4 Analyse d'algorithmes

4.1 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)

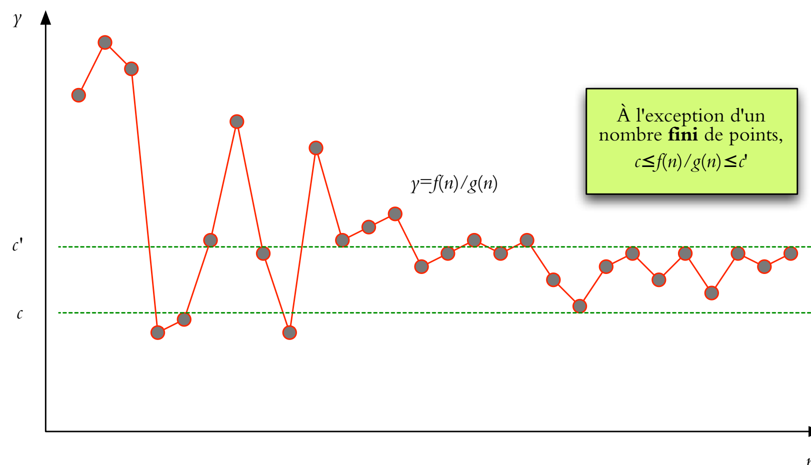
Typiquement, la complexité est caractérisée en **notation asymptotique**, comme une fonction de la taille de l'entrée. Ceci permet de comprendre et de prédire le comportement de l'algorithme. On utilisera «presque tout» dans un sens bien défini : «**presque tout *n***» veut dire qu'il y a juste un nombre 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]* $\exists c > 0\colon$ *tel que* $\frac{f(n)}{g(n)} \le c$ *pour presque tout* n.

[grand Omega] $f = \Omega(g)$ *ssi ou* $\exists c > 0\colon$ *tel que* $\frac{f(n)}{g(n)} \ge c$ *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*

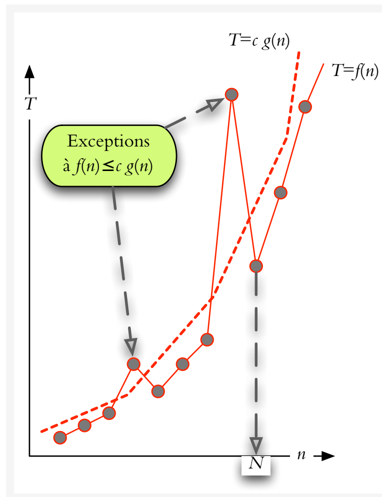


[petit o] $f = o(g)$ *ssi* $\lim_{n\to\infty} f(n)/g(n) = 0$*, ou* $\forall c > 0$*,* $\frac{f(n)}{g(n)} \le c$ *pour presque tout* n

[tilde] $f \sim g$ *ssi* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$*, ou* $f(n) = \Bigl(1 \pm o(1)\Bigr) g(n)$

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.



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 $n \geq N$.

4.2.1 Règles d'arithmétique

Les equations suivantes sont valides avec O, Θ, Ω, et o.

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

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

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

4.2.2 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 ($\log_a f = c \cdot \log_b f$ avec $c = \log_a b = 1/\log_b a$) : au lieu de $O(\log_{10} n)$ ou $O(\ln n)$, on écrit simplement $O(\log n)$.

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

W(fr)

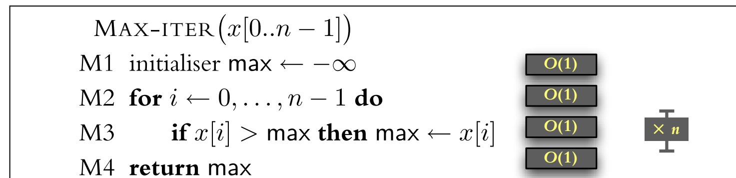
Nombres Fibonacci. $F(n) = \Bigl(\frac{1}{\sqrt{5}} + o(1)\Bigr)\phi^n = \Theta(\phi^n)$ avec $\phi = (1+\sqrt{5})/2$.

Factorielle. $n! = 1 \cdot 2 \cdots n$. Approximation de Stirling : $n! = \big(1 + o(1)\big)\sqrt{2\pi n}\left(\frac{n}{e}\right)^n = \Theta\big(n^{n+1/2}e^{-n}\big)$. Même si par intution n^n semble dominer l'expression, on doit retenir tout exposant de n dans un terme quand on simplifie vers Θ. Par contre, $\sum_{k=1}^{n} \ln k = \ln(n!) = \big(1 + o(1)\big) n \ln n$, donc $\log(n!) = \Theta(n \log n)$.

4.3 Déterminer le temps de calcul

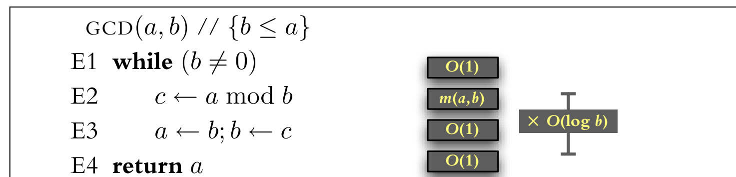
Algorithme sans récurrence. On peut exploiter les règles d'arithmétique directement dans l'analyse du pseudocode. Pour établir le temps de calcul f, on examine le code et additionne le coût de chaque instruction. On sépare f par ses termes (règle (4.1b)), et on ignore les facteurs constantes (règle (4.1a)).

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



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

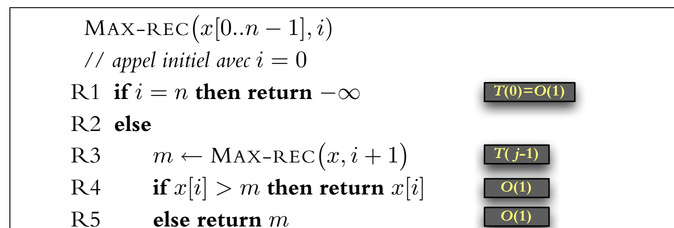
Exemple 4.2. L'algorithme d'Euclide (v. §1.6, Note 1) est important dans des applications cryptographiques, où on calcule avec des entiers de grande taille (par exemple, 2048 bits). La taille de l'entrée est donc mesurée dans le nombre de bits pour représenter les paramètres : $\lg a + \lg b$.



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 $F(k) \leq b < F(k+1)$; 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 *polynomial* 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.



Soit $T(j)$ le temps de calcul pour exécuter $\text{MAX-REC}(x[0..n-1], n-j)$. 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. ♠

4.4 Aspects pratiques

La notation asymptotique classique (O, Θ, Ω) est préferée dans la description théorique de l'efficacité d'algorithmes et de structures de données. Si un algorithme est de $O(n^2)$, on sait que même dans le pire cas, il va prendre un temps quadratique dans la taille du problème. Mais en pratique, on voudrait *prédire* le comportement de l'algorithme sur des *entrées typiques* (et non pas seulement le pire cas). On voudrait aussi savoir un peu plus sur les *constantes cachées* par O, pour pouvoir choisir entre deux algorithmes de $O(n^2)$. Par exemple, le tri rapide (quicksort) prend $O(n^2)$ dans le pire cas, mais il est en général préférable au tri par fusion (mergesort) qui est de $\Theta(n \log n)$ parce que quicksort prend $\Theta(n \log n)$ pour presque toute entrée, et les constantes cachées sont plus petites qu'en mergesort.

4.4.1 Temps de calcul comme coût

Afin de développer une caractérisation d'utilité pratique, on procède comme suit.

1. Développer un modèle de l'entrée et définir la notion de la «taille» de l'entrée. Le modèle doit permettre la génération de données à l'entrée pour mesurer la performance d'une implantation sur l'ordinateur.
2. Identifier la partie du code le plus fréquemment exécutée (qui domine la croissance du temps de calcul). Pour un algorithme itératif, cette partie est à l'intérieur de la boucle le plus profondément imbriquée.

3. Définir un modèle de coût pour le temps de calcul. En particulier, on veut écrire le temps de calcul avec le coût d'opérations typiques dans le contexte du problème. Exemples d'opération typique : comparaison de deux éléments lors du tri d'un fichier, accès à une cellule dans un tableau, ou une opération arithmétique (algorithme d'Euclid, exponentiation).
4. Déterminer la fréquence d'exécuter les opérations typiques en fonction de la taille de l'entrée.

Exemple. On prend l'exemple de chercher le maximum dans un tableau par itération sur les éléments (Exemple 4.1). (1) On mesure la taille de l'entrée par le nombre de ses éléments n et on assume que les éléments sont dans un ordre quelconque (= permutation au hasard, uniformement distribuée). (2) La boucle intérieure contient la comparaison «**if** $x[i] > \mathsf{max}$ **then** $\mathsf{max} \leftarrow x[i]$». (3) On caractérise le temps de calcul par la fréquence de comparaisons ($x[i] > \mathsf{max}$). (4) La boucle s'exécute $(n-1)$ fois quand $n > 0$. En conclusion, l'algorithme fait $C(n) \sim n$ comparaisons arithmétiques pour un tableau de taille n.

4.4.2 Mesurer et prédire le temps de calcul

L'avantage de la notation tilde est qu'elle exprime un *hypothèse scientifique* sur le comportement de l'algorithme qu'on peut tester empiriquement. En particulier, on peut implanter l'algorithme, générer des entrées de tailles différentes, et mesurer le temps ou explicitement compter la fréquence d'exécution d'opération typiques.

Mesurer le temps de calcul. Le temps d'exécution peut être mesuré :

- ⋆ dans le code (Java) :

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

- ⋆ dans le shell (Linux/Unix)

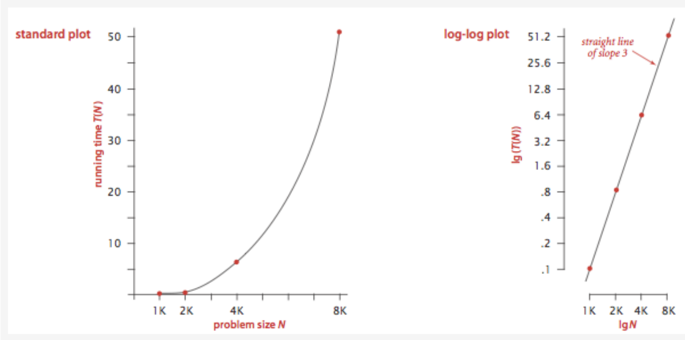
```
% time java -cp Monjar.jar mabelle.Application
0.283u 0.026s 0:00.35 85.7% 0+0k 0+53io 0pf+0w
```

temps CPU («user time»)
temps CPU («kernel time»)
temps réel («wallclock time»)
utilisation du CPU
temps d'exécution (seconds)

- ⋆ ou bien dans un environnement de développement de logiciel (Netbeans, Eclipse).

Conception d'expériences. Dans une étude empirique, on veut performer plusieurs mesures. (1) On répète l'expérience pour entrées différentes de même taille. La moyenne montre le comportement typique, et les répétitions permettent de comprendre la dispersion statistique du temps de calcul. (2) On répète l'expérience pour entrées de taille différente. Il est particulièrement utile de considérer des tailles multipliées par la même facteur (2 ou 10). Si on a l'hypothèse que le temps de calcul est $T(n) \sim a \cdot n^b$, avec des constantes quelconques $a, b > 0$, on a

$$\frac{T(2n)}{T(n)} \sim \frac{a(2n)^b}{an^b} = 2^b. \tag{4.3}$$



En fait, Equation (4.3) permet de déduire b par régression linéaire même si on ne le sait pas. On a $\log T(n) \sim a' + b \log n$, et donc on peut déterminer b par la pente dans un répère log-log. À gauche : temps d'éxécution d'un algorithme avec $T(N) \sim aN^3$.
Les mesures servent aussi à prédire (par extrapolation) le temps de calcul de l'implantation examinée pour des entrées de grande taille.

4.4.3 Usage de mémoire en Java

L'usage de mémoire par un programme Java peut varier avec la plate-forme, mais en général, on peut assumer 4 octets pour `int` et `float`, 8 octets pour `long` et `double`, ainsi que pour les objets (adressage 64-bit). Tableau avec n éléments : $a + n \cdot t$ octets où $a = 16..24$ est la taile de l'en-tête et t est la taille d'un élément (référence ou type primitif).