4 Analyse d'algorithmes

4.1 Analyse de complexité d'algorithmes

W(fr)

On veut comprendre l'efficacité d'un algorithme pour une tâche quelconque : on caractérise les ressources (temps, mémoire) nécessaires pour l'exécution.

- ⋆ **usage de mémoire** ou «complexité d'espace» (*space complexity*) : c'est le mémoire de travail nécessaire à part de stocker l'entrée même.
- ⋆ **temps d'exécution** ou «complexité de temps» (*time complexity*) : c'est le temps d'exécution dans un modèle formel de calcul.

4.2 Notation asymptotique

W(fr)

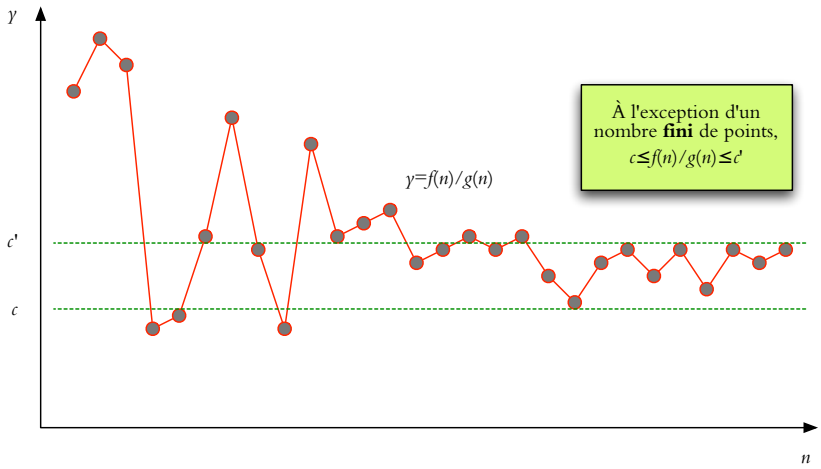
Typiquement, la complexité est caractérisée en **notation asymptotique**, comme une fonction de la taille de l'entrée. Ceci permet de comprendre et de prédire le comportement de l'algorithme. On utilisera «presque tout» dans un sens bien défini : «**presque tout *n***» veut dire qu'il y a juste un nombre fini (même aucune) d'exceptions.

Définition 4.1. *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.

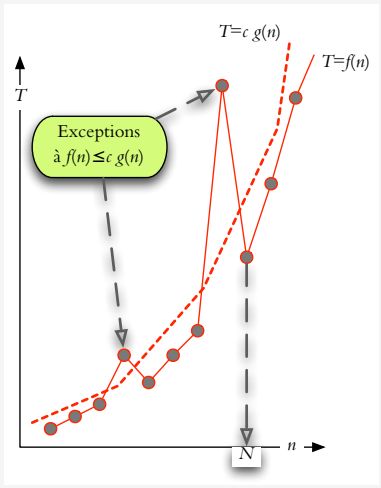
[grand Omega] $f = \Omega(g)$ *ssi ou* $\exists c > 0$: *tel que* $\frac{f(n)}{g(n)} \geq c$ *pour presque tout* n *(donc* $g = O(f)$*)*

[Theta] $f = \Theta(g)$ *ssi* $f = O(g)$ *et* $g = O(f)$*, ou* $\exists c, c' > 0$ *tels que*



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

[petit omega] $f = \omega(g)$ *ssi* $g = o(f)$*, ou* $\forall c > 0$, $\frac{f(n)}{g(n)} \geq c$ *pour presque tout* n.



Il existe des définitions équivalentes : par exemple, $f(n) = O(g(n))$ si et seulement s'il existe $c > 0$ et $N \geq 0$ tels que $f(n) \leq c \cdot g(n)$ pour tout $n \geq N$. En général, une proposition $\mathcal{P}(n)$ vaut pour presque tout n, ssi il existe $N < \infty$ tel que $\mathcal{P}(n)$ vaut pour tout $n \geq N$.

4.3 Règles d'arithmétique

$$c \cdot f = O(f) \tag{4.1a}$$

$$\underbrace{O(f) + O(g)}_{\text{pour tout } h = O(f) \text{ et } h' = O(g)} = \underbrace{O(f+g)}_{\text{il existe } h'' = O(f+g) \text{ t.q. } h + h' = h''} \tag{4.1b}$$

$$= O\big(\max\{f, g\}\big) \tag{4.1c}$$

$$O(f) \cdot O(g) = O(f \cdot g) \tag{4.1d}$$

4.4 Quelques fonctions notables

Logarithmes.

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

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

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

Dans la notation asymptotique, on ne montre pas la base du logarithme parce qu'avec une base différente, on change seulement la facteur constante : au lieu de $O(\log_{10} n)$ ou $O(\ln n)$, on écrit simplement $O(\log n)$.

Factorielle. $n! = 1 \cdot 2 \cdots n$. Approximation de Stirling : W(fr)

$$n! = \big(1 + o(1)\big)\sqrt{2\pi n}\Big(\frac{n}{e}\Big)^n = \Theta\big(n^{n+1/2}e^{-n}\big)$$

D'où $\sum_{k=1}^{n} \ln k = \ln(n!) = \big(1 + o(1)\big) n \ln n$, donc $\log(n!) = \Theta(n \log n)$.

Nombre harmonique. $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n} \approx \ln n + \gamma = \big(1 + o(1)\big) \ln n = \Theta(\log n)$ où $\gamma = \lim_{n\to\infty}(H_n - \ln n) = 0.5772\cdots$ est la constante d'Euler. W(fr)

Nombres Fibonacci. $F(n) = \left(\frac{1}{\sqrt{5}} + o(1)\right)\phi^n = \Theta(\phi^n)$ avec $\phi = (1+\sqrt{5})/2$.

4.5 Modèles de calcul et pseudocode

L'analyse formel doit assumer un modèle mathématique spécifiant les **instructions élémentaires**, et le temps d'exécution associé avec chaque instruction. Les instructions élémentaires forment le vocabulaire pour décrire un algorithme. La **machine de Turing** comprend seulement un ruban «magnétique», et une tête de lecture-écriture. La gestion de mémoire est difficile avec une machine de Turing (la tête glisse par une case à la fois). Dans une **machine RAM** (*Random Access Machine* — machine à accès direct) on peut adresser le mémoire directement, et y stocker les données. L'ensemble d'instructions est proche des langages assembleurs des CPUs courants.

W(fr) W(fr)

Modèles équivalents. La notation asymptototique permet d'énoncer des résultats qui ne dépendent pas de l'implantation concrète d'un algorithme. En particulier, la taille de l'entrée peut être mesurée en bits, octets, ou d'autres mesures convénients (nombre d'éléments dans une collection), il faut juste assurer qu'une entrée de taille t peut être representée sur $\Theta(t)$ bits. En plus, les instruction d'un modèle de calcul s'exécutent en $\Theta(1)$ sur un autre modèle de puissance équivalente (p.e., CPUs modernes sont équivalents à une machine RAM). Par conséquence, si on trouve qu'un algorithme prend $O(n)$ sur un tableau de taille n, le résultat reste valide dans des implantations, indépendamment de CPU, langage de programmation, ou encodage du tableau.

Pseudocode. On décrit souvent un algorithme en «pseudocode» : programme de lignes, mémoire infini, nombre fini de variables, instructions d'affectation, arithmétiques, et de contrôle. Le syntaxe dépend de l'auteur, mais les instructions en pseudocode sont faciles à implanter sur une machine RAM (ou en un langage de programmation), et s'exécutent typiquement en un temps constant.

4.6 Déterminer le temps de calcul

Algorithme sans récurrence. On peut exploiter les règles d'arithmétique directement dans l'analyse du pseudocode.

Exemple 4.1. On prend l'exemple de rechercher le maximum dans un tableau.

```
MAX-ITER(x[0..n − 1])
M1  initialiser max ← −∞                          O(1)
M2  for i ← 0, . . . , n − 1 do                   O(1)
M3      if x[i] > max then max ← x[i]             O(1)   × n
M4  return max                                    O(1)
```

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

Exemple 4.2. L'algorithme d'Euclide (v. §1.5, notes de cours 01) est important dans des applications cryptographiques, où on calcule avec des entiers de grande taille (par exemple, 2048 bits). La taille de l'entrée est donc mesurée dans le nombre de bits pour représenter les paramètres : $\lg a + \lg b$.

```
GCD(a, b) // {b ≤ a}
E1  while (b ≠ 0)                                 O(1)
E2      c ← a mod b                               m(a,b)   × O(log b)
E3      a ← b; b ← c                              O(1)
E4  return a                                      O(1)
```

Il faut considérer le coût de division entière (mod) : Ligne E2 prend $m(a,b) = O\big((\log a)(\log b)\big)$ temps. Théorème 1.2 montre que le nombre d'itérations est borné par $O(\log b)$ (on cherche le nombre Fibonacci $F(k) \le b < F(k+1)$; donc $k = \log_\phi b + O(1) = O(\log b)$). Par conséquence, l'algorithme d'Euclide s'exécute en $T(a,b) = O(1) + O(\log b) \times \Big(O(\log a \log b) + O(1)\Big) = O(\log^2 b \log a)$ temps, donc en temps *polynomiel* dans la taille de l'entrée, même pour des entiers très grands. ♠

Algorithme récursif. On commence par exprimer le temps de calcul par une équation de récurrence, et on cherche sa solution.

Exemple 4.3. On prend l'exemple de rechercher le maximum dans un tableau.

```
    MAX-REC(x[0..n−1], i)
    // appel initiel avec i = 0
R1  if i = n then return −∞                 T(0)=O(1)
R2  else
R3      m ← MAX-REC(x, i+1)                 T(j-1)
R4      if x[i] > m then return x[i]        O(1)
R5      else return m                       O(1)
```

Soit $T(j)$ le temps de calcul pour exécuter MAX-REC$(x[0..n-1], n-j)$. On a la récurrence

$$T(j) = \begin{cases} O(1) & \text{si } j = 0 \\ T(j-1) + O(1) & \text{si } j > 0 \end{cases} \tag{4.2}$$

Pour démontrer que la solution de la récurrence est $T(j) = O(j)$, on utilise une preuve par induction. Par Équation (4.2), il existe a tel que $T(j) \le T(j-1)+a$ pour tout $j > 0$. Soit $c = T(0)+a$. Hypothèse d'induction : $T(j) \le c \cdot j$ pour un $j > 0$. Cas de base : l'hypothèse d'induction vaut pour $j = 1$. Cas inductif : $T(j) \le T(j-1)+a \le c(j-1)+a \le cj$ (car $c = T(0) + a > a$).

On conclut que $T(j) \le cj$ pour tout $j > 0$, d'où $T(j) = O(j)$. ♠

Substitution de variables.

Exemple 4.4. On prend l'exemple du calcul de puissances. On veut calculer x^n où $n \ge 0$ est un entier non-négatif et $x \in \mathbb{R}$. La clé à une solution efficace est la récurrence suivante

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

```
P1  Algo POWER(x, n)          // (calcule x^n, n entier)
P2  if n = 0 then return 1
P3  else
P4      y ← POWER(x, ⌊n/2⌋)
P5      if n mod 2 = 0 then return y × y
P6      else return x × y × y
```

Ceci mène à la récurrence $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(1)$ pour $n > 0$. On trouve la solution par **substitution de variables** : on regarde le nombre de bits dans la représentation binaire de n. Si on dénote ce nombre par b, on a la récurrence $T'(b) = T'(b-1)+O(1)$ avec la solution $T'(b) = O(b)$ (v. Exemple 4.3). Or, $b = \lceil \lg(n+1) \rceil = O(\log n)$, donc $T(n) = T'\big(O(\log n)\big) = O(\log n)$, donc reste linéaire dans la taille b même pour des entiers n très grands. ♠