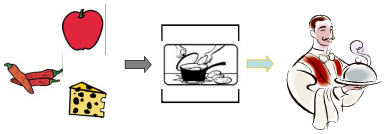
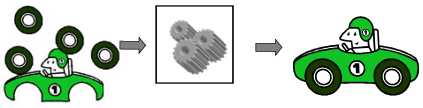
Analyse d'algorithmes



Entrée Algorithme Résultat

Un **algorithme** est une suite d'instructions qui sert à résoudre un problème donné dans un temps fini.



Entrée Algorithme Résultat

Un **algorithme** est une suite d'instructions qui sert à résoudre un problème donné dans un temps fini.

Analyser un algorithme = évaluer son **efficacité**

Efficacité ?

Temps d'exécution ….

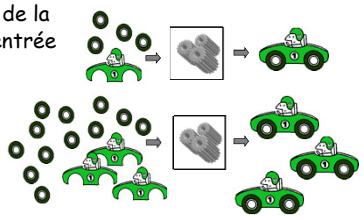
Espace mémoire occupé …

Qualité du résultat ….

Simplicité ….

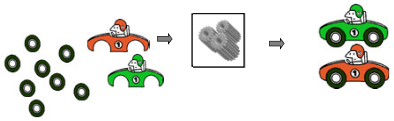
Temps d'exécution d'un algorithme

Le temps d'exécution d'un algorithme dépend de la **taille** des données d'entrée



Temps d'exécution d'un algorithme

Il dépend aussi de la nature des données à traiter (des entrées différentes peuvent avoir des temps d'exécution différents)



Exemple

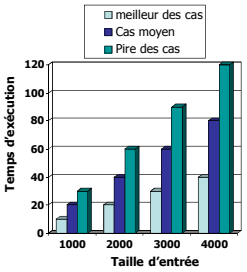
```
Si  x est impair
  return x
Si  x est pair
      calcul la somme S
      des premiers x entiers
      return S
```

15 → 15

4 → 10

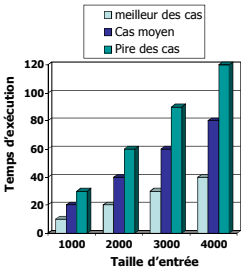
Temps d'exécution d'un algorithme

- Le temps d'exécution d'un algorithme dépend de la taille des données d'entrée
- Il dépend aussi des données (diverses exécutions peuvent avoir temps d'exécutions diverses)

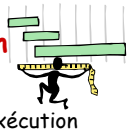


Temps d'exécution d'un algorithme

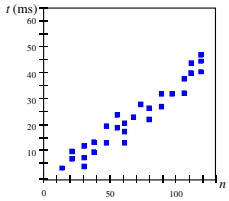
- Trouver le cas moyen peut être difficile
- On se concentre souvent sur le pire des cas.
  - plus facile a analyser
  - d'importance cruciale dans certaines applications (par ex. contrôle aérien, chirurgie, gestion de réseau)



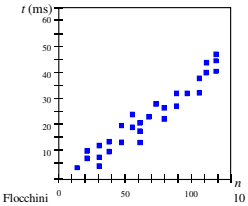
Mesurer le temps d'execution



- Comment on devrait mesurer le temps d'exécution d'un algorithme?
- Approche 1: Etude Expérimental



- Étude expérimentale:
  - Implémenter l'algorithme
  - Exécuter le programme avec des ensembles de données de taille et de contenu variés.
  - Mesurer précisément le temps d'exécution pour chaque cas



Etudes expérimentales

Limitations des mesures expérimentales:

- Il est nécessaire de implémenter
- Lors des tests, l'ensemble des données d'entrée est réduit et ne couvre pas la totalité des cas possibles
- Afin de comparer deux algorithmes, les mêmes environnements matériel et logiciel devraient être utilisés.

Analyse Théorique

Nous avons besoin d'une méthodologie générale pour analyser le temps d'exécution d'algorithmes qui:

- Utilise une description de haut niveau de l'algorithme (indépendante de l'implémentation)
- Caractérise le temps d'exécution comme une fonction de la taille des données d'entrée
- Considère touts les entrées
- Est indépendant des environnements matériels et logiciels

Analyse des algorithmes

Opérations primitives:
opérations de bas niveau qui sont indépendantes du langage de programmation, par exemple:

- Appel et retour d'une méthode
- Effectuer une opération arithmétique
- Comparer deux nombres, etc.
- Affectation d'une variable…

Analyse des algorithmes

Opérations primitives:
opérations de bas niveau qui sont indépendantes du langage de programmation, par exemple:

- Appel et retour d'une méthode
- Effectuer une opération arithmétique
- Comparer deux nombres, etc.
- Affectation d'une variable…

En observant le pseudo-code d'un algorithme on peut compter le nombre d'opérations primitives exécutées par cet algorithme et par la suite analyser son temps d'exécution et son efficacité.

Pseudo-code

- Un mélange de langage naturel et de concepts de programmation de haut niveau qui décrit les idées générales derrière la réalisation générique d'une structure de données ou d'un algorithme.
- Le pseudo-code est une description d'algorithme qui est plus structurée que la prose ordinaire mais moins formelle qu'un langage de programmation.
- Le pseudo-code est notre notation de choix pour la description d'algorithmes.
- Cependant, le pseudo-code cache plusieurs problèmes liés à la conception de programmes.

Analyse d'algorithmes - Exemple1

Trouver l'élément maximal d'un tableau d'entiers
Algorithme TabMax(A, n):
Entrée: un tableau A contenant n entiers
Sortie: L'élément maximal de A

```
currentMax ← A[0]
for i ← 1 to n -1 do
{
if currentMax < A[i] then
     currentMax ← A[i]
}
return currentMax
```

Analyse d'algorithmes - Exemple1

A

5	20	4	7	6	2	3	8	1	9

currentMax

```
currentMax ← A[0]
for i ← 1 to n -1 do
{
if currentMax < A[i] then
        currentMax ← A[i]
}
return currentMax
```


Analyse d'algorithmes - Exemple1

A

5	20	4	7	6	2	3	8	1	9

currentMax 5

```
currentMax ← A[0]
for i ← 1 to n -1 do
{
if currentMax < A[i] then
        currentMax ← A[i]
}
return currentMax
```


Analyse d'algorithmes - Exemple1

A

5	20	4	7	6	2	3	8	1	9

currentMax 5

```
currentMax ← A[0]
for i ← 1 to n -1 do
{
if currentMax < A[i] then
        currentMax ← A[i]
}
return currentMax
```


Analyse d'algorithmes - Exemple1

Quelles sont les opérations primitives à compter ?

A

5	20	4	7	6	2	3	8	1	9

Comparaisons
Affectations à currentMax

currentMax 20

```
currentMax ← A[0]
for i ← 1 to n -1 do
{
if currentMax < A[i] then
        currentMax ← A[i]
}
return currentMax
```

Analyse d'algorithmes - Exemple1

Meilleur cas

A

20	2	3	4	5	6	7	8	9	1

```
currentMax ← A[0]                          ← 1 affectation
for i ← 1 to n -1 do
{
if currentMax < A[i] then                  ← n-1 comparaisons
    currentMax ← A[i]                      ← 0 affectation
}
return currentMax
```


Analyse d'algorithmes - Exemple1

Pire cas

A

1	2	3	4	5	6	7	8	9	20

```
currentMax ← A[0]                          ← 1 affectation
for i ← 1 to n -1 do
{
if currentMax < A[i] then                  ← n-1 comparaisons
    currentMax ← A[i]                      ← n-1 affectations
}
return currentMax
```


Analyse d'algorithmes - Exemple1

Algorithme de recherche du max TabMax(A,n)

Meilleur cas	Pire cas
1 affection+(n-1) comparaisons	n affections+(n-1) comparaisons

Exemple2:

Chercher l'indice d'une valeur
dans un tableau A de taille sizeA

Opérations: affectations et verifications

```
i ← 0                          ----------> 1 affectation
while (A[i] ≠ element)         --------> sizeA verifications & affectation
      i ← i+1                            (dans le cas pire)
return i
```

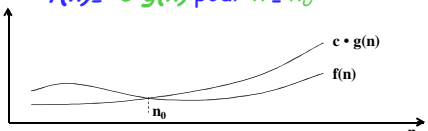
Notation asymptotique

Big-Oh (Grand Oh)

Limite supérieure

- Soit les fonctions *f(n)* et *g(n)*, nous disons que *f(n)* est O(*g(n)*) (ou *f(n)* =*O(g(n))* ou *f(n)* ∈ *O(g(n))*)
- si et seulement si il y a des constantes positives *c* et n_0 tel que

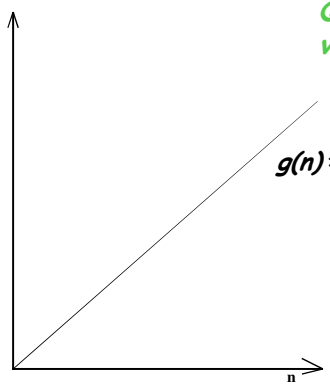
$f(n) \leq c\, g(n)$ pour $n \geq n_0$



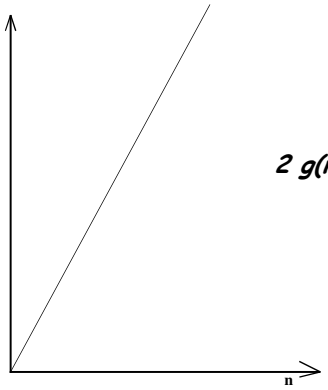
Quest-ce que ça veut dire c g(n) ?

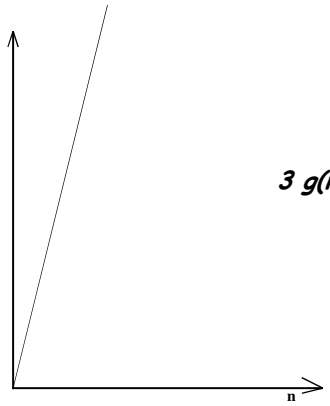
Example:

g(n) = n



2 g(n) = 2 n



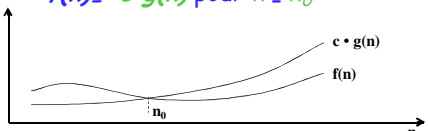


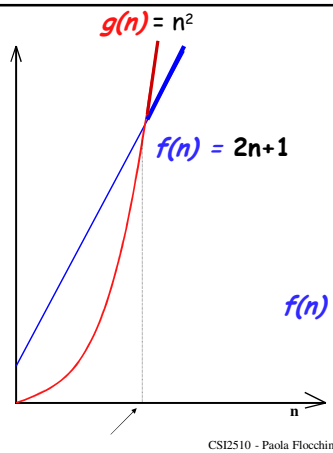
Big-Oh (Grand Oh)

Limite supérieure

- Soit les fonctions *f(n)* et *g(n)*, nous disons que *f(n)* est O(*g(n)*) (ou *f(n)* = *O(g(n))* ou *f(n)* ∈ *O(g(n))*)
- si et seulement si il y a des constantes positives *c* et n_0 tel que

$f(n) \leq c\, g(n)$ pour $n \geq n_0$





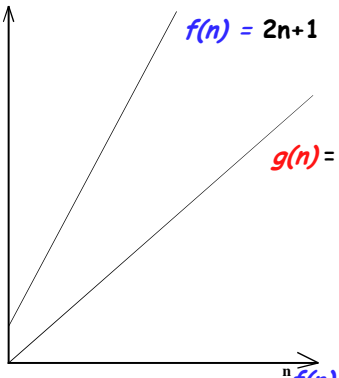
Exemple graphique

$f(n)$ est $O(n^2)$

$f(n) \leq c\, g(n)$ pour $n \geq n_0$

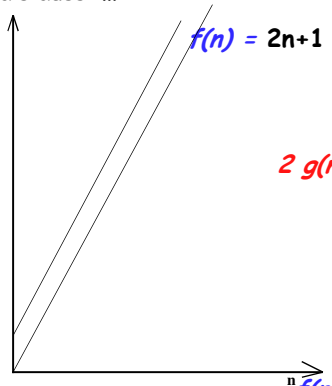
Pour quel c et n_0 ?

Mais aussi …

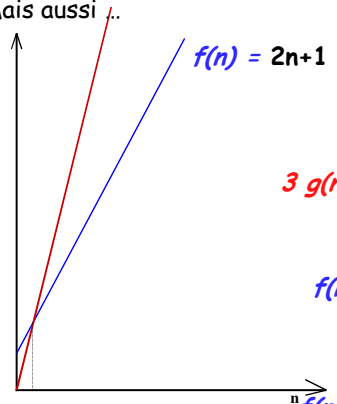


$f(n) \leq c\, g(n)$ for $n \geq n_0$

Mais aussi …



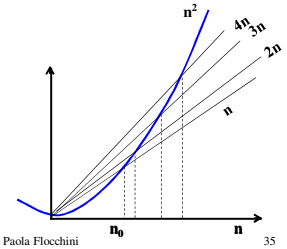
Mais aussi …



Mais …

n^2 n'est pas O(n) parce que nous ne pouvons pas trouver c et n_0 tel que $n^2 \leq cn$ pour $n \geq n_0$

(n'importe comment grand un c est choisi il y a un n assez grand tel que $n^2 > c\,n$)



Exemple

Limite supérieure - O (Grand O ou Big - Oh)

Prouver que **f(n) = 60n² + 5n + 1 est O(n²)**

Il faut trouver un nombre **c** et un nombre n_0 tel que:

$60n^2 + 5n + 1 \leq c\,n^2$ pour tout $n \geq n_0$

$5n \leq 5n^2$ pour tout **$n \geq 1$**

$1 \leq n^2$ pour tout **$n \geq 1$**

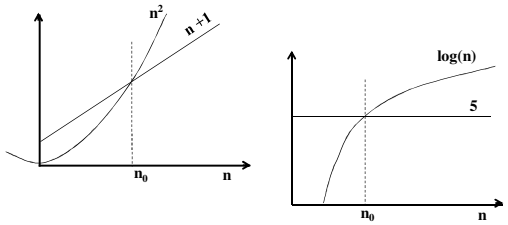
f(n) ≤ 60n² +5n² + n² pour tout **n≥1**

f(n) ≤ 66n² pour tout **n≥1**

c= 66 et n_0=1 => f(n) = O(n²)

Limite supérieure - O (Grand O ou Big - Oh)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$ … *mémorisez !!*



n =	2	16	256	1024
log log n	0	2	3	3.32
log n	1	4	8	10
n	2	16	256	1024
n log n	2	64	448	10 200
n^2	4	256	65 500	$1.05 * 10^6$
n^3	8	4 100	16 800 800	$1.07 * 10^9$
2^n	4	35 500	$11.7 * 10^6$	$1.80 * 10^{308}$

Limite supérieure - O (Grand O ou Big - Oh)

Théorème:
Si $g(n)$ est $O(f(n))$, alors pour n'importe quelle constante $c > 0$ $g(n)$ est aussi $O(c\, f(n))$

Théorème:
si $f_1(n)=O(g_1(n))$ et $f_2(n)=O(g_2(n))$ alors $f_1(n)+f_2(n)=O(\max(g_1(n),g_2(n)))$

Ex 1:

$$2n^3 + 3n^2 = O(\max(2n^3, 3n^2))$$
$$= O(2n^3) = O(n^3)$$

Ex 2:

$$n^2 + 3 \log n - 7 = O(\max(n^2, 3 \log n - 7))$$
$$= O(n^2)$$

Limite supérieure - O (Grand O ou Big - Oh)

Faire l'approximation la plus proche possible; utiliser la plus petite classe possible:

Ex.: Il est correct de dire que $5n-3$ est $O(n^3)$ mais la meilleure formulation est de dire $5n-3$ est $O(n)$

Utiliser l'expression la plus simple de la classe :

Ex.: Dire $10n+15$ est $O(n)$ au lieu de $10n+15$ est $O(10n)$

Limite supérieure - O (Grand O ou Big - Oh)

Laisser tomber les termes d'ordre inférieur ainsi que les coefficients

Ex.1: $7n-3$ est $O(n)$

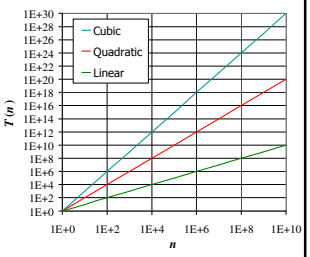
Ex.2: $6n^2\log(n) + 3n^2 + 5n$ est $O(n^2 \log n)$

Ex.3: $n^5+1000n^4+20n^3 - 8$ est $O(n^5)$

Notation asymptotique *(terminologie)*

Types de complexité :

constant:	$O(1)$
logarithmique:	$O(\log n)$
linéaire:	$O(n)$
quadratique:	$O(n^2)$
cubique:	$O(n^3)$
polynomial:	$O(n^k)$, $k \geq 1$
exponentiel:	$O(a^n)$, $a > 1$

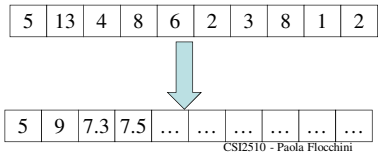


Exemple d'analyse asymptotique

Un algorithme pour calculer les moyennes préfixes

La moyenne de préfixe i-ième d'un tableau X est la moyenne des premiers $(i + 1)$ éléments de X

$$A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{(i + 1)}$$

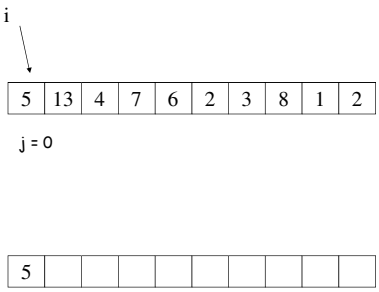


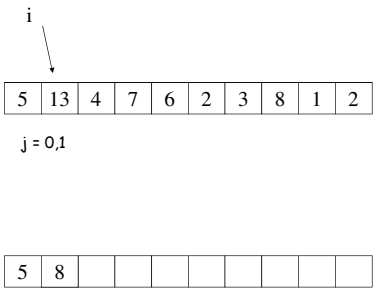
Exemple d'analyse asymptotique

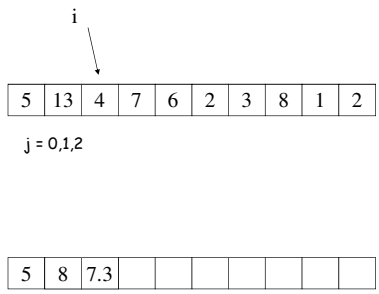
Algorithm *prefixAverages1*(X, n)

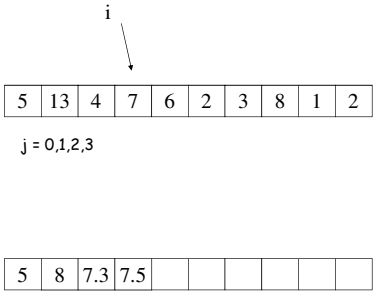
entrée Un vecteur de nombre X à n éléments
sortie un vecteur A avec la moyenne des préfixes de X

```
A ← new array of n integers
for i ← 0 to n − 1 do
    s ← X[0]
    for j ← 1 to i do
        s ← s + X[j]
    A[i] ← s / (i + 1)
return A
```


i
5 13 4 7 6 2 3 8 1 2
j = 0
5


i
5 13 4 7 6 2 3 8 1 2
j = 0,1
5 8


i
5 13 4 7 6 2 3 8 1 2
j = 0,1,2
5 8 7.3


i
5 13 4 7 6 2 3 8 1 2
j = 0,1,2,3
5 8 7.3 7.5

Exemple d'analyse asymptotique

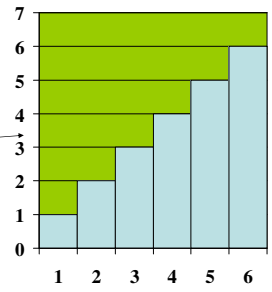
Algorithm prefixAverages1(X, n)

Entrée: Tableau **X** de **n** chiffres
Sortie: Tableau **A** de 'prefix averages'

	#operations
A ← new array of size **n**	
for i ← 0 **to n** − 1 **do**	
{ **s** ← **X**[0]	**n**
for j ← 1 **to i do**	
{**s** ← **s** + **X**[j]}	0+1+2+..+(n-1)
A[i] ← **s** / (**i** + 1)	**n**
}	
return A	**1**

$1 + 2 + \ldots + (n - 1) = ?$

- Le temps d'execution de ***prefixAverages1*** est $O(1 + 2 + \ldots + n)$
- Le sommation de les premières ***n*** éléments est $n(n + 1) / 2$
  représentations visuelle
- Alors, l'algorithme ***prefixAverages1*** est $O(n^2)$



Donc l'algorithme ***prefixAverages1*** a un temps d'execution de:

$O(n(n + 1) / 2)$

=

$O(n^2)$

NE PAS OUBLIER

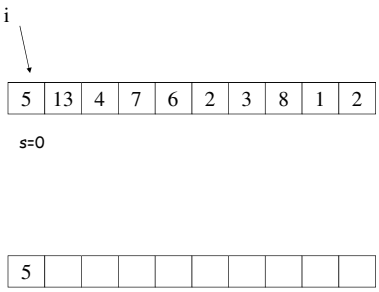
$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

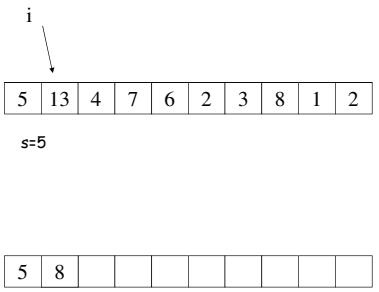
Un autre exemple:
un meilleur algorithme pour calculer les moyennes préfixes.

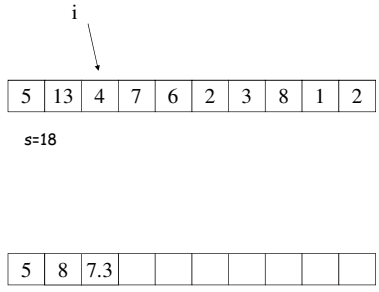
Algorithm prefixAverages2(X, n)

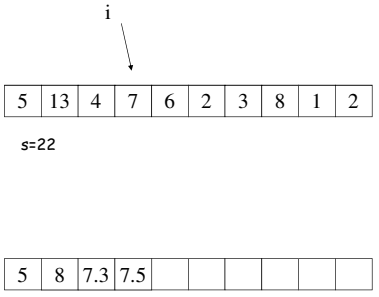
Entrée: Tableau **X** de **n** chiffres
Sortie: Tableau **A** de 'prefix averages'

```
A ← new array of size n
s ← 0
for i ← 0 to n − 1 do
    {
    s ← s + X[i]
    A[i] ← s / (i + 1)
    }
return A
```


i
5 13 4 7 6 2 3 8 1 2
s=0
5


i
5 13 4 7 6 2 3 8 1 2
s=5
5 8


i
5 13 4 7 6 2 3 8 1 2
s=18
5 8 7.3


i
5 13 4 7 6 2 3 8 1 2
s=22
5 8 7.3 7.5

Algorithm prefixAverages2(X, n)

Entrée: Tableau **X** de **n** chiffres
Sortie: Tableau **A** de 'prefix averages'

	#operations	
A ← new array of size **n**		
s ← 0		
for i ← 0 **to** n − 1 **do**		
{		
s ← s + X[i]	n	O(n)
A[i] ← s / (i + 1)	n	
}		
return A	1	

O(n)

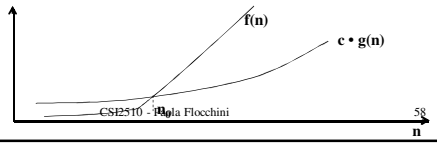
big-Omega (Grand Omega)
(Limite inferieure)

f(n) est Ω(g(n))

Si il y a $c > 0$ et $n_0 > 0$ tel que

$f(n) \geq c \cdot g(n)$ pour tout $n \geq n_0$

f(n) est Ω(g(n)) si et seulement si g(n) est O(f(n))



big-Thêta (Grand Thêta)

g(n) est Θ(f(n))

<===>

si g(n) ∈ O(f(n))

et

f(n) ∈ O(g(n))

Exemple

Nous avons déjà vu:

$f(n) = 60n^2 + 5n + 1$ est $O(n^2)$

mais $60n^2 + 5n + 1 \geq 60n^2$ pour $n \geq 1$

Donc: avec $c = 60$ et $n_0 = 1$

$f(n) \geq c \cdot n^2$ pour tout $n \geq 1$

⇨ f(n) est Ω (n^2)

Alors:

f(n) est $O(n^2)$
et
f(n) est $\Omega(n^2)$

↓

f(n) est $\Theta(n^2)$

Intuition pour les notations asymptotiques

Big-Oh

- f(n) est O(g(n)) si f(n) est **plus petite ou égal** à g(n) quand n est grand

big-Omega

- f(n) est Ω(g(n)) si f(n) est **plus grand ou égal** à g(n) quand n est grand

big-Theta

- f(n) est Θ(g(n)) si f(n) est **a peu près égal** à g(n) quand n est grand

Mathématiques à réviser

Logarithmes et exposants

Propriétés des logarithmes:

$\log_b(xy) = \log_b x + \log_b y$

$\log_b (x/y) = \log_b x - \log_b y$

$\log_b xa = a\log_b x$

$\log_b a = \log_x a / \log_x b$

Propriétés des exposants:

$a^{(b+c)} = a^b a^c$

$a^{bc} = (a^b)^c$

$a^b / a^c = a^{(b-c)}$

$b = a^{\log_a b}$

$b^c = a^{c*\log_a b}$

Mathématiques à réviser

- Plancher (Floor): $\lfloor x \rfloor$ = le plus grand entier $\leq x$ $\lfloor 2.3 \rfloor = 2$
- Plafond (Ceiling): $\lceil x \rceil$ = le plus petit entier $\geq x$ $\lceil 2.3 \rceil = 3$
- Sommations:
  - Définition général:

$$\sum_{i=s}^{t} f(i) = f(s) + f(s+1) + f(s+2) + .. + f(t)$$

- Où *f* est une fonction, *s* est l'index de départ, et *t* est l'index d'arrivée

Progression géométrique

$$S = \sum_{i=0}^{n} r^i = 1 + r + r^2 + \ldots + r^n$$

$$rS = r + r^2 + \ldots + r^n + r^{n+1}$$

$$rS - S = (r-1)S = r^{n+1} - 1$$

$$S = (r^{n+1}-1)/(r-1)$$

If r=2, $S = (2^{n+1}-1)$

Progression aritmetique

$$S = \sum_{i=0}^{n} di = 0 + d + 2d + \ldots + nd$$

$$= nd+(n-1)d+(n-2)d + \ldots + 0$$

$$2S = nd + nd + nd + \ldots + nd$$

$$= (n+1)\, nd$$

$$S = d/2\, n(n+1)$$

for $d=1$ $S = 1/2\, n(n+1)$

Sommation de logarithme

$$\sum_{n} \log k = \log n!$$

$$> \log n$$

$$< \log n^n = n \log n$$