Centre de Télé-Enseignement Sciences **Université de Provence**

DEUG MIAS

module 2I5

ALGORITHMIQUE

par

Pierre Liardet

Année 2002-2003

Table des matières

Chapitre 1

La notion d'Algorithme

Ce chapitre étudie la démarche algorithmique qui part de l'analyse d'un problème pour déterminer un algorithme[1] qui résoud ce problème de manière effective. La première étape importante est celle d'écrire (ou de décrire) l'algorithme. Celui-ci peut être vu, dans de nombreux cas, comme une procédure de calcul que doit effectuer un ordinateur abstrait. L'étape suivante consiste à rendre l'algorithme opérationnel en le traduisant dans un langage de programmation approprié et directement exploitable par ordinateur. Des exemples simples illustreront la diversité des problèmes qui relèvent de méthodes algorithmiques.

1.1 De l'algorithme au programme

Un algorithme est un ensemble fini de règles opératoires (procédures de calcul, instructions) bien définies ayant pour but la résolution d'un problème. La solution représentée par l'algorithme doit pouvoir, normalement, être implémentée dans un langage de programmation pour être traitée par ordinateur (machine à calcul).

L'analyse du problème à résoudre, étape préliminaire dans la réalisation de l'algorithme, doit permettre tout d'abord d'identifier les données d'entrée de l'algorithme ainsi que les données de sortie qui fournissent la solution du problème.

L'écriture d'un algorithme doit être concise et sans équivoque. Pour atteindre cet objectif, beaucoup d'ouvrages traitant d'algorithmes utilisent un pseudo-langage ou pseudo-code. Celui-ci a pour but de transcrire la solution du problème dans un langage destiné à une machine abstraite conçue comme un ordinateur virtuel constitué :

- d'une entrée (clavier)
- d'une sortie (écran)
- d'une mémoire (dite centrale)
- d'un processeur, capable d'exécuter les instructions du pseudo-code
- éventuellement, d'autres composants plus spécifiques.

Plus restreint que le langage usuel, le pseudo-langage, étudié dans la section suivante, est cependant plus précis. En contrepartie, son champ d'action est plus limité, mais il est sensé être plus souple et plus parlant qu'un langage de programmation. Il peut s'exprimer sous divers formats : ligne d'instructions, organigramme, ou encore un schéma (dessin, graphe,...). Le pseudo-langage sert, évidemment, à écrire l'algorithme mais il doit aussi faciliter sa transcription dans un langage de programmation quelconque.

[1] Le mot algorithme tire son origine de *al'Khwarizmi*, nom d'un mathématicien arabe né à Bagdad vers 780 et mort vers 850, connu pour ses travaux sur l'algèbre et surtout pour son traité sur la numération Hindou-Arabe dont on ne connaît qu'une traduction en Latin, incomplète et modifiée (Algoritmi de numero Indorum).

Le langage Pascal est bien adapté, d'un point de vue pédagogique, pour passer d'un pseudo-code en un code utilisable sur la plupart des plate-formes informatiques. La version standard, telle qu'elle a été développée par Wirth en 1972, sera utilisée pour écrire des programmes complets et opérationnels. Les travaux pratiques qui accompagnent ce cours s'effectueront sous un environnement Unix ou Linux avec le GNU Pascal (compilateur GPC 32/64 bits). Les programmes donnés dans le cours ou proposés en exercices seront utilisables tels quels ou après de légères modifications. L'annexe 1 est consacrée au Pascal standard (version ISO-7185) et ses variations classiques sous Unix, Dos ou encore Mac. Bien que fossilisé, le Pascal développé sur Mac à conservé quelques adeptes via le logiciel ThinkPascal (Symantec), maintenant libre de droits et accessible à l'URL http ://www.think-pascal.org/. Il offre l'avantage de manipuler simplement la boîte à outils du Mac pour se fabriquer des applications à peu de frais. Une partie de l'annexe lui sera consacrée. Une autre annexe servira d'introduction au langage C. A noter qu'il existe des utilitaires qui transforment un programme Pascal en un programme en C.

1.2 Le pseudo-langage

Nous utiliserons un pseudo-langage (ou pseudo-code) proche du langage Pascal, mais délesté de toutes les contraintes syntaxiques inhérentes à un véritable langage de programmation. Cela a des avantages mais aussi des inconvénients, dont celui de ne pas écrire un code directement utilisable. Pour palier à cet inconvénient, certains algorithmes seront directement implémentés en Pascal (standard).

Nous distinguerons 6 concepts :

- les données
- l'entrée et la sortie
- les variables
- les opérateurs
- les instructions
- les procédures et fonctions.

1.2.1 Les données

Une donnée est essentiellement définie par ce qu'elle représente et l'usage qui en est fait. Par exemple, au niveau d'un microprocesseur, une donnée (élémentaire) est une suite l'impulsion électrique qui, en entrant dans le microprocesseur, va le faire passer d'un état (électrique) dans un autre. Pour un programme, une donnée peut être un symbole ou une suite de symboles entrés au moyen d'un clavier, ou encore le contenue d'un fichier, lui-même constitué de symboles...

Dans le contexte d'un pseudo-langage, les données sont habituellement classées par familles ou types. Les principaux sont :

- le type *valeur numérique*
- le type *valeur booléenne*
- le type *caractère*

et des types plus complexes tels que

- le type *chaîne de caractères*
- le type *ensemble*
- ...

La liste peut être longue, d'autant plus que l'on se donne la possibilité de créer son propre type ou de combiner plusieurs types.

Détaillons la liste ci-dessus. Généralement, les données se présentent sous forme de valeurs numériques (en fait, toutes les données sont inéluctablement "binarisées" au sein de l'ordinateur). Elles sont, soit entières (type entier), soit réelles (type réel), éventuellement complexes ou matricielles (type vecteur). Dans l'immédiat, on ne s'intéresse pas à la manière de représenter ces données, bien que cet aspect soit important puisqu'il peut agir fortement sur la performance de l'algorithme.

Une donnée du type booléen prend la valeur *vrai* (true) ou la valeur *faux* (false). Elle s'écrit le plus souvent sous la forme d'un énoncé portant sur des variables et dont l'évaluation fournit un résultat vrai ou faux. Exemples : "$x = 1$", "n pair".

Les caractères sont des lettres, des chiffres ou des symboles utiles ; ils constituent l'alphabet de référence. La juxtaposition de plusieurs caractères (y compris l'espace "vide") constitue une chaîne de caractères (string) ou mot (word). La chaîne sans caractère est admise (chaîne ou mot vide).

Le type ensemble correspond à une liste d'objets sur laquelle aucun ordre n'est envisagé : c'est donc, au sens mathématique, un ensemble donné en extension : tous les éléments constitutifs sont explicitement présents dans la liste. Exemple : le type "couleur", identifié par l'ensemble

$$\{\text{blanc, bleu, vert, jaune, rouge, noir}\}.$$

Évidemment, cet ensemble peut être modifié en fonction des couleurs utilisées.

1.2.2 Entrée/Sortie

En pratique, l'ordinateur (concret ou virtuel) communique au moins par deux périphériques : un pour entrer les données et un autre pour sortir les résultats. Ceux-ci peuvent devenir les données d'entrée d'un autre algorithme et ainsi de suite, mais pour un nombre fini de fois. On peut introduire d'autres types de périphériques d'entrée, comme une piste magnétique, un CD, un scanner... et d'autres périphériques de sortie, dont le plus utilisé est l'imprimante.

1.2.3 Les constantes et les variables

Une constante est une donnée prédéfinie, le plus souvent de type numérique (exemples : $c = 1$, $\pi = 3.14159$). Elle est une donnée fixe de l'algorithme.

Une variable est une lettre, une chaîne de caractères, susceptible de prendre différentes valeurs appartenant à un ensemble de données d'un même type. En général, la valeur courante d'une variable change au cours de l'exécution des instructions de l'algorithme. Le type d'une variable est celui de la donnée qu'elle représente. Plus concrètement, une variable doit être considérée comme l'adresse d'une mémoire. Sa valeur courante est le contenu de cette mémoire à l'instant considéré.

1.2.4 Les opérateurs

On distingue, comme dans le langage Pascal, les opérateurs suivantes :

– L'affectation : représentée par le symbole $\leftarrow$ (:= en Pascal), elle inscrit la donnée placée après le symbole $\leftarrow$ dans la mémoire représentée par la variable située devant $\leftarrow$.

Exemple : l'affectation

$$X \leftarrow 2 + a$$

place dans X la valeur $2+a$. Ici, a peut être une constante, ou une variable de type numérique.

– Les opérateurs unaires et binaires classiques qui s'appliquent aux données ou variables de type numérique :

opposé $(-\cdot)$ et inverse $(\cdot^{-1})$;
addition $(\cdot + \cdot)$ et soustraction $(\cdot - \cdot)$;
multiplication $(\cdot * \cdot)$ et division $(\cdot / \cdot)$;
exponentiation $(\cdot ** \cdot$ ou $\cdot \wedge \cdot)$.

– Les opérateurs de l'arithmétique binaire :
le "et", le "ou" et le "xor".

– Les opérateurs relationnels tels que :
$>$, $<$, $\leq$, $\geq$, $=$, $\neq$, $\equiv \pmod m$.

– Les opérateurs logiques :
$\wedge$ (et), $\vee$ (ou), $\neg$ (négation).

– Les opérateurs sur les ensembles :
$\cap$, $\cup$, $\setminus$.

– L'opérateur de concaténation entre deux chaînes de caractères :

$$[\text{chaîne}_1] \cdot [\text{chaîne}_2] = \text{chaîne}_1\text{chaîne}_2.$$

Explicitons : si $[\text{chaîne}_1] = [a_1, \ldots, a_m]$ et $[\text{chaîne}_2] = [b_1, \ldots, b_n]$ alors, par définition,

$$[\text{chaîne}_1] \cdot [\text{chaîne}_2] = [a_1, \ldots, a_m, b_1, \ldots, b_n].$$

1.2.5 Les instructions

Une instruction est, en quelque sorte, un ordre à exécuter qui peut se présenter sous des formes diverses. Certaines instructions ont des significations convenues d'avance et sont explicitées par des mots dits réservés ; elles sont dites *primitives*. En voici quelques unes, inspirées du langage Pascal et francisées pour l'occasion.

• DEBUT et FIN : instructions couplées, marquant le commencement et la fin de l'algorithme ou servant à regrouper des instructions. Elles correspondent aux classiques BEGIN et END du Pascal.

• LIRE( ) : instruction de lecture de données (READ( ) et READLN( ) en Pascal).

• ECRIRE( ) : instruction de sortie de données (WRITE( ) et WRITELN( ) en Pascal). Typiquement, l'exécution de cette instruction consiste à afficher sur un écran, ou à imprimer sur papier, ou encore à inscrire dans un fichier électronique, les données entre les parenthèses. Celles-ci peuvent aller d'une simple valeur numérique, à une longue liste de valeurs, en passant par du texte ou des expressions composées comprenant du texte et des valeurs numériques.

Exemple :

ECRIRE(*Le nombre P est premier*).

• Une instruction primitive, au sens où nous l'entendons, peut être aussi le calcul d'une fonction prédéfinie encore appelée fonction primitive du pseudo-code. Voici la liste des fonctions primitives qui sont implémentées dans le Pascal standard :

fonction	*résultat*
ABS(x)	valeur absolue de x
ARCTAN(x)	arctangente de x
CHR(x)	donne le caractère dont le numéro d'ordre (code ascci, entre 1 et 128) est x
COS(x)	cosinus de x
EOF(f)	vrai, à la lecture du caractère *fin de fichier* dans le fichier f
EOLN(f)	vrai, à la lecture du caractère *fin de ligne* dans le fichier f
EXP(x)	exponentielle de x
LN(x)	logarithme népérien de x
ODD(x)	parité de x (vrai si x est impair)
ORD(x)	numéro d'ordre de x dans un ensemble de valeurs ordonnées
PRED(x)	prédécesseur de x
ROUND(x)	arrondi de x (au plus près)
SIN(x)	sinus de x
SQR(x)	carré de x
SQRT(x)	racine carrée de x
SUCC(x)	successeur de x
TRUNC(x)	partie entière de x

Une instruction de calcul, même complexe, peut aussi être définie, si c'est utile, comme une primitive du pseudo-langage. Exemple : la division sans reste DIV(A,B) qui prend en entrée les valeurs entières A et B, avec B non nul, et donne pour résultat la partie entière du nombre rationnel A/B (précisément notée DIV(A,B)). Pour b entier > 0 fixé, la fonction $\mathrm{MOD}_b(x)$ qui prend en entrée un entier x et donne en sortie l'entier m tel que $0 \leq m < b$ et b divise $x - m$, peut être considérée comme primitive ou le résultat de l'instruction

$$\mathrm{MOD}_b(x) \leftarrow x - b * \mathrm{DIV}(x, b)\,.$$

• ALLER A : instruction de pointage qui envoie sur une étiquette (numéro de ligne).

• *Instructions conditionnelles :* nous n'en donnons qu'une, les autres seront introduites en fonction des besoins.

SI <expression à évaluer en valeurs booléennes>
ALORS (FAIRE) <instruction 1>
SINON (FAIRE) <instruction 2>.

L'instruction 1 est exécutée si l'expression à évaluer est vraie, sinon, c'est l'instruction 2 qui est exécutée. Si cette dernière condition est omise, alors si l'expression à évaluer est fausse, l'instruction 1 est considérée comme absente et l'algorithme se poursuit à la ligne suivante.

• *Instructions de boucle.* Il y en a plusieurs, mais retenons seulement la suivante :

POUR $I = x$ A $I = y$ PAR SAUT DE r FAIRE $< \ldots >$

La partie "PAR SAUT DE r" est omise lorsque $r = 1$ ou $r = -1$. Ici, $< \ldots >$ représente une ou plusieurs instructions.

Exemple :

POUR $I = 1$ A $I = N$ FAIRE
DEBUT
$X \leftarrow X + (1/I)$;
$Y \leftarrow Y + \log(I + 1) - \log I$;
$R \leftarrow R + X - Y$;
FIN

L'ordre suivant lequel sont écrites les instructions est essentiel, ce qui signifie que l'algorithme procède séquentiellement. Nous ne chercherons pas dans ce cours à identifier les algorithmes où plusieurs instructions ou morceaux de programme peuvent être exécutés en même temps (algorithmes parallèles).

1.2.6 Procédures et fonctions

Une fonction non primitive est définie par un algorithme (ou un programme) qui associe à une donnée d'entrée une valeur de sortie unique.

Une procédure est un algorithme (ou un programme) plus général qu'une fonction. Elle assemble une ou plusieurs instructions et, contrairement à une fonction, ne donne pas nécessairement une valeur de sortie (elle peut, par exemple, renvoyer à une adresse, arrêter le programme, demander des données, ...).

Fonctions et procédures ont pour vocation d'être appelés, sans changement, par d'autres algorithmes.

1.3 Écrire un algorithme

Nous adoptons une présentation type. La première ligne donne le titre de l'algorithme. Le mot réservé **entrer** débute une phrase concise qui décrit les données prises en entrée. Le mot **sortir** débute une phrase qui explique succinctement le résultat de sortie de l'algorithme. La description de l'algorithme peut s'arrêter là, il est alors dit de forme courte par opposition à la forme développée qui consiste en une écriture complète de l'algorithme au moyen d'une succession de lignes d'instructions simples ou regroupées (blocs d'instructions), encadrés par les mots réservés **début** et **fin**. Il est souhaitable de numéroter ces lignes. Une ligne d'instructions se termine par un point-virgule sauf si la ligne suivante est utilisé pour continuer l'écriture des instruction de la ligne précédente. L'algorithme commence par la première ligne d'instructions (en pseudo-code) dont l'exécution complète se termine en arrivant sur le point-virgule de fin de ligne. Une instruction peut envoyer sur une autre ligne d'instruction et peut ne pas se terminer complètement. L'algorithme, lui, est sensé toujours se terminer et produire un résultat (celui pour lequel il a été conçu !). L'algorithme a donc l'aspect général suivant :

Algorithme *GENERAL*
entrer *les données.*
sortir *le résultat.*
début
1. *instruction(s)* ;
2. *instruction(s)* ;
...
n. *instruction(s)* ;
fin.

Des commentaires peuvent accompagner les lignes d'instructions ; ceux-ci seront données entre crochets ou, comme dans le Pascal, entre (* et *), mais les accolades seront évitées.

Illustrons ces conventions par un exemple simple : écrire un algorithme, noté *MAX3*, qui calcule le plus grand de trois nombres donnés x, y et z. Commençons par traiter le cas plus simple de deux nombres. Sachant calculer $\max\{x, y\}$, on en déduit le calcul de $\max\{x, y, z\}$ par la formule

$$\max\{x, y, z\} = \max\{\max\{x, y\}, z\}, \tag{1.1}$$

qui se généralise à n nombres :

$$\max\{x_1, x_2, x_3, \ldots, x_n\} = \max\{\ldots \max\{\max\{\{x_1, x_2\}, x_3\}, .\} \ldots\}, \tag{1.2}$$

fournissant ainsi la base d'un algorithme pour calculer le plus grand nombre d'une liste donnée. L'algorithme *MAX(X,Y)* est présenté ici comme une fonction qui sera reprise dans l'algorithme définitif *MAX3*.

Fonction *MAX(X,Y)*
entrer les nombres X,Y.
sortir le plus grand.
début
1. LIRE(X,Y) ;
2. SI X $\geq$ Y ALORS MAX(X,Y) $\leftarrow$ X, SINON MAX(X,Y) $\leftarrow$ Y;
fin.

Voyons maintenant *MAX3*

Algorithme *MAX3*
entrer les nombres X,Y,Z.
sortir le plus grand des trois.
début
1. LIRE(X,Y,Z) ;
2. M $\leftarrow$ MAX{X,Y} ;
3. MAX3 $\leftarrow$ MAX{M,Z} ;
4. ECRIRE(MAX3) ;
fin.

1.4 Le programme

N. Wirth, inventeur du langage Pascal, résume la définition d'un programme par l'équation

Algorithmes + Data Structures = Programs

En fait, une fois l'algorithme conçu, celui-ci doit être traduit en langage de programmation pour être exécuté par un ordinateur. Évidemment, non seulement le programme doit être écrit correctement, mais son exécution doit correspondre exactement à l'algorithme qu'il est sensé traduire et donc, fournir la réponse correcte au problème posé. Ces objectifs sont loin d'être faciles à atteindre et la première étape qui consiste à écrire l'algorithme joue un rôle essentiel. Une méthode efficace repose sur une analyse descendante du problème : décomposer en sous-problèmes jusqu'au niveau des opérations et instructions primitives. Le programme traitant de données, celles-ci doivent être structurées et organisées de manière appropriée ; ce sera le sujet d'étude principal du prochain chapitre.

Le langage retenu dans ce cours pour écrire les programmes est le Pascal. Bien que ses performances soient modestes et ses instructions de base assez réduites, il présente des avantages pédagogiques indéniables du fait de sa simplicité et de la structure bien organisée de tout programme Pascal. Bien maîtrisé, ce langage peut s'avérer être d'une bonne efficacité pour résoudre de nombreuses questions algorithmiques issues de problèmes de tri, ou de recherche ou encore d'origine arithmétique, numérique, géométrique, graphique et bien d'autres. Nous verrons de nombreux exemples à réaliser en travaux pratiques.

Très schématiquement, un programme en Pascal standard se présente sous la forme suivante :

```
Program mon_programme ;
(* Partie déclarative *)
    label {énumération des étiquettes}
    const {liste des constantes}
      (* Exemple : PI = 3.14159 ; *)
    type {liste des types}
      (* Exemple : ma_table = array[0..9] of integer ; *)
    var {liste des variables}
      (* Exemple : ma_variable : real ; *)
(* Partie des sous-routines *)
    {liste des procédures et fonctions utilisées}
    {dans le programme}
```

```
(* Partie programme principal *)
begin
    {corps du programme, constitué d'une suite d'instructions}
    {faisant appel aux procédures et fonctions précédentes}
end.
```

La première ligne est obligatoire et doit se terminer par un point virgule. La partie déclarative doit contenir toutes les étiquettes, constantes, types et variables qui interviennent dans le programme principal. Il en est de même de la partie contenant les procédures et fonctions. Le programme principal est écrit entre un "`begin`" et un "`end.`" (bien noter le point final). De manière générale, chaque instruction se termine par un point-virgule (`;`) et chaque groupement d'instruction (instruction composée) commence par `begin` et se termine par un `end;` (ne pas omettre le point-virgule). Donnons deux exemples simples. Le premier est le classique exercice de prise en mains du compilateur :

```
program mon_premier_programme ;
begin
  writeln(' Mon premier programme') ;
  writeln(' o-o-o-o-o-o-o-o-o-o-o')
end.
```

On notera que le point-virgule juste avant le "end" est facultatif ; il peut donc être omis comme ici. Le second exemple est un peu plus élaboré.

```
program division_par_7 ;
  const
    c = 7 ;
  var
    n, q, r : integer ;
  procédure écrire (q, r : integer) ;
    begin
      writeln('résultat') ;
      writeln(n, '=', c : 1, 'x', q : 1, '+', r : 1) ;
    end ;
begin
  writeln('division de n par ', c : 1, ' :') ;
  write(' entrer n : ') ;
  read(n) ;
  q := trunc(n / c) ;
  r := n - c * q ;
  écrire(q, r) ;
end.
```

`Writeln()`, `write()`, `read` et `trunc()` sont des procédures (ou fonctions) déjà implémentées dans le langage Pascal. Les trois premières ont des significations évidentes, bien que pour `writeln()` nous avons utilisé un formatage de sortie spécifié par les deux points suivis d'un entier. Pour plus de précision, nous renvoyons à l'annexe 1. La procédure `trunc()` prend en entrée un nombre réel (ici le résultat de la division de a par b) et renvoie sa partie entière[2].

Revenons sur l'algorithme *MAX3* étudié dans la section précédente. Il peut être implémenté en Pascal comme suit :

[2]Dans certaines implémentations, lorsque a et b sont entiers, a/b donne le quotient entier de a par b, rendant inutile l'appel à `trunc()`.

```
program max3 ;
  var
    a,b,c,m : integer ;
  function max2(x,y :integer) :integer ;
  begin
  var
    if x>= y then
        max2 := x
    else
        max2 := y ;
  end ;
begin
  writeln('Calcul du plus grand de trois entiers') ;
  writeln(' Entrer trois entiers :') ;
  readln(a, b, c) ;
  m := max(a, b) ;
  m := max(m, c) ;
  writeln('max', a : 2, ',', b : 2, ',', c : 2, '=', m : 2) ;
end.
```

Ce programme peut aisément être transformé en une fonction et servir comme telle dans un autre programme. Nous allons maintenant donner d'autres exemples pour illustrer de manière plus complète la grande diversité des problèmes traités en algorithmique.

1.5 Algorithmes de l'arithmétique élémentaire

Les systèmes de numération fournissent de nombreux exemples d'algorithmes, en particulier tous ceux qui concernent les opérations usuelles sur les nombres entiers. Celles-ci jouent un rôle de premier plan dans les ordinateurs modernes ; on les veut rapides et si possible, parallélisables. Nous allons nous intéresser à l'addition, avec l'algorithme classique par lecture des chiffres de la droite vers la gauche. L'algorithme d'Avizienis qui met en jeu une numération redondante et permet d'ajouter deux nombres séquentiellement par lecture de gauche à droite sera étudié plus tard.

1.1 *Théorème (et définition).* *Soit b un entier ≥ 2 et n un entier ≥ 0. Tout entier N tel que $0 \leq N \leq b^n - 1$, s'écrit de manière unique sous la forme*

$$N = c_0 + c_1 b + \cdots + c_{n-1} b^{n-1} \text{ avec } 0 \leq c_k \leq b-1 \text{ pour tout } k \in \{0, 1, \ldots, n-1\}. \quad (1.3)$$

Par définition, (1.3) est appelé le développement de N en base b et $c_k = c_k(N)$ est le k-ième chiffre de ce développement.

Démonstration. Nous avons choisi de donner ici une preuve non constructive mais très élémentaire, remettant à plus tard l'écriture d'un algorithme spécialement dédié au calcul du développement d'un entier dans une base donnée. Il est clair que les entiers $0, 1, \ldots, b-1$ admettent un développement (1.3) unique.
Soit M l'ensemble des entiers $m \geq 0$ qui n'admettent pas un développement (1.3) (l'unicité n'est pas encore considérée). Si M est vide, tous les entiers ≥ 0 ont un développement de la forme (1.3). Si M n'est pas vide, il admet un plus petit élément $m \geq b$. Alors $N = m - 1$ admet un développement (1.3). Si $c_0 < b - 1$ alors

$$m = (c_0 + 1) + c_1 b + \cdots + c_{n-1} b^{n-1} \quad (c_\nu = 0 \text{ si } \nu \geq n)$$

à la forme requise, en contradiction avec $m \in M$. Si $c_0 = b - 1$, alors, il existe k tel que $c_i = b - 1$ pour $0 \leq i < k$ et $c_k < b - 1$. Un calcul facile donne

$$m = (c_k + 1) b^k + c_{k+1} b^{k+1} + \ldots$$

les termes $c_j b^j$ de cette somme étant nuls pour $j \geq n+1$. Évidemment, ce développement respecte les conditions exigées par le développement de m en base b, ce qui donne encore une contradiction. En conséquence, M est vide. Voyons maintenant l'unicité. Si N admet le développement (1.3) et le développement $N = c'_0 + c'_1 b + \cdots + c'_{n-1} b^{n-1}$ avec $0 \leq c'_0 \leq b-1$ (simplement !) alors $(c_0 - c'_0)$ est multiple de b et comme

$$-b < c_0 - c'_0 < b,$$

nécessairement $c_0 = c'_0$. On a donc l'unicité du chiffre c_0. La poursuite du raisonnement avec l'entier $(N - c_0)/b$, montre que $c_1 = c'_1$. Un raisonnement bien conduit donne finalement l'unicité de tous les chiffres c_i, $i = 0, 1, 2, \ldots$

■

Le développement (1.3) est souvent représenté par $(c_{n-1} \cdots c_0)_b$, les chiffre c_k étant alors considérés comme des lettres et non pas des nombres. Classiquement, en base seize ($b = 16$), les chiffres "10, 11, 12, 13, 14, 15" sont représentés respectivement par "A, B, C, D, E, F". Si l'on remplace n par m avec $m > n$, le développement de N se déduit du précédent avec les chiffres $c_{m-1}, \ldots, c_n$ tous égaux à 0. Pour cette raison, on ne fait pas de distinction entre ces deux développements.

1.5.1 Algorithme de l'addition

Dans un langage évolué, l'additon de deux nombres entiers peut se résumer à l'algorithme suivant :

Algorithme *Add(A,B)*
entrer `deux nombres` A `et` B ;
sortir `la somme` $A + B$.
début
1. LIRE(A) ;
2. LIRE(B) ;
3. $C \leftarrow A + B$;
4. ECRIRE(C) ;
fin

Bien évidemment, cet algorithme peut être érigé en une instruction primitive notée `ADD(A,B)`, directement implémentée dans la machine, ou encore fournir le calcul d'une fonction. Maintenant, il est possible de s'intéresser aux calculs effectués au niveau du microprocesseur. Sachant que, de fait, les nombres entiers y sont représentés en binaire, le problème devient le suivant : donner un algorithme d'addition de deux nombres entiers écrits en base deux. Il n'est pas plus difficile de résoudre ce problème en base deux qu'en base quelconque b. Voici une solution :

Algorithme $Add(N, N')_b$
entrer `les entiers` $N = (0 c_{n-1} c_{n-1} \cdots c_0)_b$ `et` $N' = (0 c'_{n-1} \cdots c'_0)_b$
`représentés en base` b ;
sortir `la somme` $S = (s_n \cdots s_0)_b$ `de` N `plus` N' `en base` b.
début
`Pour` k `allant de` 0 `à` n, `calculer` t_{k+1} (`$k$-ième retenue`) `et` s_k
(`$k$-ième chiffre de la somme` $N + N'$ `en base` b) `par` :

$$\left\{ \begin{array}{lcl} t_0 & = & 0\,; \\ t_{k+1} & = & \left\{ \begin{array}{ll} 0 & \texttt{si } c_k + c'_k + t_k < b, \\ 1 & \texttt{sinon}\,; \end{array} \right. \\ s_k & = & c_k + c'_k + t_k - b t_{k+1}. \end{array} \right.$$

fin

Notons que la dépendance entre t_k et t_{k+1} oblige à commencer l'addition de droite à gauche. L'algorithme n'est pas vraiment écrit ici en pseudo-code, mais il est clairement défini. En outre, on ne s'est pas intéressé à un format de sortie pour S.

Voici maintenant une toute autre manière de concevoir et d'écrire l'algorithme précédent. Nous utilisons un graphe étiqueté avec deux sommets notés 0 et 1, associés aux retenues possibles (0 ou 1). De chaque sommet t part b^2 arcs étiquetés, d'étiquettes $(c, c', s) \in \{0, \ldots, b-1\}^3$ telles que $s = c + c' + t$ ou $s = c + c' + t - b$, de sorte que l'on ait toujours $0 \leq s \leq b-1$. Un arc issu du sommet t et d'étiquette (c, c', s) a pour extrémité le sommet 0 si $c + c' + t < b$ et le sommet 1 dans le cas contraire. Notons que la valeur de s est déterminée par le couple (c, c') et le sommet t. Ce graphe rend "mécanique" l'addition de la manière suivante. On se place initialement au sommet 0 et on lit successivement les couples $(c_0, c'_0), (c_1, c'_1), \ldots, (c_{n-1}, c'_{n-1})$. Au début de la lecture, on se place à l'extrémité de l'arc dont l'étiquette (c, c', s) commence par (c_0, c'_0). Il n'y en qu'un, issu du sommet 0 et la valeur de s (troisième coordonnée de l'étiquette) donne s_0. Maintenant, en cours de lecture, supposons que l'on soit au sommet t et qu'on lise le couple de chiffres (c_k, c'_k). Il n'y a qu'un arc issu du sommet t dont l'étiquette commence par (c_k, c'_k). La valeur s de la troisième coordonnée donne s_k et l'arc d'étiquette (c_k, c'_k, s_k) a pour extrémité le sommet t_{k+1} (de sorte que $s_k = c_k + c'_k + t - t_{k+1}b$). Ce graphe, encore appelé automate, schématise une petite machine pour calculer la somme de deux nombres entiers exprimés en base b. On a tracé, en quelque sorte, le plan d'un *coprocesseur*, spécifiquement dédié à ce calcul. L'étiquette (c, c', s) correspond à une entrée/sortie. L'entrée est formée du couple (c', c), la sortie est donnée par s. L'algorithme fonctionne *à la volée* : dès la lecture du couple des k-ièmes chiffres en entrée, on obtient en sortie le k-ième chiffre de la somme. La dernière entrée étant *forcée* à $(0, 0)$, on termine en sortie par le dernier chiffre (à gauche) de la somme.

Donnons explicitement le graphe qui correspond à l'addition en base deux, chaque étiquette (c, c', s) étant ici remplacé (pour facilité l'écriture et la lecture) par le mot $cc's$. La flèche sans étiquette arrivant sur le sommet 0 indique le sommet initial, c'est-à-dire la position de départ pour calculer la somme.

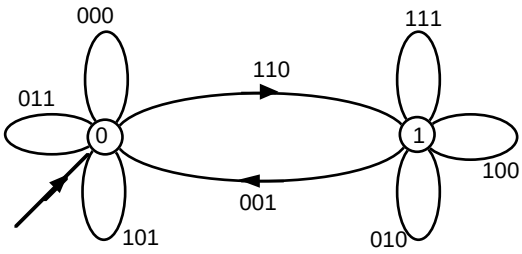


Figure 1 : graphe étiqueté de l'addition binaire.

***Exercice* 1.1** Calculer en binaire au moyen du graphe étiqueté de la figure 1 la somme $(11011)_2 + (10101)_2$. Vérifier le résultat en utilisant la base dix.

***Exercice* 1.2**
a) Utiliser un graphe étiqueté pour décrire l'algorithme d'addition de 1 en base binaire.
b) Même question qu'en a) mais en base trois au lieu de deux.
c) Construire le graphe étiqueté qui correspond à l'addition en base trois.

***Exercice* 1.3**
a) Écrire un algorithme qui prend en entrée une suite binaire $B = [b_1..b_n]$ ($b_i \in \{0, 1\}$) et donne en sortie la parité (paire, impaire) du nombre de 1 dans B.

b) Utiliser un graphe étiqueté à deux sommets pour réaliser la même opération, les sommets correspondant aux deux états possibles (*pair* et *impair*) de la sortie.

1.5.2 Algorithme de numération en base b

Cas binaire

Étant donné ce qui précède, il devient pressant de disposer d'un algorithme qui traduise un entier quelconque en son développement binaire. L'entier est habituellement entré au clavier dans sa représentation usuelle, c'est-à-dire en base dix.

Pour écrire cet algorithme, nous utiliserons une instruction primitive de parité, notée $\texttt{PAR}(A)$, qui prend la valeur 0 ou 1 selon que A est un entier pair ou impair. Pour donner le résultat sous la forme d'une chaîne (ou mot) binaire (binary string), nous conviendrons que $[\texttt{PAR}(A)]$ désignera la lettre 0 (resp. 1) qui représente la valeur numérique correspondante. Il s'agit, en effet, de ne pas confondre un chiffre (lettre) de la valeur numérique qui lui est associée. Nous introduisons aussi l'opération de concaténation des chaînes. Rappelons sa définition : soit $B = B_1 \cdots B_m$ et $C = C_1 C_2 \cdots C_n$ deux chaînes de caractères (lettres, chiffres ou tout autre symbole) ; la concaténation de B suivi de C est la chaîne $B_1 \cdots B_m C_1 \cdots C_n$ notée $B{\cdot}C$.

Algorithme *Binaire( )*
entrer un entier A ;
sortir les chiffres $C_0, \ldots, C_n$ de A écrit en base deux et les écrire suivant la chaîne binaire $C = C_n C_{n-1} \ldots C_0$.
Début
1. LIRE(A) ;
2. $C \leftarrow [\texttt{PAR}(A)]$;
3. Si $A = 0$ ALLER A $\boxed{7}$;
4. $A \leftarrow [A - \texttt{PAR}(A))/2$;
5. $C \leftarrow [\texttt{PAR}(A)].C$;
6. ALLER A $\boxed{3}$;
7. ECRIRE(l'écriture binaire de A est C) ;
fin

Faisons quelques observations. La variable C représente une chaîne binaire. Elle est déclarée comme telle par l'instruction $\boxed{2}$ qui l'initialise à [0] ou [1] suivant que A est pair ou impair. Ensuite, elle est modifiée en $\boxed{5}$ par des concaténation successives de la forme $[0]{\cdot}C$ ou $[1]{\cdot}C$, suivant la valeur courante de $\texttt{PAR}(A)$. Notons que l'instruction de la ligne $\boxed{3}$ envoie, si $A = 0$, sur l'instruction de la ligne $\boxed{7}$ à laquelle fait suite la ligne **fin**.

Prenons l'exemple de $A = 13$ et traçons les valeurs successives de A et C avec en regard le numéro I de l'instruction exécuté. Les étapes intermédiaires qui ne changent ni A ni C sont omises.

I	A	C
1 :	13	–
2 :	13	1
4 :	6	1
5 :	6	01
4 :	3	01
5 :	3	101
4 :	1	101
5 :	1	1101
4 :	0	1101
7 :	0	1101

Vérification : on a bien $13 = 2^3 + 2^2 + 0 \times 2 + 1$.

Cas d'une base b quelconque

Introduisons une instruction primitive $\texttt{MOD}_b(A)$ (cf. supra) qui prend en entrée l'entier A et renvoie l'entier r tel que $0 \leq r < b$ et $A - r$ multiple de b (r est donc le reste de la division

usuelle de A par b). Définissons maintenant $[\texttt{MOD}_b(A)]$: si le résultat $r = \texttt{MOD}_b(A)$ est inférieur ou égal à 9, alors $[r]$ désignera la lettre constituée du chiffre r ; si $10 < b \leq 37$ et $10 \leq r < b$ on choisira pour $[r]$ la $(r-9)$-ième lettre de l'alphabet. Pour $b > 37$, on créera une table spéciale pour étendre l'alphabet des chiffres afin de couvrir les besoins. Un bon moyen de la réaliser est d'utiliser les mots binaires en guise de chiffres. Remplaçons dans l'algorithme *Binaire*(A) la primitive `PAR()` par $\texttt{MOD}_b\texttt{()}$ et l'instruction $\boxed{4}$ par

4^*. $A \leftarrow (A - \texttt{MOD}_b(A))/b$;

on obtient un algorithme, désigné par `b_AIRE( )`, qui prend en entrée un entier A et renvoie le mot qui représente le développement de A en base b.

Il peut être utile de résoudre le problème inverse à savoir, de construire un algorithme qui prend en entrée un mot représentant un entier N écrit en base b et calcule sa valeur numérique (celle-ci n'est pas sensée être donnée en sortie sous la forme décimale). Introduisons une fonction primitive[3] `V( )` qui prend en entrée les chiffres de la base b et calcule sa valeur numérique N. L'algorithme suivant est une réponse au problème posé.

Algorithme *Valeur*$_b(C_n \cdots C_0)$
entrer un mot $C_n \cdots C_0$ écrit sur l'aphabet des chiffres de la base b ;
sortir la valeur numérique N dont $C_n \cdots C_0$ est le développement en base b.
1. LIRE($C_n \cdots C_0$) ;
2. $N = 0$;
3. POUR i = n A 0 FAIRE $N \leftarrow b * N + V_b(C_i)$;
4. ECRIRE(*La valeur numérique de* $C_n \cdots C_0$ *est* N) ;
fin

1.5.3 La division euclidienne

Les propriétés arithmétiques des nombres entiers sont basées sur l'existence de la division euclidienne dans $\mathbf{N}$; son importance est fondamentale. Donnons ci-dessous le théorème qui règle cette division.

1.2 *Théorème (et définition).* *Pour tout entier a et tout entier $b \neq 0$, il existe deux entiers q et r, uniques, tels que*

$$a = qb + r \quad \& \quad 0 \leq r < |b|. \tag{1.4}$$

L'entier q est, par définition, le quotient (resp. l'entier r est le reste) de la division euclidienne de a par b.

Démonstration. Nous supposons tout d'abord que a et b sont positifs. Soit $E(a,b)$ l'ensemble des entiers $k \geq 0$ tels que $kb \leq a$. Ce sous-ensemble de $\mathbf{N}$ est fini car il est majoré (par a). Il admet donc un plus grand élément q. Par construction $qb \leq a < (q+1)b$, donc l'entier $r = a - qb$ vérifie $0 \leq r < b$. Il reste à monter que q et r sont les seuls entiers satisfaisant à (1.4). Or l'égalité $a = bq + r$, jointe aux inégalités $0 \leq q < r$, montre que q est bien le plus grand élément de $E(a,b)$, lequel est unique.

Les cas où a et b ne sont pas tous les deux positifs se ramènent simplement au cas précédent. En effet, si $a > 0$ et $b < 0$, il suffit de remplacer b par $-b$ dans (1.4) pour obtenir $a = (-q)b + r$ avec $0 \leq r < |b|$. Si $a < 0$ et $b > 0$, la division euclidienne de $-a$ par b donne $-a = q'b + r'$. Pour $r' = 0$ cela donne $a = (-q')b$, et pour $r' \neq 0$ (mais $r' < b$) on a $a = (-q'-1)b + (b - r')$, d'où (1.4) avec $q = -q' - 1$ et $r = (b - r')$. Il reste le cas de a et b négatifs. Effectuons la

[3] Celle-ci étant habituellement réalisée par un algorithme (ou une suites d'instructions) spécifique, mais on ne se préoccupe pas ici de sa réalisation, d'où le qualificatif "primitive"

division euclidienne $-a = q'(-b) + r'$. Si $r' = 0$, on obtient (1.4) avec $q = q'$ et $r = 0$. Si $r' \neq 0$ on a encore (1.4) avec $q = q'+1$ et $r = -b-r'$. L'unicité du couple (q, r) vérifiant (1.4) est encore assurée, mais nous laissons au lecteur le soin de le prouver. ■

L'algorithme suivant effectue la division euclidienne entre entiers positifs. Il est basé uniquement sur les primitives d'affectation, d'addition, de soustraction et de comparaison. Il s'inspire de la démonstration du théorème. Bien évidemment, si les entiers, en données d'entrée, sont écrits en binaire, l'algorithme bénéficierait d'être "binarisé", ses instructions ne manipulant alors que des chaînes binaires.

Algorithme *Div_Euclide*(A, B)
entrer `les entiers positifs` A `et` B`, avec` B `non nul ;`
sortir `le quotient` Q `et le reste` R `de la division euclidienne de` A `par` B`.`
`1.` LIRE(A) ; LIRE(B) ;
`2.` $X \leftarrow B$;
`3.` $Q \leftarrow 0$;
`4. Si` $X > A$ `ALLER A` 8 ;
`5.` $Q \leftarrow Q + 1$;
`6.` $X \leftarrow X + B$;
`7. ALLER à` 3 ;
`8.` $R \leftarrow A - X + B$;
`9.` ECRIRE(*La division de A par B donne pour quotient Q et pour reste R*) ;
fin

Un exemple d'application de la division euclidienne

Rappelons qu'un sous-groupe G de $\mathbf{Z}$ est une partie non vide de $\mathbf{Z}$ telle que pour tout $(g, g') \in G^2$, on a

$$g \pm g' \in G.$$

En particulier 0 est dans G et si g est un élément de G, alors $-g$ appartient aussi à G.

1.3 ***Théorème.*** *Pour tout sous-groupe G de $\mathbf{Z}$, il existe un entier $\gamma \geq 0$ tel que $G = \gamma{\cdot}\mathbf{Z}$ $(=\{\gamma n\,;\ n \in \mathbf{Z}\})$.*

Démonstration. Soit G un sous-groupe de $\mathbf{Z}$. On peut écarter le cas banal $G = \{0\}$. Considérons alors l'ensemble P des éléments strictement positifs de G ; c'est une partie non vide de $\mathbf{N}$ et donc, admet un plus petit élément γ $(\in G)$. Le sous-groupe $\gamma{\cdot}\mathbf{Z}$ engendré par γ est évidemment contenu dans G ; reste à établir l'égalité. Soit $x \in G$ et montrons que x est un multiple de γ. La division euclidienne de x par γ donne $x = n\gamma + m$ avec $0 \leq m < \gamma$. Or $n\gamma$ est dans G, donc $m = x - n\gamma$ est aussi dans G. Comme g est le plus petit élément positif non nul de G, les inégalités $0 \leq m < \gamma$ ne sont vérifiées que pour $m = 0$, ce qui implique $x \in \gamma{\cdot}\mathbf{Z}$. ■

1.5.4 Le PGCD

Un entier d est un *diviseur* de l'entier a s'il divise (exactement) a c'est-à-dire, s'il existe un entier b tel que

$$a = db\,.$$

La divisibilité de a par d se note symboliquement par $d|a$. On définit ainsi une relation binaire $\cdot|\cdot$ sur l'ensemble $\mathbf{N}^+$ des entiers > 0. Cette relation est en fait un ordre puisqu'elle est :

- réflexive $(\forall\, x,\ x|x)$;
- transitive $(x|y\ \&\ y|z\ \Rightarrow\ x|z)$;
- anti-symétrique $(x|y\ \&\ y|x\ \Rightarrow\ x = y)$.

Elle admet un plus petit élément, à savoir 1.

La relation de divisibilité possède des propriétés importantes dont la suivante :

– pour tout couple d'entiers (x, y), on a

$$d|a \ \& \ d|b \ \Rightarrow \ d|(ax + by).$$

Considérons maintenant deux entiers a et b non nuls simultanément. L'ensemble des diviseurs (positifs) d communs à a et b est fini ; il possède donc un plus grand élément (pour l'ordre usuel sur les entiers) appelé *plus grand commun diviseur*. On le note $\text{pgcd}(a, b)$ ou simplement (a, b) malgré l'ambiguïté de cette notation. On conviendra de poser $\text{pgcd}(0, 0)=0$. Énonçons, sans démonstration, les propriétés classiques du pgcd :

$$\begin{aligned}
&\text{pgcd}(a, b) = \text{pgcd}(b, a)\\
&\text{pgcd}(a, b) = \text{pgcd}(-a, b)\\
&\text{pgcd}(a, b) = \text{pgcd}(|a|, |b|)\\
&\text{pgcd}(a, b) = \text{pgcd}(a, b + ka)\\
&\text{pgcd}(a, 0) = |a| \, .
\end{aligned}$$

1.4 *Théorème.* *Pour tout couple d'entiers (a, b) on a*

$$a\cdot\mathbf{Z} + b\cdot\mathbf{Z} = \text{pgcd}(a, b)\cdot\mathbf{Z}.$$

En particulier, tout diviseur commun de a et b est un diviseur de $pgcd(a, b)$.

Démonstration. D'après le théorème 1.3, il existe un entier $c \geq 0$ tel que $a\cdot\mathbf{Z} + b\cdot\mathbf{Z} = c\cdot\mathbf{Z}$. En particulier, a et b sont des multiples de c donc c est plus petit ou égal à $\text{pgcd}(a, b)$. Maintenant, si d divise a et b, il divise c qui est de la forme $au + bv$. D'où $d \leq c$ et par suite $\text{pgcd}(a, b) \leq c$. Cette démonstration montre aussi que tout diviseur de a et b est diviseur de $\text{pgcd}(a, b)$, ce qui n'est pas une conséquence immédiate de la définition. ∎

1.5 *Corollaire.* *Soient a et b deux entiers et $d = \text{pgcd}(a, b)$. Il existe deux entiers u, v tels que*

$$d = au + bv.$$

***Remarque* 1.1** Le théorème 1.4 peut servir de nouvelle définition du pgcd, l'ancienne définition devenant alors un théorème !

L'algorithme suivant qui calcule le pgcd repose sur les égalités

$$\text{pgcd}(a, b) = \text{pgcd}(b, a) = \text{pgcd}(a - b, b) \, .$$

Il a été décrit (sous une forme voisine de celle donnée ici) par Euclide dans ses Éléments.

Algorithme *PGCD*(a, b)
entrer les entiers positifs a et b, avec a ou b non nul ;
sortir $\text{pgcd}(a, b)$.
début
1. LIRE a ;
2. $A \leftarrow a$;
3. LIRE b ;
4. $B \leftarrow b$;
5. SI $B = 0$ ALORS $\text{PGCD}(A, B) = A$ & ALLER A $\boxed{10}$;
6. SI $A = 0$ ALORS $\text{PGCD}(A, B) = B$ & ALLER A $\boxed{10}$;
7. SI $B > A$ ALORS $T \leftarrow A$; $A \leftarrow B$; $B \leftarrow T$;
8. $A \leftarrow A - B$;
9. ALLER A $\boxed{6}$;

10. ECRIRE($pgcd(a, b) = \text{PGCD}(A, B)$) ;
fin

Notons que l'exécution de 7 consiste à échanger les valeurs de A et B si $B > A$. L'algorithme entre ensuite dans l'instruction 8 avec $A \geq B$ et la valeur de A est remplacée par $A - B$. Puis il va à l'instruction 6 qui renvoie sur l'instruction de sortie 10 si $A = 0$ ou revient sur 7 sinon.

A titre d'exemple, implémentons complètement cet algorithme en langage Pascal :

```
program pgcd (entrer, sortir) ;
(* Programme PGCD en séquentiel, classique, sans utiliser*)
(* la division euclidienne. La taille des entiers est *)
(* limitée à maxint (= 2^16 - 1 = 32767). *)
  var
    a, b : integer ;
  function PGCD (x, y : integer) : integer ;
    var
      t : integer ;
  begin
    repeat
      if x < y then
          begin (* on veut y <= x d'où l'échange qui suit*)
            t := x ;
            x := y ;
            y := t
          end ;
      x := x - y (* descente basée sur pgcd(x,y)=pgcd(x-y,y) *)
    until x = 0 ; (* fin de la descente *)
    PGCD := y
  end ;
begin   writeln('Calcul du pgcd de deux nombres') ;
  writeln('a et b > 0 : ') ;
  write('a=') ;
  readln(a) ;
  write('b=') ;
  readln(b) ;
  if (a > 0) and (b > 0) then
  writeln('pgcd(a,b)=', PGCD(a, b))
end.
```

Nous verrons plus loin (chapitre 5) un algorithme récursif pour calculer le pgcd, c'est-à-dire un algorithme qui fait appel à lui-même dans son exécution.

***Exercice* 1.4** Modifier l'algorithme $Add(A, B)_b$ de manière à écrire le résultat S du calcul sous la forme d'une chaîne de caractères représentant l'écriture de S en base b.

***Exercice* 1.5** Appliquer l'algorithme $Valeur_b(C_n \cdots C_0)$ dans le cas où $b = 3$ et $C_n \cdots C_0 = 12021$ en donnant les valeurs successives de N au cours de l'exécution de la boucle de la ligne 3.

***Exercice* 1.6**
a) Écrire complètement l'algorithme `b_AIRE( )`, selon la méthode proposée dans le cours, qui prend en entrée un entier A et donne en sortie le développement $C = C_n C_{n-1} \cdots C_0$ de A en base b (cet algorithme calcule d'abord C_0 puis C_1 et ainsi de suite jusqu'à C_n).
b) Donner un algorithme (dit *glouton*) qui prend aussi en entrée un entier A, donne en sortie son développement $C = C_n C_{n-1} \cdots C_0$ en base b mais calcule successivement C_n, C_{n-1},..., jusqu'à C_0.

***Exercice* 1.7** Au lieu d'utiliser les chiffres de 0 à 9 pour écrire les entiers en base dix, on choisit les chiffres "binaires" formés des mots binaires 0000 pour le chiffre 0, 0001 pour le chiffre 1, 0010 pour le chiffre 2, ...,

1001 pour le chiffre 9. Écrire un algorithme développé (i.e. instructions comprises) qui prend en entrée un entier écrit en base dix et sort le mot binaire représentant cet entier avec les nouveaux chiffres. Écrire un algorithme (développé) qui fait l'inverse : prend en entrée un entier écrit avec les 10 chiffres binaires ci-dessus et donne en sortie l'entier en base dix.

***Exercice* 1.8** Appliquer l'algorithme *Div_Euclide*(A, B) pour $A = 401$ et $B = 33$ en traçant les valeurs successives de X et Q ainsi que la suite complète des numéros de lignes à exécuter consécutivement.

***Exercice* 1.9** Adapter l'algorithme *Div_Euclide(A,B)* de manière à ce qu'il effectue la division euclidienne entre entiers positifs ou négatif (le diviseur étant toujours supposé non nul !). Traduire cet algorithme par un programme Pascal.

***Exercice* 1.10** Soient a et b des entiers non nuls simultanément.
a) Montrer que si $\text{pgcd}(a, b) = d$, alors $\text{pgcd}(\frac{a}{d}, \frac{b}{d}) = 1$;
b) Démontrer le résultat suivant (lemme d'Euclide-Gauss) : si $c|ab$ et $\text{pgcd}(c, a) = 1$, alors $c|b$.

***Exercice* 1.11** Donner un algorithme qui calcule le pgcd de trois entiers positifs non tous nuls a, b, c.

***Exercice* 1.12** Donner un algorithme qui prend en entrée un couple d'entiers (a, b) tel que $b \neq 0$ et donne en sortie un couple d'entiers (a', b') tel que $\text{pgcd}(a', b') = 1$ et $\frac{a}{b} = \frac{a'}{b'}$.

***Exercice* 1.13** On suppose donnée un fonction primitive[4] `PPP( )` qui prend en entrée un nombre entier non nul x et renvoie le plus petit nombre premier diviseur de x si $x \neq \pm 1$ et 1 sinon. Donner un algorithme sous forme développée, mettant en jeu la fonction `PPP( )` qui prend en entrée un entier x distinct de 0 et ± 1, et donne en sortie la factorisation de x en produit de nombres premiers. L'algorithme sera désigné par *FACTORISER( )* et si la factorisation de x en puissance de nombres premiers est $\pm p_1^{a_1} \dots p_k^{a_k}$, la sortie de l'algorithme *FACTORISER( )* sera donnée sous la forme $(\pm 1)(p_1, a_1) \cdots (p_k, a_k)$.

1.6 Échanger

Dans l'algorithme $PGCD(a, b)$, la ligne 7 consiste à échanger les valeurs de A et B lorsque $B > A$. Elle fournit ainsi un algorithme qui échange deux valeurs entre elles. Explicitons-le :

Algorithme *ECHANGE(A,B)*
entrer `les variables` A `et` B `affectées des valeurs` a, b `respectivement.`
sortir `les variables` A `et` B `affectées respectivement des valeurs` b `et` a.
début
1. $A \leftarrow a$;
2. $B \leftarrow b$;
3. $T \leftarrow A$;
4. $A \leftarrow B$;
5. $B \leftarrow T$;
fin

***Exercice* 1.14** Vérifier dans le cas $a = 10$ et $b = 15$ que l'algorithme $ECHANGE(A, B)$ permute bien ces deux valeurs. Donner ensuite une preuve générale montrant que cet algorithme échange bien A et B ; pour cela on tracera les contenus des mémoires A et B au cours de l'exécution de l'algorithme.

***Exercice* 1.15** Écrire un algorithme qui prend en entrée un triplet $[a, b, c]$ de nombres et donne en sortie le triplet $[b, c, a]$ (permutation circulaire).

***Exercice* 1.16** [5] Une permutation σ de $E_n = \{1, \dots, n\}$ et une application bijective de E_n dans E_n. Si $n \geq 2$, une transposition est une permutation qui échange deux éléments distincts de E_n et laisse fixe les autres éléments. Dans la suite, on suppose $n \geq 2$.
a) Montrer que pour toute permutation σ de E_n telle que $\sigma(n) \neq n$, il existe une transposition τ (resp. une transposition τ') telle que $\tau \circ \sigma(n) = n$ (resp. $\sigma \circ \tau'(n) = n$).
b) Montrer qu'il existe un algorithme qui construit au plus $n-1$ transpositions $\tau_1, \dots, \tau_m$ $(m \leq n-1)$ telles

[4] c'est-à-dire que l'on ne se préoccupe pas de son implémentation (celle-ci sera faite plus loin).
[5] Cet exercice est difficile.

que $\tau_1 \circ \ldots \circ \tau_m = \sigma$ (on pose $m = 0$ si l'algorithme ne donne pas de transposition et l'on convient que dans ce cas, le produit (vide) $\tau_1 \circ \ldots \circ \tau_m$ est égal à l'identité). Montrer que cette décomposition n'est pas nécessairement unique mais que pour toute décomposition de σ en produit de transpositions, la parité du nombre de ces transpositions est constante (s'il elle est paire, la permutation est dite paire, sinon, elle est dite impaire).
c) Écrire en détail un algorithme qui décompose une permutation quelconque de E_n en un produit d'au plus $n-1$ transpositions.
d) Donner un exemple de permutation de E_n qui s'écrit comme produit de $n-1$ transpositions et pas moins (penser à une permutation "circulaire").

1.7 Le tri par insertion

Une méthode de tri (*sorting method* en anglais) opère sur une liste d'éléments pris dans un ensemble E totalement ordonné (pour une relation d'ordre total donnée, notée ici par $\leq$) et consiste à réécrire ces éléments dans une liste allant du plus petit au plus grand (ou du plus grand au plus petit). Il n'est pas exclu que la liste ait des éléments identiques. Une telle liste est dite de longueur n si elle est constituée de n éléments (distincts ou pas) ; par construction, elle est élément de l'ensemble produit E^n, c'est-à-dire un n-uple à coordonnées dans E. Un algorithme de tri prend donc en entrée $L[1..n] = [L_1, \ldots, L_n] \in E^n$ et donne en sortie $L^\sigma[1..n] = [L_{\sigma_1}, \ldots, L_{\sigma_n}]$ où $\sigma : k \mapsto \sigma_k$ est une permutation de l'ensemble $\{1, \ldots, n\}$ (i.e. une bijection de cet ensemble sur lui-même), telle que

$$L_{\sigma_1} \leq \ldots \leq L_{\sigma_n} .$$

Si tous les L_k sont distincts, la permutation σ est unique. Dans le cas contraire, plusieurs permutations donnent le même résultat.

Nous nous intéressons, dans ce chapitre, au tri par insertion ; il en existe bien d'autres qui seront étudiés ultérieurement. Le tri par insertion est familier aux joueurs de cartes (bridge, belote,...) lorsqu'ils rangent leur donne en classant les cartes, par exemple, de gauche à droite en plaçant d'abord les cœurs, puis les piques, les carreaux et enfin les trèfles. Au sein de chaque couleur les cartes sont ordonnées suivant l'ordre 7, 8, 9, 10, Valet, Dame, Roi et As. Le tri par insertion commence par la première carte reçue (mais il n'y a rien à trier pour l'instant), la seconde carte est placée en fonction de la première. Les suivantes sont insérées entre les cartes déjà rangées de manière à préserver l'ordre indiqué plus haut. Le programme qui suit procède de manière analogue. Pour $i = 1$ à $n-1$, il compare les éléments de $L[1..(i-1)] = [L_1, \ldots, L_{i-1}]$ avec $L[i]$ jusqu'à l'insertion de $L[i]$ en bonne place.

Illustrons cette méthode sur un exemple concret en prenant la liste d'entiers $L[1..5] = [3, 8, 5, 2, 6]$, le tri se faisant suivant l'ordre naturel des entiers, ce qui donne successivement :

$$[3, 8, 5, 6], \quad [\mathbf{3}, \mathbf{8}, 5, 2, 6], \quad [\mathbf{3}, \mathbf{5}, \mathbf{8}, 2, 6], \quad [\mathbf{2}, \mathbf{3}, \mathbf{5}, \mathbf{8}, 6], \quad [\mathbf{2}, \mathbf{3}, \mathbf{5}, \mathbf{6}, \mathbf{8}] .$$

Voyons maintenant l'algorithme en toute généralité. Pour cela, remplaçons l'ensemble totalement ordonné E par $\mathbf{N}$. Cette restriction n'est qu'apparente, elle ne nuit pas à la généralité du résultat.

Algorithme *TRI_INSERTION*(L)
entrer `une liste` $L[1..n]$ `de` $n \geq 2$ `entiers` $L[i]$;
sortir `les entiers` $L[i]$ `suivant une liste ordonnée croissante` $T[1..n]$.
début
1. $i = 1$;
2. $i \leftarrow i + 1$;
3. $T \leftarrow L[i]$;
4. $j \leftarrow i$;
5. `TANT QUE` $j > 1$ `&` $A[j-1] > T$:
6. `FAIRE` $L[j] \leftarrow L[j-1]$ `&` $j \leftarrow j - 1$;
7. $L[j] \leftarrow T$;

```
8. SI i < n ALLER A [2]
fin
```

Dans cet algorithme, on ne se préoccupe pas de savoir comment est représentée la liste L, ni comment elle est enregistrée ; en général, elle est donnée sous la forme d'un fichier de texte. Remarquons que dans ce pseudo-code, la ligne [5] ne se termine pas par un point-virgule. Cela vient du fait que l'instruction débute sur cette ligne et se termine à la ligne suivante. Nous avons fait un choix de présentation tout à fait personnel. Il est parfois utile, pour éviter d'éventuelle ambiguïté, d'indenter certaines instructions. Par exemple, dans le cas de l'algorithme précédent, la présentation suivante améliorera sans doute la compréhention de celui-ci :

```
Algorithme TRI_INSERTION(L)
entrer une liste L[1..n] de n entiers L[i] ;
sortir les entiers L[i] suivant une liste ordonnée croissante encore notée
L.
début
1. i = 1 ;
2. i ← i + 1 ;
3. T ← L[i] ;
4. j ← i ;
5. TANT QUE j > 1 & A[j − 1] > T :
6.     FAIRE
         L[j] ← L[j − 1]
         j ← j − 1 ;
7. L[j] ← T ;
8. SI i < n ALLER A [2]
fin
```

Le langage Pascal est encore plus explicite ; voici la procédure `Insertion_simple` qui traduit l'algorithme ci-dessus :

```
procédure Insertion_simple ;
  var
    i, j, x : integer ;
begin
  for i := 2 to n do
    begin
      t := a[i] ;
      a[0] := t ;
      j := i ;
      while t < a[j - 1] do
        begin
          a[j] := a[j - 1] ;
          j := j - 1 ;
        end ;
      a[j] := t
    end ;
end ;
```

Dans cette procédure, `a` est un tableau `a[0..n]` de `n+1` entiers (array of integers), initialement déclaré dans le programme principal, non donné ici. Les valeurs `a[1]`, `a[2]`,...,`a[n]` sont les éléments à trier. L'introduction de `a[0]` permet d'éviter que la variable `j` ne prenne des valeurs inférieures à 1. Dans le pseudo-code de l'algorithme, cette condition est implicite puisque $L[0]$ n'a pas de sens.

***Exercice* 1.17** Appliquer le tri par insertion sur la liste $L = [4, 7, 9, 2, 10, 9]$ en donnant toutes les configurations successives de L qui apparaissent au cours de l'algorithme *TRI_INSERTION*(L).

***Exercice* 1.18** Modifier l'algorithme de tri par insertion afin de trier dans l'ordre décroissant au lieu de l'ordre croissant.

***Exercice* 1.19** Soit L une liste d'entiers dont exactement k d'entre eux sont égaux. Explicitons : $L_{i_1} = \ldots = L_{i_k}$ avec $i_1 < \cdots < i_k$. On applique *TRI_INSERTION*(L) pour trier les éléments de L. Dans quel ordre (par rapport aux indices), les L_{i_s} $(1 \leq s \leq k)$ se placent-ils ?

***Exercice* 1.20** A partir de la **procédure** `Insertion_simple` ci-dessus, écrire un programme complet en Pascal, dénommé `Tri_Insertion`, qui réalise le tri par insertion de N (fixé) nombres entiers en les rangeant dans l'ordre croissant.

1.8 Exemple d'une machine abstraite

Nous décrivons ici une forme très simplifiée de la machine de Turing. Cette machine, appelée Machine-ST, est constituée d'une tête de lecture/écriture ☐ mobile et d'une bande d'enregistrement, infinie dans les deux sens, formée de cases juxtaposées. La tête se déplace de cases en cases voisines et reconnaît les symboles B (blanc) et N (noir) qui sont écrits dans certaines cases. Le contenu des cases et la position de la tête constitue une configuration.

Exemple : BBNN[B]BNNNB

La machine passe d'une configuration à une autre en réponse à la lecture de l'instruction courante, prise dans la liste (non exhaustive) suivante :

Instructions	**Action de la machine**
$\rightarrow$	déplace la tête sur la case de droite et lit son contenu ;
$\leftarrow$	comme ci-dessus mais à gauche ;
B	la tête efface le contenu de la case où elle se trouve pour y inscrire B ;
N	comme ci-dessus mais avec N au lieu de B ;
E	la tête efface le contenu de la case où elle se trouve ;
aller à n	la machine va à l'instruction n, l'exécute et de là, poursuit la lecture du programme ;
si A, aller à n	si la tête lit A (égal à B ou N), la machine va à l'instruction n, l'exécute et de là, poursuit la lecture du programme ; sinon, elle passe à l'instruction suivante.

Un programme pour la Machine-ST est une suite d'instructions qui sont lues séquentiellement par la machine. Choisissons de représenter tout entier naturel n par la configuration BN...N[B], le nombre de N étant égal à n. Cette configuration peut aussi être représentée de manière plus significative par BN^n[B].

L'algorithme suivant, appliqué à la configuration représentant un quelconque entier naturel n fait passer à la configuration de $n + 1$:

Algorithme *AJOUTER 1*
début
`1. N ;`
`2. →;`
`3. B ;`
fin

Noter que cet algorithme-programme, isolé de son contexte, est incompréhensible sans mode d'emploi donnant la signification des instructions et la machine capable de les exécuter.

***Exercice* 1.21**
a) La configuration BNNNBBNNNNN$\boxed{\text{B}}$ d'une Machine-ST représente le couple d'entiers $(3, 5)$. Donner un algorithme permettant de passer de cette configuration à celle représentant l'entier $8 = (3 + 5)$.
b) Généraliser cet exemple en définissant la configuration (d'une machine-ST) représentant un couple d'entiers positifs (m, n), puis écrire un algorithme qui fait passer ce cette configuration à celle qui représente la somme $m + n$.
c) Partant de la configuration représentant (m, n), donner un algorithme qui permet de passer à la configuration représentant (n, m).

***Exercice* 1.22** Comment multiplier par 2 avec une Machine-ST ?

***Exercice* 1.23** Écrire un algorithme pour une Machine-ST qui retranche 1.

Chapitre 2

Structurer et organiser les données

2.1 Principes de base

Dans ce chapitre, nous étudions les méthodes élémentaires d'organisation des données au moyen de structures simples et faciles à traiter du double point de vue algorithmique et informatique. Elles seront donc choisies en fonction des opérations élémentaires pouvant être effectuées sur les données, la nature de celles-ci (valeurs logiques ou numériques, éléments de listes, de tableaux ou d'ensembles plus abstraits...) ne devant pas être prise en compte. On est ainsi amené à poser le problème de la représentation des ensembles finis (structurés ou pas) et de construire des opérations de base sur ces ensembles. Les solutions adoptées devront tenir compte du fait que le nombre d'éléments d'un ensemble (de données) peut varier au cours de l'exécution d'un algorithme ou d'un programme. Nous considérerons les opérations *primitives* suivantes :

- TESTER si un ensemble est vide ;
- AJOUTER un élément à un ensemble ;
- RECHERCHER si un élément appartient à un ensemble ;
- SUPPRIMER un élément d'un ensemble.

La manipulation des ensembles par une machine doit satisfaire à deux contraintes naturelles mais souvent contradictoires, la première est d'occuper un minimum de place en mémoire et la seconde, de réaliser les opérations primitives à l'aide d'un nombre restreint d'instructions machines. Notons bien que les opérations TESTER et AJOUTER ne dépendent pas de la taille de l'ensemble. Par contre, les opérations RECHERCHER et SUPPRIMER ne peuvent pas être exécutées en un nombre d'instructions élémentaires indépendant du nombre d'éléments de l'ensemble, à moins d'en limiter leurs effets. Pour cette raison, on introduit des structures de données où les opérations de recherche et de suppression ne portent que sur le dernier ou le premier élément (structures en pile ou en file).

En fait, la manière de représenter les données en entrée et sortie est une étape importante dans la conception d'un algorithme, afin de réaliser une implémentation efficace : optimiser l'espace mémoire et les temps calculs. Il s'avère que dans de nombreuses applications, le choix d'une structure pour les données à traiter est souvent la principale difficulté à résoudre.

Les structures élémentaires que nous allons étudier sont les tables (*arrays* en anglais), les listes chaînées (*linked lists*), les piles (*stacks*) et les files d'attente (*queues*). A cela, il faut y associer le classement par type, qui a pour but de regrouper les données suivant la nature des valeurs ou des objets qu'elles représentent, ou encore suivant les opérations qui peuvent être effectuées sur elles, y compris leur mode de stockage. A côté des types primitifs standard tels que ENTIER (nombres entiers), REEL (nombres réels), BOOLEEN (valeurs *vrai* et *faux*)

et CHAR (caractères alpha-numériques), il existe un grand nombre de types dérivés tels que les tables, les fonctions, les pointeurs, les suites de caractères (STRING)... Finalement, il est toujours possible de créer un type adapté à un problème donné, notamment par simple énumération de toutes les valeurs possibles du type en question. Par exemple, le type FORME peut être défini par l'ensemble

$$F = \{ \text{ triangle, carré, rectangle, losange, cercle, ellipse, ovale}\};$$

évidemment, on veut assez de souplesse dans une telle définition pour pouvoir ajouter ou supprimer un forme.

2.2 Tables

La table (ou liste simple) est définie comme une structure primitive dans le langage Pascal et dans la plupart des autres langages de programmation. Elle est constituée d'un nombre n fixé de données (ou d'éléments, ou de composants) d'un même type, stockées de manière contiguë et accessibles par un index de sélection. L'accès au `k`-ième composant d'une table `T` se fait par `T[k]` et tous les composants de la table sont également accessibles (même temps d'accès). Le nombre n définit la taille de la table ; celle-ci est fixée et doit être connue avant usage. Le numéro d'ordre du composant `T[k]` est égal à l'index `k`, de sorte que la notion de table s'identifie à celle de n-uple en Mathématiques.

***Exemple* 2.1** L'algorithme suivant utilise d'une table binaire pour imprimer tous les entiers carrés inférieurs à N.

Algorithme *CARRES(≤ N)*
entrer `un entier N,`
sortir `les entiers carrés entre 1 et N.`
début

```
1. LIRE(N) ;
2. a=a[1..N] ;  % création d'une table de taille N %
3. POUR I=1 A N FAIRE
4.    a[I] ← 0 ; % mise à zéro de la table a %
5. I ← 1 ;
6. TANT QUE I*I ≤ N FAIRE
6.    a[I*I] ← 1 ET I ← I+1 ;
8. POUR j=1 A N FAIRE
9.    SI a[j]=1 ALORS
10.         DEBUT
11.             ECRIRE j ;
12.             NOUVELLE LIGNE ;
13.         FIN ;
```

fin

Noter les commentaires entre les balises %. C'est un moyen classique pour expliquer ou documenter certaines instructions de l'algorithme. Une convention analogue est utilisée dans les programmes. En Pascal, par exemple, les commentaires s'écrivent entre les balises (* et *), ligne par ligne.

Voici l'algorithme précédent, programmé en Pascal. La déclaration

```
P : array[1..Max] of boolean ;
```

crée une table de valeurs booléennes, de taille `Max`.

```
program squares ;
 const
  Max = 500 ;
 var
  a : array[1..N] of boolean ;
  i,j : integer ;
begin
 for j :=1 to Max do
  a[j] :=false ;
 i :=1 ;
 while i*i < Max + 1 do
  begin
   a[i*i] :=true ;
   i :=i+1 ;
  end ;
 for j :=1 to Max do
  if a[j] then
   writeln(j :4) ;
end
```

***Exemple* 2.2**
Le petit programme Pascal qui suit est un peu plus élaboré que le précédent ; il calcule les nombres premiers entre 2 et 10^4 et utilise pour cela une table `P` de 10^4 (= `Max`) données booléennes.

```
program Premier1 ;
 const
  Max = 10000 ;
 var
  P : array[1..Max] of boolean ;(* création de la table P *)
  m0, m, i, j, k, s : integer ;
  x : real ;
  mesdata : text ;
begin
 rewrite(mesdata, 'premier.dat') ;
 i := 0 ;
 m0 := trunc(sqrt(Max)) ;
 for j := 1 to Max do
  P[j] := true ;
 P[1] := false ;
 for s := 2 to m0 do
  begin
   m := trunc(Max / s) ;
   for k := 2 to m do
    P[s * k] := false ;
  end ;
 showtext ; (* instruction propre à Think Pascal sur Mac *)
 writeln(' nombres premiers entre 2 et 10000 ') ;
 writeln(mesdata, ' nombres premiers entre 2 et 10000 ') ;
 for j := 1 to Max do
  if P[j] = true then
   begin
    i := i + 1 ;
    writeln(i : 4, ' ', j : 6) ;
```

```
      writeln(mesdata, i : 5, j : 7) ;
    end ;
  close(mesdata) ;
end.
```

Ce programme commence par créer un fichier de texte sur le disque dur par l'instruction d'ouverture "`rewrite(mesdata, 'premier.dat') ;`". L'affectation `i := 0` initialise la valeur de la variable `i` qui sera utilisée pour indexer les nombres premiers. La table `P` est initialisée avec l'instruction

for `j := 1` **to** `Max` **do** `P[j] := true ;`

À cet instant du programme, tous les nombres sont considérés comme premiers. L'affectation `P[1] := false` exprime que 1 n'est pas premier, puis les affectations `P[2*k] :=false` pour $2 \leq \mathtt{k} \leq \mathtt{Max}/2$ éliminent les multiples de 2, autres que 2 lui-même, et ainsi de suite : pour $2 \leq \mathtt{s} \leq \sqrt{\mathtt{Max}}$ la valeur de `P[s]` est conservée et les affectations `P[s*k] :=false`, pour $2 \leq \mathtt{k} \leq \mathtt{Max}/\mathtt{s}$, éliminent les multiples de `s`, autre que `s`, qui sont dans le champ d'indexation de `P`. Cette méthode est assez primitive (on en verra une autre plus loin) ; elle s'apparente au classique crible d'Ératosthène. À la fin du programme, le contenu de la table `P` vérifie

$$\mathtt{P[i] = true} \Leftrightarrow \mathtt{i} \textit{ est premier.}$$

Notons au passage que la liste des nombres premiers inférieurs à 10 000 est sortie à l'écran sous la forme d'une suite de couples '$\mathtt{i,p}_i$' où $\mathtt{p}_i$ est le i-ième nombre premier ; dans le même temps, on a inscrit ces résultats dans le fichier `mesdata:text`. On disposera ainsi d'une table des nombres premiers plus petits que 10000.

La table est une structure fondamentale : elle a une assignation directe en mémoire de l'ordinateur. L'index de sélection permet d'extraire ou de modifier la donnée stockée en mémoire qui correspond au composant sélectionné.

Une autre structure familière est la notion de tableau qui peut se voir comme une table de tables ou encore une table à deux indices, l'un pour la ligne, l'autre pour la colonne. La taille d'un tableau est donnée par le produit du nombre de lignes par le nombre de colonnes. Il est possible d'étendre cette notion à des tableaux de dimension d supérieure à 2 (table de tables de tables...). Un élément quelconque d'un tel tableau `T` est un d-uple $(\mathtt{t}_1, \ldots, \mathtt{t}_d)$ accessible dans `T` par un multi-indice $i_1 \cdots i_d$; il est noté $\mathtt{T}[i_1 \cdots i_d]$.

***Exercice* 2.1** Écrire en pseudo-code la fonction qui sort le max d'une table de N nombres en utilisant la fonction *MAX(X,Y)* (cf. section 1.3). Programmer (en pseudo-code ou en Pascal) cette même fonction mais sans faire usage de *MAX(X,Y)*.

***Exercice* 2.2** **a.** Donner un algorithme qui construit une table booléenne $T[1..N]$ telle que $T[i] =$ `true` si, et seulement si, i est un cube ($i = j^3$ avec j entier).
b. Écrire un programme en Pascal, nommé `cubes`, qui imprime les cubes entre 1 et N.
c. Comparer les temps d'exécution des programmes `squares` et `cubes` pour $N = 500$, 1000 et 10 000.

***Exercice* 2.3** Écrire un programme qui met dans un tableau bidimensionnel `T` les valeurs booléennes `T[i,j]= true` si `i` et `j` sont premiers entre eux et `T[i,j]= false` sinon ($2 \leq \mathtt{i,\ j} \leq \mathtt{N}$). On conseille d'utiliser le fait que si `T[i,j]= true`, alors `T[s*i,s*j]= false` pour $i < \mathtt{s} \leq \mathtt{N/s}$.

***Exercice* 2.4** **a.** ´Écrire un algorithme qui donne dans un tableau `F` les N premières valeurs de la fréquence des entiers ≥ 1 sans facteurs carrés, *i.e.*

$$\mathtt{F}[k] = 1 - \frac{\text{card}\{\ n \in \mathbf{N}\ ;\ (1 \leq n \leq k)\ \&\ (\exists\, x \in \mathbf{N},\ x \geq 2\ \&\ x^2 | n)\ \}}{k} \quad (1 \leq k \leq N).$$

b. Implémenter cet algorithme de manière à pouvoir comparer les valeurs $\mathtt{F}[k]$ et la constante $6/\pi^2$.

`Note :` la réponse à cette question suggère que $\lim_{k\to\infty} F[k] = \frac{6}{\pi^2}$, ce qui est exact. Pour le lecteur intéressé, signalons que ce résultat peut se déduire des formules suivantes :

$$\begin{aligned}\frac{6}{\pi^2} &= \sum_{k=1}^{\infty} \frac{1}{k^2} \\ &= \lim_{K} \prod_{n=1}^{K} \left(1 - \frac{1}{{p_n}^2}\right)^{-1}\end{aligned}$$

où $(p_n)_n$ désigne la suite (croissante) des nombres premiers, et

$$\lim_{K} \prod_{n=1}^{K} \left(1 - \frac{1}{{p_n}^2}\right)^{-1} = 1 + \sum_{n=1}^{\infty} (-1)^n \frac{1}{(p_1 \cdots p_n)^2} = \lim_{k} F[k].$$

2.3 Listes Chaînées

Une *liste chaînée* (ou simplement une *liste*) est un ensemble de données (objets), organisées séquentiellement comme une table à laquelle serait associé un ordre total. Plus précisément, on peut représenter cette structure comme un ensemble totalement ordonné de *cellules*, encore appelées *nœuds* (*nodes* en anglais). Chaque donnée est contenue dans une cellule et cette dernière *pointe* sur la cellule suivante, c'est-à-dire qu'elle contient, en plus de la donnée, l'adresse de la cellule qui la suit dans l'ordre, créant ainsi un *lien* ou *pointeur*. Ce lien associe donc chaque cellule à la suivante, sauf la dernière (ou la première selon le point de vue) qui n'a pas de suivante. Par convention, il est commode, mais pas indispensable, d'introduire une cellule dite de garde, appelée *tête* (de liste), ainsi qu'une autre, appelée *queue* (de liste). La cellule tête pointe sur la première cellule de la liste, la dernière cellule pointe sur la cellule queue et cette dernière pointe sur elle-même. La figure suivante schématise une telle structure :

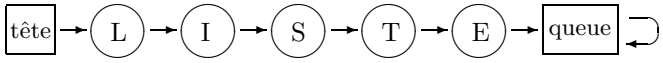


Figure 1 : schéma représentant une liste chaînée de 5 cellules avec une cellule tête et un cellule queue.

La liste (chaînée) est l'une des structures de base de la programmation. Elle est intensivement utilisée dans le langage LISP et se révèle très utile dans le calcul symbolique.

Le langage Pascal possède des primitives qui permettent d'implémenter des listes chaînées et d'effectuer sur elles les opérations de base envisagées à la section 2.1. Pour ce faire, l'utilisation des pointeurs est pratique : les cellules correspondent à des enregistrements (type **record**) et chacun d'eux est constitué d'au moins un champ qui contient la donnée et un champ réservé à l'adresse de la cellule suivante. L'adresse de la première cellule est elle-même contenue dans la cellule de tête.

La déclaration d'une cellule de liste chaînée en Pascal par la méthode décrite ci-dessus se présente donc comme suit :

```
type
    lien=↑cellule ;
    cellule=record
        contenu : objet ;
        suivant : lien ;
    end ;
var
    tete, a, queue : lien ;
```

Ici, `lien` est le nom (identificateur) d'un type pointeur, crée par la procédure pointeur "↑`type-base`"du Pascal (lorsqu'on écrit le programme, on tape ^ dans l'éditeur au lieu de ↑ qui n'est habituellement pas un symbole reconnu à la compilation). Le type-base est de type "`record`" (ou enregistrement) dénommé `cellule`. Rappelons que d'une manière générale, le type **record** en Pascal consiste en un ensemble fixé de *champs* qui peuvent être de différents types. Dans notre cas, celui-ci est constitué du champ "`contenu`" de type objet (primitif comme 'integer' ou tout autre type déjà déclaré) et du champ "`suivant`" de type pointeur. La dernière ligne crée les variables `tete`, `a` et `queue` de type pointeur, d'identificateur `lien`. On peut voir le type pointeur comme une adresse en mémoire où sont inscrites les données ; dans le cas présent, c'est le contenu de la cellule. Celle-ci étant de type record, on extrait la valeur du champ `contenu` de `a`, par `a↑.contenu` et l'adresse de la cellule suivante par `a↑.suivant`. Pascal permet de concaténer : par exemple, si `tete` est bien la tête d'une liste, `tete↑.suivant↑.contenu` fait référence au premier objet de la liste, tandis que `tete↑.suivant↑.suivant↑.contenu` fait référence au second objet de la liste.

Tout pointeur en Pascal peut prendre la valeur **nil** qui est l'adresse d'aucune cellule et peut servir pour indiquer la fin de liste. La procédure interne à Pascal **new**`(a)` permet d'affecter une adresse et un espace mémoire à l'identificateur `a` (ici, de type lien) qui seront physiquement affectés lors de la compilation pour créer une nouvelle cellule et donc pouvoir pointer sur `a`. Le rôle de cette procédure est de prendre en charge, à la place du programmeur, le travail d'allocation mémoire pour l'enregistrement de nouvelles cellules à ajouter dans une liste. La procédure `dispose(a)` libère l'adresse et l'espace mémoire primitivement réservés pour `a`.

Avec les déclarations précédentes, la création d'une liste `a` en Pascal peut commencer par la procédure créant une liste vide :

```
procédure faire_liste_vide(var a : lien) ; (*création de la liste vide*)
  begin
   a :=nil ;
  end ;
```

On teste si une liste est vide par la fonction suivante :

```
function tester_liste_vide(var a : lien) : boolean ; (*teste si a est vide*)
  begin
   liste_vide :=a = nil ;
  end ;
```

Une liste peut aussi être construite à partir de la création des cellules `tête` et `queue` qui seront déclarées comme variables à l'extérieur de la procédure :

```
procédure creer_liste ;
  begin
   new(tete) ;
   new(queue) ;
   tete↑.suivant :=queue ;
   queue↑.suivant :=queue ;
  end ;
```

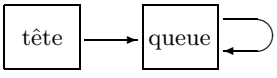


Figure 2 : Création d'une liste chaînée par une cellule tête pointant sur une cellule queue qui pointe sur elle-même. La liste ne contient pas de donnée.

La procédure *AJOUTER* insère un objet `x` dans une liste en le plaçant dans une nouvelle cellule `u`. L'affectation `u↑.contenu :=x` place `x` dans le champ `contenu` de `u`. La procédure suivante permet de construire une liste à partir de la liste vide et d'insérer les objets successifs de la liste, le dernier objet inséré étant alors en début de liste. Le second argument de la procédure est appelé par référence ; la lette `a` désigne une variable (abstraite, de type lien) : elle prend une adresse à l'entrée de la procédure (valeur de la variable `a`) ainsi qu'un objet `x` et insère une cellule `u` contenant `x` qui pointe sur `a` avant que la variable `a` ne soit affecté de la valeur de `u` :

```
procedure ajouter_liste(x : objet ; var a : lien) ;
 var
  u : lien ;
begin
 new(u) ;
 u↑.contenu := x ;
 u↑.suivant := a ;
 a :=u ;
end ;
```

Une autre méthode consiste à créer tout d'abord une cellule (`new(u)`) que l'on pointe sur la première cellule de la liste (`u↑.suivant :=tete↑.suivant`). Ensuite (ou avant), on range `x` dans `u` (`u↑.contenu :=x`) et l'on pointe la tête sur `u` (`tete↑.suivant :=u`). Cela donne la procédure suivante, où `tete` est la cellule de garde d'une liste supposée déjà existante :

```
procedure ajouter_au_debut(x : objet) ;
 var
  u : lien ;
begin
 new(u) ;
 u↑.contenu := x ;
 u↑.suivant := tete↑.suivant ;
 tete↑.suivant := u ;
end ;
```

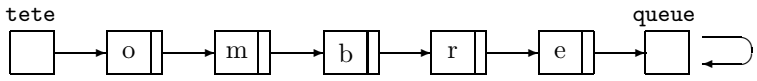


Figure 3 : liste chaînée avant d'ajouter un objet.

Chaque cellule (sauf les cellules `tete` et `queue`) est constituée de deux champs, gauche et droit. Celui de gauche contient un objet et celui de droite l'adresse de la cellule suivante. Cet adressage est matérialisé, figure 3 et 4, par une flèche. Pour ajouter l'objet " n " en début de liste, on a créé une cellule puis modifié les adresses de manière appropriée :

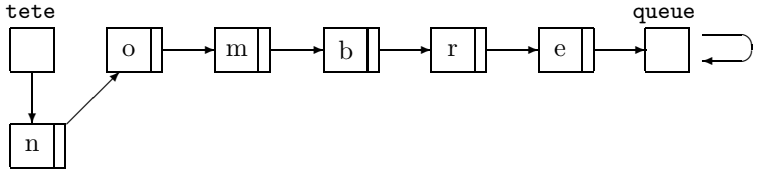


Figure 4 : liste chaînée après insertion en début de liste.

La procédure suivante réalise l'insertion de `x` dans une cellule qui fait suite à une cellule donnée `t` de la liste :

```
procedure ajouter_apres_liste(v : objet ; t : lien) ;
 var
  x : lien ;
begin
 new(x) ;
 x↑.contenu := v ;
 x↑.suivant := t↑.suivant ;
 t↑.suivant := x ;
end ;
```

La fonction *RECHERCHE* suivante effectue un parcours de la liste afin d'y trouver un objet donné `x`. La variable `a` en entrée est modifiée successivement par `a := a↑.suivant` afin de parcourir tous les objets de la liste, jusqu'à `x` ou la fin de liste, par `a=nil` (ou `a=queue` suivant la représentation adoptée).

```
function rechercher_liste(x : objet ; a : lien) : boolean ;
 var
  ok : boolean ;
begin
 ok :=false ;
 while (a<>nil) and (not ok) do
  begin
   ok := a↑.contenu=x ;
   a := a↑.suivant ;
  end ;
 rechercher_liste := ok ;
end ;
```

La procédure `supprimer_le_suivant` supprime la cellule après une cellule donnée `t` ; elle est réalisée en donnant à `t↑.suivant` la valeur `suivant` de la cellule d'après. L'introduction de la cellule de queue prévient un appel à supprimer un objet d'une liste vide :

```
procedure supprimer_le_suivant(t : lien) ;
begin
 t↑.suivant := t↑.suivant↑.suivant ;
end ;
```

Pour supprimer la cellule qui contient l'objet `x`, la valeur de `suivant` de la cellule précédente est modifiée en prenant la valeur de la cellule qui suit celle contenant `x`. Avec cette méthode, il faut habituellement tenir compte du cas où l'élément à supprimer est en début de liste.

Dans la procédure récursive de suppression ci-dessous, la fin de liste est marquée par la cellule `queue` et la fonction `dispose( )` libère la place mémoire utilisée par `x` ; celle-ci pourra donc être réutilisée par `new( )` :

```
procedure supprimer_liste(x : objet ; var t : lien) ;
(* la liste restera inchangée si x n'est pas dans une cellule de celle-ci *)
 var
  a : lien ;
begin
 if not(t = queue) then
  if t↑.contenu=x then
   begin
    a :=t ;
    t := t↑.suivant ;
    dispose(a) ;
   end
```

```
  else
   supprimer_liste(x, t↑.suivant) ;
end ;
```

Une version non récursive est évidemment possible.

Une autre convention pratique pour terminer une liste est de pointer la dernière cellule sur la première. On obtient ainsi une *liste circulaire*. Sans précautions, cette représentation peut conduire, par exemple, à un programme qui se poursuit indéfiniment à la recherche d'un objet à supprimer qui n'existe pas.

Il est possible de définir d'autres structures de liste comme par exemple d'associer à chaque cellule un double lien, l'un pour la cellule suivante, l'autre pour la cellule précédente. Cela permet de définir des opérations consistant à ajouter ou à supprimer l'objet de la cellule précédente. Évidemment, ces constructions conduisent à un surcoût mémoire, principalement dû au doublement du nombre de liens, de sorte que de telles structures ne sont pas utilisées à moins d'en avoir un usage spécifique.

Les opérations sur les listes chaînées ont de nombreuses applications ; en voici deux, très simples.

***Exemple* 2.3 Liste de nombres premiers**

Fixons un entier n et cherchons à construire la liste des nombres premiers inférieurs à n. Une solution a déjà été donnée en utilisant la structure de table (cf. exemple 2.2). La liste sera construite ici en partant d'une liste vide, puis en lui insérant des cellules en tête de liste contenant successivement les entiers n, $n-1$, $n-2$,..., jusqu'à 2, ce qui donne une liste des entiers de 2 à n. On pratique ensuite la méthode du crible d'Eratosthène en supprimant successivement les cellules contenant les multiples stricts des entiers de la liste parcourue dans l'ordre croissant. On supprime donc les cellules qui contiennent les multiple de `k=2` (à partir de 2^2, de `k=3`, à partir de 3^2, etc. tant que $\texttt{k}^2 \leq n$). Le programme utilise les procédures `creer`, `ajouter` et `supprimer` ; il n'est pas très rapide et ne fonctionne bien que pour des petites valeurs de n par manque de mémoire centrale. Mais il a l'avantage de mettre en pratique les trois procédures indiquées :

```
program premier2 ;
 const
  n = 600 ;
 type
  objet = integer ;
  lien = ↑cellule ;
  cellule = record
   contenu : objet ;
   suivant : lien ;
  end ;
 var
  j, k : integer ;
  tete, queue, a : lien ;

 procedure creer_liste ;
 begin
  new(tete) ;
  new(queue) ;
  tete↑.suivant := queue ;
  queue↑.suivant := queue ;
 end ;

 procedure ajouter_au_debut (x : objet) ;
  var
```

```
  u : lien ;
 begin
  new(u) ;
  u↑.contenu := x ;
  u↑.suivant := tete↑.suivant ;
  tete↑.suivant := u ;
 end ;

 procedure supprimer_liste (x : objet ; var t : lien) ;
  var
   b : lien ;
 begin
  if not (t = queue) then
   if t↑.contenu = x then
    begin
     b := t ;
     t := t↑.suivant ;
     dispose(b)
    end
   else
    supprimer_liste(x, t↑.suivant) ;
 end ;

(* programme principal ———————————— *)
begin
 showtext ;
 creer_liste ;
 for k := n downto 2 do
  ajouter_au_debut(k) ;
 a := tete↑.suivant ;
 while not (a = queue) do
  begin
   k := a↑.contenu ;
   for j := k to (n div k) do
    supprimer_liste(j * k, tete) ;
   a := a↑.suivant ;
   k := a↑.contenu ;
  end ;
 a := tete↑.suivant ;
 while not (a = queue) do
  begin
   writeln(a↑.contenu) ;
   a := a↑.suivant ;
  end ;
end.
```

***Exemple* 2.4 Comptine**
Le programme suivant réalise une comptine bien classique. Les participants au jeu se placent en cercle, le meneur de jeu fixe un entier $M \geq 1$ puis élimine de M en M les joueurs en les comptant l'un après l'autre, toujours dans le même sens. Le dernier éliminé (ou qui reste !) sort gagnant.

```
program comptine ;

 type
```

```
  lien = ↑cellule ;
  cellule = record
   contenu : integer ;
   suivant : lien ;
  end ;
 var
  i, N, M : integer ;
  t, x : lien ;
(* programme principal ———————— *)
begin
 writeln('nombre de joueurs N=') ;
 read(N) ; (* N joueurs identifiés par les numéros 1,2,...,N *)
 writeln('nombre de saut M=') ;
 readln(M) ;
 new(t) ;
 t↑.contenu := 1 ;
 x := t ; (* début de la création d'une liste circulaire *)
 for i := 2 to N do
  begin
   new(t↑.suivant) ;
   t := t↑.suivant ;
   t↑.contenu := i ;
  end ;
 t↑.suivant := x ; (* fin de la création de la liste circulaire *)
 while t <> t↑.suivant do
  begin          (* élimination des joueurs de M en M *)
   for i := 1 to M - 1 do
    t := t↑.suivant ;
    writeln(t↑.suivant↑.contenu) ; (* on imprime les éliminés successifs *)
    x := t↑.suivant ;
    t↑.suivant := t↑.suivant↑.suivant ;
    dispose(x) ;
  end ;
 writeln(t↑.contenu) ;
end.
```

Le programme commence par créer une liste circulaire de `N` cellules contenant les entiers de 1 à `N`. La cellule variable `x` est mise en place pour marquer le début de la liste ainsi construite, la dernière cellule de la liste pointant sur `x`. La boucle "`for i :=1 to M-1`" compte `M-1` cellules, imprime le contenu de la suivante et la supprime, et ainsi de suite jusqu'à ce qu'il ne reste q'une cellule, qui pointe alors sur elle-même, ce qui arrête la boucle lancée par `while`, puis affiche le gagnant.

***Exercice* 2.5** ´Écrire en pseudo-code les algorithmes *AJOUTER*, *SUPPRIMER*, *RECHERCHER*, *FAIRE_LISTE_VIDE*, *TESTER_LISTE_VIDE* opérant sur les listes chaînées.

***Exercice* 2.6** Implémenter en Pascal une procédure qui, pour une liste donnée, ajoute un objet avant une cellule donnée.

***Exercice* 2.7** ´Écrire une procédure permettant de supprimer :
a. la première cellule de la liste (on conserve la tête si elle est présente) ;
b. un objet dans une liste lorsque la liste utilise une cellule tête et une cellule queue ;
c. un objet dans une liste circulaire.

***Exercice* 2.8 a.** ´Écrire une procédure `suivante_au_debut(t : suivant)` qui, appliquée à une liste chaînée, place en début de liste la cellule qui suit la cellule pointée par `t`.
b. ´Écrire une procédure `permute(a, b : suivant)` pour une liste chaînée qui échange entre elles les cellules `a` et `b`.

***Exercice* 2.9** Implémenter en Pascal, pour une liste chaînée, les opérations consistant à *AJOUTER* et *SUPPRIMER* en fin de liste.

***Exercice* 2.10** ´Écrire la déclaration de type pour les cellules d'une liste chaînée avec double lien (le lien "suivant" habituel et un lien "avant" qui pointe sur la cellule précédente. Implémenter, en Pascal, les procédures *AJOUTER* et *SUPPRIMER* en fin de liste pour une telle structure de liste.

***Exercice* 2.11** Appliquer le programme `comptine` pour `N = 11` et `M = 4`. Dans quel ordre les joueurs sont-ils éliminés ?

***Exercice* 2.12** Donner un algorithme en pseudo-code ou en Pascal qui sort, dans l'ordre croissant, parmi les entiers de 1 à `N`, les pairs successifs, puis les multiples de 3 non pairs, puis les multiples de 5 non encore écrits, puis les multiples de 7 qui restent et ainsi de suite.

2.4 Listes et allocations mémoires

Les pointeurs en Pascal fournissent un outil pratique pour implémenter des listes chaînées. Il existe cependant d'autres possibilités, utilisant les tables, qui permettent aussi d'implémenter des pointeurs et des objets. Ces méthodes s'avèrent particulièrement utiles lorsque le langage de programmation ne fournit pas de solution en standard, comme par exemple pour le langage Fortran. Une méthode de substitution consiste à utiliser directement les indices de tables comme liens. L'exemple suivant montre comment enregistrer une liste chaînée en se réservant un espace mémoire suffisant au moyen d'une table servant d'allocation mémoire.

```
program liste_en_table ;
 var
  cmi :array[1..14] of char ;
begin
 for j :=1 to 18 do
  cmi[j] :=0 ;
  cmi[1] :=t ; cmi[2] :=3 ;
  cmi[3] :=c ; cmi[4] :=7 ;
  cmi[5] :=i ; cmi[6] :=9 ;
  cmi[7] :=m ; cmi[8] :=5 ;
  cmi[9] :=q ; cmi[10] :=9 ;
 end .
```

Cela donne la table suivante

1	2	3	4	5	6	7	8	9	10	11	12	13	14
t	3	c	7	i	9	m	5	q	10	0	0	0	0

Figure 5 : liste chaînée représentée par une table

où les lettres t et q indiquent respectivement la tête et la queue de liste. Ici, une cellule est représentée par une double case

2k-1	2k
donnée	*adresse*

Figure 6 : cellule d'une liste chaînée par une table

($k = 1, \ldots, 7$) ; l'objet contenu dans cette cellule est enregistré dans `cmi[2k-1]` et l'adresse de la cellule suivante est donnée par `cmi[2k]`. Enfin, par convention, 0 représente un enregistrement vide.

Une autre manière de procéder, qui s'avère souvent plus efficace, est d'utiliser des tables en parallèle. Cette méthode consiste à créer une table pour les contenus (`contenu[1..N]`) et une autre pour les liens (`suivant[1..N]`). Un avantage de ce dispositif est de définir la structure des liens indépendamment de la table des contenus. La table des liens peut donc être utilisée par une autre table de données.

Le code suivant implémente les trois opérations de base sur les listes chaînées données par table en parallèle.

```
...
var
 contenu : array[0..N] of char ;
 suivant : array[0..N] of integer ;
 new, tete, queue : integer ;

procedure debut_liste ;
begin
  tete :=0 ;
  queue :=1 ;
  new :=1 ;
  suivant[tete] := queue ;
  suivant[queue] := queue
end ;

procedure supprimer(t : integer) ;
begin
  suivant[t] := suivant[suivant[t]]
end

procedure ajouterapres(v : char ; t : integer) ;
begin
  new := new+1 ;
  contenu[new] := v ;
  suivant[new] := suivant[t] ;
  suivant[t] :=new
end ;
...
```

La variable `new` joue le rôle de la fonction d'allocation *new( )* du Pascal, à condition qu'elle prenne des valeurs d'indices non encore utilisées dans les tables. Notons que la procédure `ajouterapres(v : char ; t : integer)` insère `v` dans la liste juste après le `t`-ième élément de la table `contenu`. La figure 7 montre comment la liste de la figure 3 peut être représentée par une table double-colonne de taille 8 (N=7, 8 lignes) et donne également le résultat de l'insertion effectuée à la figure 4.

(tête)		2
(queue)		1
	0	3
	M	4
	B	5
	R	6
	E	1

(`new=6`)

	7
	1
0	3
M	4
B	5
R	6
E	1
N	2

(`new=7`)

Figure 7 : implémentation d'une liste par tables en parallèle et exemple d'insertion.

Lorsqu'on procède à une suppression, la question se pose de pouvoir disposer de l'emplacement laissé libre. La solution consiste à utiliser une liste "`libre`" qui enregistre les emplacements libres : la suppression d'une cellule de la liste est suivie par une insertion dans la liste `libre` de la cellule devenue disponible. A l'inverse, pour ajouter une nouvelle cellule à la liste, on la prend dans la liste `libre` tout en la supprimant dans celle-ci. La figure suivante reprend l'exemple précédent couplé avec une liste `libre` matérialisée par une troisième colonne ajoutée à la table double-colonne précédente

	2	7
	1	1
0	3	
M	4	
B	5	
R	6	
E	1	
		1

avant insertion

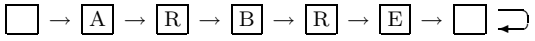
	7	1
	1	1
0	3	
M	4	
B	5	
R	6	
E	1	
N	2	(1)

après insertion

Figure 8 : liste par tables, accompagnée de sa liste `libre`, avant et après insertion et mise à jour des cellules disponibles.

Dans cet exemple, la liste `libre`, notée `L[0..7]`, se présente avant insertion comme suit : la cellule de tête `L[0]` à pour valeur 7 qui est l'adresse de la première cellule libre `L[7]` ; celle-ci contient l'adresse de la cellule queue `L[1]`, puisqu'il n'y a pas d'autres cellules libres. Après insertion, la cellule `L[7]` n'est plus libre, d'où la mise à jour de `L[0]` avec la valeur de `L[7]`. On peut alors "effacer" la valeur de `L[7]`, ou la laisser en place.

***Exercice* 2.13 a.** Implémenter en Pascal les trois opérations de base (`debut_liste, supprimer(t :integer), ajouterapres(v : char ; t : integer)`) sur les listes chaînées construites par tables en parallèle accompagnées de la liste des cellules libres.
b. Implémenter un programme qui, à partir de la liste donnée dans la figure 8, supprime O, M et insère A, R de manière à obtenir la liste



c. Donner l'état des tables après chacune des opérations précédentes.

2.5 Piles et files

Les piles (stacks) et les files (queues) sont des listes chaînées pour lesquelles l'opération *SUPPRIMER* est soumise à une restriction portant sur la manière d'accéder à l'élément qui doit être supprimé. Dans le cas d'une pile, le dernier élément ajouté est le premier à pouvoir être supprimé : elle applique le principe du *dernier entré, premier sorti* ou *LIFO* (Last-In, First-Out). Concrètement, tout se passe comme pour un empilement vertical d'objets dans une boîte d'où l'on ne peut sortir que le dernier objet placé. C'est une situation assez réaliste. Pour une file, au contraire, l'élément pouvant être supprimé est celui qui a été mis avant tous les autres : elle applique le principe du *premier entré, premier sorti* ou *FIFO* (First-In, First-Out). C'est typiquement le cas d'une file d'attente pour aller à un spectacle.

2.5.1 Piles

L'opération *AJOUTER* dans une pile est appelée *EMPILER* ; celle de *SUPPRIMER* est appelée DÉPILLER. De manière formelle, soit E un ensemble d'objets et *Pil*(E) l'ensemble des piles dont les objets contenus dans les cellules appartiennent à E. La pile vide, notée $\mathtt{P}_0$, est une pile particulière qui appartient à *Pil*(E) par définition.

Il y a de nombreuses manières d'implémenter une pile. Considérons ici le cas d'une implémentation par une table `P[1.. sommet(P)]`, où `P[1]` désigne l'élément situé à la base de la pile et `P[sommet(P)]` l'élément placé au sommet de la pile. Par définition, `sommet(`$\mathtt{P}_0$`)=0`. Les opérations de base consistant à tester si une pile est vide, à empiler un élément `x` ou à dépiler, s'implémentent facilement en pseudo-code :

PILE_VIDE(`P`)
entrer une pile P de *Pil(E)* ;
sortir "true" si la pile est vide, "false" sinon.
début
1. SI sommet(P) =0 ALORS *PILE_VIDE*(P)=true ;
2. SINON *PILE_VIDE*(P)=false ;
fin

EMPILER(`P,x`)
entrer une pile P de *Pil(E)* et un objet x de E ;
sortir la pile P après insertion de x au sommet.
début
1. sommet(P) ← sommet(P)+1 ;
2. P[sommet(P)] ← x ;
3. *EMPILER*(P) ← P[1..sommet(P)] ;
fin

DEPILER(`P`)
entrer une pile P non vide de *Pil(E)* ;
sortir la pile P après suppression de l'objet au sommet de la pile.
début
1. SI *PILE_VIDE* ALORS RETOURNER *erreur* ;
2. SINON sommet(P) ← sommet(P)-1 ;
3. *DEPILER*(P) ← P[1..sommet(P)] ;
fin

VAL(`P`)
entrer une pile P non vide de *Pil(E)* ;
sortir la valeur de P[sommet(P)].
1. SI *PILE_VIDE* ALORS RETOURNER *erreur* ;
2. SINON *VAL*(P) ← P[sommet] ;
fin

Ces opérations sur piles, représentées par tables, peuvent être réalisées en Pascal au moyen d'une structure d'enregistrement à deux champs ; le premier donne la hauteur de la pile, le second est réservé à la table qui doit contenir les objets (dans le cas présent, les objets sont des caractères). Rappelons que pour faire référence à un élément d'un enregistrement, la syntaxe est de la forme I.C où I est le nom (identificateur) de l'enregistrement et C le nom (identificateur) du champ concerné.

```
...
const
 maxP = 50 ; (* hauteur maximale choisie pour la pile *)

type
```

```
 table = array[1..maxP] of char ;
 pile = record
  hauteur : integer ;
  contenu : table ;
 end ;

procedure FairePvide (var q : pile) ;
begin
 q.hauteur := 0 ;
end ;

function Pilevide (var q : pile) : boolean ;
begin
 Pilevide := (q.hauteur = 0) ;
end ;

function Pvaleur (var q : pile) : char ;
begin
 Pvaleur := q.contenu[q.hauteur] ;
end ;

procedure Pajouter (x : char ; var q : pile) ;
begin
 q.hauteur := q.hauteur + 1 ;
 q.contenu[q.hauteur] := x ;
end ;

procedure Psupprimer (x : char ; var q : pile) ;
begin
 q.hauteur := q.hauteur - 1 ;
end ;
...
```

L'implémentation de ces opérations dépend du type de la cellule utilisée pour construire la pile. Puisqu'une pile est simplement une liste chaînée où les opérations d'ajouter, de supprimer et de lecture d'une valeur sont restreintes, la construction par pointeurs déjà utilisée, à savoir

```
type
 objet = ...
 lien = ↑cellule ;
 cellule = record
  contenu : objet ;
  suivant : lien ;
end
```

peut très bien servir. L'implémentation des opérations sur pile selon cette construction est laissée en exercice.

Nous allons maintenant montrer comment utiliser une pile dans certains algorithmes ; ceux-ci seront donnés en pseudo-code, leur traduction en langage Pascal se fera en travaux pratiques, sur machine.

2.5.2 Calculs arithmétiques

Le calcul de la valeur d'une expression algébrique sur les nombres (que nous supposerons entiers pour simplifier) utilisant les opérations d'addition et de multiplication pose le problème de définir les règles de construction de ces expressions et de déterminer un algorithme pour les évaluer. D'une manière générale, l'expression algébrique est constitué d'un assemblage de symboles opératoires $+$, $*$, de symboles de regroupement (,) ou encore [,] (associés par paire) et de nombres[1]. On distingue trois types d'expressions suivant les règles de productions utilisées.

• *Expressions intrafixées (infix).* Elles correspondent à l'écriture usuelle pour effectuer des calculs sur les nombres. Elles se construisent récursivement comme suit :

(a) un nombre est une expression intrafixée ;
(b) si α et β sont des expressions intrafixées, alors

$$(\alpha + \beta) \quad et \quad (\alpha * \beta)$$

sont des expressions intrafixées.

Par exemple, l'expression

$$2 * ((((5 * 4) + 2) * (6 * 4)) + 3)$$

qui utilise les règles classiques de parenthésage, est du type intrafixée. Il est commode de visualiser la construction de cette exemple au moyen d'un arbre (enraciné)[2] comme indiqué sur la figure 9 : les nœuds de l'arbre sont occupés par des signes d'opération et les feuilles (extrémités de l'arbre) sont occupées par des nombres. Les parenthèses n'ont pas lieu d'être.

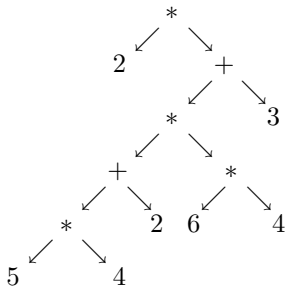


Figure 9 : l'arbre de l'expression $2 * ((((5 * 4) + 2) * (6 * 4)) + 3)$

• *Expressions postfixées (postfix).* Les règles de construction de ces expressions sont les suivantes :

(a) un nombre est une expression postfixée ;
(b) si α et β sont des expressions postfixées, alors

$$(\alpha\beta+) \quad et \quad (\alpha\beta*)$$

sont des expressions postfixées.

Contrairement aux expressions intrafixées, il n'est pas nécessaire d'utiliser des parenthèses pour les expressions postfixées. Avec cette notation, l'exemple précédent (figure 9) s'écrit

$$(\ 2\ (\ (\ (\ 5\ \ 4\ *\ )\ 2\ +\ )\ (\ 6\ \ 4\ *\ )\ *\ )\ 3\ +\ )\ *\ )$$

ou plus simplement

$$2\ \ 5\ \ 4\ *\ 2\ +\ 6\ \ 4\ *\ *\ \ 3\ +\ *$$

[1] On écarte les expressions algébriques faisant intervenir des variables où d'autres symboles représentant, par exemple, des fonctions et qui soulèvent une problématique propre au calcul formel.

[2] La structure d'arbre sera étudiée au chapitre 3. Il n'est pas utile de la connaître ici pour comprendre la signification de la figure 9.

Le calcul de cette expression se fait par lecture de gauche à droite ; la lecture du premier symbole d'opération fait opérer celui-ci sur les deux symboles qui le précèdent (et qui sont nécessairement deux nombres) mettant en évidence une sous-expression que l'on remplace par le résultat du calcul. La lecture se poursuit, jusqu'à la rencontre du prochain signe opératoire. Sur l'exemple, cela donne successivement :

$2\ \ 5\ \ 4\ *\ 2\ +\ 6\ \ 4\ *\ *\ 3\ +\ *$
$2\ \ 20\ \ 2\ +\ 6\ \ 4\ *\ *\ 3\ +\ *$
$2\ \ 22\ \ 6\ \ 4\ *\ *\ 3\ +\ *$
$2\ \ 22\ \ 24\ *\ 3\ +\ *$
$2\ \ 528\ \ 3\ +\ *$
$2\ \ 531\ *$
1062

• *Expressions préfixées (prefix).* Elles correspondent à la notation dite *polonaise* qui donne la priorité aux symboles opératoires ; leurs règles de construction sont les suivantes :

(a) un nombre est une expression préfixée ;
(b) si α et β sont des expressions préfixées, alors

$$(+\alpha\beta) \quad et \quad (*\alpha\beta)$$

sont des expressions préfixées.

Ici encore, les parenthèses peuvent être omises. Reprenons l'expression de la figure 9, en notation préfixée elle devient

$$(\ *\ 2\ (\ +\ (\ *\ (\ +\ (\ *\ 5\ \ 4\ )\ 2\ )\ (\ *\ 6\ \ 4\ )\ )\ 3\ )\ )$$

ou plus simplement :

$$*\ 2\ +\ *\ +\ *\ 5\ \ 4\ \ 2\ *\ 6\ \ 4\ \ 3$$

L'évaluation de cette expression peut se faire comme pour la notation postfixée mais en lisant 'à l'envers', c'est-à-dire de droite à gauche. L'algorithme polonais est une méthode qui consiste à lire l'expression de gauche à droite, d'insérer un à un les symboles lus dans une pile tout en effectuant les calculs en haut la de pile pour la dépiler chaque fois que cela est possible. Représentons cette empilement verticalement. À tout instant, les calculs intermédiaires étant faits, l'état de la pile est constitué de signes opératoires et de nombres mais de manière à ce qu'il n'y ait jamais deux nombres l'un au dessus de l'autre. Si, à une étape de lecture donnée, le sommet de la pile est occupé par un nombre a, qui est donc au-dessus d'un symbole opératoire noté $\odot$, alors l'insertion d'un nombre b en haut de la pile implique de la dépiler trois fois pour effectuer l'opération $(\odot\ a\ b)$ et insérer le résultat en haut de pile, puis de recommencer cette opération si on a encore deux nombres en haut de la pile. Voilà les piles successives obtenues au cours de l'évaluation de l'expression précédente par la méthode polonaise :

						5				6		
					*	*	20		*	*		
				+	+	+	+	22	22	22		
			*	*	*	*	*	*	*	*	258	
		+	+	+	+	+	+	+	+	+	+	
	2	2	2	2	2	2	2	2	2	2	2	
*	*	*	*	*	*	*	*	*	*	*	*	1062
*	2	+	*	+	*	5	4	2	*	6	4	3

Figure 10 : évaluation de l'expression usuelle $2 * ((((5 * 4) + 2) * (6 * 4)) + 3)$ à partir de son écriture en notation préfixée. La ligne du bas donne les insertions successives dues à cette notation et l'état de la pile qui en résulte après avoir effectué les calculs partiels qu'entraînent l'insertion de chaque symbole.

L'algorithme polonais. Avant de donner cet algorithme, introduisons une fonction *EVALL*(N,P) qui prend en entrée un nombre N et une pile P pour laquelle P[sommet(P)] est un nombre et P[sommet(Psupprimer(P))] (élément sous le sommet de la pile) est un symbole opératoire (+ ou ∗). En sortie, *EVALL*(N,P) vaut N + P[sommet(P)] si P[sommet(Psupprimer(P)] = + et N ∗ P[sommet(P)] si P[sommet(Psupprimer(P)] = ∗.

algorithme polonais *EVAL_PRE*(E)
entrer une expression algébrique E1..En en notation préfixée.
sortir la valeur numérique de E.
début
1. FairePvide(P) ;
2. POUR i=1 A n : ; % lire l'expression entrée au clavier
3. DEBUT
4. Pajouter(Ei,P) ;
5. TANT QUE Pvaleur(P)=NOMBRE
ET Pvaleur(Psupprimer(P))=NOMBRE
6. FAIRE
V ← EVALL(Pvaleur(P), Psupprimer(P))
P ← Pajouter(V , Psupprimer(Psupprimer(P)))
7. FIN
8. RETOURNER Pvaleur(P) ;
fin.

L'avantage de cet algorithme est que, pour n fixé, il peut calculer des expressions en notation polonaise pouvant utiliser jusqu'à $2n - 1$ termes, en comptant les signes opératoires et les nombres, distincts ou pas, qui la compose (voir l'exercice 2.20).

2.5.3 Files

Une autre structure fondamentale avec accès restreints aux données est celle de file. Par définition, une file est une liste qui comporte une cellule de tête et une cellule de queue, comme cela a été envisagé plus haut, avec deux opérations de bases possibles : ajouter en queue de file ou supprimer en tête de file (ou l'inverse, selon les préférences de l'utilisateur). Cette structure est utilisée pour gérer des objets qui sont en attente pour un usage ultérieur. L'action qui consiste à ajouter un élément (en queue de file) est appelée *ENFILER*. L'action qui consiste à supprimer un élément (en tête de file) est appelée *DEFILLER*.

L'implémentation d'une file comme liste chaînée avec les opérations ci-dessus est immédiate ; elle est laissée en exercice. Si la taille (nombre de cellules) maximale d'une file est connue ou estimée, une bonne manière de l'implémenter est d'utiliser une table F[1..n] de longueur n=longueur(F). La file F possède alors un attribut tete(F) qui indexe (ou pointe) la cellule tête de la file et un attribut queue(F) qui indexe la cellule où sera inséré un élément nouveau. Les éléments (contenus des cellules) de la file sont donc aux emplacements tete(F), tete(F)+1, ..., queue(F)-1. La table est parcourue de manière circulaire de 1 à n, puis 1 fait suite à n. La file est vide si tete(F)=queue(F). La création d'une file vide se fait par tete(F)=queue(F). Une file est peine si tete(F)=queue(F)+1-n.

Les actions enfiler et défiler sont données ci-dessous en pseudo-code ; les causes de débordement apparaissant quand on défile une file vide ou quand on enfile une file pleine ne sont pas pris en compte.

algorithme *ENFILER*(x,F)
entrer un élément x et une file F ;
sortir la file obtenue après avoir enfiler x sur F.
debut
1. F[queue(F)] ← x ;

```
2. SI queue(F)=longueur(F) ALORS queue(F) ← 1
3.     SINON queue(F) ← 1+ queue(F) ;
4. RETOURNER F ;
fin.
```

algorithme *DEFILER*(F)
entrer une file F ;
sortir l'élément de tête et la file obtenue après avoir supprimer l'élément de tête.
debut

```
1. S ← F[tete(F)] ;
2. SI tete(F)=longueur(F) ALORS tete(F) ← 1
3.     SINON tete(F)← 1+ tete(F) ;
4. RETOURNER S ;
5. RETOURNER F ;
fin.
```

La figure 11 illustre, sur un exemple de file, l'effet des opérations enfiler et défiler.

1	2	3	4	5	6	7	8	9	10
.	.	.	.	3	51	0	4	.	.
				↑				↑	
				tête				queue	

1	2	3	4	5	6	7	8	9	10
.	.	.	.	3	51	0	4	1	∗
↑				↑					
queue				tête					

1	2	3	4	5	6	7	8	9	10
.	.	.	.	(3)	51	0	4	1	∗
↑					↑				
queue					tête				

Figure 11 : En haut, exemple d'une file occupant les emplacements `F[5..9]`. Au milieu, la file qui en résulte après avoir effectué *ENFILLER*(`1,F`), *ENFILER*(`*,F`) et en bas, le résultat après *DEFILER*(`F`) qui retourne la valeur `F[5]=3`.

***Exercice* 2.14** Illustrer le résultat de chacune des opérations `EMPILER(2,P)`, `EMPILER(1,P)`, `EMPILER(a,P)`, `DEPILER(P)`, `EMPILER(23,P)`, `DEPILER(P)`, `DEPILER(P)` effectuées successivement sur une pile `P` initialement vide et représentée par une table `P[1..6]`.

***Exercice* 2.15** ´Écrire en pseudo-code la fonction *EVALL*(`E,P`) qui intervient dans l'algorithme polonais *EVAL_PRE*(`E`) (cf. section 2.5.2)

***Exercice* 2.16** Implémenter, en Pascal, les opérations de base sur une file lorsqu'on fait usage d'une table pour la représenter : plus explicitement, implémenter la procédure enfiler (que l'on désignera par `Fajouter`) et la fonction défiler (désignée par `Fvaleur`). Donner également la procédure `faireFvide` et la fonction booléenne `Fvide`.

***Exercice* 2.17** Implémenter en Pascal les opérations de base pour une file lorsqu'on fait usage d'une liste pour la représenter.

***Exercice* 2.18** Corriger les algorithmes *ENFILER* et *DEFILER* (section 2.5.3) de manière à éviter les débordements (en plus ou en moins).

***Exercice* 2.19** Définir une structure de `file à double entrée` représentée par une liste qui permet les opérations d'ajouter et de supprimer (avec retour de la valeur supprimée) aux deux extrémités de la liste (tête et queue). Implémenter les procédures et fonctions correspondantes lorsque la file est représentée par une table.

***Exercice* 2.20** Pour toute expression algébrique α en notation polonaise, on note $|\alpha|$ (resp. $|\alpha|_o$) le nombre de symboles (resp. de signes opératoires) utilisés pour l'écrire (sans parenthèses).
a. Montrer que $|\alpha| = 2|\alpha|_o + 1$ (en d'autres termes il y a un signe opératoire de moins que de nombres écrits dans α).
b. Lorsque l'expression polonaise α est évaluée par l'algorithme polonais (cf. section 2.5.2), la pile utilisée pour effectuer les calculs intermédiaires atteint une hauteur maximale qui sera notée $H(\alpha)$. Ainsi, dans l'exemple de la figure 10 cette hauteur maximale est 7. Montrer que si β est une autre expression polonaise alors $H(+\alpha\beta) = H(*\alpha\beta) = 1 + \max\{H(\alpha), H(\beta) + 1\}$. En déduire que, d'une manière générale,

$$H(\alpha) \leq \frac{|\alpha| + 1}{2}.$$

c. Déterminer, pour tout entier $n \geq 1$, une expression préfixée α_n telle que $|\alpha_n| = 2n - 1$ et $H(\alpha_n) = n$.

***Exercice* 2.21** **a.** ´Écrire un algorithme *EVAL_POST* qui utilise une pile pour évaluer les expressions algébriques en notation postfixée. Donner les états successifs de la pile pour l'exemple traité à la section 2.5.2.

b. L'exemple de la section 2.5.2 montre que les ordres dans lesquels sont écrits les entiers dans l'expression intrafixée ou dans les expressions postfixée et préfixée correspondantes sont les mêmes (seul la disposition des signes opératoires change). Montrer que c'est un résultat général.

c. Donner un algorithme qui transforme l'écriture d'une expression algébrique intrafixée en écriture postfixée correspondante (sans les parenthèses).

d. Même question qu'en c), mais pour transformer l'écriture préfixée en écriture postfixée (on pensera à lire de droite à gauche l'expression intrafixée).

Chapitre 3

Arbres

3.1 Définitions

Le chapitre précédent était consacré à l'étude des structures de type séquentiel. Nous abordons maintenant une organisation hiérarchisée dont la plus simple est celle d'*arbre*. Nous verrons que cette structure a de multiples applications.

Il existe plusieurs définitions équivalentes d'un arbre. Il peut, par exemple, être vu comme un ensemble vide ou formé d'éléments appelés *nœuds* dans lequel un des nœuds est sélectionné, prenant le nom de *racine*, les autres étant répartis en des sous-ensembles disjoints qui sont eux-mêmes des arbres, appelés sous-arbres de la racine. Chacun de ces sous-arbres se décompose en sous-arbres (éventuellement, il peut ne pas y en avoir) et ainsi de suite jusqu'à obtention de sous-arbres vides ou simplement constitués de leur racine. Cette définition est dite récursive, dans le sens où un arbre est défini à partir d'autres arbres. Une formulation rigoureuse de cette définition est possible, elle nous intéressera plus particulièrement lors de l'étude des arbres binaires (cf. section 3.5).

Choisissons maintenant une définition d'arbre qui utilise le langage des graphes non orientés. Rappelons, pour mémoire, qu'un graphe (fini) non orienté est un couple (S, A) où S est un ensemble fini dont les éléments sont appelés *sommets* et A un ensemble de paires. Les élément de A sont appelés *arêtes*. La différence essentielle entre graphe orienté (S', A') et non orienté vient de ce que A' est un ensemble de couples (*i.e.* $A' \subset S' \times S'$). Ces couples sont appelés arcs.

3.1 ***Définition.*** *Une arbre est un couple (A, B), où A est un ensemble fini et B un ensemble de paires $\{x, y\}$ d'éléments (distinct) x et y de A, qui satisfait à la propriété suivante :*

($\diamond$) pour toute paire $\{x, y\} \subset A$, il existe une suite finie unique d'éléments $x_1, \ldots, x_m$ tous distincts de A tels que $x_1 = x$, $x_m = y$ et pour $i = 1, \ldots, m-1$, la paire $\{x_i, x_{i+1}\}$ appartient à B.

L'exercice 3.12 donne plusieurs définitions équivalentes de la notion d'arbre. Un *sous-arbre* de l'arbre (A, B) est un arbre (A', B') tel que $A' \subset A$ et $B' \subset B$. Il est clair que la donnée de A' détermine un ensemble optimal B' constitué des éléments de B inclus dans A'. Mais une partie A' de A ne conduit pas nécessairement à un ensemble B' de paires tel que (A', B') soit un arbre. On définit également l'*arbre vide* : il est constitué d'aucun nœud. L'arbre vide est sous-arbre de tout arbre, on le désigne par `nil`.

Une terminologie standard accompagne cette définition. Dans toute la suite, le couple (A, B) désignera un arbre. Les éléments de A sont appelés *nœuds* et les paires $\{x, y\}$, éléments de B, sont appelées *branches* (d'extémités x et y).

L'unique suite de paires $\{x_i, x_{i+1}\}$ dans la propriété ($\diamond$) est appelée *chemin simple* (dans l'arbre) ; il est dit aller de x à y. Ce chemin est unique et en inversant l'ordre des indices,

on obtient le chemin $(\{x_{m+1-i}, x_{m-i}\})_{1\leq i\leq m-1}$ qui va de y à x. Le nombre $d(x,y)$ de paires distinctes qui composent le chemin allant de x à y (ou de y à x) est appelé *distance* entre les nœuds x et y et si $x = y$ on pose $d(x,y) = 0$. L'appellation de distance se justifie du fait que l'application $d : (x,y) \mapsto d(x,y)$ $((x,y) \in A \times A)$ vérifie bien les axiomes d'une distance au sens mathématique du terme, c'est-à-dire :

(i) $d(x,y) = d(y,x) \geq 0$;

(ii) $d(x,y) = 0 \Rightarrow x = y$;

(iii) $d(x,z) \leq d(x,y) + d(y,z)$ (inégalité du triangle).

A chaque nœud x est associé le nombre

$$I(x) := \mathrm{card}\{y \in A;\ \{x,y\} \in B\}$$

appelé indice ou degré de x.

Il est habituel de choisir un nœud r de référence, la *racine*. On dit alors que l'arbre (A,B) est enraciné (en r) et il se note – si besoin est – par le triplet (A,B,r). Remarquons que la notion de racine était déjà incluse dans la définition récursive donnée plus haut.

Le choix de la racine permet de hiérarchiser les nœuds de l'arbre en différents niveaux. Par définition, au niveau zéro se trouve la racine, au niveau 1 sont les nœuds extrémités des branches ayant la racine comme autre extrémité ; ce sont donc les nœuds à distance 1 de la racine. Plus généralement, les nœuds distants de k de la racine sont dits être au niveau k. Une conséquence directe de la propriété ($\diamond$) est que pour tout nœud x qui n'est pas la racine, s'il est au niveau k (donc $k > 0$) il existe un unique nœud a au niveau $k-1$ appelé *antécédent* ou *père* de x. Les nœuds s dont x est l'antécédent, s'ils existent, sont appelés *successeurs* ou *fils* de x ; ils sont au niveau $k+1$. Si x n'a pas de successeur il est dit nœud *extérieur* ou *feuille*, les autres nœuds sont dits *intérieurs*. Notons qu'un nœud d'indice 1 est extérieur s'il n'est pas la racine. Enfin, la racine n'est une feuille que si l'ensemble des nœuds de l'arbre se réduit à la racine.

Remarquons maintenant que tout nœud x d'un arbre enraciné (A,B,r) est racine d'un arbre unique, noté $[x] = (A[x], B[x], x)$, dont l'ensemble des nœuds $A[x]$ est constitué de x et des nœuds y aux niveaux supérieurs tels que, pour chacun d'eux, il existe un chemin allant de x à y. L'ensemble $B[x]$ des branches constitutives de ces chemins forment la partie des branches de cet arbre qui sera dit sous-arbre (enraciné) descendant de x. A noter que ce sous-arbre est fonction du choix de la racine de l'arbre (A,B).

Si y est un fils de x, on dira que $[y]$ est l'arbre fils (ou successeur) de x correspondant au fils y. Lorsque x n'a pas de fils, on conviendra qu'il possède un arbre successeur, l'arbre vide `nil`.

Le plus grand niveau où se trouve au moins un nœud de l'arbre (A,B,r) définit la *profondeur* $p(A,B,r)$ de l'arbre. On a donc

$$p(A,B,r) := \max_{x\in A} d(x,r)\,.$$

La profondeur d'un arbre renseigne guère sur sa complexité, que l'on pourrait concevoir comme la longueur minimale d'un chemin qui passerait par tous les nœuds. La notion suivante est proche de ce concept. On appelle *longueur de traversée (ou de parcours)* d'un arbre, la somme $T(A,B,r)$ des longueurs des chemins simples qui vont de chaque nœud à la racine. En d'autres termes

$$T(A,B,r) = \sum_{x\in A} d(x,r)\,. \tag{3.1}$$

Si la somme (3.1) n'est étendue qu'aux nœuds internes, on parle de longueur de traversée interne, notée $T_i(A,B,r)$. Si la somme (3.1) n'est étendue qu'aux nœuds externes, on parle de longueur de traversée externe que l'on note $T_e(A,B,r)$. On a donc

$$T = T_i + T_e\,.$$

D'autre part

$$T_e(A, B, r) = \sum_{x \in A, I(x)=1} d(x, r)\,.$$

3.2 Se donner un arbre

Il y a plusieurs manières de représenter un arbre. Lorsqu'il ne possède qu'un petit nombre de nœuds, il est pratique de le représenter par un dessin. Chaque nœud est figuré par un point, un rond ou tout autre symbole. Éventuellement, on nomme le nœud pour le repérer plus facilement. Chaque branche $\{x, y\}$ de l'arbre est représentée par le segment de droite qui joint les symboles représentant les nœuds x et y. Si l'arbre est enraciné, les nœuds sont disposés sur une succession de lignes, la racine est placée sur la première ligne qui sert de référence, les nœuds du niveau 1 sont disposés sur la ligne suivante, ceux du niveau 2 sur la ligne d'après et ainsi de suite.

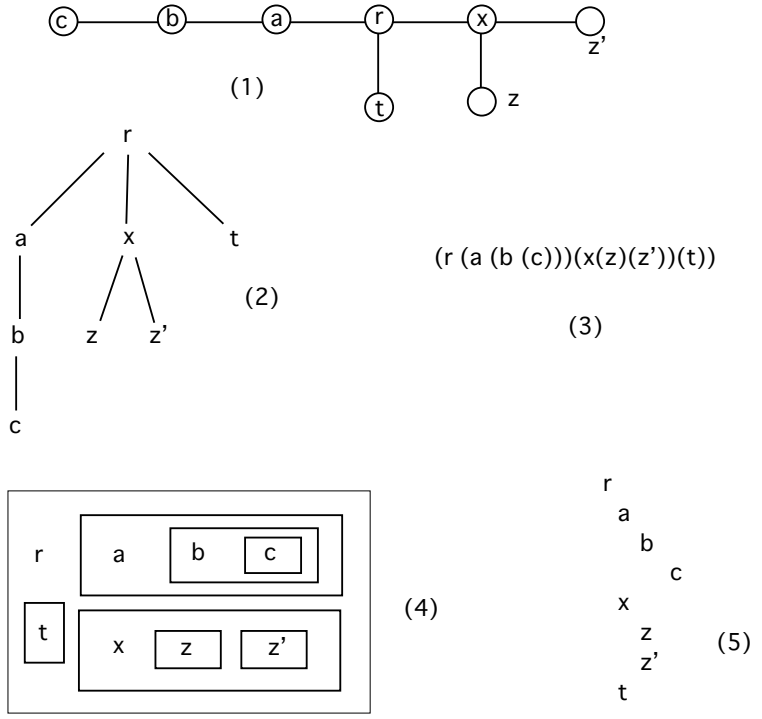


Figure 1 : représentation d'un arbre : (1) l'arbre sans racine. Le même arbre mais enraciné en r représenté par : (2) un graphe ; (3) le parenthésage ; (4) des ensembles emboîtés ; (5) l'indentation. Cet arbre est de profondeur 3 et de longueur de traversée $T = 12$.

L'arbre enraciné peut être également donné sous la forme d'un système de parenthèses emboîtées dont le modèle de base est $(r\ (A_1)\ldots(A_k))$ où r est le nœud racine et les A_i des arbres. Cette représentation signifie que pour chaque A_i de racine x_i, il existe une branche de l'arbre d'extrémités r et x_i. Elle repose aussi sur la définition récursive d'arbre. Comme les A_i peuvent être représentés à partir du modèle de base, on finit par obtenir une représentation de l'arbre par un système de parenthèses (parenthésage) qui peut devenir rapidement inextricable.

Une autre représentation, basée sur le dessin et la définition récursive d'arbre donnée au début, consiste à délimiter tout d'abord l'ensemble des nœuds de l'arbre par un contour

(cercle, rectangle, . . .) ; puis on fait de même avec les sous-arbres qui partitionnent l'ensemble des nœuds de l'arbre autre que sa racine ; enfin, on procède de même pour tous les sous-arbres successifs. On obtient ainsi une représentation au moyen d'un système emboîté d'ensembles.

Donnons une dernière méthode pour représenter un arbre enraciné. Elle s'accommode assez bien d'une sortie imprimante en mode texte. Il s'agit de la présentation par indentation : la racine est inscrite sur la première colonne de la première ligne, puis, sur les colonnes successives sont inscrits les nœuds du niveau 1, puis 2, jusqu'aux nœuds de niveau p. Un nœud étant placé, ses fils sont inscrits dans la colonne suivante, à droite, et sur des lignes inférieures. Le (5) de la figure 1 donne un exemple de cette représentation.

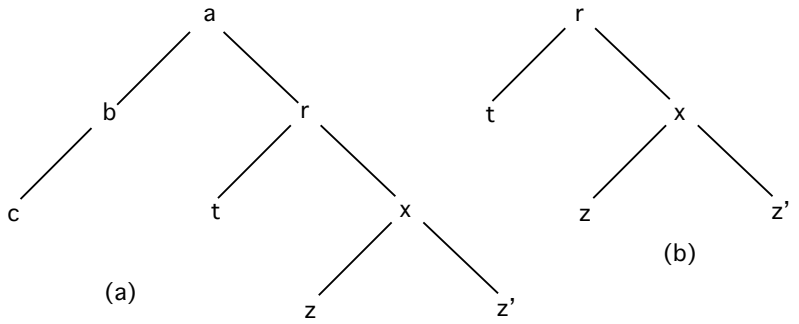


Figure 2 : représentation d'un arbre (suite). En (a) : l'arbre de la figure 1 mais enraciné en a. En (b) : le sous-arbre correspondant au nœud r, très différent de l'arbre en (2), fig. 1.

***Exemple* 3.1 Arbre d'une expression algébrique**

On a déjà vu à la section 2.5.2 qu'une expression algébrique avec les opérations usuelles d'addition, de soustraction, de multiplication et de division, peut se représenter sous la forme d'un arbre dont les nœuds intérieurs sont étiquetés par les symboles opératoires, les feuilles correspondant à des valeurs numériques ou des variables.

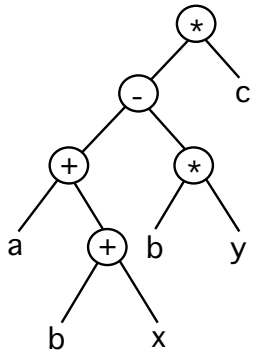


Figure 3 : arbre d'une expression algébrique.

Une méthode pour évaluer l'expression consiste à passer successivement du plus grand niveau au niveau inférieur, ramenant le calcul à un arbre de profondeur moindre. Ce passage s'obtient en effectuant l'opération indiquée sur le nœud intérieur qui sont pères des feuilles ; elle correspond à un groupement par parenthèses dans l'expression algébrique. En itérant cette réduction, on arrive jusqu'à la racine avec la valeur de l'expression. La figure 3 montre

l'arbre associé à l'expression algébrique $((a+(b+x))-b*y)*c$. La figure suivante représente les étapes successives de l'évaluation de l'expression lorsque $a = b = 1$, $x = 0$ et $y = -1$

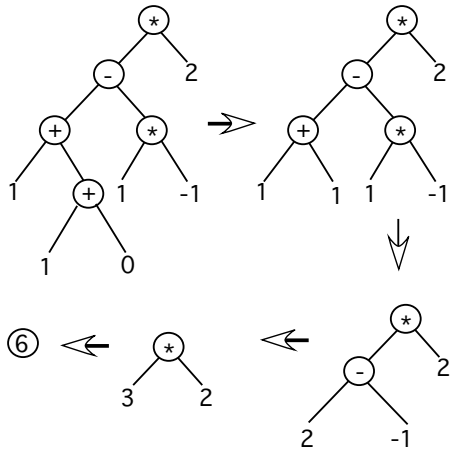


Figure 3.bis : les étapes successive d'une évaluation numérique de la formule $((a + (b + x)) - b * y) * c$.

3.3 Représentation matricielle d'un arbre

Un arbre (A, B) est évidemment connu dès que l'ensemble de ses nœuds A et de ses branches B sont donnés. Pour un arbre fixé, une question est de déterminer la représentation la plus adéquate. Dans le cas de l'ensemble A des nœuds, le plus classique est d'indexer A par des entiers, d'où $A = \{a_1, \ldots, a_s\}$ (donc $s = \text{card}(A)$), ce qui conduit souvent à identifier A avec l'ensemble $\{1, \ldots, s\}$. Une manière équivalente d'indexer A est d'ordonner A par un ordre total. L'ensemble B des branches peut être donné sous plusieurs formes, en fonction de considérations théoriques, algorithmiques ou informatiques. Tout d'abord, B peut être décrit tout simplement en extension (liste exhaustive des éléments de B) ou en compréhension (l'ensemble est défini par une propriété caractéristique) ou de bien d'autres manières. La méthode qui nous intéresse ici fait intervenir la notion de matrice d'*adjacence* (ou encore d'*incidence*) b du graphe non orienté constitué par l'arbre. Indexons A *i.e.* $A = \{a_1, \ldots, a_s\}$; par définition, b est une matrice carrée d'ordre s de composantes $a_{i,j}$ (i-ième ligne, j-ième colonne, $1 \leq i, j \leq s$) définies par

$$a_{i,j} = \begin{cases} 1 & \text{si } \{a_i, a_j\} \in B\,; \\ 0 & \text{sinon.} \end{cases}$$

Par définition d'un graphe non orienté, la matrice b est symétrique, la diagonale principale ne contient que des zéros et si le graphe est un arbre, aucune colonne (resp. ligne) n'est nulle ; mais la propriété ($\diamond$) ne se lit pas immédiatement sur la matrice b. Nous supposons maintenant l'arbre enraciné, ce qui induit une orientation des branches, le *sens positif* sera choisi comme étant celui qui va du père au fils. Le choix de la racine ne peut pas se lire dans la matrice b ; mais on peut convenir que la première ligne de b correspond à cette racine. L'orientation positive de l'arbre enraciné conduit naturellement à remplacer la matrice d'adjacence par la matrice b' des filiations définie par

$$(b')_{i,j} = \begin{cases} 1 & \text{si } a_i \text{ est fils de } a_j \\ -1 & \text{si } a_i \text{ est père de } a_j \\ 0 & \text{si } \{a_i, a_j\} \notin B. \end{cases}$$

Cette matrice est évidement antisymétrique (*i.e.* elle est égale à l'opposé de sa transposée) ; de plus :

– *chaque colonne (resp. ligne) a une composante égale à* -1 *(resp.* 1*) et une seule ;*
– *la racine correspond à l'unique colonne de* b' *qui n'a pas de* -1 *(ou l'unique ligne qui n'a pas de* 1*) ;*
– *les feuilles sont caractérisées par les colonnes (resp. lignes) de composantes toutes nulles à l'exception d'une seule qui vaut* -1 *(resp.* $+1$*) ;*

***Exemple* 3.2** L'arbre (4) de la figure 2, en indexant les nœuds suivant l'ordre alphabétique, donne la matrice de filiation suivante :

$$\begin{bmatrix} 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Nous allons montrer qu'il est possible de choisir une indexation de manière à ce que toutes les composantes égales à -1 soient situées au-dessus de la diagonale.

3.2 *Théorème.* *Pour tout arbre non vide* (A, B)*, il existe une indexation* $\{a_i, 1 \leq i \leq \mathrm{card}(A)\}$ *de* A *telle que la matrice des filiations* b' *vérifie :*

$$(b')_{i,j} = -1 \;\Rightarrow\; i < j.$$

En particulier, a_1 *est la racine.*

Démonstration. Pour $\mathrm{card}(A) = 1$, la matrice de filiation est nulle et le résultat est banal. Pour $\mathrm{card}(A) = 2$, la matrice de filiation est l'une des matrices suivantes :

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \qquad \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}.$$

Il suffit donc que le nœud a_1 soit la racine.

Supposons le théorème vrai pour tout arbre enraciné ayant au plus n nœuds. Soit un arbre enraciné (A, B, r) de $n+1$ nœuds et $x_1, \ldots, x_k$ les fils de la racine qui est supposée être indexée par 1 ($r = a_1$). Notons n_i le nombre de nœuds de l'arbre fils $[x_i]$. Puisque $n_i \leq n$, par hypothèse de récurrence, il existe une indexation $(x_{i,k})_{1 \leq k \leq n_i}$ des nœuds de $[x_i]$ telle que la matrice de filiation b'_i correspondante ait toutes ses composantes égales à -1 situées au dessus de la diagonale. En juxtaposant ces indexations on obtient immédiatement une indexation de A. Par exemple, pour $k = 2$ cela donne :

$$a_1 = r, a_2 = x_1 = x_{1,1}, \ldots, a_{1+n_1} = x_{1,n_1}, a_{2+n_1} = x_2 = x_{2,1}, \ldots, a_{1+n_1+n_2} = x_{2,n_2}.$$

Dans le cas général, $a_1 = r$, $a_{1+i} = x_{1,i}$ pour $1 \leq i \leq n_1$ et $a_{1+n_1+\cdots+n_{j-1}+i} = x_{j,i}$ pour $1 < j \leq k$ et $1 \leq i \leq n_j$. La matrice de filiation pour cette indexation satisfait alors aux conclusions du théorème. Explicitons-là pour $k = 3$:

$$\begin{bmatrix}
0 & -1 & 0 & \dots & 0 & -1 & 0 & \dots & 0 & -1 & 0 & \dots & 0 \\
1 & \multicolumn{4}{c}{\multirow{3}{*}{b'_1}} & 0 & \dots & \dots & 0 & 0 & \dots & \dots & 0 \\
0 & & & & & \vdots & & & \vdots & \vdots & & & \vdots \\
\vdots & & & & & \vdots & & & \vdots & \vdots & & & \vdots \\
0 & & & & & 0 & \dots & \dots & 0 & 0 & \dots & \dots & 0 \\
1 & 0 & \dots & \dots & 0 & \multicolumn{4}{c}{\multirow{3}{*}{b'_2}} & 0 & \dots & \dots & 0 \\
0 & \vdots & & & \vdots & & & & & \vdots & & & \vdots \\
\vdots & \vdots & & & \vdots & & & & & \vdots & & & \vdots \\
0 & 0 & \dots & \dots & 0 & & & & & 0 & \dots & \dots & 0 \\
1 & 0 & \dots & \dots & 0 & 0 & \dots & \dots & 0 & \multicolumn{4}{c}{\multirow{3}{*}{b'_3}} \\
0 & \vdots & & & \vdots & \vdots & & & \vdots & & & & \\
\vdots & \vdots & & & \vdots & \vdots & & & \vdots & & & & \\
0 & 0 & \dots & \dots & 0 & 0 & \dots & \dots & 0 & & & &
\end{bmatrix}$$

■

Considérons maintenant la matrice f des fils qui se déduit de la matrice des filiations en remplaçant les -1 par des 0. On a ainsi $f_{i,j} = 1$ si a_i est fils de a_j et $f_{i,j} = 0$ dans les autres cas. L'intérêt d'introduire f est que pour tout entier k, f^k est la matrice des chemins de longueur k qui vont de pères en fils. Toutes ces matrices ont leurs composantes égales à 0 ou 1. De plus, si p est la profondeur de l'arbre, on a $f^p \neq 0$ et $f^{p+1} = 0$.

***Exercice* 3.1** Calculer la matrice d'adjacence b et la matrice des fils f de l'arbre (3) à la figure 1.

***Exercice* 3.2** On suppose que pour une indexation $\{a_1, \dots, a_n\}$ des nœuds d'un arbre enraciné, la matrice des fils f est triangulaire inférieur (*i.e.* $f_{i,j} = 0$ si $i \leq j$). Montrer que les nœuds a_k qui appartiennent au sous-arbre de racine a_i vérifie $i \geq k$.

***Exercice* 3.3** Soit b la matrice d'adjacence d'un arbre de sommets indexés $a_1, \dots, a_n$ avec a_1 comme racine.
a. Pour quelle raison a-t-on $(b^2)_{i,i} = 1$ pour tout indice i ?
b. Montrer que pour couple d'indice (i, j), on a $b_{i,j}(b^2)_{i,j} = 0$.
c. Que représente les composantes $(b^n)_{i,j}$ en terme de chemins dans l'arbre ? Même question pour $b^2 - b$.
d. On effectue une nouvelle indexation des nœuds qui consiste à échanger deux nœuds entre eux (par exemple a_2 devient a_3 et a_3 devient a_2). Comment passe-t-on de l'ancienne à la nouvelle matrice d'adjacence ?

***Exercice* 3.4** Soit b' la matrice des filiations d'un arbre ayant n nœuds. Sur l'exemple 3.2, on peut observer qu'il y a autant de 1 que de -1 sur la ligne diagonale des $b'_{i,j}$ tels que $i + j = c$, (c constante choisie entre 2 et $2n$). Montrer que c'est un résultat général.

***Exercice* 3.5** Soit f la matrice des fils d'un arbre donné α.
a. Montrer que la racine de α correspond à une ligne nulle dans f (il n'y en a donc qu'une).
b. Comment reconnaître dans la matrice f les feuilles de l'arbre ?
c. Montrer que si p est la profondeur de l'arbre, $I + f + \cdots + f^p$ est la matrice triangulaire inférieur t dont toutes les composantes sous la diagonale sont égales à 1 (*i.e.* $t_{i,j} = 1$ si $i \geq j$ et $t_{i,j} = 0$ dans les autres cas).
d. Montrer comment reconstruire, à partir de f, la matrice des filiations b' et la matrice d'adjacence associées à l'arbre.

***Exercice* 3.6** Soit b la matrice d'adjacence d'un arbre où la première ligne correspond à la racine. Donner un algorithme qui permet de calculer la matrice des filiations b' en partant de b.

3.4 Propriétés combinatoires

Revenons sur la notion générale de graphe non orienté et soit $G = (S, B)$ l'un d'entre eux. Rappelons qu'un *chemin* entre deux sommets distincts x et y de G, de longueur n, est une suite d'arêtes $(\{x_i, y_i\})_{1 \leq i \leq n}$ telle que $x_1 = x$, $y_n = y$ et $y_i = x_{i+1}$ pour $1 \leq i \leq n-1$. Pour $n = 0$ la suite est vide et correspond au chemin *vide*. On peut évidemment se donner le chemin plus simplement en ne considérant que la suite $(x_i)_{1 \leq i \leq n+1}$ de sommets, avec $x_{n+1} = y$. Un cycle est un chemin non vide entre deux sommets identiques. Le chemin est dit *simple* si chaque côté constitutif du chemin n'apparaît qu'une fois. Le graphe G est dit *connexe* si pour toute paire de sommets distincts, il existe un chemin d'extémités ces deux sommets. On peut maintenant énoncer une propriété qui caractérise les arbres parmi les graphes non orientés :

3.3 *Théorème.* *Un graphe non orienté est un arbre si, et seulement si, il est connexe et sans cycle.*

Démonstration. Soit un graphe G connexe sans cycle et vérifions la propriété $(\diamond)$. Soient x et y deux sommets distincts. Par hypothèse de connexité, il existe un chemin allant de x à y que l'on peut supposer être simple (*i.e.* ne contenant par d'arêtes en double). S'il en existe un autre, en faisant suivre le premier par celui qui va de y à x, on crée un cycle, ce qui est contraire à l'hypothèse faite sur G.

Réciproquement, soit un arbre (A, B) ; en tant que graphe, la propriété $(\diamond)$ implique qu'il est connexe. De plus, s'il admet un cycle, on en déduit immédiatement une contradiction avec $(\diamond)$ en exhibant deux nœuds joints par deux chemins simples distincts. ■

Certaines propriétés des arbres sont utiles du point de vue algorithmique. La suivante est une généralisation du bon vieux problème des intervalles, rencontré en classes primaires :

3.4 *Proposition.* *Un arbre de n nœuds ($n \geq 1$) possède exactement $n - 1$ branches.*

Démonstration. La propriété à démontrer est indépendante du choix d'une racine. Remarquons qu'après ce choix, tout nœud, sauf la racine, admet un seul antécédent et que pour toute branche, l'une de ses extrémités à pour antécédent l'autre extrémité. Il en résulte une application bijective entre les nœuds distincts de la racine et les branches de l'arbre. Cette bijection associe un nœud à la branche qu'il forme avec son antécédent. ■

***Exercice* 3.7** Dessiner tous les graphes formés de trois sommets. Même question avec les arbres et les arbres enracinés.

***Exercice* 3.8** Soit (S, A) un graphe non orienté et $I(\cdot)$ l'application indice définie sur S par $I(x) := \mathrm{card}(\{s \in S;\ \{x, s\} \in A\}$.
a. Démontrer l'égalité

$$\sum_{s \in S} I(s) = 2\,\mathrm{card}(S)\,.$$

b. Établir l'inégalité

$$\mathrm{card}(S) \leq \mathrm{card}(A) + 1\,.$$

***Exercice* 3.9** Un graphe non orienté sans cycle (S, A) est appelé une forêt. Soit $\sim$ la relation binaire sur l'ensemble S des sommets définie par $x \sim y$ si, et seulement si, il existe un chemin allant de x à y. Montrer que $\sim$ est une relation d'équivalence dont les classes déterminent des sous-graphes de (S, A) qui sont des arbres.

3.5 Arbres ordonnés et arbres binaires

Dans un arbre enraciné, lorsque pour chaque nœud l'ensemble de ses fils est muni d'un ordre total, on dit que l'arbre est ordonné. Une classe importante d'arbres qui admette une structure d'ordre total naturel est celle où la racine est d'indice au plus 2 et les autres nœuds d'indice au plus 3. Une définition équivalente plus parlante est donnée en termes de successeurs :

3.5 ***Définition.*** *Un arbre est dit binaire si tout nœud admet deux arbres fils, gauche et droit, un des arbres ou les deux pouvant être vides.*

Lorsque tous les nœuds admettent exactement soit deux, soit zéro successeurs, on dit que l'arbre binaire est *plein*.

Étant donné un arbre binaire, définition, tout nœud intérieur x à exactement un ou deux successeurs que l'on désignera par successeur à gauche (fils gauche) noté $\texttt{filsG}(x)$ et un successeur à droite (fils droit) noté $\texttt{filsD}(x)$. L'un de ces successeurs peut ne pas exister mais dans ce cas, on continue de distinguer, pour le nœud qui reste, s'il est à droite ou à gauche. Lorsque les deux successeurs sont présents, celui de gauche est déclaré plus petit que celui de droite. De même, l'arbre fils du successeur gauche (resp. droit) d'un nœud x quelconque est appelé arbre fils gauche (resp. droit) et sera noté par $\texttt{filsG}[x]$ (resp. $\texttt{filsD}[x]$) ; ces arbres peuvent être vides. On associe également à tout nœud x le nœud père noté $\texttt{père}(x)$ ou plus simplement $\texttt{p}(x)$. Si x est la racine, $\texttt{p}(x)$ est vide (il prend la valeur `nil`).

Ordonnons maintenant l'arbre que l'on suppose non vide et de racine r. S'il n'est formé que de sa racine, l'ordre est banal. Sinon, prenons deux nœuds x et y, et définissons la relation $x \mathcal{A} y$ (lire x *avant* y si l'une des situations est réalisée : $x = y$, $x \in \texttt{filsG}(y)$, $y \in \texttt{filsD}(x)$ ou il existe un noeud z tel que simultanément $x \in \texttt{filsG}(z)$ et $y \in \texttt{filsD}(z)$. Il est facile de vérifier que $\mathcal{A}$ est une relation d'ordre total.

Les arbres binaires peuvent se définir récursivement de la manière suivante :

– *l'arbre vide est binaire.*
– *Tout arbre binaire non vide est enraciné, avec exactement deux sous-arbres de la racine, un à droite et un à gauche, et ce sont des arbres binaires.*

Avec ces définitions, l'échange entre les arbres fils d'un même nœud détermine un autre arbre et cela vaut aussi bien pour les nœuds ayant deux fils non vides que pour ceux ayant un arbre fils vide. Ainsi, la donnée d'un arbre binaire doit bien distinguer l'arbre fils gauche de l'arbre fils droit. On décrira donc un arbre binaire selon le schéma récursif `(racine, arbre_fils_gauche, arbre_fils_droit)`. Les arbres (a, β, γ) et (a, γ, β) sont donc distincts si β et γ sont deux arbres distincts. En fait, ils ont le même ensemble de sommets, mais les ordres qu'ils induisent sur ces sommets sont distincts. En particulier les arbres $(a, \beta, \texttt{nil})$ et $(a, \texttt{nil}, \beta)$ sont distincts si β n'est pas vide (dans le premier cas, a est le plus grand sommet, dans le second, il est le plus petit). Dans ces notations, les arbres β et γ seront identifiés à leur racine s'il se réduisent à un seul nœud.

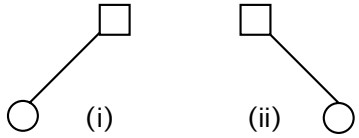


Figure 4 : exemples d'arbres identiques qui, en tant qu'arbres binaires, sont distincts : en (i), la racine a son arbre fils droit vide, en (ii) la racine a son arbre fil gauche vide.

Chemins dans un arbre binaire. Pour tout nœud x d'un arbre enraciné (A, B, r), on sait qu'il existe un seul chemin simple $C(x) := (\{c_i, c_{i+1}\})_{1 \leq i \leq m}$ allant de la racine ($c_1 = r$) à x ($c_{m+1} = x$). Si $x = r$, ce chemin est vide.

Supposons l'arbre binaire et introduisons l'ensemble $\{g, d\}^* = \cup_{m \geq 0} \{g, d\}^m$. Un élément de $\{g, d\}^m$ est un m-uple $w = (w_1, \ldots, w_m)$ dont les composantes w_i sont égales à g ou d. On écrira aussi $w = w_1 \cdots w_m$ ce qui conduit à considérer w comme un mot et l'ensemble $\{g, d\}$ comme un alphabet. Par définition, l'ensemble $\{g, d\}^0$ se réduit à un seul élément, noté $\wedge$, qui correspond à une suite vide encore appelé *mot vide*. Définissons maintenant l'application $W : A \to \{g, d\}^*$ qui associe à x le mot $W(x) = w_1 \cdots w_m$ défini à partir du chemin $C(x)$ en posant :

$$w_i = \begin{cases} g & \text{si } c_{i+1} \text{ est fils gauche de } c_i, \\ d & \text{sinon.} \end{cases}$$

Remarquons que x est déterminé par la donnée du mot $W(x)$, c'est-à-dire que l'application $W(\cdot)$ est injective. En fait, partant de la racine $c_1 = r$, la valeur (g ou d) de w_1 détermine c_2 et ainsi de suite : en c_i la valeur de w_i permet de déterminer c_{i+1}. Le mot $W(x)$ sera appelé *étiquette* du chemin $C(x)$. Il y a donc une correspondance biunivoque entre les chemins simples sur l'arbre issus de la racine et leurs étiquettes respectives. Notons que le chemin vide qui, par convention, va de la racine à elle-même, a pour étiquette le mot vide.

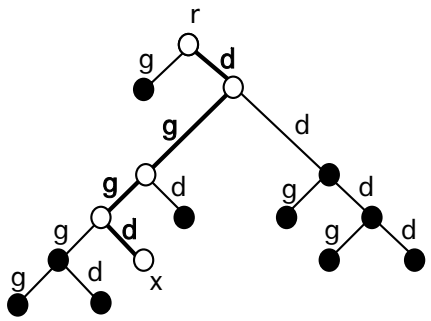


Figure 5 : dans cet arbre binaire, le nœud x et le chemin allant de la racine r à x sont codés par le mot **W**(x)=**dggd**.

On peut déterminer la relation d'ordre $\mathcal{A}$ avec la représentation par les mots (cf. exercice 3.16).

Un arbre binaire est dit *complet* s'il est plein avec toutes ses feuilles au même niveau. Si p est ce niveau (qui est aussi la profondeur de l'arbre), on obtient facilement que le nombre de nœuds est égal à $2^{p+1} - 1$. En résumé :

3.6 *Proposition.* *Si p est la profondeur d'un arbre binaire complet, le nombre de ses nœuds est $2^{p+1} - 1$.*

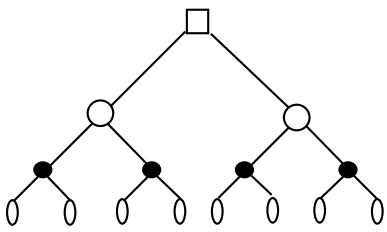


Figure 6 : arbre binaire complet de profondeur 3.

Les deux propriétés suivantes concernent spécifiquement les arbres binaires pleins.

3.7 *Proposition.* *Un arbre binaire plein de n nœuds intérieurs possède exactement $n+1$ feuilles.*

Démonstration. Par récurrence sur le nombre de nœuds. Si $n = 0$, l'arbre se réduit à sa racine qui est une feuille. Si $n = 1$, c'est que la racine est l'unique nœud intérieur et possède donc deux fils qui sont des feuilles. Supposons maintenant la propriété vraie pour tout arbre binaire plein de n nœuds intérieurs et soit un arbre binaire plein de $n+1$ nœuds intérieurs. Choisissons une de ses feuilles et le nœud dont elle est le successeur. Supprimons les deux fils de ce nœud (lequel devient une feuille) et les deux branches qui vont avec. Il reste un arbre ayant n nœuds intérieurs. Par hypothèse de récurrence, cet arbre possède $n+1$ feuilles dont l'une d'elle est le nœud dont on a supprimé les deux feuilles. Le retour à l'arbre initial montre donc que celui-ci a $n+1-1+2 = n+2$ feuilles.

Autre méthode de démonstration : en supprimant la racine, on met en évidence deux sous-arbres binaires pleins sur lesquels on applique l'hypothèse de récurrence. ■

3.8 *Proposition.* *Pour tout arbre binaire plein de n nœuds intérieurs, on a*

$$T_e = T_i + 2n \tag{3.2}$$

où T_e est la longueur de traversée externe de l'arbre et T_i sa longueur de traversée interne.

Démonstration. Le résultat est clair pour $n = 0$ ou $n = 1$. S'il est établi pour n nœuds intérieurs, examinons le cas d'un arbre binaire α ayant $n+1$ nœuds intérieurs. Soit x un de ces nœuds à distance k maximale de la racine. Supprimons-le avec ses deux feuilles, comme dans la démonstration précédente. On obtient une arbre binaire β pour lequel la formule (3.2) est établie. Le retour à l'arbre initial montre que $T_i(\alpha) = T_i(\beta) + k$ tandis que $T_e(\alpha) = T_e(\beta) + k + 2$ puisque la longueur k du chemin allant de la racine à x est déjà comptée une fois dans $T_e(\beta)$. On a dont $T_e(\alpha) = (T_i(\beta) + 2n) + k + 2 = (T_i(\alpha) + 2n) + 2$, soit (3.2) avec $n+1$ au lieu de n. ■

***Exercice* 3.10** Démontrer la proposition 3.6.

***Exercice* 3.11** Peut-on reconnaître un arbre binaire directement à partir de sa matrice des fils ?

***Exercice* 3.12** Soit $G = (S, A)$ un graphe non orienté. Démontrer l'équivalence des propriétés suivantes :

(i) G est un arbre ;
(ii) G est connexe mais si on ôte un de ses sommets (et les arêtes dont une extrémité est ce sommet), on obtient un graphe qui n'est plus connexe ;
(iii) G est connexe et $\mathrm{card}(S) = \mathrm{card}(A) + 1$;
(iv) G est sans cycle et $\mathrm{card}(S) = \mathrm{card}(A) + 1$;
(v) G est sans cycle, mais si une arête est ajoutée à A, le graphe résultant possède un cycle.

***Exercice* 3.13** Démontrer la proposition 3.8 par récurrence en considérant les deux arbres associés à la racine.

***Exercice* 3.14** Un arbre binaire de profondeur p est dit semi-complet si les feuilles n'apparaissent qu'aux niveaux p ou $p-1$. Montrer que si n est le nombre de nœuds internes d'un tel arbre, on a

$$2^{p-1} < n + 1 \leq 2^p.$$

Dans quel cas a-t-on l'égalité $n + 1 = 2^p$?

***Exercice* 3.15** Montrer qu'un arbre binaire de n nœuds à un profondeur $\geq \lfloor \log_2 n \rfloor$.

***Exercice* 3.16** **a.** Démontrer l'équivalence entre la définition 3.5 d'arbre binaire et la définition récursive. **b.** Montrer que la conclusions de la proposition 3.7 pour la famille des arbres binaires pleins est fausse pour les arbres binaires en général, mais que la formule suivante

$$N_2 - N_0 + 1 = 0\,,$$

où N_i désigne le nombre de nœuds ayant exactement i descendants, est vraie.
c. Soit W l'application qui à tout noeud x d'un arbre binaire donné associe le mot $W(x) \in \{g, d\}^*$ défini à partir du chemin $C(x)$ allant de la racine r de l'arbre à x (cf. section 3.5). On definit sur les sommets de l'arbre la relation $\mathcal{P}$ (lire *précède*) par $x \mathcal{P} y$ si l'une des situations suivantes se présente :

– $x = y$;
– $W(x)$ commence par g et $y = r$ (donc $W(y)$ est le mot vide) ;
– $W(y)$ commence par d et $x = r$ (donc $W(x)$ est le mot vide) ;
– $W(x)$ est plus grand que $W(y)$ au sens usuel de l'ordre lexicographique sur les mots (la lettre d venant avant la lettre g).

Montrer que $\mathcal{P}$ est une relation d'ordre qui coincide avec la relation $\mathcal{A}$ donnée à la section 3.5.

3.6 Implémentation d'un arbre binaire

La manière la plus directe pour implémenter un arbre binaire est de représenter chaque nœud par un champ d'enregistrement mémoire dans lequel est inscrit les données associées au nœud ainsi que deux pointeurs, un pointeur d donnant l'adresse du successeur droit du nœud et un pointeur g donnant l'adresse du successeur gauche. On peut ajouter éventuellement un pointeur sur l'adresse du père. Lorsqu'il n'y a pas de successeur droit ou gauche (ou de père), le pointeur prend la valeur `nil` ; c'est notamment le cas pour les feuilles. La figure 8 ci-dessous schématise ce type d'implémentation par pointeurs et concerne l'arbre binaire représenté à la figure 7.

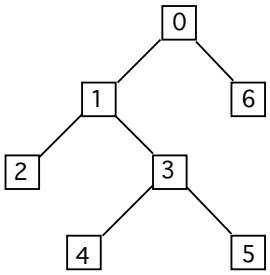


Figure 7 : un arbre binaire.

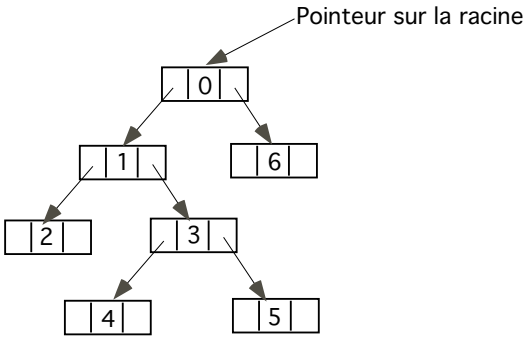


Figure 8 : schéma d'une implémentation de l'arbre de la figure 7 par pointeurs “à droite” et “à gauche”.

Pour certaines applications, il est souhaitable d'avoir deux types d'enregistrement des nœuds, l'un pour les internes, l'autre pour les externes. Dans d'autres applications, il est plus pertinent de n'utiliser qu'un seul type de nœuds, les autres étant réservés à un usage différent.

La méthode courante pour créer un arbre consiste tout d'abord à définir un type pointeur (arbre) faisant référence à “nœud” de type enregistrement :

```
type
    arbre=↑nœud ;
    nœud=record
            contenu : objet ;
            filsG : arbre ;
            filsD : arbre ;
         end ;
```

Pour construire un arbre à partir des arbres `a` et `b`, un nouveau champ d'enregistrement (`nœud`) sera créé par la variable `c` de type pointeur (`arbre`). Il permettra d'enregistrer la racine et les pointeurs sur les arbres `a` et `b` qui seront les arbres gauche et droit de cette racine. La fonction correspondante prend la valeur du nouvel arbre :

```
function NouvelArbre(v :objet ; a,b : arbre) : arbre ;
    var c : arbre ;
    begin
    new(c) ;
    c↑.contenu := v ;
    c↑.filsG := a ;
    c↑.filsD := b ;
    NouvelArbre := c ;
    end ;
```

Voici comment implémenter l'arbre de la figure 7.

```
program ArbreBinaire_A ;
type
    arbre=↑nœud ;
    nœud=record
            contenu : integer ;
            filsG : arbre ;
            filsD : arbre ;
         end ;
var A, B : arbre ;
function NouvelArbre(v :integer ; a,b : arbre) : arbre ;
    var c : arbre ;
    begin
    new(c) ;
    c↑.contenu := v ;
    c↑.filsG := a ;
    c↑.filsD := b ;
    NouvelArbre := c ;
    end ;
begin
B :=NouvelArbre(
            1,
            NouvelArbre(2, nil, nil),
            NouvelArbre(
                        3,
                        NouvelArbre(4, nil, nil),
                        NouvelArbre(5, nil, nil)
                        ) ;
A :=NouvelArbre(0, B, NouvelArbre(6, nil, nil)) ;
end.
```

A noter l'intervention de l'arbre vide (`nil`), ce qui indique que l'implémentation fait appel

à la définition récursive. Evidemment, le programme qui réalise l'implémentation n'est pas unique.

Une caractéristique commune aux arbres (binaires ou pas) est que chaque nœud possède un unique père (en admettant que la racine soit son propre père). Une représentation compacte d'un arbre général est donc obtenue par deux tables `a` et `p`, où `a[k]` contient les données du nœud indexé par `k` et `p[k]` est égale à l'index correspondant au père de nœud d'index `k`. Ainsi, l'arbre de la figure 7 est implémenté de manière très compacte par le tableau suivant :

a[k]=	0	1	2	3	4	5	6
p[k]=	0	0	1	1	3	3	0

Cette représentation est particulièrement recommandée si l'on n'a besoin que de remonter dans l'arbre. Elle ne donne pas d'informations sur les fils droit et gauche, mais on peut pallier facilement à cet inconvénient si c'est nécessaire.
Une manière plus complète de représenter un arbre en machine peut se faire en numérotant chaque nœud à partir de 1, par une liste A où $A[i]$ contient trois champs, le premier identifie le nœud, les deux autres contient le numéro du fils gauche et celui du fils droit. L'absence de fils droit ou gauche étant signalé par le numéro 0 qui remplace la variable pointeur `nil`. Ci-contre, voici ce que donne cette représentation dans le cas de l'arbre de la figure 3 :

1	$*$	2	11
2	$-$	3	4
3	$+$	6	5
4	$*$	9	10
5	$+$	7	8
6	a	0	0
7	b	0	0
8	x	0	0
9	b	0	0
10	y	0	0
11	c	0	0

3.7 Les traversées d'un arbre binaire

Une fois un arbre construit, une des premières choses à savoir faire est de déterminer un chemin tel que tout nœud appartienne à au moins un nœud du chemin. On dit, de manière imagée, que le chemin traverse ou visite un nœud lorsque celui-ci est extrémité d'une branche du chemin en question. Evidemment, on cherche un chemin de longueur minimale et en fonction de l'implémentation adoptée.

L'algorithme général de traversée est le suivant :

Algorithme *Traverser_Arbre*(α)
Entrer `un arbre binaire` α `de` n `nœuds ;`
Sortir `tous les nœuds avec une indexation` $a_1, \ldots, a_n$ `définissant un ordre total sur l'ensemble des nœuds.`

Nous allons étudier plusieurs algorithmes de parcours d'arbre lorsque celui-ci est binaires. Ils diffèrent entre eux essentiellement par l'ordre suivant lequel on visite les nœuds et leurs arbres fils gauches et droits.

Le premier algorithme est la *traversée préfixée* ; son implémentation repose sur la procédure récursive suivante :

```
TRAVERSER_PREFIXE(α) :
1. SI α ≠ NIL ALORS
2.     VISITER(Racine(α)) ;
3.     TRAVERSER_PREFIXE(FilsG(α)) ;
4.     TRAVERSER_PREFIXE(FilsD(α)) ;
```

On initialise la procédure avec pour entrée l'arbre α à parcourir. L'instruction `VISITER`(x) peut se traduire par l'impression ou l'enregistrement dans une pile (par exemple) du nœud x ou de tout autre clef y faisant référence. Par définition, `Racine(`α`)` retourne la racine de l'arbre α et `Fils_G(`α`)` (resp. `Fils_D(`α`)`) renvoie pour valeur l'arbre fils gauche (resp. droit) de la racine de α. Par exemple, la traversée préfixe de l'arbre binaire de la figure 6 commence par visiter la racine 0 puis visite successivement les nœuds 1, 2, 3, 4, 5, 6 dans cet ordre.

Le second algorithme est la *traversée intrafixée*. C'est la méthode la plus utilisée ; elle est donnée par la procédure récursive suivante :

TRAVERSER_INTRAFIXE(α*)* :
```
1. SI α ≠ NIL ALORS
2.    TRAVERSER_INTRAFIXE(FilsG(α) ;
3.    VISITER(Racine(α)) ;
4.    TRAVERSER_INTRAFIXE(FilsD(α) ;
```

La traversée intrafixée de l'arbre binaire de la figure 6 prend en entrée l'arbre, poursuit en prenant pour nouvelle entrée l'arbre fils gauche, puis l'arbre fils gauche de ce dernier qui n'a pas d'arbre fils gauche d'où la visite de sa racine qui est 2, l'arbre fils droit de [2] étant vide, l'algorithme revient sur l'arbre fils gauche de la racine pour traverser maintenant l'arbre fils droit de [1], etc. On obtient ainsi les visites successives des nœuds 2, 1, 4, 3, 5, 0, 6.

Le troisième algorithme est la *traversée postfixe* qui est sans doute la traversée la moins naturelle. Elle correspond à la procédure récursive suivante :

TRAVERSER_POSTFIXE(α*)* :
```
1. SI α ≠ NIL ALORS
2.    TRAVERSER_POSTFIXE(FilsG(α) ;
3.    TRAVERSER_POSTFIXE(FilsD(α) ;
4.    VISITER(Racine(α)) ;
```

Reprenons l'arbre binaire de la figure 6, sa traversée postfixée visite successivement les nœuds 2, 4, 5, 3, 1, 6, 0.

Le programme de parcours d'arbre correspondant aux algorithmes de traversées préfixée, intrafixée et postfixée est donnée ci-dessous. Les procédures préfixée, intrafixée et postfixée représentent les trois types principaux de traversées d'un arbre binaire.

```
Programme Parcours_arbre(A)
type
    pointeur=↑nœud ;
    nœud=record
            contenu : integer ;
            filsG : pointeur ;
            filsD : pointeur ;
         end ;
var
A : pointeur ;

procedure parcours_prefixe(sousarbre : pointeur) ;
begin
    if sousarbre <> nil then
        begin
            write(sousarbre↑.integer,'') ;
            parcours_prefixe(sousarbre↑.filsG) ;
            parcours_prefixe(sousarbre↑.filsD) ;
```

```
        end ;
end ;
procedure parcours_intrafixe(sousarbre : pointeur) ;
begin
    if sousarbre <> nil then
        begin
            parcours_intrafixe(sousarbre↑.filsG) ;
            write(sousarbre↑.integer,'') ;
            parcours_intrafixe(sousarbre↑.filsD) ;
        end ;
end ;
procedure parcours_postfixe(sousarbre : pointeur) ;
begin
    if sousarbre <> nil then
        begin
            parcours_postfixe(sousarbre↑.filsG) ;
            parcours_postfixe(sousarbre↑.filsD) ;
            write(sousarbre↑.integer,'') ;
        end ;
end ;
procedure arbre(racine : pointeur) ;
var
    clef : integer ; begin
        readln(clef) ;
        if clef=0 then racine := nil
            else
                begin
                    new(racine) ;
                    racine↑.contenu :=clef ;
                    write(racine↑.contenu, 'gauche :') ;
                    arbre(racine↑.gauche) ;
                    write(racine↑.contenu, 'droite :') ;
                    arbre(racine↑.droite) ;
                end ;
end ;
begin
    arbre(A) ;
    writeln ; parcours_prefixe(A) ;
    writeln ; parcours_intrafixe(A) ;
    writeln ; parcours_postfixe(A) ;
end.
```

Une quatrième méthode de traversée consiste à visiter de gauche à droite les nœuds à distance k de la racine en partant de $k = p$ (profondeur de l'arbre) puis de remonter les niveaux un à un jusqu'au niveau 0 où se trouve la racine. C'est la traversée par niveau ; elle n'est pas récursive. Ce type de traversée, appliquée à l'arbre de la figure 6, donne pour la suite des nœuds visités : 4, 5, 2, 3, 1, 6, 0.

***Exercice* 3.17** **a.** Appliquer les procédures de traversées préfixée, intrafixée et postfixée à l'arbre de la figure 3.
b. Vérifier que les traversées préfixée et postfixée donnent respectivement l'écriture préfixée et l'écriture postfixée de l'expression algébrique $((a + (b + x)) - b * y) + c$. Même question avec l'arbre binaire associé à l'expression algébrique $(a - b) * (a + b) + (c + d)$.
c. Démontrer que les résultats observés à la question précédente sont en fait vrais pour toutes les expressions algébriques (correctement formées) utilisant les opérations $+$, $-$, $*$ et $/$.

3.8 Arbres binaires de recherche

Les arbres binaires sont souvent utilisés pour représenter un ensemble de données dont les éléments peuvent être retrouvés au moyen d'une clef. Rechercher, insérer et supprimer des nœuds sont des opérations couramment effectuées sur les arbres. Afin de les réaliser rapidement, on introduit la notion d'*arbre binaire de recherche*. Il s'agit d'un arbre binaire enraciné (A, B, r) muni d'une application `clef` : $A \to \mathbf{N}$ qui vérifie les deux propriétés suivantes :

Pour tout nœud x

(i) si y est un nœud du sous-arbre fils gauche de x, alors

$$\texttt{clef}(y) \leq \texttt{clef}(x)\,;$$

(i) si y est un nœud du sous-arbre fils droit de x, alors

$$\texttt{clef}(x) \leq \texttt{clef}(y)\,;$$

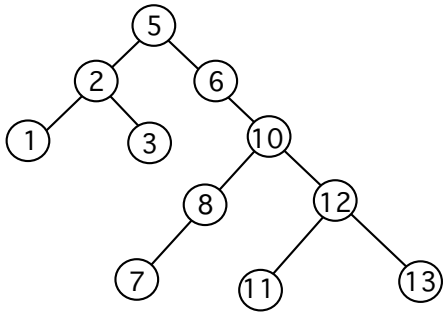


Figure 9 : un arbre binaire de recherche.

Il est habituel de supposer que l'application `clef` est injective. Remarquons que la procédure `TRAVERSER_INTRAFIXE` visite les nœuds suivant l'ordre croissant de leur clef puisque cette traversée commence par visiter en premier lieu la feuille la plus profonde et la plus à gauche puis remonte au père de cette feuille pour visiter l'arbre fils droit et ainsi de suite. Par exemple, pour l'arbre binaire de recherche de la figure 9, la traversée intrafixée donne la succession des clefs suivantes : 1, 2, 3, 5, 6, 7, 8, 10, 11, 12, 13.

Les requêtes

La requête la plus courante, effectuée sur un arbre binaire de recherche, est de rechercher une clef. Bien d'autres requêtes sont possibles comme par exemple celles de rechercher le nœud de clef minimale (resp. maximale) ou de renvoyer l'arbre successeur ou prédécesseur.

Recherche. La procédure suivante (en pseudo-code) recherche un nœud de clef donnée k dans un arbre binaire de recherche. La procédure commence sa recherche en partant de la racine et suit un chemin descendant dans l'arbre (sans positif) et pour chaque nœud x rencontré elle compare $\texttt{clef}(x)$ avec k. Si $k < \texttt{clef}(x)$, la recherche continue dans l'arbre fils gauche ; sinon, la recherche se poursuit dans l'arbre fils droit. Dans l'exemple de l'arbre de la figure 9, la recherche de la clef 7 traverse successivement les nœuds de clefs 5, 6, 10, 8 et 7.

Procédure *RECHERCHE_NŒUD*(x, k)
```
1. TANT QUE x ≠ NIL & k ≠ clef(x)
2.    FAIRE SI k < clef(x)
3.          ALORS x ← filsG(x)
4.          SINON x ← filsD(x) ;
5. RETOURNER x ;
```

Écrivons la procédure recherche en Pascal :

```
function Recherche(v :objet ; a : arbre) : arbre
    var r : arbre ;
    begin
        if a = nil then
            r := nil
        else if v = a↑.contenu then r := a
        else if v < a↑.contenu then
            r := Recherche(v, a↑.filsG)
        else
            r := Recherche(v, a↑.filsD)
        Recherche := r ;
        end ;
```

La fonction recherche renvoie un pointeur vers le nœud dont la clef est la valeur recherchée. Ici le type objet du champ `contenu` est un entier (valeur de clef). On ne tient pas compte des données attachées à chaque nœud, mais on peut évidemment les ajouter dans des champs d'enregistrement associés aux nœuds. La recherche teste d'abord si le contenu de la racine est égal à la valeur recherchée, sinon on lance récursivement la recherche dans l'arbre fils gauche si la valeur est inférieure au contenu de la racine, ou dans l'arbre fils droit dans le cas contraire.

Minimum et maximum. Le problème du minimum est de trouver un nœud d'un arbre binaire de recherche dont la clef est minimale. La procédure suivante exécute cette recherche en parcourant les pointeurs `filsG`$(\cdot)$ jusqu'à trouver la valeur `nil` :

Procédure *NŒUD_MINIMUM*(x)
1. TANT QUE `filsG`$(x) \neq$ NIL
2. FAIRE $x \leftarrow$`filsG`(x) ;
3. RETOURNER x ;

La recherche d'un nœud de clef maximale est analogue, il suffit, dans la procédure `NŒUD_MINIMUM`(x), de remplacer `filsG(`x`)` par `filsD(`x`)`.

Successeur et prédécesseur. Étant donné un nœud x dans un arbre binaire de recherche, on se propose de trouver le nœud successeur de x suivant l'ordre croissant des clefs lorsque celle-ci sont toutes distinctes. Cet ordre correspond à l'ordre de visite des nœuds dans la traversée intrafixée, ce qui permet de déterminer le successeur (et prédécesseur) sans effectuer de comparaison entre les clefs.

Procédure *NŒUD_SUCCESSEUR*(x)
1. SI `filsD`$(x) \neq$ NIL
2. ALORS RETOURNER `NŒUD_MINIMUM(filsD[`x`])` ;
3. $y \leftarrow$`père`(x) ;
4. TANT QUE $y \neq$`NIL` & $x =$`filsD`(y)
5. FAIRE $x \leftarrow y$
6. $y \leftarrow$`père`(y) ;
8. RETOURNER y ;

La procédure distingue le cas où l'arbre droit de x est vide ou pas. S'il n'est pas vide, le successeur de x est le nœud de clef minimale dans l'arbre fils droit ; celui-ci est donc obtenu en faisant appel à la procédure `NŒUD_MINIMUM` (ligne 2). Maintenant, si l'arbre droit de x est vide, on enregistre son père dans y (ligne 3) ; le père est vide (= `NIL`) si x est la racine. Dans les lignes 3 à 7, on remonte l'arbre à partir de x jusqu'à trouver un nœud qui soit fils gauche de son père, mais si on est retourné à la racine, $y =$`NIL` ce qui entraîne que `clef`(x) est maximum. On peut ajouter une ligne de pseudo-code pour donner cette information.

Nous laissons en exercice le travail d'écrire une procédure *NŒUD_PREDECESSEUR* qui détermine le nœud qui précède un nœud donné suivant l'ordre des clefs.

Insertion. La procédure d'insertion d'une nouvelle valeur de clef est analogue à la procédure recherche mais dans le cas d'égalité de clef, celle-ci est rangée dans l'arbre fils de gauche.

```
procedure Insertion(v :objet ; var a : arbre) ;
begin
    if a = nil then
        a := NouvelArbre(v,nil,nil)
    else if v <= a↑.contenu then
        Insertion(v, a↑.filsG)
    else
        Insertion(v, a↑.filsD) ;
end ;
```

Noter le passage par référence de l'argument `a` qui est la seule manière de modifier l'arbre.

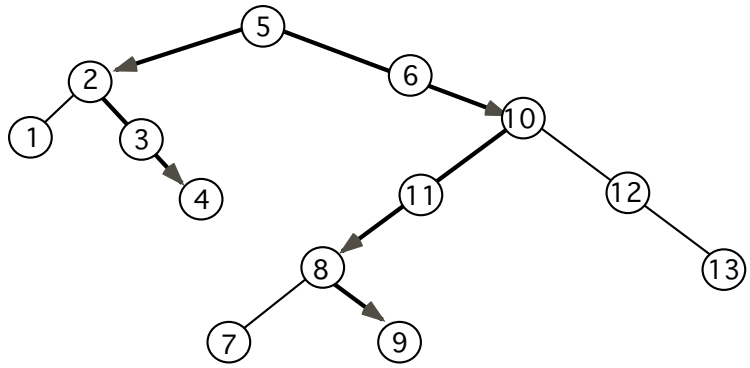


Figure 10 : insertion des clefs 4 et 9 dans l'arbre de recherche de la figure 9. Les flèches marquent le cheminement de l'algorithme à partir de la racine pour placer les nœuds associés à ces nouvelles clefs

***Exercice* 3.18** Écrire un programme pour implémenter la traversée par niveau d'un arbre binaire de recherche.

***Exercice* 3.19** Soit α un arbre binaire de recherche dont les clefs sont distinctes. Soit x une feuille de α et soit y son père. Montrer que `clef`(y) est soit la plus petite clef de α supérieure à `clef`(x), soit la plus grande clef de α inférieure à `clef`(x).

***Exercice* 3.20** La procédure `RECHERCHE_NŒUD` ci-dessus utilise une boucle `TANT QUE` pour déterminer le chemin descendant menant à la clef k. Réécrire cette procédure de manière récursive.

***Exercice* 3.21** Soit l'arbre binaire de recherche donné par le parenthésage suivant, où chaque nœud est désigné par sa clef de référence :

$$(5\ (1)(8))\,.$$

a. Déterminer les arbres binaires obtenus en insérant successivement les clefs 2, 3, 4, 6, 7, 9.
b. Obtient-on le même résultat final si on insère ces clefs dans un ordre différent ?

***Exercice* 3.22** Pour un arbre binaire de recherche, écrire en pseudo-code une procédure pour déterminer le nœud qui précède un nœud donné suivant l'ordre des clefs.

***Exercice* 3.23** Soit x un nœud d'un arbre binaire de recherche ayant deux nœuds fils. Montrer qu'alors son successeur n'a pas de fils et son prédécesseur n'a pas de fils droit.

***Exercice* 3.24** Montrer comment ordonner un ensemble d'entiers en construisant un arbre de recherche ayant pour clefs ces nombres (utiliser la procédure *Insertion* puis la procédure *parcours_intrafixe*). Écrire le programme (nommé `tri_arbre`) et tester le sur des exemples.

Chapitre 4

Algorithmes simples de tri et de recherche

4.1 Le problème du tri

Un annuaire téléphonique, une table des matières, un dictionnaire, la liste des numéros de sinistres d'une compagnie d'assurance..., fournissent autant d'exemples montrant l'intérêt d'un classement ordonné pour rechercher une information. En informatique, le tri est une opération de base qui a pour but de ranger des données ou objets suivant un ordre spécifique, soit directement lié aux données, soit à partir de clefs. Le tri est donc une activité essentielle dans l'exploitation des données. En outre, au résultat unique que constitue le tri s'oppose une grande diversité algorithmique, fournissant un sujet d'études particulièrement riche et diversifié. Prenons l'exemple de classer par ordre croissant une liste de nombres entiers. Qu'elle est l'algorithme le plus approprié pour réaliser ce classement ? On attend raisonnablement de celui-ci qu'il satisfasse au mieux aux trois critères suivants :

1. mise en œuvre simple ;
2. durée d'exécution minimale ;
3. espace mémoire optimisé.

Mais il faut aussi tenir compte de certaines contraintes. Par exemple, un petit nombre de données (disons une cinquantaine) ne se traitera pas avec le même algorithme que celui utilisé pour un grand nombre de données ; et si celles-ci sont déjà pré-classées, on utilisera de préférence un algorithme qui prend avantageusement en compte cette particularité.

La dépendance du choix d'un algorithme de tri en fonction de la structure même des données conduit aussi à distinguer deux catégories : le tri interne (ou par tableau) et le tri externe (ou pas fichiers). Dans le premier cas les données sont en mémoire avec accès direct et rapide. Dans le second cas, les données sont stockées sur des supports nécessitant des mécanismes de lecture et d'écriture qui vont pénaliser la rapidité du tri. Pour se faire une idée de la différence entre un tri interne et un tri externe, considérons le cas où l'on doit trier des fiches numérotées afin de les ranger dans une pile par leur numéro, dans l'ordre croissant en partant du bas. L'algorithme pour trier un petit nombre de fiches étalées sur une table sera très différent de celui à mettre en œuvre pour trier un grand nombre de fiches placées en un tas sur la table, de sorte que seule la fiche du haut soit visible. Nous laissons au lecteur le soin de trouver un algorithme dans le premier cas (et de le pratiquer concrètement, disons avec dix cartes extraites d'un jeu de carte classique). La seconde situation ne sera pas abordée dans ce chapitre.

Avant de poursuivre, énonçons le problème du tri sous la forme d'un algorithme général :

Algorithme *TRI* :
entrer une suite d'éléments

$$a_1, a_2, \ldots, a_n$$

à trier, chaque élément a_i étant associé une clef de classement k_i (l'ensemble des clefs étant muni d'un ordre total noté $\leq$) ;
sortir une permutation σ sur les indices telle que

$$k_{\sigma(1)} \leq k_{\sigma(2)} \leq \ldots \leq k_{\sigma(n)}.$$

Les clefs sont habituellement enregistrées en même temps que les éléments a_i. Ceux-ci seront désignés par le terme générique d'item ou d'objet. L'algorithme porte en fait sur les clefs qui sont souvent données sous forme alpha-numérique. Avec cette présentation, la structure d'enregistrement du langage Pascal est tout-à-fait adaptée. Rappelons à ce sujet qu'une telle structure est composée par un ensemble de données occupant des espaces mémoires appelés *champs* et pouvant être de divers types. Nous définissons par conséquent un type *item* ayant la structure générale suivante (et qui sera utilisée par la suite) :

type *item* = **record**
clef : `integer` ;
<*champ*>,..., <*champ*> : <`type`> ; ...
<*champ*>,..., <*champ*> : <`type`>
end

Ci-dessus, <*champ*> désigne un identificateur (qui sert aussi à nommer le champ) dont on doit préciser le type en lieu et place de <`type`> sur la ligne correspondante. Le champ *clef*[1] sert à identifier l'item et le trier ; le choix du type `integer` est ici arbitraire ; il peut être remplacé par tout type sur lequel est défini un ordre total.

L'algorithme de tri est dit *stable* si l'ordre initial des items de même clef est conservé au cours de l'exécution de l'algorithme. Une telle stabilité est souvent souhaitée, notamment lorsque un pré-tri est déjà effectué au moyen de clefs intermédiaires.

Certains items sont donnés suivant d'autres structures, comme par exemple celle d'une liste ou d'un arbre. Ces structures peuvent entrer directement dans la conception même de l'algorithme de tri.

Nous n'aborderons ici que les tris élémentaires. Ils offrent plusieurs avantages. Tout d'abord, les principes généraux de ces algorithmes interviennent dans un grand nombre de programmes. De plus, leur programmation est assez simple. Ils serviront, d'autre part, de premiers exemples d'analyse de la complexité d'un programme (ou de l'algorithme programmé). En fait, cette analyse est utile pour juger de l'efficacité – en termes de rapidité ou d'encombrement mémoire – d'un programme et permet de disposer d'éléments quantitatifs pour une étude comparative entre divers programmes prenant les mêmes données en entrée pour donner le même résultat en sortie.

4.2 Méthodes élémentaires de tri

Les items à trier sont supposés être enregistrés sous la forme d'un tableau. Ils sont directement accessibles. L'algorithme opère donc sur une variable a, de type `array`, dont les composants sont de type *item* :

var a : **array**$[1..n]$ **of** *item*

La permutation qui effectue le tri consistera donc à changer les entrées du tableau ou à écrire un autre tableau. Deux paramètres peuvent servir à mesurer la performance de ces

[1] L'orthographe "clef" est choisie à la place de "clé" pour éviter une lettre accentuée, habituellement rejetée par le langage Pascal pour les identifiants.

algorithmes ; ils sont, comme on peut s'y attendre, fonction du nombre n d'item à trier et de l'ordre suivant lequel ceux-ci sont enregistrés dans le tableau. Le premier paramètre choisi est le temps T d'exécution. Une bonne manière de l'évaluer est de compter le nombre C de comparaisons de clefs qui sont nécessaires pour réaliser le tri ainsi que le nombre M de déplacements d'items (*i.e.* de transpositions entre deux items ou de la mise en attente d'un item dans un emplacement mémoire).

Dans cette étude, on distingue tout d'abord les cas extrêmes. Il y a les pires cas, qui correspondent aux valeurs maximales de C et/ou M et il y a les meilleurs cas, avec des valeurs minimales pour C et/ou M. Il est ensuite intéressant de considérer la valeur moyenne de chacun de ces paramètres. On suppose alors que toutes les clefs sont distinctes. Remarquons maintenant que dans tout algorithme de tri, ce n'est pas les valeurs précises des clefs qui importent mais leur comparaison entre elles. On peut donc supposer, sans nuire à la généralité, que l'ensemble de ces clefs est formé des n premiers entiers naturels et se ramener au cas où les items sont réduits aux seules clefs. Ainsi, une méthode de tri consiste à passer de la permutation $\alpha := (a_1, \ldots, a_n)$ définie par le tableau a à la permutation *identité*, en effectuant une succession de permutations intermédiaires. Le modèle probabiliste qui s'impose est donc celui de l'ensemble $\mathcal{S}_n$ des permutations de $\{1, \ldots, n\}$ muni de l'équiprobabilité. Rappelons que $\mathcal{S}_n$ est de cardinalité $n!$. Il sera utile, pour chaque permutation $\sigma \in \mathcal{S}_n$, d'introduire son nombre d'inversions $I(\sigma)$, c'est-à-dire le nombre de couples (i, j) tels que $1 \leq i < j \leq n$ et $\sigma(i) > \sigma(i)$. La permutation est dite paire (resp. impaire) si $I(\sigma)$ est pair (resp. impair).

Calculons le nombre moyen d'inversions d'une permutation, c'est-à-dire, en termes probabilistes l'espérance de la *variable aléatoire* $I : \mathcal{S}_n \to \mathbf{N}$ ou, de manière usuelle, le rapport

$$I_{\text{moy}} = \frac{1}{n!} \sum_{\sigma \in \mathcal{S}_n} I(\sigma).$$

Une méthode pour calculer I_{moy} est d'associer à toute permutation $\sigma = (\sigma(1), \sigma(2) \ldots, \sigma(n))$ sa permutation miroir $\tilde{\sigma} = (\sigma(n), \sigma(n-1), \ldots, \sigma(1))$. En fait, un couple (i, j) est une inversion pour σ si, et seulement si, il n'est pas une inversion pour $\tilde{\sigma}$, de sorte que $I(\sigma) + I(\tilde{\sigma}) = \frac{n(n-1)}{2}$ (nombre de paires $\{i, j\}$ formées d'entiers i et j distincts, compris entre 1 et n) et en conséquence :

$$I_{\text{moy}} = \frac{n(n-1)}{4}. \tag{4.1}$$

***Exercice* 4.1** Soit $E_n = \{1, \ldots, n\}$ l'ensemble des n premiers nombres entiers naturels, avec $n \geq 2$. Une permutation τ de E_n est appelée une transposition s'il existe a et b dans E_n, distincts, tels que $\tau(a) = b$, $\tau(b) = a$ et $\tau(x) = x$ pour tous les autres entiers de E_n. Cette transposition est alors notée τ_{ab}.
a) Montrer que le nombre $I(\tau_{ab})$ d'inversions de $\tau_{a,b}$ est impair.
b) Plus précisément, montrer que pour $1 < s \leq n$ on a $I(\tau_{1,s}) = 2s - 3$.
c) Soit la permutation circulaire γ définie par $\gamma(1) = n$ et $\gamma(s + 1) = s$ pour $1 \leq s \leq n - 1$. Montrer que $I(\gamma) = n - 1$.
d) Démontrer la relation $\gamma = \tau_{n-1,n} \circ \tau_{n-2,n-1} \circ \cdots \circ \tau_{1,2}$.
e) Donner un algorithme qui prend en entrée une permutation de E_n et donne en sortie le nombre d'inversions de la permutation.
f) Calculer la plus grande et la plus petite des valeurs prises par $I(\sigma)$ lorsque σ parcourt l'ensemble des permutations de E_n. Déterminer les permutations qui réalisent ces valeurs extrêmes.

La quantité de mémoire supplémentaire pour effectuer un tri constitue un second paramètre important à considérer. Nous distinguons grossièrement trois sortes de tris internes. Ceux effectués *in situ* qui n'utilisent pas ou très peu de mémoire supplémentaire. C'est le cas des tris élémentaires que nous allons étudier. Ceux qui utilisent une liste chaînée nécessitant n espaces mémoires additionnels pour lister les pointeurs, par exemple, et ceux qui ont besoin de suffisamment d'espace mémoire pour effectuer un second enregistrement du tableau à trier.

Bien que les algorithmes rapides s'exécutent avec un nombre de comparaisons proportionnel à $n \log n$, nous restreindrons notre études aux algorithmes (dits élémentaires) qui nécessitent

un nombre de comparaisons de l'ordre de n^2. Il y a de bonnes raisons pour s'intéresser à ces algorithmes. Ils sont simples à programmer, entre dans les mécanismes de base d'algorithmes plus efficaces et enfin, dans le cas d'un petit nombre d'items à trier, ils sont très performants, faciles à mettre en oeuvre et peuvent parfois s'avérer plus rapides qu'un algorithme sophistiqué, conçu pour un très grand nombre de données.

Les méthodes de tris élémentaires *in situ* se classent en trois grandes catégories :

1. Le tri par insertion (simple) ;
2. le tri par sélection ;
3. le tri par échange.

Nous allons étudier chacun d'eux.

4.2.1 Tri par insertion

Cette méthode a déjà fait l'objet d'une étude dans le chapitre d'introduction où, pour illustrer la notion d'algorithme, nous nous étions contentés de trier un tableau de nombres. Donnons maintenant une procédure plus complète en accord avec la structure choisie pour enregistrer les données. L'algorithme est conçu comme suit. Les éléments à trier constituent une suite de clefs $a_1, \ldots, a_n$ rangées dans une table, de gauche à droite. Pour i allant de 2 à n, en incrémentant de 1 à chaque passage, on sélectionne le i-ème élément de la suite pour l'insérer à la bonne place dans la suite $t_1, \ldots, t_{i-1}$ qui résulte du tri de la liste partielle $a_1, \ldots, a_{i-1}$. Le processus d'insertion de $x := a_i$ à la bonne place se fait par comparaison avec les éléments successifs t_j suivi d'un déplacement de t_j vers la droite[2] si la comparaison fournit $x < t_j$. Il y a deux conditions d'arrêt, l'une lorsque $t_j \leq x$, l'autre lorsqu'on atteint le premier de la liste déjà triée (donc le plus petit). Notons au passage que nous n'avions pas tenu compte de cette condition dans la procédure "insertion" donnée au chapitre 1, section 6. Nous unifions ces conditions d'arrêt en introduisant l'indice 0 par une extension du champ des indices de a. L'affectation $a[0] := x$ permet donc de se ramener à une seule condition d'arrêt, comme cela est spécifié dans la procédure Pascal suivante :

```
procedure InsertionSimple ;
  var
    i, j : integer ;
    x : item ;
begin
  for i :=2 to  n do
  begin
    x :=a[i] ;
    a[0] :=x ;
    j :=i ;
    while x.clef < a[j-1].clef do
      begin
        a[j] := a[j-1] ;
        j := j-1 ;
      end ;
    a[j] :=x
  end ;
end ;
```

***Exercice* 4.2** On considère le tableau d'entiers `a[0..5]` suivant :

	5	2	3	3	1

sur lequel est effectuée la procédure *InsertionSimple* pour le trier (*i.e.* pour mettre les entiers dans l'ordre croissant de gauche à droite). Au début de la procédure, on a `i=2`, `x=2`, `j=2` et l'exécution de l'instruction

[2] avec, dans la suite, le déplacement vers la droite des éléments t_k pour $j < k \leq i - 1$.

while se fait en une boucle et place la valeur de `a[2]` dans `a[1]` ; en sortie de boucle `j=1` ce qui provoque le fin de l'instruction **while** par le test `x.`*clef* `< a[j-1].`*clef*, entraînant l'incrémentation de `i`. L'ensemble des valeurs de `a`, `i` et `j` au cours de ce début d'exécution est représenté par la table d'évolution suivante :

i	a_0	a_1	a_2	a_3	a_4	a_5	j
2	2	5	2	3	3	1	2
2	2	5	5	3	3	1	2
2	2	2	5	3	3	1	1
3	2	2	5	3	3	1	1

Continuer cette présentation pour décrire l'état du tableau `a` et des variables `i` et `j` à chaque modification.

Analyse du tri par insertion. Notons C_i le nombre de comparaisons de clefs effectuées dans la procédure *InsertionSimple* pour i fixé et j variable de i à 1 (au moins). Soit M_i le nombre d'affectations correspondantes ne portant que sur les items, y compris l'affectation $a[0] := x$. Si les clefs sont déjà dans l'ordre, C_i et M_i sont minimum pour tout $i = 2, \ldots, n$, à savoir $C_i = 1$ et $M_i = C_i + 2 = 3$. Dans ce cas, le nombre total de comparaisons et d'affectations d'items sont respectivement

$$C_{\min} = n - 1 \quad \text{et} \quad M_{\min} = 3(n-1).$$

Si maintenant les clefs sont dans l'ordre opposé, l'algorithme nécessite un maximum de comparaisons et d'affectations. Dans ce cas, $C_i = i$ et $M_i = C_i + 2$, d'où les valeurs maximales

$$C_{\max} = \frac{n(n+1)}{2} - 1 \quad \text{et} \quad M_{\max} = \frac{n(n+5)}{2} - 3.$$

Il est intéressant de déterminer le nombre moyen de comparaisons et d'affectations lorsque le tri opère sur toutes les permutations de clefs (comme convenu plus haut, les clefs sont supposées distinctes et leurs permutations sont toutes équiprobables). Le nombre de comparaisons pour insérer une clef dans la suite déjà triée des clefs précédentes est égal au nombre d'inversions qu'elle présente avec ces clefs, plus un. En d'autres termes

$$C_i = 1 + \operatorname{card}\{a_j \,;\, a_j > a_i, j < i\}.$$

Si α est la permutation déterminée par la suite à trier, le nombre total de comparaisons pour la trier est donc

$$C(\alpha) = \sum_{i=2}^{n} C_i = n - 1 + I(\alpha),$$

où $I(\alpha)$ désigne le nombre d'inversions de α. La valeur moyenne de $C(\cdot)$, en tenant compte de la valeur moyenne des inversions I_{moy} (cf. (4.1)), est donc

$$C_{\text{moy}} = n - 1 + \frac{n(n-1)}{4} = \frac{n^2 + 3n}{4} - 1.$$

Notons que la performance de l'algorithme de tri par insertion est directement lié au nombre d'inversions de la permutation α déterminée par le tableau à trier. Il est d'autant plus rapide que ce tableau est presque ordonné au sens que le nombre d'inversions de α est petit.

L'algorithme peut être amélioré au cours de l'insertion de $x = a[i].$*clef* dans la suite $a[1].$*clef*,…,$a[i-1].$*clef* qui est toujours ordonnée croissante. On peut en effet, au lieu d'effectuer les comparaisons allant de gauche à droite (*i.e.* de $a[i-1]$ vers $a[1]$), commencer par comparer x avec le terme médian de a, ce qui permet de poursuivre la comparaison à droite ou à gauche de ce terme et ainsi de suite, par dichotomies successives, pour arriver à l'emplacement où x doit être inséré.

4.2.2 Tri par sélection

Ce tri est sans doute le plus simple, il est basé sur la recherche du plus petit élément dans une suite de nombres entiers :

Algorithme *TRI_SELECTION*
Entrer une suite de nombres (clefs) $a_1, \dots a_n$.
Sortir un arrangement $i \mapsto a_{\sigma(i)}$ des termes de la suite tel que $a_{\sigma(1)} \leq a_{\sigma(2} \leq \dots \leq a_{\sigma(n)}$.
début
1. POUR $i = 1$ A $n-1$ FAIRE
2. $x \leftarrow a_i$;
3. $j \leftarrow \min\{k, i \leq k \leq n \ \& \ a_k = \min\{a_i, \dots, a_n\}\}$;
4. $a_i \leftarrow a_j$;
5. $a_j \leftarrow x$;
fin.

On commence donc par échanger a_1 avec a_j où j est le plus petit indice correspondant à la plus petit valeur du tableau a, ce qui donne une premier arrangement des valeurs du tableau qui met en première position (à gauche) la plus petite valeur. On obtient un nouveau tableau, toujours noté par a, et on répète l'opération sur le tableau a tronqué de son premier terme (*i.e.* sur $[a_2, \dots, a_n]$) et ainsi de suite jusqu'à obtenir un tableau à deux termes dont on détermine le plus petit. Les instructions 1 à 5 ci-dessus correspondent à deux boucles emboîtées que l'on peut écrire en Pascal comme suit :

```
begin for i :=1 to n-1 do
  begin
    k :=i ;
    x :=a[i] ;
    for j :=i+1 to n do
      if a[j].clef < x.clef then
      begin
        k :=j ;
        x :=a[j] ;
      end ;
    a[k] :=a[i] ;
    a[i] :=x ;
  end ;
end ;
```

Analyse du tri par sélection. Le nombre C de comparaisons de clefs ne fait pas intervenir l'ordre initial dans lequel se trouve le tableau des clefs. Ces comparaisons se font, pour chaque i donné, sur toutes les valeurs des clefs `a[i+1].clef, ..., a[n].clef`, d'où un nombre totale de comparaisons :

$$C = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}.$$

Si les clefs sont déjà ordonnées, pour chaque valeur de i il faut compter trois affectations (ce qui est le minimum). Le nombre minimum d'affectations pour l'algorithme est donc atteint dans ce cas particulier et vaut

$$M_{\min} = 3(n-1).$$

Si les clefs sont dans l'ordre décroissant, outre les trois affectations incontournables, il faut ajouter les affectations dans les boucles portant sur `j`. Or l'échange entre `a[1]` et le minimum des `a[j]` pour `j>1` met `a[1]` (la plus grande valeur) à la place `n` ; pour `i=2` l'algorithme met `a[2]` à la place `n-1` et ainsi de suite. A la sortie de la boucle correspondant à `i=s`, avec `s` $\leq$

`n/2`, on a la fin du tableau dans le bon ordre, à savoir `a[n-s+1]<a[n-s+2]<...<a[n]`. Cela conduit au nombre maximum d'affectations suivant :

$$M_{\max} = 3(n-1) + \sum_{1 \le i \le n/2} (n - 2i + 1).$$

Rappelons que $\sum_{m=1}^{k}(2m-1) = k^2$ (preuve par récurrence) d'où (en distinguant selon la parité de n),

$$M = 3(n-1) + \left\lfloor \frac{n}{4} \right\rfloor . \text{ (3)}$$

La valeur moyenne de M_{moy} est plus complexe à évaluer, elle est fonction évidemment de n. Son ordre de grandeur est en $n \log n$. Comme l'opération d'affectation est plus coûteuse en temps, on voit que le tri par sélection directe est en moyenne plus avantageux que celui par insertion, même si dans le cas où les clefs sont déjà dans l'ordre ou presque, le tri par insertion s'avère plus rapide.

***Exercice* 4.3** a) Donner deux démonstrations différentes de la formule $\sum_{i=1}^{n} = \frac{n(n-1)}{2}$.
b) En utilisant l'identité $(x+1)^2 - x^2 = 2x + 1$, montrer que

$$(n+1)^2 = \sum_{k=0}^{n} (2k+1).$$

c) Déterminer un polynôme P tel que pour tout entier $n \ge 1$, on a

$$P(n) = \sum_{k=1}^{n} n^2$$

(indication : développer $(x+1)^3 - x^3$).

4.2.3 Le tri par échange ou tri-bulle

Ce tri est basé sur la comparaison des clés adjacentes et leur permutation si elles ne sont pas dans l'ordre croissant. Ces comparaisons se font par balayages successifs du tableau ; à chaque passe, les éléments plus petits glissent à gauche. Ce tri est encore appelé tri-bulle (bubblesorting) à cause de sa représentation imagée suivant une colonne de bulles dans l'eau qui se positionnent verticalement selon leur taille (la plus petite en bas, la plus grosse en haut...). L'algorithme peut se décrire comme suit :

Algorithme *TRI_BULLE*
Entrer une suite de nombres (clefs) $a_1, \ldots a_n$.
Sortir un arrangement $i \mapsto a_{\sigma(i)}$ des termes de la suite tel que $a_{\sigma(1)} \le a_{\sigma(2} \le \ldots \le a_{\sigma(n)}$.
début
1. POUR $i = 2$ A n FAIRE
2. POUR $j = n$ A i PAR PAS DE -1 ;
3. TANT QUE $a_{j-1} > a_j$ FAIRE ECHANGE(a_{j-1}, a_j) ;
fin.

Nous laissons en exercice l'écriture en Pascal d'une procédure *TriBulle* réalisant l'algorithme ci-dessus.

***Exercice* 4.4** Reprendre la table `a` de l'exercice 4.2, sans tenir compte de a_0, et effectuer le tri de celle-ci par l'algorithme *TRI_BULLE*. On donnera tous les résultats intermédiaires du tri sur ce tableau en fonction des valeurs de `i` et `j`.

[3] On note $\lfloor x \rfloor$ la partie entière de x.

Analyse du tri-bulle. Le nombre C de comparaisons est indépendant de l'ordre des clefs et c'est la somme des $n-1$ premiers entiers (1 pour $i = n$, 2 pour $i = n-1, \ldots, n-1$ pour $i = 2$) soit

$$C = \frac{n(n-1)}{2}.$$

Si les clefs sont déjà dans l'ordre requis, il n'y a aucune affectation et donc $M_{\min} = 0$. Si l'ordre des clefs est décroissant, il faut à chaque comparaisons effectuer un échange qui nécessite, comme dans le tri précédent, 3 affectations pour effectuer l'échange, d'où un nombre maximum d'affectations égal à $M_{\max} = \frac{3}{2}n(n-1)$.

Étudions le nombre moyen d'affectations. Soit donc à trier le tableau a constitué des n premiers entiers naturels. Soit α la permutation définie par a. On peut noter que dans l'algorithme tri-bulle, chaque échange d'items supprime une inversion de α et une seule. Ainsi le nombre total d'inversions dans ce tri est égal au nombre d'inversions $I(\alpha)$ de α. Calculer le nombre moyen d'affectations revient à calculer le nombre moyen d'inversions de l'ensemble des permutations de $\{1, \ldots, n\}$. Ce calcul a déjà été fait. En conséquence, le nombre moyen d'affectations dans le tri-bulles est

$$M_{\text{moy}} = \frac{3n(n-1)}{4}.$$

***Exercice* 4.5** Parmi les trois algorithmes de tris élémentaires (insertion, sélection, échange),
a) lesquels sont stables ?
b) Quel est le plus rapide de ces algorithmes pour trier un tableau déjà dans l'ordre ? dans l'ordre inverse ?

***Exercice* 4.6** Écrire un programme Pascal complet effectuant un tri
a) par insertion,
b) par sélection,
sur un tableau de nombres de taille $n = 10$, créé de manière aléatoire. On utilisera pour cela – outre une procédure de tri appropriée – les deux procédures suivantes :

```
procedure Initialisation ; (* construction d'un tableau de nombres choisis au hasard *)
    var i : integer ;
    begin
    for i :=1 to n do
        a[i] := 50+ round((random/maxint)*50) ;
    end ;
```

(Rappelons que `maxint` est un identificateur de constante qui représente la plus grande valeur absolue que peut prendre un entier dans le système utilisé).

```
procedure Impression ;
    var i : integer ;
    begin
    for i :=1 to n do
        write(a[i], ' ') ;
        writeln ;
    end ;
```

***Exercice* 4.7** Écrire un programme complet en Pascal qui effectue un tri-bulle sur une tableau de 10 entrées, ce tableau étant créé de manière aléatoire. On utilisera les procédures d'initialisation et d'impression données dans l'exercice précédent et on imprimera (affichage à l'écran) les résultats du tableau après chaque balayage.

4.3 Tri par un arbre de recherche

Les tris étudiés ci-dessus portent sur des données statiques. Il peut arriver qu'il soit nécessaire de supprimer des données ou d'en ajouter de nouvelles. On est donc dans une situation dynamique correspondant à un ensemble variable d'items à trier régulièrement. On a développé au chapitre précédent la notion d'arbre (binaire) de recherche que l'on va utiliser pour répondre à ce nouveau type de rangement. Partant de l'arbre vide, la procédure insertion permet de construire pas à pas un arbre binaire de recherche dont les nœuds servent

d'enregistrement des clefs et des items qui leur sont rattachés. La procédure *supprimer* un item correspond à supprimer un nœud dans un arbre de recherche. Cette procédure est plus complexe que celle de l'insertion. En fait, la suppression est banale si le nœud est une feuille ou ne possède qu'un fils. Maintenant, si le nœud possède deux fils, l'item à supprimer doit être remplacé par l'item le plus à droite de son sous-arbre de gauche ou, à défaut, par l'item le plus à gauche de son sous-arbre de droite. Le tri s'effectue ensuite par un parcours d'arbre.

4.4 Recherche dans une table

Nous avons vu précédemment comment trier une table. Examinons maintenant deux méthodes de recherche dans une table. La première est dite recherche séquentielle où les éléments de la table sont lus, les uns après les autres, jusqu'à trouver l'élément recherché. La seconde est dite dichotomique ; elle porte sur un table déjà triée.

4.4.1 Recherche séquentielle

On se place dans la situation concrète suivante : la table est un annuaire téléphonique (électronique, bien sûr) que l'on suppose constituée par un tableau `nom` et un tableau `tel` de sorte que `nom[i]` est le nom de la personne dont le numéro de téléphone est `tel[i]`. La recherche consiste à trouver le numéro de téléphone d'une personne abonnée. Pour cela, on parcours séquentiellement le tableau `nom`. La fonction recherche correspondante peut se définir comme suit :

```
function RechercheSeq (x : string) : integer ;
  var
    i : integer ;
    ok : boolean ;
begin
  i := 1 ;
  ok := false ;
  while not ok and (i <= N) do
    begin
      ok := x = nom[i] ;
      i := i + 1 ;
      (* "ok" prend la valeur "true" dès que "x=nom[i0]" et *)
      (* dans ce cas, la boucle "while" s'arrête avec la valeur *)
      (* "i=i0+1" *)
    end ;
  if ok then
    RechercheSeq := tel[i-1]
  else
    RechercheSeq := 0 ;
end ;
```

Le nombre maximal d'exécution de la boucle `while` est évidemment `N`, tandis que le nombre minimal est 1. L'étude du nombre moyen est proposé en exercice.

4.4.2 Recherche dichotomique

Plaçons nous dans le cas de l'annuaire téléphonique précédent, mais avec un tableau `nom` déjà trié dans l'ordre alphabétique. Au lieu de rechercher séquentiellement dans ce tableau, on procède par dichotomie :

(i) comparer la clef (qui est le nom `x` *à chercher) avec le nom* `m` *du milieu du tableau ;*
(ii) si `x=m`*, retourner le numéro de téléphone de* `m` *;*
(iii) sinon, procéder de la même manière avec le tableau constitué de la
moitié gauche ou
moitié droite
du tableau initial, selon que `x` *est lexicographiquement*
plus petit ou
plus grand
que `m` *;*
(iv) itérer le procédé.

La fonction recherche suivante donne le code en Pascal de cet algorithme ; elle s'accompagne des procédures d'initialisation des tableaux `nom` et `tel` qui ne sont pas données ici :

```
function RechercheDichoto (x :string) : integer ;
  var i, g, d :integer ;
begin
(* initialisation des valeurs extrêmes du tableau de recherche *)
  g := 1 ;
  d := N ;
  repeat
    i := (g + d) div 2 ;
    (* la variable i se positionne en milieu de tableau *)
    if x < nom[i] then
      d := i - 1
    else
      g := i + 1 ;
  until (x = nom[i]) or (g > d) ;
    if x = nom[i] then
      RechercheDichoto := tel[i]
    else
      RechercheDichoto := 0 ;
end ;
```

Il est facile de voir que le nombre maximum B_N d'exécutions de la boucle `repeat` pour un tableau de taille N vérifie la relation

$$B_N \leq 1 + B_{\lfloor N/2 \rfloor}$$

avec $B_1 = 1$. Comme B_N est une fonction croissante de N, pour tout k tel que $2^k < N$ on en déduit la majoration

$$B_N \leq k + B_{\lfloor N/2^k \rfloor},$$

d'où

$$B_N \leq 1 + \frac{\log N}{\log 2}.$$

La recherche dichotomique est donc nettement plus efficace que la recherche séquentielle dès que le tableau de recherche est de grande taille et déjà trié.

***Exercice* 4.8** On suppose toutes les clefs identiques dans le tri par arbre de recherche. Quel arbre construit-on ?

***Exercice* 4.9** Dans la fonction *RechercheSeq* donnée dans le cours (sous-section 4.4.1), on effectue deux tests : l'un sur la valeur de `ok` et l'autre sur la valeur booléenne de `(i <= N)`. Si les tables `nom` et `tel` ont `N+1` entrées, on peut alors mettre une valeur *sentinelle* en bout de table pour supprimer la close d'arrêt `i>N`. Pour cela, on initialise la recherche avec

```
nom[N+1]=x ; tel[N+1]=0
```

puis, on parcourt le tableau `nom`, pour `i=1,2,...`, tant que `x` $\neq$ `nom[i]`. Si `x` n'est pas dans le tableau initial, le parcours se termine avec `i=N+1` et la fonction *RechercheSeq* retourne la valeur 0. Écrire en Pascal un fonction *RechercheSeq* en suivant ce principe.

***Exercice* 4.10** Avant d'utiliser la fonction *RechercheSeq* du cours (sous-section 4.4.1) avec `x` fixé, on effectue sur les tableaux `nom` et `tel` une permutation σ (qui remplace `nom[i]` et `tel[i]` respectivement par `nom[`σ`(i)]` et `tel[`σ`(i)]`).
a) Soit $T(\sigma)$ le nombre de fois que le test de la boucle "while" est exécuté. Calculer la valeur moyenne T_{moy} de T, c'est-à-dire :

$$T_{\text{moy}} = \frac{1}{N!} \sum_{\sigma \in \mathcal{S}_N} T(\sigma).$$

b) Les tableaux `nom` et `tel` étant donnés, on note t(x) le nombre de fois que le test de la boucle "while" est exécuté dans la recherche de `x`. Calculer la valeur moyenne de t, les entrées x étant supposées équiprobables.

***Exercice* 4.11** Écrire un programme complet de recherche séquentielle en table. Pour cela, appliquer le programme sur une table de noms de dix personnes (réelles ou imaginaires).

***Exercice* 4.12** La borne de B_N donnée dans la recherche dichotomique (sous-section 4.4.2) peut-elle être atteinte ?

***Exercice* 4.13** Écrire un programme complet de recherche dichotomique en table. Pour cela appliquer le programme sur une table de noms de dix personnes rangés par ordre alphabétique.

Chapitre 5

Récursivité

5.1 Introduction

Un algorithme ou une procédure sont dits récursifs s'il font appels à eux-même dans leur construction. De même, une fonction est dite récursive si elle est définie en termes d'elle-mêmes. La forme générale d'un algorithme récursif est donc la suivante

Algorithme *RECURSION()*
entrer des données,
utiliser *RECURSION()*,
sortir des données .
début
1. INSTRUCTIONS ;
2. . . . FAIRE *RECURSION()* ;
3. INSTRUCTIONS ;
fin.

Un tel algorithme peut conduire à l'exécution d'une infinité d'instructions, il est donc nécessaire de prévoir une condition d'arrêt. Il y a plusieurs méthodes pour la réaliser. Par exemple, si `A` est une condition qui devient fausse en cours d'exécution de l'algorithme ci-dessus, alors la ligne 2 peut être remplacée par

`2. Si ''A'' ALORS...`
RECURSION() ;

En pratique, l'algorithme dépend d'un paramètre entier n (voire de plusieurs), ce que nous explicitons par *RECURSION(n)* et "A" se présente, par exemple, sous la forme "$n > 0$", l'entier n étant décrémenté de 1 à chaque appel de *RECURSION*. Dans ces conditions, la ligne 2 devient

`2. Si` $n > 0$ `ALORS...`
RECURSION$(n-1)$

Nous avons déjà eu l'occasion d'utiliser des procédures récursives. Au chapitre 2 (section 2.3) par exemple, la procédure `supprimer_liste` permet de parcourir une liste (à partir d'une cellule donnée) à la recherche d'un objet à supprimer par appel successif à la procédure elle-même.

Les arbres binaires ont, par définition, une structure typiquement récursive ; il n'est donc par surprenant que les algorithmes de parcours (`parcours_prefixe`, `parcours_intrafixe` et `parcours_postfixe`) soient naturellement récursifs (cf. section 3.7).

La récursivité est une méthode intéressante pour construire des ensembles infinis au moyen d'un programme fini ; elle est habituellement bien adaptée pour traiter des données définies

de manière récursive, comme les arbres. Mais il faut prendre garde à ce que l'exécution de l'algorithme ne conduise pas une complexité explosive pouvant rapidement conduire à un dépassement de capacité mémoire.

Donnons maintenant deux exemples classiques et très simples illustrant la puissance (ou les faiblesses) d'un programme récursif.

5.1.1 La fonction factorielle

Il s'agit de calculer $n! = 1.2.3 \ldots n$ pour tout entier $n \geq 1$. Commençons par donner un programme non récursif assez naïf :

```
program factoriel ;
  var
    i, n, F : integer ;
begin
  read(n) ;
  i := n ;
  F := n ;
  while i > 1 do
    begin
      i := i - 1 ;
      F := F * i ;
    end ;
  writeln('factorielle de', n, ' =', F) ;
end.
```

Le programme suivant fait le même calcul avec appel à la fonction factorielle mise sous une forme récursive :

```
program factorielR ; (* R pour récursif *)
  var
    n : integer ;
  function Fact (n : integer) : integer ;
    var
      F : integer ;
  begin
    if n <= 1 then
      F := 1
    else
      F := n * Fact(n - 1) ;
    Fact := F ;
  end ;
begin
  read(n) ;
  writeln('factorielle de', n, '=', Fact(n)) ;
end.
```

Notons que l'on dépasse vite les capacités de calcul en mode standard puisque maxint=32767 pour une machine travaillant en 16 bits, alors que $8! = 40320$. Pour aller plus loin, il est nécessaire d'adapter le programme à ces contraintes et donc d'écrire une arithmétique permettant de calculer avec de grands nombres.

5.1.2 Nombres de Fibonacci

Le deuxième algorithme consiste à calculer le n-ième terme f_n de la suite des nombres de Fibonacci, suite définie classiquement par la récurrence linéaire d'ordre 2 suivante avec

conditions initiales :

$$f_0 = f_1 = 1 \quad \& \quad f_n = f_{n-1} + f_{n-2}. \tag{5.1}$$

Écrivons ces relations sous la forme matricielle

$$\begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \end{pmatrix} \tag{5.2}$$

en posant $f_{-1} = 0$ pour inclure le cas $n = 1$. Les égalités (5.2) donnent, par induction sur l'entier $n \geq 1$,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_n & f_{n-1} \\ f_{n-1} & f_{n-2} \end{pmatrix}. \tag{5.3}$$

Cela suggère d'utiliser un programme itératif basé sur les affectations

$$\begin{aligned} u &:= u + v \\ v &:= u\,. \end{aligned}$$

Donnons le code de la fonction $n \mapsto f_n$ en suivant cette méthode :

```
function fibo (n : integer) : integer ;
  var
    u, v : integer ;
    x, y : integer ;
    i : integer ;
  begin
    u := 1 ;
    v := 1 ;
    for i := 2 to n do
      begin
        x := u ;
        y := v ;
        u := x + y ;
        v := x ;
      end ;
    fibo := u ;
end ;
```

Voyons maintenant la fonction récursive correspondante, basée sur la relation (5.1) :

```
function fiboR (n : integer) : integer ;
  begin
  if n<= 1 then
    fiboR :=1
  else
    fiboR :=fiboR(n-1)+fiboR(n-2) ;
end ;
```

Dans l'appel récursif, `fiboR(n)` et `fiboR(n-1)` sont calculés indépendamment. `fiboR(n)` fait appel à `fiboR(n-1)` et `fiboR(n-2)` tandis que `fiboR(n-1)` fait appel à `fiboR(n-2)` et `fiboR(n-3)`, et ainsi de suite. Mais il n'y a pas de coopération entre ces appels. Représentons

sous la forme d'un arbre les appels récursifs effectués au cours du calcul de `fibo(5)` :

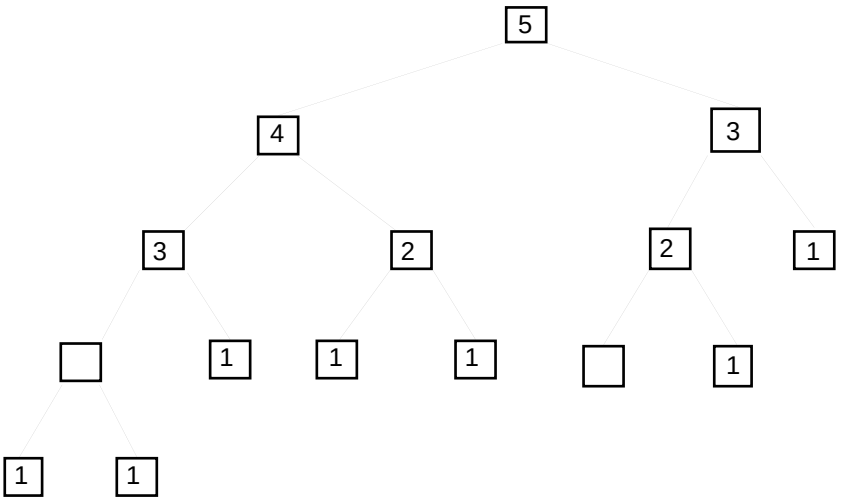


Figure 5.1. Arbre des appels récursifs pour `fiboR(5)` ; un nœud d'étiquette n représente l'appel à `fiboR(`n`)`

Plus généralement, notons r_n le nombre d'appels de la fonction `fiboR()` au cours du programme récursif qui calcule f_n. On a immédiatement, au vu du programme, $r_n = r_{n-1} + r_{n-2} + 1$ pour $n > 1$ et $r_0 = r_1 = 1$. Remplaçons r_n par $R_n = r_n + 1$ de sorte que $R_0 = R_1 = 2$ et $R_n = R_{n-1} + R_{n-2}$. La solution se déduit facilement de la récurrence donnant f_n, à savoir

$$R_n = 2f_n.$$

Le nombre d'appels récursifs r_n est donc du même ordre de grandeur que f_n.

5.1.3 Croissance explosive

La suite de Fibonacci croît exponentiellement (cf. exercice 5.1), mais il existe des fonctions récursives qui croissent plus vite que toute itération d'exponentielles. C'est notamment le cas de la fonction de Ackermann, définie pour les entiers $m, n \geq 1$ par l'algorithme récursif suivant[1] :

```
function Ack(m,n :integer) :integer ;
  begin
  if m=1 then
    Ack :=2^n
  else
  if n=1 then
    Ack :=Ack(m-1,2)
  else
    Ack :=Ack(m-1,Ack(m,n-1)) ;
end ;
```

Pour les deux premières valeurs de m on a

$$\texttt{Ack}(1,n) \quad = \quad 2^n$$

et

$$\texttt{Ack}(2,n) \quad = \quad 2^{2^{\cdot^{\cdot^{2}}}} \Big\}\, n$$

[1] Pour raison de simplification, on a modifié la définition originale au niveau de la condition de bord en prenant $\mathrm{Ack}(1,n) = 2^n$ au lieu de $\mathrm{Ack}(1,n) = n + 2$ (cf exercice 5.2).

En particulier,

$$\texttt{Ack}(4,1) = \texttt{Ack}(3,2) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\}^{16},$$

valeur qui dépasse très nettement le nombre estimé d'atomes contenus dans l'univers, à savoir 10^{80}.

***Exercice* 5.1** On note f_n le n-ième nombre de Fibonacci ($f_0 = f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$, voir l'introduction). Soit θ le nombre d'or, c'est-à-dire la racine positive du polynôme $X^2 - X - 1 = 0$.
a) Vérifier que $1 < \theta < 2$.
b) Montrer que pour tout entier $n \geq 1$, on a $\theta^n < f_{n+1} < \theta^{n+1}$. A partir de la relation matricielle (5.2), montrer que $f_{n-1} = \frac{1}{\sqrt{5}}(\theta^n - (-\theta)^{-n})$. Donner une autre méthode de calcul qui utilise plus directement la relation (5.1).
c) Donner une procédure pour sortir à l'écran le triangle de Pascal : le p-ième terme de la n-ième ligne étant l'entier $C(n,p)$ (voir la définition à l'exercice 5.4).

***Exercice* 5.2** Écrire un programme qui construit la table des 25 premiers nombres de Fibonacci.

***Exercice* 5.3** a) Donner un algorithme récursif pour la fonction $n \mapsto 2^n$, puis la fonction $n \mapsto \texttt{Ack}(2,n)$, toutes les deux étant supposées définies sur les entiers naturels.
b) On désigne par AckV(m,n) la véritable fonction de Ackermann qui est définie par :

$$AckV(0,n) = n+1, \quad AckV(m,0) = AckV(m-1,1)$$

et

$$AckV(m,n) = AckV(m-1, AckV(m,n-1)).$$

Montrer que $AckV(4,n) \simeq Ack(2,n)$.

***Exercice* 5.4** Notons $C(n,p)$ le symbole binomial défini, par exemple, comme le coefficient de $x^p y^{n-p}$ dans le développement de $(x+y)^n$, ce qui donne la très classique formule du binôme

$$(x+y)^n = \sum_{p=0}^{n} C(n,p) x^p y^{n-p}.$$

En combinatoire moderne, on note habituellement $C(n,p)$ par $\binom{n}{p}$ et on identifie $(x+y)^0$ avec le polynôme constant égale à 1.
a) Montrer que $C(n,p)$ est le nombre de parties de p éléments incluses dans un ensemble donné de n éléments (résultat bien connu, qui sert parfois de définition pour $C(n,p)$).
b) Calculer $C(n,0)$ et $C(n,n)$.
c) Démontrer la formule de récurrence

$$C(n,p) = C(n-1,p-1) + C(n-1,p) \quad (1 \leq p \leq n).$$

d) En se basant sur les questions précédentes, déterminer un algorithme récursif permettant de calculer $C(n,p)$ (la réponse pourra consister à écrire le programme d'une fonction récursive en Pascal).

5.2 Indécidabilité de l'arrêt

Peut-on tester si un programme définissant une fonction récursive est effectif, c'est-à-dire s'il termine toujours son calcul ? La question peut être élargie à toutes les fonctions écrites en Pascal ou encore être restreinte à une sous-famille donnée. On suppose évidemment que les programmes ne contiennent pas d'erreur. On doit à Gödel et Turing d'avoir démontré, dans un cadre rigoureux, que cette question d'arrêt de programme n'a pas de solution : il n'existe pas de programme qui teste si le programme d'une fonction (récursive) est effectif.

Donnons une indication sommaire sur la démonstration en se plaçant dans le langage Pascal. Un programme est tout simplement une suite de symboles, appelée pour l'occasion mot-programme (écrit suivant des règles syntaxiques précises qui n'est pas utile d'expliciter ici). Admettons maintenant qu'il existe une fonction de décision `D` qui prend en entrée tout mot-programme M est donne en sortie la valeur booléenne $\texttt{D}(M) \in \{\text{true, false}\}$ selon que le programme se termine ou ne se termine pas. Considérons alors un programme du type suivant :

```
programme absurdie ;
    var
      M :string ;
  function D(M :string) :boolean
    begin
      ...
    end ;

begin
  read(M) (* M est un mot-programme *)
  while D(M) do
    ;
  writeln('tilt') ;
end.
```

Si `read(M)` lit le mot-programme `A` du programme `absurdie` et que `D(A)`=true, alors le programme `absurdie` boucle indéfiniment. Si `D(A)`=false, alors le programme `absurdie` se termine. Il y a donc une contradiction sur la valeur de `D(A)` à moins que `D` ne soit pas une fonction programmable en Pascal. En d'autres termes, il y a indécidabilité de l'arrêt des programmes : il n'existe pas de fonction Pascal qui prenne en argument le mot-programme de tout programme et détermine si le programme en question est effectif.

Les problèmes d'indécidabilité de ce genre sont nombreux et variés ; leur traitement rigoureux passe par l'introduction des machines de Turing – que l'on peut simuler en langage Pascal – et le résultat fondamental d'indécidabilité à prouver est simplement celui de déterminer si une machine de Turing donnée s'arrête (*i.e.* exécute un nombre finie d'instructions) pour un mot (string) d'entrée donné.

***Exercice* 5.5** Un programme est dit effectif si, quel que soit les données d'entrée, le programme se termine.
a) écrire un programme effectif et un programme non effectif. L'arrêt ou l'absence d'arrêt du programme sera visualisé sur la sortie standard.
b) Montrer qu'il n'y a qu'un nombre au plus dénombrable de programmes Pascal.
c) Soit E l'ensemble des fonctions définies sur $\mathbf{N}$ et à valeurs dans $\{0, 1\}$ au moyen de programmes récursifs. Montrer que E est infini.

***Exercice* 5.6** a) écrire un programme qui affiche la suite des entiers successifs n en commençant par $n = 1$.
b) Que se passe-t-il lorsque n =maxint ?
c) Modifier (ou changer) le programme de manière à pouvoir afficher les entiers plus grands que maxint et cela jusqu'à 2^5.maxint.

5.3 De l'usage de la récursivité

Cette partie fournit un catalogue de procédures récursives au sens de l'introduction. Il n'est évidemment pas complet. Les exemples choisis ont pour but de montrer la grande diversité des applications de la récursivité. Notons que certains exemples utiliseront des fonctions graphiques dédiées que nous emprunterons à la bibliothèque QuickDraw du Machintosh[2]. Nous indiquerons ultérieurement (cf. les travaux pratiques) comment tracer des courbes sur une plateforme Linux ou Unix avec GnuPascal combiné avec GnuPlot.

5.3.1 Le PGCD

Revenons sur l'algorithme du pgcd étudié au chapitre 1, sous-section 1.4.4. Celui-ci était séquentiel.

On peut cependant l'écrire sous une forme récursive :

[2]Ce qui nous donnera l'avantage de vérifier tranquillement ces programmes sur notre mac.

Algorithme *PGCDr*(a, b)
entrer les entiers a et b.
sortir pgcd(a, b).
début
1. SI $b = 0$ ALORS *PGCDr*$= a$;
2. SINON FAIRE *PGCDr*$(b, a$ MOD $b)$;
3. ECRIRE('pgcd(a, b)=' *PGCDr*(a, b)).
fin.

Traduit en Pascal, cela donne simplement

```
program pgcd ;
  var
    a,b : integer ;
  function pgcdR(a,b :integer) :integer ;
  begin
    if b=0 then
      pgcdR :=a ;
    else
      pgcdR(b,a mod b) ;
  end ;
begin
  while not eof do
  begin
  readln(a,b) ;
  writeln('pgcd(' :5, a :2,',' :1, b :2,') =' :3, pgcdR(a,b) :2) ;
  end ;
end.
```

Au passage, on a pris le temps de formater la sortie. L'instruction "**while not** eof **do**" permet, un fois le pgcd calculé, d'effectuer un nouveau calcul de pgcd ; on n'a pas prévu d'instruction pour 'quitter' le programme !

Analysons le temps d'exécution de `pgcdR` en évaluant le nombre d'appel $r(a, b)$ de la fonction récursive `pgcdR` au cours du calcul de pgcd(a, b). On a

5.1 ***Lemme.*** *Si $a > b \geq 0$ et $r(a, b) = k \geq 1$, alors $a \geq f_k$ et $b \geq f_{k-1}$ où f_n désigne le n-ième nombre de Fibonacci (cf. (5.1.2)).*

Démonstration. La preuve se fait par induction sur k. Si $k = 1$, alors b n'est pas nul mais a **mod** b l'est puisqu'il n'y a qu'un appel à `pgcdR`. D'où $a > b \geq 1 = f_1 = f_0$, ce qui prouve le lemme pour $k = 1$. Supposons-le démontré pour $k - 1 \geq 1$. On a toujours $b > 0$ et le premier appel récursif remplace a par b et b par a **mod** b. Par hypothèse de récurrence,

$$b \geq f_{k-1} \quad \text{et} \quad a - b\lfloor a/b \rfloor \geq f_{k-2}.$$

Mais $\lfloor a/b \rfloor \geq 1$, d'où $a \geq f_{k-2} + b \geq f_{k-2} + f_{k-1} = f_k$, ce qui prouve le lemme pour k. ∎

De ce lemme résulte immédiatement le théorème suivant, dû à Lamé :

5.2 ***Théorème.*** *Soient les entiers a, b vérifiant les inégalités $a > b \geq 0$. Si $b < f_k$ (= k-ième nombre de Fibonacci), alors le calcul de* pgcd(a, b) *par la fonction récursive* `pgcdR` *nécessite au plus k appels récursifs.* ∎

5.3.2 Tours de Hanoï

Le jeu des tours de Hanoï consiste en trois piquets notés 1, 2, 3 et n disques de différents diamètres, percés en leur centre et pouvant s'enfiler sur les piquets. Au départ, les disques

sont empilés sur le piquet 1 par diamètres décroissants, formant ainsi une tour pyramidale. Le but du jeu est de déplacer ces disques sur le piquet 3 en respectant les règles suivantes :

1. à chaque coup, un seul disque au sommet d'une tour est déplacé pour être enfilé sur un autre piquet ;
2. l'empilement sur chaque piquet, s'il n'est pas vide, forme toujours une tour pyramidale (aucun disque n'est placé au dessus d'un disque plus petit) ;
3. les trois piquets peuvent être utilisés pour déplacer les disques.

Montrons, par induction sur n, qu'il existe toujours une solution. Si $n = 1$, celle-ci est banale. Si $n = 2$, c'est tout aussi simple. Pour $n = 3$, la solution s'obtient assez vite et il n'y a pas beaucoup de choix : on commence par déplacer les deux disques du haut sur le piquet 2 avec les déplacements de piquet à piquet selon la séquence $1 \to 3$, $1 \to 2$, $3 \to 2$. Le piquet 1 ne contient plus que le grand disque, d'où son déplacement $1 \to 3$, puis on déplace la tour du piquet 2 sur le piquet 3.

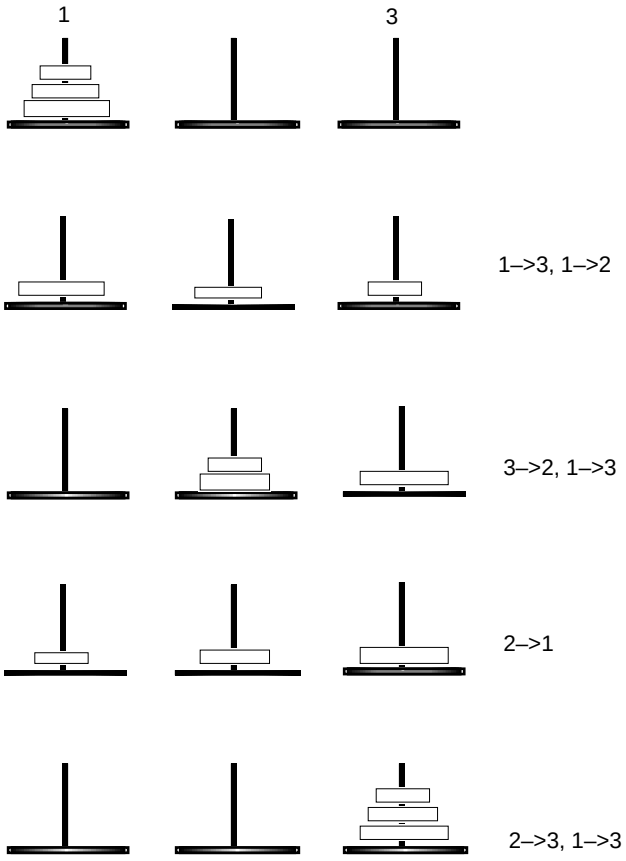


Figure 5.2. Tour de Hanoï, cas $n = 3$: déplacements de trois disques, du piquet 1 vers le piquet 3.

Supposons maintenant le problème résolu pour $n - 1$ disques. Evidemment, il est résolu indépendamment des numéros de piquet. On peut donc partir de n disques empilés sur le piquet a ($\in \{1, 2, 3\}$) pour les enfiler, en respectant les règles, sur le piquet b ($\in \{1, 2, 3\} \setminus \{a\}$) le troisième piquet ayant alors le numéro $6 - a - b$. L'algorithme suivant donne comme solution :

1. déplacer les $n - 1$ disques supérieurs du piquet a sur le piquet $6 - a - b$;
2. déplacer le plus grand disque restant sur le piquet b ;

3. déplacer les $n-1$ disques du piquet $6-a-b$ sur le piquet b ;

Cet algorithme conduit à la procédure suivante :

```
procedure TourdeHanoi(n :integer ; a ,b :integer) ;
  begin
  if n > 0 then
    begin
    TourdeHanoi(n-1, a, 6-a-b) ;
    writeln(a :3,'->' :3,6-a-b :3) ;
    TourdeHanoi(n-1, 6-a-b, b) ;
    end ;
end ;
```

5.3.3 Tri-rapide (Quicksort)

L'algorithme suivant associe un tri séquentiel partiel au sein d'un processus récursif. Introduite en 1960 par C. A. R. Hoare, cette méthode de tri est probablement la plus utilisée. Sa complexité est assez bien comprise : pour un fichier aléatoire de taille N, ce tri est effectué en $N \log N$ opérations (à la multiplication près par une constante > 0), ce qui est très performant. Il s'agit, bien sûr, d'une valeur en moyenne, le pire des cas étant en $\mathcal{O}(N^2)$ opérations.

Le tri-rapide (appelé Quicksort par les anglo-saxons) s'appuie sur une stratégie visant à diviser pour conquérir et sur le principe suivant : étant donnée une suite de nombres a_n, $1 \leq n \leq N$, à ranger par ordre croissant, un terme a_j est déjà bien placé si $a_k \leq a_j$ pour tout $k < j$ et $a_j \leq a_\ell$ pour tout $\ell < k$. Evidemment, si tous les termes de la suite sont bien placés, alors la suite est croissante.

L'algorithme, sur une suite finie de nombres, procède donc de la manière suivante :

1. Un terme b est choisi dans la suite ;
2. le reste des termes est partagé en deux parties : la partie, notée $G(b)$, formée des termes inférieurs ou égaux à b et la partie complémentaire, notée $D(g)$;
3. indexer la suite de manière à ce que b soit à la bonne place ;
4. recommencer l'opération de partage et d'indexation (étapes précédentes) pour la suite obtenue à partir de $G(b)$ (resp. $D(b)$) tant que celle-ci ne se réduit pas à 1 termes.

En pratique, le tri se fait sur un tableau `a[1..N]` de `N` *clefs*. Soit `a[g..d]` le sous-tableau à trier (de limite gauche `g` et de limite droite `d`) ; l'algorithme Quicksort repose sur procédure récursive suivante :

```
procedure QuickSort(g,d :integer) ;
  var m : integer ; b : clef ;
  begin
  if g < d then
    begin
    b := a[k] ; (* exemple du choix d'une valeur pour partitionner le tableau *)
    m := position(g,d ;b)
    QuickSort(g,m-1) ;
    QuickSort(m+1,d) ;
    end ;
  end ;
```

La fonction *position(*`g,d ;b`*)* prend en entrée le tableau `a[g..d]` et la clef `b`, range le tableau de manière à bien placer `b` et renvoie l'indice `m` qui positionne `b` après ce rangement. On

obtient ainsi un tableau `a` tel que `a[m]=b`, les clefs `a[g]`,...,`a[m-1]` sont plus petites ou égales à `a[m]` et les clefs `a[m+1]`,...,`a[d]` sont plus grandes (ou égales) à `a[m]`. Il y a bien sûr plusieurs manières de procéder pour calculer *position(*`g,d ;b`*)* et au lieu de prendre `b` = `a[g]`, on aurait pu prendre toute autre valeur de clef. Notons que le choix `b=a[g]` conduit à une impasse lorsque `b` est le plus petit des éléments `a[i]`, `g` ≤ `i` ≤ `d` (le programme s'arrête avec l'appel de `QuickSort(g,g-1)`. C'est en particulier le cas lorsque le tableau est déjà trié ! Oublions pour l'instant ces cas exceptionnels et déterminons un premier programme pour *position(*`g,d ;b`*)* en prenant `b` = `a[g]` :

```
m := g ;
for i := g+1 to d do
  if a[i] < b then
    begin
    m := m+1 ;
    échanger a[m] et a[i] ;
  end ;
échanger a[m] et a[g] ;
```

Sur un exemple simple, traçons les valeurs successives de `i` et `m`, avec les états correspondants du tableau, après exécution de l'instruction conditionnelle pilotée par **if** :

valeurs initiales : `m`=1, `b`=2
pour `i` = 1 à 7

	i	m	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
début									
	2	1	2	3	1	7	5	0	1
	3	2	2	**1**	3	7	5	0	1
	4	2	2	**1**	3	7	5	0	1
	5	2	2	**1**	3	7	5	0	1
	6	3	2	**1**	**0**	7	5	3	1
	7	4	2	**1**	**0**	**1**	5	3	7
fin									
dernier échange									
	7	1	**1**	**1**	**0**	2	5	3	7

Figure 5.3. Partition de `a[1..7]` autour de la valeur `a[1]=2` : étapes successives menant à la partition.

Ce programme introduit les indices `m` et `i` ; l'indice `i` parcourt les entiers successifs de `g+1` à `d` et `m` progresse par pas de 0 ou 1, de manière à ce qu'à tout instant, `a[j]` < `b` si `g` < `j` ≤ `m` et `b` ≤ `a[j]` si `m` < `j` < `i`.

écrivons maintenant la procédure complète implémentant ce choix de la fonction *position( )* :

```
procedure TriRapide(g,d :integer) ;
  var m,i : integer ; b, x : clef ;
  begin
  if g < d then
    begin
    b := a[g] ;
    m := g ;
    for i := g+1 to d do
      if a[i] < b then
```

```
        begin
        m := m+1 ;
        x := a[m] ; a[m] := a[i] ; a[i] := x ;
        end ;
    x := a[m] ; a[m] := a[g] ; a[g] := x ;
    TriRapide(g,m-1) ;
    TriRapide(m+1,d) ;
    end ;
end ;
```

Si, au cours de la procédure, `b` = `a[g]` est le plus petit des termes du tableau `a[g..d]`, le tri s'arrête. Une manière de le poursuivre serait de tester la valeur de sortie de *position(*`g,d ;b`*)*. Si elle est égale à `g` alors on envoie sur `QuickSort(g+1,d)`. Avec cette solution, si le tableau est déjà trié (ordre croissant des clefs), ces renvois se font pour `m` = 1, 2,..., `N-1` et pour chaque valeur k de `m`, l'algorithme fait `N` $-k$ comparaisons de clefs, soit un total de $\sum_{k=1}^{N-1}(\texttt{N}-k) = \frac{\texttt{N(N-1)}}{2}$ comparaisons. Notons que si le tableau est dans l'ordre décroissant, le nombre de comparaisons pour mettre les clefs dans l'ordre croissant est le même que pour le tableau déjà dans l'ordre. Dans ces deux cas particuliers, l'algorithme est au maximum de sa complexité.

Dans l'algorithme Quicksort classique, la fonction *position( )* traite de manière symétrique le tableau `a[g..d]`, contrairement au cas précédente. La méthode consiste ici à utiliser deux indices, le premier, `i` partant de 1 avec un pas +1 et le second `j` partant de `N` avec un pas −1. Alternativement, on recherche les valeurs de `a[i]` qui dépassent `b` et les valeurs de `a[j]` inférieures à `b` afin de les échanger entre elles, jusqu'à épuisement du tableau. Cela donne

```
procedure QuickSort(g,d :integer) ;
  var i,j : integer ; b, x : clef ;
  begin
  if g < d then
    begin
    b := a[d] ;
    i := g-1 ;
    j := d ;
    repeat
      repeat i := i+1 until a[i] >= b ;
      repeat j := j-1 until a[j] <= b ;
      x := a[i] ; a[i] := a[j] ; a[j] : =x ;
    until j <= i ;
    a[j] := a[i] ; a[i] := a[d] ; a[d] := x ;
    QuickSort(g,i-1) ;
    QuickSort(i+1,d) ;
    end ;
  end ;
```

On peut observer que l'algorithme s'arrête si `b` est la plus petite ou plus grande clef du tableau `a[g..d]`. Une méthode pour éviter cette situation et qui améliore les performances de l'algorithme (notamment pour un tableau déjà trié), consiste à choisir trois éléments du tableau `a[g..d]` et prendre comme élément b pour partitionner celui qui est entre les deux autres. Par exemple, on peut choisir `a[g]`, `a[d]` et `a[m]` avec `m` $= \lfloor \frac{g+d}{2} \rfloor$.

Performance en moyenne de l'algorithme Quicksort

La partition idéale dans Quicksort est celle qui divise le tableau en deux parts égales. Dans ce cas, si C_N désigne le nombres de comparaisons pour un tableau de taille N, alors

$$C_N = 2C_{N/2} + N, \tag{5.4}$$

où $2C_{N/2}$ représente le nombre de comparaisons pour le tri des deux sous-tableaux résultant de la partition auquel s'ajoute N pour tenir compte des comparaisons qui portent sur les divers éléments du tableau. Dans le meilleurs des cas et pour $N = 2^n$, cette situation idéale se retrouve pour toutes les partitions successives. L'équation (5.4) devient alors

$$\frac{C_{2^n}}{2^n} = \frac{C_{2^{n-1}}}{2^{n-1}} + 1.$$

Avec $C_1 = 0$ ($n = 0$) un examen rapide donne

$$\frac{C_{2^n}}{2^n} = n.$$

Revenons à N et remplaçons le par la plus petite puissance de 2 qui dépasse N ; alors, le calcul précédent montre que dans le meilleur des cas $C_N \simeq N \log_2 N$ (pour N tendant vers l'infini).

Si on effectue la procédure Quicksort sur un tableau T représentant une permutation aléatoire des N premiers entiers (le modèle probabiliste sous-jacent est donc l'ensemble des permutations de N éléments munit de l'équiprobabilité), on peut montrer que l'espérance C_N de la variable aléatoire donnant le nombre de comparaisons pour trier le tableau T vérifie la relation

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}) \quad (N \geq 2)$$

avec $C_0 = C_1 = 0$ (pas de tri pour un tableau vide ou réduit à un élément). La solution de cette récurrence est assez facile. Tout d'abord, $\sum_{1 \leq k \leq N} C_{k-1} = \sum_{1 \leq k \leq N} C_{N-k}$, ce qui conduit à l'expression plus simple

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \leq k \leq N} C_{k-1} \quad (N \geq 2)$$

d'où l'on déduit, en considérant NC_N et $(N-1)C_{N-1}$,

$$NC_N - (N-1)C_{N_1} = N(N+1) - (N-1)N + 2C_{N-1}.$$

Après simplification et division par $N(N+1)$ la récurrence devient

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

et en itérant,

$$\frac{C_N}{N+1} = \frac{C_2}{3} + \frac{2}{N+1} + \frac{2}{N} + \ldots + \frac{2}{4}.$$

Classiquement $\sum_{k=1}^{N} \frac{1}{k} \simeq \int_1^N \frac{dx}{x} = \log N$, d'où

$$C_N \simeq (2 \log 2) \log_2 N.$$

Ainsi, le nombre moyen de comparaisons est environ 1,38 ($= 2 \log 2$) fois celui obtenu dans le meilleur des cas. Le tri Quicksort se montre donc très performant en moyenne, ce qui explique son utilisation fréquente lorsque les tableaux à trier sont peu structurés.

***Exercice* 5.7** a) Décrire l'état des appels successifs de la fonction récursive `pgcdR` lors du calcul de `pgcdR`(f_k, f_{k-1}) (f_k désignant le k-ième nombre de Fibonacci (cf (5.1.2)).
b) En déduire que la borne du théorème 5.2 est optimale.
c) Que vaut pgcd(f_k, f_{k-1}) ?
d) On souhaite obtenir le résultat du théorème 1.2, chapitre 1, c'est-à-dire de construire une solution de l'équation de Bezout

$$\text{pgcd}(a, b) = xa + yb. \tag{5.5}$$

Étendre l'algorithme `pgcdR` de manière à résoudre dans $\mathbf{Z}$ l'équation (5.5) par une procédure $PGCD\_ETENDUE(a, b)$ qui prend en entrée une paire d'entiers positifs ou nuls (a, b) et retourne un triplet $(\text{pgcd}(a, b), x, y)$ vérifiant (5.5).
e) Quel est le résultat de $PGCD\_ETENDUE(f_k, f_{k-1})$?
f) écrire un programme qui calcule `ppcm`(a, b) en utilisant `pgcdR`.

***Exercice* 5.8** a) écrire complètement l'algorithme récursif `TourHanoi` du problème des tours de Hanoï ; celui-ci prenant en entrée le nombre n de disques et donnant en sortie la suite des déplacements successifs des disques, de piquet à piquet.
b) On note d_n le nombre de déplacements de disques effectués au cours de l'exécution du programme pour n disques. Calculer d_1, d_2, d_3, et montrer que $d_n = 2d_{n-1} + 1$ si $n \geq 2$.
c) Donner une expression simple pour d_n.

***Exercice* 5.9 [Continuation de l'exercice précédent]**. On indice les disques de 1 à n, le k-ème disque, noté D_k étant supposé de diamètre strictement plus grand que ceux des disques D_i pour $1 \leq i < k$. Ensuite, on partage l'ensemble de ces disques en trois sous-ensembles (disjoints) E_1, E_2, E_3, les disques de chaque E_a étant ensuite enfilés sur le piquet a de manière à former une pyramide. La construction ainsi obtenue est dite "configuration permise". Réciproquement, à toute configuration permise est associée une (et une seule) partition des disques en trois sous-ensembles donnant la configuration permise dont on est parti. Pour n donné, on note c_n le nombre de configurations permises.
a) Calculer c_1, c_2 et c_3, puis montrer par récurrence que $c_n = 3^n$.
b) Montrer qu'il est possible de passer de toute configuration permise à toute autre configuration permise, en déplaçant les disques dans le respect des 3 règles énoncées pour le problème des tours de Hanoï (on ne demande pas un programme comme preuve).
c) A toute configuration permise P, déterminée par la partition $\{E_1, E_2, E_3\}$, on associe l'entier

$$N(P) := \sum_{k=1}^{n} p_k 3^{k-1}$$

où l'application $k \mapsto p_k$ de $\{1 \ldots, n\}$ dans $\{0, 1, 2\}$ est définie par

$$p_k = a - 1 \Longleftrightarrow D_k \in E_a.$$

Montrer que l'application $P \mapsto N(P)$ est une bijection de l'ensemble des configurations permises (de n disques) sur l'ensemble $\{0, \ldots, 3^n - 1\}$.
d) Examiner la construction de l'application inverse $T(\cdot)$ de $N(\cdot)$. Déterminer $T(0)$, $T(3^n - 1)$ et $T(\frac{3^n-2}{2})$, puis écrire un algorithme (en Pascal) qui à tout entier p entre 0 et $3^n - 1$ (n fixé) donne son développement en base trois (avec n chiffres) et sort sur trois colonnes l'empilement des disques (représentés par leurs indices). Par exemple, pour $n = 6$ et $p = 123$ ($= 0.3^5 + 1.3^4 + 1.3^3 + 1.3^2 + 2.3^1 + 0.3^0$) en entrée, on obtient le développement

0 1 1 1 2 0

et l'empilement

```
    3
1   4
6   5   2
```

(disques 1 et 6 sur le piquet 1, disques 3, 4 et 5 sur le piquet 2, disque 2 sur le piquet 3).

***Exercice* 5.10** Montrer que le tri Quicksort n'est pas stable.

***Exercice* 5.11** a) Compléter la procédure `TriRapide( )` de manière à éviter les cas d'arrêts prématurés en suivant les indications fournies dans le cours.
b) Utiliser le programme obtenu pour trier d'une table de 10 nombres produits de manière aléatoire (voir l'exercice 4.2).

***Exercice* 5.12** a) Considérer la fonction *position( )* correspondante à la procédure `Quicksort( )` et l'appliquer au tableau utilisé dans le cours pour illustrer la fonction *position( )* introduite dans la procédure `TriRapide( )`. On donnera les états successifs du tableau `a` qui mènent à sa partition autour de la valeur `b=a[d]`.
b) Compléter la procédure `QuickSort( )` suivant les indications du cours (consistant à choisir comme valeur de partition celle qui est intermédiaire entre les valeurs des clefs extrêmes `a[g]`, `a[d]` et de la clef `a[m]` pour `m` $= \lfloor \frac{g+d}{2} \rfloor$).

5.4 Courbes fractales

5.4.1 Notion de courbe et fonctions graphiques

Dans le plan (euclidien) $\mathbf{P}$, une courbe est, par définition, l'image d'une application continue $f : [0, 1] \to \mathbf{P}$. Le point $f(0)$ est l'origine de la courbe, par opposition à f(1) qui en est l'extrémité. La courbe est simple si f est injective, ce qui permet d'orienter la courbe de manière naturelle de l'origine vers l'extrémité. La courbe est de classe C^1 si, en utilisant

un repère orthonormé du plan, les fonctions coordonnées définissant f dans ce repère sont dérivables, de fonctions dérivées continues. L'application f est, par définition, une paramétrisation de la courbe.

Bien que la notion particulière de courbe fractale n'ait pas encore reçue de signification qui fasse l'unanimité, nous retiendrons la définition suivante :

> *une courbe est fractale si elle peut se reconstruire à partir d'images d'elle-même obtenues par des applications contractantes.*

A l'étudier de près, cette définition permet de classer un segment de droite comme une courbe fractale. On ne la rejettera pas pour autant, considérant normal que certaines courbes banales puissent répondre à définition aussi générale. En quelque sorte, la ligne droite est un cas pathologique parmi les courbes fractale.

Image fractale. Dans le plan (euclidien, identifié à $\mathbf{R}^2$ par le choix d'un repère orthonormé), une image (noir-et-blanc) est, par définition, une partie compacte (ou, ce qui est équivalent, une partie fermée bornée). En informatique, on s'intéresse plus spécifiquement à l'ensemble $\mathcal{J}$ des images contenues dans un carré donné Q. Sans nuire à la généralité de l'étude, on peut supposer que $Q = [0,1] \times [0,1]$. Notons Q_0 la partie intérieur de Q, c'est-à-dire $]0,1[\times]0,1[$. Nous simplifions l'étude en ne considérant que des applications affines f de $\mathbf{R}^2$ telles que $f(Q) \subset Q_0$. Ainsi f est de la forme $f(X) = A + L(X)$ où L est une transformation linéaire de $\mathbf{R}^2$ (donc définie par une matrice 2×2). La condition d'inclusion $f(Q) \subset Q_0$ implique la propriété de contraction suivante

$$\forall\, X \in \mathbf{R}^2\ ,\ ||L(X)|| \leq \rho ||X||$$

où ρ est un nombre réels positifs strictement plus petit que 1 et $||\cdot||$ dénote la norme max sur $\mathbf{R}^2$ dont on rappelle la définition :

$$||\begin{pmatrix} x \\ y \end{pmatrix}|| = \max\{|x|, |y|\}.$$

Considérons maintenant N transformations affines $f_1, \ldots, f_N$ du plan telles que $f_i(Q) \subset Q_0$ pour tous les indices i. On définit la transformation $F : \mathcal{J} \to \mathcal{J}$ sur l'espace des image $\mathcal{I}$ par

$$F(K) = f_1(K) \cup \ldots \cup f_N(K).$$

On dit que F est un système itératif de contractions affines du plan (localisé dans Q). Le théorème suivant, dont la démonstration n'est pas au programme, justifie les définitions précédentes :

5.3 *Théorème.* *Un système itératif F de contractions du plan admet un point fixe et un seul. Cette image point fixe est notée* $\mathrm{Fix}(F)$.

Le théorème est constructif en ce sens que pour tout $\epsilon > 0$, il détermine un entier $m(\varepsilon)$ tel que pour toute image K dans Q et tout $n \geq m(\varepsilon)$ on a

$$\mathrm{Fix}(F) \subset V_\epsilon(F^n(K)) \ \&\ F^n(K) \subset V_\epsilon(\mathrm{Fix}(F)),$$

où, pour toute partie A de $\mathbf{R}^2$, on définit

$$V_\epsilon(A) = \{x \in \mathbf{R}^2\,;\ \exists\, a \in A,\ ||x - a|| \leq \epsilon\}.$$

Nous aurons besoin de tracer des courbes et notamment des courbes fractales. Pour cela, il est nécessaire de faire appel à une bibliothèque spécialisée de fonctions graphiques. Celles-ci dépendent de la plateforme sur laquelle est exécuté le programme. Nous choisissons ici la bibliothèque QuickDraw sur Macintosh à cause de ses fonctions graphiques simples et

intuitives. Dans QuickDraw il y a la notion de point courant (h,v), de plume (ou crayon) avec une taille et d'une couleur par défaut, modifiables. Pour dessiner on peut déplacer la plume à partir de sa position courante, soit en la déplaçant (sans tracer) soit en la déplaçant avec tracé, au moyen des fonctions suivantes.

`MoveTo(x,y)` : déplace la plume aux coordonnées absolue `x, y` ;
`Move(dx,dy)` : déplace la plume de sa position courante `(h,v)` à la position `(h+dx,v+dy)` ;
`LineTo(x,y)` : trace une ligne du point courant au point de coordonnées `(x,y)` ;
`Line(dx,dy)` : trace une ligne depuis la position courante `(h,v)` de la plume à la position `(h+dx,v+dy)` ;
`DrawLine(x,y,z,t)` : trace une ligne du point `(x,y)` au point `(z,t)`.

Cela suffira pour l'instant mais précisons encore que l'unité de longueur est le pixel et que le référentiel absolue, lié à l'écran, a son origine située au coin supérieur gauche de l'écran ; l'axe des abscisses est horizontal, orienté de gauche à droite et l'axe des ordonnées est vertical, orienté de haut en bas. Il convient aussi de disposer d'une fenêtre graphique dont les axes de référence sont analogues à ceux de l'écran avec pour origine le coin supérieur gauche de la fenêtre. Pour construire cette fenêtre, on commence par enregistrer un rectangle :

`SetRect(S,g,h,d,b)` : définit un rectangle nommé S avec les coordonnées des cotés (gauche, haut, droite, bas). En fait `Rec` est de type record (de même que pour `Point`) ;
`SetDrawingRect(S)` : affiche une fenêtre de la taille de S.

C'est fait. Pour dessiner un rectangle, on utilise une fonction spécifique :

`FrameRect(R)` : dessine le rectangle R (dans une fenêtre) ;

Ne pas oublier :

`ShowDrawing` : ouvre la fenêtre Drawing (sortie graphique) pour voir ce qu'on dessine (avec `ShowText` on affiche la fenêtre Texte, qui est la sortie standard par défaut). L'épaisseur du trait est réglée par `PenSize(dx,dy)` ; par défaut `(dx,dy)=(1,1)`.

Enfin, il peut être utile d'écrire du texte :

`DrawChar(c)` affiche, dans la fenêtre graphique et au point courant, le caractère `c` ;
`DrawString(s)` fait la même chose avec la chaîne de caractère `s`.

Le programme suivant montre sur un exemple simple (mais utile pour la suite) comment afficher une fenêtre pour y tracer un carré et une ligne brisée A_0 en forme de C :

```
program Hilbert1 ;
  const
    X0 = 100 ;
    X1 = 500 ;
    Y0 = 100 ;
    Y1 = 500 ;
  var
    x, y, dx, dy : integer ;
    r, s : Rect ;
    i : integer ;
begin
  SetRect(s, X0 - 50, Y0 - 50, X1 + 150, Y1 + 150) ;
  SetDrawingRect(s) ;
  SetRect(r, X0, Y0, X1, Y1) ;
  ShowDrawing ;
  FrameRect(r) ;
```

```
    Moveto(X0 + 300, Y0 + 100) ;
    PenSize(2, 2) ;
    Line(-200, 0) ;
    Line(0, 200) ;
    Line(200, 0) ;
    Moveto(400, 550) ;
    DrawString('Hilbert1') ;
end.
```

Ce programme, traité par THINK Pascal, affiche sur l'écran du Mac une fenêtre où est tracé le rectangle r et la courbe A_1. La figure 5.4 présente une copie partielle de l'écran (rappelons que l'origine de l'écran est en haut à gauche (HG) et l'unité de longueur est le pixel). Le sommet HG de la fenêtre construite par `SetDrawingRect(s)` (appelée "Drawing" et dont le design est géré par la bibliothèque des interfaces) a donc pour coordonnées absolues (50,50). Le sommet HG (Haut-Gauche) du rectangle `r`, appelé par `FrameRec(r)`, a pour coordonnées absolues (150,150) (par rapport à l'écran) et donc ses coordonnées relatives à la fenêtre "Drawing" sont `(X0,Y0)`. Ainsi, les paramètres de `r` sont passés en valeurs relatives à la fenêtre `s` (ce qui est légitime puisque tout déplacement de la fenêtre "Drawing" déplace identiquement son contenu. De même, le point origine de la courbe est donnée en coordonnées relatives à la fenêtre "Drawing", soit (400,200) (400 pixel du bord gauche et 200 pixels du bord supérieur). La courbe est tracée dans le sens inverse des aiguilles d'une montre. Tous ces détails ne sont pas anodins et doivent être bien compris.

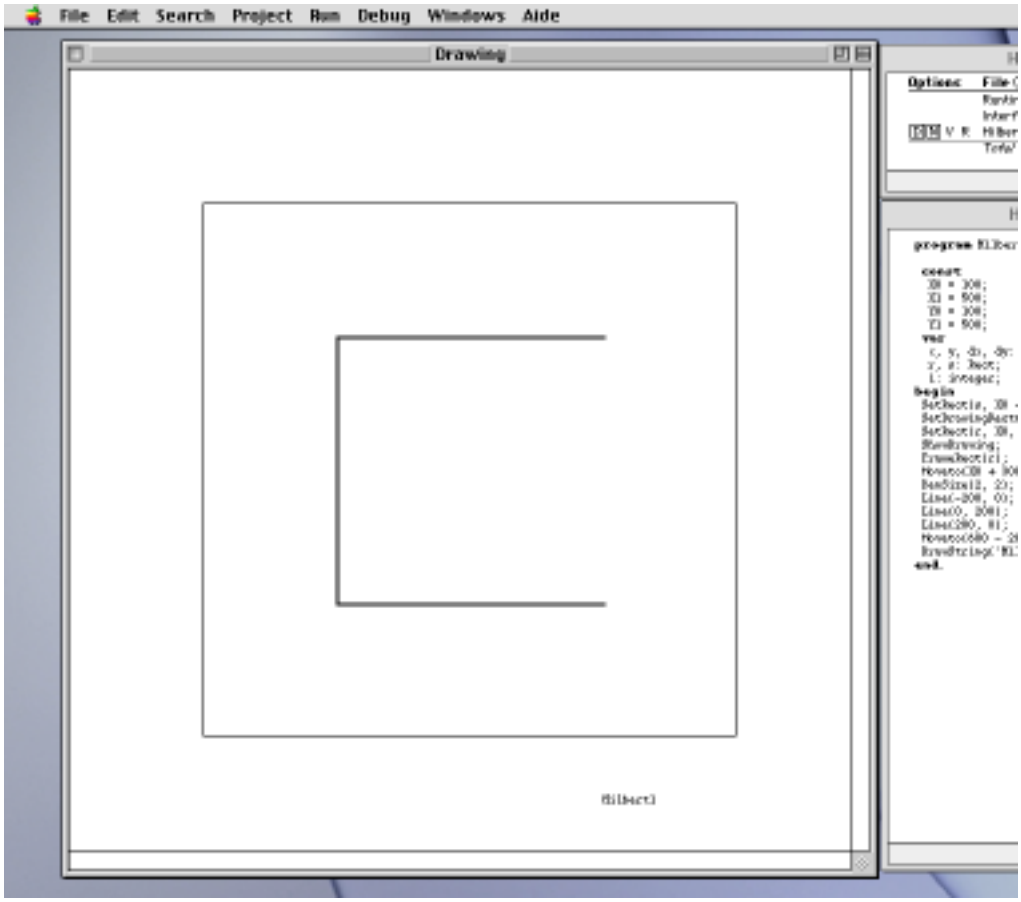


Figure 5.4. Création d'une fenêtre et tracés de courbes au moyen de la bibliothèque QuickDraw d'un Macintosh.

5.4.2 Courbes de Hilbert

Le but avoué est de construire un courbe qui remplisse tout le carré $Q := [-1/2, 1/2]^2$. Une telle courbe peut s'obtenir comme limite d'une suite de courbes en lignes brisées A_n ($\subset Q$). Plus précisément, chaque A_n est image d'une application continue $f_n : [0, 1] \to Q$ et la suite $(f_n)_n$ converge uniformément. La limite $f : [0, 1] \to Q$ des f_n est donc continue et donne une paramétrisation de la courbe cherchée.

Nous décrivons la méthode de Hilbert qui consiste à choisir une courbe initiale A_0 (première courbe de Hilbert, celle déjà tracée à la figure 5.4) et à définir une transformation T qui associe à toute courbe α dans Q, une autre courbe dans Q construite à partir de quatre copies réduites de moitié de α que l'on transforme en effectuant des translations, rotations et symétries convenables et que l'on relie par des segments. Finalement, la n-ième courbe (approchée) A_n de Hilbert se déduit de la précédente par la relation de récurrence

$$A_n = T(A_{n-1}).$$

Nous n'entrerons pas dans la construction des applications f_n elle-mêmes. Celle-ci est sans mystère mais assez technique car il faut s'assurer de la convergence uniforme de la suite $(f_n)_n$. La figure 5.5 démonte le mécanisme de définition de T sur une courbe α.

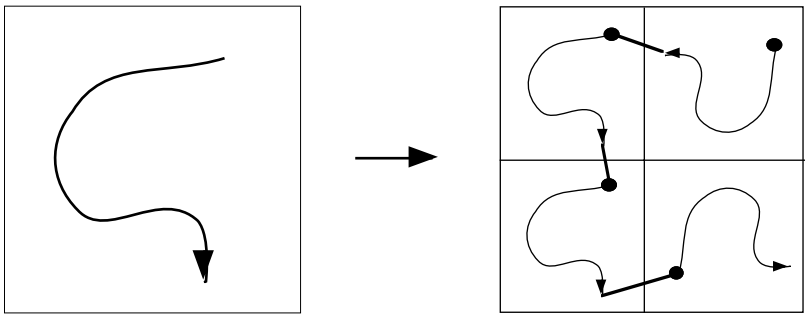


Figure 5.5. Image d'une courbe α par la transformation T

Deux itérations de T sur α donnent alors la courbe suivante :

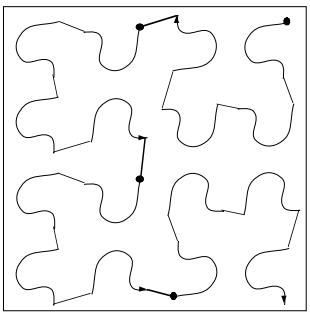


Figure 5.6. Tracé de la courbe $T^2(\alpha)$.

Pour expliciter la transformation T, introduisons :

– la rotation R de $\pi/2$ ($\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} -y \\ x \end{pmatrix}$),

– la symétrie d'axe vertical S ($\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} -x \\ y \end{pmatrix}$).

A toute courbe α de Q, d'origine a et d'extrémité b, on associe les courbes

$$\beta = RS(\alpha), \quad \gamma = R^2(\alpha) \quad \text{et} \quad \delta = SR(\alpha).$$

Si on prend pour α la courbe C de la figure 5.5, on obtient les quatre courbes suivantes :

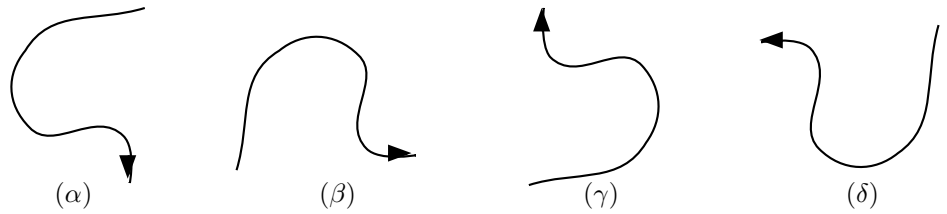


Figure 5.7. Les quatre courbes de base dans la construction des courbes de Hilbert

Il est utile de remarquer que l'ensemble des quatre transformations[3] I (identité), RS, SR $(= -SR)$ et R^2 $(= -I)$, muni de la loi de composition des applications, forme un groupe commutatif, d'élément neutre I, les autres éléments étant d'ordre 2 (c'est donc le groupe de Klein). La loi du groupe est donnée par la table de multiplication suivante :

	I	RS	SR	−I
I	I	RS	SR	-I
RS	RS	I	-I	SR
SR	SR	-I	I	RS
-I	-I	SR	RS	I

Figure 5.8. Le groupe de Klein.

La première conséquence de cette structure algébrique est que si U désigne une quelconque des transformations de ce groupe, considérée comme une application $X \mapsto U(X)$ sur l'ensemble des images de Q, elle induit une permutation sur $\{\alpha, \beta, \gamma, \delta\}$, pour peu que α contienne trois points non alignés.

Introduisons les homothéties H_i, $i = 1, 2, 3, 4$ définies par

$$\begin{aligned}
H_1\begin{pmatrix} x \\ y \end{pmatrix} &= \frac{1}{4} SR \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1/4 \\ 1/4 \end{pmatrix}, \\
H_2\begin{pmatrix} x \\ y \end{pmatrix} &= \frac{1}{4} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} -1/4 \\ 1/4 \end{pmatrix}, \\
H_3\begin{pmatrix} x \\ y \end{pmatrix} &= \frac{1}{4} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} -1/4 \\ -1/4 \end{pmatrix}, \\
H_4\begin{pmatrix} x \\ y \end{pmatrix} &= \frac{1}{4} RS \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1/4 \\ -1/4 \end{pmatrix}.
\end{aligned}$$

Nous sommes maintenant en mesure de donner une définition analytique de la transformation T. Pour toute courbe continue α dans Q d'origine a et d'extrémité b, $T(\alpha)$ est la courbe constituée des 4 courbes

$$\delta_1 := H_1(\alpha) \quad \alpha_1^+ := H_2(\alpha) \quad \alpha_1^- := H_3(\alpha) \quad \beta_1 := H_4(\alpha)$$

reliées par trois segments de droite : un qui joint l'extrémité $H_1(b)$ de δ_1 à l'origine $H_2(a)$ de α_1^+, un autre qui joint l'extrémité $H_2(b)$ de α_1^+ à l'origine de α_1^- et enfin le segment qui joint l'extrémité $H_3(b)$ de α_1^- à l'origine $H_4(a)$ de β_1.

[3] Ce sont des isométries de $\mathbf{R}^2$ muni de la norme euclidienne usuelle.

La courbe de Hilbert s'obtient comme limite des courbes $A_n := T^n(A_0)$ où A_0 est la courbe déjà vue, constituée de la ligne brisée qui joint le centre du carré $Q_1 = [0, 1/2] \times [0, 1/2]$ aux centres des carrés successifs

$$Q_2 = \begin{pmatrix} -1/2 \\ 0 \end{pmatrix} + Q_1, \quad Q_3 = \begin{pmatrix} -1/2 \\ -1/2 \end{pmatrix} + Q_1 \quad \text{et} \quad Q_4 = \begin{pmatrix} 0 \\ -1/2 \end{pmatrix} + Q_1.$$

Remarquons qu'avec ces définitions, on a $H_i(Q) = Q_i$ pour $i = 1, \ldots, 4$ et que les segments qui interviennent dans la définition de $T(A_0)$ (et de $T(A_n)$) sont horizontaux ou verticaux. En précisant leurs orientations par des flèches, on peut coder simplement la construction de $T(A_0)$ par la règle de substitution suivante

$$\downarrow^{\leftarrow}_{\rightarrow} \text{A} : \qquad \text{D} \quad \leftarrow \quad \text{A} \quad \downarrow \quad \text{A} \quad \rightarrow \quad \text{B}.$$

qui est calquée sur la construction de $T(\alpha)$ décrite plus haut, représentée par le schéma

$$\begin{array}{ccc} \alpha_1^+ & \leftarrow & \delta_1 \\ \downarrow & & \\ \alpha_1^- & \rightarrow & \beta_1 \ , \end{array}$$

et que l'on comparera utilement avec les tracés de la figure 5.5.

Pour procéder à l'itération de T, il faut décrire les règles de substitutions correspondantes pour B, C et D. Elles se déterminent facilement à partir de la définition de T et des propriétés du groupe étudié plus haut. On obtient les règles suivantes :

$$\begin{array}{lccccccc} \downarrow^{\leftarrow}_{\rightarrow} \text{A} : & \text{D} & \leftarrow & \text{A} & \downarrow & \text{A} & \rightarrow & \text{B} \\ \uparrow^{\rightarrow}_{\downarrow} \text{B} : & \text{C} & \uparrow & \text{B} & \rightarrow & \text{B} & \downarrow & \text{A} \\ {}^{\leftarrow}_{\rightarrow}\uparrow \text{C} : & \text{B} & \rightarrow & \text{C} & \uparrow & \text{C} & \leftarrow & \text{D} \\ \uparrow_{\leftarrow}\downarrow \text{D} : & \text{A} & \downarrow & \text{D} & \leftarrow & \text{D} & \uparrow & \text{C} \end{array}$$

Envisageons maintenant de tracer les courbes A_n, pour de petites valeurs de n, au moyen d'un algorithme récursif. Celui-ci va résulter de la combinaison des procédures récursives associées aux schémas substitutifs ci-dessus. La première est donnée par

```
procedure A (n : integer) ;
  begin
    if n > 0 then
      begin
        D(n - 1) ;
        Line(-u, 0) ;
        A(n - 1) ;
        Line(0, u) ;
        A(n - 1) ;
        Line(u, 0) ;
        B(n - 1)
      end ;
  end ;
```

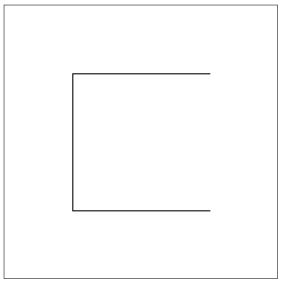
L'entier `n` correspond à la courbe A_n, `u` est la longueur commune des segments de droites que l'on trace dans la construction de A_n, sa valeur est calculée dans le programme principal en fonction de `n`. Nous laissons en exercice l'écriture des autres procédures (procédures `B(n)`, `C(n)` et `D(n)`). Les appels croisés entre plusieurs procédures peuvent entraîner une erreur de syntaxe, comme celle d'appeler dans une procédure, une autre procédure qui n'est pas encore définie. Nous avons décidé de contourner ce problème en écrivant une procédure `A(k,n)` où `k` est un paramètre entier, la valeur `k=1` correspondant à la procédure `A(n)` donnée ci-dessus ; de même les valeurs 2, 3, 4 de `k`, correspondent respectivement aux procédures `B(n)`, `C(n)` et `D(n)`. Une autre méthode, plus classique, utilise la déclaration `forward` (voir exercice 5.14).

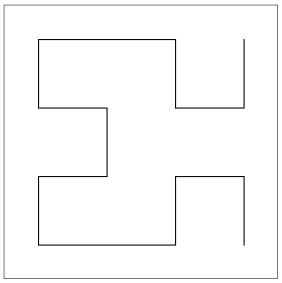
Le programme complet (cf. infra) pour tracer la courbe A_n commence par demander une valeur de n (entre 1 et 9) puis positionne une fenêtre sur l'écran dans laquelle il trace un carré de coté 512 ($= 2^9$) pixel. Ensuite, le programme calcule l'origine `(x0,y0)` de la courbe et y place la plume par `MoveTo (x0,y0)`. Dans la foulée, le pas `u` du déplacement est calculé en fonction de `n`. Ensuite, on rentre dans la procédure récursive par `A(1,n)`. Examinons ce que fait ensuite le programme lorsque `n=1`. Il exécute `A(1)`, donc commence par exécuter `D(0)` qui ne fait rien, puis déplace la plume de `(-u,0)`, exécute `A(0)` c'est-à-dire ne fait rien, puis déplace la plume de `(0,u)`, passe `A(0)` pour déplacer la plume de `(u,0)` et sort de `A(1)` par `B(0)`, donc ne trace plus rien et s'arrête. Au total, on a bien tracé la courbe initiale A_0.

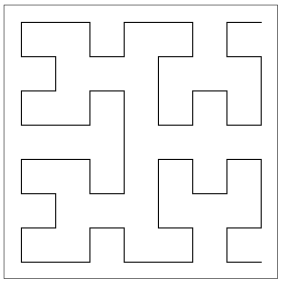
```
program HilbertN ;
  const
    h0 = 512 ;
    d0 = 100 ;
    g0 = 100 ;
  var
    n, i, k, x0, y0, u : integer ;
    r, s : Rect ;

procedure A (k, n : integer) ;
begin
  if k = 1 then
    begin
      if n > 0 then
        begin
          A(4, n - 1) ;
          Line(-u, 0) ;
          A(1, n - 1) ;
          Line(0, u) ;
          A(1, n - 1) ;
          Line(u, 0) ;
          A(2, n - 1)
        end ;
    end ;
  if k = 2 then
    begin
      if n > 0 then
        begin
          A(3, n - 1) ;
          Line(0, -u) ;
          A(2, n - 1) ;
          Line(u, 0) ;
          A(2, n - 1) ;
          Line(0, u) ;
          A(1, n - 1)
        end ;
    end ;
  if k = 3 then
    begin
      if n > 0 then
        begin
          A(2, n - 1) ;
          Line(u, 0) ;
```

```
          A(3, n - 1) ;
          Line(0, -u) ;
          A(3, n - 1) ;
          Line(-u, 0) ;
          A(4, n - 1)
        end ;
    end ;
  if k = 4 then
    begin
      if n > 0 then
        begin
          A(1, n - 1) ;
          Line(0, u) ;
          A(4, n - 1) ;
          Line(-u, 0) ;
          A(4, n - 1) ;
          Line(0, -u) ;
          A(3, n - 1)
        end ;
    end ;
  end ;

begin
  writeln('Courbe de Hilbert :') ;
  writeln('entrer un entier entre 1 et 9 ') ;
  read(n) ;
  SetRect(s, d0 - 50, g0 - 50, d0 + h0 + 50, g0 + h0 + 50) ;
  SetRect(r, d0 - 50, g0 - 50, d0 + h0 - 50, g0 + h0 - 50) ;
  SetDrawingRect(s) ;
  ShowDrawing ;
  FrameRect(r) ;
  x0 := d0 + h0 div 2 - 50 ;
  y0 := g0 + h0 div 2 - 50 ;
  u := h0 ;
  i := 0 ;
  repeat
    i := i + 1 ;
    u := u div 2 ;
    x0 := x0 + (u div 2) ;
    y0 := y0 - (u div 2) ;
  until i = n ;
  MoveTo(x0, y0) ;
  A(1, n) ;
end.
```

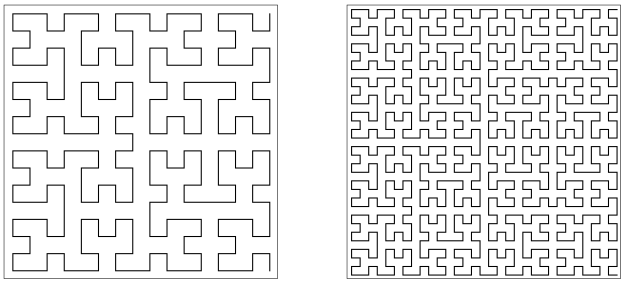
En utilisant ce programme, on obtient les 5 premières courbes de Hilbert.







$n = 1$ $n = 2$ $n = 3$

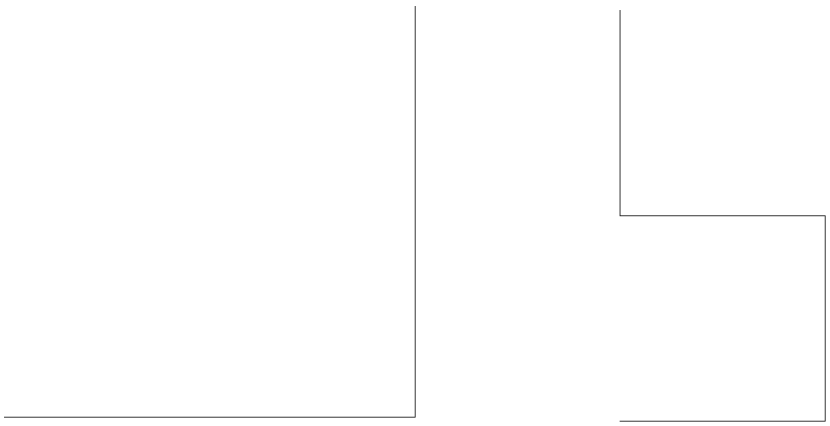


$n = 4$ $n = 5$

Figure 5.9. Tracés des courbes de Hilbert A_n pour $n = 1, 2, 3, 4$ et 5.

5.4.3 Le dragon

La courbe est célèbre. Une manière ludique de l'introduire est de prendre une bande de papier est de la plier, partie droite sur la partie gauche. Déplier la bande en préservant le pli en angle droit. Vue par la tranche, la bande dépliée forme une ligne brisée L_1 en forme de L inversé. Plions maintenant la bande 2,3,4 fois ... de la même manière que la première fois et déplions en formant des angles droits en chaque pli. On obtient les lignes brisées L_2, L_3, L_4,... reproduites ci-dessous. Noter l'effet d'enroulement autour du segment initial d'origine $(0, 0)$ et d'extrémité $(\ell/2^n, 0)$, où ℓ dénote la longueur de la bande.



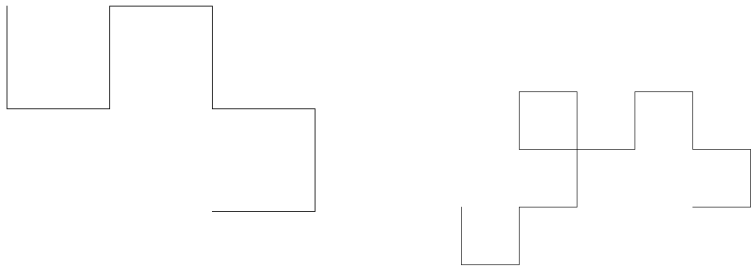


Figure 5.10. Plier et déplier une bande de papier 1, 2, 3 et 4 fois.

On renormalise chaque courbe L_n obtenue idéalement après n pliages et dépliages en angles droits par une similitude S_n de centre l'origine $(0,0)$ de la bande de papier, et qui place l'autre extrémité de la bande en `(u,u)` (`u` désignant l'unité de longueur choisie). Notons D_n la ligne brisée ainsi construite à partir de L_n. Ce choix permet de stabiliser la suite des courbes qui, à la limite, va remplir une partie compacte du plan : le Dragon. Il se visualise facilement sur écran, dès que n est suffisamment grand. La figure 5.11a donnent D_4, D_5, D_6 et D_7 :

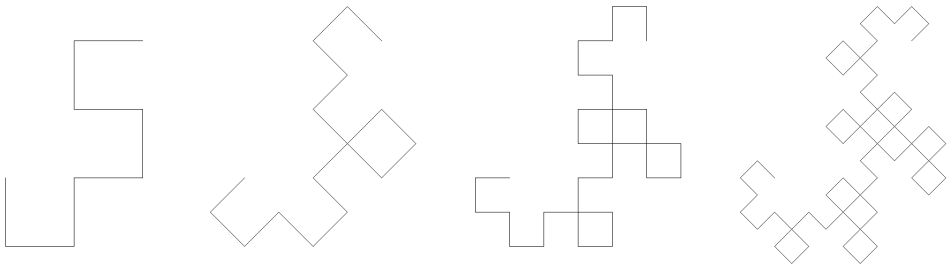


Figure 5.11a. Premières courbes ($n = 4, 5, 6, 7$) dans la construction du Dragon.

Poussons plus loin :

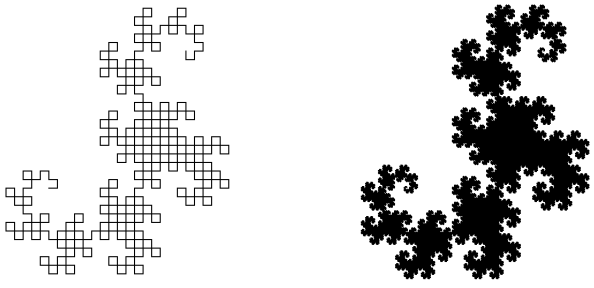


Figure 5.11b. Courbes Dragon pour $n = 10$ et le Dragon final ($n = 20$).

Passons à l'algorithme de construction des courbes D_n. La première, D_0, est un segment orienté $[A, B]$, d'origine A et d'extrémité B. On passe de D_0 à $D_1 = D(A, B)$ en calculant tout d'abord le point C tel que ACB forme un triangle isocèle rectangle en C, le parcours A

vers C, C vers B et retour sur A étant effectué dans le sens positif (et donc C est "à droite de l'axe (A, B)". On trace ensuite D_2 : c'est la ligne brisée formée des segments orientés $[A, C]$ et $[B, C]$. Les données d'orientation sont importantes. Si les coordonnées de A et B sont respectivement (x, y), (z, t), alors les coordonnées (u, v) de C sont données simplement par

$$u = \frac{x + z + t - y}{2}; \tag{5.6}$$

$$v = \frac{x - z + t + y}{2}. \tag{5.7}$$

Pour passer de D_{n-1} à D_n, on remplace chaque segment orienté $[a, b]$ par les deux segments orientés $D(a, b)$ construit comme ci-dessus.

Appelons `DragonPlus` la procédure qui correspond à cette construction. Dans l'affichage à l'écran, il faut tenir compte que l'axe des ordonnées est orienté positivement vers le bas, ce qui a pour effet de changer y en $-y$, t en $-t$, v en $-v$ dans les formules (5.5) et (5.6).

```
program Dragon ;
  var
    n : integer ;
    s : Rect ;
  procedure DragonPlus (n : integer ; x, y, z, t : integer) ;
    var
      u, v : integer ;
  begin
    if n = 1 then
      begin
        MoveTo(x, y) ;
        LineTo(z, t) ;
      end
    else
      begin
        u := (z + x + y - t) div 2 ; (* en accord avec les coordonnées *)
        v := (z - x + t + y) div 2 ; (* à l'écran *)
        DragonPlus(n - 1, x, y, u, v) ;
        DragonPlus(n - 1, z, t, u, v) ;
    end ;
  end ;
begin
  SetRect(s, 50, 50, 552 + 50, 552 + 50) ;
  SetDrawingRect(s) ;
  ShowDrawing ;
  read(n) ;
  Dragon(n, 150, 100 + 256, 150 + 256, 100) ;
end .
```

Dans cette construction, si on suit pas à pas l'exécution du programme, il faut lever la plume (examiner le cas $n = 1$). Avec le programme `DDragon` qui suit, on trace D_n deux fois mais sans lever la plume. On utilise pour cela une deuxième procédure désignée par `DragonMoins` qui trace la courbe avec des orientations inversées, ce qui, bien enfilé avec la procédure `DragonPlus` modifiée, donne un tracé sans saut de plume. L'exercice est quelque peu futile, mais il nous donne l'occasion de montrer comment appeler une procédure non

encore définie dans une autre procédure. Lors de la construction des courbes de Hilbert, le problème avait été soulevé et résolu en paramétrisant les procédures pour en faire une seule. Cette solution peut s'avérer impraticable. Le langage Pascal fournit cependant une solution très simple avec la déclaration spécifique `forward` qui indique que la procédure (ou la fonction) est définie plus loin dans le programme. C'est ce choix qui a été fait dans le programme suivant.

```
program DDragon ;
  var
    n : integer ;
    s : Rect ;
  procedure DDragonMoins (n : integer ; x, y, z, t : integer) ;
  forward ;
  procedure DDragonPlus (n : integer ; x, y, z, t : integer) ;
    var
      u, v : integer ;
  begin
    if n = 1 then
      begin
        MoveTo(x, y) ;
        LineTo(z, t) ;
      end
    else
      begin
        u := (z + x + y - t) div 2 ;
        v := (z - x + t + y) div 2 ;
        DDragonPlus(n - 1, x, y, u, v) ;
        DDragonMoins(n - 1, u, v, z, t) ;
    end ;
  end ;
  procedure DDragonMoins (n : integer ; x, y, z, t : integer) ;
    var
      u, v : integer ;
  begin
    if n = 1 then
      begin
        MoveTo(x, y) ;
        LineTo(z, t) ;
      end
    else
      begin
        u := (z + x - y + t) div 2 ;
        v := (t + x - z + y) div 2 ;
        DDragonPlus(n - 1, x, y, u, v) ;
        DDragonMoins(n - 1, u, v, z, t) ;
    end ;
  end ;
begin
  SetRect(s, 50, 50, 552 + 50, 552 + 50) ;
  SetDrawingRect(s) ;
  ShowDrawing ;
  read(n) ;
  DDragonPlus(n, 150, 100 + 256, 150 + 256, 100) ;
end .
```

***Exercice* 5.13** a) Examiner pas à pas l'exécution du programme `HilbertN` pour la valeur $n = 2$.
b) On désigne par L_n le nombre d'appels à la fonction graphique `Line( )` pour tracer A_n. Montrer que L_n vérifie une relation de récurrence d'ordre 1 et calculer explicitement L_n.

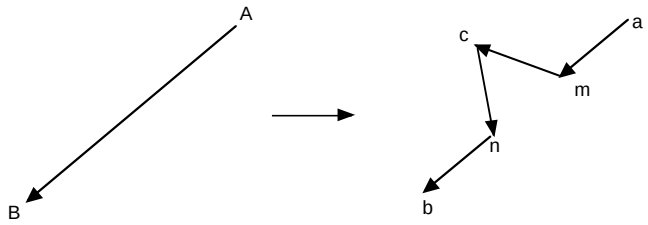
c) En se plaçant dans le carré unité, la courbe initiale A_0 de Hilbert à pour longueur 3/2. Calculer la longueur de la n-ième courbe A_n.
d) Quel est le résultat obtenu sur la sortie écran lorsque $n = 9$? Que se passe-t-il pour $n = 10$? e) Utiliser la fonction graphique `Line(h,v)` (déplacement de la plume du vecteur `(h,v)`) pour construire une fonction `LLine(h,v,t)` qui effectue le même tracé que `Line(h,v)` mais de manière plus lente, `t` étant un paramètre proportionnel au temps mis par la plume pour effectuer le tracé. Injecter cette fonction dans le programme HilbertN de manière à ralentir la construction de A_n.

***Exercice* 5.14** a) Réécrire le programme `HilbertN` en utilisant la déclaration `forward` et 4 procédures basées sur les schémas substitutifs de la construction des courbes de Hilbert.
b) Remplacer dans la construction la courbe initiale A_0 dans le carré unité Q (cf. sous-section 5.4.2) par la ligne brisée passant par les points de coordonnées $(1/2, 1/2)$, $(-1/2, 0)$ et $(1/2, -1/2)$.

***Exercice* 5.15 Flocon de Koch**.
a) Tracer un triangle équilatéral Δ dans une fenêtre.
b) On définit une transformation T sur les segments orientés $[A, B]$ de la manière suivante : l'image $T(A, B)$ de $[A, B]$ est une ligne brisée formée de 4 segments orientés $[a, m]$, $[m, c]$ $[c, n]$ et $[n, b]$, de même longueur, égale au tiers de celle de $[A, B]$, et tels que.

1. $[a, m]$ est le tiers initial de $[A, B]$ (donc $A = a$) ;
2. $[n, b]$ est le tiers final de $[A, B]$ (donc $B = b$) ;
3. $[m, c]$ forme avec $[A, B]$ un angle de $\pi/3$;
4. $[c, n]$ forme avec $[A, B]$ un angle de $-\pi/3$.

Le dessin ci-contre montre cette construction.



écrire une procédure `Flocon` qui réalise la transformation T (elle prend en entrée deux points A, B et renvoie la ligne brisée $T(A, B)$).
c) Le triangle équilatéral Δ est appelé flocon d'ordre 0 ; il est orienté de manière à parcourir le périmètre dans le sens positif. Le flocon Δ_1 d'ordre 1 est obtenu en remplaçant chaque coté (orienté) de Δ par leur image suivant T ; c'est une ligne brisée fermée de 12 côtés. Le flocon Δ_n d'ordre n se déduit du flocon d'ordre $n - 1$ en remplaçant chacun de ses côtés (orientés) par son image suivant T. écrire un programme récursif qui trace les flocons Δ_n à la demande ($n \leq 20$).
d) Soit N_n le nombre de côtés de Δ_n. Calculer N_n explicitement.
e) Calculer, en fonction de la longueur des cotés de Δ_0, l'aire A_n de la région intérieure délimitée par le flocon Δ_n.