All is well that ends well.
William Shakespeare

CHAPITRE VIII

Algorithme, correction, complexité

Objectifs

Un des objectifs de ce cours est de développer une notion de plus en plus précise de *complexité* de calcul. Pour ceci il faut préciser la *méthode* utilisée. Afin de choisir parmi les méthodes admises il faut d'abord une *spécification*. On est ainsi mené à regarder de plus près la notion *d'algorithme*. Ce chapitre discutera ce concept, en soulignant les trois points clés : *terminaison*, *correction*, *complexité*.

Sommaire

1. Qu'est-ce qu'un algorithme ?

Exemple 1.1. En utilisant l'arithmétique des entiers, on veut calculer le coefficient binomial $\binom{n}{k}$:

Spécification VIII.1 Calcul du coefficient binomial $\binom{n}{k}$

Entrée: deux entiers n et k
Sortie: le coefficient binomial $\binom{n}{k}$

En voici quatre méthodes candidates à résoudre ce problème.

Méthode VIII.2 Calcul du coefficient binomial $\binom{n}{k}$

si $k<0$ ou $k>n$ **alors retourner** 0 **sinon retourner** $\binom{n-1}{k-1}+\binom{n-1}{k}$

Cette méthode est carrément fausse. Voyez-vous pourquoi ?

Méthode VIII.3 Calcul du coefficient binomial $\binom{n}{k}$

si $k=0$ ou $k=n$ **alors retourner** 1 **sinon retourner** $\binom{n-1}{k-1}+\binom{n-1}{k}$

Cette méthode est fausse dans le sens qu'elle ne se termine pas toujours. (Expliquer pourquoi.) D'un autre coté, quand elle s'arrête, elle renvoie le résultat correct. (Le détailler.)

Méthode VIII.4 Calcul du coefficient binomial $\binom{n}{k}$

si $k<0$ ou $k>n$ **alors retourner** 0
si $k=0$ ou $k=n$ **alors retourner** 1
retourner $\binom{n-1}{k-1}+\binom{n-1}{k}$

Cette méthode se termine et donne le résultat correct. (Le montrer.)

Méthode VIII.5 Calcul du coefficient binomial $\binom{n}{k}$

si $k<0$ ou $k>n$ **alors retourner** 0
$b\leftarrow 1,\quad k\leftarrow \min(k,n-k)$
pour i **de** 1 **à** k **faire** $b\leftarrow \big(b\cdot(n-i+1)\big)/i$
retourner b

Cette méthode est également correcte, mais plus efficace que la précédente. (Pourquoi ?)

1.1. Algorithme = spécification + méthode. Comme on ne développera pas la théorie abstraite des algorithmes, nous nous contentons ici d'une définition pragmatique :

Définition 1.2. Un algorithme décrit un procédé, susceptible d'une réalisation mécanique, pour résoudre un problème donné. Il consiste en une *spécification* (ce qu'il doit faire) et une *méthode* (comment il le fait) :

- La **spécification** précise les données d'entrée avec les *préconditions* que l'on exige d'elles, ainsi que les données de sortie avec les *postconditions* que l'algorithme doit assurer. Autrement dit, les préconditions définissent les données auxquelles l'algorithme s'applique, alors que les postconditions résument le résultat auquel il doit aboutir.
- La **méthode** consiste en une suite finie d'instructions, dont chacune est soit une *instruction primitive* (directement exécutable sans explications plus détaillées) soit une *instruction complexe* (qui se réalise en faisant appel à un algorithme déjà défini). En particulier chaque instruction doit être exécutable de manière univoque, et ne doit pas laisser place à l'interprétation ou à l'intuition.

Exemple 1.3. Un algorithme décrit souvent le calcul d'une application $f : X \to Y$.

- La donnée d'entrée est d'un certain type, elle appartient à une classe X_0.
- La précondition précise le domaine de définition $X \subset X_0$, éventuellement restreint.
- La donnée de sortie est d'un certain type, elle appartient à une classe Y.
- La postcondition précise pour tout $x \in X$ la valeur cherchée $y = f(x)$.
- La méthode explicite les étapes du calcul en tout détail.

Dans l'exemple 1.1 les données d'entrée (n,k) appartiennent à l'ensemble $X_0 = \mathbb{Z}^2$, et on exige que la méthode s'applique à tout l'ensemble $X = X_0$. Le résultat est un entier et appartient donc à $Y = \mathbb{Z}$. La postcondition exige que $f(n,k)$ soit égal au coefficient binomial $\binom{n}{k}$.

Remarque 1.4 (première reformulation)**.** La spécification décrit le problème que l'on cherche à résoudre, en spécifiant les « conditions aux bords » : le point de départ est fixé par les données d'entrée, dont on suppose certaines préconditions ; le but est d'arriver à des données de sortie, pour lesquelles on doit garantir certaines postconditions. La méthode propose une solution à ce problème, c'est-à-dire un chemin allant du départ au but. (Le mot « méthode » est dérivé du grec ὁδός (*hodos*) signifiant « la voie ».) Bien sûr, pour un problème donné, plusieurs méthodes sont possibles, plusieurs chemins mènent au but.

Exemple 1.5. Il existe plusieurs solutions pour le problème de recherche d'un objet dans une liste (cf. chapitre V). Lorsque la précondition assure que les données sont ordonnées, la méthode peut en profiter. Ainsi l'algorithme suivant effectue correctement une recherche dichotomique sur une liste ordonnée. Travaille-t-il toujours correctement sur une liste non ordonnée ? (Donner un contre-exemple le cas échéant.)

Algorithme VIII.6 Recherche dichotomique

Entrée: un élément b et une liste ordonnée $A = (a_1, a_2, \ldots, a_n)$.

Sortie: le premier indice k tel que $a_k = b$, ou bien *non trouvé* si b n'appartient pas à A.

$i \leftarrow 1, \quad j \leftarrow n$
tant que $i < j$ **faire** $m \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$: **si** $b \leq a_m$ **alors** $j \leftarrow m$ **sinon** $i \leftarrow m+1$
si $a_i = b$ **alors retourner** i **sinon retourner** *non trouvé*

Remarque 1.6 (deuxième reformulation)**.** Souvent on interprète la *spécification* comme un contrat : si l'instance appelante assure les préconditions, alors la méthode appelée se charge d'assurer les postconditions. Comment elle le fait est explicité dans la *méthode*. (La méthode ne fait pas partie du contrat ; dans le cas extrême, elle peut rester un secret professionnel de l'instance appelée.) Il ne faut pas s'étonner si la méthode échoue si les préconditions ne sont pas remplies. Ceci n'est pas la faute à la méthode : les données ne sont simplement pas dans son domaine d'application.

Exemple 1.7. Dans la pratique il est en général prudent de tester les préconditions dans les limites du raisonnable. Mais juridiquement parlant c'est la responsabilité de l'instance appelante et non de la méthode appelée. Souvent de tels tests sont coûteux (voire impossibles). Un exemple frappant est la recherche dichotomique : elle requiert une liste *ordonnée* pour effectuer une recherche efficace, mais il serait ridicule, en début de chaque recherche, de parcourir toute la liste pour vérifier l'ordre.

1.2. Preuve de correction. Parfois on voit un « algorithme » sans spécification. C'est une mauvaise habitude qui n'est justifiée que dans les rares cas où le contexte laisse sous-entendre la spécification sans équivoque. Voici la principale raison pour laquelle la spécification doit être explicitée :

Définition 1.8. On dit qu'un algorithme est *correct* si la méthode fait ce qu'exige la spécification. Plus précisément : un algorithme est correct si pour toute donnée d'entrée vérifiant la précondition, la méthode se termine et renvoie une donnée de sortie vérifiant la postcondition.

Attention. — La preuve qu'un algorithme est correct comporte toujours deux parties. *D'abord* il faut montrer que l'algorithme s'arrête pour toute donnée d'entrée valable ; la méthode aboutit alors à un résultat. *Ensuite* il faut montrer que ce résultat vérifie la postcondition. Autrement dit :

☞ *Avant de prouver que le résultat est correct, il faut montrer qu'il y en a un.* ✍

Question 1.9. Dans l'exemple 1.1 la méthode VIII.3 est fausse par rapport à la spécification VIII.1. Montrer que la même méthode est correcte dans l'algorithme VIII.7 grâce à une précondition plus restrictive. Obtient-on un résultat correct aussi pour des données d'entrée (n,k) avec $k<0$ ou $k>n$? Pourquoi cette question ne met pas en cause la correction de l'algorithme VIII.7 ?

Algorithme VIII.7 Calcul de coefficients binomiaux

Entrée: deux entiers n et k tels que $0 \leq k \leq n$

Sortie: l'entier b vérifiant $b = \binom{n}{k}$

si $k=0$ ou $k=n$ **alors retourner** 1 **sinon retourner** $\binom{n-1}{k-1}+\binom{n-1}{k}$

Exemple 1.10. Une preuve de correction est en général difficile et demande une analyse détaillée. Le projet IV, par exemple, avait pour but de prouver puis d'implémenter l'algorithme suivant :

Algorithme VIII.8 Calcul approché de π à une précision p

Entrée: deux nombres naturels $b \geq 2$ et $p \in [\![0, 10^6]\!]$

Sortie: une suite d'entiers $t_0, t_1, \ldots, t_p$ avec $0 \leq t_i < b$ pour tout $i = 1, \ldots, p$

Garanties: la valeur $t = \sum_{k=0}^{k=p} t_k b^{-k}$ vérifie $t < \pi < t + b^{-p}$.

$\tilde{p} \leftarrow p+50$, $n \leftarrow \tilde{p}+3$, $m \leftarrow 3+\lfloor \tilde{p} \log_2 b \rfloor$
pour i **de** 0 **à** m **faire** $s_i \leftarrow 2$ **fin pour**
pour j **de** 0 **à** n **faire**
  $t_j \leftarrow s_0$, $s_0 \leftarrow 0$, $s_m \leftarrow b s_m$
  pour i **de** m **à** 1 **faire** $q \leftarrow 2i+1$, $s_{i-1} \leftarrow b s_{i-1} + i\,(s_i \operatorname{div} q)$, $s_i \leftarrow s_i \bmod q$ **fin pour**
fin pour
pour j **de** n **à** 1 **faire** $t_{i-1} \leftarrow t_{i-1} + (t_i \operatorname{div} b)$, $t_i \leftarrow (t_i \bmod b)$ **fin pour**
retourner $(t_0, t_1, \ldots, t_p)$

Vous pouvez constater qu'un tel algorithme « tombé du ciel » est peu compréhensible. Heureusement l'approche du chapitre IV fut plus soigneuse — si vous voulez, vous pouvez résumer la preuve de correction. Sans être précédé par un tel développement détaillé, l'algorithme ci-dessus paraît sans doute assez cryptique. (Il en est de même pour le programme I.21.)

☞ *Idéalement on construit un algorithme parallèlement à sa preuve de correction.* ✍

Remarque 1.11. Une fonction en C++ est une méthode : elle reçoit des données d'entrée (ses paramètres) et en déduit des données de sortie (son résultat). Dans ce but le C++, comme tout langage de programmation, exige une description très détaillée de la méthode, formulée à l'aide des instructions primitives.

Soulignons cependant que le C++ ne prévoit pas le concept de spécification. C'est bien dommage, car sans spécification la question de correction reste vide : on ne sait même pas formuler l'énoncé à démontrer.

Le mieux que l'on puisse faire en C++ est de détailler la spécification sous forme de commentaire, ou dans la documentation annexe, destiné non au compilateur mais au lecteur humain. Ainsi on peut énoncé puis démontrer la correction d'une fonction en toute rigueur. Cette justification reste bien entendu à l'extérieur du C++, et malheureusement elle est trop souvent négligée.

Remarque 1.12. Il existe des langages de programmation, comme par exemple le langage *Coq* (voir le site web `coq.inria.fr`), qui intègrent spécifications et preuves dans le code source. Pour cela il faut évidemment un formalisme assez élaboré ; la programmation n'est plus seulement le développement d'une méthode, mais comprend aussi le développement d'une preuve formelle.

Le principal avantage est le suivant : muni du code source avec preuve intégrée, *le compilateur peut vérifier la correction du programme !* Soulignons que les deux tâches sont nettement séparées : le programmeur *développe* la preuve, le compilateur la *vérifie* : il serait irréaliste de croire que le compilateur puisse inventer, à lui tout seul, une preuve de correction. À nos jours cette approche, dite *vérification automatique*, ne reste plus théorique. L'investissement supplémentaire s'amortit dans des applications sensibles qui exigent un très haut niveau de sécurité, l'idéal étant des *logiciels certifiés zéro défaut*.

1.3. Le problème de terminaison. Comme nous avons souligné plus haut, pour tout algorithme la preuve de correction commence par une preuve de terminaison. Suivant la méthode en question ceci peut être plus ou moins difficile. Évidemment, si la méthode n'utilise ni de boucles ni d'appels récursifs, alors l'algorithme se termine toujours. (Le justifier.) Ce cas simpliste est rare ; en général il faut trouver un argument plus profond, typiquement une preuve par une récurrence bien choisie.

Exemple 1.13. Pour un exemple peu trivial, regardons la *fonction d'Ackermann*. Elle est chère aux informaticiens, principalement parce qu'elle croît à une vitesse hallucinante. On pourrait prendre l'algorithme VIII.9 ci-dessous comme sa définition. Seul problème : montrer qu'il se termine.

Algorithme VIII.9 Calcul de la fonction d'Ackermann $a\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$

Entrée: deux nombres naturels n et k

Sortie: la valeur $a(n,k)$ de la fonction d'Ackermann

si $n=0$ **alors retourner** $k+1$
si $k=0$ **alors retourner** $a(n-1,1)$
retourner $a(n-1,a(n,k-1))$

Exercice 1.14. La terminaison est une propriété qu'il faut démontrer, et que l'on ne peut pas déterminer de manière expérimentale. Pour vous en convaincre, vous êtes vivement invités à faire l'expérience suivante : écrire une fonction `ackermann` en C++ et faire calculer quelques petites valeurs (disons $0\leq n\leq 4$ et $0\leq k\leq 10$). C'est un bon exercice pratique, car le résultat est peu prévisible…

Pendant que l'ordinateur rame, vous pouvez déterminer « à la main » les valeurs suivantes : par définition on a $a(0,k)=k+1$. Montrer par récurrence que $a(1,k)=k+2$, et $a(2,k)=2k+3$, puis $a(3,k)=2^{k+3}-3$. Finalement, déterminer $a(4,k)$. Même sans hâte, vous finirez avant que l'ordinateur ne calcule la valeur $a(4,2)$. Félicitations, vous calculez plus vite que votre ordinateur !

Exemple 1.15. On reprend l'exercice I.8.4 sur le « problème $3x+1$ » qui donne lieu à l'algorithme VIII.10 ci-dessous. Le lecteur en quête de célébrité peut se lancer dans la preuve de terminaison.

Algorithme VIII.10 Le problème $3x+1$

Entrée: un nombre naturel $x\geq 1$

Sortie: un nombre naturel $l\geq 0$ qui compte les itérations

$l\leftarrow 0$
tant que $x>1$ **faire**
  $l\leftarrow l+1$
  si x est pair **alors** $x\leftarrow x/2$ **sinon** $x\leftarrow 3x+1$
fin tant que
retourner l

2. La notion de complexité et le fameux « grand O »

Pour la plupart des problèmes il existe un grand nombre d'algorithmes possibles. Comment en choisir le meilleur ? Quels sont les différents degrés de complexité que l'on peut rencontrer ?

> Throughout all modern logic, the only thing that is important is whether a result can be achieved in a finite number of elementary steps or not. (…) In the case of an automaton the thing which matters is not only whether it can reach a certain result in a finite number of steps at all, but also how many such steps are needed. (John von Neumann, Hixon Symposium lecture, 1948)

Ces questions sont souvent d'une grande importance pratique, et la théorie associée constitue un domaine d'étude très poussée de l'informatique. (Pour un développement plus détaillé voir Graham-Knuth-Patashnik [**17**], chapitre 9, et l'appendice de Gathen-Gerhard [**11**] pour un tour d'horizon.)

2.1. Le coût d'un algorithme. Étant donné un algorithme et une donnée d'entrée x, on peut mesurer le *coût* $c(x)$, aussi appelé la *complexité*, de la méthode en question. Ce qui nous intéresse le plus souvent est la *complexité temporelle* : l'exécution de chaque instruction primitive nécessite un certain temps, et le temps d'exécution de l'algorithme est le temps cumulatif de toutes les instructions exécutées l'une après l'autre. (Pour simplifier on suppose parfois que les instructions primitives ont toutes le même coût, ainsi défini comme *coût unitaire*.) Très souvent le coût cumulatif est proportionnel au nombre d'itérations, et on se contente de déterminer ce dernier. L'analyse de complexité consiste ainsi à étudier la fonction $c\colon x \to c(x)$ qui associe le coût $c(x)$ à chaque entrée possible x soumise à l'algorithme.

Exemple 2.1. L'addition de deux nombres x et y à n chiffres a un coût αn, avec une constante $\alpha > 0$. Leur multiplication avec la méthode scolaire a un coût βn^2, avec une constante $\beta > 0$. (Rappeler pourquoi.) Même sans connaître les détails de l'implémentation, ceci veut dire que la méthode de la multiplication est plus coûteuse, pour n grand, que la méthode de l'addition.

Exemple 2.2. Pour trier une liste $x = (x_1, \dots, x_n)$ le tri par sélection (algorithme V.3) effectue $\frac{1}{2}n(n-1)$ itérations et son coût est $c_1(x) = \frac{1}{2}\alpha_1 n(n-1) + \beta_1(n-1)$. Le tri fusion (algorithme V.6) effectue des appels récursifs ; son coût exact $c_2(x)$ est encadré par le théorème V.3.5 : $\alpha_2 n\lfloor \log_2 n\rfloor + \beta_2(n-1) \le c_2(x) \le \alpha_2 n\lceil \log_2 n\rceil + \beta_2(n-1)$. Après avoir mesuré les constantes α_1, β_1 et α_2, β_2 sur une implémentation bien testée et optimisée, vous pouvez choisir la meilleure des deux méthodes en fonction de la taille n. Même sans connaître ces détails de l'implémentation, on voit que le tri élémentaire est compétitif seulement pour n assez petit. (Le détailler.)

2.2. Le coût moyen, dans le pire et dans le meilleur des cas. L'analyse fine que l'on vient de décrire n'est souvent pas praticable. Dans un tel cas on préfère des informations plus globales, en regroupant les données d'une même « taille ». Typiquement les données d'entrée x admettent une notion de taille naturelle, que nous noterons $|x|$: pour une liste ou un vecteur c'est sa longueur, pour un nombre entier c'est la longueur de son développement binaire, pour un polynôme c'est son degré, etc.

Pour une taille n donnée, on peut alors regarder l'ensemble $X_n = \{x \mid |x| = n\}$ de toutes les données de taille n. On définit le coût moyen par

$$\overline{c}(n) := \frac{1}{|X_n|} \sum_{x \in X_n} c(x).$$

Ici on sous-entend que l'ensemble X_n est fini et que toutes les données sont équiprobables. (D'autres distributions de probabilités sont parfois mieux adaptées, selon le contexte.) Pour être prudent, on regarde également le coût dans le pire des cas :

$$\hat{c}(n) = \max_{x \in X_n} c(x).$$

Ce point de vue est indispensable si l'on veut *garantir* une certaine performance dans *tous* les cas, non seulement en moyenne. Il est parfois instructif de regarder le coût dans le meilleur des cas :

$$\check{c}(n) = \min_{x \in X_n} c(x).$$

Exemple 2.3. Le coût du tri rapide (algorithme V.7) dépend fortement de la liste à trier et non seulement de la taille n. Le coût dans le meilleur des cas est $\check{c}(n) = \alpha n \log_2 n$, le coût dans le pire des cas est $\hat{c}(n) = \frac{\alpha}{2}n(n-1)$, mais le coût en moyenne est $\overline{c}(n) = 2\alpha n \ln n$ seulement.

2.3. Complexité asymptotique. Souvent c'est le principe de l'algorithme que l'on veut juger, et non les détails de l'implémentation. On veut donc abstraire de tous les facteurs constants ; de toute façon, ils changeront d'une implémentation à une autre, et d'une machine à une autre. Pour ne retenir que l'essentiel, on regroupe les fonctions suivant leur « ordre de grandeur » :

Définition 2.4. Soit $g\colon \mathbb{N} \to \mathbb{R}_+$ une fonction positive. On définit alors les classes suivantes :

(1) $O(g) = \{\, f\colon \mathbb{N} \to \mathbb{R}_+ \mid \exists C > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq Cg(n) \,\}$
Ce sont les fonctions qui croissent au plus aussi vite que g (finalement *majorées* par Cg).

(2) $\Omega(g) = \{\, f\colon \mathbb{N} \to \mathbb{R}_+ \mid \exists c > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \geq cg(n) \,\}$
Ce sont les fonctions qui croissent au moins aussi vite que g (finalement *minorées* par cg).

(3) $\Theta(g) = O(g) \cap \Omega(g) = \{\, f\colon \mathbb{N} \to \mathbb{R}_+ \mid \exists c, C > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : cg(n) \leq f(n) \leq Cg(n) \,\}$
Ce sont les fonctions qui croissent aussi vite que g (qui ont *même ordre* de grandeur).

(4) $o(g) = \{\, f\colon \mathbb{N} \to \mathbb{R}_+ \mid f(n)/g(n) \to 0 \text{ pour } n \to \infty \,\}$
Ce sont les fonctions qui croissent moins vite que g (qui sont *négligeables* devant g).

(5) Finalement, si $f(n)/g(n) \to 1$ pour $n \to \infty$, on dit que f et g sont équivalentes, noté $f \sim g$.

Ces notations sont traditionnellement attribuées à Landau. La tradition veut aussi que l'on écrive $f = O(g)$ à la place de $f \in O(g)$, abus de langage, hélas, auquel il faut s'habituer.

Exemple 2.5. On voit par exemple que $O(1)$ est l'ensemble des fonctions bornées, et $o(1)$ est l'ensemble des fonctions qui tendent vers 0 pour $n \to \infty$.

Exemple 2.6. Dans le projet I on a utilisé k boucles imbriquées pour parcourir tous les k-uplets $x \in \mathbb{Z}^k$ vérifiant $1 \leq x_1 \leq \cdots \leq x_k \leq n$. Le nombre $f(n)$ de tels k-uplets mesure le coût temporel de cet algorithme. On voit facilement que $f \in O(n^k)$, même $f \in \Theta(n^k)$, plus précisément on a l'équivalence $f \sim \frac{1}{k!} n^k$. Bien sûr la description la plus précise est de dire $f(n) = \binom{n+k-1}{k} = \frac{1}{k!} n(n+1)\cdots(n+k-1)$.

Exemple 2.7. Le tri fusion a un coût $f(n) = \alpha n \ln n + \beta n$ avec $\alpha > 0$. Cette fonction appartient à $O(n \ln n)$, même $\Theta(n \ln n)$, plus précisément on a l'équivalence $f \sim \alpha n \ln n$.

Exemple 2.8. La fonction $f(n) = n!$ est dans $O(n^n)$, même dans $o(n^n)$. Plus précisément, selon la formule de Stirling, f est équivalente à $n^n \cdot e^{-n} \cdot \sqrt{2\pi n}$. Chercher dans un cours d'analyse l'énoncé précis sous forme d'encadrement ; en quoi est-il plus précis qu'une simple équivalence asymptotique ?

Exercice/M 2.9. Si $f \in \Theta(g)$ a-t-on $g \in \Theta(f)$? puis $O(f) = O(g)$? et $o(f) = o(g)$?

Exercice/M 2.10. Montrer que $\ln n \in o(n^\varepsilon)$ pour tout $\varepsilon > 0$. Par conséquent $n^\alpha \ln^\beta n \in O(n^{\alpha+\varepsilon})$ pour tout $\alpha \geq 0$ et $\varepsilon > 0$. En négligeant le terme logarithmique, on dit ainsi que $n^\alpha \ln^\beta n$ est *presque* d'ordre n^α. Plus généralement on peut définir la classe $O^+(g) = \cap_{\varepsilon>0} O(g(n)n^\varepsilon)$. Ce sont les fonction de croissance *presque* d'ordre de g. Si $f_1 \in O^+(g_1)$ et $f_2 \in O^+(g_2)$, a-t-on $f_1 f_2 \in O^+(g_1 g_2)$?

2.4. Les principales classes de complexité. Dans la pratique on a souvent affaire à des fonctions de complexité dont le comportement asymptotique est un des suivants :

Complexité constante, $O(1)$***:*** Les instructions primitives en C++ sont de coût constant, par exemple tous les calculs effectués sur les variables de type `int` (dont la taille est fixée). De même pour accéder à l'élément `v[i]` d'un vecteur `v`. *Exemple :* déterminer le reste modulo 2 d'un entier en numération décimale.

Complexité logarithmique, $O(\ln n)$***:*** Un programme de complexité logarithmique devient seulement très légèrement plus lent quand n croît. Chaque fois que n est doublé, le coût n'augmente que par addition d'une constante. *Exemple :* recherche dichotomique.

Complexité linéaire, $O(n)$***:*** C'est le mieux que l'on puisse espérer pour un algorithme qui doit traiter n données une par une. Chaque fois que n est doublé, le coût double lui aussi. *Exemples :* parcourir une liste de longueur n pour trouver le maximum ou le minimum ; addition de deux entiers de longueur n en numération décimale ; déterminer le reste modulo 3 d'un nombre naturel en numération décimale. (Le détailler.)

Complexité presque linéaire, $O(n \ln n)$***:*** C'est la complexité typique pour les algorithmes de type « diviser pour régner ». Chaque fois que n est doublé, le coût est un peu plus que doublé (mais guère plus). *Exemple :* le tri fusion ou le tri rapide (dans le cas générique).

Complexité sous-quadratique, $O(n^\alpha)$ ***avec*** $\alpha < 2$***:*** La multiplication de deux entiers de longueur n en numération décimale nécessite un temps $O(n^{1,585})$ avec la méthode de Karatsuba (voir chap. II, §3). Cette complexité se situe donc entre la complexité linéaire de l'addition et la complexité quadratique de la multiplication scolaire.

Complexité quadratique, $O(n^2)$***:*** Les complexités quadratiques sont typiques pour les algorithmes traitant tous les couples parmi n données (éventuellement par deux boucles imbriquées). *Exemples :* la multiplication scolaire de deux entiers de longueur n en numération décimale (la rappeler) ; le tri élémentaire (par sélection, transposition, ou insertion).

Complexité polynomiale, $O(n^k)$ ***avec*** $k > 1$***:*** Typiquement un algorithme qui traite tous les k-uplets parmi n données est de complexité $O(n^k)$. De tels algorithmes ne sont utilisables que pour des problèmes relativement petits. *Exemple :* La recherche exhaustive des solutions de $a^4 + b^4 + c^4 = d^4$ effectuée dans le projet I est de complexité $O(n^4)$, puis on l'a réduite à $O(n^3)$. Dans le projet V elle sera réduite à $O^+(n^2)$, ce qui permettra finalement de résoudre le problème.

Complexité exponentielle, $O(e^{\alpha n})$ ***avec*** $\alpha > 0$***, voire*** $O(e^{p(n)})$ ***avec un polynôme*** p***:*** Un algorithme de complexité exponentielle est pratiquement inutilisable, sauf peut-être pour les problèmes très petits. *Exemple :* parcourir tous les 2^n sous-ensembles $S \subset X$ d'un ensemble X de taille n, ou parcourir toutes les $n!$ permutations de X.

Complexité sur-exponentielle: Il existe aussi des fonctions de croissance sur-exponentielle, comme $\exp(\exp(n))$. Des algorithmes d'une telle complexité n'ont pas d'intérêt pratique ; ils peuvent néanmoins être très importants au niveau théorique, par exemple pour montrer *l'existence* d'une solution, aussi inefficace qu'elle soit.

2.5. À la recherche du temps perdu. Dans une application sérieuse il est impensable d'utiliser un algorithme sans aucune connaissance de sa complexité. Les notions décrites ci-dessus nous aideront à fournir des indications, en particulier l'idée de classer la complexité en quelques catégories est utile pour expliquer le comportement asymptotique. Ceci est souvent un bon guide pour le choix d'algorithme.

Mais soyons clairs : la complexité asymptotique ne dit absolument rien sur le temps d'exécution dans un cas concret. Autant doit-on réfléchir longuement avant d'utiliser un algorithme cubique au lieu d'un algorithme quadratique, autant doit-on se méfier de suivre aveuglement les résultats de complexité exprimés en notation grand O. L'analyse de complexité asymptotique n'est qu'une première approximation dans un développement de plus en plus fin.

Exemple 2.11. Comparons deux méthodes, de coût n^2 heures et n^3 secondes, respectivement. Certes, elles sont d'ordre $\Theta(n^2)$ et $\Theta(n^3)$, mais regardons les *constantes cachées* : la première méthode (de complexité quadratique) n'est avantageuse que pour $n \geq 3600$, elle s'amortit donc à partir d'un temps d'exécution de 1500 années environ. Pour des applications avec $n < 3600$ on choisira la deuxième méthode (de complexité cubique). Cet exemple banal sert d'avertissement : vous avez tout intérêt à choisir la meilleure méthode dans le cas concret, disons $n = 1000$, ce qui peut ou non correspondre au comportement asymptotique. Ainsi on choisira l'algorithme en fonction de la donnée, ce que l'on appelle un *algorithme hybride*. Le bon choix des intervalles d'application (*cross-over points* en anglais) est une technique importante d'optimisation.

Exercices supplémentaires

Exercice/M 2.12. L'énoncé $O(f+g) = O(\max(f,g))$ est-il justifié ? Plus précisément : soient $f, g \colon \mathbb{N} \to \mathbb{R}_+$ deux fonctions positives et soit $h \colon \mathbb{N} \to \mathbb{R}_+$ définie par $h(n) = \max\{f(n), g(n)\}$. Comparer l'ordre de grandeur de $f+g$ et de h : a-t-on $f+g \in \Theta(h)$? puis $O(f+g) \subset O(h)$? et $O(h) \subset O(f+g)$?

Exercice/M 2.13. Dans la pratique on a souvent affaire à des boucles imbriquées ou des appels de sous-algorithmes. Dans de tels cas, la majoration suivante peut être utile. Si $f_1 \in O(g_1)$ et $f_2 \in O(g_2)$, montrer que $f_1 f_2 \in O(g_1 g_2)$. Ceci peut se résumer comme $O(f) \cdot O(g) \subset O(fg)$. Est-ce que l'énoncé $O(f) \cdot O(g) = O(fg)$ est aussi correct ?

Exercice/M 2.14. Montrer que $f \sim g$ est une relation d'équivalence. Il en est de même pour la relation $f \approx g$ définie par $f \in \Theta(g)$. Montrer que $f \sim g$ implique $f \approx g$, mais la réciproque est fausse. Montrer que la relation $f \preccurlyeq g$ définie par $f \in O(g)$ est un ordre partiel. Vérifier que $f \preccurlyeq g$ et $g \preccurlyeq f$ implique $f \approx g$. Par contre, ce n'est pas un ordre total : trouver deux fonctions croissantes $f, g \colon \mathbb{N} \to \mathbb{N}$ de sorte que ni $f \in O(g)$ ni $g \in O(f)$.

Quelques réponses et indications

[Exemple 1.1, calcul du coefficient binomial] La méthode VIII.2 se termine et elle renvoie toujours 0, ce qui n'est pas la valeur cherchée. La méthode VIII.3, se termine pour toutes les entrées n, k vérifiant $0 \leq k \leq n$ en renvoyant la valeur correcte. Pour $k < 0$ ou $k > n$, par contre, elle ne se termine pas. La méthode VIII.4 se termine toujours : déjà pour $k < 0$ ou $k > n$ elle renvoie 0, comme il faut. Pour $0 \leq k \leq n$ on peut montrer la terminaison, puis la correction, par une récurrence sur n. Quant à la méthode VIII.5, sa correction et sa performance, voir la discussion du chapitre II, §1.3.

[Exercice 2.14, ordres inéquivalents] On pourrait regarder les deux fonctions $f, g \colon \mathbb{N} \to \mathbb{N}$ définies par $f(n) = n!$ et $g(n) = (n-1)!$ si n est pair, puis $f(n) = (n-1)!$ et $g(n) = n!$ si n est impair.

Passent les jours et passent les semaines
Ni temps passé ni les amours reviennent
Vienne la nuit sonne l'heure
Les jours s'en vont je demeure
Guillaume Apollinaire, *Le pont Mirabeau*

PROJET VIII

Puissance dichotomique

Objectifs

Ce projet discute le problème de calculer efficacement la *puissance modulaire* x^n modulo m pour $x, n, m \in \mathbb{N}$. Il s'agit d'un outil omniprésent dans l'algorithmique des entiers, et quand n et m sont grands il est indispensable de comprendre les pièges et les astuces.

Sommaire

1. Le critère de Pépin

Commençons par une application typique de la puissance dichotomique :

Exemple 1.1 (nombres de Fermat)**.** Pour $k \in \mathbb{N}$ on appelle $F_k := 2^{2^k} + 1$ le $k$ième *nombre de Fermat*. Les cinq premiers termes sont $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$, $F_4 = 65537$, et Fermat constata qu'il s'agit de cinq nombres premiers. Il conjectura même que F_k serait premier pour tout $k \in \mathbb{N}$, mais ses tests empiriques n'allaient pas plus loin que $k = 4$. Le problème est évidemment que les nombres F_k croissent rapidement : comme F_{k+1} est à peu près le carré de F_k, le nombre de chiffres double en augmentant le rang. Ainsi la nature de $F_5 = 4294967297$ est déjà moins évidente. Plus ces nombres sont grands, plus on a besoin de méthodes efficaces. Comment tester efficacement la primalité de ces nombres ?

On admettra ici le critère de Pépin, que l'on établira un peu plus tard dans le projet X :

Lemme 1.2. *Pour $k \geq 1$ le nombre F_k est premier si et seulement si* $3^{\frac{F_k-1}{2}} \equiv -1 \pmod{F_k}$.

On souhaite donc calculer la puissance modulaire pour $x = 3$ et $n = 2^{2^k-1}$ et $m = 2n+1$. On va soumettre chacune des méthodes ci-dessous à la question cruciale : jusqu'à quelle valeur de k pouvons-nous effectuer le test de Pépin ? Lesquels des nombres F_k sont premiers, lesquels composés ?

On verra que parmi quatre méthodes qui viennent à l'esprit, une seule arrivera au bout… Selon vos préférences, l'exploration de fausses pistes vous semblera ou pédagogiquement réaliste ou artificiellement pénible. En tout cas elle servira, je l'espère, à vous vacciner durablement contre la programmation naïve.

2. Puissance linéaire

2.1. L'algorithme. La puissance linéaire est la première méthode qui vient à l'esprit :

Algorithme VIII.11 Puissance linéaire

Entrée: la base x et l'exposant $n \in \mathbb{N}$
Sortie: la puissance $p = x^n$

$p \leftarrow 1$	// Après initialisation on a $px^n = x^n$.
tant que $n > 0$ **faire** $p \leftarrow p * x, \quad n \leftarrow n-1$	// Le produit px^n ne change pas de valeur.
retourner p	// Ici $n = 0$ et on renvoie $p = px^n$ comme souhaité.

2.2. L'implémentation. Regardons deux implémentations de la puissance linéaire :

Exercice/P 2.1. Écrire une fonction `puissance_lineaire(x,n)` qui calcule x^n dans $\mathbb{Z}$. Pour chacun des paramètres choisir le mode de passage qui convient le mieux. Jusqu'à quelle valeur de k pouvez-vous effectuer le test de Pépin ? Quel en est le résultat ?

Exercice/P 2.2. Ajouter une fonction `puissance_lineaire(x,n,m)` qui calcule x^n modulo m. (Choisir les modes de passage.) Pourquoi a-t-on intérêt à réduire systématiquement modulo m ? Jusqu'à quelle valeur de k pouvez-vous effectuer le test de Pépin ? Quel en est le résultat ?

Exercice/P 2.3 (ici « P » non comme « programmation » mais comme « poésie »)**.** Durant les longs mois d'hiver que vous attendez une réponse de votre ordinateur, la poésie pourra vous offrir de réconfort. Récitez le poème d'Appolinaire ci-dessus puis ajoutez quelques vers. Voici les contributions de Guenaëlle De Julis et Cédric Frambourg (Licence de Mathématiques à l'Institut Fourier en 2005) :

— ∗ —

Le temps se prélasse
Les minutes se tassent
Programme infinissable
Attente détestable
Devant mon ordinateur
Toujours je demeure

— ∗ —

Stoïque devant
mon écran blafard
De rares souvenirs
montent à ma mémoire
Fixant d'un œil hagard
un tas de pixels noires

— ∗ —

Contemplons aussi le poème de Steve Planchin, Licence de Mathématiques à l'Institut Fourier en 2007 :

Les nombres restent figés sur un sombre écran
Tels le pauvre reflet sur le miroir navrant
De notre esprit embrumé empli du néant
D'un langoureux calcul bloqué entre deux temps

Question 2.4. Les expressions `puissance_lineaire(x,n)%m` et `puissance_lineaire(x,n,m)` rendent-elles le même résultat ? Laquelle est préférable et pourquoi ? Êtes-vous satisfaits du gain de performance ?

3. Puissance dichotomique

3.1. L'algorithme. Il est particulièrement facile de calculer les puissances $x^1, x^2, x^4, x^8, x^{16}, \ldots$: on pose $x_0 := x$ et calcule $x_i := x_{i-1} * x_{i-1}$ par récurrence. On obtient ainsi les valeurs $(x_0, x_1, \ldots, x_k)$ avec $x_i = x^{2^i}$ en effectuant k multiplications seulement !

Quant à une puissance x^n avec $n \geq 1$ quelconque, on peut écrire l'exposant n en base 2, c'est-à-dire $n = \sum_{i=0}^{i=k} n_i 2^i$ avec $n_i \in \{0,1\}$. Nous pouvons supposer que $n_k = 1$, ce qui rend cette écriture unique ; en particulier on a $2^k \leq n < 2^{k+1}$, donc $k = \lfloor \log_2 n \rfloor$. En supprimant les termes nuls on obtient $n = \sum_{i \in I} 2^i$, où l'ensemble I consiste des indices i avec $n_i = 1$. Ceci permet d'écrire

$$x^n = x^{\sum_{i \in I} 2^i} = \prod_{i \in I} x^{2^i} = \prod_{i \in I} x_i.$$

Ainsi on calcule x^n avec $k + |I| - 1$ multiplications seulement ! L'algorithme VIII.12 en donne une variante prête-à-programmer, qui se passe de la liste $(x_0, x_1, \ldots, x_k)$ et n'utilise que les trois variables x, n, p :

Algorithme VIII.12 Puissance dichotomique

Entrée: la base x et l'exposant $n \in \mathbb{N}$
Sortie: la puissance $p = x^n$

$p \leftarrow 1$	// Après initialisation on a $px^n = x^n$.
tant que $n > 0$ **faire**	
tant que n est pair **faire** $x \leftarrow x * x, \quad n \leftarrow n/2$	// Le produit px^n ne change pas de valeur.
$p \leftarrow p * x, \quad n \leftarrow n - 1$	// Le produit px^n ne change pas de valeur.
fin tant que	
retourner p	// Ici $n = 0$ et on renvoie $p = px^n$ comme souhaité.

Exercice/M 3.1. Montrer que l'algorithme VIII.12 se termine, puis prouver sa correction en détaillant l'invariant indiqué dans les commentaires. *Indication.* — Il sera utile d'introduire un indice i pour indiquer les valeurs p_i, x_i, n_i après la $i$ème itération, puis établir une récurrence sur $i = 0, 1, 2, \ldots$.

Exercice/M 3.2. Déterminer le nombre exact de multiplications effectuées par l'algorithme VIII.12, noté $\mu(n)$. Quel est son ordre de grandeur ? a-t-on $\mu \in \Theta(\log_2 n)$? voire $\mu \sim \alpha \log_2 n$?

Remarque 3.3. À première vue la puissance dichotomique semble miraculeuse. Après s'y être habitué, on pourrait soupçonner que cette méthode soit toujours optimale. Pourtant, pour certains exposants n on peut faire mieux, le plus petit exemple étant $n = 15$: la méthode dichotomique permet de calculer x^{15} par les 6 étapes suivantes : $(x, x^2, x^4, x^8, x^{12}, x^{14}, x^{15})$. On peut faire avec 5 multiplications seulement : $(x, x^2, x^3, x^5, x^{10}, x^{15})$. Le gain est assez faible, mais une fonction optimisée peut être justifiée si vous utilisez des puissances x^{15} très fréquemment ou si les multiplications sont très coûteuses. Pour l'usage général la méthode dichotomique est largement suffisante. Pour une discussion détaillée, consultez Knuth [**8**], §4.6.3.

3.2. L'implémentation. Pour vous convaincre de l'utilité — ou plutôt de la *nécessité* — de la puissance dichotomique, implémentez-la puis comparez sa performance à la puissance linéaire :

Exercice/P 3.4. Implémenter une fonction `puissance(x,n)` qui profite de la puissance dichotomique. Jusqu'à quelle valeur de k pouvez-vous effectuer le test de Pépin ? Quel en est le résultat ?

Exercice/P 3.5. Ajouter une fonction `puissance(x,n,m)` qui calcule x^n modulo m. Pourquoi a-t-on intérêt à réduire systématiquement modulo m durant le calcul ? Jusqu'à quelle valeur de k pouvez-vous finalement effectuer le test de Pépin ? Quel en est le résultat ?

Question 3.6. Les deux expressions `puissance(x,n)%m` et `puissance(x,n,m)` rendent-elles le même résultat ? Laquelle est préférable et pourquoi ? Êtes-vous satisfaits du gain de performance ?

Exercice/P 3.7. Pour comparaison, compiler votre programme final en utilisant la classe `Naturel` pour les grands entiers au lieu de la bibliothèque GMP. (C'est notre classe faite maison discutée au chapitre II.) Les résultats coïncident-ils ? Qu'observez-vous quant à la performance ? Comment expliquer ce phénomène ?

4. Analyse de complexité

Les exercices suivants illustrent dans quelle généralité on peut appliquer les deux algorithmes de puissance. De plus, ils analysent la complexité afin d'expliquer la performance observée.

☞ Pour simplifier vous pouvez supposer que le coût de la multiplication et de la division euclidienne de deux entiers de longueur $\leq \ell$ est (presque) linéaire en ℓ, ce qui est réaliste avec une implémentation efficace. (Revoir le chapitre II pour les opérations arithmétiques élémentaires.)

4.1. Puissance linéaire. Nos implémentations de puissance placent le calcul dans l'anneau $\mathbb{Z}$ ou $\mathbb{Z}_m$. Comme seule la multiplication est utilisée, on peut regarder la situation générale d'un ensemble M muni d'une multiplication $*\colon M \times M \to M$. Pour simplifier la notation, nous supposons que la multiplication admet un élément neutre à gauche, noté 1. On définit les puissances x^n d'un élément $x \in M$ de manière récursive par $x^0 := 1$ et $x^{n+1} := x^n * x$. Ceci correspond au parenthésage $x^n = (\cdots((x * x) * x)\cdots * x)$, dit parenthésage à gauche. (Sans hypothèse sur l'associativité il faut faire un choix ou l'autre.)

Exercice/M 4.1. L'algorithme VIII.11 avec initialisation $p \leftarrow$ (élément neutre) calcule-t-il correctement la puissance x^n dans $(M, *)$? Le résultat reste-t-il le même quand on remplace $p \leftarrow p * x$ par $p \leftarrow x * p$?

Exercice/M 4.2. Afin d'estimer le coût on comptera le nombre de multiplications : dans l'algorithme VIII.11 il en faut n. On sous-entend que le coût d'une multiplication est constant, ce qui est parfaitement justifié quand M est fini, comme $\mathbb{Z}_m$ par exemple. Plus précisément, en supposant que m est de longueur ℓ, estimer le coût du calcul de x^n dans $\mathbb{Z}_m$ en fonction de n et ℓ.

Exercice/M 4.3. Quant aux calculs dans $\mathbb{Z}$, les nombres peuvent devenir arbitrairement grands. Une telle situation nécessite une analyse plus fine : étant donné $x \in \mathbb{Z}$ ayant ℓ chiffres on s'attend à un résultat x^n ayant $n\ell$ chiffres environ. Estimer le coût de ce calcul en fonction de n et ℓ. Justifier ainsi que les calculs de `puissance_lineaire(x,n)%m` et `puissance_lineaire(x,n,m)` ne sont pas de même complexité. Ce résultat théorique correspond-il à vos expériences pratiques ?

4.2. Puissance dichotomique. La puissance linéaire effectue n multiplications pour calculer x^n. On peut faire beaucoup mieux, sous condition que la multiplication soit *associative*.

Un *monoïde* $(M,*)$ est un ensemble M muni d'une multiplication $*: M\times M\to M$ qui est associative et admet un élément neutre. Par exemple $(\mathbb{N},+)$ est un monoïde, ainsi que $(\mathbb{N},\cdot)$ et $(\mathbb{Z},\cdot)$ et plus généralement la structure multiplicative $(A,\cdot)$ d'un anneau A quelconque, commutatif ou non, par exemple l'anneau des matrices $\mathrm{Mat}(d\times d;A)$. C'est cette hypothèse d'associativité qui fait marcher notre affaire :

Proposition 4.4. *Dans un monoïde M on a $x^{m+n}=x^m*x^n$ pour tout $x\in M$ et $m,n\in\mathbb{N}$. Autrement dit, l'application $\mathbb{N}\to M$, $n\mapsto x^n$ est un homomorphisme entre les monoïdes $(\mathbb{N},+)$ et $(M,*)$. En particulier les puissances x^n d'un élément x commutent entre elles.* □

Exercice/M 4.5. Montrer soigneusement la proposition précédente. En déduire que l'algorithme VIII.12 reste correct dans un monoïde quelconque. En particulier on conclut :

Corollaire 4.6. *Dans un monoïde on peut calculer x^n avec au plus $2\lfloor\log_2 n\rfloor$ multiplications.* □

Exercice/M 4.7. Afin d'estimer le coût d'une puissance dichotomique dans $\mathbb{Z}_m$, supposons que m est de longueur ℓ. Exprimer le coût du calcul de x^n dans $\mathbb{Z}_m$ en fonction de n et ℓ.

Exercice/M 4.8. Étant donné $x\in\mathbb{Z}$ ayant ℓ chiffres, estimer le coût du calcul de x^n en fonction de n et ℓ. Justifier ainsi que les calculs de `puissance(x,n)%m` et `puissance(x,n,m)` ne sont pas de même complexité. Ce résultat théorique correspond-il à vos expériences pratiques ?

Anecdotique

En s'inspirant du paradigme « diviser pour régner » ou plutôt « régner pour diviser », les étudiants Rosencrantz et Guildenstern, amis d'école de Hamlet, proposent les deux fonctions suivantes, `puiss` et `poiss`, pour implémenter la puissance dichotomique. Sont-elles correctes ? Sont-elles efficaces ? Comment dénouer cette tragicomédie ? (Cf. Tom Stoppard, *Rosencrantz et Guildenstern ont tort*, Éditions du Seuil, Paris 1967.)

Programme VIII.1 Puissance anecdotique `hamlet.cc`

```
1  //========================================================================
2  // Though this be madness, yet there is method in't. (Hamlet, 2.2)
3  //========================================================================
4
5  // Puissance dichotragique selon Rosencrantz
6  Integer puiss( const Integer& x, const Integer& n )
7  {
8    Integer y= puiss( x, n/2 );
9    if ( n%2 == 0 ) return y*y;
10   else return y*y*x;
11 }
12
13 // Puissance dichocomique selon Guildenstern
14 Integer poiss( const Integer& x, const Integer& n )
15 {
16   if ( n < 0 )  return 0;
17   if ( n == 0 ) return 1;
18   if ( n == 1 ) return x;
19   Integer m= n/2;
20   return poiss( x, m ) * poiss( x, n-m );
21 }
```