Table des matières

INTRODUCTION

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'étapes de calcul qui transforment l'entrée en sortie. Il permet de résoudre un problème précis, et pour un problème donné, on peut avoir plusieurs algorithmes de résolution. Idéal étant de trouver de la solution la plus efficace ou optimale, il est préférable de soumettre ces algorithmes à une évaluation : d'où la notion de **complexité des algorithmes**.

I. GENERALITES SUR LA COMPLEXITE DES ALGORITHMES

1. Définition et but de calcul de complexité

La **complexité d'un algorithme** est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée. L'objectif premier étant de prendre deux algorithmes traitant le même problème, les comparer afin de trouver celui qui est le plus efficace ou optimale. L'efficacité ou l'optimalité d'un algorithme est mesurée par l'augmentation du temps de calcul en fonction du nombre des données. Nous avons donc deux éléments à prendre en compte :

- La taille des données :
- Le temps de calcul.

2. Types de complexité

Le calcul de la **complexité** d'un algorithme permet de mesurer sa **performance**. Il existe deux types de complexité :

- **complexité spatiale** : permet de quantifier l'utilisation de la **mémoire.**
- **complexité temporelle** : permet de quantifier la **vitesse** d'exécution. L'objectif d'un calcul de **complexité algorithmique temporelle** est de pouvoir comparer l'**efficacité** d'algorithmes résolvant le même problème. Dans une situation donnée, cela permet donc d'établir lequel des algorithmes disponibles est le plus **optimal**. Réaliser un calcul de complexité en temps revient à compter le nombre d'**opérations élémentaires** (affectation, calcul arithmétique ou logique, comparaison…) effectuées par l'algorithme. Puisqu'il s'agit seulement de comparer des algorithmes, les règles de ce calcul doivent être indépendantes :

- du langage de programmation utilisé ;
- du processeur de l'ordinateur sur lequel sera exécuté le code ;
- de l'éventuel compilateur employé.

Par soucis de simplicité, on fera l'hypothèse que toutes les opérations élémentaires sont à **égalité de coût**, soit 1 « unité » de temps.

*Exemple : a = b * 3 : 1 multiplication + 1 affectation = 2 « unité*s »

La **complexité** en temps d'un algorithme sera exprimée par une fonction, notée *T* (pour *Time*), qui dépend :

- de la taille des données passées en paramètres : plus ces données seront volumineuses, plus il faudra d'opérations élémentaires pour les traiter. On notera *n* le nombre de données à traiter.
- de la donnée en elle-même, de la façon dont sont réparties les différentes valeurs qui la constituent.

Pour évaluer la complexité temporelle, nous pouvons passer par plusieurs formes.

3. Formes de complexité

La complexité d'un algorithme est évaluée sur trois formes qui sont les suivantes :

- La complexité dans le meilleur des cas : c'est la situation la plus favorable,
  Par exemple : recherche d'un élément situé à la première position d'une liste
- La complexité dans le moyen des cas : fait la moyenne des temps de calcul pour toutes les donnes de taille n.
- La complexité dans le pire des cas : c'est la situation la plus défavorable,
  Par exemple : recherche d'un élément dans une liste alors qu'il n'y figure pas

II. COMPORTEMENT ASYMPTOTIQUE

1. La notation O (grand O)

Etant donnée une fonction f(n), O(f(n)) est l'ensemble des fonctions g(n) telles qu'il existe une constante positive réelle c et un entier non négatif n_0 tel que pour tout $n > n_0$, $g(n) <= c \times f(n)$. La notation O(f) décrit le comportement asymptotique d'une fonction, c'est à dire pour des grandes valeurs. En d'autres mots, g est O(f) lorsque, pour des valeurs suffisamment grandes de n ($n > n_0$), le rapport g(n)/f (n) reste toujours bornée par c.

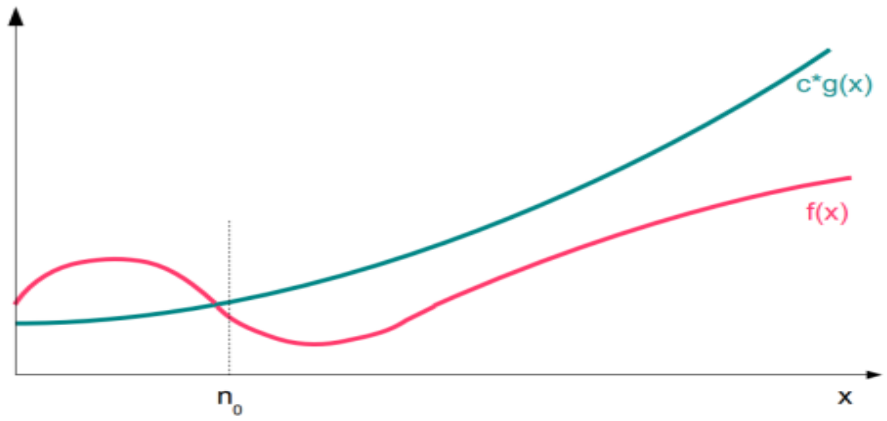


Figure 1: ILLUSTRATION DE LA NOTATION O

Propriétés :

- ✓ Si $f1(n) \in O(g1(n))$ et $f2(n) \in O(g2(n))$ alors $(f1+f2)(n) \in O(g1(n)+g2(n))$
- ✓ Si $f1(n) \in O(g1(n))$ et $f2(n) \in O(g2(n))$ alors $(f1 \times f2)(n) \in O(g1(n) \times g2(n))$

Ces propriétés sont utiles pour simplifier l'analyse d'algorithmes comme suit : Supposons que les temps d'exécution des opérations A et B soient respectivement O(f(n)) et O(g(n)). Alors le temps pour A suivi de B sera de O(f (n) + g(n)).

Notons que dans le cas où, par exemple, f domine g de façon stricte ($g \in o(f)$), on peut alors en conclure que A suivi de B est de temps O(f(n)). De même, si $f \in \Theta((g))$, le temps sera alors simplement O(f(n)).

Supposons que chaque exécution d'une boucle requiert un temps d'exécution O(f(n)) et que la boucle est exécutée O(g(n)) fois. Alors le temps pour la boucle sera $O(f(n) \times g(n))$.

Encore une fois, ces simplifications restent valides si on remplace O par Θ ou &Omega.

Exemple :

soient les fonctions suivantes:

$f(n) = n^2 + 10n + 2$ et $g(n) = n^2$

dire que f(n) =O(g(n)) signifie qu'à un certain rang $n_0>0$ donné, il existe une constante positive c telle que f(n) <= c.g(n) pour tout $n>n_0$.

supposons donc que notre $n_0 = 3$ et trouvons la constante c :

$n_0 = 3 \Rightarrow n>3$ donc $n^2 + 10n + 2 < c.n^2 \Leftrightarrow (3)^2 + 10*3 +2 < c.(3)^2$

$\Leftrightarrow 41 < 9.c \Leftrightarrow 41/9 < c$

donc nous pouvons prendre comme valeur de c toute constante réelle positive strictement supérieur à 41/9

1. La notation Ω (grand oméga)

Etant donnée une fonction f(n), Ω(f(n)) est l'ensemble des fonctions g(n) telles qu'il existe une constante positive réelle c et un entier non négatif n_0 tel que pour tout $n>n_0$, $g(n) \geq c\times f(n)$. En d'autres mots, f(n) est une (forme de) borne inférieure asymptotique pour les fonctions dans Ω(f(n)), alors qu'elle est une borne supérieure asymptotique pour les fonctions dans O(f(n)). Si une fonction est à la fois dans Ω(f(n)) et dans O(f(n)), alors on peut en conclure que son taux de croissance est équivalent à celui de f(n).

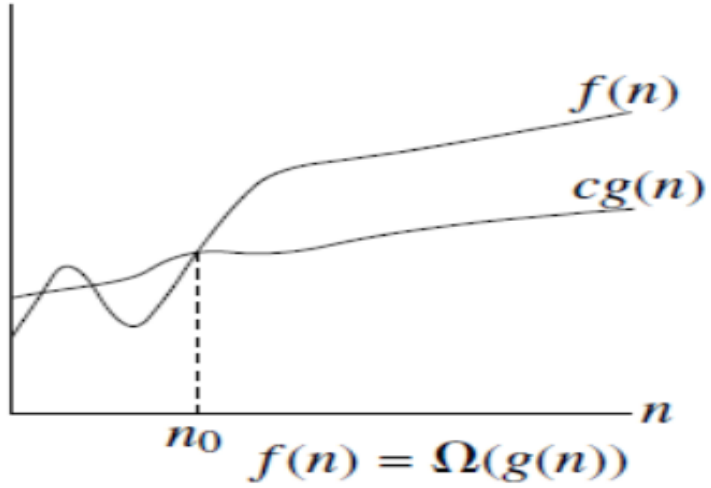


Figure 2: ILLUSTRATION DE LA NOTATION Ω

Exemple:

Soient les fonctions suivantes:

$f(n) = n^2 + 10n +2$ et $g(n) = n^2$

Dire que f(n) =Ω(g(n)) signifie qu'à un certain rang n_0 donné, il existe une constante positive c tel que $f(n) \geq c.g(n)$ pour tout $n\geq n_0$.

Supposons donc que notre $n_0 = 2$ et trouvons la constante c :

$n_0 = 2 \Rightarrow n\geq 2$ donc $n^2 + n +2 \geq c.n^2 \Leftrightarrow (2)^2 + 10*2 +2 \geq c.(2)^2$

$\Leftrightarrow 26 \geq 4.c \Leftrightarrow 26/4 \geq c \Leftrightarrow c \leq 6$

donc nous pouvons prendre comme valeur de c 6 ou toute constante réelle positive inferieure ou égal à 6. cet exemple peut être interprété comme suit : a partir du rang 2 ($n_0 = 2$ et donc $n\geq 2$) , toute valeur de c positive inférieur ou égale a 6 vérifie l'inégalité $f(n) \geq c.g(n)$. On peut donc conclure que f(n) =Ω(g(n))

2. La notation Θ (Thêta)

Etant donnée une fonction f(n), Θ(f(n)) est l'ensemble des fonctions g(n) telles qu'il existe une constante positive réelle c, une constante positive réelle d, et un entier non négatif n_0 tel que pour tout $n > n_0$, $c \times f(n) \leq g(n) \leq d \times f(n)$. En d'autres mots : Θ(f(n)) =Ω(f(n))∩O(f(n)).

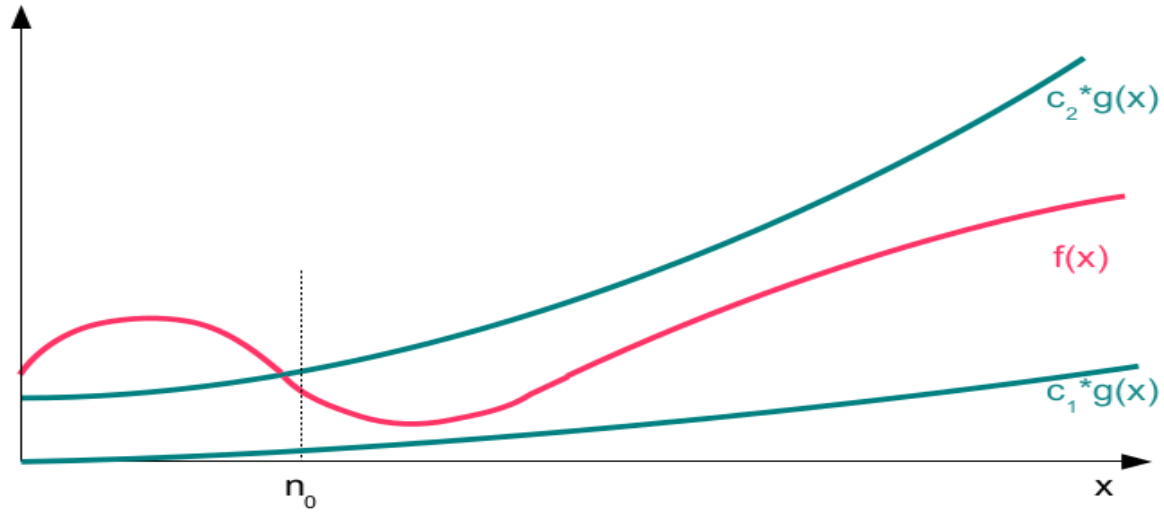


Figure 3: ILLUSTRATION DE LA NOTATION Θ

3. La notation o (petit o)

Etant donnée une fonction f(n), o(f(n)) est l'ensemble des fonctions g(n) telles que pour toute constante positive réelle c, il existe un entier non négatif N tel que pour tout n>N, g(n) < c× f(n) En d'autres mots, g(n) est dominée asymptotiquement de façon stricte par f(n). Ainsi, on a la propriété suivante : g(n) ∈ o(f(n)) ⇒ g(n) ∈ O(f(n)) - &Omega(f(n)). La notation Θ sépare l'ensemble des fonctions de complexité en une collection de sous-ensembles disjoints (classes d'équivalence). Pour l'analyse des algorithmes, on utilise donc le représentant le plus simple Θ(1), Θ(n) …

4. Propriétés

1) g(n) ∈ O(f(n)) ⇔ f(n) ∈ Ω(g(n))
2) g(n) ∈ Θ(f(n)) ⇔ f(n) ∈ Θ(g(n))
3) Si b > 1 et a > 1, alors $\log_a(n) \in \log_b(n)$ (Le logarithme utilisé n'a pas d'importance)
4) Si b > a > 0, alors an ∈ o(bn) (toutes les fonctions exponentielles sont dans des classes différentes)
5) Pour tout a > 0, an ∈ o(n!) (n! est la pire des fonctions de complexité exponentielle)
6) Soit les fonctions de complexité ordonnées comme suit, où k > j > 2 et b > a > 1 : Θ(1) Θ(lg n) Θ(n) Θ(n lg n) Θ(n2) Θ(nj) Θ(nk) Θ(an) Θ(bn) Θ(n!). Si une fonction g(n) est dans une catégorie à gauche de la catégorie contenant f(n), alors g(n) ∈ o(f(n)).
7) Si c≥0, d>0, g(n) ∈ O(f(n)) et h(n) ∈ Θ(f(n)) alors c×g(n)+d×h(n) ∈ f(n)

De façon plus générale, la propriété 6 indique que toute fonction logarithmique est éventuellement meilleure que n'importe quelle fonction polynomiale, que toute fonction polynomiale est éventuellement meilleure que n'importe quelle fonction exponentielle, et que toute fonction exponentielle est meilleure que la fonction factorielle. Les propriétés 6 et 7 peuvent être utilisées de façon répétitive pour simplifier des expressions et déterminer la catégorie la plus simple à laquelle appartient une fonction.

Exemple : On veut montrer que $5n + 3 \lg n + 10 n \lg n + 7n2 \in \Theta(n^2)$:

- $7n^2 \in \Theta(n^2)$
- $10 n \lg n + 7n^2 \in \Theta(n^2)$
- $3 \lg n + 10 n \lg n + 7n^2 \in \Theta(n^2)$
- $5n + 3 \lg n + 10 n \lg n + 7n^2 \in \Theta(n^2)$

Remarques sur l'utilisation de O vers Ω vers Θ

La notation O décrit une borne supérieure. On peut donc l'utiliser pour obtenir une borne supérieure sur le temps d'exécution dans le pire des cas. Evidemment, ce faisant, on obtient aussi une borne supérieure pour des données quelconques. Par contre, on peut aussi utiliser Θ pour décrire une borne, à la fois inférieure et supérieure, du temps d'exécution dans le pire des cas. Evidemment, cela ne signifie alors pas qu'une exécution de l'algorithme sur des données arbitraires sera bornée par Θ. La notation Ω, quant à elle, décrit une borne inférieure. Habituellement, on l'utilise donc pour borner le temps d'exécution dans le meilleur des cas. Par contre, on peut aussi l'utiliser pour décrire une borne inférieure dans le pire cas. Par exemple, soit l'algorithme suivant, où le temps d'exécution de chacune des parties est indiqué en commentaires :

```
void f(int[] t){

 for( int i =0 ; i<t.length; ++i){

if(test(i))

        f1(); // Θ(1)

else        f2(); // Θ(n)    } }
```

Toutes les affirmations suivantes seraient alors correctes :

- f est $O(n^2)$.
- f est $\Omega(n)$.
- Dans le pire des cas, f est $\Theta(n^2)$.
- Dans le meilleur des cas, f est $\Theta(n)$.

Par contre, si le temps pour f1() était plutôt $\Theta(n)$, on pourrait alors conclure simplement que f, dans tous les cas, est $\Theta(n^2)$.

5. Classe de Complexité

Notations	Dénominations
O(1)	**Constante**
O(log(n))	**Logarithmique**
O(n)	**Linéaire**
O(n log(n))	**Quasi-linéaire**
$O(n^2)$	**Quadratique**
$O(n^3)$	**Cubique**
$O(n^k)$	**Polynomiale**
$O(c^n)$	**Exponentielle (c>1)**
O(n !)	**Factorielle**

III. METHODES DE CALCULS DES COMPLEXITES

1. Algorithmes itératifs

1.1. Algorithme sans structure de contrôle

Soient deux suites d'actions A1 et A2, et A1+A2, la suite « A 1 suivi de A2 »

```
Début
        Action A1 ;
        Action A2 ;
Fin
```

Le cout d'exécution de : $T_{(A1+A2)}(n) = T_{A1}(n) + T_{A2}(n)$.

1.2. Algorithme avec structure conditionnelle

Soit l'algorithme suivant :

```
Début
        Si Condition alors
                Action A1 ;
        Sinon
                Action A2 ;
Fin
```

Le temps d'exécution de A est : $T_A(n) = T_C(n) + \max\{(T_{A1}(n), T_{A2}(n))\}$

1.3. Algorithme avec structures itératives

- La boucle pour (For en Anglais)

Algorithme

```
Debut
        Pour (I=0, I<n, I++) faire
                Action A1
        F pour
Fin
```

Le cout d'exécution de cet algorithme est donné par : $T(n) = \Sigma^{n}_{i=0} T(A1) = n*T(A1)$.

- Boucle Tantque…faire

```
Début
        Tantque (Condition) faire
                Action A ;
        FinTantque
```

Le cout d'exécution de cet algorithme est donné par : **T(n) = TC(n) *Σ (TA)**

2. Algorithmes récursifs

Les algorithmes récursifs sont des algorithmes qui s'appellent elles même, elles sont également un moyen rapide pour poser certains problèmes algorithmiques nous allons voir en détails comment elles fonctionnent

Prenons un problème simple mais auquel vous n'avez pas pensé à utiliser la récursivité : le calcul d'une factorielle. Considérons *n !* (qui se lit factorielle de n) comme étant la factorielle à calculer, nous aurons ceci : *6!=6*5*4*3*2*1.* Dans cette situation, nous pouvons déjà terminer notre règle de sortie de notre fonction récursive : la valeur 1 qui symbolise la fin de la récursion. En effet, il faut une condition de sortie pour la fonction, mais il faut être très vigilant quant au choix de la condition, vous devez être sûr qu'elle soit validée à un moment ou un autre sinon c'est comme si vous créez une boucle infinie sans condition de sortie. Dans ces algorithmes récursifs, la complexité suit généralement une relation de récurrence. Cette relation est de la forme :

$$\mathbf{U_n = f\ (u_0, u_1, \ldots, u_{n-1}, n)}$$

Ces récurrences sont classées sous deux formes :

Les récurrences linéaires ou simples

Récurrence linéaire a coefficient linéaire d'ordre k. La récurrence linéaire a coefficient se définir sur deux formes tel que :

une équation de récurrence linéaire a coefficient d'ordre k a pour forme générale :

$Ui = ci\ (0 \leq I \leq k\text{-}i)$

$$Un = \Sigma ki = 1\ ai\ Un - I + f(n)$$

Elle se met sous deux forme savoir :

- **La forme homogène**

Une équation de récurrence est dite homogène lorsque la fonction f(n) est nulle sinon elle est dite hétérogène. Pour résoudre une équation de ce type, on doit premièrement trouver un polynôme caractéristique. Le polynôme caractéristique associer à équation récurrence linéaire homogène d'ordre k est :

$$Un = a1Un\text{-}1 + \ldots + akUn\text{-}k$$

et le polynôme: $P(x) = x^k - a1x^{k-1} - \ldots - ak\text{-}1x^{1-ak}$

Méthode de résolution :

Déterminer les solutions du polynôme caractéristique : p racines ri, $1 \leq I \leq p$ de multiplicité mi, $1 \leq I \leq p$

Déterminer les solutions de la récurrence homogène :

L'ensemble des suites solution de la récurrence homogène SH a une structure d'espace vectoriel de dimension $k = \Sigma i=1p\ mi$

Chaque racine r de p d'ordre de multiplicité m donne m vecteurs d'une base de SH :

$$r^n, nr^n, \ldots, r^{m-1}r^n \text{ et on a :}$$

$SH = vect\ (U\ \{r^{ni}, nr^{ni}, \ldots, r^{mi}\ r_i^n\})$

Remarque :

Lorsqu'on a une solution double suite à la résolution de l'équation caractéristique alors la solution finale se met sur la forme de : $C(n = Ar^n + Bnr^n$.

Exemple : soit la récurrence suivante

$$c(n) = c(n\ -1) + c(n - 2)$$

$$Un = Un\text{-}1 + Un\text{-}2$$

L'ordre de k vaut k =2 et le polynôme caractéristique est :

$$P(r) = r^2 - r - 1$$

- **la forme non homogène**

Elle dit non homogène lorsqu'elle se met sous la forme :

$$Un = \Sigma ki = 1\ ai\ Un - I + f(n)$$

Pour résoudre une telle équation, on déterminer la solution particulière de l'équation récurrente avec second membre c'est-à-dire en déterminant d'abord la solution de l'équation homogène en suite vérifier les contions sur la fonction f(n). On cherche cette solution particulière sous une forme liée à la fonction f(n) en vérifiant les conditions suivantes :

Si f(n) est de la forme bn.Q(n) ou Q est un polynôme de degré d:

la solution particulière est bn.T(n) ou T(n) est un polynôme de degré :

d si b est n'est pas racine du polynôme caractéristiques P

d + mj , si b = rj

Si f(n) est polynôme de degré d (idem pour le cas précèdent avec b = 1)

La solution particulière est aussi est un polynôme de degré :

d si 1 n'est pas une racine du polynôme caractéristique P

d + mj , si 1 = rj

Exemple d'application :

D(n) = d(n-1) + d(n-2) + n

Solution :

1. Déterminons d'abord la solution de l'équation homogène

   P(r) = (r - (1 + √5)/2)(r - (1 - √5)/2)

2. Vérifier les conditions sur la fonction f,qui f(n) = n

   f(n) = n = 1n.n qui n'est pas une solution de P

3. Déterminer la solution particulière

   on cherche une solution particulière sous forme :

   d(n) = 1n(αn + ß)

   d(n-1) = α(n-1) + ß

   d(n-2) = α(n-2) + ß

   αn + ß = α(n-1) + ß + α(n-2) + ß + n

   -(α + 1)n + 3α -ß = 0, pour tout n ≥ 2

   α = -1

   3α – ß = 0

   finalement α = -1 et ß = -3

   Un = A(1 + √5)/2)n + B(1 - √5)/2)n -n – 3

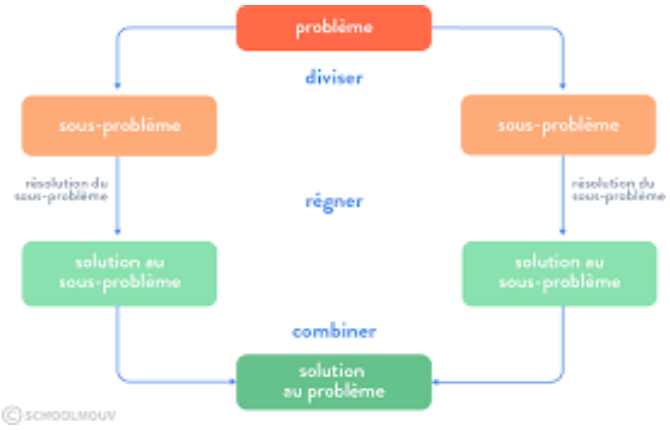
   D'ou d(n) = O((1 + √5)/2)n

3. Algorithmes de types « Diviser pour Régner »

Les algorithmes utilisant la stratégie "diviser pour régner" sont décomposés en 3 phases :

- **DIVISER** : le problème d'origine est divisé en un certain nombre de sous-problèmes
- **RÉGNER** : on résout les sous-problèmes (les sous-problèmes sont plus faciles à résoudre que le problème d'origine)
- **COMBINER** : les solutions des sous-problèmes sont combinées afin d'obtenir la solution du problème d'origine.

Pour un problème à résoudre (généralement complexe à résoudre), on divise ce problème en une multitude de petits problèmes, l'idée étant que les "petits problèmes" seront plus simples à résoudre que le problème original. Une fois les petits problèmes résolus, on recombine les "petits problèmes résolus" afin d'obtenir la solution du problème de départ. Le schéma ci-dessous donne le fonctionnement de la méthode « Diviser pour Régner » :



L'étude de la complexité de ce type d'algorithme récursifs peut se ramener à une récurrence au sens mathématique. Soit un algorithme qui pour traiter un problème de taille n le divise en a sous problèmes de taille n/b (a≥1, b>1). Si le problème est suffisamment petit (n = 1) il est résolu en un temps constant. Le temps nécessaire pour diviser le problème et pour combiner les différents sous-problèmes sont fonction de n (respectivement D(n) et C(n)). On a alors la récurrence suivante :

$$\mathbf{T(n)} = \begin{cases} \boldsymbol{\Theta(1)} & \boldsymbol{n \leq c} \\ \boldsymbol{aT(n/b) + D(n) + C(n)} & \boldsymbol{sinon} \end{cases}$$

Nous allons maintenant étudier un de ces algorithmes basés sur le principe diviser pour régner : le tri-fusion

Exemple du tri fusion

Le **tri fusion** consiste à trier récursivement les deux moitiés de la liste, puis à fusionner ces deux sous-listes triées en une seule. La condition d'arrêt à la récursivité sera l'obtention d'une liste à un seul élément, car une telle liste est évidemment déjà triée.

Voici donc les **trois étapes** (diviser, régner et combiner) de cet algorithme :

1. Diviser la liste en deux sous-listes de même taille (à un élément près) en la "divisant" par la moitié.

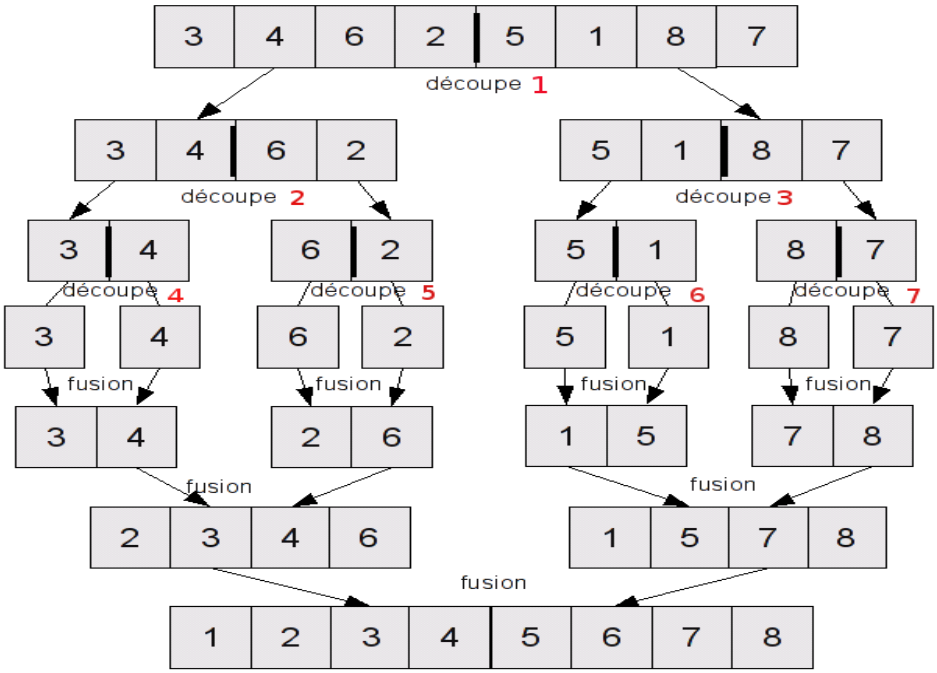
2. Trier récursivement chacune de ces deux sous-listes. Arrêter la récursion lorsque les listes n'ont plus qu'un seul élément (c'est- à dire lorsque l'indice de début est égale à l'indice de fin).
3. Fusionner les deux sous-listes triées en une seule.

L'algorithme est donné par ;

```
FUSSION (A, d, f, n)
  i ← d
  j ← d+1
  Initialiser un tableau C de taille n-d+1
  k← 1
  tant que i ≤ f et j ≤ r faire
       si A[i] < A[j] alors
          C[k] ← A[i]
          i ← i + 1
       sinon
          C[k] ← A[j]
          j ← j + 1
      k ← k + 1
  tant que i ≤ f faire
          C[k] ← A[j]
          i ← i + 1
          k ← k + 1
  tant que j ≤ r faire
          C[k] ← A[j]
          j ← j + 1
          k ← k + 1
    pour k ← 1 à r – d + 1 faire
A[p+k-1] ← C[k]
```

```
TRI-FUSION(A, p, r)
1  si p < r
2     alors q ← ⌊(p + r)/2⌋
3          TRI-FUSION(A, p, q)
4          TRI-FUSION(A, q + 1, r)
5          FUSION(A, p, q, r)
```

Exemple :



Etude de la complexité du TRI-FUSSION :

Pour déterminer l'équation de récurrence qui nous donnera la complexité de l'algorithme du tri-Fusion, nous étudions les trois phases de cet algorithme « diviser pour régner » :

Diviser : cette étape se réduit au calcul du milieu de l'intervalle [p ;r], sa complexité est donc en Θ(1).

Régner : l'algorithme résout récursivement deux sous-problèmes de tailles n/2, d'où une complexité en 2T(n/2).

Combiner : La complexité de cette étape est celle de l'algorithme de fusion qui est en Θ(n) pour la construction d'un tableau solution de taille n.

Par conséquent, la complexité de l'algorithme du tri fusion est donnée par le relation de récurrence suivante :

$$\mathbf{T(n)} = \begin{cases} \boldsymbol{\Theta(1)} & \boldsymbol{si\ n = 1} \\ \boldsymbol{2T(n/2) + \Theta(n)} & \boldsymbol{sinon} \end{cases}$$

Pour évaluer T(n), nous pouvons utiliser deux méthodes a savoir :

- La méthode par substitution

  $T(n) = 2T(n/2) + \Theta(n) \approx 2T(n/2) + n$

  Nous allons donc travailler avec $T(n) = 2T(n/2) + n$

  $T(n) = 2T(n/2) + n$

  $= 2^2T(n/4) + 2n$

  $= 2^3T(n/8) + 3n$

  $= 2^4T(n/16) + 4n$

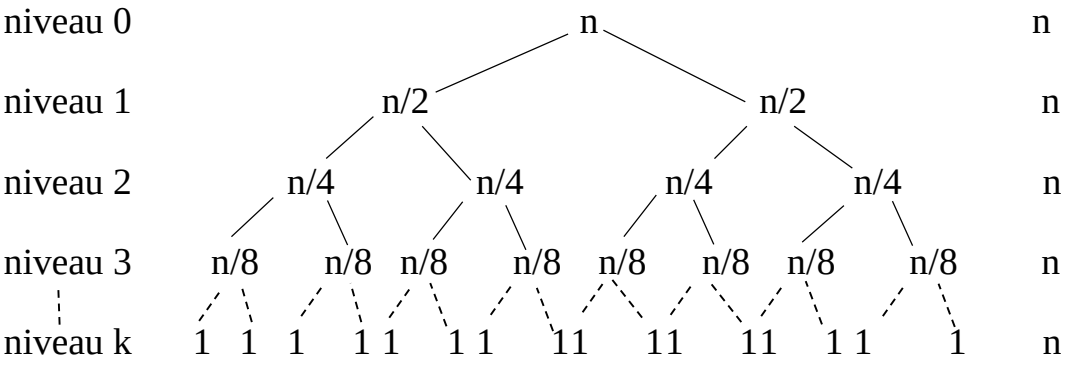
  $= 2^kT(n/2^k) + kn$

  $T(n) = kn$ or $n/2^k = 1$ d'où $T(n/2^k) = \Theta(1)$

  $n = 2^k \Rightarrow k = \log_2 n$

  $T(n) = n\log_2 n$ d'où la complexité de cet algorithme est donc $\Theta(n\log n)$

- La méthode des arbres récursif



Ici chaque niveau de l'arbre est opéré en $\Theta(n)$ et nous avons k niveaux de l'arbre.

Notre arbre récursif nous permet donc d'écrire que :

$T(n) = nk$ or $k = \log_2 n$

D'où $T(n) = n * \log_2 n$

Donc la complexité du tri est alors $\Theta(n\log n)$.

Bibliographie

https://info.blaisepascal.fr/nsi-complexite-dun-algorithme

https://www.supinfo.com/cours/2ADS/chapitres/01-notion-complexite-algorithmique

-Chapitre 02 – complexité des algorithmes récursifs :
https://www.supinfo.com/cours/2ADS/chapitres/02-notion-complexite-algorithmique

-https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes

PROPOSITION DE LA FICHE DE TD

Exercice 1

1. Le temps d'exécution d'un algorithme A est décrit par la récurrence T(n) = 2T(n/2 + 17) + n. Quelle est sa complexité ?
2. Utiliser un arbre récursif pour déterminer une bonne borne supérieure asymptotique pour la récurrence T(n) = 3T(n/2) + n. Vérifier la réponse à l'aide de la méthode de substitution.
3. Le temps d'exécution d'un algorithme A est décrit par la récurrence $T(n) = 2T(n^{1/2}) + \lg n$. Quelle est sa complexité ?
4. Utiliser un arbre récursif pour déterminer une bonne borne supérieure
5. Utiliser la méthode générale pour déterminer la complexité de la récurrence suivante : $T(n) = 2T(n/2) + n \lg n$.
6. Le temps d'exécution d'un programme nous donne la récurrence suivante : $T(n)=2T(n/4)+3\log n$.. Trouver le comportement asymptotique de $T(n)$.

Exercice 2. (Ordres de grandeurs).

On dispose d'une machine qui est capable de faire 1000000 opérations par secondes. On considère des algorithmes dont les complexités sont les suivantes : $\log(n)$, $\sqrt{n}$, n, $50 * n$, $n \log(n)$, n^2, n^3, 2^n.

1. Dessiner chacune de ces fonctions sur une échelle doublement logarithmique.
2. Quel est pour chaque algorithme, la taille des problèmes que l'on peut résoudre en une seconde ?
3. Même question si l'on dispose d'une machine 1000 fois plus rapide et de 1000s

Exercice 3

Soit la suite définie ci-dessous. On considère le problème de calculer pour l'entrée n. Soit l'algorithme naïf récursif associé :

```
int T(int n){
    if (n<=2)
        return 1;
    else
        return T(n-
1)+2*T(n-2)+T(n-3);
}
```

1. Soit A(n) le nombre d'appels récursifs à T effectués lors de l'exécution de la fonctionT(n).

Exprimez A(n) en fonction de A(n - 1), A(n - 2), A(n - 3). Qu'en déduire sur la complexitedel'algorithmenaïf?

2. Proposez un algorithme en (en supposant qu'on est dans le cadre du coût uniforme, *i.e. en ne prenant pas en compte la taille des opérandes*).

Exercice 4

On considère, pour effectuer la recherche d'un élément dans un tableau, la recherche séquentielle et la recherche dichotomique. On s'intéresse à leur complexité temporelle. Pour cela, considérer un tableau ayant mille éléments (version triée, et version non triée). Pour chaque algorithme, et pour chaque version du tableau, combien de comparaisons sont à effectuer pour ?

- Trouver un élément qui y figure ?
- Trouver un élément qui n'y figure pas ?

Quels sont les cas où le tableau est parcouru complètement et les cas où un parcours partiel est suffisant ? Conclure en donnant la complexité temporelle pour chaque algorithme.

Exercice 5

Étant donné un tableau trié d'entiers A[s..f] et deux entiers ("bornes") a <= b, on cherche s'il existe un élément A[i] du tableau tel que a <= A[i] <= b (s'il y en a plusieurs trouvez un).

Exemple : Soit le tableau A[1..5] = [3, 7, 8, 43, 556] et les bornes a = 40, b = 50. Dans ce cas, la valeur encadrée existe : c'est A[4] = 43.

Donnez (*en pseudocode*) un algorithme "*diviser pour régner*" qui résout ce problème. Expliquez brièvement. Analyser la complexité de cet algorithme.

Exercice 6

On considère deux manières de représenter ce que l'on appelle des « matrices creuses », c'est- à-dire des matrices d'entiers contenant environ 90% d'éléments nuls:

a) La matrice est représentée par un tableau à deux dimensions dont les cases contiennent les éléments.

b) La matrice est représentée par un tableau à une dimension. On ne s'intéresse qu'aux éléments de la matrice qui ne sont pas nuls. Chaque case du tableau contient un triplet (i,j,a) correspondant à l'indice de ligne, l'indice de colonne, et la valeur d'un élément non nul.

Le problème considéré consiste à calculer la somme des éléments d'une matrice. On demande d'écrire un algorithme permettant de calculer cette somme, pour chacune des deux représentations, puis de comparer leur complexité spatiale (espace mémoire occupé) et leur complexité temporelle (nombre d'opérations à effectuer).

Que peut-on conclure de cette comparaison ? Montrer qu'il existe une valeur critique du nombre d'éléments non nuls à partir de laquelle une méthode l'emporte sur l'autre.

Exercice 7

On considère un tableau à une dimension contenant des lettres majuscules. On désire compter la fréquence de chacune des 26 lettres de l'alphabet. Ecrire deux procédures qui donnent en sortie un tableau de fréquence : l'une où le tableau est parcouru 26 fois, et l'autre (*plus performante !*) où le calcul est fait en un seul parcours. On pourra supposer que l'on dispose d'une fonction auxiliaire position(lettre) qui pour chaque lettre donne sa position dans l'alphabet : position('A') = 1, …, position('Z) = 26.

Exercice 8

Démontrer le master théorème.

Exercice 9 (Manipulation des Ordres de grandeurs).

Dans cet exercice f, g, h sont des fonctions positives.

1. Montrer que si $f \in O(g)$ et $g \in \Theta(h)$ alors $f \in O(h)$.
2. 2. Montrer que si $f \in \Theta(g)$ et $g \in O(h)$ alors $f \in O(h)$.
3. 3. Montrer par un contre-exemple que si $f \in \Theta(g)$ et $g \in O(h)$ alors f n'est pas nécessairement dans $\Theta(h)$.

Exercice 10. (Calcul du maximum d'un tableau).

1. Ecrire un algorithme qui prend en paramètre un entier n et un tableau d'entiers T de taille n, et qui retourne le maximum m des entiers du tableau et l'indice k d'une occurrence de m dans le tableau.
2. Donner un énoncé qui caractérise le fait que m et k sont correct. Un tel énoncé est appelé une spécification. C'est le contrat que l'utilisateur de l'algorithme passe avec le développeur.
3. À l'aide de la spécification, écrire un invariant de boucle et montrer qu'il est correct, c'est-à-dire qu'il est vrai au début de la boucle et qu'il est conservé à chaque étape de boucle.
4. En déduire, que l'algorithme fonctionne.
5. Quel est la complexité si l'on prend comme opérations élémentaires les comparaisons.
6. Quel est la complexité si l'on prend comme opérations élémentaires les affectations.
7. Quel est la complexité si l'on prend comme opérations élémentaires les comparaisons et les affectations.

Exercice 11

1. Le temps d'exécution d'un algorithme A est décrit par la récurrence $T(n) = 2T(\frac{n}{2} + 17) + n$. Quelle est sa complexité ?
2. Utiliser un arbre récursif pour déterminer une bonne borne supérieure asymptotique pour la récurrence $T(n) = 3T(\frac{n}{2}) + n$. Vérifier la réponse à l'aide de la méthode de substitution.
3. Le temps d'exécution d'un algorithme *A* est décrit par la récurrence $T(n) = 2T(n^{\frac{1}{2}}) + \log n$. Quelle est sa complexité ?
4. Le temps d'exécution d'un programme nous donne la récurrence suivante :$T(n) = 2T(\frac{n}{4}) + 3\log n$. Trouver le comportement asymptotique de *T(n).*
5. Utiliser un arbre récursif pour déterminer une bonne borne supérieure asymptotique pour la récurrence T n) = 4T(n/2) + n. Vérifier la réponse à l'aide de la méthode générale.
6. Utiliser la méthode générale pour déterminer la complexité de la récurrence suivante : T n) = 2T n/2) + n lg n.

Exercice 12.

On rappelle que les complexités en pire cas de l'algorithme de tri-fusion et de l'algorithme de tri rapide sont respectivement en O(n log n) et en O(n^2) (tableau déjà trié). Montrer que leurs complexités en moyenne sont en O(n log n).