

# ANALYSE D'ALGORITHMES

# Complexité d'algorithmes

temps de calcul / complexité de temps : exécution dans un modèle formel de calcul

usage de mémoire / complexité d'espace

→ importance théorique (théorie de complexité)

→ importance pratique : on veut une approche scientifique

# Démarche scientifique

modèle  $\rightarrow$  prédiction  $\rightarrow$  expérience

1. définir : modèle de l'entrée + notion de taille (modèle permet la simulation)
2. identifier : partie critique du code
3. modèle de coût : opération typique (comparaison ou déplacement dans un tri, accès aux cases du tableau, opération arithmétique dans algorithme numérique, ...)
4. donner la fréquence d'opérations typiques en fonction de la taille de l'entrée

```
Entrée:  $x[0..n - 1]$  //  $\{n > 0\}$   
1 initialiser  $\min \leftarrow x[0]$   
2 for  $i \leftarrow 1, \dots, n - 1$  do  
3   | if  $x[i] < \min$  then  $\min \leftarrow x[i]$   
4 end  
5 return  $\min$ 
```

# Meilleur, pire, moyen

Temps de calcul  $T(n)$  pour entrée de taille  $n$

- ★ **meilleur** cas :  $\min T(n)$  parmi toutes entrées de taille  $n$
- ★ **pire** cas :  $\max T(n)$  parmi toutes entrées de taille  $n$
- ★ **moyen** cas :  $T(n)$  en espérance — dépend de la distribution des entrées (p.e. uniforme)

# Tri par insertion

```
Entrée:  $T[0..n - 1]$  //  $\{n > 0\}$   
1 for  $i \leftarrow 1, 2, \dots, n - 1$  do  
2   |  $x \leftarrow T[i]; j \leftarrow i$   
3   | while  $j > 0 \ \&\& \ T[j - 1] > x$  do  $T[j] \leftarrow T[j - 1]; j \leftarrow j - 1$   
4   |  $T[j] \leftarrow x$   
5 end
```

1. entrée : taille  $n$ , éléments distincts
2. le plus fréquemment exécuté : boucle interne
3. opération typique : déplacement ( $T[j] \leftarrow T[j - 1]$ )

meilleur cas :  $\min D(n) = 0$

pire cas :  $\sum_{i=0}^{n-1} i \sim n^2/2$

moyen cas (permutation uniforme) :  $\sum_{i=0}^{n-1} i/2 \sim n^2/4$

# Expérimentation

on a un algorithme  $\Rightarrow$  implémenter + tester

expérimentation avec ++ de jeux d'entrées : simulation ou repository de benchmarks

évaluation des résultats (comparaison avec la prédiction)

on mesure :

★ **temps actuel** d'exécution : reporter information sur l'environnement de test : CPU, RAM, système d'exploitation, ...

★ **compte d'opérations** : ne dépend pas de l'environnement

expériences reproductibles !  $\Rightarrow$  falsifiabilité de la prédiction + vérifiabilité de tests

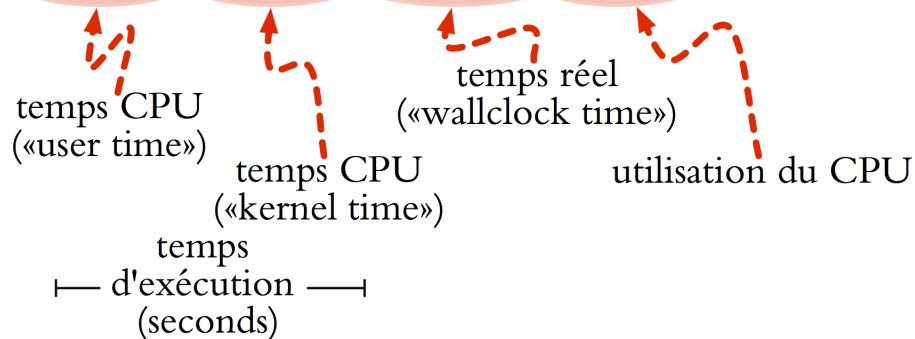
# Temps actuel de calcul

- ★ dans le code

```
...  
long T0 = System.currentTimeMillis(); // temps de début  
...  
long dT = System.currentTimeMillis()-T0; // temps (ms) dépassé
```

- ★ dans la ligne de commande (time)

```
% time java -cp Monjar.jar mabelle.Application  
0.283u 0.026s 0:00.35 85.7% 0+0k 0+53io 0pf+0w
```



- ★ profilage de code dans l'IDE (Eclipse/Netbeans/...)

# Nombre d'opérations

★ dans le code

```
private static final boolean COUNT=true;
private static int op_count=0;
private static void insert(double[] T, int n, double x){
    // T[0]<=T[1]<=...<=T[n-1] déjà trié
    int i=n;
    while(i>0 && T[i-1]>x) {
        T[i]=T[i-1]; if (COUNT) $op_count++;
        i--;
    }
    T[i]=x;
}
public static void insertionSort(double[] T){
    for (int n=0; n<T.length; n++) insert(T,n,T[n]);
    if (COUNT)
        System.out.println(T.length+"\t"+op_count);
}
```

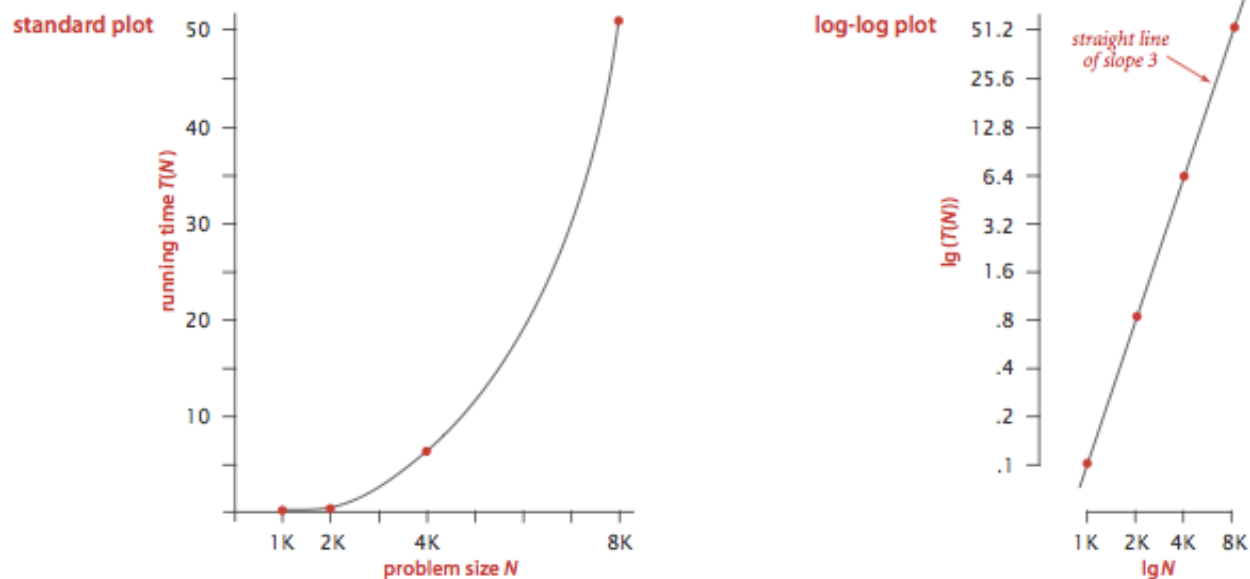
★ profilage avec l'IDE (par ligne ou méthode)



# Conception d'expériences

1. répéter avec la même taille  $\Rightarrow$  moyen + dispersion statistique
2. répéter avec tailles différentes : doubler ou  $10\times$

notre hypothèse  $T(n) \sim a \cdot n^b : \frac{T(2n)}{T(n)} \sim \frac{a(2n)^b}{an^b} = 2^b$   
 $\rightarrow$  visualiser sur log-log : la pente indique l'exposant  $b$



# Diviser pour régner

*divide-and-conquer*

Démarche algorithmique à un problème de structure récursive :

- ★ partitionner le problème dans des sous-problèmes similaires,
- ★ chercher la solution optimale aux sous-problèmes par récursion,
- ★ combiner les résultats.

Récursion pour temps de calcul

$$T(n) = \tau_{\text{partition}}(n) + \underbrace{\sum_{i=1, \dots, m} T(n_i)}_{m \text{ sous-problèmes}} + \tau_{\text{combiner}}(n)$$

# Algorithme récursif

- ★ calcul descendant (*top-down*) : avec appels récursifs
- ★ calcul ascendant (*bottom-up*) : programmation dynamique

# Minimum - récursions

```
1 MINREC( $x[0..n - 1], g, d$ )
2 // minimum parmi  $x[g..d - 1]$ 
3 if  $d - g = 0$  then return  $\infty$ 
4 if  $d - g = 1$  then return  $x[g]$ 
5  $mid \leftarrow \lfloor (d + g)/2 \rfloor$ 
6  $m_1 \leftarrow \text{MINREC}(x, g, mid)$ 
7  $m_2 \leftarrow \text{MINREC}(x, mid, d)$ 
8 return  $\min\{m_1, m_2\}$ 
```

Premier appel :  $\text{MINREC}(x, 0, n)$

on veut  $C(\ell)$ , comparaisons quand  $d - g = \ell$  :

$$C(\ell) = \begin{cases} 0 & \{ \ell = 0, 1 \} \\ C(\lfloor \ell/2 \rfloor) + C(\lceil \ell/2 \rceil) + 1. & \{ \ell > 1 \} \end{cases}$$

Solution :  $C(\ell) = \ell - 1$  pour  $\ell > 0$  ; preuve : par induction.

# Calculer $x^n$

$$x^n = \begin{cases} 1 & \{n = 0\} \\ x^{n/2} \cdot x^{n/2} & \{n > 0, n \text{ est pair}\} \\ x \cdot x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} & \{n > 0, n \text{ est impair}\} \end{cases}$$

```
1 POWER( $x, n$ )
2 if  $n = 0$  then return 1
3  $y \leftarrow$  POWER( $x, \lfloor \frac{n}{2} \rfloor$ )
4  $z \leftarrow y \times y$ 
5 if  $n \bmod 2 = 0$  then return  $z$ 
6 else return  $z \times x$ 
```

# Exponentiation

$R(n)$  : nombre d'appels récursifs

$$R(n) = \begin{cases} 0 & \{n = 0\} \\ R(\lfloor n/2 \rfloor) + 1 & \{n > 0\} \end{cases}$$

**Substitution** :  $b(n) =$  nombre de bites dans la représentation binaire de  $n$

$$R'(b) = \begin{cases} 0 & \{n = 0\} \\ R'(b - 1) + 1 & \{n > 0\} \end{cases}$$

**Solution** :

$$\begin{aligned} R'(b) &= b && \text{resubstitution} && R(n) &= R'(b(n)) \\ R(n) &= b(n) = \lceil \lg(n + 1) \rceil = \underbrace{1 + \lfloor \lg n \rfloor}_{\text{si } n > 0} \sim \lg n. \end{aligned}$$

# Recherche binaire

on cherche élément  $x$  dans un tableau trié  $A[0..n - 1]$

sous-tâche : chercher dans  $A[g..d - 1]$

partition : par  $m = \lfloor (g + d)/2 \rfloor$

cas de base :  $d - g = 0$

nombre de comparaisons pour recherche infructueuse au pire avec  $n = d - g$

$$C(n) = \begin{cases} 0 & \{n = 0\} \\ C(\lfloor n/2 \rfloor) + 1 & \{n > 0\} \end{cases}$$

# Méthode de la bisection

on veut trouver la racine de  $f(x) = 0$

sous-tâche : trouver la racine de  $f(x) = 0$  à  $a \leq x \leq b$ :  $f(a) < 0, f(b) > 0$   
(*bracketing*)

partition : par  $m = \lfloor (a + b)/2 \rfloor$

cas de base  $x_0 - \epsilon/2 \leq a < b \leq x_0 + \epsilon/2$  pour assurer erreur numérique  $\epsilon$

nombre d'appels récurrents au pire avec  $\ell = b - a$

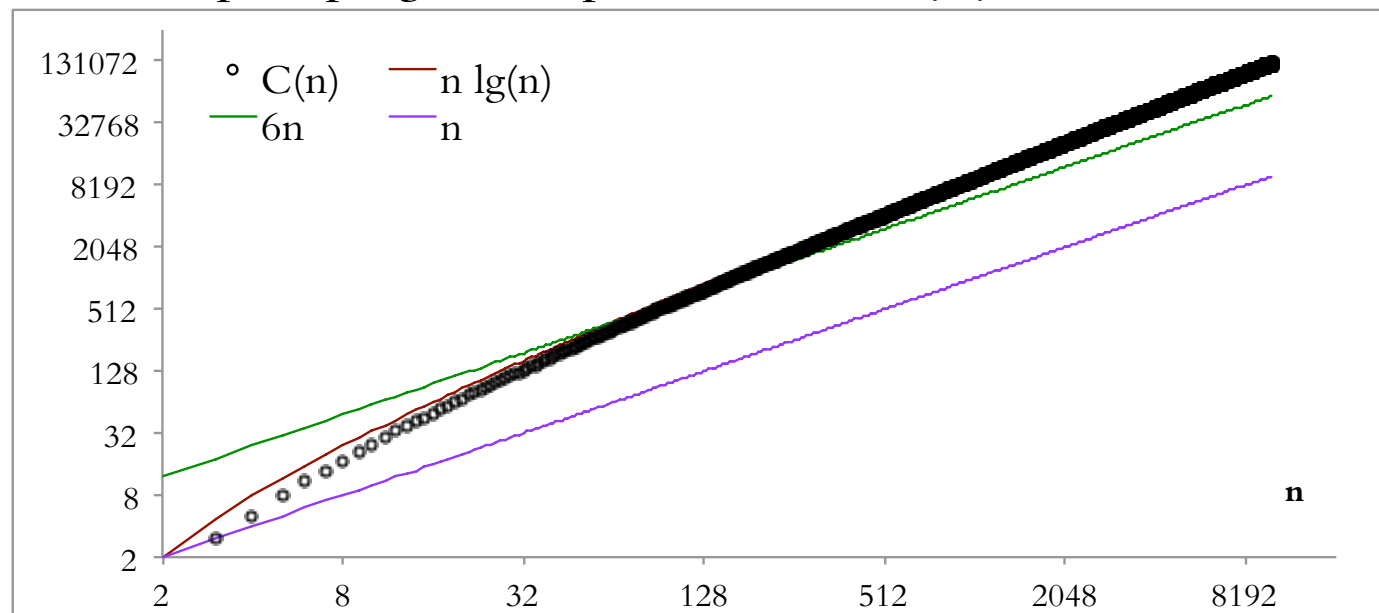
$$R(\ell) = \begin{cases} 0 & \{ \ell \leq \epsilon \} \\ R(\lfloor \ell/2 \rfloor) + 1 & \{ \ell > \epsilon \} \end{cases}$$



# Récurrance — mergesort

$$C(n) = \begin{cases} 0 & \{n = 0, 1\} \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + (n - 1) & \{n > 1\} \end{cases}$$

on peut écrire un petit programme pour calculer  $C(n)$  à  $n = 0, 1, 2, 3, \dots$  !



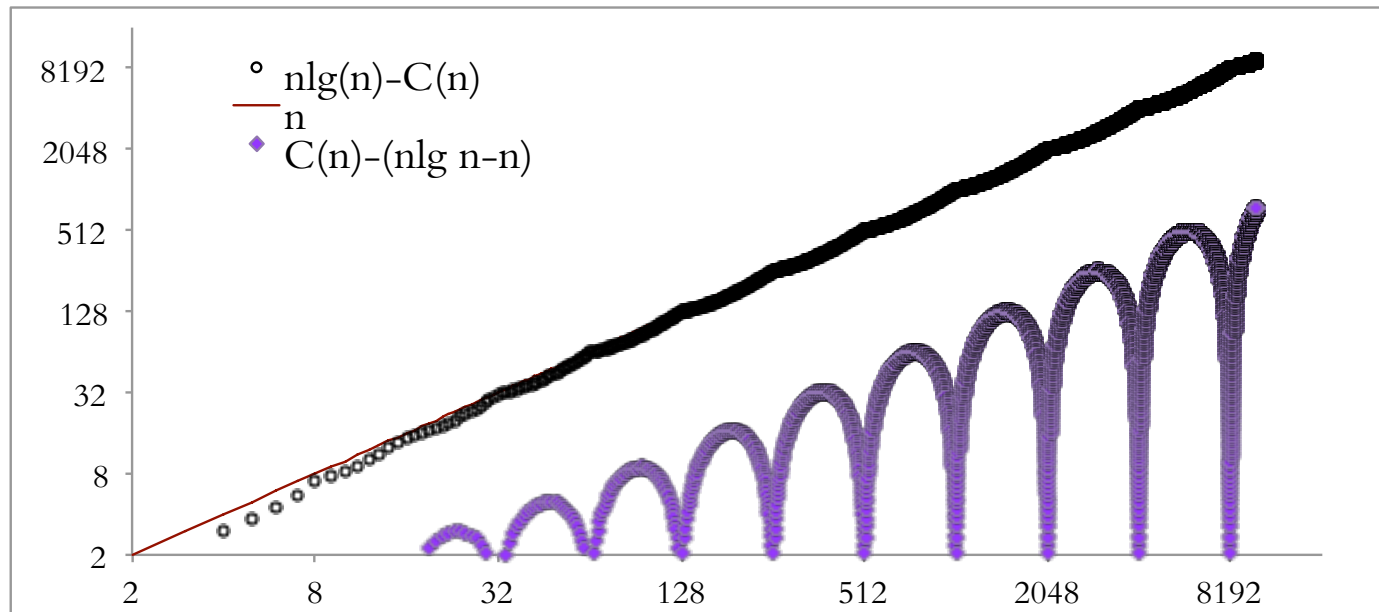
certainement non pas linéaire :  $C(n) \sim n \lg n$

# Mergesort 2

Notre hypothèse :  $C(n) \sim n \lg n$

mais  $C(n)$  est une beauté en vérité !

composante périodique — typique pour des récurrences avec arrondi

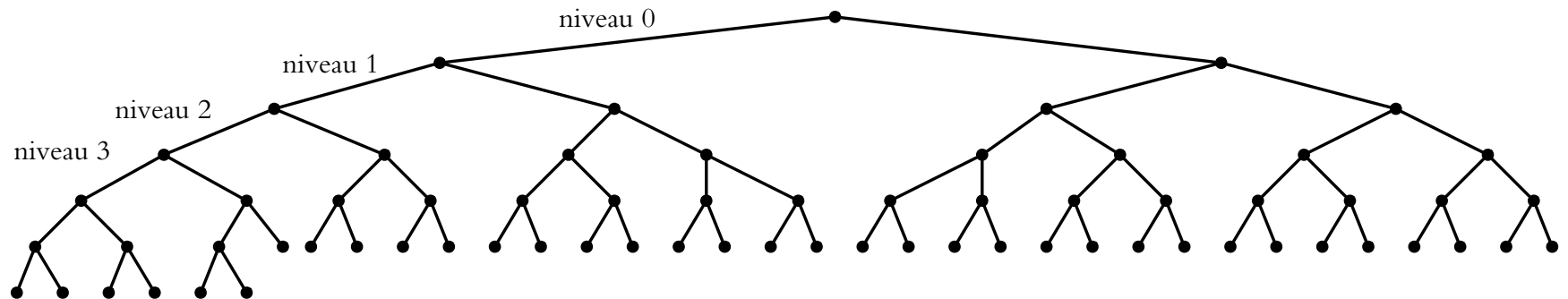


solution exacte avec partie fractionnaire  $\{x\} = x - \lfloor x \rfloor$

$$C(n) = n \lg n - nA(\{\lg n\}) + 1 \quad A(u) = u + 2^{1-u} - 1 \in [0.9139 \dots, 1]$$

# Mergesort 3

arbre de récurrences (sous-tâches aux nœuds) : arbre binaire complet,  $n - 1$  à chaque niveau, total est  $\sim h(n - 1)$



**Thm.** La hauteur  $h$  d'un arbre binaire à  $n$  nœuds internes est bornée par  $\lceil \lg(n + 1) \rceil \leq h \leq n$ .

**Démo.**  $n_k$  : nœuds internes au niveau  $k = 0, 1, 2, \dots, h - 1$ ;  $n = \sum_{k=0}^{h-1} n_k$

$$1 \leq n_k \leq 2n_{k-1}$$

# Notation asymptotique

$$f(n) \sim cg(n) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$$

$g(n)$  est l'ordre de croissance — choix à la carte sur l'**échelle standard** :

<b>ordre</b>	<b><math>f(n)</math></b>
constante	1
logarithmique	$\log n$
linéaire	$n$
linéarithmique	$n \log n$
quadratique	$n^2$
cubique	$n^3$
polynomial	$n^a \log^b n$
exponentiel	$A^n ; A > 1$

# Notation de Landau

**Déf.** presque = nombre fini d'exceptions

Soit  $f$  et  $g$  deux fonctions sur les nombres entiers telles que  $f(n), g(n) > 0$  pour presque tout  $n$ .

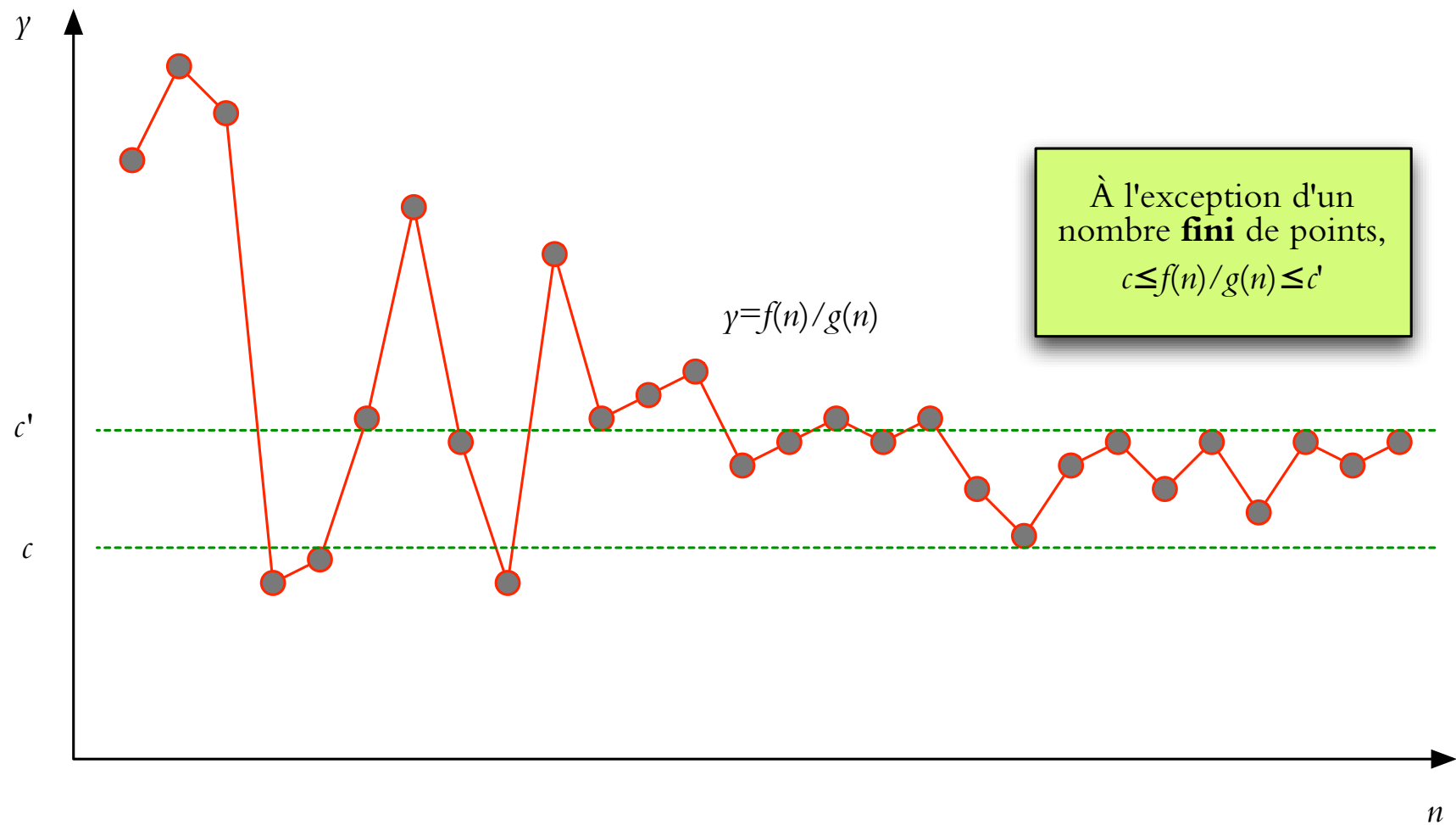
**[grand O]**  $f = O(g)$  si et seulement si [ssi]  $\exists c > 0$ : tel que  $\frac{f(n)}{g(n)} \leq c$  pour presque tout  $n$ .

**[grand Omega]**  $f = \Omega(g)$  ssi ou  $\exists c > 0$ : tel que  $\frac{f(n)}{g(n)} \geq c$  pour presque tout  $n$ . (Et donc  $g = O(f)$ .)

**[Theta]**  $f = \Theta(g)$  ssi  $f = O(g)$  et  $g = O(f)$ , ou  $\exists c, c' > 0$  tels que  $c \leq f(n)/g(n) \leq c'$  pour presque tout  $n$ .

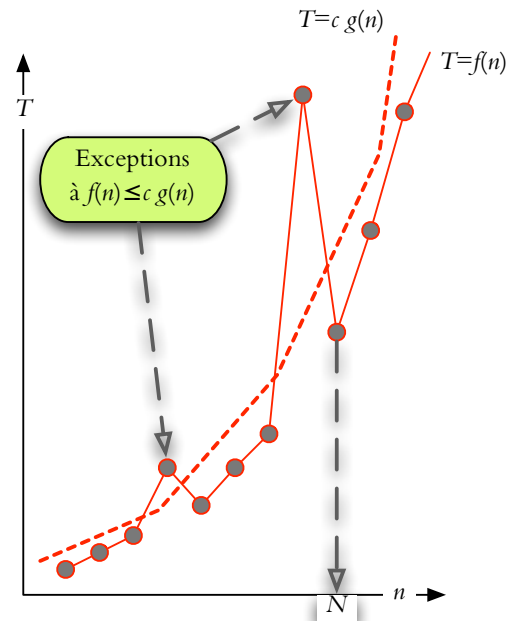
**[petit o]**  $f = o(g)$  ssi  $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$ , ou  $\forall c > 0$ ,  $\frac{f(n)}{g(n)} \leq c$  pour presque tout  $n$

# Theta



# Grand-O

nombre fini d'exceptions  $\leftrightarrow$  seuil pour convergence



$f = O(g)$  si et seulement s'il existe  $c > 0$  et  $N \geq 0$  tels que  $f(n) \leq c \cdot g(n)$  pour tout  $n \geq N$ .

# Expansion asymptotique

on a une fonction  $f(n)$  et on veut caractériser sa croissance sur l'échelle standard :

$$f(n) \sim c_0 g_0(n) + c_1 g_1(n) + c_2 g_2(n) + \dots$$

avec  $g_{k+1} = o(g_k)$  ( $g_k$  «négligeable» par rapport à  $g_k$ )

précision croissante selon le but de la caractérisation :

$$f = O(g_0)$$

$$f = c_0 g_0 + O(g_1)$$

$$f = c_0 g_0 + c_1 g_1 + O(g_2)$$

...

$O$  : info sur la grandeur de termes omis



# Logarithmes

$$\lg x = \log_2 x \quad \text{et} \quad \ln x = \log_e x \quad \{x > 0\}$$

$$\log(xy) = \log x + \log y; \log(x^a) = a \log x$$

$$2^{\lg n} = n; n^n = 2^{n \lg n}; \log_a n = \frac{\lg n}{\lg a} = \Theta(\lg n) \quad ; a^{\lg n} = n^{\lg a}$$

logarithmes itérés :  $\log \log n, \log \log \log n, \dots$

→ on omet la base du logarithme dans  $O, o, \Theta, \Omega$

## Comparer :

mergesort prend  $\Theta(n \log n)$  — énoncé théorique non-testable

mergesort prend  $\sim n \lg n + O(n)$  — correspond à un hypothèse + info sur l'erreur

# Fonctions notables

**nombre harmonique** : logarithmique

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \ln n + \gamma + O(1/n) \sim \ln n = \Theta(\log n)$$

avec  $\gamma = \lim_{n \rightarrow \infty} (H_n - \ln n) = 0.5772 \dots$  (constante d'Euler)

**Fibonacci** : exponentiel

$$F(n) = \frac{\phi^n}{\sqrt{5}} + O(\phi^{-n}) \sim \frac{\phi^n}{\sqrt{5}} = \Theta(\phi^n)$$

avec  $\phi = (1 + \sqrt{5})/2 = 1.6 \dots$

**factorielle** (Stirling) : superexponentiel

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n = \Theta\left(n^{n+1/2} e^{-n}\right)$$

mais  $\log(n!)$  est linéarithmique :

$$\sum_{k=1}^n \ln k = \ln(n!) = n \ln n - n + O(\log n) = \Theta(n \log n)$$

# Déterminer l'ordre de croissance

arithmétique :

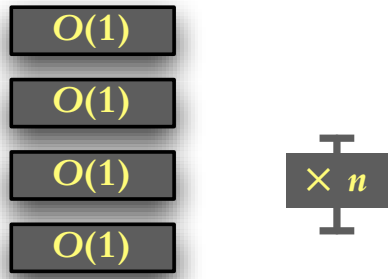
$$c \cdot f = O(f) \tag{1}$$

$$\underbrace{O(f) + O(g)} = \underbrace{O(f + g)} \tag{2}$$

pour tout  $h = O(f)$  et  $h' = O(g)$  il existe  $h'' = O(f + g)$  t.q.  $h + h' = h''$

$$O(f) \cdot O(g) = O(f \cdot g) \tag{3}$$

# Algo itératif — max

MAX-ITER ( $x[0..n - 1]$ )		
M1 initialiser $\text{max} \leftarrow -\infty$	$O(1)$	
M2 <b>for</b> $i \leftarrow 0, \dots, n - 1$ <b>do</b>	$O(1)$	
M3 <b>if</b> $x[i] > \text{max}$ <b>then</b> $\text{max} \leftarrow x[i]$	$O(1)$	
M4 <b>return</b> $\text{max}$	$O(1)$	

Le temps de calcul pour un tableau de taille  $n$  est  $T(n) = O(1) + O(1) + n \cdot O(1) + O(1) = O(n)$ .

# Algo itératif — Euclide

GCD( $a, b$ ) // $\{b \leq a\}$		
E1 <b>while</b> ( $b \neq 0$ )	$O(1)$	$\times O(\log b)$
E2 $c \leftarrow a \bmod b$	$m(a, b)$	
E3 $a \leftarrow b; b \leftarrow c$	$O(1)$	
E4 <b>return</b> $a$	$O(1)$	

nombre d'itérations  $O(\log b)$

$$F(k) \leq b < F(k + 1) \Rightarrow k = \log_{\phi} b + O(1) = O(\log b)$$

temps de calcul :

$$T(a, b) = O(1) + O(\log b) \times \left( \underbrace{O(\log a \log b)}_{\text{division modulaire}} + O(1) \right) = O(\log^2 b \log a)$$

logarithmique dans  $a, b$ , polynomial dans le nombre de bits (taille de l'entrée)