# Complexité asymptotique

Considérer le comportement à l'infini de la complexité. Le comportement de la complexité d’un algorithme quand la taille des données devient grande.

Les données des algorithmes sont en général de grande taille et qu'on se préoccupe surtout de la croissance de cette complexité en fonction de la taille des données.

# Les notations Landau

- La notation O (grand O)
- La notation o (petit o)
- La notation Ω (grand oméga)
- La notation ω (petit oméga)
- La notation Θ  (grand thêta)

La notation ~ (de l’ordre de ; équivalent à)

# Les notations Landau

- **Notation grand O :**

$$O(g(n)) = \{ f : IN \rightarrow IN \mid \exists\ \mathrm{c}>0 \text{ et } \mathrm{n}_0 \geq 0$$
$$\text{tels que} \quad \mathbf{0 \leq f(n) \leq c.g(n)} \quad \forall\, n \geq \mathrm{n}_0 \ \}$$

- *On dit que g(n) est une* ***borne supérieure*** *asymptotique pour f(n), on note abusivement f(n)= O(g(n)).*

# Notation O

- *f(n) est de l'ordre de* O(g(n))

  $O(g(n))=\{\ f : IN \rightarrow IN \mid c>0 \text{ et } n_0 \geq 0 \text{ tels que } \mathbf{0 \leq f(n) \leq c.g(n)}\ \forall n \geq n_0\ \}$

*Exemple:*

$$f(n) = 2n + 5 \; est \; de \; l'ordre \; de \; O(n)$$

$$car \; 2n + 5 \leq 3n \qquad \forall n \geq 5$$

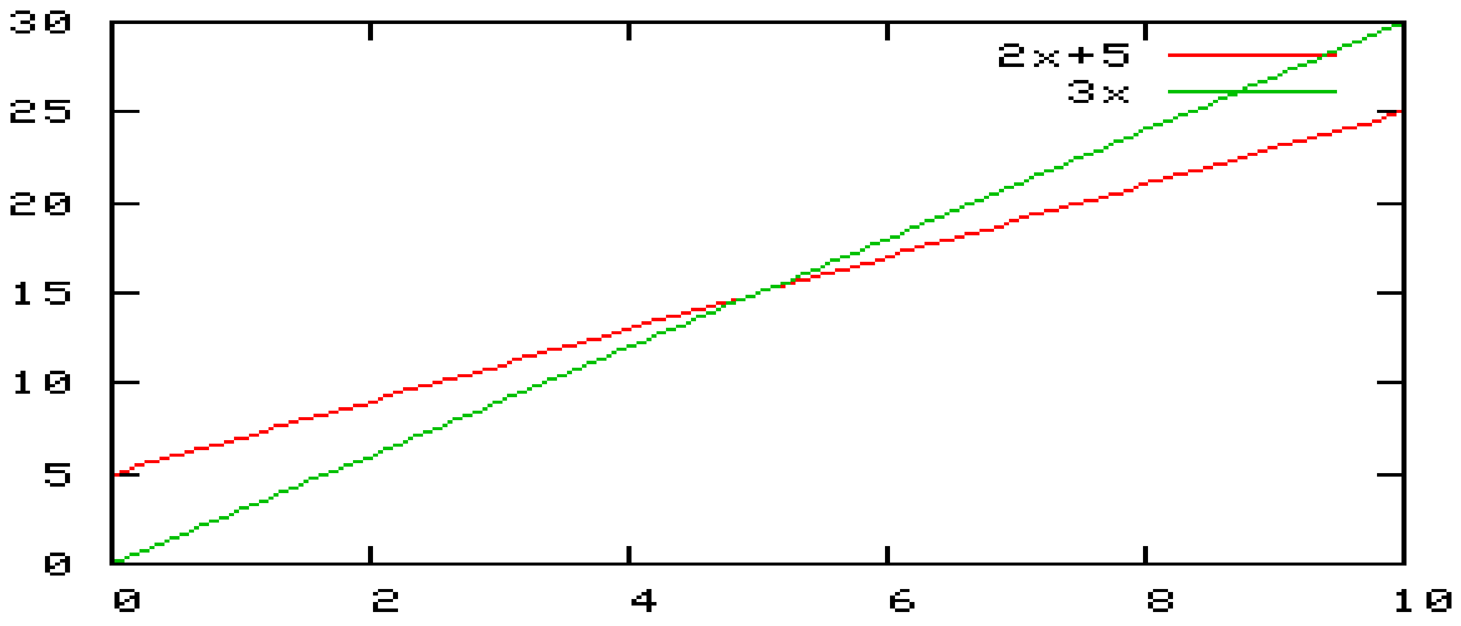
$$d'où \; c = 3 \; et \; n_0 = 5$$

# Notation O

*Exemple : f(n) = 2n+5 est de l'ordre de O(n)*

$$\text{car } 2n+5 \leq 3n \quad \forall\, n \geq 5$$

*Le graphe ci-dessous illustre l'exemple :*



# Les notations Landau

- **Notation Ω**

  $\Omega(g(n)) = \{ f : IN \rightarrow IN \mid \exists\ c>0 \text{ et } n_0 \geq 0$

  *tels que* $f(n) \geq c.g(n) \geq 0 \ \forall\ n \geq n_0 \}$

- *On dit que g(n) est une* ***borne inférieure*** *asymptotique pour f(n), on note abusivement* $f(n) = \Omega(g(n))$.

# Notation Ω

- *f(n) est de l'ordre de* $\Omega(g(n))$

$\Omega(g(n)) = \{ f : IN \rightarrow IN \mid c > 0$ et $n_0 \geq 0$ *tels que* $\mathbf{0 \leq c.g(n) \leq f(n)}$ $\forall n \geq n_0 \}$

*Exemple:*

$$f(n) = 2n + 5 \text{ est de l'ordre de } \Omega(n)$$

$$car\ 2n + 5 \geq 2n \qquad \forall n \geq 0$$

$$d'où\ c = 2\ et\ n_0 = 0$$

# Les notations Landau

- **Notation Θ**

$\Theta(g(n)) = \{ f : IN \rightarrow IN \mid \exists\, c_1 > 0, c_2 > 0 \text{ et } n_0 \geq 0 \text{ tels que } \mathbf{0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)} \ \forall\, n \geq n_0 \}$

*On dit que g(n) est* ***une borne asymptotique*** *pour f(n), on note abusivement* $f(n) = \Theta(g(n))$

# Notation $\Theta$

- *f(n) est de l'ordre de* $\Theta(g(n))$

$\Theta(g(n)) = \{ f : IN \rightarrow IN \mid c_1 > 0, c_2 > 0$ *et* $n_0 \geq 0$ *tels que*

$\mathbf{0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)}$ $\forall n \geq n_0 \}$ *g(n) borne asymptotique*

*Exemple:*

$$f(n) = 2n + 5 \; est \; de \; l'ordre \; de \; \Theta(n)$$

$$car \quad 2n \leq 2n + 5 \leq 3n \qquad \forall n \geq 5$$

$$d'où \quad c_1 = 2, C_2 = 3 \; et \; n_0 = 5$$

# Les notations Landau

*Remarque :*

*Notations o et ω avec des inégalités strictes.*

# Notations Landau

*Exercice:* Montrer que:

$$f(n)=5n^2 - 6n = \Theta(n^2)$$

$$f(n)=6n^3 \neq \Theta(n^2)$$

$$f(n)=n^2 = O(10^{-5}\, n^3)$$

$$f(n) = 10n^3 + 3n^2 + 5n +1=O(n^3)$$

# Propriétés

- Réflexivité : $f(n) = O(f(n))$.
- Transitivité : si $f(n) = O(g(n))$ et $g(n)=O(h(n))$ alors $f(n)=O(h(n))$.
- Somme :
  - ✓ si $f(n) = O(g1(n))$ et $h(n) = O(g2(n))$ alors $f(n)+h(n)=O(g1(n)+g2(n))$.
  - ✓ $c+ f(n) = O(c+g(n))=O(g(n))$ avec c une constante
- Produit :
  - ✓ si $f(n) = O(g1(n))$ et $h(n) = O(g2(n))$ alors $f(n)xh(n) = O(g1(n)x\ g2(n))$ ;
  - ✓ $c.f(n) = O(c.g(n))=O(g(n))$, car toute fonction constante est O(1).

*Remarque* : On a les mêmes propriétés avec $\Omega$ et $\Theta$.

# Propriétés

On peut aisément déduire les propriétés suivantes :

- Si $p(n)$ est un polynôme de degré $k$ alors $\boldsymbol{p(n)=\Theta(n^k)}$.
- $\boldsymbol{Log_b\, n = \Theta(log_b\, n) = \Theta(log\, n)}$. Nous pouvons donc dire qu'un algorithme est de complexité *log n* sans avoir à spécifier la base.
- $\boldsymbol{log(n+c) = \Theta(log\, n)}$ pour toute base.
- $\boldsymbol{(n+c)^k = O(n^k)}$.

# Vocabulaire

- *O(1) :* temps constant, indépendant de la taille des données. C'est le cas le plus optimal. (*résolution de l'équation du second degré.*
- O(n) : linéaire (*la recherche d'une valeur dans u*n tableau)
- O($n^2$) : quadratique (*tri des éléments d'un tableau*)
- O($n^3$) : cubique (*produit de deux matrices*)
- *O($n^k$) :* complexité polynômiale
- O(log n), O(n log n ),… complexité logarithmique (recherche dichotomique)
- O($2^n$), O(n !), …  : complexité exponentielle, factorielle,
- …

# Résumé

On compare les algorithmes sur la base de leur complexité.

La fonction exponentielle est toujours plus forte que la fonction polynomiale qui est plus forte que la fonction linéaire qui est plus forte que la fonction logarithmique.

# Estimation du coût d’un algorithme itératif

C'est le nombre d'opérations élémentaires, affectations, comparaisons, opérations arithmétiques, effectuées par l'algorithme.

# Estimation du coût d’un algorithme itératif

Affectation, lecture, écriture se mesurent par:

**O(1)**

# Estimation du coût d'un algorithme itératif

<u>Somme des coûts de traitements successifs:</u>

Traitement_*1* = coût_*1*

Traitement_*2* = coût_*2*

…

Traitement_*i* = coût_*i*

…

Coût_total = coût_1+coût_2+…coût_i+…

# Estimation du coût d’un algorithme itératif

## Instruction conditionnelle:

```
f(…)
{  si  (…)  alors g(…) ;
            Sinon h(…) ;
   finsi ;
}
```

**Coût de f(…)=MAX (coût de g(…), coût de h(…))**

# Estimation du coût d'un algorithme itératif

Instruction itérative « ***pour*** » (Boucle *pour*):

```
f(…,  n :entier)
{   pour i :=1 à n faire
          g(…) ;
}
```

**Coût de f(…,n)= n x coût de g(…)**

Si g(…) dépend de la valeur de i alors:

$$\boldsymbol{Coût\ de\ f(\ldots, n)} = \sum_{i=1}^{n} coût(\boldsymbol{g}(\ldots, \boldsymbol{i}))$$

# Estimation du coût d’un algorithme itératif

## Instruction itérative « ***tantque*** » :

f(…,  n :entier)

{ tantque <condition> faire

    traitement g(…) ;

}

Coût de f(…,n)=$\sum_{i=1}^{k} coût(g(n))$

# Règles de la notation O

***Facteurs constants :***

1. ***Tout facteur constant est simplifié à 1.***

   *Exemple: O(5)~ O(1)*

2. ***Les constantes multiplicatives sont omises.***

   *Exemple: O(5n)~ O(n)*

# Règles de la notation O

- *Règle de la somme* :

$$O(h(n)+g(n))=O(max(h(n),g(n)))$$

- *Règle de la multiplication*

$$O(h(n)xg(n)) = \begin{cases} O((h(n)xh(n))) & si\ h(n)>g(n) \\ O((g(n)xg(n))) & si\ g(n)>h(n) \end{cases}$$

# Estimation du coût d'un algorithme

Exemple:

Calcul de $e^x$ à l'aide d'un développement limité de n termes:

$$S = e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$$

Quel est l'ordre de grandeur du temps d'exécution?

# Estimation du coût d'un algorithme

<u>1ère solution:</u>

```
s=1;
Pour i=1 à n faire
    p=1;
    pour j=1 à i faire
        p=p*x/j;
    finpour
s=s+p
finpour;
```

# Estimation du coût d'un algorithme

La boucle interne est exécutée:

$$1 + 2 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Cette solution a une complexité de l'ordre de :

$$\boldsymbol{O(n^2)}$$

# Estimation du coût d’un algorithme

2ème solution:

$$\frac{x^i}{i!} = \frac{x}{i} * \frac{x^{i-1}}{(i-1)!}$$

```
S=1; p=1;
pour i=1 à n faire
    p=p*x/i;
    s=s+p;
fait;
```

# Estimation du coût d’un algorithme

Cette fois le corps de la boucle n’est exécutée que **n fois** donc la complexité est de l’ordre de:

$$O(n)$$