Table de matière

Partie I

COURS

Introduction

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 une ensemble de valeurs est un algorithme autrement dit une séquence d'étapes de calcul qui transforme l'entrée en sortie. Ici il ne s'agit pas seulement d'écrire des algorithmes mais d'écrire des algorithmes efficaces et performants ; ainsi entre en jeux la notion de complexité d'un algorithme qui se définie par 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'efficacité d'un algorithme est fonction de la taille de données et du temps de calcul ; ainsi, on distingue généralement la complexité temporelle (liée au temps d'exécution des opérations) et la complexité spatiale (liée à l'espace mémoire qu'occupent les opérations et variables) ; mais le plus souvent lorsqu'on parle de complexité, il s'agit de la complexité temporelle ; et c'est cette complexité temporelle que nous étudierons tout au long de ce cours vu sa définition plus haut.

I Généralités sur la complexité des algorithmes

I.1 Type de complexité

A un même problème, différentes solutions algorithmiques peuvent être proposées. La première qualité d'un algorithme est sa terminaison c'est-à-dire qu'il n'admet aucune instance pour laquelle l'exécution entre en boucle infinie. Un autre critère qui permet de comparer les algorithmes est celui de la faible utilisation de deux ressources :

- Le temps : Il sera évalué en considérant le nombre d'opérations élémentaires devant être exécutées.
- L'espace : IL correspond à l'espace mémoire qui sera alloué pour l'exécution d'un algorithme.

De ce fait on distingue donc la complexité temporelle et la complexité spaciale Mais de nos jours, les machines sont de plus en plus rapides et puissantes, avec de plus grandes mémoires...mais la technologie évolue plus rapidement en terme d'espace mémoire que de rapidité:contrairement aux machines ayant des grandes mémoires,les machines rapides sont très rares et avec de coûts très élevés.D'ou l'interèt de s'atarder sur la complexité temporelle. De ce fait, pour pouvoir certifier quel algorithme sera le plus efficace, nous devrons être capable de dire lequel de ces algorithmes est le plus rapide en terme de temps et pour ce faire, rien de mieux que d'évaluer la vitesse de chacun en calculant sa complexité en temps. La durée d'exécution d'une opération étant fonction de la puissance de la machine, nous évaluerons le temps d'exécution d'un algorithme en fonction du nombre élémentaire d'opération de cet algorithme. Le temps d'exécution peut dépendre de la disposition des données (par exemple un algorithme de recherche séquentielle en utilisant la structure itérative tantque d'une valeur dans un tableau s'arrête dès qu'il a trouvé la première occurrence de cette valeur. Si la valeur se trouve toujours au début du tableau (en 1ère position), le temps d'exécution est plus faible que si elle se trouve en milieu du tableau de même plus que si elle se trouve toujours à la fin (en dernière position)). Nous comprenons par-là que l'évaluation de la complexité d'un algorithme peut aboutir sur 03 cas de figures particulières.

I.2 Temps de calcul

Le temps de calcul d'un programme dépend de plusieurs éléments :

- la quantité de données bien sûr
- mais aussi de leur encodage
- de la qualité du code engendré par le compilateur
- de la nature et la rapidité des instructions du langage
- de la qualité de la programmation
- et de l'efficacité de l'algorithme

Nous ne voulons pas mésurer le temps de calcul par rapport à toutes ces variables. Mais nous cherchons à calculer la complexité des algorithmes qui ne dépendra ni de l'ordinateur, ni du langage utilisé, ni du programmeur, ni de l'implémentation. Pour cela, nous allons nous mettre dans le cas où nous utilisons un ordinateur RAM (Random Access Machine) :

- ordinateur idéalisé
- mémoire infinie
- accès à la mémoire en temps constant
- généralement à processeur unique (pas d'opérations simultanées)

Pour connaître le temps de calcul, nous choisissons une opération fondamentale et nous calculons le nombre d'opérations fondamentales exécutées par l'algorithme.

I.3 Opération fondamentale

C'est la nature du problème qui fait que certaines opérations deviennent plus fondamentales que d'autres dans un algorithme. Par exemple :

Probleme	Opération fondamentale
Recherche d'un élement dans une liste	Comparaison
Tri d'une liste,d'un fichier	comparaison,déplacement
Multiplication des matrices réelles	multuplication,addition
Addition des entiers binaires	Operations binaires

Table 1: Opérations fondamentales

I.4 Mésure de complexité

Il existe trois mesures différentes, la complexité dans le meilleur des cas, la complexité en moyenne et la complexité dans le pire des cas. Mais la troisième est beaucoup plus employée que les autres.
Soit A un algorithme, n un entier, D_n l'ensemble des entrées de taille n et une entrée $d \in D_n$. Posons : $cout_A(d)$ le nombre d'opérations fondamentales effectuées par A avec l'entrée d .

I.4.1 Complexité dans le meilleur des cas

C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille n, c'est-à-dire le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur ce jeu de données. C'est également un minorant possible du temps d'exécution pour toutes les entrées possibles d'une même taille. Le meilleur des cas correspond à la disposition des données en entrée pour laquelle l'algorithme s'exécute le plus rapidement possible : c'est la situation la plus favorable. Il correspond par exemple à la recherche d'un élément situé à la première position d'une liste. En clair, il s'agit du nombre minimal d'instructions élémentaires sur l'ensemble des données de taille n.
Elle est obtenue par : $Min_A(n) = min\{cout_A(d) \mid d \in D_n\}$.

I.4.2 Complexité dans le moyen des cas

Aussi appelée complexité attendue, c'est une évaluation du temps d'exécution moyen portant sur toutes les entrées possibles d'une même taille supposées équiprobables. Elle diffère de celle dans le pire des cas en considérant non le nombre maximal d'instructions élémentaires mais la moyenne du nombre d'instructions élémentaires sur l'ensemble des entrées de taille n. Pour la définir, il faut disposer pour tout n d'une mesure de probabilité p sur l'ensemble des données de taille n; cette mesure ne fait souvent qu'approximer l'espace réel des données, car il est très dur de proposer un modèle des données "réelles" -on suppose par exemple souvent que toutes les données de même taille sont équiprobables, ce qui est peu réaliste. Autrement dit, c'est la moyenne du nombre d'instructions élémentaires sur l'ensemble des entrées de taille n. La complexité au moyen des cas est plus compréhensible en pratique qu'en théorie.
C'est le temps d'exécution moyen ou attendu d'un algorithme. La difficulté pour ce cas est de définir une entrée "moyenne" pour un problème particulier.
Elle est obtenue par: $Moy_A(n) = \sum p(d).cout_A(d)$ avec $p(d)$une loi de probabilité sur ses entrées
Il reste a définir p(d).Une hypothèse est que toutes les entrées ayant une

tailles données sont équiprobables.
D'autres hypothèses se basent sur l'analyse probaliste.

I.4.3 Complexité dans le pire des cas

C'est un majorant du temps d'exécution pour toutes les entrées possibles d'une même taille, c'est-à-dire le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée. Le pire des cas correspond à la disposition des données en entrée pour laquelle l'algorithme s'exécute le plus lentement possible : c'est la situation la plus défavorable. Il correspond par exemple à la recherche d'un élément dans une liste alors qu'il n'y figure pas, ou encore le tri fusion d'une liste déjà triée. Autrement dit, c'est la fonction qui a tout entier n associe le nombre d'instructions élémentaires max exécutées par l'algorithme sur des entrées de taille n.
Elle est donnée par : $Max_A(n) = max[cout_A(d)|d \in D_n]$
C'est cette complexité qui est généralement calculée car c'est une borne supérieure du temps d'exécution associé à une entrée quelconque.

I.5 Paramètre de complexité

Le paramètre de la complexité est la donnée du traitement qui va (le plus) faire varier le temps d'exécution de l'algorithme. Par exemple, en fonction du problème, les entrées et leur taille peuvent être :

- des éléments : le nombre d'éléments ;
- des nombres : nombre de bits nécessaires à la représentation de ceux-là ;
- des polynômes : le degré, le nombre de coefficients non nuls ;
- des matrices m $\times$ n : max(m,n), m.n, m + n ;
- des graphes : nombre de sommets, nombre d'arcs, produit des deux ;
- des listes, tableaux, fichiers : nombre de cases, d'éléments ;
- des mots : leur longueur.

Example I.1. *Exemple de la fonction factorielle*

Variables :
n : *entier naturel ;*
resultat : *entier naturel;*

resulat $\leftarrow 1$;
pour i *de* 1 *à* n **faire**
| *resulat* $\leftarrow resultat * i$;
fin
retourne *resultat;*

Algorithme 1 : Fonction factorielle

Ici,le paramètre de complexité c'est la valeur de l'entier n

Example I.2. *multiplier tous les éléments d'un tableau par un nombre donné*

Entrées : **tab** : tableau d'entiers ;
x : entier naturel;
Variables : **i** : entier naturel ;
pour i *de* 1 *à* n **faire**
| $tab[i] \leftarrow tab[i] * x$
fin

Algorithme 2 : Algorithme de multiplication des éléments d'un tableau par un nombre

Le Paramètre de complexité correspond à n qui est la taille du tableau

II Notations mathematiques

Quand nous calculons la complexité d'un algorithme, nous ne calculons pas sa complexité exacte, mais son ordre de grandeur. Pour se faire, nous avons besoin de notations asymptotiques.

II.1 Notation de Landau ou O

Soient f et g deux fonctions. f, g : $\mathbb{N} \rightarrow$. On note $f(n) = O(g(n))$ lorsque: $\exists c > 0, n_0 \in \mathbb{N} | \forall n \geq n_0, f(n) \leq c.g(n)$.

Intuitivement, cela signifie que f est inférieur a g, a une constante multiplicative près pour les instances (données) de tailles suffisamment grandes. Nous lisons ≪ grand o ≫ de g pour $O(g)$

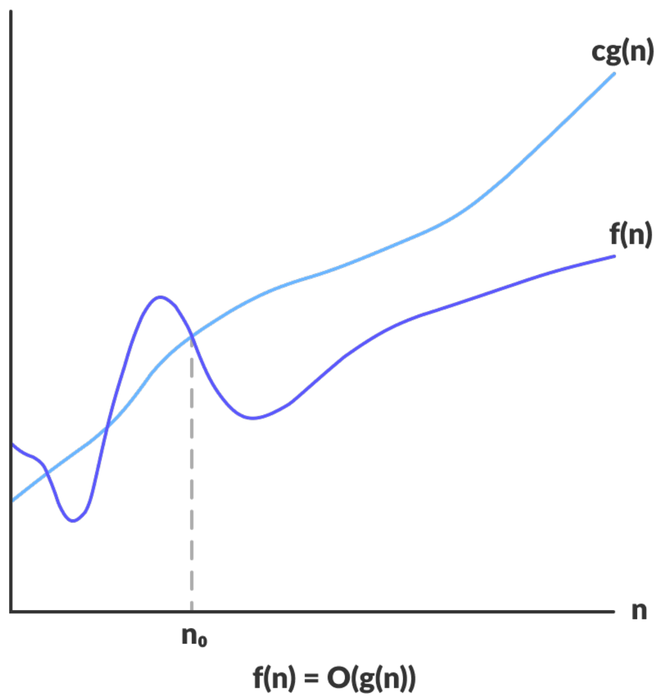

Figure 1: Notation O

Example II.1. *Soient les fonctions suivantes :* $f(n) = 2n^2+n+5$ *et* $g(n) = n^2$ *Dire que* $f(n) = O(g(n))$ *signifie qu'à un certain rang* n_0 *donné,* $\exists c > 0 | f(n) < c.g(n) \forall n \geq n_0$ *c tel que f(n) ¡ Supposons donc que* $n_0 = 5$ *et trouvons la constante c :* $n_0 = 5 \Rightarrow n > 5$
donc $2n^2 + n + 5 < c.n^2 \Rightarrow 2(5)^2 + 5 + 5 < c.(5)^2$
$\Leftrightarrow 60 < 25.c$
$\Leftrightarrow 60/25 < c$
Donc nous pouvons prendre comme valeur de c toute constante réelle positive strictement supérieur à 2.4.
Interprétation *: cet exemple peut être interprété comme suit : A partir du rang* $5, (n_0 = 5 et donc n > 5)$ *, toute valeur de c strictement supérieur à 2.4 vérifie l'inégalité* $f(n) < c.g(n)$*. On peut donc conclure que* $f(n) = O(g(n))$

Propriete 1.

Soit g une fonction de $\mathbb{R}^+$ dans $\mathbb{R}^+$. Nous avons :
$\forall d \in \mathbb{R}^{+*}, O(d.g) = O(g)$

Propriete 2.
Soient f et g deux fonctions de $\mathbb{R}^+$ dans $\mathbb{R}^+$. L'addition est effectuée en prenant la valeur maximale :
$O(g) + O(h) = O(g + h) = O(max(g, h))$

Propriete 3.
Soient f et g deux fonctions de $\mathbb{R}^+$ dans $\mathbb{R}^+$. Nous notons $O(g).O(h)$ l'ensemble $\{f.r|f \in O(g), r \in O(h)\}$. Nous avons alors :$O(g).O(h) = O(gh)$

Propriete 4.
Soient trois fonctions f , g et h de $\mathbb{R}^+$ dans $\mathbb{R}^+$ telles que $f \in O(g)$ et $g \in O(h)$, alors $f \in O(h)$.

Propriete 5.
Soient deux fonctions f et g de $\mathbb{R}^+$ dans $\mathbb{R}^+$ telles que $f \in O(g)$ alors $O(f) \subset O(g)$.

Theoreme 1.
Soit $n \in \mathbb{N}$. Nous avons :

$$\boxed{O(1) \subset O(log(n)) \subset O(\sqrt{n}) \subset O(n) \subset O(n\ log(n)) \subset O(n^2) \subset O(n^3) \subset ... \subset O(n^{10}) \subset ... \subset O(2^n) \subset O(n!)}$$

II.2 Notation en o

La notation en o est utilisée pour indiquer que la borne supérieure n'est pas asymptotiquement approchée. Soit g une fonction définie de $\mathbb{R}_+ \rightarrow \mathbb{R}_+$. L'ensemble $o(g)$ est défini ainsi $o(g) = \{f : \mathbb{R}_+ \rightarrow \mathbb{R}_+|\forall c > 0 \exists n_0 \in \mathbb{N}|\forall n \geq n_0, o \leq f(n) < c.g(n)\}$

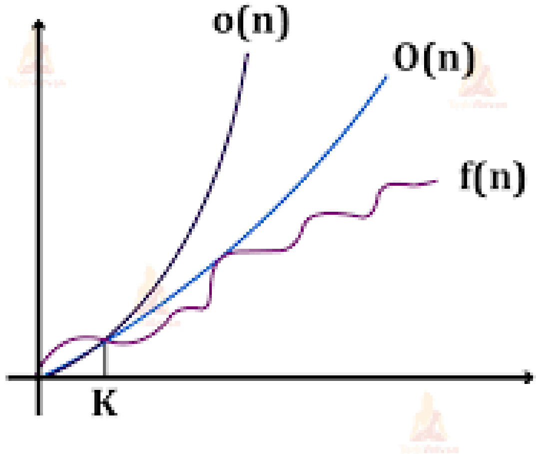

Figure 2: Comparaison de o et O

Nous disons pour $fo(g)$ que f est un petit o de g et que f est négligeable devant g.

Propriete 6.
Soit f une fonction définie de $\mathbb{R}^+$ dans $\mathbb{R}^+$. Nous avons :$o(f)+o(f)=o(f)$

II.3 Notation en Ω

La notation Ω est utilisée pour indiquer la borne inférieure asymptotique. Soit g une fonction définie sur une partie de $\mathbb{R}$. L'ensemble $\Omega(g)$ est définie ainsi :$\Omega(g)=\{f fonction\ définie\ sur\ une\ partie\ de\ \mathbb{R}|\exists n_0 \in \mathbb{N}, \exists c \geq 0|\forall n \geq n_0, 0 \leq c.g(n) \leq f(n)\}$.

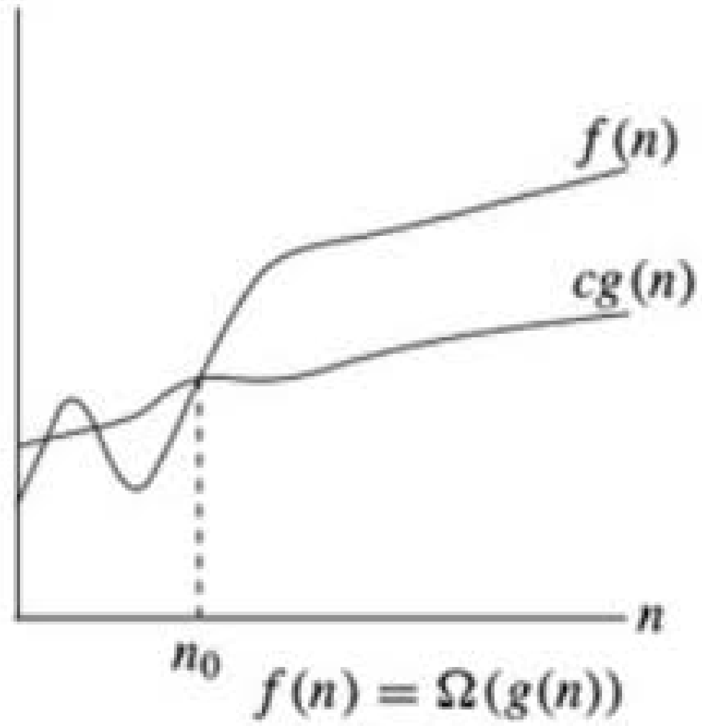

Figure 3: Notation Ω

Example II.2. *Soient les fonctions suivantes :* $f(n) = 2n^2+n+5$ *et* $g(n) = n^2$ *Dire que* $f(n) = \Omega(g(n))$ *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* $n_0 = 5$ *et trouvons la constante c :* $n_0 = 5 \Rightarrow n5$
donc $2n^2 + n + 5 \geq c.n^2 \Leftrightarrow 2(5)^2 + 5 + 5 \geq c.(5)^2 \Leftrightarrow 6025.c \Leftrightarrow 60/25c$

Donc nous pouvons prendre comme valeur de c 2.4 *ou toute constante réelle positive inférieure ou égal à* 2.4.

Interprétation *: cet exemple peut être interprété comme suit : A partir du rang 5 (*$n_0 = 5$ *et donc* $n \geq 5$*), toute valeur de c positive inférieur ou égale à 2.4 vérifie l'inégalité* $f(n) \geq c.g(n)$. *On peut donc conclure que* $f(n) = \Omega(g(n))$

Propriete 7.
Soient f et g des fonctions définies sur une partie des nombres réels. Une autre manière de définir $\Omega(g)$ *est :* $f \in \Omega(g)$*si et seulement si* $g \in O(f)$.

II.4 Notation ω

La notation ω est utilisée pour indiquer que la borne inférieure n'est pas asymptotiquement approchée. Soit g une fonction définie sur une partie des nombres réels. L'ensemble $\omega(g)$ est défini ainsi :
$\omega(g) = \{f\ fonction\ d\acute{e}finie\ sur\ une\ partie\ de\ \mathbb{R} | \forall\ c > 0, \exists\ n_0 \in \mathbb{N} tels\ que\ \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$
Soient f et g des fonctions définies sur une partie des nombres réels. Une autre manière de définir $\omega(g)$ est : $f(n) \in \omega(g(n)) si\ et\ seulement\ si\ g(n) \in o(f(n))$

II.5 Notation en Θ

La notation Θ sert à minorer et majorer une fonction, à des facteurs constants près. Elle permet en réalité de donner une borne asymptotique pour une fonction donnée. Soit g une fonction définie sur une partie des nombres réels. L'ensemble $\Theta(g)$ Est définie ainsi : $\Theta(g) = \{f\ fonction\ d\acute{e}finie\ sur\ une\ partie\ de\ \mathbb{R} | \exists c_1 > 0 et c_2 > 0, \exists n_0 \in \mathbb{N} | \forall n \geq n_0, 0 \leq g(n) \leq f(n) \leq g(n)\}$

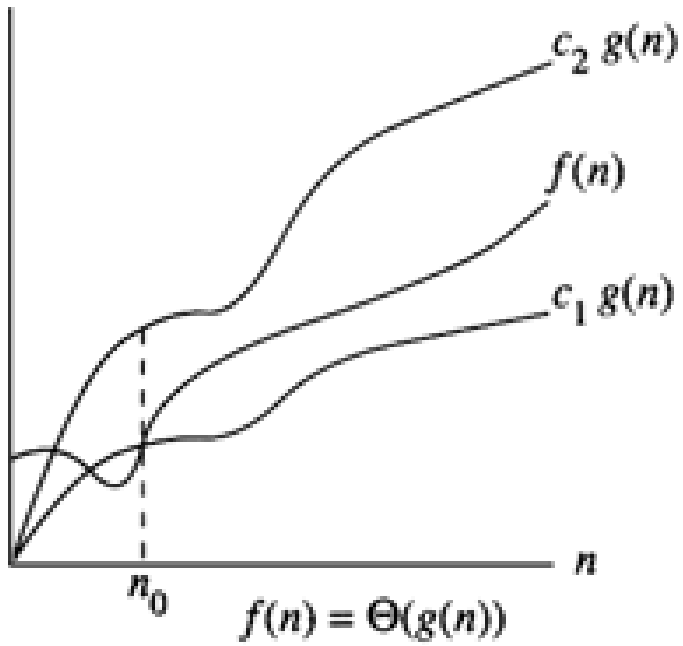

Figure 4: Notation Θ

Example II.3. *A titre d'exemple, considérons nos mêmes fonctions* f *et* g *précédentes :* $f(n) = 2n^2 + n + 5$ *et* $g(n) = n^2$ *Nous avons dans le premier exemple montré que* $f(n) = O(g(n))$ *et dans le deuxième montré que* $f(n) = \Omega(g(n))$*. Ceci permet donc de conclure que* $f(n) = \Theta(g(n))$

Propriete 8.
Soient f *et* g *deux fonctions définies sur une partie des nombres réels.* $f \in \Theta(g)$*si et seulement si* $g \in \Theta(f)$ *.*

Theoreme 2.

Pour deux fonctions quelconques f *et* g *définies sur une partie des nombres réels :* $f \in \Theta(g)$*si et seulement si* $f \in O(g)$*et* $f \in \Omega(g)$ *.*

II.6 Notation en $\sim$

Soient f et g deux fonctions définies sur une partie des nombres réels. f et g sont équivalentes si $f(x) = g(x)(1 + \epsilon\ (x))$ *avec* $\lim\limits_{x \to \infty} \epsilon(x) = 0$. La notation est : $f \sim g$

II.7 Récapitulatif des comparaisons de fonctions

L'ensemble de ces notation est repartit comme sur la figure suivante:

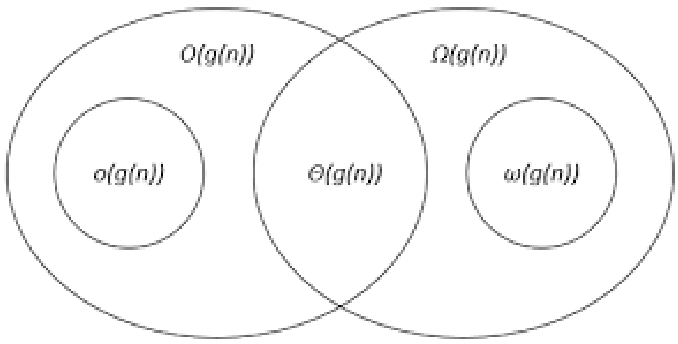

Figure 5: Ensemble des notations

Grâce à ces différentes notions, nous pouvons effectuer des comparaisons de fonctions à l'infini, autrement dit avoir le comportement asymptotique des fonctions.

- f ne croît pas plus vite que g : $f(n) = O(g(n))$

- f croît plus lentement que g :$f(n) = o(g(n))$
- f ne croît pas plus lentement que g :$f(n) = \Omega(g(n))$
- f croît plus vite que g :$f(n) = \omega(g(n))$
- f et g ont des croissances comparables :$f(n) = \Theta(g(n))$
- f et g sont asymptotiquement identiques :$f \sim g$

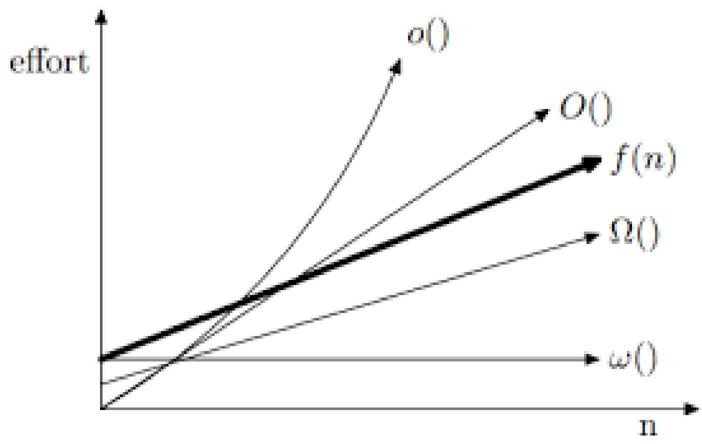

Figure 6: graphe regroupant l'ensemble des notations

Mais attention, deux fonctions ne sont pas nécessairement comparables asymptotiquement (c'est-à-dire que nous n'avons pas toujours $f = O(g)$ ou $f \in \Omega(g)$.

Par exemple n et n^{1+sinn} ne peuvent pas être comparées asymptotiquement car 1sin n varie entre 0 et 2.

Propriete 9.
Soient f et g deux fonctions. Voici une synthèse du calcul de $\lim\limits_{x\to\infty} \frac{f(x)}{g(x)}$

- $\lim\limits_{x\to\infty} \frac{f(x)}{g(x)} = 0$ *implique* $f \in O(g)$ *et* $g \notin O(f)$
- $\lim\limits_{x\to\infty} \frac{f(x)}{g(x)} = 1$ *implique* $f \sim g$
- $\lim\limits_{x\to\infty} \frac{f(x)}{g(x)} = c \neq 0$ *implique* $f \in O(g),\ g \in O(f),\ f \in \Theta(g)$ *et* $g \in \Theta(f)$
- $\lim\limits_{x\to\infty} \frac{f(x)}{g(x)} = \infty$ *implique* $f \notin O(g)$ *et* $g \in O(f)$

III Méthode d'évaluation de la complexité des algorithmes

Pour déterminer la complexité d'un algorithme (complexité en temps) généralement on déterminer le temps d'exécution de chaque instruction de de l'algorithme et à la fin, on fait la somme de tous ces temps afin de trouver donc la complexité totale en temps voulu. Selon le type d'algorithme (Séquentiel, Itératif, récursif) on doit savoir comment procéder. Nous allons donc nous appuyer sur chacun de ces type d'algorithmes pour ainsi évaluer leurs complexités.

III.1 Algorithme purement séquentiel (sans boucle ou itération)

Pour évaluer la complexité d'un algorithme en générale, on pondère chaque instruction par un coût et à la fin la somme de tous ces coûts donne la complexité en temps total.

Example III.1. *Soit un modèle d'algorithme séquentiel suivant :*

Entres : $\boldsymbol{n}$ *:entier ;*
si C ; // C_1
alors
    J ; // C_2
sinon
    k ; // C_3
fin

Algorithme 3 : Algorithme séquentiel

$T(n) = C_1 + max(C2, C3)$

Ici, la complexité que l'on calcule sera la plus grande possible et on dit pour cela que c'est le pire des cas. Pour le meilleur des cas, on aurai pu tout simplement mettre à la place de la fonction max(), la fonction min().

Example III.2. *Exemple concret:*

Entrées : ***R*** *:entier ;*
p *:entier ;*
g *:entier ;*
$R \leftarrow nmod2$; // C_1
si $R \neq 0$; // C_2
alors
  $p \neq n/2$; // C_3
  $g \neq p - 1$; // C_4
  Ecrire : *la moitie de n diminuée de 1 est g*
  ; // C_5
sinon
  Ecrire : *Nombre impair: Rien a faire*
  ; // C_6
fin

Algorithme 4 : Algorithme parite

Comme tout en haut, pour commencer à examiner la complexité de cet algorithme, on impose un cout à chaque instruction. On a ici $C_1, C_2, C_3, C_4, C_5, C_6$ *. Ensuite, on calcule exactement comment on l'a vu plus haut:*
$T(n) = C_1 + C_2 + max(C_3 + C_4 + C_5, C_6)$*(Au pire des cas) Qui est une constante:*$T(n) = \Theta(1)$

III.2 Algorithme itératif

III.2.1 Algorithme itératif simple

Example III.3. *Considérons un modèle d'algorithme itératif suivant avec la boucle pour:*

Entrées : ***n*** *:entier ;*
pour *i de p a n ;* // C_1
faire
  Ecrire : *Bonjour*
  ; // C_1
fin

Algorithme 5 : Dire-n-Bonjour

Comme pour le cas simplement séquentiel, on va pondérer les instructions par les (cout) et on a . La simplicité que l'on a ici avec la boucle 'pour' est que le nombre d'itération est connu à l'avance.
$T(n) = (n - p + 1)(C_1 + C_2)$
$T(n) = nC_1 + nC_2 + (-p + 1)(C_1 + C_2)$
$T(n) = n(C_1 + C_2) + (-p + 1)(C_1 + C_2)$

$T(n) = An + B$ avec $A = C_1 + C_2$ et $B = (-p+1)(C_1 + C_2)$
Donc $T(n) = O(n)$ d'apres la notation de Landau

La détermination de la complexité des cet algorithme est simple nous l'avons déjà dit car le nombre d'itération est connu à l'avance. Voyons le cas où cela n'est pas connu.

III.2.2 Algorithme itératif dans lequel le nombre d'itération n'est pas connu à l'avance

Example III.4. *Considérons un modèle d'algorithme itératif suivant:*

Ecrire : *preparation du couscous*
; // C_1
$Recuperation\_des\_ingredient()$; // C_2
$Mettre\_le\_couscous\_au\_feu()$; // C_3
tant que *Il y a des boules ;* // C_4
faire
| *Tourner le couscous ;* // C_5
fin

Algorithme 6 : Préparation Du couscous

On suppose que l'ensemble de fonction utiliser dans cet algorithme s'exécute en temps constant. On pondère à nouveau les instruction et on a C_1, C_2, C_3, C_4, C_5
Si on essaye d'évaluer la complexité, on aura :
$T(n) = C_1 + C_2 + C_3 + kC_4 + (k-1)C_5$
La complexité de cette algorithme dépend de la boucle 'tantque' et plus k est grand, le temps d'execution sera grand.on aura donc $T(n) = O(k)$

Dans le cas où l'exécution d'un algorithme fait appel à des fonctions externes alors l'évaluation de la complexité de l'algorithme ne passe pas l'évaluation de la complexité de ces fonctions externes premièrement avant l'algorithme.

III.2.3 Analyse des algorithmes itératifs plus complexe

Example III.5. *calcul de la complexité de l'algorithme du Tri Sélection*

```
pour i de 1 a n − 1 ;                         // C1  n − 1 fois
 faire
    pour j de i + 1 a n ;                     // C2  n − i fois
     faire
        si T(j) < T(i) alors
          | Permuter(T(j), T(i))
        fin
    fin
fin
```

Algorithme 7 : Tri Sélection

$T(n) = (n-1)C_1 + (\frac{n(n-1)}{2})C_2 + (\frac{n(n-1)}{2} - 1)C_3 + (\frac{n(n-1)}{2} - 1)\ C_4$
$= nC_1 - C_1 + \frac{n^2}{2}C_2 - \frac{n}{2}C_2 + \frac{n^2}{2}C_3 - \frac{n}{2}C_3 - C_3 + \frac{n^2}{2}C_4 - \frac{n}{2}C_4 - C_4$
$= n^2(\frac{1}{2}C_2 + \frac{1}{2}C_3 + \frac{1}{2}C_4) + n(C_1 - \frac{1}{2}C_2 - \frac{1}{2}C_3 - \frac{1}{2}C_4) - C_1 - C_3 - C_4$

$$An^2 + Bn + C \ avec \begin{cases} \frac{1}{2}C_2 + \frac{1}{2}C_3 + \frac{1}{2}C_4 \\ C_1 - \frac{1}{2}C_2 - \frac{1}{2}C_3 - \frac{1}{2}C_4 \\ C_1 - C_3 - C_4 \end{cases}$$

Donc on peut conclure avec la notation de landau que $T(n) = O(n^2)$

Example III.6. *calcul de la complexité de l'algorithme du Tri Insertion*

```
pour j de 2 a n faire
    cle ← T(j)
    i ← j − 1
    tant que i > 0 et T(i) > cle faire
        T(i + 1) ← T(i)
        i ← i − 1
    fin
    T(i + 1) ← cle
fin
```

Algorithme 8 : Tri Insertion

$T(n) = nC_1 + (n-1)C_2 + (n-1)C_3 + C_4 \sum_{i=2}^{n} t_j + C_5 \sum_{i=2}^{n}(t_j - 1) + C_6 \sum_{i=2}^{n}(t_j - 1) + C_7(n-1)$

Dans le meilleur des cas on a :

$T(n) = nC_1 + (n-1)C_2 + (n-1)C_3 + C_7(n-1) = nC_1 + nC_2 - C_2 + nC_3 - C_3 + nC_7 - C_7$

$= n(C_1+C_2+C_3+C_7)-C_2-C_3-C_7 = An+B$ *avec* $\begin{cases} A = C_1 + C_2 + C_3 + C_7 \\ B = -C_2 - C_3 - C_7 \end{cases}$

d'ou $T(n) = O(n)$ *dans le meilleur des cas.*

Dans le pire des cas on a :

Ici le développement reste le même que pour le trie sélection et on obtient $T(n) = O(n^2)$

III.3 Algorithme récursif

Un algorithme qui au cour de son exécution fait appel à elle-même, on dit que c'est un algorithme récursif. Le terme récursif ici employé fait à la récurrence. C'est donc une façon de programmer qui s'apparente à un fonctionnement très mathématique très simple et en moins de code c'est une façon de simplifier les choses (la programmation). Il se programme par la donnée de la condition d'arrêt et de l'appel de récurrence. Son temps d'exécution est exprimé sur une équation de récurrence dont la résolution donnera le temps finale d'exécution de l'algorithme.

Example III.7. *exemple de quelque algorithme récursif et de l'expression de leur équation respectif de cout*

$factoriel(n)$

si $n = 0$ **alors**
| $factoriel \leftarrow 0$
sinon
| $factoriel \leftarrow n * factoriel(n-1)$
fin

Algorithme 9 : Factoriel d'un entier

$$T(n) = \begin{cases} O(1) \\ T(n- \quad 1) \end{cases}$$

$fibonacci(n)$

si $n = 0$ **alors**
  $fibonacci \leftarrow 0$
sinon
  si $n = 1$ **alors**
    $fibonacci \leftarrow 1$
  sinon
    $fibonacci \leftarrow fibonacci(n-1) + fibonacci(n-2)$
  fin
fin

Algorithme 10 : La suite de Fibonacci

$T(n) = T(n-1) + T(n-2)$

$combinaison(k, n)$

si $k = 0$ *ou* $k = n$ **alors**
  $combinaison \leftarrow 1$
sinon
  $combinaison \leftarrow combinaison(k-1, n) + combinaison(k-1, n-1)$
fin

Algorithme 11 : combinaison $\binom{n}{k} = \frac{n!}{p!(n-p)!}$

$T(k, n) = T(k, n-1) + T(k-1, n-1)$

$pgcd(a, b)$

si $b = 0$ **alors**
  $pgcd \leftarrow a$
sinon
  $pgcd \leftarrow pgcd(a, a \ mod(b))$
fin

Algorithme 12 : pgcd (plus petit diviseur commun)

$T(a, b) = T(\frac{n}{2}) + c$

Résolution des équations récurrente des temps d'execution des algorithmes dans le but de trouver l'expression final de leur complexité temporelle.

III.3.1 Utilisation de la méthode par substitution

La substitution comme son nom l'indique, permet de résoudre les équation récurrence en remplaçant chaque fois un rang supérieur par un rang inferieur jusqu'à ce que l'on arrive au cas trivial et on déduit la complexité.

exemple: $T(n) = T(n-1) + C$

Ici nous avons à faire à l'expression de la complexité de la fonction factoriel. De ce fait on fera les substitutions jusqu'au cas trivial qui est n = 0. On aura donc :

$T(n) = T(n-1) + C$
$T(n-1) = T(n-2) + C$
$T(n-2) = T(n-3) + C$
$T(n-3) = T(n-4) + C$
$T(n-4) = T(n-5) + C$
...
$T(n-k) = T(n-k-1) + Caveck < n$
...
$T(1) = T(0) + C$

Une fois ayant fait cette décente récursive on fait une remonte en remplaçant tour à tous les inconnues de chaque équation :

$T(n) = T(n-2) + C + C$
$T(n) = T(n-3) + C + C + C$
$T(n) = T(n-4) + C + C + C + C$
$T(n) = T(n-5) + C + C + C + C + C$
...
$T(n) = T(0) + (C + C + C + C + +C)_{(nfois)}$
$T(n) = 1 + nC$

donc au final on aura $T(n) = O(n)$

Cette méthode devient très fastidieuse quand l'équation devient de plus en plus compliqué. C'est pour cela que l'on utilisera la méthode de l'arbre.

III.3.2 Méthode de l'arbre récursif

C'est une méthode qui nous permettra de déterminer plus rapidement le résultat de l'équation récursive dont les nœuds sont les tailles de chaque sous problèmes et chaque niveau de l'arbre a un cout.

Exemple : $T(n) = 2T(\frac{n}{2}) + n$

Sous la résolution de l'arbre de cette équation, on aura :

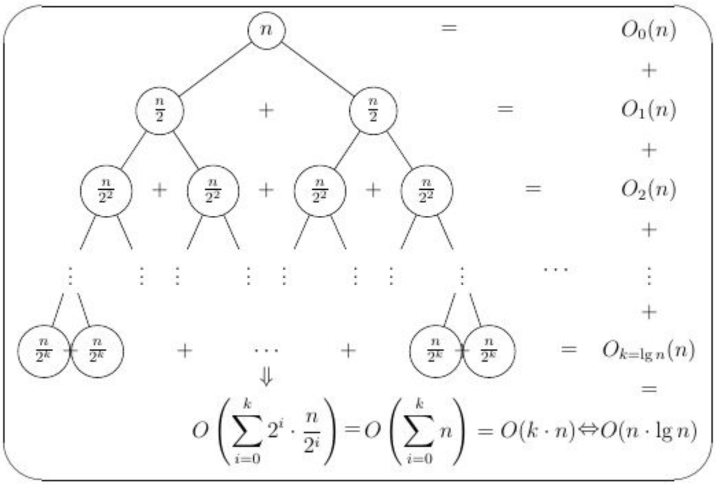

Figure 7: résolution par arbre récursif

III.3.3 Méthode par induction

La méthode par induction est une méthode mathématique de résolution d'équation de récurrence. C'est-à-dire on suppose la récurrence vraie pour un certain rang et on montre qu'il est vrai pour un certain rang suivant (supérieur). Donc ici on va anticiper la solution de l'équation et démontre qu'effectivement c'est ça la complexité. Pour cette méthode il faut de cette faite au préalable avoir une aperçue des complexités de certaines formes de complexité de certaines forme d'équation récursive.

Considérons l'exemple de l'équation de coût du pgcd : $T(n) = T(\frac{n}{2}) + C$

Ici comme nous avons étudié pas mal de complexité et nous reconnaissons la forme de cette équation, on va tous simplement suppose que
$T(n) = O(log(n))$

On va procéder ici par la définition de landau (la définition de grand O.)

cherchons $c > 0 \in \mathbb{R}, n_0 \in \mathbb{N} | \forall n \geq n_0, T(n) \leq clog(n)$.

supposons que cette equation est vrai au rang $\frac{n}{2}$ c' est a dire $T(\frac{n}{2}) \leq clog\frac{n}{2}$ et montrons qu' il est vrai au rang n.

$T(n/2) < clogn/2$

$=> T(n) < clogn/2 + C$

$=> T(n) < clogn - clog2 + C$

$=> T(n) < clogn - c + C.$

prendre $c = 2$ et $n > 2$ donc $T(n) = O(logn)$

III.3.4 Le master théorème

Théorème de la méthode générale
Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ définie pour les entiers non négatifs par la récurrence :
$T(n) = aT(\frac{n}{b}) + f(n)$, O'u l'on interprète $\frac{n}{b}$ comme signifiant $[\frac{n}{b}]$.
$T(n)$ peut alors être bornée asymptotiquement de la façon suivante :

1. Si $f(n) = O(n^{(log_b a)-\epsilon})$ pour une certaine constante $\epsilon > 0$, alors $T(n) = \theta(n^{log_b a})$.

2. Si $f(n) = \theta(n^{log_b a})$, alors $T(n) = \theta(n^{log_b a} log_2 n)$.

3. Si $f(n) = \Omega(n^{(log_b a)+\epsilon})$ pour une certaine constante $\epsilon > 0$, et si $af(\frac{n}{b}) \geq cf(n)$ pour une certaine constante $c < 1$ et pour tout n suffisamment grand, alors $T(n) = (f(n))$.

Example III.8. *exemple d'utilisation du master théorème*
Pour utiliser la méthode générale, on se contente de déterminer le cas du théorème général qui s'applique (s'il y en a un) et ensuite on écrit la réponse. Ainsi considérons la récurrence suivante : $T(n) = 9T(\frac{n}{3}) + n$, *Pour cette récurrence, on a :* $a = 9$, $b = 3$ *et* $f(n) = n$ *; on a donc* $n^{log_b a} = n^{log_3 9} = \theta(n^2)$. *Puisque f(n)* $= O(n^{log_3 9-\epsilon})$, *ou* $\epsilon = 1$, *on peut appliquer le cas* 1 *du théorème général et dire que la solution est*
$T(n) = \theta(n^2)$

Considérons à présent la récurrence suivante : $T(n) = T(\frac{2n}{3}) + 1$, *ou* $a = 1$, $b = \frac{3}{2}$, $f(n) = 1$ *et* $n^{log_b a} = n^{log_{\frac{3}{2}} 1} = n^0 = 1$ *On est dans le cas* 2 *puisque* $f(n) = \theta(n^{log_b a}) = \theta(1)$, *et donc la solution de la récurrence est* $T(n) = \theta(log_2 n)$.

Il y a des complexités d'ordre exponentielle qui ne preuve tous simplement pas facilement être résolue avec ces méthodes évoquées plus haut. Si on prend l'exemple de la complexité de l'algorithme de la suite de Fibonacci il est difficile voire impossible d'utiliser la substitution. Pour résoudre cela on applique des méthodes purement mathématique.

III.3.5 Résolution mathématique

Une équation homogène récurrente linéaire à coefficient constant de degré d, est une équation de la forme :

$a_n = C_1 a_{n-1} + C_2 a_{n-2} + C_3 a_{n-3} + ... + C_d a_{n-d}$
ou les d coefficients Ci ($\forall$ i)sont des constantes et $C_d \neq 0$
Une séquence constante-récursive est une séquence satisfaisant une récurrence de cette forme. Il existe d dégré de liberté pour les solutions de cette récurrence, c'est-à-dire les valeur initiales $a_0, a_1, a_2, ..., a_{d1}$ peut-être considérer comme n'importe quelle valeur, mais la récurrence détermine la séquence de manière unique. Les mêmes coefficients donnent le polynôme caractéristique (également ''polynôme auxiliaire" ou ''polynôme compagnon") suivant :
$P(t) = t^d - C_1 t^{d-1} - C_2 t^{d-2} - ... - C_d$
Dont les d racines jouent un rôle crucial dans la recherche et la compréhension des séquences satisfaisant la récurrence. Si les racine $r_1, r_2, ..., r_d$ sont toutes distinctes.
Alors chaque solution de récurrence prend la forme :
$a_n = K_1 r_1^n + K_2 r_2^n + ... + K_d r_d^n$
Ou les coefficients K_i sont déterminer pour s'adapter aux conditions initiales de la récurrence. Lorsque les même racine apparaissent plusieurs fois, les termes de cette formule correspondant à la deuxième occurrence et au occurrence ultérieures de la même racine sont multipliés par les puissances croissante de n. Par exemple si le polynôme caractéristique peut être factoriser comme $(X - r)^3$ avec la même racine r apparaissant trois fois, alors la solution prenant la forme :
$a_n = K_1 r^n + K_2 n r^n + K_3 n^2 r^n$

Example III.9. *Passons maintenant à un exemple pratique avec de la suite de Fibonacci:*

$$\textit{qui s'exprime comme suit :} \begin{cases} & T(0) = 0 \\ & T(1) = 1 \\ T(n) & = T(n-1) + T(n-2) \end{cases}$$

Premièrement on retrouve l'équation caractéristique :
$r^2 - r - 1 = 0$
Deuxièmement, on résout l'´equation caractéristique
$r^2 - r - 1 = 0$
$\Delta = 1 - 4(1)(-1) = 1 + 4 = 5$
$r_1 = \frac{1-\sqrt{5}}{2}, r_2 = \frac{1+\sqrt{5}}{2}$
Troisièmement, on écrit la solution générale
$T(n) = K_1(\frac{1-\sqrt{5}}{2})^n + K_2(\frac{1+\sqrt{5}}{2})^n$
Enfin, on détermine la solution particulière à partir de la donnée de la condition initiale

$$=> \begin{cases} T(0) = K_1(\frac{1-\sqrt{5}}{2})^0 + K_2(\frac{1+\sqrt{5}}{2})^0 = 0 \\ T(1) = K_1(\frac{1-\sqrt{5}}{2})^1 + K_2(\frac{1-\sqrt{5}}{2})^1 = 1 \end{cases} = \begin{cases} K_1 + K_2 = 0 \\ \frac{1-\sqrt{5}}{2}K_1 + \frac{1+\sqrt{5}}{2}K_2 = 1 \end{cases}$$

$=> K_1 = -\frac{\sqrt{5}}{5}$ *et* $=> K_2 = \frac{\sqrt{5}}{5}$

$=> T(n) = -\frac{\sqrt{5}}{5}(\frac{1-\sqrt{5}}{2})^n + \frac{\sqrt{5}}{5}(\frac{1+\sqrt{5}}{2})^n$

pour $c = 1$ *et* $n_0 = 1$ *on a* $T(n) \leq c\frac{\sqrt{5}}{5}(\frac{1-\sqrt{5}}{2})^n$

donc on conclut que $T(n) = O(\frac{1-\sqrt{5}}{2})^n$

Voici donc déterminer la complexité de la suite de Fibonacci en utilisant la méthode purement mathématique. Au début de l'énonce de cette méthode, on a parlé d'équation homogène linéaire ce qui signifie une équation linéaire sans second membre et de ce fait la méthode s'applique bien et on a notre complexité. La question que l'on se pose maintenant est de savoir : Si l'équation récursive de la complexité d'un algorithme est sous la forme linéaire non homogène comment procéder ?

III.3.6 méthodes de résolution des équation linéaire non homogène

Le principe de résolution, adopté dans cette section, consiste à éliminer d'abord la fonction $g(n)$ (qui est le second membre de l'equation), et ensuite résoudre l'équation homogène ainsi trouvée à l'aide de la méthode discutée précédemment. Voyons donc les différente méthode pour y parvenir a cette transformation (de équation non-homogène à équation homogène).

1. lorsque le second membre de l'equation récurrente est une constante dans ce cas ce que nous allons faire c'est de faire passer l'equation récurrente au rang $n + 1$ et de soustraire cette nouvelle équation a l'anciene de rang n suppose vrai. voyons un exemple.

   Example III.10. *Considérons l'équation de récurrence suivant:* $T(n + 2) - T(n + 1) - T(n) = 4$ *cette équation est non-homogène et on ne peut pas la résoudre en utilisant la méthode de résolution vu précédemment a cause de la précédente de g(n) = 4. Le but ici serai donc de faire disparaitre cette constante pour ainsi applique cette méthode et trouver la complexité. pour faire cela on a:*

   $$\begin{cases} T(n+2) - T(n+1) - T(n) = 4(1) \\ T(n+3) - T(n+2) - T(n+1) = 4(2) \end{cases}$$

   en faisant $(2) - (1)$, *on obtient* $T(n+3) - 2T(n+2) + T(n) = 0$ *et il ne reste plus qu'a la résoudre en utilisant la méthode discuter plus haut et on trouve* $t(n) = k_1 + k_2(1 + \sqrt{5})n + k_3(1 - \sqrt{5})n$

2. L'utilisation de l'opérateur E

   L'utilisation de l'opérateur E est aussi important pour ce qui est de transformer un équation linéaire homogène en un autre homogène. il est très efficace et très comprehensible lorsque le second membre de l'équation est une fonction $g(n)$ de la forme $g(n) = (\alpha)^n$ avec $\alpha \in \mathbb{N}$

   Example III.11. *transformation en utilisant l'opérateur E*
   Soit à résoudre l'équation suivante : $t(n) = t(n-1) + 2^n$ *et* $t(0) = 1$ *on commence tout d'abord par multiplier toute l'équation par l'expression* $E - 2$ *on a:* $(E-2)t(n) = (E-2)t(n-1) + (E-2)2^n$
   après on passe a la factorisation
   $(E-2)t(n) - (E-2)t(n-1) = (E-2)2^n$
   or $E2^n - 2^{n+1} = 0$
   $(E-2)(t(n) - t(n-1)) = 0$
   on aura deux racine de cette équation. $r_1 = 1$ *et* $r_2 = 2$
   par conséquent, la solution générale sera donnée par $t(n) = K_1 + K_2 2^n$
   $t(0) = K_1 = 1$ *et* $t(1) = 1 + 2K_2 = 3 \Rightarrow K_2 = 2$ *on a donc la solution suivante* $t(n) = 1 + 22^n$

IV Comparaison de deux algorithmes

IV.1 Méthode de comparaison

Si le nombre d'entrées utilisées pour un algorithme est très faible, généralement le temps d'exécution de l'algorithme l'est aussi. Ce qui nous intéresse, c'est comment va réagir l'algorithme pour un nombre important de données.
Donc pour comparer deux algorithmes, nous allons comparer leur taux de croissance ou ordre de grandeur.
On considère généralement qu'un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable à un ordre de grandeur inférieur.

IV.2 Classification des algorithmes

Les algorithmes habituellement rencontrés peuvent être classés dans les catégories suivantes :

Complexité	Description
Complexité $O(1)$ Complexité constante ou temps constant	L'exécution ne dépend pas du nombre d'éléments en entrée mais s'effectue toujours en un nombre constant d'opérations
Complexité $O(log(n))$ Complexité logarithmique.	La durée d'exécution croît légèrement avec n . Ce cas de figure se rencontre quand la taille du problème est divisée par une entité constante à chaque itération.
Complexité $O(n)$ Complexité linéaire	C'est typiquement le cas d'un programme avec une boucle de 1 à n et le corps de la boucle effectue un travail de durée constante et indépendante de n .
Complexité $O(nlog(n))$ Complexité n-logarithmique	Se rencontre dans les algorithmes où à chaque itération la taille du problème est divisée par une constante avec à chaque fois un parcours linéaire des données. Un exemple typique de ce genre de complexité est l'algorithme de tri "quick sort" qui, de manière récursive, divise le tableau à trier en deux morceaux par rapport à une valeur particulière appelée pivot, trie ensuite la moitié du tableau puis l'autre moitié ... S'il n'y avait pas cette opération de "division" l'algorithme serait logarithmique puisqu'on divise par 2 la taille du problème à chaque étape. Mais, le fait de reconstituer à chaque fois le tableau en parcourant séquentiellement les données ajoute ce facteur n au log n . Noter que la complexité n-logarithmique est tributaire du bon choix du pivot.
Complexité $O(n^2)$ Complexité quadratique	Typiquement c'est le cas d'algorithmes avec deux boucles imbriquées chacune allant de 1 à n et avec le corps de la boucle interne qui est constant.
Complexité $O(n^3)$ Complexité cubique	Idem quadratique mais avec ici par exemple trois boucles imbriquées.
Complexité $O(n^p)$ Complexité polynomiale	Algorithme dont la complexité est de la forme $O(n^p)$ pour un certain p . Toutes les complexités précédentes sont incluses dans celle-ci
Complexité $O(2^n)$ Complexité exponentielle	Les algorithmes de ce genre sont dits "naïfs" car ils sont inefficaces et inutilisables dès que n dépasse 50.

Table 2: classification des algorithmes

À l'inverse d'un algorithme naïf (complexité exponentielle) et par convention, un algorithme est dit praticable, efficace s'il est polynomial.

IV.3 Comportement des fonctions usuelles

Voici quelques données illustrant le comportement des fonctions vues dans la classification précédente.

n	10	100	1000	1E+6	1E+9
$\log_2(n)$	3,32	6,64	9,97	19,93	29,9
n	10	100	1000	1E+06	1E+09
$n.\log_2(n)$	33,22	664,39	1E+04	2E+07	3E+10
n^2	100	1E+04	1E+06	1E+12	1E+18
n^3	1000	1E+06	1E+09	1E+18	1E+27
n^5	1E+05	1E+10	1E+15	1E+30	1E+45
2^n	1024	1,27E+030	1,07E+301		

Figure 8: nombre d'opérations nécessaires pour exécuter un algorithme en fonction de sa complexité et de la taille des entrées du problème traité

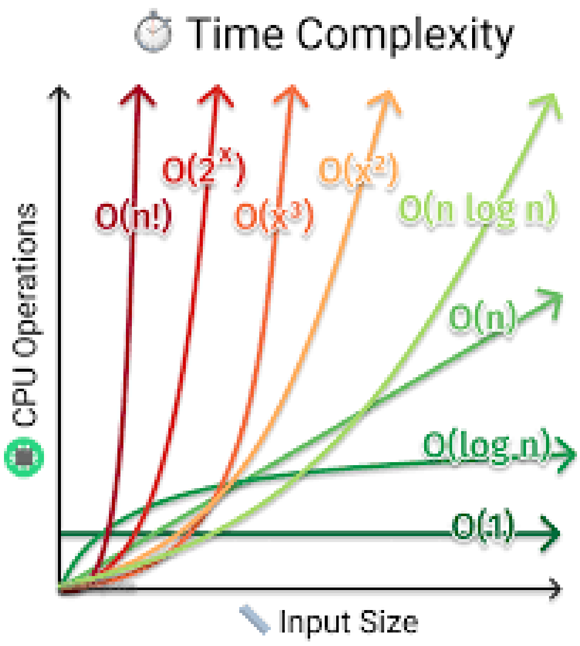

Figure 9: graphique représentant les différentes courbes des fonctions utilisées dans les complexités usuelles

n	10		100		1 000		1E+06		1E+09	
$Log_2(n)$	3,32E-009	10^{-9} s	6,64E-009	10^{-9} s	9,97E-009	10^{-8} s	1,99E-008	10^{-8} s	2,99E-008	10^{-8} s
n	1,00E-008	10^{-8} s	1,00E-007	10^{-7} s	1,00E-006	10^{-6} s	1,00E-003	10^{-3} s	1,00E+000	1 s
n.log(n)	3,32E-008	10^{-8} s	6,64E-007	10^{-6} s	9,97E-006	10^{-5} s	1,99E-002	10^{-2} s	2,99E+001	30 s
n^2	1,00E-007	10^{-7} s	1,00E-005	10^{-5} s	1,00E-003	10^{-3} s	1,00E+003	17 min	1,00E+009	32 ans
n^3	1,00E-006	10^{-6} s	1,00E-003	10^{-3} s	1,00E+000	1 s	1,00E+009	32 ans	1,00E+018	3.10^8 siècles
n^5	1,00E-004	10^{-4} s	1,00E+001	10 s	1,00E+006	11,5 jours	1,00E+021	3.10^{11} siècles	1,00E+036	
2^n	1,02E-006	10^{-6} s	1,27E+021	4.10^{11} siècles	1,07E+292					

Figure 10: temps nécessaire pour exécuter un algorithme en fonction de sa complexité et de la taille des entrées du problème traité en supposant que nous disposons d'un ordinateur capable de traiter 10 9 opérations par seconde

Conclusion

Résoudre des problèmes algorithmiques et de manière efficace est la base en informatique et pour jauger le niveau d'efficacité d'un algorithme on utilise une notion incontournable qui est la complexité. Ainsi l'évaluation de la complexité étant la base de ce cours, nous avons vu différentes méthodes permettant d'évaluer celle-ci que ce soit des algorithmes fictifs que récursifs. Et il est ressort que la classification des algorithmes suivant leur efficacité est relatif l'ordre de grandeur de leur complexité ie plus la complexité est grande moins l'algorithme est efficace.

Partie II

FICHE DE TD

Exercice 1

Soient trois fonctions $f(n), g(n), h(n)$.
Démonter les assertions suivantes

1. $f(x) = \Theta(g(n)) <=> f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$.
2. $f(x) = \Theta(g(n)) <=> g(x) = \Theta(f(n))$

Exercice 2

1. Ecrire l'algorithme naïf récursif du calcul de la combinaison $\binom{n}{k} = \frac{n!}{p!(n-p)!}$
2. Evaluer la complexité de cette algorithme dans le pire des cas sachant que $n! = \sqrt{2\pi n}\frac{n^n}{e^n}$ (c'est l'aproximation de Stirling).
3. Ecrire l'algorithme itératif pour le calcul de cette combinaisons en utilisant une matrice
4. Evaluer la complexité de ce nouvel algorithme
5. On suppose que la complexité calculer en a) et en d) sont respectivement $O(f(n))$ et $O(g(n))$. Démontrer que $f(n) = o(g(n))$.

Exercice 3

1. Ecrire l'algorithme du trie Rapide.
2. Exécuter cet algorithme sur le tableau suivant afin de montrer comment le trie s'opère.
3. Pour cet algorithme le pire des cas et le meilleur donne des complexités différentes

14	12	16	10	2	8	13	11	9	6

4. Expliquer dans quel cas subviens le pire des cas
5. Explique dans quel cas subviens le meilleur des cas
6. Déterminer les équations de complexité dans ces deux cas
7. En utilisant respectivement les méthodes de l'arbre récursif et de l'induction déterminer la complexité finale respectivement dans le meilleur des cas et dans le pire des cas.

Exercice 4

Soit la suite $T(n)$ définie par $T(2) = T(1) = T(0) = 1$
et $3T(n) = T(n-1) + T(n-2) + T(n-3), n > 2$
On considère le problème de calculer T (n) pour l'entrée n. Soit l'algorithme naïf récursif suivant

1. en utilisant la méthode de résolution mathématique des complexité, donner la forme générale solution de l'equation ci-dessus
2. déduire de cette formule générale, la solution particulier en utilisant les termes initiaux
3. Proposez un algorithme en O(n) (en supposant qu'on est dans le cadre du coût uniforme, i.e. en ne prenant pas en compte la taille des opérandes).

Exercice 5

Pour chacun des fonctions $Ti(n)$ suivant, démontrer les appartenances suivantes en utilisant les formules vues plus dans la cour.

1. $T1(n) = 6n^3 + 10n^2 + 5n + 2 \in O(n^3)$
2. $T2(n) = 3log_2n + 4 \in O(logn)$
3. $T3(n) = 2^n + 6n^2 + 7n \in O(2^n)$
4. $T4(n) = 7k + 2 \in O(1)$

5. $T4(n) = 4log_2 n + n \in O(n)$

6. $T5(n) = 2log_{10}k + kn^2 \in O(n^2)$

Exercice 6

Que vaut a à la fin de l'algorithme suivant, en fonction de n ? Quelle est la complexité de l'algorithme ?

```
Variables :   a  : entier naturel ;
a ← 5;
pour i de 1 à n faire
    pour j de 1 à n faire
        pour k de 1 à n faire
            a ← a + 3;
        fin
    fin
fin
```

Exercice 7

1. Ecrire l'algorithme du trie Fusion.

2. Exécuter cet algorithme sur le tableau suivant afin de montrer comment ce trie s'opère

14	12	16	10	2	8	13	11	9	6

3. Déterminer l'équations de complexité de cet algorithme.

4. En utilisant respectivement les méthodes de l'arbre récursif déterminer la complexité finale en temps de cet algorithme.

Exercice 8

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 9

Démontrer le master théorème.

Exercice 10

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 :

1. Trouver un élément qui y figure ?
2. 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 11

L'algorithme d'exponentiation rapide est basé sur la remarque suivante : on a $a^{2p} = a^p * a^p$ et $a^{2p+1} = a^p * a^p * a$. Ainsi, pour calculer a^{2p}, il suffit de savoir calculer a^p et de faire une multiplication, et pour calculer a^{2p+1}, il suffit de savoir calculer a^p et de faire deux multiplications. L'algorithme suivant implémente récursivement l'algorithme d'exponentiation rapide :

```
expoRapide(a,n):
si n == 0 alors
 | return 1
fin
b =expoRapide(a,n/2)
si n mod(2) == 1 alors
 | return b*b*a
sinon
 | return b*b
fin
```

Démontrer que, pour tout $n \geq 1$, le nombre total de multiplications effectuées par un appel à $expoRapide(a, n)$ est inférieur ou égal à $2 + 2log_2 n$.

Exercice 12

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, $50n$, $nlog(n)$, 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 $1000s$