Fiche 2: Ordres de grandeur de complexité

Complexité d'un algorithme

1. On rappelle qu'en informatique un *algorithme* se définit :
   — relativement à un *modèle de calcul*
   — comme un ensemble d'instructions élémentaires autorisées dans ce modèle et permettant,
      — étant donnée une *configuration initiale* qui encode une donnée d'entrée sur laquelle on veut appliquer l'algorithme (pour calculer la valeur correspondante d'une fonction ou pour répondre à une question d'un problème de décision)
      — d'atteindre une *configuration terminale* (qui correspond au résultat de l'algorithme sur la donnée fournie en entrée dans la configuration initiale) par une suite finie de *pas de calcul*, chaque pas de calcul correspondant à l'application d'une instruction élémentaire sur la configuration en cours.

   Par exemple, dans le modèle des MT simples, une MT Z est définie par un sous-ensemble fini de symboles de $E(Z) \subset S$ (contenant au moins B), par un ensemble fini $E(Z) \subset Q$ d'états (contenant au moins q_0), et par un ensemble consistant de quadruplets. Elle constitue un algorithme dont les entrées possibles sont les mots γ de S^+ tels que Z s'arrête quand elle est lancée sur la configuration initiale $q_0\gamma$.
2. La *complexité* (en temps) d'un algorithme (dans un modèle de calcul donné) est une fonction $\lambda n\ C(n)$ qui associe à un entier n le nombre maximum de pas de calcul nécessaires pour atteindre une configuration terminale à partir d'une configuration initiale encodant une donnée d'entrée par un mot de taille n.
3. La complexité ainsi définie est souvent appelée complexité *au pire*, car pour une taille donnée n, pour estimer le nombre *maximum* de pas de calcul nécessaires pour finir le calcul, il faut déterminer les mots de taille n qui font faire le plus de pas de calcul à l'algorithme lors de son exécution. On peut aussi définir la complexité dans le cas le meilleur, ou en moyenne (ce qui est beaucoup plus difficile à calculer).
   Question 1 : Soit une MT simple Z qui, sur des mots $|^{k+1}$ fournis en entrée, efface le premier |, puis, si le nombre de | restants est pair, s'arrête, sinon duplique le mot $|^k$ à la suite sur la bande de lecture-écriture et s'arrête.
   (a) donner le principe d'une telle MT (en particulier pour déterminer si le nombre de | est pair) et en déduire sa complexité dans le cas le pire et dans le cas le meilleur.
   (b) En interprétant un mot $|^{k+1}$ fourni en entrée comme le codage unaire de l'entier k, et le nombre de | au moment de l'arrêt comme l'entier résultat, dire quelle fonction $\Psi_Z^{(1)}$ calcule la MT Z.
4. La complexité d'un algorithme dépend non seulement du modèle de calcul considéré pour le modéliser, mais aussi du choix de codage des données d'entrée. En particulier, la taille du codage d'un entier peut varier de plusieurs ordres de grandeur selon qu'il est codé en unaire ou binaire. Par exemple, la taille du codage unaire de l'entier 1024 est 1025 s'il est fourni en entrée car on le code par le mot $|^{1025}$, alors que la taille de son codage binaire n'est que que 11.
   Question 2 : Donner le codage binaire de 1024. Rappel : le codage binaire d'un entier i, noté $b(i)$, est le mot $b_k b_{k-1} \dots b_0$ où les symboles b_j sont appelés *bits* et valent 0 ou 1, avec $b_k = 1$ si $i \neq 0$, et $b_0 = 0$ $(k = 0)$, si $i = 0$, et qui est tel que : $i = \Sigma_{j=0}^{k} b_j 2^j$.
   La taille binaire de l'entier i, notée $l(i)$, est la taille du mot qui le code en binaire : $l(i) = |b(i)| = k + 1$.
   De façon générale, la taille du codage unaire d'un entier i est un mot de taille $i + 1$, alors que son codage binaire est un mot de taille $\lfloor log_2(i) \rfloor + 1$ (pour $i > 0$).
   Question 3 : Le montrer. Pour cela, on rappelle que :
   — la notation log_2 représente la fonction logarithme en base 2 définie sur les réels strictement positifs par, pour tout $x > 0$: $log_2(x) = \frac{Log(x)}{Log(2)}$ (où Log est la fonction Logarithme Népérien), et qui est telle que :
      — $log_2(2) = 1$ et $log_2(1) = 0$
      — pour tout $x, y > 0$, $log_2(x \times y) = log_2(x) + log_2(y)$ et donc : $log_2(x^n) = n \times log_2(x)$
      — $lim_{n \to +\infty} \frac{log_2(n)}{n} = 0$ et $lim_{n \to +\infty} \frac{n}{log_2(n)} = +\infty$
   — pour n'importe quel réel $x \geq 0$, le "plancher" de x, noté $\lfloor x \rfloor$, est l'entier p tel que $p \leq x < (p + 1)$.
   Question 4 :
   (a) Donner une MT Z' définie sur l'ensemble de symboles $A(Z') = \{B, 0, 1\}$, qui accepte en entrée un entier codé en binaire (sa configuration initiale sera donc $c_0 = q_0\ b(k)$), et qui calcule la même fonction de $\mathbb{N}$ dans $\mathbb{N}$ que la fonction $\Psi_Z^{(1)}$ de la Question 1(b), le résultat étant fourni sous la même forme que l'entrée, c'est-à-dire que Z' s'arrêtera dans la configuration $q_a\ b(\Psi_Z^{(1)}(k))$.

(b) Quelle est sa complexité ?

5. Pour pouvoir comparer des fonctions de complexité d'algorithmes, ce qui est important, ce ne sont pas les fonctions elle-mêmes, mais leur *ordre de grandeur (exact, supérieur ou inférieur)*, et en particulier, leur ordre de grandeur par rapport à des fonctions de référence pertinentes pour l'étude de la complexité, comme (supposer $n > 0$ pour les logarithmes) :
$\lambda n[log_2 n], \lambda n[n], \lambda n[n\ log_2 n], \lambda n[n^2], \lambda n[n^3], \ldots, \lambda n[n^k], \lambda n[2^n], \lambda n[n!], \lambda n[n^n]$

Ordres de grandeur (inférieur, supérieur, égal ou exact) de fonctions

On s'intéresse à des fonctions des entiers vers les réels positifs et pas seulement vers les entiers, pour pouvoir parler de coût moyen pour une certaine taille n de l'entrée d'un programme.

Definition 1 :

— Une fonction g est dite en *ordre de grandeur inférieur* à une fonction f, en abrégé : en *O(f)*,
si $(\exists k_1 > 0)\ (\exists n_1 \geq 0)\ (\forall n \geq n_1)\ |g(n)| \leq k_1.f(n)$.

— Une fonction g est dite en *ordre de grandeur supérieur* à une fonction f, en abrégé : en $\Omega(f)$,
si $(\exists k_2 > 0)\ (\exists n_2 \geq 0)\ (\forall n \geq n_2)\ g(n) \geq k_2.f(n)$.

— Une fonction g est dite en *ordre de grandeur égal* à une fonction f, en abrégé : en $\Theta(f)$,
si $(\exists k_1 > 0)\ (\exists k_2 > 0)\ (\exists n_0 \geq 0)\ (\forall n \geq n_0)\ k_2.f(n) \leq g(n) \leq k_1.f(n)$.

On a donc les ensembles de fonctions suivants :
$O(f) = \{\ g \mid (\exists k_1 > 0)\ (\exists n_1 \geq 0)\ (\forall n \geq n_1)\ [|g(n)| \leq k_1.f(n)\ ]\ \}$
$\Omega(f) = \{\ g \mid (\exists k_2 > 0)\ (\exists n_2 \geq 0)\ (\forall n \geq n_2)\ [\ g(n) \geq k_2.f(n)\ ]\ \}$
$\Theta(f) = O(f) \cap \Omega(f) = \{\ g \mid (\exists k_1 > 0)\ (\exists k_2 > 0)\ (\exists n_0 \geq 0)\ (\forall n \geq n_0)\ [k_2.f(n) \leq g(n) \leq k_1.f(n)\ ]\ \}$
On pourra utiliser les notations suivantes :
$O(f(n))$ pour $O(f)$, $\Omega(f(n))$ pour $\Omega(f)$, $\Theta(f(n))$ pour $\Theta(f)$.
Quand $g \in O(f)$ (respectivement $g \in \Omega(f)$, $g \in \Theta(f)$) on pourra noter :
$g(n) \in O(f(n))$ ou même $g(n) = O(f(n))$
(respectivement $g(n) \in \Omega(f(n))$ ou $g(n) = \Omega(f(n))$, $g(n) \in \Theta(f(n))$ ou $g(n) = \Theta(f(n))$)
On dira :
"g est grand O de f" (ou "$g(n)$ est grand O de $f(n)$"), ou encore "g (ou $g(n)$) a un ordre de grandeur inférieur à f (ou $f(n)$)."
"g est *Omega* de f" (ou "$g(n)$ est *Omega* de $f(n)$"), ou encore "g (ou $g(n)$) a un ordre de grandeur supérieur à f (ou $f(n)$)."
"g est *Theta* de f" (ou "$g(n)$ est *Theta* de $f(n)$"), ou encore "g (ou $g(n)$) a un ordre de grandeur égal à f (ou $f(n)$)."
On pourra ainsi écrire : $5.n^2 + 3 = O(n^2)$ (et aussi $5.n^2 + 3 = \Theta(n^2)$).
L'écriture la plus correcte reste : $5.n^2 + 3 \in O(n^2)$ (et aussi $5.n^2 + 3 \in \Theta(n^2)$).

Les ordres de grandeur des fonctions de référence sont notés :
$\Theta(log_2 n), \Theta(n), \Theta(nlog_2 n), \Theta(n^2), \Theta(n^3), \ldots, \Theta(n^k), \Theta(2^n), \Theta(n!), \Theta(n^n)$
On utilise les notations analogues pour O et Ω.

ATTENTION : il ne s'agit pas d'ordres de grandeur sur les *valeurs*, car le facteur constant peut être très grand, mais plutôt du *type de croissance*. Par exemple, si, pour $n \geq 10^4$ (déjà grand), on a $4 \times n^2 \leq g(n) \leq 4 \times 10^9 \times n^2$, $g \in \Theta(f)$, mais, si on prend $n = 10^5$, la valeur$f(n)$ est entre 4×10^{10} et 4×10^{19} : on ne peut pas dire qu'on a un ordre de grandeur de la valeur ! Et il ne s'agit même pas d'un ordre de grandeur de la croissance : dans cet exemple, elle est entre $8 \times n$ et $8 \times 10^9 \times n$.

Définition 2 : Soit $\lambda n[ref(n)]$ une fonction de référence, on dira qu'un algorithme (défini dans un modèle de calcul donné) a une complexité en ordre de grandeur égal (respectivement inférieur, supérieur) à ref si sa complexité est en $\Theta(ref)$ (respectivement $O(ref)$, $\Omega(ref)$).

Définition 3 :

— **Complexité linéaire :** en $\Theta(n)$
— **Complexité logarithmique :** en $\Theta(log_2 n)$
— **Complexité quadratique :** en $\Theta(n^2)$
— **Complexité polynomiale :** en $\Theta(n^k)$ (où k est une constante).
Partout, et ici en particulier, la notation lambda permet de préciser l'argument : "en $\lambda n[\Theta(n^k)]$".
— **Complexité exponentielle :** en $\Theta(2^n)$