri, qui est récursive :

- Si C_n est le nombre d'opérations faites par Tri sur une Tableau de taille n, nous avons :

$$C_n=O(n) + 2*C_{n/2} + O(n).$$

  - Le premier O(n) correspond à la séparation en S1 et S2 et le deuxième leur fusion.
  - $C_{n/2}$ le nombre d'itération pour la création de T1 T2;

Nous allons maintenant dessiner, dans la figure 3, « l'arbre d'appels » d'une exécution de Tri, où la valeur de chaque nœud représente la taille de la séquence d'un appel (récursif) à Tri.

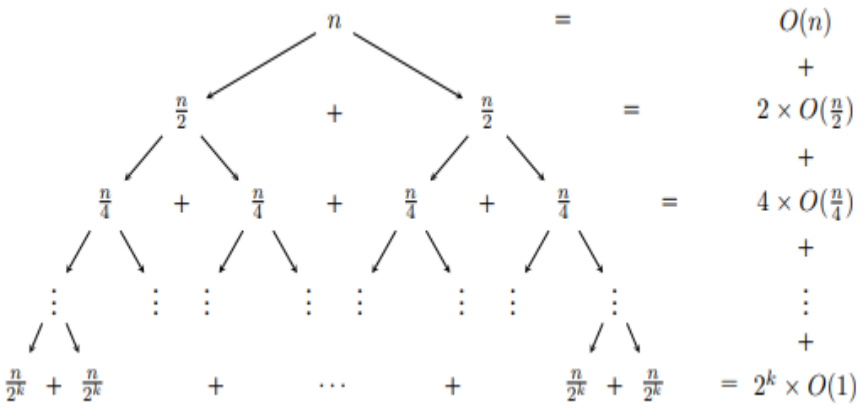


Figure 3 :Arbre d'appels d'une exécution de Tri par Fusion

Le coût du tri est la somme des coûts de tous les nœuds de cet arbre d'appels. Le coût d'un nœud est maintenant simplement le coût de la séparation et de la fusion, puisque le coût des deux appels récursifs est maintenant compté dans les nœuds des fils.

Au total, la complexité de l'algorithme du tri-fusion est **O(n) * le nombre de niveaux**, c'est-à-dire la hauteur de l'arbre.

Comme pour la recherche dichotomique, on trouve que cette hauteur est exactement le nombre de fois qu'il faut **diviser n par 2 pour atteindre 1, soit $k=\log_2(n)$.**

La complexité du tri fusion est donc en :

$$\mathbf{O(n*\log_2(n));}$$

Ce qui est bien meilleur que le tri par sélection analysé plus précédemment.

VI. Conclusion

Ce chapitre était, en premier, un rappel des notions fondamentales de la complexité algorithmique. Il nous a permis, ensuite, d'appliquer ces notions sur des algorithmes de recherches et de tris sur des tableaux.

Dans le chapitre suivant, nous verrons comment manipuler d'autres structures de données (listes, piles, fichiers,…..).

Références

- Sylvain Perifel, Complexité algorithmique ,2014, édition Ellipses.
- Le cours de Complexité algorithmique par Florent Bouchez Tichadou université de grenoble-alpes.