Partie 3 : Complexité

- La notion de complexité fait référence à l'efficacité d'un algorithme.
- Il existe 2 formes de complexité pour un algorithme :
  - Complexité spatiale = espace mémoire occupé.
  - **Complexité temporelle** = temps d'exécution

Temps d'exécution

- Temps d'exécution d'un algorithme: pourquoi le mesurer ?
- Temps d'exécution d'un algorithme: Comment le mesurer ?

Temps d'exécution : Pourquoi ?

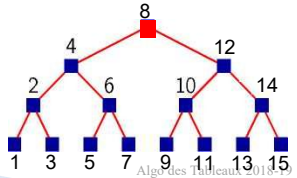
- Pour un problème donné, il y a plusieurs algorithmes.
- Il est facile d'écrire des algorithmes faux ou inefficaces.
- Un algorithme inefficace peut faire la différence entre plusieurs années et quelques minutes de calculs sur une même machine.

Temps d'exécution : Pourquoi ?

Exemple :
Construire une ville de 15 maisons en évitant aux livreurs de pizzas qui suivent les rues un trajet trop long depuis la pizzeria.

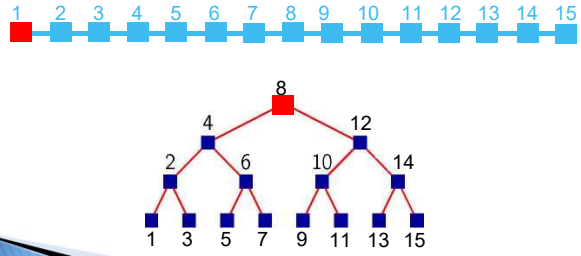
Organisation 1 : Linéaire. Numéros croissants. Pizzeria au numéro 1.

Organisation 2 : Embranchements.
À l'ouest de la maison *k*, n° < *k*, et à l'est, n° > *k*.
La pizzeria est au numéro 8.



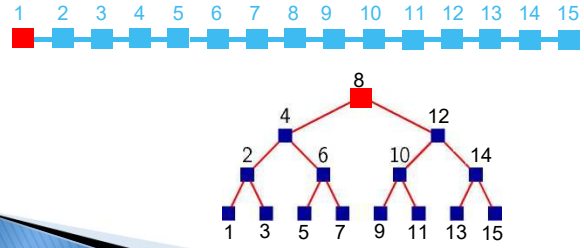
Temps d'exécution : Pourquoi ?

- Dans les deux organisations, le livreur a une méthode simple pour trouver une maison en partant de la pizzeria.



Temps d'exécution : Pourquoi ?

- On suppose qu'il faut une unité de temps pour passer d'une maison à une autre (en suivant une rue).
- Dans **le cas le pire**, quel est le temps mis par un livreur pour aller jusqu'à une maison depuis la pizzeria ?



Temps d'exécution : Pourquoi ?

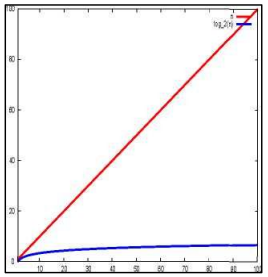
Nombre de maisons	Temps avec organisation 1	Temps avec organisation 2
15	14	3
1023	1022	9
1 073 741 823	1073741822	29
n	n−1	$\sim\log_2(n)$

Temps d'exécution : Pourquoi ?

- **Différence entre *n* et log *n* :**

Pour notre livreur de pizza :

- Si $n = 10^6$, alors $\log_2 \approx 20$
  Il fait 50 000 fois moins de déplacements si les maisons sont organisés par « embranchements ».
- Si $n = 10^9$, alors $\log_2 n \approx 30$, il fait alors 30 000 000 fois moins de déplacements.

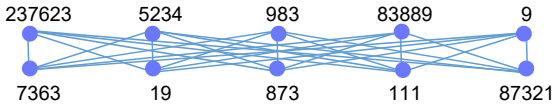


Temps d'exécution : Pourquoi

2ème exemple

- Problème : déterminer si 2 ensembles *E1*, *E2* de *n* entiers ont une valeur commune.

- Algorithme 1 : comparer successivement chaque élément de *E1* avec chaque élément de *E2*. Il faudra environ n^2 comparaisons.



Temps d'exécution : Pourquoi

2ème exemple

- Problème : déterminer si 2 ensembles *E1*, *E2* de *n* entiers ont une valeur commune.

- Algorithme 2 : Avec une structure de données adaptée, on peut résoudre le problème avec environ n*log(n)

|E1| = |E2|	Algorithme 1	Algorithme 2
n	n^2	n*log(n)
10	100	10
1000	1000000	3000
100000	10000000000	500000

Temps d'exécution : Pourquoi

Différence entre n^2 et n*log(n)
Sur un ordinateur exécutant une instruction élémentaire en 10^{-9} s.

- Si les ensembles *E1* et *E2* ont $n = 1\ 000\ 000 = 10^6$ éléments
  - Exécuter $n * \log n$ instructions élémentaires nécessite 0,006s.
  - Exécuter n^2 instructions élémentaires nécessite 10^3s soit environ 16mn40s.

- Si les ensembles *E1* et *E2* ont $n = 10\ 000\ 000 = 10^7$ éléments
  - Exécuter $n * \log n$ instructions élémentaires nécessite 0,07s.
  - Exécuter n^2 instructions élémentaires nécessite 10^5s soit plus d'une journée.

- En informatique, on manipule parfois des ensembles énormes
  - Google indexe 30 000 milliards de pages web,
  - Google reçoit près de 5,5 milliards de requêtes/jour

Temps d'exécution : Comment ?

- 1ère idée : Faire une étude expérimentale.
  - Implémenter le ou les algorithmes.
  - Exécuter le programme pour différentes valeurs de n.
  - Mesurer précisément le temps d'exécution.
  - En déduire une approximation du temps d'exécution.

- Exemple: déterminer si 2 ensembles *E1*, *E2* de *n* entiers ont une valeur commune.

|E1| = |E2|	Algorithme 1	Algorithme 2
10	100	10
1000	1000000	3000
100000	10000000000	500000
n	n^2	n*log(n)

c3

Temps d'exécution : Comment ?

- 1ère idée : Faire une étude expérimentale.

Inconvénients :

- Il faut programmer les algorithmes.
  - Conséquences ?
- On ne pourra tester qu'un sous-ensemble des cas possibles.
  - Conséquences ?
- Pour comparer 2 algorithmes on devra utiliser les mêmes environnements matériel et logiciel.
  - Conséquences ?

Temps d'exécution : Comment ?

- 2ème idée : Utiliser une méthode théorique qui :
- Utilise l'algorithme et est indépendante de l'implémentation et du langage de programmation.
- Définit le temps d'exécution comme une fonction dépendante de la taille des données d'entrée.
- Prend en compte toutes les entrées possibles.
- Est indépendante des environnements matériel et logiciel.

Complexité temporelle : Définition

- Le temps de calcul (ou complexité) d'un algorithme est la fonction qui à un entier **n** associe le nombre maximal d'instructions élémentaires que l'algorithme effectue, lorsqu'on travaille sur des objets de taille *n*.
- La fonction compte les instructions élémentaires au lieu de mesurer le temps.
- On s'intéresse en particulier à la complexité dans le pire des cas.
- Exemples d'opérations élémentaires :
  - additionner, soustraire, multiplier ou diviser deux nombres,
  - tester si une valeur est égale à une autre valeur,
  - affecter une valeur à une variable.

Complexité temporelle : Définition

- Le décompte des instructions peut être fastidieux si on compte aussi les accès aux structures de données, les E/S, appels de fonctions, etc
- Même en se limitant à une seule opération on peut obtenir une expression qu'il faudra approximer pour obtenir une solution
- Même si les opérations élémentaires ont des temps d'exécution constants sur une machine donnée, ils varient d'une machine à l'autre.
- Donc pour simplifier le calcul sans perdre sa pertinence on négligera les constantes.
- En pratique, on se contente d'un ordre de grandeur asymptotique.

Diapositive 130

c3 carole; 01/03/2018

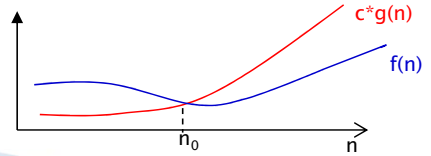
c2

Ordres de grandeur

Limite asymptotique supérieure – O (Grand O)

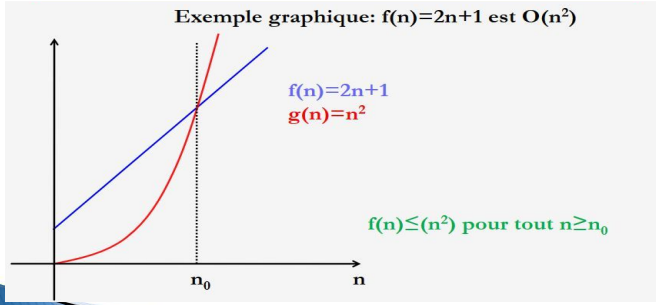
Soient 2 fonctions $f(n)$ et $g(n)$ de $\mathbb{N}$ dans $\mathbb{R}$.
On dit que $f(n)$ est $O(g(n))$ ou $f(n) = O(g(n))$
si et seulement si :

$$\exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^* \,/\, f(n) \leq c * g(n) \;\forall\, n \geq n_0$$



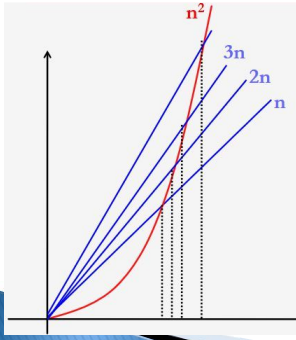
Ordres de grandeur

Limite asymptotique supérieure O (Grand O)



Ordres de grandeur

Limite asymptotique supérieure O (Grand O)



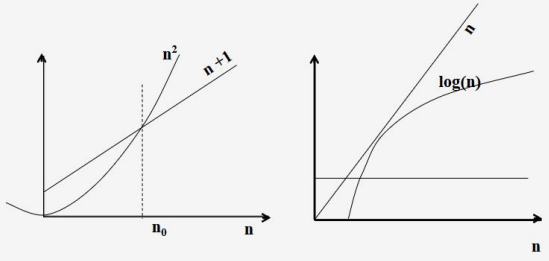
$f(n)=n^2$ n'est pas $O(n)$ parce qu'on peut pas trouver un c tel que $c*n \geq n^2$ pour $n \geq n_0$

f(n) croit plus vite que g(n)

Ordres de grandeur

Limite asymptotique supérieure O (Grand O)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)\ldots$



Diapositive 134

c2 analogie motard/VIP

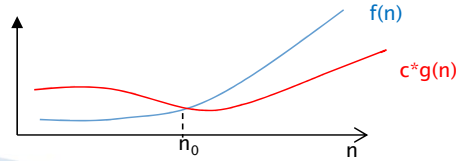
carole; 17/02/2017

Ordres de grandeur

Limite asymptotique inférieure– Ω (Grand Oméga)

Soient 2 fonctions $f(n)$ *et* $g(n)$ *de* $\mathbb{N}$ *dans* $\mathbb{R}$.
On dit que $f(n)$ *est* $\Omega(g(n))$ *ou* $f(n) = \Omega(g(n))$
si et seulement si :

$$\exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^* \,/\, f(n) \geq c * g(n) \;\forall\, n \geq n_0$$



Ordres de grandeur

Limite asymptotique – Θ (Grand thêta)

Soient 2 fonctions $f(n)$ *et* $g(n)$ *de* $\mathbb{N}$ *dans* $\mathbb{R}$.
On dit que $f(n)$ *est* $\Theta(g(n))$ *ou* $f(n) = \Theta(g(n))$
si et seulement si :

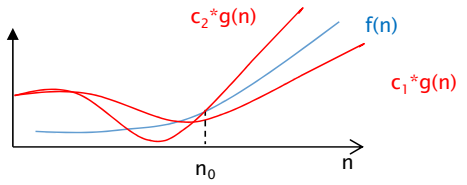
$f(n)$ *est* $O(g(n))$ *et* $f(n)$ *est* $\Omega\,(g(n))$

Ordres de grandeur

Limite asymptotique – $\boldsymbol{\Theta}$ (Grand thêta) :

$$\forall\, n \geq n_0, \exists\, c_1 \in \mathbb{R}^{+*}, \exists\, c_2 \in \mathbb{R}^{+*} \,/$$

$$c_1 * \mathrm{g(n)} \leq f(n) \leq c_2 * g(n)$$



Ordres de grandeur : comportements asymptotiques

Intuitivement :

- Grand O : f(n) est O(g(n)) si f(n) est plus petite ou égale à g(n) quand n est grand.
- Grand Oméga : f(n) est Ω (g(n)) si f(n) est plus grande ou égale à g(n) quand n est grand.
- Grand thêta : f(n) est θ (g(n)) si f(n) est à peu près égale g(n) quand n est grand.

Complexité : Exemple

```
def multiplication ( x, y):
  p=0
  for i in range(y):
      p=p+x
  return (p)
```

op élémentaires

p=0 → 1

for i in range(y): → 1

p=p+x → 2

y fois

return (p) → 1

2 + 3*y

$\exists c_1 = 1, c_2 = 5$

$1 * y \leq 2 + 3 * y \leq 5 * y \quad \forall y > 1,$

donc la complexité est $\theta(y)$

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

```
x=7    y=9   p=0
x=7    y=9   p=7
  *2     /2
x=14   y=4   p=7
x=28   y=2   p=7
x=56   y=1   p=63
x=112  y=0   p=63
return 63
```


Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x
        y = y//2
    return (p)
```

Basé sur l'équation mathématique :

x*y=2*x*y/2

Faux en Informatique !

Complexité : Exemple

```
x=7    y=9   p=0
x=7    y=9   p=7
  *2     /2
x=14   y=4   p=7
x=28   y=2   p=7
x=56   y=1   p=63
x=112  y=0   p=63
return 63
```

```
x=111      y=1001  p=0
x=111      y=1001  p=7
  *2         /2
x=1110     y=100   p=7
x=11100    y=10    p=7
x=111000   y=1     p=63
x=1110000  y=0     p=63
return 63
```

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

op élémentaires

1
1 (+1 sortie)
2
0 ou 2 (k fois)
2
2
1

[3+7*k,3+9*k]

Que vaut k ?

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

op élémentaires

1
1 (+1 sortie)
2
0 ou 2 (k fois)
2
2
1

[3+7*k,3+9*k]

k *est tel que* $2^{k-1} \leq y < 2^k$

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

op élémentaires

1
1 (+1 sortie)
2
0 ou 2 (k fois)
2
2
1

[3+7*k,3+9*k]

k *est tel que* $\log(2^{k-1}) \leq \log(y) < \log(2^k)$

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

op élémentaires

1
1 (+1 sortie)
2
0 ou 2 (k fois)
2
2
1

[3+7*k,3+9*k]

k *est tel que* $(k-1)\log(2) \leq \log(y) < k * \log(2)$

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

op élémentaires

p=0	1
while (y>0):	1 (+1 sortie)
if (y%2 ==1):	2
p=p+x	0 ou 2 (k fois)
x = 2*x #decalage	2
y = y//2 #decalage	2
return (p)	1

[3+7*k,3+9*k]

k est tel que $(k-1) \leq log_2(y) < k$

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

op élémentaires

p=0	1
while (y>0):	1 (+1 sortie)
if (y%2 ==1):	2
p=p+x	0 ou 2 (k fois)
x = 2*x #decalage	2
y = y//2 #decalage	2
return (p)	1

[3+7*k,3+9*k]

k est tel que $k \leq log_2(y) + 1 < k + 1$

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

op élémentaires

p=0	1
while (y>0):	1 (+1 sortie)
if (y%2 ==1):	2
p=p+x	0 ou 2 (k fois)
x = 2*x #decalage	2
y = y//2 #decalage	2
return (p)	1

[3+7*k,3+9*k]

donc $k < log_2(y)$

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

op élémentaires

p=0	1
while (y>0):	1 (+1 sortie)
if (y%2 ==1):	2
p=p+x	0 ou 2 (k fois)
x = 2*x #decalage	2
y = y//2 #decalage	2
return (p)	1

[3+7*k,3+9*k]

donc $3 + 7 * log_2(y) < \#\text{op élémentaires} < 3 + 9 * log_2(y)$

Complexité : Exemple

```
def multiplication (x, y):
    p=0
    while (y>0):
        if (y%2 ==1):
            p=p+x
        x = 2*x  #decalage
        y = y//2 #decalage
    return (p)
```

# op élémentaires	
1	
1	(+1 sortie)
2	
0 ou 2	k fois
2	
2	
1	

[3+7*k,3+9*k]

donc la complexité est $\Theta(log_2(y))$

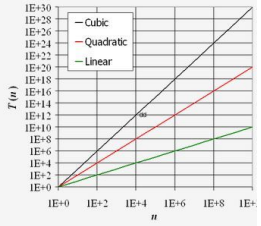
Complexité : ordre de grandeur

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	2048	10 240
n^2	4	256	65 536	1 048 576
n^3	8	4 096	16 777 216	$1.07 * 10^9$
2^n	4	65 536	$1.15 * 10^{77}$	$1.79 * 10^{308}$

Complexité : vocabulaire

Terminologie: Types de complexité

Constante:	O(1)
Logarithmique:	O(log(n))
Linéaire:	O(n)
Quasi-linéaire:	O(n.log(n))
Quadratique:	$O(n^2)$
Cubique:	$O(n^3)$
Polynomiale:	$O(n^k)$, k >0
Quasi-polynomiale	$O(n^{log(n)})$
Exponentielle:	$O(a^n)$, n > 1
Factorielle	O(n!)



Pour mémoire

Propriétés des logarithmes:

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

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

$\log_b x^a = a*\log_b x$

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

Complexité : exercice

Pour chacun des algorithmes suivants donner votre intuition du temps de calcul :

- Chercher un élément x dans un tableau de n éléments
- Insérer un élément dans un tableau de n éléments
- Tracer une ligne verticale sur une image de taille (L*H)
- Tracer une ligne verticale d'épaisseur e sur une image de taille (L*H).
- Calculer a^k
- Calculer la somme des puissances de a : $a^0+a^1+\ldots+a^n$