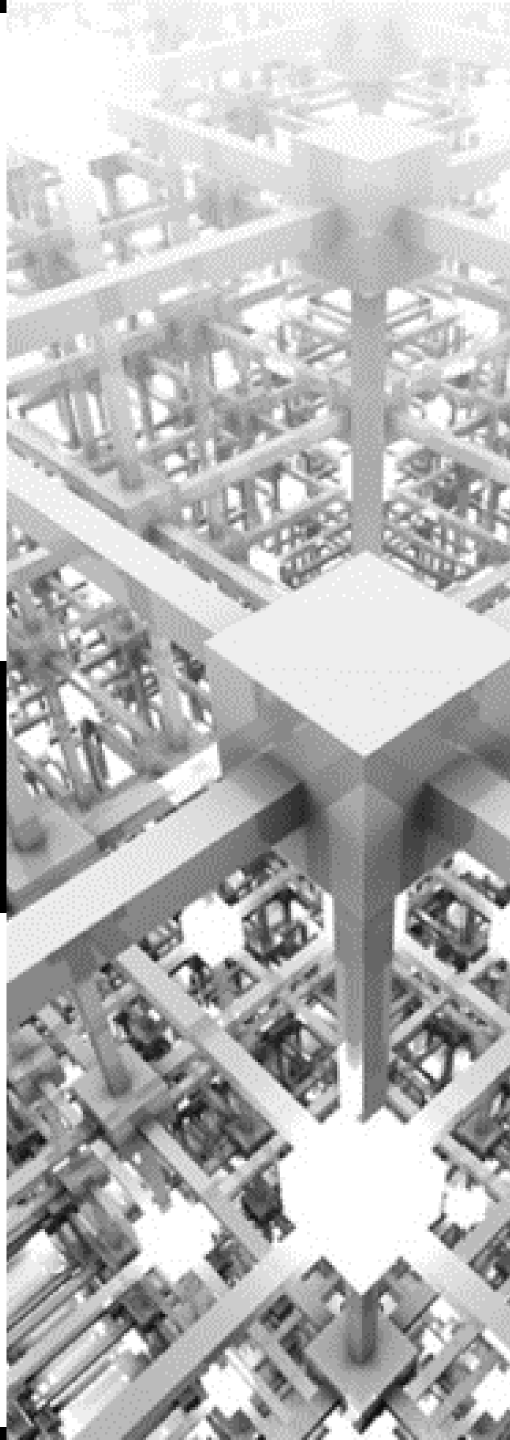
LE LANGAGE C

2ème Année «Années Préparatoires Intégrés»

2019/2020

Dep. Génie Industriel
Pf. CHERGUI Adil



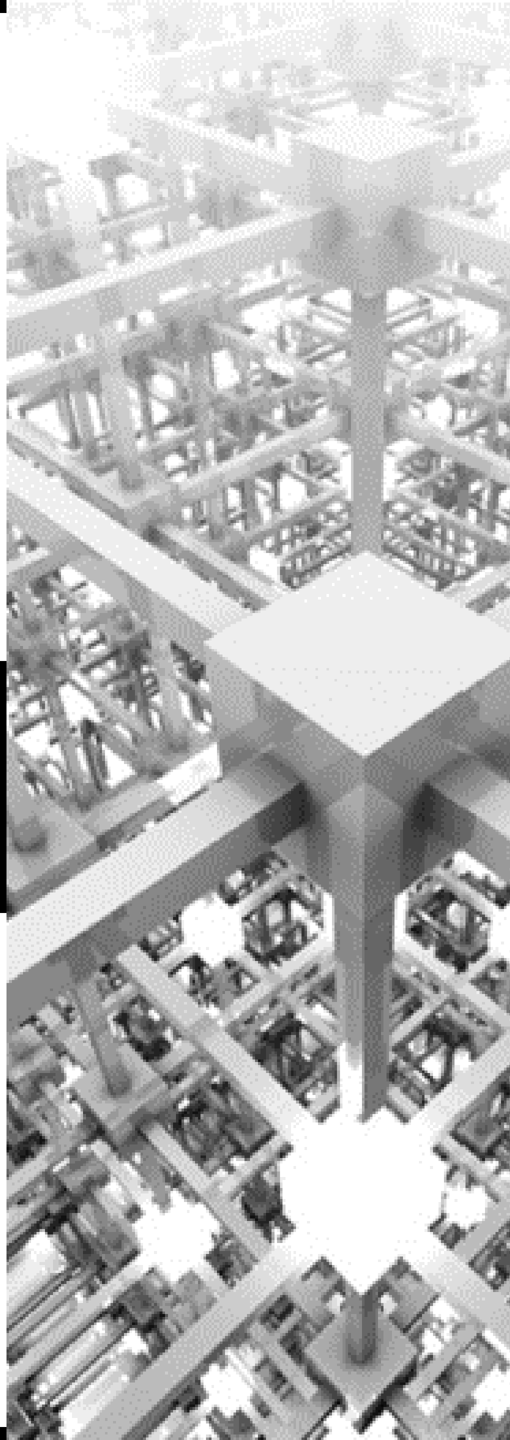
Séance 7

L'ANALYSE DE LA COMPLEXITÉ DES ALGORITHMES

Objectifs de la séance :

- Comprendre la notion de complexité pour étudier l'efficacité des programmes

Pf. Adil CHERGUI



LA COMPLEXITÉ

Préambule & introduction

“In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.”

Ada Lovelace (1815-1852) - Notes on the Sketch of The Analytical Engine.

Lors de l’exécution d’un algorithme, l’ordinateur effectue une succession d’opérations très simples comme comparer des nombres, des affectations, des opérations arithmétiques par exemple.
On mesure alors la complexité en temps d’un algorithme comme le nombre de ces opérations élémentaires.
Objectifs des calculs de **complexité** :

- pouvoir prévoir le **temps d'exécution** d'un algorithme.
- pouvoir **comparer** deux algorithmes réalisant le même traitement.

Ada Lovelace, de son nom complet **Augusta Ada King**, comtesse de Lovelace, née le 10 décembre 1815 à Londres et morte le 27 novembre 1852 dans la même ville, c’est une pionnière de la **science informatique**. Elle est principalement connue pour avoir réalisé le premier véritable programme informatique, lors de son travail sur un ancêtre de l'ordinateur : la **machine analytique** de **Charles Babbage**.

LA COMPLEXITÉ

Types de complexité

La complexité d'un algorithme peut être évalué en temps et en espace :

- **complexité en temps** : évaluation du temps d'exécution de l'algorithme.
- **complexité en espace** : évaluation de l'espace mémoire occupé par l'exécution de l'algorithme.

Exemple : échange de deux valeurs entières

Méthode 1:

```
// échange des valeurs de deux variables x
et y entier x, y, z;
... // initialisation de x et y
z <- x;
x <- y;
y <- z;
```

Méthode 2 :

```
// échange des valeurs de deux variables x
et y entier x, y;
... // initialisation de x et y
x <- y-x;
y <- y-x;
x <- y+x;
```

- la première méthode utilise une variable supplémentaire et réalise 3 affectations.
- la deuxième méthode n'utilise que les deux variables dont on veut échanger les valeurs, mais réalise 3 affectations et 3 opérations.

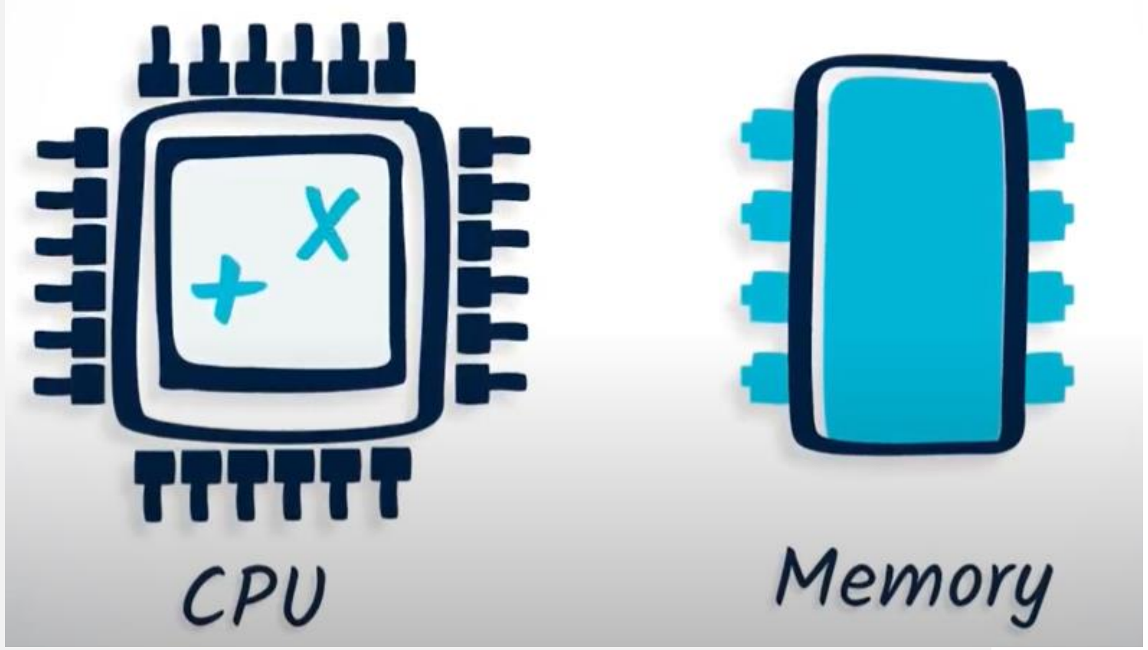
LA COMPLEXITÉ

Types de complexité

La conjoncture du programmeur (non officielle) de l'espace-temps informatique :

« pour gagner du temps de calcul, on doit utiliser davantage d'espace mémoire, et vice-versa. »

On s'intéresse actuellement essentiellement à la complexité en temps (ce qui n'était pas forcément le cas quand les mémoires coutaient cher)



LA COMPLEXITÉ

Paramètre de la complexité

On procède alors comme suite.

- On identifie **le paramètre de complexité** sous forme d'une grandeur **n** pour quantifier les données d'entrée.
- On calcule les performances seulement en fonction de **n**.
- On évalue le nombre d'opérations élémentaires.

Définition

Le **paramètre de la complexité** est la donnée du traitement qui va (le plus) faire varier le temps d'exécution de l'algorithme.

Exemples

Exemple 1: Pour un programme qui calcul de la factorielle de **n** :

Le paramètre de complexité est la valeur de **n.**

Exemple 2: Multiplication de deux entiers n et m.

Paramètre de complexité?

Exemple 3: Multiplier tous les éléments d'un tableau d'entiers par un entier donné.

Paramètre de complexité?

Exemple 4: Somme de 2 matrices.

Paramètre de complexité?

Exemple 5: Puissance 3 d'une matrice carrée.

Paramètre de complexité?

Exemple 6: Recherche dichotomique dans un tableau.

Paramètre de complexité?

LA COMPLEXITÉ

Les opérations prisent en compte

L’analyse de complexité consiste à calculer le nombre d'instructions le plus souvent des types suivants :

- Affectation : **na**
- Comparaison : **nc**
- Opération élémentaire: **no**

Chaque type d’instructions élémentaires prend un temps d’exécution particulier, en fonction des types de données, de l’environnement d’exécution.

L’étude de complexité peut agir sur tous ces types instructions ou sur un type en particulier lorsqu’on veut simplifier l’analyse (selon l’analyse demandée et le rôle de l’algorithme)

Certaines instructions sont négligées, telles que le coût des déclarations et des retours.

Le nombre d’opérations

Une fois le paramètre de complexité **n** est déterminé, et que les instructions sur lesquelles doit se faire l’analyse sont bien choisies.

Il faut calculer ensuite le nombre de ces opérations **T(n)** en fonction de **n.**

<u>Remarque:</u>

En général, le T(n) a une forme d’expression bien particulière qui est composée exclusivement par des fonctions de références.

1, n, n^2 ,n^3 …	Polynômiales
ln(n)	Logarithmiques
a^n	Exponentiels

LA COMPLEXITÉ

Exemples de calcul

L’analyse de complexité peut être effectué sur un ensemble d’instructions susceptibles de faire une tâches bien particulières, donc que cela soit sous forme de programme principal ou de fonction, l’analyse de complexité est particulièrement liés à une tâche; en dit par exemple étudier la complexité de la recherche séquentiel, la recherche dichotomique, le calcul du pgcd…
Dans les exemples qui vont suivre, nous allons étudié la complexité de certaines tâches sous forme de fonction pour éviter de confondre les instructions d’introduction ou l’affichage des données.

Exemple :

Le calcul de la somme suivante:

$$\sum_{i=1}^{n} i^2$$

LA COMPLEXITÉ

Exemple de calcul :1

Tâche :

Le calcul de la somme suivante:

$$\sum_{i=1}^{n} i^2$$

Programme :

```
long int somme( int n)
{
    long int s=1;
    for(i=1;i<=n;i++)      +1
    {
        s=s+i*i;           +2
    }
    return s;
}
```

n fois

3 x n

Paramètre de complexité :

La valeur de n.

Type d'instructions :

Opérations arithmétiques

T(n)= 3*n

Exemple de calcul : 2

Tâche :

Le calcul de la somme suivante:

$$\sum_{i=0}^{n}(2\times\sum_{j=0}^{i}(4+\sum_{k=j}^{i}(i+j-k)))$$

Paramètre de complexité :

La valeur de n.

Type d'instructions :

Opérations arithmétiques

Programme :

```
long int somme (int n)
{
    long S1,S2,S;
    int i,j,k;
    S=0;
    for(i=0;i<=n;i++)                 +1
    {
        S2=0;
        for(j=0;j<=i;j++)             +1
        {
            S1=0;
            for(k=j;k<=i;k++)         +1
            {
                S1+=i+j-k;            +3
            }
            S2+=S1+4;                 +2
        }
        S+=2*S2;                      +2
    }
    return S;
}
```

n fois

i+1 fois

i - j fois

$\sum_{i=1}^{n}((\sum_{j=0}^{i}((1+3)\times(i-j)+3))+3)=\sum_{i=1}^{n}((\sum_{j=0}^{i}(4\times(i-j)+3))+3)$

$\sum_{i=1}^{n}\left(\left(\sum_{j=0}^{i}(4\times(i-j)+3)\right)+3\right)=\sum_{i=1}^{n}\left(\left(4\times i\times i+3\times i-4\times\sum_{j=0}^{i}j\right)+3\right)$

$=\sum_{i=1}^{n}\left(\left(4\times i\times i+3\times i-2\times i\times(i+1)\right)+3\right)$

$=\sum_{i=1}^{n}(4\times i^2+3\times i-2\times i^2-2\times i+3)=\sum_{i=1}^{n}(2\times i^2+2\times i+3)=$
$2\times\sum_{i=1}^{n}(i^2)+2\times\sum_{i=1}^{n}(i)+\sum_{i=1}^{n}(3)=\frac{2n(n+1)(2n+1)}{6}+n(n+1)+3n=$
$\frac{(2n^2+2n)(2n+1)}{6}+n^2+n+3n=\frac{(4n^3+2n^2+4n^2+2n)}{6}+n^2+4n=\frac{2n^3}{3}+n^2+$
$\frac{n}{3}+n^2+4n$

$$=\mathbf{\frac{2n^3}{3}+2n^2+\frac{13n}{3}}$$

Exemple de calcul : 2

Tâche :

Le calcul de la somme suivante:

$$\sum_{i=0}^{n}(2 \times \sum_{j=0}^{i}(4 + \sum_{k=j}^{i}(i + j - k)))$$

Paramètre de complexité :

La valeur de n.

Type d'instructions :

Opérations arithmétiques

Programme :

```
long int somme (int n)
{
    long S1,S2,S;
    int i,j,k;
    S=0;
    for(i=0;i<=n;i++)                 +1
    {
        S2=0;
        for(j=0;j<=i;j++)             +1
        {
            S1=0;
            for(k=j;k<=i;k++)         +1
            {
                S1+=i+j-k;            +3
            }
            S2+=S1+4;                 +2
        }
        S+=2*S2;                      +2
    }
    return S;
}
```

n fois

i+1 fois

i - j fois

$$T(n) = \frac{2n^3}{3} + 2n^2 + \frac{13n}{3}$$

Exemple de calcul : 3

Tâche :

Le calcul du nombre de chiffres d'un entier n:

Programme :

Paramètre de complexité :

La valeur de n.

Type d'instructions :

Opérations arithmétiques

```
int nombre_chiffre(long int n)
{
    int c=1;
    while(fabs(n)>=10)
    {
        n/=10;   +1
        c++;     +1
    }
    return c;
}
```

$$T(n) = Log(n) \times 2$$

$$= 2 \times \frac{ln(n)}{ln(10)}$$

Exemple de calcul : 4

Tâche :

Recherche d'un élément dans un tableau de taille n:

Programme :

```
int recherche(float T[], int n, float x)
{
    int i=0;
    while((T[i]!=x)&&(i<n))
    {
        i++;
    }
    return !(i/n);
}
```

$\boxed{+3}$

Paramètre de complexité :

La taille du tableau n.

Type d'instructions :

Opérations logiques et comparaisons

Mais la position de x varie dans le tableau !!

➔ On ne sait pas exactement combien de fois cette boucle vas se répéter !!

➔ On ne peut pas calculer exactement le nombre d'opérations !!!

LA COMPLEXITÉ

Le complexité au pire, complexité au mieux, complexité moyenne

Lorsque, pour une valeur donnée du paramètre de complexité, le temps d'exécution varie selon les données d'entrée, on peut distinguer :

La complexité au pire : temps d'exécution maximum, dans le cas le plus défavorable.

La complexité au mieux : temps d'exécution minimum, dans le cas le plus favorable (en pratique, cette complexité n'est pas très utile).

La complexité moyenne : temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

Remarque :

Normalement **La complexité moyenne** est la mieux adapté pour l'analyse des performances d'un programme, Mais La complexité moyenne n'est pas toujours facile à calculer, et l'objectif principale est de donner des approximation pour borner le temps d'exécution..

le plus souvent, on utilise la **complexité au pire**, car

Dans l'exemple précédant :

Complexité au pire (x n'est pas dans le tableau) Donc :

$$T_{au\ pire}(\mathrm{n}) = 3 \times n.$$

Complexité au mieux (x est dans la première case du tableau) Donc :

$$T_{au\ mieux}(\mathrm{n}) = 3.$$

Complexité moyenne(si x peut se trouver équiprobablement dans tous les cases du tableau le calcul sera facile) Donc :

$$T_{au\ mieux}(\mathrm{n}) = (T_{au\ pire}(\mathrm{n}) + T_{au\ mieux}(\mathrm{n}))/2 = \frac{3 \times (n+1)}{2}$$

Mais Ce n'est pas toujours le cas

LA COMPLEXITÉ

Comportement asymptotique des fonctions de référence

Le but de cette partie va être de comparer les complexités calculées avec des fonctions de référence (puissance, logarithme, exponentielle, etc.). Il faudra préalablement introduire quelques notations classiques des études de fonctions.

Les définitions qui vont suivre permettent de comparer le comportement à l'infini de deux fonctions définies sur N. Plus précisément, il s'agit de critères pour affirmer qu'une fonction en domine une autre, ou si elles sont du même ordre de grandeur, voir même équivalente.
Tous cela pour permettre de lier le T(n) avec l’une des fonctions de références.

LA COMPLEXITÉ

La notion du grand O (the big oh)

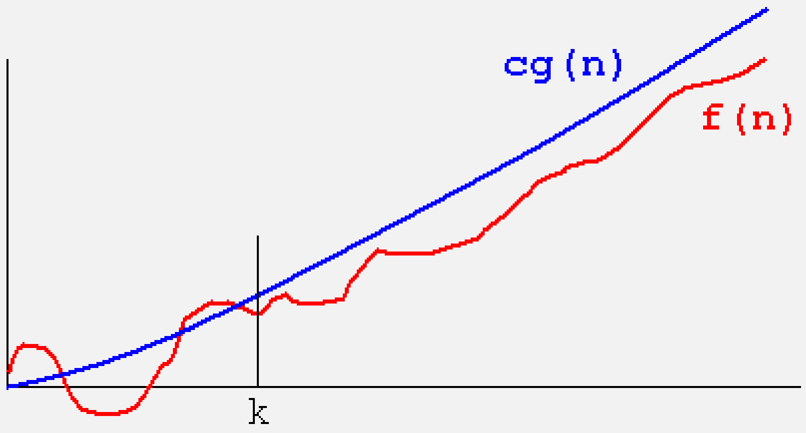
Borne supérieure asymptotique

On dit qu'une fonction f est un **grand O** d'une fonction g si et seulement si

$$\exists c > 0,\ \exists n_0 > 0 \text{ tel que } \forall n > n_0,\ f(n) < c \times g(n)$$

On note alors $f(n) = O(g(n))$.

cela signifie qu'à partir d'un certain rang la fonction f est majorée par une constante fois la fonction g. Il s'agit donc d'une situation de domination de la fonction f par la fonction g.



Quelques relations grand O

- Si $T(n) = 4$ alors $T(n) = O(1)$. Pour le prouver, prendre par exemple $c = 5$ et $n_0 = 0$.
- Si $T(n) = 3n + 2$ alors $T(n) = O(n)$. Pour le prouver, prendre par exemple $c = 4$ et $n_0 = 2$.
- Si $T(n) = 2n + 3$ alors $T(n) = O\left(n^2\right)$. Pour le prouver, prendre par exemple $c = 3$ et $n_0 = 1$.

LA COMPLEXITÉ

Notion de grand Ω (the big omega)

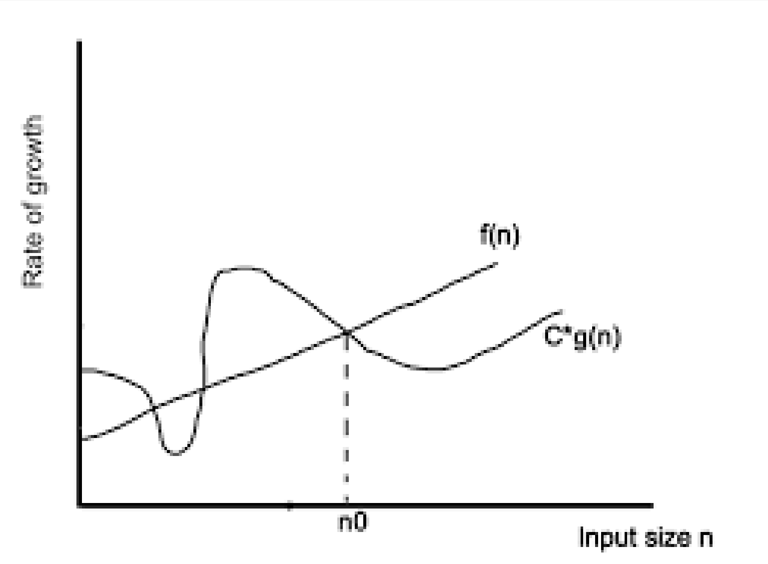
Borne inférieure asymptotique

On dit qu'une fonction f est un **grand Oméga** d'une fonction g si et seulement si

$$\exists c > 0,\ \exists n_0 > 0 \text{ tel que } \forall n > n_0,\ c \times g(n) < f(n)$$

On note alors $f(n) = \Omega(g(n))$.

Cette fois-ci, à partir d'un certain rang la fonction f est minorée par une constante fois la fonction g. Il s'agit donc d'une situation de domination de la fonction g par la fonction f.



Quelques relations grand Oméga

- Si $T(n) = 4$ alors $T(n) = \Omega(1)$.
- Si $T(n) = 4n + 2$ alors $T(n) = \Omega(n)$.
- Si $T(n) = 4n^2 + 1$ alors $T(n) = \Omega(n)$.

LA COMPLEXITÉ

Notion de grand Θ (the big theta)

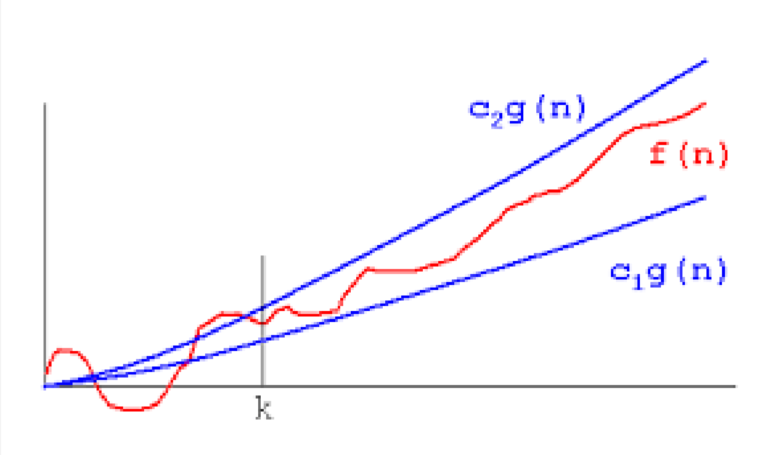
Borne asymptotique

On dit qu'une fonction f est un **grand Théta** d'une fonction g si et seulement si

$$\exists c_1 > 0,\ \exists c_2 > 0,\ \exists n_0 > 0 \text{ tel que } \forall n > n_0,\ c_1 \times g(n) < f(n) < c_2 \times g(n)$$

On note alors $f(n) = \Theta(g(n))$.

Cette situation combine les deux précédentes, à partir d'un certain rang la fonction f est encadrée par des multiples de la fonction g. Cela signifie que les fonctions f et g sont du même **ordre de grandeur.**



Quelques relations grand Théta

- Si $T(n) = 4$ alors $T(n) = \Theta(1)$.
- Si $T(n) = 4n + 2$ alors $T(n) = \Theta(n)$.
- Si $T(n) = 4n^2 + 1$ alors $T(n) = \Theta(n^2)$.

LA COMPLEXITÉ

Classes de complexité

Les complexités algorithmiques que nous allons calculer vont dorénavant être exprimées comme des grand **Thêta Θ** de fonctions de **références**. Cela va nous permettre de les classer.

$$\mathbf{1 < ln(n) < n < ln(n)\,n < n^2 < ln(n)\,n^2 < n^3 < \cdots < a^n < n^n}$$

Des algorithmes appartenant à une même classe seront alors considérés comme de complexité équivalente. Cela signifiera que l'on considèrera qu'ils ont la même efficacité.

Le tableau suivant récapitule quelque complexités de référence :

Tableau

O	Type de complexité
$O(1)$	constant
$O(\log(n))$	logarithmique
$O(n)$	linéaire
$O(n \times \log(n))$	quasi-linéaire
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel
$O(n!)$	factoriel

James Stirling $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

LA COMPLEXITÉ

Revenant au exemples

Exemple 1: Pour un programme qui calcul de la factorielle de **n** :

La complexité est linéaire

Exemple 2: Multiplication de deux entiers n et m.

La complexité est constante

Exemple 3: Multiplier tous les éléments d'un tableau d'entiers par un entier donné.

La complexité est linéaire

Exemple 4: Somme de 2 matrices.

La complexité est quadratique

Exemple 5: Puissance 3 d'une matrice carrée.

La complexité est cubique

Exemple 6: Recherche dichotomique dans un tableau.

La complexité est logarithmique

James Stirling $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$