Chapitre 1

Préliminaires

Dans ce chapitre préliminaire, nous donnons d'abord une classification des problèmes du point de vue de leur complexité algorithmique, nous discutons ensuite les mesures de complexité, puis nous donnons une borne inférieure sur la complexité du tri par comparaison.

1.1 Les algorithmes et leur coût

1.1.1 Algorithmes

Un *algorithme* est un ensemble d'opérations de calcul élémentaires, organisé selon des règles précises dans le but de résoudre un problème donné. Pour chaque donnée du problème, l'algorithme retourne une réponse après un nombre fini d'opérations. Les *opérations élémentaires* sont par exemple les opérations arithmétiques usuelles, les transferts de données, les comparaisons entre données, etc. Selon le niveau d'abstraction où l'on se place, les opérations arithmétiques et les objets sur lesquels elles portent peuvent être plus ou moins compliquées. Il peut s'agir simplement d'additionner des entiers naturels, ou de multiplier des polynômes, ou encore de calculer les valeurs propres d'une matrice. Pour un système de calcul formel, il s'agit là d'opérations élémentaires, parce qu'elles ont été programmées et sont disponibles dans des bibliothèques, plus ou moins transparentes à l'utilisateur; dans un langage comme Pascal, ces opérations ne sont pas disponibles.

Il apparaît utile de ne considérer comme véritablement élémentaires que les opérations dont le *temps de calcul* est constant, c'est-à-dire ne dépend pas de la *taille* des opérandes. Par exemple, l'addition d'entiers de taille bornée *a priori* (les `integer` en Pascal) est une opération élémentaire; l'addition d'entiers de taille quelconque ne l'est pas. De même, le test d'appartenance d'un élément à un ensemble n'est pas une opération élémentaire en ce sens, parce que son

temps d'exécution dépend de la taille de l'ensemble, et ceci même si dans certains langages de programmation, il existe des instructions de base qui permettent de réaliser cette opération.

L'*organisation* des calculs, dans un algorithme, est souvent appelée sa *structure de contrôle*. Cette structure détermine l'ordre dans lequel il convient de tester des conditions, de répéter des opérations, de recommencer tout ou partie des calculs sur un sous-ensemble de données, etc. Là aussi, des conventions existent sur la façon de mesurer le coût des opérations; la richesse des structures de contrôle dépend fortement des langages de programmation. Le langage Pascal est peut-être l'un des plus pauvres dans ce domaine parmi les langages modernes. En général, le coût des structures de contrôle peut être négligé, parce qu'il est pris en compte, asymptotiquement, par les opérations élémentaires. En fait, le surcoût provenant par exemple d'une programmation récursive n'est vraiment sensible que lorsqu'elle entraîne des recopies massives de données.

1.1.2 Problèmes intraitables

Contrairement à une idée largement répandue, tout problème ne peut être résolu par un algorithme. Ce n'est pas une question de taille des données, mais une impossibilité fondamentale. On doit la preuve de ce fait aux logiciens des années 30 et 40. Pour en donner une démonstration rigoureuse, il convient évidemment de formuler très précisément la notion d'algorithme. Il apparaît que les possibilités des ordinateurs et des langages de programmation sont parfaitement couvertes par la définition mathématique de ce qui est résoluble algorithmiquement.

On est donc en présence d'une première dichotomie, entre problèmes *insolubles* algorithmiquement et les autres. Parmi les problèmes qui sont algorithmiquement solubles, on peut encore distinguer une hiérarchie en fonction de la *complexité* des algorithmes, complexité mesurée en temps de calcul, c'est-à-dire en nombre d'opérations élémentaires, et exprimée en fonction de la taille du problème. La *taille* elle-même est un paramètre qui mesure le nombre de caractères nécessaires pour décrire une donnée du problème. Pour une matrice carrée à coefficients bornés par exemple, son *ordre* est un bon paramètre de la taille; pour un polynôme, le *degré* peut être une indication de la taille, sauf si l'on sait par exemple que le polynôme n'a que très peu de coefficients non nuls, auquel cas le nombre de coefficients non nuls est un meilleur indicateur. Le temps de calcul d'un algorithme croît en général en fonction de la taille des données, et la vitesse de la croissance est une mesure de la complexité du problème. Une croissance *exponentielle* ou plus rend un problème intraitable pour des données de grande taille. Même une croissance polynomiale, disons comme une puissance cinquième de la taille des données, rend la résolution pratiquement impossible pour des données d'une certaine taille.

Evidemment, il existe de très nombreux problèmes pour lesquels tout algorithme a une croissance au moins exponentielle. Dans cette catégorie, on trouve les algo-

rithmes dont le résultat lui-même est constitué d'un nombre exponentiel ou plus de données. Par exemple, la génération de toutes les permutations de n éléments prend un temps plus qu'exponentiel, simplement parce qu'il y a $n!$ permutations. Il existe toute une classe de problèmes, qui sont des problèmes d'optimisation combinatoire, où l'on cherche une solution optimale parmi un nombre exponentiel de solutions réalisables. Une bonne stratégie, si elle existe, permet de trouver une solution optimale sans énumérer l'ensemble des solutions réalisables; on peut alors résoudre le problème en temps disons polynomial. Bien entendu, on ne connaît pas toujours une telle stratégie; mais on ne sait pas à l'heure actuelle si, pour tout problème de ce type, il existe un algorithme polynomial. La conjecture la plus problable est qu'un très grand nombre de problèmes de ce type ne puissent pas être résolus par un algorithme polynomial. On aurait ainsi une deuxième division des problèmes, entre ceux qui sont solubles, mais intraitables, et les autres, pour lesquels il existe des algorithmes efficaces. L'un des objectifs de l'algorithmique est l'étude de ces problèmes, pour lesquels on cherche les algorithmes les plus efficaces, tant du point de vue du temps de calcul que des besoins en place.

1.1.3 Sur la présentation d'algorithmes

Pour présenter les algorithmes de manière formelle, nous utilisons dans ce livre un langage proche de Pascal. Toutefois, en vue de ne pas alourdir inutilement la lecture, nous n'en suivons pas strictement la syntaxe; de plus, nous introduisons quelques ellipses, et quelques variantes sur les structures de contrôle. La plupart parlent d'elles-mêmes.

Ainsi, pour économiser l'écriture de parenthèses (début,fin), nous utilisons les mots finsi, finpour, et fintantque comme délimiteurs de portée, lorsqu'un tel délimiteur est utile. Nous écrivons donc :

si *test* alors *suite d'instructions* sinon *suite d'instructions* finsi
tantque *test* faire *suite d'instructions* fintantque
pour $\cdots$ faire *suite d'instructions* finpour

D'autres variantes concernent la structure de contrôle. Ainsi, retourner, et plus rarement exit provoque (comme en C) la sortie de la procédure ou fonction courante, et le retour vers la procédure appelante.

Enfin, nous utilisons largement la version séquentielle des opérateurs booléens et et ou (comme en Modula), baptisés etalors et oualors. Dans l'évaluation d'une expression de la forme a etalors b, on évalue d'abord a, et on n'évalue b que si a est vrai. Le même mécanisme vaut *mutatis mutandis* pour oualors. On économise ainsi beaucoup de programmation confuse.

1.2 Mesures du coût

Considérons un problème donné, et un algorithme pour le résoudre. Sur une donnée x de taille n, l'algorithme requiert un certain temps, mesuré en nombre d'opérations élémentaires, soit $c(x)$. Le coût en temps varie évidemment avec la taille de la donnée, mais peut aussi varier sur les différentes données de même taille n. Par exemple, considérons l'algorithme de tri qui, partant d'une suite $(a_1, \ldots, a_n)$ de nombres réels distincts à trier en ordre croissant, cherche la première descente, c'est-à-dire le plus petit entier i tel que $a_i > a_{i+1}$, échange ces deux éléments, et recommence sur la suite obtenue. Si l'on compte le nombre d'*inversions* ainsi réalisées, il varie de 0 pour une suite triée à $n(n-1)/2$ pour une suite décroissante. Notre but est d'évaluer le coût d'un algorithme, selon certains critères, et en fonction de la taille n des données.

1.2.1 Coût dans le cas le plus défavorable

Le coût $C(n)$ d'un algorithme dans le cas *le plus défavorable*ou dans le cas *le pire* («worst-case» en anglais) est par définition le maximum des coûts, sur toutes les données de taille n :

$$C(n) = \max_{|x|=n} c(x)$$

(on note $|x|$ la taille de x.) Dans l'algorithme ci-dessus, ce coût est $n(n-1)/2$. Cette mesure du coût est réaliste parce qu'elle prend en compte toutes les possibilités.

1.2.2 Coût moyen

Dans des situations où l'on pense que le cas le plus défavorable ne se présente que rarement, on est plutôt intéressé par le coût moyen de l'algorithme. Une formulation correcte de ce coût moyen suppose que l'on connaisse une *distribution de probabilités* sur les données de taille n. Si $p(x)$ est la probabilité de la donnée x, le *coût moyen* $\gamma(n)$ d'un algorithme sur les données de taille n est par définition

$$\gamma(n) = \sum_{|x|=n} p(x)c(x)$$

Le plus souvent, on suppose que la distribution est uniforme, c'est-à-dire que $p(x) = 1/T(n)$, où $T(n)$ est le nombre de données de taille n. Alors, l'expression du coût moyen prend la forme

$$\gamma(n) = \frac{1}{T(n)} \sum_{|x|=n} c(x) \tag{2.1}$$

Continuons notre exemple du tri par transposition. Le nombre de données de taille n, c'est-à-dire de suites de n nombres réels distincts, est infini. Or, seul l'ordre relatif des éléments d'un suite intervient dans l'algorithme. Une donnée de taille n peut donc être assimilée à une permutation de l'ensemble $\{1, \ldots, n\}$. On a alors $T(n) = n!$. Le coût $c(x)$ d'une permutation x est le nombre de transpositions d'éléments adjacents nécessaires pour transformer x en la permutation identique; ce nombre est le *nombre d'inversions* de x, qui est par définition le nombre $b_1 + \cdots + b_n$, où b_j est le nombre d'entiers $1 \leq i < j$ tels que $x_i > x_j$. On peut montrer que le nombre moyen d'inversions est $n(n-1)/4$, de sorte que le coût moyen de l'algorithme, pour la distribution uniforme, est $n(n-1)/4$. Comme le montre cet exemple, l'évaluation du coût moyen est en général bien plus compliquée que l'évaluation du coût dans le cas le plus défavorable.

Considérons un autre exemple, à savoir la génération de toutes les parties d'un ensemble E à n éléments, disons de $\{1, \ldots, n\}$. Chaque partie X est représentée par son vecteur caractéristique x, avec

$$x[i] = \begin{cases} 1 & \text{si } i \in X \\ 0 & \text{sinon} \end{cases}$$

Concrètement, on utilise un type `suite` pour représenter ces vecteurs. Le calcul du premier sous-ensemble, la partie vide de E, revient à initialiser la suite x à $(0, \ldots, 0)$. Les parties suivantes s'obtiennent en procédant de droite à gauche dans le vecteur courant x. On remplace tous les 1 par 0 jusqu'à rencontrer un 0. Celui-ci est remplacé par 1 (c'est très exactement l'algorithme d'incrémentation, en binaire). Voici une réalisation :

```
PROCEDURE Suivante (VAR x:suite; n:integer; VAR d:boolean);
   VAR
      i: integer;
   BEGIN
      i := n;
      WHILE (i > 0) AND (x[i] = 1) DO BEGIN
         x[i] := 0; i := i - 1
      END;
      d := i = 0; IF NOT d THEN x[i] := 1
   END;
```

Dans la boucle `WHILE`, l'opérateur `AND` est séquentiel. La variable booléenne `d` repère si la suite donnée en argument représente la «dernière» partie de l'ensemble E, à savoir E lui-même.

Analysons le coût d'un appel de la procédure `Suivante`, sur le tableau x, mesuré par le nombre de comparaisons $x[i] = 1$. Le coût est n lorsque les $x[i]$ valent 1 pour $1 \leq i \leq n$, c'est-à-dire lorsque l'on a atteint la dernière partie. Le coût est aussi n lorsque $x[1] = 0$ et les autres éléments de x valent 1. Quel est le coût moyen de la procédure, sur tous les appels? Pour l'évaluer, nous reprenons l'équation (2.1)

et l'écrivons sous la forme

$$\gamma(n) = \frac{1}{T(n)} \sum_{i=1}^{n} i c_i$$

où c_i est le nombre de données x pour lequelles le coût est i, c'est-à-dire telles que $c(x) = i$. Pour $1 \leq i < n$, le nombre de comparaisons est i, si x se termine par un élément 0 suivi par $i-1$ éléments égaux à 1. Le nombre de suites de cette forme est 2^{n-i}. Enfin, on a $c_n = 2$, de sorte que

$$\gamma(n) = \frac{1}{2^n} \left(2n + \sum_{i=1}^{n-1} i 2^{n-i} \right)$$

Le nombre entre parenthèses est égal à $2^{n+1} - 2$, et on a

$$\gamma(n) = \frac{2^{n+1} - 2}{2^n}$$

donc le coût moyen du calcul de la partie suivante est approximativement 2.

Comme ces exemples l'indiquent, l'estimation du coût moyen d'un algorithme est en général plus délicate que l'estimation du coût dans le cas le plus défavorable, et nécessite une bonne dose de connaissances combinatoires sur les objets traités.

1.2.3 Coût amorti

Lorsqu'une *suite* d'opérations est effectuée sur une structure, le coût total est égal à la somme des coûts des opérations individuelles. Connaissant des estimations du coût (dans le cas le plus défavorable) des opérations individuelles, on obtient une estimation du coût total en sommant ces majorations du coût des opérations individuelles. Or très souvent, cette majoration du coût total qui, à chaque étape, procède par une évaluation pessimiste, est trop grossière. On observe en effet que le cas le plus défavorable «ne se répète pas». Plus précisément, dans de nombreux algorithmes, le cas le plus défavorable provient d'un déséquilibre exceptionnel qui ne peut pas se produire plusieurs fois de suite, car le traitement de ce cas défavorable «rétablit» l'équilibre. Il y a alors un phénomène de compensation entre opérations consécutives, d'amortissement des coûts, que nous explicitons dans cette section. Commençons par un exemple.

Exemple. Considérons une *pile* (voir chapitre 3), et définissons une *opération* comme étant une suite (éventuellement vide) de dépilements suivie d'un empilement (ce genre d'opérations se rencontre par exemple en analyse syntaxique). Le *coût* d'une opération o est $c(o) = 1 + d$, où d est le nombre de dépilements précédant l'empilement.

Partant de la pile vide P_0, on effectue une suite $o_1, \ldots, o_n$ d'opérations :

$$P_0 \xrightarrow{o_1} P_1 \longrightarrow \ldots \xrightarrow{o_n} P_n$$

où P_k est la pile après la k-ième opération, et l'on veut estimer le coût total

$$c_n = c(o_1) + \cdots + c(o_n)$$

Une méthode simple consiste à majorer le coût de chaque opération o_k par k. En effet, dans le cas le plus défavorable, l'opération o_k vide complètement la pile P_{k-1} avant de procéder à l'empilement, et P_{k-1} contient au plus $k-1$ éléments. On a donc $c(o_k) \leq k$, d'où $c(n) = O(n^2)$.

Mais, comme déjà dit plus haut, le cas le plus défavorable (ici : la pile P_{k-1} contient $k-1$ éléments) ne se produit pas deux fois de suite. En d'autres termes, notre analyse n'a pas pris en compte une contrainte globale qui, dans le cas d'une pile, exprime que l'on ne peut pas dépiler des éléments avant de les avoir empilés. Plus précisément, notons d_k le nombre de dépilements de la k-ième opération, de sorte que

$$c(o_k) = 1 + d_k$$

Alors $c_n = n + d$ avec $d = d_1 + \cdots + d_n$. Or n est le nombre total d'empilements et d est le nombre total de dépilements. Comme on ne peut dépiler plus que l'on a empilé, on a $d \leq n$ et donc $c_n \leq 2n$. Cet argument s'explicite par ce que l'on nomme la *technique du potentiel* : à chaque pile P_k, on associe un nombre h_k (son «potentiel») qui est toujours positif ou nul; le coût de l'opération o_k s'exprime à l'aide d'une variation du potentiel. Dans notre cas, prenons pour h_k la hauteur de la pile P_k. On a $h_0 = 0$,

$$h_k = h_{k-1} - d_k + 1 \qquad (k \geq 1)$$

et bien sûr $h_k \geq 0$ pour $k = 1, \ldots, n$. Comme $d_k = h_{k-1} - h_k + 1$, le coût de l'opération o_k est

$$c(o_k) = 2 + h_{k-1} - h_k$$

Il en résulte bien entendu que $c_n = c(o_1) + \cdots + c(o_n) = 2n + h_0 - h_n = 2n - h_n \leq 2n$. On appelle *coût amorti* de l'opération o (relativement au potentiel h) la valeur

$$a(o) = c(o) + h'' - h'$$

où h' est le potentiel de la pile avant l'opération, et h'' est le potentiel de la pile après l'opération o. Dans notre cas, le coût amorti est égal à 2. On obtient

$$a(o_1) + \cdots + a(o_n) = c(o_1) + \cdots + c(o_n) + h_n - h_0$$

et comme $h_n \geq 0$, $h_0 = 0$, on a

$$c(o_1) + \cdots + c(o_n) \leq a(o_1) + \cdots + a(o_n)$$

donc le coût amorti de la suite des opérations est une majoration du coût total.

On peut voir ce procédé de comptage d'une autre manière, plus ludique. Supposons que l'ordinateur qui doit exécuter la suite d'opérations travaille avec des

jetons, comme une vulgaire machine à sous («coin-operated computer», disent les Américains). Chaque fois que l'on insère un jeton, l'ordinateur est disposé à travailler pendant une durée fixe, puis il s'arrête et attend le jeton suivant. Si une opération se termine avant épuisement du temps acheté, ce temps de calcul est disponible pour l'opération suivante. Le coût total d'une suite d'opérations est alors proportionnel au nombre de jetons qu'il faut introduire, et l'avoir est égal à la valeur du potentiel. Le coût amorti correspond au nombre de jetons à introduire par opération (ici 2) pour pouvoir effectuer les opérations.

Plus formellement, on considère une famille de «structures» sujettes à des opérations qui les transforment. De plus, on suppose défini un *potentiel*, c'est-à-dire une fonction h qui à chaque structure associe un nombre réel positif ou nul. En particulier, on demande que le potentiel de la structure vide soit nul.

Par définition, le *coût amorti* d'une opération o qui, appliquée à la structure S, donne la structure $S' = o(S)$, relativement au potentiel h, est

$$a(o) = c(o) + h(S') - h(S)$$

Le coût amorti d'une suite d'opérations $o_1, \ldots, o_n$ est la somme des coûts amortis des opérations individuelles.

Proposition 2.1. *Soit h un potentiel, et soit $o_1, \ldots, o_n$ une suite d'opérations appliquée à une structure de potentiel nul. La somme des coûts de la suite d'opérations est majorée par son coût amorti relativement à h :*

$$c(o_1) + \cdots + c(o_n) \leq a(o_1) + \cdots + a(o_n)$$

Preuve. Soit S_0 la structure de départ initialement vide, et soit S_k la structure obtenue après la k-ième opération :

$$S_0 \xrightarrow{o_1} S_1 \xrightarrow{o_2} \ldots \xrightarrow{o_n} S_n$$

On a $a(o_k) = c(o_k) + h(S_k) - h(S_{k-1})$. Comme les valeurs du potentiel s'éliminent deux à deux, on obtient

$$a(o_1) + \cdots + a(o_n) = c(o_1) + \cdots c(o_n) + h(S_n) - h(S_0)$$

Or $h(S_0) = 0$ et $h(S_n) \geq 0$, ce qui établit l'inégalité. ∎

Dans la pratique, on essaie de trouver, en utilisant son intuition, un potentiel qui fournit une majoration aisée du coût amorti des opérations, ce qui donne ensuite une majoration du coût total.

Exemple. Revenons sur l'exemple de la génération des parties d'un ensemble à n éléments. Notons o_k le k-ième appel de la procédure `Suivante`. Le coût total c_N du calcul des N premiers sous-ensembles est $c_N = c(o_1) + \cdots c(o_N)$. Définissons un potentiel h sur le tableau x par : $h(x) =$ le nombre d'éléments égaux à 1 dans

x. Le coût de l'opération o_k qui transforme le $k-1$-ième tableau $x^{(k-1)}$ en le k-ième tableau $x^{(k)}$ est alors

$$c(o_k) = 2 + h(x^{(k-1)}) - h(x^{(k)})$$

et le coût amorti est

$$a(o_k) = c(o_k) + h(x^{(k)}) - h(x^{(k-1)}) = 2$$

La somme des coûts amortis est $2N$, et en vertu de la proposition précédente, ceci majore le coût total. Ce résultat est d'une autre nature que l'évaluation en moyenne donnée plus haut, dans la mesure où il ne fait appel à aucune hypothèse probabiliste.

1.3 Une borne inférieure

Après avoir trouvé et analysé un algorithme qui résout un problème donné, il est naturel de se demander si cet algorithme est le meilleur possible. Il existe plusieurs façons d'améliorer, parfois substantiellement, un algorithme, ou un programme, en organisant mieux les données, en économisant des calculs d'indices, en regroupant certaines opérations. Ces optimisations peuvent prendre en compte les caractéristiques d'un langage de programmation spécifique, voire les particularités d'un compilateur. La question de l'optimalité d'un algorithme, vue indépendamment d'une implémentation particulière, se pose différemment : on se demande s'il existe un algorithme qui nécessite moins d'opérations élémentaires. Pour cela, on estime le nombre d'opérations requises par *tout* algorithme qui résout le problème donné. Là encore, la formulation du problème et de la réponse se fait en fonction de la taille des données. On cherche le nombre d'opérations nécessaires pour résoudre le problème dans le cas le plus défavorable pour les données de taille n.

Pour apporter une réponse à cette question, il faut d'abord définir précisément la classe d'algorithmes et les opérations élémentaires. Nous décrivons ci-dessous le modèle des *arbres de décision*qui permet de prouver que tout algorithme de tri qui opère par comparaisons nécessite au moins $\lceil n \ln n \rceil - n$ comparaisons pour trier des suites de n éléments. Cette borne inférieure vaut pour le cas le plus défavorable : pour tout algorithme, et pour tout n, il existe une suite dont le tri exige au moins $\lceil n \ln n \rceil - n$ comparaisons.

Dans un algorithme de tri par comparaisons, on utilise uniquement des comparaisons pour obtenir des informations sur la suite $(a_1, \ldots, a_n)$ de données. En d'autres termes, étant donnés a_i et a_j, on effectue l'un des test $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$ pour déterminer leur position relative. Aucune autre information n'est accessible sur ces éléments. Pour prouver le résultat annoncé, il suffit de se restreindre aux suites dont les éléments sont distincts; nous pouvons donc nous contenter du test $a_i \leq a_j$.

Le comportement d'un algorithme de tri, sur des suites de longueur n, peut être représenté par un arbre binaire (voir chapitre 4), appelé arbre de décision. Dans un arbre de décision, chaque nœud est étiqueté par une comparaison, notée $a_i : a_j$, pour des entiers $1 \leq i, j \leq n$, et chaque feuille est étiquetée par une permutation $(\sigma(1), \ldots, \sigma(n))$ (voir figure 3.1).

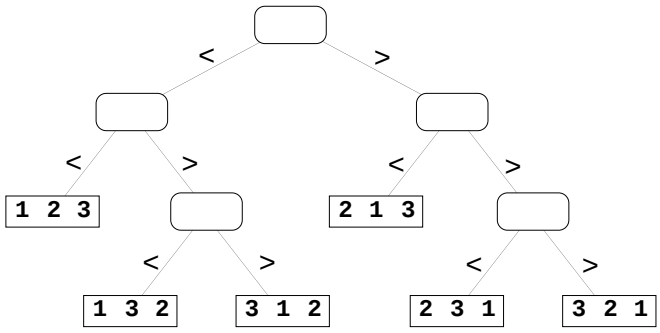

Figure 3.1: *L'arbre de décision d'un algorithme de tri.*

L'exécution de l'algorithme de tri sur une suite $(a_1, \ldots, a_n)$ correspond à un chemin dans l'arbre, menant de la racine à une feuille. A chaque nœud, une comparaison $a_i < a_j$ est faite, et le résultat de la comparaison indique si les comparaisons suivantes se font dans le sous-arbre gauche ou le sous-arbre droit. Lorsque une feuille est atteinte, l'algorithme de tri a établi que l'ordre est $a_{\sigma(1)} < a_{\sigma(2)} < \cdots < a_{\sigma(n)}$. Chacune des $n!$ permutations doit apparaître comme étiquette d'une feuille, pour que l'algorithme puisse proprement séparer tous les cas de figure.

La longueur du chemin de la racine à une feuille donne le nombre de comparaisons effectuées, et dans le cas le plus défavorable, le nombre de comparaisons nécessaires est donc égal à la hauteur de l'arbre. Une minoration de la hauteur d'un arbre de décision fournit donc une borne inférieure sur le nombre de comparaisons.

Lemme 3.1. *Soit A un arbre binaire. Si A a $n!$ feuilles, alors la hauteur de A est au moins $\lceil n \ln n \rceil - n$.*

Preuve. Soit h la hauteur de A. Alors $n! \leq 2^h$, donc $\ln n! \leq h \ln 2 \leq h$. Or $\ln n! \geq n \ln n - n$ (voir chapitre 2), d'où le résultat. ∎

Corollaire 3.2. *Tout algorithme de tri par comparaisons nécessite, dans le cas le plus défavorable, au moins $\lceil n \ln n \rceil - n$ comparaisons pour trier des suites de longueur n.* ∎

Il est important de noter que ce résultat ne porte que sur une famille bien précise d'algorithmes de tri, à savoir les tris par comparaisons. Il existe d'autres techniques de tris qui peuvent s'appliquer lorsque l'on a des informations sur la nature

des clés. Le tri par champs par exemple (« bucket sort » en anglais) est un tri en temps linéaire qui s'emploie lorsque les clés sont des chaînes de caractères.

La borne inférieure sur les algorithmes de tri est « générique » en ce sens que, pour de nombreux problèmes apparemment complètement différents, on obtient également une borne inférieure en $n \ln n$, en se ramenant au problème du tri. Il en est ainsi par exemple pour le calcul de l'enveloppe convexe d'un ensemble de points dans le plan (chapitre 11, section 2). La démarche est la suivante : on considère un algorithme A pour le problème donné, et on montre que cet algorithme peut aussi être utilisé, parfois après quelques transformations, comme algorithme de tri par comparaisons. En vue du corollaire ci-dessus, le nombre d'opérations de l'algorithme A est minoré. Comme l'argument vaut pour n'importe quel algorithme A, on obtient la borne inférieure recherchée.

Chapitre 2

Evaluations

La première section de ce chapitre contient la définition des notations dites de Landau, à savoir O, Ω, et θ, ainsi que des exemples de manipulation. Dans la deuxième section, nous abordons la résolution de divers types de récurrences. En effet, l'analyse d'un algorithme conduit le plus souvent à des équations de récurrence. Dans certains cas, on peut les résoudre complètement et donner une forme close pour les fonctions qu'elles définissent; plus nombreux sont les cas où l'on doit se contenter d'une évaluation asymptotique; on ne donne alors qu'un ordre de grandeur.

Introduction

L'efficacité en temps d'un algorithme se mesure en fonction d'un paramètre, généralement la « taille » du problème. Ce n'est évidemment pas le temps physique, exprimé en millisecondes ou en heures qui importe. Ce temps dépend trop de toutes sortes de contingences matérielles, comme l'équipement dont on dispose, le langage de programmation et la version du compilateur employés, etc. On mesure le temps requis par un algorithme en comptant le nombre d'opérations élémentaires effectuées, où une opération est élémentaire lorsqu'elle prend un temps constant. Si l'on veut être très précis, on doit aller plus loin, et classer les opérations élémentaires selon leur nature, comme des incréments de compteurs, des opérations arithmétiques, des opérations logiques, etc. Il est apparu qu'une telle précision n'apportait pas d'information substantielle, et il est aujourd'hui communément admis que même le décompte exact du nombre d'opérations n'est utile que dans une phase ultime de la réalisation d'un algorithme par un programme : dans les autres situations, on se contente d'un ordre de grandeur.

2.1 Notations de Landau

On évalue l'efficacité d'un algorithme en donnant l'ordre de grandeur du nombre d'opérations qu'il effectue lorsque la taille du problème qu'il résout augmente. On parle ainsi d'algorithme linéaire, quadratique, logarithmique, etc. Les *notations de Landau* sont un moyen commode d'exprimer cet ordre de grandeur. Trois situations sont décrites par ces notations. La plus fréquente, la notation O introduite ci-dessous, donne une *majoration* de l'ordre de grandeur; la notation Ω en donne une *minoration*, et la notation θ deux bornes sur l'ordre de grandeur. La force des notations de Landau réside dans leur concision. En contre-partie, leur emploi demande de la vigilance et un certain entraînement.

2.1.1 Notation O

On considère une fonction $g : \mathbb{R} \to \mathbb{R}$. Etant donné un point $x_0 \in \mathbb{R} \cup \{-\infty, +\infty\}$, on désigne par $O(g)$ l'ensemble des fonctions f pour lesquelles il existe un voisinage$V$ de x_0 et une constante $k > 0$ tels que

$$|f(x)| \leq k\,|g(x)| \qquad (x \in V)$$

Dans le cas des fonctions définies sur $\mathbb{R}$, un *voisinage* d'un point x_0 est une partie de $\mathbb{R}$ contenant un intervalle ouvert contenant le point x_0. On peut donc, dans la définition ci-dessus, remplacer le terme «voisinage» par «intervalle ouvert». Si la fonction g ne s'annule pas, il revient au même de dire que le rapport

$$\left|\frac{f(x)}{g(x)}\right|$$

est *borné* pour $x \in V$.

Exemple. Au voisinage de 0, on a

$$x^2 \in O(x), \quad \ln(1+x) \in O(x)$$

Par commodité, on écrit x^2 à la place de «la fonction qui à x associe x^2»; nous utiliserons par la suite cette abus d'écriture. Pour l'évaluation de la complexité des algorithmes, nous nous intéressons à la comparaison de fonctions au voisinage de $+\infty$. Ce cas est couvert, dans la définition précédente, si l'on prend $x_0 = +\infty$; un voisinage est alors un ensemble contenant un intervalle (ouvert) de la forme $]a, +\infty[$. Par conséquent, on a $f \in O(g)$ au voisinage de $+\infty$ s'il existe deux nombres $k, a > 0$ tels que

$$|f(x)| \leq k\,|g(x)| \text{ pour tout } x > a$$

Exemple. Au voisinage de l'infini, on a

$$x \in O(x^2), \quad \frac{\ln x}{x} \in O(1), \quad x + 1 \in O(x)$$

Exemple. Pour tout polynôme $P(x) = a_0x^k + a_1x^{k-1} + \cdots + a_k$, on a $P(x) \in O(x^k)$ au voisinage de l'infini. En effet, pour $x \geq 1$,

$$|P(x)| \leq |a_0|\, x^k + |a_1|\, x^{k-1} + \cdots + |a_k| \leq (|a_0| + \cdots + |a_k|)\, x^k$$

Souvent, les fonctions à comparer sont elles-mêmes à valeurs positives. Dans ce cas, on peut omettre les valeurs absolues.

Lors de l'étude de fonctions mesurant les performances d'algorithmes, l'argument est en général entier (*taille* du problème) et on se place au voisinage de l'infini, ce qui signifie que l'on considère les performances de l'algorithme pour des tailles importantes du problème; dans ce cas, les mêmes considérations restent valables lorsque le domaine des fonctions est restreint à l'ensemble $\mathbb{N}$ des entiers naturels.

Une des difficultés dans la familiarisation avec ces concepts provient de la *convention de notation* (justement «de Landau») qui veut que l'on écrive

$$f = O(g), \qquad \text{ou encore} \qquad f(x) = O(g(x)) \qquad \text{au lieu de} \qquad f \in O(g)$$

De manière analogue, on écrit

$$O(f) = O(g) \qquad \text{lorsque} \qquad O(f) \subset O(g)$$

Notons tout de suite que la relation $f = O(g)$ n'implique pas nécessairement que $g = O(f)$.

Exemple. Les formules

$$x^2 + 5x = O(x^2) = O(x^3)$$

dénotent en fait les relations

$$x^2 + 5x \in O(x^2) \subset O(x^3)$$

La notation de Landau permet l'écriture usuelle de certaines opérations arithmétiques; en définissant

$$f + O(g) = \{f + h \mid h \in O(g)\}$$
$$fO(g) = \{fh \mid h \in O(g)\}$$

on a par exemple $h = f + O(g)$ si et seulement si $h - f \in O(g)$. Les formules suivantes sont faciles à vérifier et fort utiles (ici c est une constante; sans la notation de Landau, il faudrait utiliser un signe d'appartenance au lieu de l'égalité dans la première formule, et un signe d'inclusion dans les autres).

$$\begin{aligned} f &= O(f) \\ O(-f) &= O(f) \\ cO(f) &= O(f) \\ O(f) + O(f) &= O(f) \\ O(f)O(g) &= O(fg) \\ fO(g) &= O(fg) \end{aligned}$$

Si f et g sont à valeurs positives, alors

$$O(f) + O(g) = O(f + g)$$

Vérifions par exemple la dernière formule. Si $h \in O(f)+O(g)$, alors il existe deux constantes $k, k' > 0$ et $a > 0$ telles que $|h(x)| \leq kf(x) + k'g(x)$ pour $x \geq a$. Soit K le plus grand des nombres k et k'. Alors on a $|h(x)| \leq K(f(x) + g(x))$ parce que f et g sont à valeurs positives. La formule est fausse si l'on prend $g = -f$ par exemple.

Exemple. On a $2^{n+1} = O(2^n)$ mais en revanche $3^n \notin O(2^n)$ et de même $(n + 1)! \notin O(n!)$ (puisque $(n + 1)!/n!$ tend vers $+\infty$ avec n). Observons aussi que $f(n) = O(n)$ implique $f(n)^2 = O(n^2)$, mais n'implique pas que $2^{f(n)} = O(2^n)$.

2.1.2 Notations Ω et θ

Deux notations semblables à la notation O sont couramment employées pour la description de la complexité des algorithmes.On désigne par $\Omega(g)$ l'ensemble des fonctions f pour lesquelles il existe deux nombres $k, a > 0$ tels que

$$|f(x)| \geq k|g(x)| \text{ pour tout } x > a$$

En d'autres termes, on a

$$f \in \Omega(g) \iff g \in O(f)$$

Exemple. Le nombre de comparaisons de tout algorithme de tri de suites de longueur n est $\Omega(n \ln n)$, d'après le chapitre précédent.

Enfin, on désigne par $\theta(g)$ l'ensemble des fonctions f pour lesquelles il existe des nombres k_1, k_2, $a > 0$ tels que

$$k_1 |g(x)| \leq |f(x)| \leq k_2 |g(x)|$$

pour tout $x > a$. En d'autres termes, on a

$$\theta(g) = O(g) \cap \Omega(g)$$

Ceci signifie donc que f et g croissent «de façon comparable». Plus précisément, si g ne s'annule pas, alors pour tout $x > a$

$$k_1 \leq \frac{|f(x)|}{|g(x)|} \leq k_2$$

Exemple. Pour tout $a, b > 1$, on a

$$\log_a n = \theta(\log_b n)$$

puisque $\log_a n = \log_a b \log_b n$.

Cet exemple ne doit pas faire croire que si $f = \theta(g)$, alors le quotient $|f(x)/g(x)|$ tend vers une limite non nulle, c'est-à-dire $|f| \sim c|g|$ pour une constante $c > 0$ (rappelons que $f \sim g$ signifie que le quotient $f(x)/g(x)$ tend vers la limite 1). En revanche, si $|f| \sim c|g|$, alors $f = \theta(g)$.

Exemple. Soit $f(x) = x(2 + \sin x)$. On a $x \leq f(x) \leq 3x$ pour tout $x > 0$, donc $f(x) = \theta(x)$. En revanche, le quotient $f(x)/x$ ne tend pas vers une limite lorsque x tend vers $+\infty$.

2.1.3 Exemples

Suivant l'usage dans l'analyse d'algorithmes, on entendra désormais que toutes les estimations de fonctions se font au voisinage de l'infini.

Proposition 1.1. *Pour tout $k \geq 1$, on a*

$$\sum_{i=1}^{n} i^k = \theta(n^{k+1}) \tag{1.1}$$
$$\log n! = \theta(n \log n)$$
$$\sum_{i=1}^{n} \frac{1}{i} = \theta(\log n)$$

La preuve de ces formules est l'occasion d'introduire une technique éprouvée d'évaluation de sommes utilisant des intégrales. Les inégalités requises sont énoncées dans le lemme élémentaire suivant.

Lemme 1.2. *Soient $a \leq b$ deux entiers, et soit $f : [a, b] \to \mathbb{R}$ une fonction croissante et continue. Alors*

$$\sum_{i=a}^{b-1} f(i) \leq \int_a^b f(x)\, dx \leq \sum_{i=a+1}^{b} f(i) \qquad \blacksquare$$

Bien entendu, des inégalités analogues s'obtiennent pour des fonctions décroissantes.

Preuve de la proposition. Commençons par (1.1). Il est clair que $\sum_{i=1}^{n} i^k = O(n^{k+1})$: il suffit pour cela de majorer chaque terme de la somme par n^k. Il est clair aussi que $\sum_{i=1}^{n} i^k = \Omega(n^k)$: il suffit pour cela de négliger tous les termes sauf le dernier. La difficulté est donc de remplacer la minoration grossière $\Omega(n^k)$ par la minoration plus précise $\Omega(n^{k+1})$. Pour ce faire, nous utilisons le lemme

précédent. La fonction $x \mapsto x^k$ est croissante et continue sur l'intervalle $[0, n]$. On a donc

$$\sum_{i=1}^{n} i^k \geq \int_0^n x^k \, dx = \frac{n^{k+1}}{k+1}$$

ce qui prouve que la somme est dans $\Omega(n^{k+1})$.

Prouvons la deuxième formule. On a clairement $\log n! = O(n \log n)$. D'autre part, la fonction $x \mapsto \ln x$ est croissante et continue sur l'intervalle $[1, n]$. Par le lemme, on a donc

$$\ln n! = \sum_{i=2}^{n} \ln i \geq \int_1^n \ln x \, dx = [x \ln x - x]_1^n$$

donc

$$\ln n! \geq n \ln n - n + 1 \geq \frac{n \ln n}{2}$$

dès que $\ln n \geq 2$. Ceci montre que $\ln n! \in \Omega(n \ln n)$. Comme $\ln n = \theta(\log n)$, la même formule vaut pour log.

Considérons enfin la troisième formule. La fonction $x \mapsto 1/x$ est décroissante et continue sur l'intervalle $[1, n+1]$. On a donc par le lemme

$$\sum_{i=1}^{n} \frac{1}{i} \geq \int_1^{n+1} \frac{1}{x} \, dx = \ln(n+1)$$

De même,

$$\sum_{i=2}^{n} \frac{1}{i} \leq \int_1^n \frac{1}{x} \, dx = \ln n$$

d'où

$$\ln(n+1) \leq \sum_{i=1}^{n} \frac{1}{i} \leq 1 + \ln n. \qquad \blacksquare$$

Remarquons que, pour les trois formules de la proposition, on dispose en fait d'expressions plus précises dont l'établissement dépasse le cadre de ce livre. On a

$$\sum_{i=1}^{n} i^k = \frac{B_{n+1}(n+1) - B_{n+1}}{k+1}$$

où $B_n(x)$ est le n-ième *polynôme de Bernoulli* et $B_n = B_n(0)$ est le n-ième *nombre de Bernoulli*. La *formule de Stirling*

$$n! \sim n^n e^{-n} \sqrt{2\pi n} \left(1 + \frac{1}{12n} + \frac{23}{288n^2} + \cdots\right)$$

permet de retrouver l'évaluation asymptotique de $\log n!$. Enfin, les *nombres harmoniques* $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ satisfont

$$H_n = \ln n + \gamma + \frac{1}{2n} + \cdots$$

où γ est la constante d'Euler.

Exemple. Considérons un exemple plus substantiel. Il a pour but de développer une estimation de la croissance asymptotique d'une fonction définie implicitement, en employant une méthode d'affinages successifs appelée en anglais « bootstrapping ». Soit la fonction f donnée par l'équation

$$f(t)e^{f(t)} = t$$

On veut l'étudier lorsque t tend vers $+\infty$. On constate d'abord que, pour $t > 0$, on a $f(t) > 0$. On peut donc passer au logarithme, et on obtient

$$\ln f(t) + f(t) = \ln t$$

et l'expression

$$f(t) = \ln t - \ln f(t) \tag{1.2}$$

L'équation de définition de $f(t)$ montre que pour $t \geq e$, on a $f(t) \geq 1$. Par conséquent, $\ln f(t) \geq 0$, et l'équation (1.2) montre que $f(t) \leq \ln t$ et donc

$$f(t) = O(\ln t)$$

Ceci « amorce la pompe ». Il résulte ensuite de cette relation que

$$\ln f(t) = \ln O(\ln t) = O(\ln \ln t)$$

et par conséquent on peut affiner l'équation (1.2) en

$$f(t) = \ln t + O(\ln \ln t)$$

On reporte ceci à nouveau dans l'équation (1.2), et on obtient

$$\begin{aligned} f(t) &= \ln t - \ln(\ln t + O(\ln \ln t)) \\ &= \ln t - \ln(\ln t\,(1 + O(\ln \ln t)/\ln t)) \\ &= \ln t - \ln \ln t - \ln\,(1 + O(\ln \ln t)/\ln t) \\ &= \ln t - \ln \ln t + O(\ln \ln t/\ln t) \end{aligned}$$

Cette dernière expression donne une description déjà fort précise du comportement asymptotique de la fonction f.

Exemple. On veut donner une expression asymptotique, lorsque n tend vers $+\infty$, de

$$\sqrt[n]{n} \quad \text{ et de } \quad n\left(\sqrt[n]{n} - 1\right)$$

Pour cela, on se rappelle qu'au voisinage de 0, la fonction e^x admet le développement limité

$$e^x = 1 + x + O(x^2)$$

Comme $\sqrt[n]{n} = e^{\ln n/n}$, et que $\ln n/n$ est voisin de 0 lorsque n est voisin de ∞, on a

$$\sqrt[n]{n} = e^{\ln n/n} = 1 + \ln n/n + O\left(\ln n/n\right)^2$$

Cela montre également que

$$n\left(\sqrt[n]{n} - 1\right) = \ln n + O\left(\ln^2 n/n\right).$$

2.2 Récurrences

L'analyse des performances d'un algorithme donne en général des équations implicites, où le temps de calcul, pour une taille des données, est exprimé en fonction du temps de calcul pour des données plus petites. Résoudre ces équations n'est pas toujours possible, et l'on se contente de donner des équivalents qui décrivent, de manière satisfaisante, le comportement des algorithmes. Dans cette section, on passe en revue certaines techniques permettant d'obtenir des expressions explicites exactes ou approchées pour des fonctions données par des relations de récurrence.

2.2.1 Récurrences linéaires à coefficients constants

Nous considérons ici les relations de récurrence linéaires à coefficients constants. Des relations plus générales sont décrites dans la section suivante.

Relations de récurrence homogènes

Une *suite récurrente linéaire* est une suite $(u_n)_{n\geq 0}$ de nombres (réels) qui vérifie une relation du type suivant :

$$u_{n+h} = a_{h-1}u_{n+h-1} + \cdots + a_0 u_n, \qquad n = 0, 1, 2, \ldots \quad a_0 \neq 0 \tag{2.1}$$

où h est un entier strictement positif et $a_{h-1}, \ldots, a_0$ sont des nombres fixés. Une telle suite est entièrement déterminée par ses h premières valeurs $u_0, \ldots, u_{h-1}$. On dit que la suite est d'ordre h. Le *polynôme associé* ou *caractéristique*de l'équation (2.1) est par définition

$$G(X) = X^h - a_{h-1}X^{h-1} - \cdots - a_1 X - a_0$$

On note ses racines $\omega_1, \ldots, \omega_r$, la multiplicité de ω_i étant notée n_i pour $i = 1, \ldots, r$.

Considérons la série génératrice (formelle) de la suite (u_n) :

$$u(X) = \sum_{n=0}^{\infty} u_n X^n \tag{2.2}$$

et soit $B(X) = X^h G(1/X)$ le polynôme réciproque de $G(X)$, c'est-à-dire

$$B(X) = 1 - a_{h-1}X - \cdots - a_0 X^h$$

Posons $A(X) = B(X)u(X)$. En vertu de la formule (2.1), il vient

$$A(X) = \sum_{j=0}^{h-1} \left(u_j - \sum_{i=1}^{j} a_{h-1-i} u_{j-i} \right) X^j$$

Ainsi $A(X)$ est un polynôme de degré au plus $h-1$, et la formule

$$u(X) = \frac{A(X)}{B(X)} \tag{2.3}$$

montre que la série formelle est une série rationnelle. En décomposant la fraction rationnelle (2.3) en éléments simples, on obtient une expression du type

$$u(X) = \sum_{i=1}^{r} \sum_{j=1}^{n_i} \frac{\beta_{i,j}}{(1-\omega_i X)^j}$$

(rappelons que n_i est la multiplicité de la racine ω_i). Comme

$$\frac{1}{(1-\omega X)^j} = \sum_{n=0}^{\infty} \binom{n+j-1}{j-1} \omega^n X^n$$

le terme général de la suite (u_n) est donné par une expression de la forme

$$u_n = \sum_{i=1}^{r} P_i(n)\omega_i^n \tag{2.4}$$

où

$$P_i(X) = \sum_{j=1}^{n_i} \beta_{i,j} \binom{X+j-1}{j-1} \tag{2.5}$$

est un polynôme de degré au plus $n_i - 1$ pour $i = 1, \ldots, r$. La formule (2.4) est très utile; c'est elle qui donne la forme close de u_n; c'est aussi elle qui permet d'obtenir une estimation asymptotique lorsque la racine de plus grand module est unique et réelle positive par exemple. Une expression (2.4) est appelée un *polynôme exponentiel*.

Réciproquement, tout polynôme $P_i(X)$ de degré n_i s'écrit sous la forme (2.5) parce que les polynômes

$$\binom{X+j}{j} = \frac{1}{j!}(X+1)\cdots(X+j) \qquad j \geq 0$$

forment une base de l'espace des polynômes. Si le terme général de la suite (u_n) est donné par (2.4), on obtient donc une expression (2.3) puis une relation (2.1) pour la suite. Nous avons donc montré qu'une suite récurrente a trois représentations équivalentes : la relation de récurrence, l'expression comme fraction rationnelle de sa série génératrice, et l'expression close de ses termes sous forme de polynôme exponentiel.

Exemple. Considérons la suite

$$\begin{aligned} &t_n = 3t_{n-1} + 4t_{n-2} \qquad n \geq 2 \\ &t_0 = 0, \quad t_1 = 1 \end{aligned}$$

Elle est d'ordre 2, son polynôme caractéristique est $X^2 - 3X - 4$, et ses racines sont -1 et 4. On obtient

$$u(X) = \frac{X}{1 - 3X - 4X^2} = \frac{1}{5}\left(\frac{1}{1-4X} - \frac{1}{1+X}\right)$$

d'où l'expression

$$t_n = \frac{4^n - (-1)^n}{5} \qquad n \geq 0$$

Exemple. La suite

$$\begin{array}{l} t_{n+3} = 5t_{n+2} - 8t_{n+1} + 4t_n \qquad n \geq 0 \\ t_0 = 0, \quad t_1 = 1, \quad t_2 = 2 \end{array}$$

est d'ordre 3, et son polynôme caractéristique $X^3 - 5X^2 + 8X - 4$ a la racine simple 1 et la racine double 2. On obtient

$$u(X) = \frac{X - 3X^2}{1 - 5X + 8X^2 - 4X^3} = -\frac{2}{1-X} + \frac{5/2}{1-2X} - \frac{1/2}{(1-2X)^2}$$

En développant, ceci donne

$$t_n = 2^{n+1} - n2^{n-1} - 2 \qquad n \geq 0$$

Notons que la même relation de récurrence, avec les conditions initiales $t_0 = 1$, $t_1 = 3$, et $t_2 = 7$, conduit à l'expression $t_n = 2^{n+1} - 1$. Ceci est dû au fait que la fraction

$$u(X) = \frac{1 - 2X}{1 - 5X + 8X^2 - 4X^3}$$

se simplifie, et que la suite (t_n) est, dans ce cas, aussi solution d'une relation d'ordre 2.

Exemple. L'exemple le plus populaire et sans doute aussi le plus ancien (il date de 1202) est la suite (F_n) de Fibonacci définie par

$$\begin{array}{l} F_{n+2} = F_{n+1} + F_n \qquad n \geq 0 \\ F_0 = 0, \quad F_1 = 1 \end{array}$$

Le polynôme caractéristique $X^2 - X - 1$ a les deux racines

$$\phi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad \bar{\phi} = \frac{1-\sqrt{5}}{2}$$

et sa série génératrice

$$u(X) = \frac{X}{1 - X - X^2} = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi X} - \frac{1}{1-\bar{\phi}X}\right)$$

donne l'expression

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \bar{\phi}^n) \qquad n \geq 0$$

Asymptotiquement, on a

$$F_n \sim \frac{1}{\sqrt{5}}\phi^n$$

Comme exemple d'application, considérons l'*algorithme d'Euclide*de calcul du pgcd de deux entiers positifs, non tous les deux nuls, par divisions successives :

```
fonction PGCD(x, y);
   si y = 0 alors PGCD:= x
   sinon PGCD:=PGCD(y, x mod y)
   finsi.
```

Notons $n(x, y)$ le nombre de divisions avec reste effectuées par l'algorithme. Alors

$$n(x,y) \;=\; \text{si } y = 0 \text{ alors } 0 \text{ sinon } 1 + n(y, x \bmod y)$$

Nous allons prouver que pour $x > y \geq 0$,

$$n(x,y) = k \;\Rightarrow\; x \geq F_{k+2} \tag{2.6}$$

Pour $k = 0$, on a $x \geq 1 = F_2$, et pour $k = 1$, on a $x \geq 2 = F_3$. Supposons donc $k \geq 2$, et considérons les divisions euclidiennes

$$x = qy + z,\ 0 \leq z < y$$
$$y = q'z + u, 0 \leq u < z$$

On a $n(y, z) = k - 1$, donc $y \geq F_{k+1}$, et de même $z \geq F_k$, par conséquent $x \geq F_{k+1} + F_k = F_{k+2}$. Ceci prouve (2.6). Comme $F_n \geq \frac{1}{\sqrt{5}}(\phi^n - 1)$, on a $\sqrt{5}x + 1 \geq \phi^{k+2}$, donc pour $x > y \geq 0$,

$$n(x,y) \leq \log_\phi(5x + 1) - 2$$

Ce résultat est le théorème de Lamé. En particulier, $n(x, y) = O(\log x)$.

Relations de récurrence à second membre

Nous considérons maintenant des suites $(u_n)_{n\geq 0}$ de nombres qui vérifient une relation du type suivant :

$$u_{n+h} = a_{h-1}u_{n+h-1} + \cdots + a_0 u_n + v_n, \qquad n = 0, 1, 2, \ldots \quad a_0 \neq 0 \tag{2.7}$$

où h est un entier positif et $a_{h-1}, \ldots, a_0$ sont des nombres fixés, et où v_n est une fonction de n. On peut réécrire (2.7) en

$$u_{n+h} - a_{h-1}u_{n+h-1} - \cdots - a_0 u_n = v_n, \qquad n \geq 0, \quad a_0 \neq 0$$

ce qui explique le terme de relation de récurrence avec second membre. Comme dans le cas traité plus haut, une telle suite est entièrement déterminée par ses h premières valeurs $u_0, \ldots, u_{h-1}$. On dit que la suite est d'ordre h. Considérons les séries génératrices (formelles) des suites (u_n) et (v_n) :

$$u(X) = \sum_{n=0}^{\infty} u_n X^n \qquad v(X) = \sum_{n=0}^{\infty} v_n X^n$$

Les calculs faits pour les récurrences homogènes conduisent cette fois-ci à l'expression

$$u(X) = \frac{A(X) + X^h v(X)}{B(X)} \tag{2.8}$$

où comme plus haut $B(X) = 1 - a_{h-1}X - \cdots - a_0 X^h$ et

$$A(X) = \sum_{j=0}^{h-1} \left(u_j - \sum_{i=1}^{j} a_{h-1-i} u_{j-i} \right) X^j$$

Calculer les polynômes $A(X)$ et $B(X)$ revient à résoudre l'équation sans second membre, qui est ensuite «corrigée» par la série $v(X)$.

Nous nous intéressons au cas particulier où la suite (v_n) vérifie elle-même une relation de récurrence. Dans ce cas, la série génératrice v est elle-même une fraction rationnelle :

$$v(X) = \frac{P(X)}{Q(X)}$$

pour des polynômes P et Q. L'équation (2.8) prend alors la forme

$$u(X) = \frac{A(X)Q(X) + X^h P(X)}{B(X)Q(X)} \tag{2.9}$$

ce qui montre que la série u est encore rationnelle.

Exemple. Considérons la suite (t_n) définie par

$$\begin{aligned} t_n &= 2t_{n-1} + 1 \qquad n \geq 1 \\ t_0 &= 0 \end{aligned}$$

La suite associée à l'équation homogène $t_n = 2t_{n-1}$, avec $t_0 = 0$ est nulle. Donc $A(X) = 0$, $B(X) = 1 - 2X$. Par ailleurs, $v(X) = 1/(1-X)$, et

$$u(X) = \frac{X}{(1-2X)(1-X)}$$

En particulier, la suite (t_n) vérifie la relation de récurrence $t_{n+2} = 3t_{n+1} - 2t_n$, et bien entendu $t_n = 2^n - 1$.

Une autre façon de résoudre (2.7) lorsque la suite v_n est une suite récurrente linéaire, est de substituer cette relation de récurrence. Si

$$v_{n+k} = b_1 v_{n+k-1} + \cdots + b_k v_n \qquad n \geq 0$$

alors pour $n \geq 0$

$$u_{n+h+k} = \sum_{i=1}^{h} a_i u_{n+h+k-i} + \sum_{j=1}^{k} b_j \left(u_{n+h+j} - \sum_{i=1}^{h} a_i u_{n+h-i+k-j} \right)$$

ce qui, après réarrangement, donne une relation de récurrence explicite pour (u_n).

Exemple. Considérons la suite

$$\begin{aligned} &t_{n+1} - 2t_n = n + 2^n \qquad n \geq 1 \\ &t_0 = 0 \end{aligned}$$

La suite $v_n = n + 2^n$ vérifie la relation $v_{n+3} = 4v_{n+2} - 5v_{n+1} + 2v_n$, d'où

$$t_{n+4} - 2t_{n+3} = 4(t_{n+3} - 2t_{n+2}) - 5(t_{n+2} - 2t_{n+1}) + 2(t_{n+1} - 2t_n)$$

et finalement

$$t_{n+4} = 6t_{n+3} - 7t_{n+2} - 2t_n$$

Exemple. La fonction récursive de calcul des nombres de Fibonacci est souvent utilisée pour tester l'efficacité de compilateurs ou d'interprètes. Elle s'écrit :

```
fonction FIBONACCI(n);
  si n = 0 ou n = 1 alors
    FIBONACCI := n
  sinon
    FIBONACCI := FIBONACCI(n − 1)+FIBONACCI(n − 2)
  finsi.
```

Notons r_n le nombre d'appels de FIBONACCI pour le calcul du nombre F_n. On a

$$\begin{aligned} &r_n = 1 + r_{n-1} + r_{n-2} \qquad n \geq 2 \\ &r_0 = r_1 = 1 \end{aligned}$$

On trouve sans peine que

$$r_n = 2F_{n+1} - 1 = \theta(\phi^n)$$

donc que le temps d'exécution est proportionnel à la valeur calculée. Ceci n'est pas étonnant puisque la procédure forme le résultat en additionnant des 0 et des 1.

Exemple. Parfois, on peut se ramener à des récurrences linéaires par un changement de variables, ou un changement de valeurs (remplacer une fonction par son logarithme par exemple). Considérons la fonction t définie sur les entiers qui sont des puissances de 2 par

$$\begin{aligned} t(1) &= 6 \\ t(n) &= n(t(n/2))^2 \qquad n > 1 \text{ puissance de } 2 \end{aligned}$$

On peut vérifier directement que $t(n) = 2^{3n-2}3^n/n$; pour trouver cette formule, on fait d'abord un changement de variable en posant $s(k) = t(2^k)$ pour $k \geq 0$, d'où la relation

$$\begin{aligned} s(0) &= 6 \\ s(k) &= 2^k(s(k-1))^2 \end{aligned}$$

Ensuite, on pose $u_k = \log_2 s(k)$, et on obtient la relation de récurrence linéaire

$$\begin{aligned} u_0 &= \log 6 \\ u_k &= k + 2u_{k-1} \end{aligned}$$

Cette dernière se résout par les méthodes indiquées ci-dessus, et donne

$$u_k = 2^k u_0 + 2^{k+1} - k - 2$$

d'où en reportant :

$$s(k) = 2^{u_k} = 6^{2^k} 2^{2^{k+1}} 2^{-k-2}$$

et le résultat, et posant $n = 2^k$.

2.2.2 Récurrences diverses

Voici quelques techniques permettant de résoudre certaines relations de récurrences qui ne sont pas de la forme précédente.

Méthode des facteurs sommants

La méthode des facteurs sommants permet de traiter des suites (u_n) définies par une relation de récurrence linéaire d'ordre 1, de la forme

$$a_n u_n = b_n u_{n-1} + c_n \qquad n \geq 1$$

où a_n, b_n, c_n sont des fonctions de n. On pose alors

$$f_n = \frac{a_1 \cdots a_{n-1}}{b_1 \cdots b_n} \qquad n \geq 1$$

avec $f_1 = 1/b_1$. On a $f_n a_n = f_{n+1} b_{n+1}$. En posant

$$y_n = f_n a_n u_n$$

la relation se réécrit

$$y_n = y_{n-1} + f_n c_n$$

Cette relation se résout immédiatement en

$$y_n = y_0 + \sum_{i=1}^{n} f_i c_i$$

et on obtient, pour u_n, l'expression

$$u_n = \frac{u_0 + \sum_{i=1}^{n} f_i c_i}{a_n f_n}$$

Exemple. Considérons la suite (u_n) définie par

$$\begin{aligned} u_0 &= a \\ u_n &= bn^k + nu_{n-1} \qquad n > 1 \end{aligned}$$

où a et b sont des réels, et $k \in \mathbb{N}$. Les formules précédentes s'appliquent avec $c_n = bn^k$, $a_n = 1$, $b_n = n$, d'où $f_n = 1/n!$ et

$$u_n = n! \left(a + b \sum_{i=1}^{n} \frac{i^k}{i!} \right)$$

Maintenant, la série de terme général $i^k/i!$ converge, et le nombre

$$B_k = \frac{1}{e} \sum_{i=1}^{\infty} \frac{i^k}{i!}$$

est le k-ième *nombre de Bell*. On a donc

$$u_n \sim n!(a + beB_k)$$

Changement de valeurs

Dans certaines situations, on peut remplacer la fonction à évaluer par son logarithme, et obtenir une expression plus simple pour la relation de récurrence.

Exemple. Le nombre de fonctions booléennes à n variables booléennes est donné par

$$\begin{aligned} t(1) &= 4 \\ t(n) &= (t(n-1))^2 \qquad n > 1 \end{aligned}$$

Posons $u_n = \log t(n)$. On a alors

$$\begin{aligned} u_1 &= 2 \\ u_n &= 2u_{n-1} \qquad n > 1 \end{aligned}$$

d'où $u_n = 2^n$ et $t(n) = 2^{2^n}$.

Voici un exemple où le même procédé est appliqué, mais où les calculs sont beaucoup plus compliqués.

Exemple. Le nombre d'arbres binaires de hauteur strictement inférieure à n est donné par

$$\begin{aligned} x_0 &= 1 \\ x_{n+1} &= x_n^2 + 1 \qquad n \geq 0 \end{aligned}$$

L'arbre vide est de hauteur -1. Nous allons montrer qu'il existe une constante

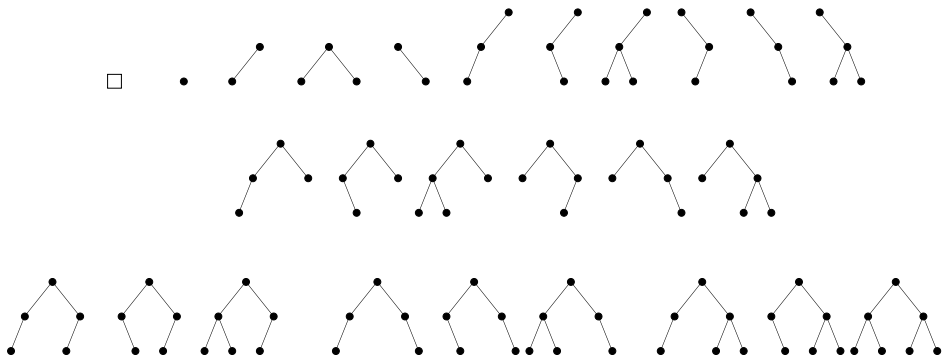

Figure 2.1: *Les arbres binaires de hauteur* $-1 \leq h \leq 2$.

k telle que $x_n = \left\lfloor k^{2^n} \right\rfloor$. Posons pour cela $y_n = \ln x_n$. On a

$$y_{n+1} = 2y_n + \alpha_n$$

avec

$$\alpha_n = \ln\left(1 + \frac{1}{x_n^2}\right)$$

On en déduit l'expression

$$y_n = 2^n \left(\frac{\alpha_0}{2} + \cdots + \frac{\alpha_{n-1}}{2^n}\right)$$

Posons

$$Y_n = 2^n \sum_{i=0}^{\infty} \frac{\alpha_i}{2^{i+1}}, \qquad r_n = Y_n - y_n$$

Comme la suite (x_n) est croissante, et donc la suite (α_n) est décroissante, on a l'estimation suivante pour r_n :

$$r_n = Y_n - y_n = 2^n \sum_{i=n}^{\infty} \frac{\alpha_i}{2^{i+1}} \leq \alpha_n \sum_{i=0}^{\infty} \frac{1}{2^{i+1}} \leq \alpha_n$$

Si l'on définit le nombre k par

$$k = \exp\left(\sum_{i=0}^{\infty} \frac{\alpha_i}{2^{i+1}}\right)$$

on a $Y_n = 2^n \ln k$, donc $x_n = e^{y_n} = e^{Y_n - r_n} = k^{2^n} e^{-r_n}$ et il ne reste plus qu'à utiliser la majoration pour r_n :

$$x_n \leq k^{2^n} = x_n e^{r_n} \leq x_n e^{\alpha_n} = x_n \left(1 + \frac{1}{x_n^2}\right) = x_n + \frac{1}{x_n} < x_n + 1$$

pour $n > 1$, d'où

$$x_n = \left\lfloor k^{2^n} \right\rfloor$$

Evidemment, l'expression de k n'est pas facile à évaluer...

2.2.3 Récurrences de partitions

Les relations de récurrence que nous considérons maintenant sont de la forme :

$$\begin{aligned} &t(n_0) = d \\ &t(n) = at(n/b) + f(n) \qquad n > n_0 \end{aligned}$$

On les rencontre naturellement lorsque l'on cherche un algorithme récursif pour résoudre un problème de taille n par la méthode des sous-problèmes (« diviser pour régner ») : on remplace le problème par a sous-problèmes, chacun de taille n/b. Si $t(n)$ est le coût de l'algorithme pour la taille n, il se compose donc de $at(n/b)$ plus le temps $f(n)$ pour recomposer les solutions des problèmes partiels en une solution du problème total. Dans de nombreux cas, les équations ci-dessus sont en fait des inéquations. Le coût $\tau(n)$ vérifie

$$\begin{aligned} &\tau(n_0) = d \\ &\tau(n) \leq a\tau(n/b) + f(n) \qquad n > n_0 \end{aligned}$$

Il en résulte que $\tau(n) \leq t(n)$ pour tout n, et donc que la fonction t constitue une majoration du coût de l'algorithme. Avant de continuer, un exemple.

Exemple. Soit à calculer le produit de deux entiers u, v vérifiant $0 \leq u, v < 2^{2n}$ pour un entier n. On peut donc écrire u et v en binaire sur $2n$ bits. Posons

$$u = 2^n U_1 + U_0, \quad v = 2^n V_1 + V_0$$

avec $0 \leq U_0, U_1, V_0, V_1 < 2^n$. Ainsi U_1 est formé des n bits de plus fort poids de u, etc. On a

$$uv = (2^{2n} + 2^n)U_1V_1 + 2^n(U_1 - U_0)(V_0 - V_1) + (2^n + 1)U_0V_0$$

Cette formule montre que l'on peut décomposer le problème de la multiplication de deux nombres à $2n$ bits en 3 multiplications de nombres à n bits, à savoir U_1V_1, $(U_1 - U_0)(V_0 - V_1)$, U_0V_0, et quelques additions et décalages. On a donc ramené un problème de taille $2n$ en 3 sous-problèmes de taille n ($a = 3, b = 2$ dans les formules ci-dessus). Soit $t(n)$ le temps requis pour multiplier deux nombres à n bits par cette méthode. Alors

$$t(2n) = 3t(n) + cn, \quad t(1) = 1$$

pour une constante c, puisque les décalages et l'addition prennent un temps linéaire en n. Il résulte de la proposition ci-dessous que $t(n) = \theta(n^{\log_2 3})$.

Un grand nombre de situations rencontrées dans l'analyse d'algorithmes sont couvertes par l'énoncé que voici :

Proposition 2.1. *Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction croissante au sens large à partir d'un certain rang, telle qu'il existe des entiers $n_0 \geq 1$, $b \geq 2$ et des réels $k \geq 0$, $a > 0$, $c > 0$ et $d > 0$ pour lesquels*

$$\begin{array}{ll} t(n_0) = d & \\ t(n) \ = at(n/b) + cn^k & \quad n > n_0,\ n/n_0 \ \textit{puissance de}\ b \end{array} \tag{2.10}$$

Alors on a

$$t(n) = \begin{cases} \theta(n^k) & \textit{si } a < b^k \\ \theta(n^k \log_b n) & \textit{si } a = b^k \\ \theta(n^{\log_b a}) & \textit{si } a > b^k \end{cases} \tag{2.11}$$

Preuve. Soit d'abord $n = n_0 b^p$ pour un entier p. Alors

$$t(n) = da^p \sum_{i=0}^{p-1} ca^i \left(n/b^i\right)^k = \delta(n) + cn^k \gamma(n)$$

avec

$$\delta(n) = da^p = \theta(n^{\log_b a}) \ \text{ et } \ \gamma(n) = \sum_{i=0}^{p-1} \left(a/b^k\right)^i$$

parce que $a^p = n^{\log_b a}/a^{\log_b n_0}$. Or

$$\begin{array}{ll} \gamma(n) \sim \dfrac{1}{1 - a/b^k} & \text{si } \ a/b^k < 1 \\ \gamma(n) = p \sim \log_b n & \text{si } \ a/b^k = 1 \\ \gamma(n) \sim \alpha \dfrac{a^p}{b^{kp}} & \text{si } \ a/b^k > 1 \end{array}$$

avec $\alpha = 1/(a/b^k - 1)$. Les deux premières formules s'en déduisent. Si $a > b^k$, on a

$$cn^k\gamma(n) \sim \alpha c n_0^k a^p = \theta(n^{\log_b a})$$

D'où le résultat.

Soit maintenant n assez grand, et soit p tel que $n_0 b^p < n \leq n_0 b^{p+1}$. Posons $n_- = n_0 b^p$ et $n_+ = n_0 b^{p+1} = b n_-$. Comme $t(n_-) \leq t(n) \leq t(n_+)$, et que $f(bn) = \theta(f(n))$ pour chacune des trois fonctions f intervenant au second membre de (2.11), on a $t(n) = \theta(f(n))$. ∎

Voici quelques applications de ces formules dans le cas le plus fréquent, où $b = 2$ et $n_0 = 1$:

Exemple. Soit t une fonction croissante qui vérifie $t(1) = 1$. Si pour $n > 1$, puissance de 2, on a

$$\begin{array}{lll}
t(n) = t(n/2) + c & \text{alors} & t(n) = \theta(\log n) \qquad (a = 1, k = 0) \\
t(n) = 2t(n/2) + cn & \text{alors} & t(n) = \theta(n \log n) \qquad (a = 2, k = 1) \\
t(n) = 2t(n/2) + cn^2 & \text{alors} & t(n) = \theta(n^2) \qquad (a = 2, k = 2) \\
t(n) = 4t(n/2) + cn^2 & \text{alors} & t(n) = \theta(n^2 \log n) \qquad (a = 4, k = 2) \\
t(n) = 7t(n/2) + cn^2 & \text{alors} & t(n) = \theta(n^{\log 7}) \qquad (a = 7, k = 2)
\end{array}$$

Remarque. Parfois, l'analyse d'un algorithme ne permet pas d'obtenir une information aussi précise que celle de l'équation (2.10). Si, dans cette équation, on remplace le terme cn^k par $O(n^k)$, la même conclusion reste vraie, en remplaçant dans (2.11) les θ par des O.

Exemple. Soit t une fonction croissante au sens large et vérifiant

$$\begin{array}{ll}
t(1) & = 1 \\
t(n) & = 2t(n/2) + \log n \qquad n > 1 \text{ et puissance de } 2
\end{array}$$

Comme $\log n = O(n)$, on a $t(n) = O(n \log n)$. En fait, on peut montrer que $t(n) = \theta(n)$ (voir ci-dessous).

Proposition 2.2. *Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction croissante au sens large à partir d'un certain rang, telle qu'il existe des entiers $n_0 \geq 1$, $b \geq 2$ et des réels $a > 0$, et $d > 0$ et une fonction $f : \mathbb{N} \to \mathbb{R}_+$pour lesquels*

$$\begin{array}{ll}
t(n_0) & = d \\
t(n) & = at(n/b) + f(n) \qquad n > n_0,\ n/n_0 \text{ une puissance de } b
\end{array}$$

Supposons de plus que

$$f(n) = cn^k(\log_b n)^q$$

pour des réels $c > 0$, $k \geq 0$ et q. Alors

$$t(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \text{ et } q = 0 \\ \theta(n^k (\log_b n)^{1+q}) & \text{si } a = b^k \text{ et } q > -1 \\ \theta(n^k \log\log_b n) & \text{si } a = b^k \text{ et } q = -1 \\ \theta(n^{\log_b a}) & \text{si } a = b^k \text{ et } q < -1 \\ \theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Preuve. Procédons comme dans la démonstration du résultat précédent. Soit d'abord $n = n_0 b^p$ pour un entier p. Alors

$$\begin{aligned} t(n) &= da^p + \sum_{i=0}^{p-1} a^i f(n/b^i) \\ &= da^p + \sum_{i=0}^{p-1} a^i f(n_0 b^{p-i}) = da^p + \sum_{i=1}^{p} a^{p-i} f(n_0 b^i) \\ &= da^p + c \sum_{i=1}^{p} a^{p-i} (n_0 b^i)^k (\log_b n_0 b^i)^q \\ &= da^p + c n_0^k a^p \sum_{i=1}^{p} \left(b^k/a\right)^i (\log_b n_0 + i)^q \end{aligned}$$

et comme $a^p = \theta(n^{\log_b a})$, on a $t(n) = \theta(n^{\log_b a} \gamma(n))$, où

$$\gamma(n) = \sum_{i=1}^{p} h^i (r+i)^q$$

avec $h = b^k/a$ et $r = \log_b n_0$. Si $h = 1$, alors

$$\gamma(n) = \sum_{i=1}^{p} (r+i)^q = \begin{cases} \theta(p^{1+q}) = \theta(\log_b n^{1+q}) & \text{si } q > -1 \\ \theta(\log_b p) = \theta(\log_b \log_b n) & \text{si } q = -1 \\ \theta(1) & \text{si } q < -1 \end{cases}$$

Si $h < 1$, alors $\gamma(n) = \theta(1)$. Enfin, si $h > 1$ et $q = 0$, alors $\gamma(n) = \theta(n^k / n^{\log_b a})$. D'où le résultat. ∎

Exemple. Si la fonction t vérifie, pour tout n puissance de 2,

$t(n) = 2t(n/2) + \log n$	alors	$t(n) = \theta(n)$	$(h = 1/2, k = 0, q = 1)$
$t(n) = 3t(n/2) + n \log n$	alors	$t(n) = \theta(n^{\log 3})$	$(h = 2/3, k = q = 1)$
$t(n) = 2t(n/2) + n \log n$	alors	$t(n) = \theta(n \log^2 n)$	$(h = 1, k = q = 1)$
$t(n) = 5t(n/2) + (n \log n)^2$	alors	$t(n) = \theta(n^{\log 5})$	$(h = 4/5, k = q = 2)$

Exemple. Soit t une fonction vérifiant

$$\begin{aligned} t(1) &= a \\ t(n) &= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn \qquad n > 1 \end{aligned}$$

Clairement, t est croissante, et comme $t(n) = 2t(n/2) + bn$ lorsque n est une puissance de 2, on a $t(n) = \theta(n \log n)$. Certaines définitions récurrentes de ce type peuvent être résolues de manière complète (voir exercices).

Proposition 2.3. *Soient $\alpha_1, \cdots, \alpha_k$ des fonctions de $\mathbb{N}$ dans lui-même telles qu'il existe un nombre réel $0 < K < 1$ avec*

$$\alpha_1(n) + \cdots + \alpha_k(n) \leq Kn \qquad (n > n_0)$$

Si une fonction t vérifie

$$t(n) = \begin{cases} a_0 & n \leq n_0 \\ t(\alpha_1(n)) + \cdots + t(\alpha_k(n)) + bn & n > n_0 \end{cases}$$

alors $t(n) = O(n)$.

Preuve. Nous allons montrer que

$$t(n) \leq Dn + a_0 \quad \text{avec} \quad D = (b + (k-1)a_0)/(1-K)$$

Cette inégalité est évidemment vérifiée pour $n \leq n_0$. Si $n > n_0$, alors

$$\begin{aligned} t(n) &= t(\alpha_1(n)) + \cdots + t(\alpha_k(n)) + bn \\ &\leq ka_0 + D(\alpha_1(n) + \cdots + \alpha_k(n)) + bn \\ &\leq ka_0 + DKn + bn \end{aligned}$$

Or $ka_0 + DKn + bn \leq Dn + a_0$ si, et seulement si $(k-1)a_0 + bn \leq D(1-K)n$, et cette dernière inégalité est satisfaite par l'expression donnée pour D. ∎

Exemple. Considérons la fonction définie par

$$t(n) = \begin{cases} a & n \leq n_0 \\ t(\lfloor n/5 \rfloor) + t(\lfloor 7n/10 \rfloor) + bn & n > n_0 \end{cases}$$

On a ici $\alpha_1(n) = \lfloor n/5 \rfloor$ et $\alpha_2(n) = \lfloor 7n/10 \rfloor$, et

$$\alpha_1(n) + \alpha_2(n) \leq 9n/10$$

Donc $t(n) = O(n)$.

2.2.4 Récurrences complètes

Il s'agit de suites (u_n) définies par des relations de récurrences portant sur toute la suite, plus précisément où u_n est défini en fonction de tous les u_k, pour $k < n$.

Exemple. Les relations de récurrence

$$t_n = cn + \frac{2}{n}\sum_{k=0}^{n-1} t_k \quad \text{ou} \quad t_n = n + \max_{1 \leq k < n} \left(\sum_{i=1}^{k-1} t_{n-i} + \sum_{i=k}^{n-1} t_i \right)$$

sont des relations de récurrence complètes.

Considérons la suite (t_n) vérifiant la relation de récurrence

$$t_n = h_n + \frac{2}{n}\sum_{k=0}^{n-1} t_k$$

pour $n \geq 1$, et $t_0 = 0$. Dans une première étape, nous allons remplacer cette relation de récurrence par une relation plus simple. Pour cela, observons que

$$n(t_n - h_n) = 2\sum_{k=0}^{n-1} t_k$$

On soustrait de cette équation la même équation pour $n-1$, ce qui donne

$$n(t_n - h_n) - (n-1)(t_{n-1} - h_{n-1}) = 2t_{n-1}$$

ou encore

$$nt_n = (n+1)t_{n-1} + g_n$$

en posant $g_n = nh_n - (n-1)h_{n-1}$.

Dans une deuxième étape, on applique la méthode des facteurs sommants. On a

$$t_n = \frac{1}{nf_n}\left(t_0 + \sum_{k=1}^{n} f_k g_k\right)$$

avec

$$f_n = \frac{(n-1)!}{(n+1)!} = \frac{1}{n(n+1)}$$

d'où

$$t_n = (n+1)\left(\sum_{k=1}^{n} \frac{g_k}{k(k+1)}\right)$$

Considérons maintenant le cas particulier, qui intervient dans l'analyse du tri rapide (voir chapitre 5), où

$$\begin{aligned} h_0 &= h_1 = 0 \\ h_k &= k+1 \qquad (k \geq 2) \end{aligned}$$

On a alors

$$\begin{aligned} g_0 &= 0,\ g_1 = 6 \\ g_k &= 2k \qquad (k \geq 2) \end{aligned}$$

et par conséquent

$$\begin{aligned} \sum_{k=1}^{n} \frac{g_k}{k(k+1)} &= 1 + \sum_{k=3}^{n} \frac{2k}{k(k+1)} = 1 + 2\sum_{k=4}^{n} \frac{1}{k} \\ &= 1 + 2(H_{n+1} - 1 - \frac{1}{2} - \frac{1}{3}) = 2(H_{n+1} - \frac{4}{3}) \end{aligned}$$

où H_n est le n-ième nombre harmonique. Il en résulte que

$$t_n = 2(n+1)(H_{n+1} - \frac{4}{3})$$

Notes

L'analyse d'algorithmes fait fréquemment appel à des outils mathématiques bien plus sophistiqués que ceux présentés ici. En plus du traité de Knuth

D. E. Knuth, *The Art of Computer Programming*, Vol I–III, Addison-Wesley, 1968–1973, on peut consulter, pour un bonne introduction et des compléments,

R. Graham, D. E. Knuth, O. Patashnik, *Concrete Mathematics*, Addison-Wesley, 1989,

D. H. Greene, D. E. Knuth, *Mathematics for the Analysis of Algorithms*, Birkhäuser, 1982.

Nous avons adopté la terminologie de

C. Froidevaux, M. C. Gaudel, M. Soria, *Types de données et algorithmes*, McGraw-Hill, 1990.

Exercices

2.1. Donner une expression simple pour la fonction t définie sur les entiers de la forme $n = 2^{2^k}$ par

$$\begin{aligned} t(2) &= 1 \\ t(n) &= 2t(\sqrt{n}) + \log n \qquad n \geq 4 \end{aligned}$$

2.2. Montrer que, quels que soient les entiers positifs n et d, on a

$$\sum_{k=0}^{d} 2^k \log(n/2^k) = 2^{d+1} \log \frac{n}{2^{d-1}} - 2 - \log n$$

En déduire que

$$\sum_{i=1}^{n} \log(n/i) \leq 5n$$

2.3. Montrer que la fonction t définie par

$$\begin{aligned} t(1) &= 0 \\ t(n) &= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + n \qquad n > 2 \end{aligned}$$

est $t(n) = n \lfloor \log n \rfloor + 2n - 2^{1+\lfloor \log n \rfloor}$.

2.4. Montrer que la fonction t définie par

$$\begin{aligned} t(1) &= 0 \\ t(n) &= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + n - 1 \qquad n > 2 \end{aligned}$$

est $t(n) = n\lceil \log n \rceil + 1 - 2^{\lceil \log n \rceil}$.

2.5. Montrer que

$$\sum_{i=1}^{n} \lfloor \log i \rfloor = 2 + (n+1)\lfloor \log n \rfloor - 2^{1+\lfloor \log n \rfloor}$$

2.6. Donner l'expression exacte de la fonction t définie par

$$\begin{aligned} t(1) &= 0, \quad t(2) = 2 \\ t(n) &= t(\lfloor n/2 \rfloor - 1) + t(\lceil n/2 \rceil) + t(1 + \lceil n/2 \rceil) + n \qquad n > 2 \end{aligned}$$

2.7. Montrer que pour

$$\begin{aligned} t(1) &= a \\ t(n) &= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + t(1 + \lfloor n/2 \rfloor) + bn \qquad n > 1 \end{aligned}$$

on a $t(n) = \theta(n^{\log 3})$.

2.8. Donner une expression pour la fonction t définie sur les entiers naturels par

$$t(n) = \begin{cases} 0 & n = 0 \\ 1 + t(n-1) & n \text{ impair} \\ 1 + t(n/2) & n \text{ pair} \end{cases}$$

2.9. Montrer que la suite (t_n) définie par $t_1 = 1$ et

$$t_n = n + \max_{1 \le k < n} \left(\sum_{i=1}^{k-1} t_{n-i} + \sum_{i=k}^{n-1} t_i \right)$$

vérifie $t_n \le 4n$.

2.10. On considère la suite (t_n) définie par

$$\begin{aligned} t_0 &= 1/4 \\ t_n &= \frac{1}{4 - t_{n-1}} \qquad n > 0 \end{aligned}$$

Montrer que $t_n = u_n/u_{n+1}$, où (u_n) est une suite récurrente linéaire d'ordre 2; donner son expression close. Quelle est la limite de t_n lorsque n tend vers l'infini?

2.11. Montrer que la suite définie par

$$\begin{aligned} t(n+2) &= \frac{1 + t(n+1)}{t(n)} \qquad n \ge 2 \\ t(0) &= a, \quad t(1) = b \end{aligned}$$

est périodique.