Notes du

Cours d'informatique

Math. Sup. MPSI Lycée Fermat

A. Soyeur

Table des matières

1 Introduction à CAML **4**
- 1.1 Session interactive . . . 4
- 1.2 Généralités . . . 4
  - 1.2.1 Définitions. . . . 4
  - 1.2.2 Parenthèses. . . . 5
  - 1.2.3 Commentaires . . . 6
  - 1.2.4 Types. . . . 6
  - 1.2.5 Fonctions. . . . 8
- 1.3 Aspects impératifs du langage . . . 8
  - 1.3.1 Références. . . . 9
  - 1.3.2 Vecteurs. . . . 9
  - 1.3.3 Structures de contrôle. . . . 10
- 1.4 Preuve de programmes impératifs . . . 11
  - 1.4.1 Indécidabilité de la terminaison . . . 11
  - 1.4.2 Invariants de boucles . . . 12
- 1.5 Aspects fonctionnels du langage . . . 14
  - 1.5.1 Filtrage. . . . 15
  - 1.5.2 Types fonctionnels . . . 16

2 Récursivité **18**
- 2.1 Fonctions récursives . . . 18
- 2.2 Terminaison et correction d'une fonction récursive . . . 19
- 2.3 Induction structurelle . . . 23
- 2.4 Listes . . . 25
- 2.5 Implémentation de listes dans d'autres langages . . . 28
- 2.6 Programmation fonctionnelle contre programmation impérative . . . 29
  - 2.6.1 Récursivité terminale . . . 30
  - 2.6.2 Accumulateurs . . . 32

3 Analyse mathématique des algorithmes **34**
- 3.1 Complexité . . . 34
- 3.2 Etude d'un cas : recherche d'un élément dans un tableau . . . 36
  - Recherche linéaire . . . 36
  - Recherche dichotomique . . . 37
- 3.3 Tris élémentaires . . . 38
  - 3.3.1 Le tri par sélection . . . 38
  - 3.3.2 Le tri bulle . . . 39
  - 3.3.3 tri par insertion . . . 41
  - 3.3.4 Comparaison des tris . . . 44

4 Stratégie diviser pour régner **46**
- 4.1 La stratégie diviser pour régner . . . 46
- 4.2 Un cas modèle . . . 46
- 4.3 Résolution générale des récurrences « diviser pour régner » . . . 47
- 4.4 L'algorithme d'exponentiation rapide . . . 48
- 4.5 Le tri fusion (mergesort) . . . 48

4.6 Le tri rapide (quicksort) . . . 52
4.7 Multiplication de polynômes . . . 55
4.8 Multiplication de matrices : algorithme de Strassen . . . 58

5 Piles, expressions algébriques **59**
5.1 Types de données abstraits . . . 59
5.2 Pile . . . 60
5.3 Files . . . 61
5.4 Expressions algébriques . . . 61
5.4.1 Évaluation d'une expression arithmétique postfixée à l'aide d'une pile . . . 64

6 Logique **67**
6.1 Propositions logiques . . . 67
6.2 Syntaxe des propositions logiques . . . 68
6.2.1 Définition inductive des propositions logiques . . . 68
6.2.2 Notation linéaire d'une proposition logique . . . 69
6.3 Sémantique . . . 70
6.3.1 Évaluation d'une proposition logique . . . 70
6.3.2 Propositions logiquement équivalentes . . . 70
6.3.3 Tautologies . . . 71
6.3.4 Formes conjonctives et disjonctives . . . 73
6.4 Méthode de résolution . . . 75
6.5 Circuits logiques . . . 76
6.5.1 Fonctions booléennes . . . 76
6.5.2 Circuits logiques . . . 77
6.5.3 Additionneurs . . . 79

A Types construits en CAML **82**
A.1 Types produit . . . 82
A.2 Types somme . . . 83

B Aide mémoire CAML **87**

C Conseils de présentation des programmes **90**
C.1 Espaces . . . 90
C.2 Indentation des blocs . . . 90
C.3 Filtrage . . . 91
C.4 Parenthèses . . . 91
C.5 Commentaires . . . 91

D Solutions des exercices **92**

Chapitre 1

Introduction à CAML

1.1 Session interactive

Le langage de programmation utilisé cette année s'appelle CAML. Il a été développé par des chercheurs de l'INRIA, et est disponible gratuitement aux URL suivantes :

`http://pauillac.inria.fr/caml/index-fra.html`
`http://pauillac.inria.fr/caml/distrib-caml-light-fra.html`

CAML est un langage *compilé* (comme PASCAL ou C...) mais il est fourni avec une « boucle interactive », qui permet de saisir des définitions, de valider l'entrée, de les compiler en les ajoutant aux définitions précédentes et de voir immédiatement le résultat des évaluations.

CAML
```
#2 + 2;;
- : int = 4
```

Au prompt (le #), l'utilisateur rentre une commande, suivie de `;;`
La seconde ligne est la réponse de CAML.
Remarquez le *type* (`int` pour un entier) qui précède la *valeur* calculée.

CAML
```
#2. +. 3.13;;
- : float = 5.13
```

Remarquez l'utilisation du point pour désigner des nombres « flottant », ainsi que l'opérateur `+.` pour les additionner.

1.2 Généralités

1.2.1 Définitions.

On définit des *constantes* avec : `let ident = valeur` :

CAML
```
#let a = 3;;
a : int = 3
#a + 1;;
- : int = 4
```

La constante a est maintenant définie et l'identificateur a sera remplacé par la suite par la valeur 3. On dit que l'indentificateur a est *lié* à la valeur 3. Il est impossible de modifir la valeur de cette constante (on peut néanmoins redéfinir une nouvelle constante de même nom).

Remarque 1. Les définitions sont « statiques » comme le montre l'exemple suivant : au départ, a est lié à la valeur 1, on définit la constante b grâce à a. Si a est lié à une nouvelle valeur par la suite, b reste lié à sa valeur initiale.

CAML
```
#let a = 1;;
a : int = 1
#let b = a + 1;;
b : int = 2
#let a = 2;;
a : int = 2
#b;;
- : int = 2
```

On définit des *liaisons locales* grâce à : `let ... in ...` (comme en mathématiques : « soit ... dans ce qui suit »). Les liaisons n'ont une portée que dans la partie de l'expression qui suit :

CAML
```
#let a = 3 in a + 4;;
- : int = 7
#a;;
Entrée interactive:
>a;;<EOF>
>^
L'identificateur a n'est pas défini.
```

On peut définir plusieurs liaisons locales simultanément :

CAML
```
#let a = 2 and b = 3 in
  (a + b) * 2;;
- : int = 10
```

Mais il faut qu'elles soient indépendantes :

CAML
```
#let a = 2 and b = 2 * a in
  a + b;;
Entrée interactive:
>let a = 2 and b = 2 * a in
>                     ^
L'identificateur a n'est pas défini.
```

On utilise deux `let in` successifs :

CAML
```
#let a = 2 in
let b = 2 * a in
  a + b;;
- : int = 6
```

1.2.2 Parenthèses.

Comme dans la plupart des langages, les opérateurs possèdent une *priorité*. Par exemple la multiplication est prioritaire par rapport à l'addition :

CAML
```
#2 + 3 * 5;;
- : int = 17
```

Si l'on veut modifier l'ordre d'évaluation des expressions, on met entre parenthèses ce qu'il faut évaluer en priorité :

CAML
```
#(2 + 3) * 5;;
- : int = 25
```

Il en est de même pour toutes les constructions en CAML.

1.2.3 Commentaires

Tout ce qui est placé entre les symboles `(*` `*)` est ignoré par le compilateur, ce qui permet d'introduire des commentaires dans les programmes (éventuellement sur plusieurs lignes).

1.2.4 Types.

CAML est un langage « fortement typé ». Chaque objet possède un *type* particulier et des fonctions qui s'appliquent aux objets de ce type uniquement.

- `int`
  Un entier est codé en CAML en binaire sur 31 bits, c'est à dire que vous pouvez manipuler des entiers compris entre -2^{30} et $2^{30}-1$. En dehors de cet intervalle, les résultats affichés sont faux.
  Vous pouvez utiliser les opérateurs usuels $+$, $-$,$*$ sur les entiers. L'opérateur `/` et `mod` permettent d'obtenir respectivement le quotient et le reste de la division euclidienne (très utile en informatique...) : $22 = 7 \times 3 + 1$:

  CAML
  ```
  #22 / 3;;
  - : int = 7
  #22 mod 3;;
  - : int = 1
  ```

  On utilise la fonction `print_int` pour afficher un entier.
- `float`
  La représentation d'un nombre flottant est complètement différente de celle d'un nombre entier en machine : mantisse plus puissance de 10 éventuelle (notée `E`).
  La limitation de la taille du nombre est moins restrictive, mais attention aux erreurs d'arrondi ! On utilise la notation `.` après les opérateurs usuels sur les flottants :

  CAML
  ```
  #4. *. 5.;;
  - : float = 20.0
  #2.E4 /. 2.4;;
  - : float = 8333.33333333
  ```

  La fonction `print_float` permet d'afficher un flottant.
- `char`
  Un caractère est codé par un entier compris entre 0 et 256 .
  Voici la table ASCII des correspondances les plus intéressantes :

  ```
   32      33 !    34 "    35 #    36 $    37 %     38 &    39 '
   40 (    41 )    42 *    43 +    44 ,    45 -    46 .    47 /
   48 0    49 1    50 2    51 3    52 4    53 5    54 6    55 7
   56 8    57 9    58 :    59 ;    60 <    61 =    62 >    63 ?
   64 @    65 A    66 B    67 C    68 D    69 E    70 F    71 G
   72 H    73 I    74 J    75 K    76 L    77 M    78 N    79 O
   80 P    81 Q    82 R    83 S    84 T    85 U    86 V    87 W
   88 X    89 Y    90 Z    91 [    92 \    93 ]    94 ^    95 _
   96 `    97 a    98 b    99 c   100 d   101 e   102 f   103 g
  104 h   105 i   106 j   107 k   108 l   109 m   110 n   111 o
  112 p   113 q   114 r   115 s   116 t   117 u   118 v   119 w
  120 x   121 y   122 z   123 {   124 |   125 }   126 ~
  ```

  Les fonctions `int_of_char` et `char_of_int` permettent de passer du caractère à son code ASCII.

  CAML
  ```
  #char_of_int 65;;
  - : char = 'A'
  #int_of_char 'T';;
  - : int = 84
  ```

  La fonction `print_char` permet d'afficher un caractère.

– `string`

Une chaîne de caractère est définie entre guillemets :

CAML
```
#let chaine = "bonjour!";;
chaine : string = "bonjour!"
```

On peut *concaténer* deux chaînes :

CAML
```
   #let chaine1 = "Bonjour" and chaine2 = "le monde" in
  print_string  chaine1 ^ chaine2;;
  Bonjourle monde- : unit = ()
```

Longueur d'une chaîne et accès au ième caractère (attention le premier caractère est d'indice 0) :

CAML
```
#let chaine = "Bonjour";;
chaine : string = "Bonjour"
#string_length chaine;;
- : int = 7
#chaine.[0];;
- : char = 'B'
#chaine.[6];;
- : char = 'r'
#chaine.[7];;
Exception non rattrapée: Invalid_argument "nth_char"
```

Attention aux modifications des chaînes (elles se comportent comme les vecteurs) :

CAML
```
#let c1 = "abcd";;
c1 : string = "abcd"
#let c2 = c1;;
c2 : string = "abcd"
#c2.[0] <- 'A';;
- : unit = ()
#c1;;
- : string = "Abcd"
```

– Couples et n-uplets.

A partir de types existants, on peut définir un nouveau type (correspondant à un produit cartésien en mathématiques) :

CAML
```
#(2, 3);;
- : int * int = 2, 3
#5, true, 4.;;
- : int * bool * float = 5, true, 4.0
```

Remarquez le symbole `*` pour désigner des produits cartésiens. Les parenthèses sont facultatives pour définir des n-uplets.

– unit

Il existe un type particulier : `unit` avec un seul élément `()`. Il correspond à un « effet de bord », c'est à dire à une modification de l'état de la machine. La valeur `()` est une valeur de retour comme une autre d'une fonction CAML !

Par exemple :

CAML
```
#print_string "Bonjour";;
Bonjour- : unit = ()
```

Nous verrons d'autres types plus tard. Il faut également savoir que CAML permet aux utilisateurs de définir leurs propres types.

1.2.5 Fonctions.

- Fonction à un argument :

CAML
```
let f x =
x + 2;;
f : int -> int = <fun>
#f 3;;
- : int = 5
```

On a défini un nouvel identificateur `f` correspondant à une fonction :
`f` de type : `int ->int = <fun>`
Cette fonction prend un entier comme argument et renvoie un entier.
Comment CAML a-t-il déduit le type de cette fonction ? Dans sa définition, la fonction utilise l'opérateur + réservé aux entiers. Donc x est nécessairement de type `int`.
Il est très important de bien observer les types en CAML . C'est une aide utile pour corriger les erreurs de syntaxe, et vérifier la cohérence des fonctions.
Remarquez les parenthèses facultatives autour des arguments des fonctions.

CAML
```
#let fonction x =
  x *. 2. ;;
fonction : float -> float = <fun>
#fonction 3.3 +. 2.;;
- : float = 8.6
#fonction (3.3 +. 2.);;
- : float = 10.6
```

- Fonction à plusieurs arguments.

CAML
```
#let fonction2 x y =
x * y + 2;;
fonction2 : int -> int -> int = <fun>

#fonction2 3 5;;
- : int = 17
```

Remarquez le type `int->int->int` pour désigner une fonction à deux variables.

- Chaque fonction retourne une *unique valeur*. Le type de cette valeur peut être `unit`, ce qui signifie que la fonction a effectué un *effet de bord* :

CAML
```
  #let f x =
    print_int x;
    print_newline()
    ;;

f : int -> unit = <fun>
```

1.3 Aspects impératifs du langage

Le style de programmation *impératif* est proche du fonctionnement d'un ordinateur : on spécifie dans l'ordre une série d'*instructions* que l'ordinateur doit effectuer. Ces instructions modifient des *variables*, effectuent des branchements... Les langages C, Pascal, Fortran, ADA ... sont des langages qui utilisent ce style de programmation.
CAML n'est pas un langage impératif (c'est un langage fonctionnel), mais il est doté de certaines caractéristiques qui permettent de « mimer » la programmation impérative.

1.3.1 Références.

Une référence est la donnée d'une plage mémoire de longueur fixe. On peut la voir comme une boîte qui contient des objets d'une taille fixe, la référence elle-même étant une étiquette collée sur la boîte. On peut modifier (avec l'opérateur `:=`) et lire (avec l'opérateur `!`) le contenu de la boîte :

CAML
```
#let ma_ref = ref 0;;
ma_ref : int ref = ref 0
#ma_ref := 1;;
- : unit = ()
#!ma_ref;;
- : int = 1
#ma_ref;;
- : int ref = ref 1
```

On a défini une référence sur des entiers : (type `int ref`).
Ne pas confondre la référence (`ma_ref` de type `int ref`) avec son contenu (`!ma_ref` de type `int`).

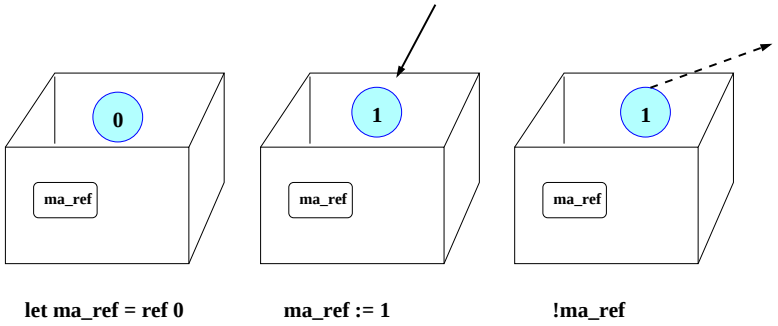


FIG. 1.1 – *Référence*

1.3.2 Vecteurs.

C'est un ensemble de cases mémoires de taille *fixe*. Chaque case contient un élément du même type. On peut accéder directement ou modifier chaque élément du vecteur en temps constant.

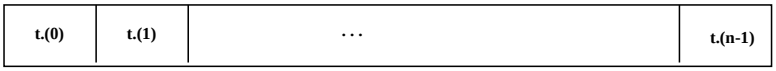


FIG. 1.2 – *Vecteur*

CAML
```
#let tab = [| 1; 2; 3 |];;
tab : int vect = [|1; 2; 3|]

#vect_length tab;;
- : int = 3

#tab.(0);;
- : int = 1
#tab.(2);;
- : int = 3
#tab.(3);;
Exception non rattrapée: Invalid_argument "vect_item"
```

```
#tab.(1) <- 5;;
- : unit = ()
#tab;;
- : int vect = [|1; 5; 3|]
```

On crée un vecteur de taille n rempli d'éléments x :

CAML
```
#let t = make_vect 10 0;;
t : int vect = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Les vecteurs sont toujours passés par référence dans les fonctions, c'est à dire qu'ils sont modifiés par des opérations locales aux fonctions :

CAML
```
#let t = [|0; 1; 2|];;
t : int vect = [|0; 1; 2|]
#let f tab = tab.(0) <- -1;;
f : int vect -> unit = <fun>
#f t;;
- : unit = ()
#t;;
- : int vect = [|-1; 1; 2|]
```

Attention également aux liaisons :

CAML
```
#let t1 = [| 1; 2; 3 |];;
t1 : int vect = [|1; 2; 3|]
#let t2 = t1;;
t2 : int vect = [|1; 2; 3|]
#t2.(0) <- 5;;
- : unit = ()
#t1;;
- : int vect = [|5; 2; 3|]
```

1.3.3 Structures de contrôle.

1. Test simple :

   CAML
   ```
   #let x = 1;;
   x : int = 1
   #if x > 0 then print_string "positif";;
   positif- : unit = ()
   ```

2. Branchement :

   CAML
   ```
   #if x > 0 then print_string "positif" else print_string "négatif";;
   ```

3. Blocs :
   tout ce qui est placé entre `begin` et `end` est considéré comme une unique instruction par le compilateur.

   CAML
   ```
   #let f x =
     if x > 0 then
       begin
         print_string "x > 0";
         print_newline()
       end
     else
   ```

```
    begin
      print_string "x <= 0 ";
      print_newline()
    end;;
```

4. Boucles FOR :

CAML
```
#let i =2;;
i : int = 2

#for i = 1 to 3 do
  print_int i;
  print_newline();
done;;

1
2
3
- : unit = ()
#i;;
- : int = 2
```

5. Boucles WHILE :
   La fonction suivante calcule le plus petit entier n tel que $2^n \geq x$:

CAML
```
let log_int x  =
  let puiss2 = ref 1 and expo = ref  0 in
    while !puiss2 < x do
      puiss2 := 2 * !puiss2;
      expo := !expo + 1;
    done;
    !expo;;
```

6. Opérateurs logiques.
   Le « et » s'écrit `&` ou encore `&&`. Le « ou » s'écrit `or` ou encore `||`.

Exercice 1-1

Ecrire une fonction
`maximum` de type : `int vect ->int`
qui détermine le plus grand élément d'un tableau d'entiers.

1.4 Preuve de programmes impératifs

En programmation impérative, il est important de prouver deux choses :

- La terminaison du programme ;
- La validité du programme : il fait bien ce que l'on attend de lui.

1.4.1 Indécidabilité de la terminaison

Il a été montré au début de ce siècle par Gödel, qu'il ne pouvait pas exister de programme universel qui dit si un programme donné se termine pour tous ses arguments ou pas. L'idée de la démonstration est la suivante. Sans rentrer dans les détails, on peut montrer que toute fonction calculable par un ordinateur peut être codée par un entier que nous noterons `code(f)` (ce qui montre d'ailleurs qu'il existe des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ qui ne sont pas calculables, car $\mathbb{N}$ est dénombrable alors que $\mathcal{F}(\mathbb{N},\mathbb{N})$ ne l'est pas !)

Supposons qu'il existe une fonction `termine : int -> bool` qui prend en argument le code d'une fonction quelconque et qui dit si oui ou non cette fonction se termine. On pourrait alors construire à partir de cette fonction `termine`, la fonction suivante :

```
let absurde () =
  while (termine code(absurde)) do
  done;;
```

- Si la fonction `absurde` se termine, alors `code(absurde)` vaut `true` et la fonction `absurde` boucle sans fin ;
- si par contre la fonction `absurde` ne se termine pas, alors `code(absurde)` vaut `false` mais alors la fonction `absurde` se termine de façon évidente.

Dans les deux cas, on aboutit à une absurdité.

1.4.2 Invariants de boucles

Étant donné un bloc d'instructions B, on appelle *contexte* du bloc, les variables auxquelles les instructions du bloc peuvent accéder (en lecture ou en écriture). Considérons deux propriétés C_1 et C_2 faisant intervenir le contexte du bloc. On dit que le bloc B satisfait la *précondition* C_1 et la *postcondition* C_2 lorsque la proposition suivante est satisfaite :

si la condition C_1 est vérifiée par le contexte du bloc, après exécution des instructions du bloc B, la propriété C_2 est aussi vérifiée.

On peut noter schématiquement cette propriété par

$$C_1 \xrightarrow{B} C_2$$

Il existe une théorie entière consacrée à la « preuve formelle » de programmes que nous n'étudierons pas.
Les blocs les plus intéressants dans un programme impératifs sont les boucles `for` et `while`. On utilise un « invariant de boucle » pour prouver ce type de programmes, c'est à dire une propriété I vérifiant

$$I \xrightarrow{B} I$$

où B désigne les instructions de la boucle.

Exercice 1-2

Montrer que la fonction `maximum` de l'exercice 1.3.3 est correcte.

Pour prouver un programme en utilisant un invariant de boucle :

1. On définit une *précondition* : elle décrit l'état des variables avant d'entrer dans la boucle ;
2. On prouve que l'invariant de boucle est vrai à chaque passage dans la boucle ;
3. La *condition de sortie de boucle* en conjonction avec l'invariant de boucle permet de prouver une *postcondition*. Cette postcondition permet de prouver la validité du programme ;

Remarque 2.

- On peut également se servir de l'invariant de boucle pour montrer la terminaison d'un programme ou pour étudier la complexité d'un programme ;
- Un invariant de boucle est un excellent commentaire qui permet de comprendre comment le programme fonctionne.

Exercice 1-3

Soit un polynôme $P = p_0 + p_1 X + \cdots + p_{n-1} X^{n-1}$ représenté par le vecteur de ses coefficients $p = [|p_0; \ldots; p_{n-1}|]$. Montrer que la fonction suivante (algorithme de Hörner) calcule pour un réel x, le réel $p(x)$.

```
let horner p x =
  let n = vect_length p in
  let y = ref p.(n - 1) in
    for k = n - 2 downto 0 do
      y := !y *. x +. p.(k)
    done;
    !y;;
```

Exercice 1-4

On suppose que les entiers sont codés en base 2 sur 8 bits. Par exemple, l'entier 14 s'écrit

$$14 = 0*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 0*2^0 \quad \text{représenté par } (00001110)_2$$

Les opérations simples à effectuer sont :

- La multiplication d'un entier par 2. Il suffit de décaler les bits d'un cran à gauche :

  $$2*(14) \hookrightarrow (00011100)_2$$

  (En CAML, `n lsl p` renvoie l'entier obtenu en décalant les bits de l'entier n de p crans à gauche).
- Le quotient de la division d'un entier par 2 noté $n/2$. Pour l'obtenir, il suffit de décaler les bits d'un cran vers la droite :

  $$14/2 \hookrightarrow (00000111)_2$$

  (En CAML, `n lsr p` renvoie l'entier obtenu en décalant les bits de l'entier n de p crans à droite).
- Le reste de la division d'un entier par 2. C'est le bit de poids le plus faible (le dernier à droite) :

  $$a \hookrightarrow (0000101)_2 \Rightarrow a \mod 2 = 1$$

  $$b \hookrightarrow (0001010)_2 \Rightarrow b \mod 2 = 0$$

  En CAML `n land p` retourne l'entier obtenu en faisant un « et » logique bit à bit des bits de n et p. Pour calculer le reste de n par 2, il suffit de calculer `n land 1`.

On se propose de voir un algorithme qui effectue la multiplication de deux entiers en n'utilisant que ces trois opérations élémentaires: $*2$, $\mod 2$, $/2$ et l'addition de deux entiers.

algorithme

```
MULTIPLIE x y   (* renvoie xy *)
m <- 0, a <- x, b <- y
TANT QUE b > 0 FAIRE
    SI (b mod 2) = 1 ALORS m <- m + a FIN SI
    a <- a * 2
    b <- b / 2
FIN TANT QUE
RETOURNER m
```

a. Ecrire en CAML la fonction correspondante (on n'utilisera pas pour simplifier les opérations sur les bits).

b. On note m_i,a_i,b_i le contenu de m,a,b après le ième passage dans la boucle TANT QUE. Trouver une propriété $\mathcal{P}(i)$ simple dépendant de m_i,a_i,b_i et prouver cette propriété. (On pourra regarder comment fonctionne l'algorithme avec $x = 2$ et $y = 5$).

c. En déduire que si l'on sort de la boucle TANT QUE, le programme renvoie bien xy.

d. Déterminer le nombre de passages dans la boucle TANT QUE.

Un néophyte programme souvent de la façon suivante :

1. Il écrit une première procédure (en tapant directement le code et sans commentaires évidemment) sans vraiment savoir où il va ;
2. Il la teste pour s'apercevoir qu'elle ne marche pas ;
3. Il la bricole en la modifiant jusqu'à ce qu'elle devienne incompréhensible (même pour lui), mais pour qu'elle fonctionne sur quelques exemples.
4. Il rajoute quelques commentaires sans intérêt pour la forme ;

Il est clair qu'une telle démarche est à proscrire pour trois raisons au moins :

1. Le temps passé à programmer de cette façon est beaucoup plus important que celui passé à réfléchir de façon sérieuse au problème initial ;

2. Les risques d'erreurs cachées sont énormes :
3. Le programme obtenu est souvent trop compliqué, incompréhensible et impossible à modifier.

Une meilleure méthode de conception de programme :

1. Trouver un invariant de boucle. Le problème à traiter possède un état initial et un état final. L'invariant de boucle décrit un état « intermédiaire » ;
2. Écrire les instructions à l'intérieur de la boucle de façon à maintenir l'invariant ;
3. Initialiser les variables de telle sorte que l'invariant soit vrai au premier passage dans la boucle ;
4. Compléter la condition de boucle de telle façon qu'à la sortie de boucle, la condition de sortie soit vraie.

Exercice 1-5

On considère un tableau contenant les éléments -1,0,1. Il s'agit d'écrire une fonction
`d_hollandais` de type : `int vect->int vect`
qui réorganise les éléments du tableau de telle façon que les -1 soient placés au début, les 0 au milieu et les 1 à la fin.

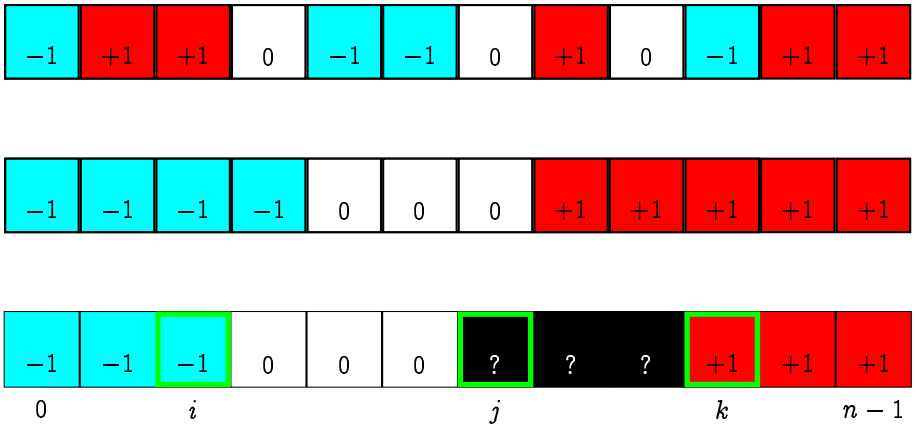


FIG. 1.3 – *Drapeau hollandais*

On utilisera la fonction
`echange` de type : `'a vect->int->int->unit`
telle que `echange t i j` échange les éléments $t.(i)$ et $t.(j)$ du tableau.

Exercice 1-6

On rappelle l'algorithme d'Euclide pour trouver le pgcd de deux entiers a et b. On considère la suite

$$r_0 = a,\ r_1 = b,\ \ldots, r_n \neq 0,\ r_{n+1} = 0$$

où la suite (r_k) est définie par récurrence par $\forall k \in \mathbb{N}$, r_{k+1} est le reste de la division euclidienne de r_{k-1} par r_k :

$$r_{k-1} = q_k r_k + r_{k+1} \quad 0 \leq r_{k+1} < r_k$$

Alors le pgcd de a et de b est le dernier reste non nul de cette suite, r_n.
Trouver un algorithme qui permet de calculer un couple de Bezout $(u,v) \in \mathbb{Z}^2$ tel que

$$ua + vb = \text{pgcd}(a,b)$$

1.5 Aspects fonctionnels du langage

Contrairement au style de programmation impérative, où le programmeur réfléchit aux différentes instructions à exécuter dans un ordre donné pour résoudre un problème, la programmation fonctionnelle est axée autour du concept de fonctions (souvent récursives).

Le programmeur *décrit* des propriétés mathématiques de la fonction, et laisse le travail d'évaluation de cette fonction au compilateur (voir l'exemple des tours de Hanoï).
Les programmes sont souvent plus clairs, plus simples à prouver et plus proches de la formulation mathématique du problème.
Une *instruction* travaille par *effet de bord*, c'est à dire qu'elle modifie l'état de la machine : une affectation := par exemple modifie l'état de la mémoire de la machine.
Une *fonction* prend des arguments d'un certain type et retourne des *valeurs* d'un certain type : elle effectue un *calcul*.
Dans un style de programmation purement fonctionnel, on effectue des *évaluations* d'expressions pour produire une *valeur*.
On s'interdira donc l'utilisation de références, affectations...

1.5.1 Filtrage.

Il est possible d'écrire des fonctions en envisageant des cas correspondant à des *motifs*. Par exemple, la fonction sur les entiers définie par $f(0) = 0$, $f(1) = 2$ et $\forall n > 2$, $f(n) = 2n + 2$ peut s'écrire:

CAML
```
let f  n =
match n with
| 0 -> 0
| 1 -> 2
| _ -> 2 * n + 1
;;
```

Le filtrage est défini par le mot-clé `match`. La fonction compare l'argument à filtrer (ici n) avec le premier motif de filtrage (ici 0). Si l'argument correspond, elle renvoie l'évaluation du membre droit, sinon elle passe au motif suivant. Le dernier motif correspond à « n'importe quoi », ce qui garantit que la fonction renvoie toujours un résultat.
Attention aux motifs de filtrage : ils doivent correspondre à un élément du type de l'argument. De même, les évaluations des membres droits, doivent produire le même type.
On peut écrire également la fonction précédente sans nommer l'argument n, en utilisant le mot clé `function` :

CAML
```
let  f = function
  | 0 -> 0
  | 1 -> 2
  | n -> 2 * n + 1;;
```

Il nous faut dans ce cas un identificateur n dans le dernier motif de filtrage car on s'y réfère dans le membre droit du filtrage.
On peut également filtrer des couples. La fonction suivante renvoie 0 si l'un des ses arguments est nul, la somme des arguments sinon :

CAML
```
let somme x y =
  match (x, y) with
    | (0, _) -> 0
    | (_, 0) -> 0
    | _ -> x + y;;
```

La fonction suivante (qui prend comme argument un couple) n'est pas correcte :

CAML
```
#let egalite_composantes   = function
    | (x, x) -> true
    | _ -> false;;
Entrée interactive:
>    | (x, x) -> true
>          ^
L'identificateur x est défini plusieurs fois dans ce motif.
```

On peut utiliser dans ces cas, une *garde de filtrage* c'est à dire une condition booléenne qui doit être vérifiée en plus du filtrage.

CAML
```
let egalite_composantes = function
  | (x, y) when x = y -> true
  | _ -> false;;
egalite_composantes : 'a * 'a -> bool = <fun>
```

La fonction suivante dit si l'une des coordonnées d'un couple est nulle ou si les deux éléments du couple sont égaux :

CAML
```
let egalite_ou_zero x y  =
  match (x, y) with
    | (0, _) -> true
    | (_, 0) -> true
    | (x, y) when x = y -> true
    | _ -> false;;
```

1.5.2 Types fonctionnels

Lorsque vous écrivez une fonction, CAML *synthétise* les types : il détermine les contraintes sur le type des arguments et du résultat en fonction de votre fonction. Si par exemple, il y a une expression `n + p` qui fait intervenir deux arguments n et p dans la définition de la fonction, puisque vous utilisez l'opérateur + qui nécessite deux entiers, CAML en déduit que les deux arguments sont de type `int`.
S'il n'y a aucune contrainte sur le type d'un argument, CAML utilise une notation `'a`, `'b` ... (prononcer alpha, beta ...) pour désigner un type quelconque.
Par exemple :

CAML
```
  let f x n =
    if n = x then (n, x)
    else (x, n)
;;
f : 'a -> 'a -> 'a * 'a = <fun>
```

L'opérateur d'égalité est *polymorphe* et n'introduit aucune contrainte sur le type de n et x. Par contre, il est nécessaire que n et x soient du même type. Le type retourné est un couple de type `'a * 'a`.
Remarquez l'utilisation de la flèche pour désigner les fonctions. La fonction précédente a deux arguments, de type `'a` et `'a`, et la fonction retourne un couple de type `'a * 'a`.
Une fonction peut également prendre comme argument une fonction, et retourner une fonction. En CAML, les fonctions sont des objets de « premier ordre ».

CAML
```
let minim comp a b =
  if comp a b then a
  else  b
;;
minim : ('a -> 'a -> bool) -> 'a -> 'a -> 'a = <fun>
```

Ici, l'argument `comp` doit être une fonction à deux arguments (car elle est appliquée à a et b à la deuxième ligne). Cette fonction `comp` doit retourner un booléen, car sa valeur de retour est utilisée dans un test, et la fonction f retourne a ou b. Comme la valeur de retour d'une fonction est d'un type unique, on peut en déduire que a et b doivent être de même type. CAML effectue tout ce travail de déduction et retourne les contraintes de type.
Remarquez les parenthèses pour désigner que le premier argument doit être une fonction.

la fonction précédente retourne le minimum des deux objets a et b, en utilisant la relation d'ordre définie par la fonction `comp`. Cette capacité à écrire des fonctions qui pourront être utilisées sur des objets de types différents s'appelle le *polymorphisme*.

CAML
```
#let plus_n n =
  (fun x -> x + n)
  ;;
plus_n : int -> int -> int = <fun>

#let f = plus_n 3;;
f : int -> int = <fun>

#f 5;;
- : int = 8
```

Le mot clé `fun` est utilisé pour construire une *fonction anonyme* (sans nom). La fonction `plus_n` renvoie une fonction.
Une fonction à deux arguments, de type `'a, 'b` et qui renvoie une valeur de type `'c` peut être vue comme une fonction à un argument de type `'a` qui renvoie une fonction de type `'b -> 'c`, ce qui explique les types fonctionnels renvoyés par CAML :

CAML
```
#let f x y =
 int_of_string x + int_of_float y;;
f : string -> float -> int = <fun>
#let g = f "100";;
g : float -> int = <fun>
#g 3.;;
- : int = 103
```

Chapitre 2

Récursivité

2.1 Fonctions récursives

Une fonction CAML peut être amenée à s'appeler elle-même pour évaluer un calcul. On dit qu'elle est *récursive*. Par exemple, la fonction factorielle possède la propriété mathématique suivante :

$$fact(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * fact(n-1) & \text{si } n > 1 \end{cases}$$

Elle se définit en CAML :

CAML
```
let rec fact n =
match n with
| 0 -> 1
| _ -> n * fact (n - 1);;
```

Notez la correspondance entre la propriété mathématique et la fonction CAML. C'est ce qui fait l'intérêt de la programmation fonctionnelle. Il faut utiliser le mot-clé `rec` pour définir une fonction récursive. Remarquez l'utilisation du filtrage : on étudie d'abord les *cas terminaux* pour lesquels la fonction renvoie un résultat, et ensuite le cas général où la fonction doit s'appeler elle-même. C'est de cette façon que nous programmerons les fonctions récursives par la suite.
On peut suivre les différents appels récursifs de la fonction ainsi que les valeurs retournées en utilisant `trace` :

CAML
```
#trace "fact";;
La fonction fact est dorénavant tracée.
- : unit = ()
#fact 4;;
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
- : int = 24
```

On voit que pour évaluer `fact 4`, il faut calculer `fact 3`, `fact 2`, `fact 1` et `fact 0`. Les valeurs de retour sont alors 0,1,2,6 et le résultat final, 24.
Il est possible de définir plusieurs fonctions mutuellement récursives en utilisant le mot-clé `and`. Par exemple, la fonction `pair` suivante détermine (de façon bien compliquée !) si un entier est pair :

CAML
```
let rec pair n =
  match n with
```

```
    | 0 -> true
    | _ -> impair (n - 1)
and
  impair n =
  match n with
    | 0 -> false
    | _ -> pair (n - 1)
;;
```

Exercice 2-1

Que font les fonctions suivantes?

CAML
```
let rec mystere n =
  match n with
    | 0 -> print_newline()
    | _ ->
        print_int n; print_char ‘ ‘;
        mystere (n-1);
        print_int n;
        print_char ‘ ‘;;

mystere 5;;
```

CAML
```
(* revoie le caractère en minuscule *)
let minuscule c =
  char_of_int (32 + int_of_char c);;

let rec mystere s i =
    if i = string_length s then print_newline()
    else  begin
      print_char s.[i];
      s.[i] <- minuscule s.[i];
      mystere s (i + 1);
      print_char s.[i]
    end;;

mystere "BONJOUR" 0;;
```

2.2 Terminaison et correction d’une fonction récursive

Il n’est pas évident à priori que le calcul d’une fonction récursive se termine. Dans les exemples simples que l’on a vu jusqu’à présent, il était clair que le nombre d’appels récursifs était fini, mais que penser de la fonction suivante?

CAML
```
let rec collatz n =
  match n with
    | n when n <= 1 -> 0
    | n when n mod 2 = 0 -> collatz (n / 2)
    | _ -> collatz (3 * n + 1);;
```

La conjecture de Collatz dit que cette fonction se termine pour toute valeur de n, mais ce résultat n’a jamais été prouvé ...

Un résultat classique de calculabilité nous dit qu'il est impossible d'écrire une fonction qui teste si une fonction s'arrêtera ou non (indécidabilité de la terminaison). L'argument suivant est assez convaincant : s'il existait une fonction `termine` qui prend le code source d'une fonction quelconque et qui dit si la fonction f s'arrête ou non, on pourrait considérer la fonction suivante :

CAML
```
let rec absurde () =
  match termine "absurde" with
  | true -> absurde()
  | false -> 1;;
```

- Si la fonction `absurde` se termine, `termine "absurde"` vaut `true`, et donc la fonction `absurde` ne se termine pas ;
- Si la fonction `absurde` ne se termine pas, alors `termine "absurde"` vaudrait `false` et dans ce cas, la fonction `absurde` se termine.

Dans les deux cas, on obtient une contradiction.
En conclusion, c'est au programmeur qu'incombe la preuve de terminaison de sa fonction !
Une méthode classique simple pour montrer qu'une fonction récursive se termine, consiste à exhiber une *graduation* sur l'ensemble de départ de la fonction, qui à un élément associe un entier, et à montrer que la fonction s'appelle récursivement sur un argument de taille strictement inférieure. Alors, en notant a_n la taille de la énième valeur à calculer récursivement, on obtient une suite d'entiers strictement décroissante. Elle doit être finie, et donc les appels récursifs sont en nombre fini.
Par exemple, pour les entiers, la graduation utilisée peut être l'identité. Sur les couples d'entiers, on peut par exemple utiliser

$$grad : \left\{ \begin{array}{ccc} \mathbb{N}^2 & \longrightarrow & \mathbb{N} \\ (n,p) & \mapsto & n+p \end{array} \right.$$

ou encore :

$$grad : \left\{ \begin{array}{ccc} \mathbb{N}^2 & \longrightarrow & \mathbb{N} \\ (n,p) & \mapsto & \max(n,p) \end{array} \right.$$

Voyons une généralisation de cette idée qui sera d'un emploi plus simple par la suite :

Définition 2.1: **Ordre bien fondé**
Soit E un ensemble, et $\preceq$ une relation d'ordre sur E (pas nécessairement totale). On note $\prec$ l'ordre strict correspondant :

$$\forall x \in E, \quad x \prec y \Longleftrightarrow x \preceq y \text{ et } x \neq y$$

On dit que l'ordre $\preceq$ est *bien fondé* s'il n'existe pas de suite d'éléments de E strictement décroissante.

Définition 2.2: **Élément minimal**
Soit un ensemble ordonné $(E, \preceq)$. Un élément $m \in E$ est dit *minimal* si et seulement si :

$$\forall a \in A, \quad a \preceq m \Rightarrow a = m$$

Remarque 3. Si l'ordre $\preceq$ est *total*, alors un $m \in E$ est minimal si et seulement si c'est le plus petit élément de E.

`Exemple 1.` *Sur* $\mathbb{N} \setminus \{0,1\}$*, on peut définir l'ordre de divisibilité :*

$$n \preceq p \Longleftrightarrow n \text{ divise } p$$

C'est un ordre bien fondé qui n'est pas total. Les éléments minimaux sont les nombres premiers. Il n'y a pas de plus petit élément pour cet ordre.

Théorème 2.1: **Caractérisation des ordres bien fondés**
L'ordre $\preceq$ est bien fondé si et seulement si toute partie A non-vide de E possède un élément minimal.

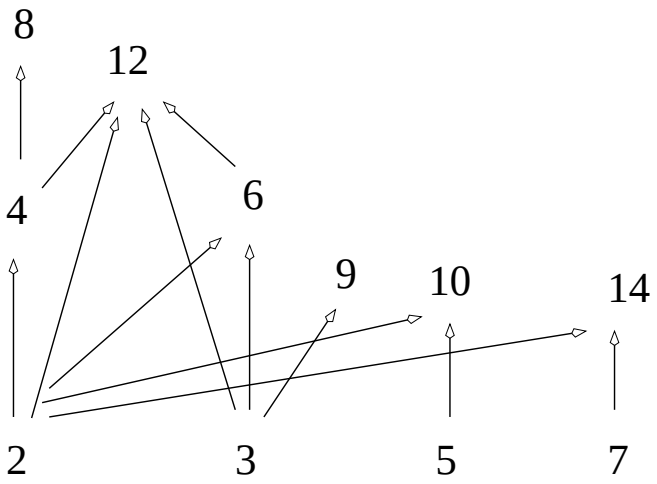
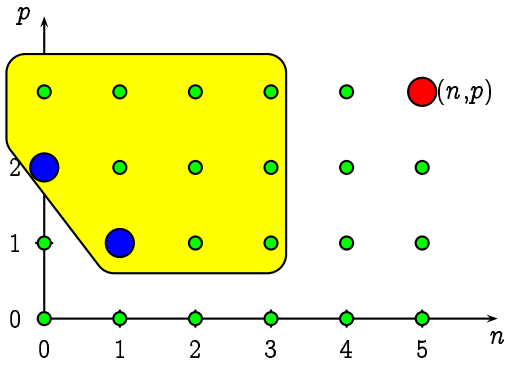


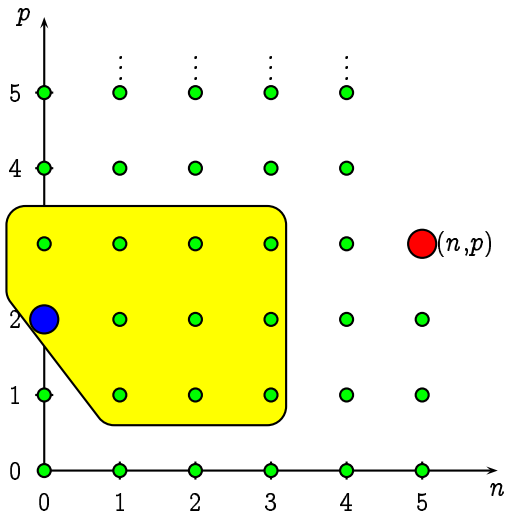
FIG. 2.1 – *Ordre de la divisibilité sur* $\mathbb{N} \setminus \{0,1\}$

Exemple 2. *Sur* $\mathbb{N}^2$*, les ordres suivants sont bien fondés :*

- *L'ordre produit :* $(n,p) \preceq (n',p') \iff n \leq n'$ *et* $p \leq p'$*. Cet ordre n'est pas total (par exemple on n'a ni* $(1,0) \preceq (0,1)$ *ni* $(0,1) \preceq (1,0)$*).* $(0,0)$ *est le seul élément minimal pour cet ordre ;*



- *L'ordre lexicographique :* $(n,p) \preceq (n',p') \iff n \leq n'$ *ou* $n = n'$ *et* $p \leq p'$*. C'est un ordre total (deux éléments quelconques de* $\mathbb{N}^2$ *sont comparables pour cet ordre). Le seul élément minimal est* $(0,0)$*.*



Exercice 2-2

On peut définir sur $\mathbb{N}^2$ l'ordre bien fondé suivant :

$$(n,p) \preceq (n',p') \iff n \leq n'$$

Ce n'est pas un ordre total. Quels sont les éléments minimaux pour cet ordre?

CAML

```
Ce sont tous les couples de la forme $(0, p)$ où $p \in \nn$.
```

THÉORÈME 2.2: **Induction sur un ensemble bien fondé**
Soit $(E, \preceq)$ un ensemble bien fondé, et $\mathbf{p}$ un prédicat sur E. On note $\mathcal{B}$ l'ensemble des éléments minimaux de E. Si l'on a :

1. $\forall x \in \mathcal{B}$, $\mathbf{p}(x)$,
2. $\forall x \in E$, $\big(\forall y \prec x, \mathbf{p}(y)\big) \Rightarrow \mathbf{p}(x)$,

Alors $\forall x \in E$, $\mathbf{p}(x)$.

THÉORÈME 2.3: **Justification de la terminaison d'une fonction récursive**
Soit $(E, \preceq)$ un ensemble muni d'un ordre bien fondé, $f : A \mapsto B$ une fonction récursive et $\phi : A \mapsto E$ une application. On note

$$\mathcal{M} = \{x \in A \mid \phi(x) \text{ est minimal dans } \phi(A)\}$$

On fait les hypothèses suivantes :

1. le calcul de $f(x)$ se termine pour tous les éléments $x \in \mathcal{M}$;
2. pour tout $x \in A$, le calcul de $f(x)$, n'utilise qu'un nombre fini de calculs $f(y_1) \dots f(y_k)$ où $\phi(y_i) \prec \phi(x)$.

Alors le calcul de $f(x)$ se termine pour toute valeur $x \in A$.

THÉORÈME 2.4: **Preuve de correction d'une fonction récursive**
Soit $f : A \mapsto B$ une fonction récursive, et $\phi : A \mapsto E$ une application où $\preceq$ est un ordre bien fondé sur E. On note

$$\mathcal{M} = \{x \in A \mid \phi(x) \text{ est minimal dans } \phi(A)\}$$

Soit $\mathbf{p_f}(x)$ un prédicat faisant intervenir $f(x)$. On fait les hypothèses suivantes :

1. $\forall x \in \mathcal{M}$, $\mathbf{p_f}(x)$ est vrai ;
2. Si $x \in A$, le calcul de $f(x)$ n'utilise qu'un nombre fini de calculs de $f(y_1) \dots f(y_k)$ avec $\phi(y_1) \prec \phi(x), \dots, \phi(y_k) \prec \phi(x)$ et

$$\mathbf{p_f}(y_1) \dots \mathbf{p_f}(y_k) \Rightarrow \mathbf{p_f}(x)$$

alors, $\forall x \in A$, $\mathbf{p_f}(x)$ est vrai.

On montre simultanément la terminaison et la correction d'une fonction récursive en utilisant ces théorèmes. En réalité, il est plus simple de faire d'abord le raisonnement inductif car l'écriture de la fonction récursive en découle directement. Le raisonnement typique est le suivant :

1. trouver un ordre bien fondé adapté au problème ;
2. quels sont les cas de base pour lesquels on renvoie directement le résultat ? Les éléments « minimaux » sont-ils bien inclus dans les cas de base ?
3. (la partie cruciale). Soit x un élément arbitraire. En supposant que l'on sache calculer $f(y)$ pour tous les éléments $y \prec x$, comment en déduire le calcul de $f(x)$ en n'utilisant qu'un nombre fini d'éléments strictement inférieurs ?
4. la terminaison et la correction de la fonction découlent des raisonnements précédents.

Exercice 2-3

Ecrire deux fonctions
`binomial` de type : `int ->int ->int`
telles que `binomial n p` retourne l'entier $\binom{n}{p}$.

Exercice 2-4

Montrer que $\forall (x,n) \in \mathbb{N}^2$, la fonction suivante calcule x^n :

CAML
```
let rec exp x n =
  match n with
```

```
    | 0 -> 1
    | _ - > x * exp x (n - 1);;
```

Exercice 2-5

La fonction d'Ackermann est célèbre en informatique pour sa croissance extrêmement rapide :

CAML
```
let rec ackermann n p =
  match (n, p) with
    | 0, p -> p + 1
    | n, 0 -> ackermann (n - 1) 1
    | n, p -> ackermann (n-1) (ackermann n (p - 1));;
```

a) Montrer la terminaison de cette fonction.
b) Montrer que $\forall(n,p) \in \mathbb{N}^2$, ackermann$(n,p) > p$.

Exercice 2-6

Le problème des tours de Hanoï. On dispose de trois tiges sur lesquelles s'enfilent des disques de taille différente. On définit la contrainte suivante : sur chaque tige, on ne peut empiler un disque que si son diamètre est plus petit que ceux des disques déjà empilés sur cette tige. Au départ, tous les disques se trouvent sur la tige n°1. Il faut empiler tous les disques sur la tige n°3.
a) Ecrire une fonction
`hanoi` de type : `hanoi : string ->string ->string ->int ->unit`
telle que `hanoi "A" "B" "C" n` affiche la solution pour n disques :

```
#hanoi "A"  "B" "C" 3;;
A --> C
A --> B
C --> B
A --> C
B --> A
B --> C
A --> C
- : unit = ()
```

On utilisera la fonction suivante qui imprime les déplacements :

CAML
```
    let deplace de vers =
    print_string  (de ^ " --> " ^ vers );
    print_newline();;
```

b) Déterminer le nombre de déplacements nécessaires pour résoudre le problème avec n disques.

2.3 Induction structurelle

En informatique théorique, on construit souvent un ensemble $\mathcal{I}$ d'objets d'un ensemble E par « productions ».

1. On se donne un ensemble $\mathcal{B} \subset E$ d'objets de *base* ;
2. On se donne un ensemble de *constructeurs*. A partir de k objets $x_1, \ldots, x_k \in E$, un constructeur produit un nouvel objet $\mathbf{C}(x_1, \ldots, x_k) \in E$. Le constructeur $\mathbf{C}$ est dit d'*arité* k s'il s'applique à k objets ;
3. $\mathcal{I}$ est le sous-ensemble de E qui contient tous les objets obtenus par application d'un nombre fini de fois les constructeurs à partir des objets de la base $\mathcal{B}$.

Remarque 4. C'est également le plus petit sous-ensemble de E contenant B et stable par chaque constructeur.

Exemple 3. *On peut définir les entiers naturels en les « comptant » de la façon suivante :*

1. *Un seul objet de base, l'objet* 0 *;*
2. *Un seul constructeur d'arité* 1 *:* Succ *qui prend un objet et qui lui associe son « successeur ».*

L'entier 3 *correspond à l'objet obtenu par trois applications successives du constructeur à l'élément de base* 0.

Exemple 4. *En calcul formel, on peut définir une « expression algébrique » (simplifiée) de la façon suivante :*

- *On définit l'ensemble* $\mathcal{B}$ *des « variables » ;*
- *On définit les constructeurs suivants :*
  - Plus *(d'arité* 2*) : Si* A1 *et* A2 *sont deux expressions algébriques,* Plus(A1, A2) *est encore une expression algébrique,*
  - Mult *(d'arité* 2*) : Si* A1 *et* A2 *sont deux expressions algébriques,* Mult(A1, A2) *est encore une expression algébrique,*
  - Exp *(d'arité* 1*) : Si* A *est une expression algébrique,* Exp A *est encore une expression algébrique.*

Par exemple, l'expression algébrique :

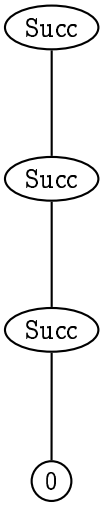
```
(a+b)*(exp(a))*(exp(c)+b)
```

se construit de la façon suivante :

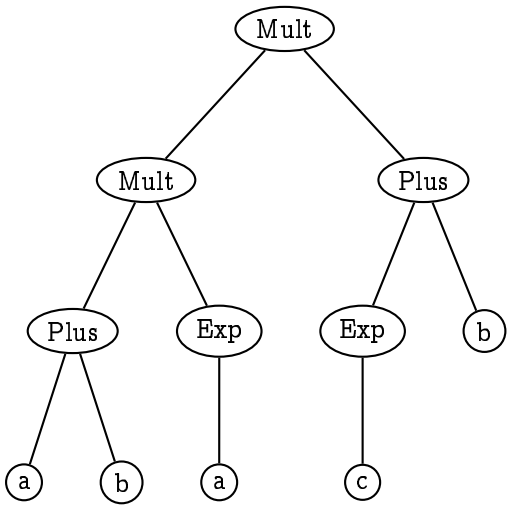
1. a *et* b *sont des variables de* $\mathcal{B}$ *donc* A1 = Plus(a,b) *est une expression algébrique ;*
2. a *est une variable, donc* A2 = Exp(a) *est une expression algébrique ;*
3. *Comme* A1 *et* A2 *sont des expressions algébriques,* A3 = Mult(A1,A2) *en est encore une ;*
4. *Comme* c *est une variable,* A4 = Exp(c) *est une expression algébrique ;*
5. *Comme* b *est une variable et que* A4 *est une expression,* A5 = Plus(A4,b) *est une expression algébrique ;*
6. *Comme* A3 *et* A5 *sont des expressions,* Mult(A4,A5) *est une expression.*

On peut représenter un élément d'un ensemble inductif par un *arbre*, décrivant sa production à l'aide des constructeurs. On place aux feuilles de l'arbre les éléments de $\mathcal{B}$, et aux noeuds, les constructeurs utilisés.

Dans l'exemple des entiers, l'entier 3 obtenu par application successive du constructeur Succ se représente par l'arbre suivant :



Et notre expression algébrique se représente par l'arbre :



Remarque 5. Une question importante est de savoir si l'arbre représentant un objet inductif est unique ou non.

Pour prouver qu'une propriété $\mathcal{P}(x)$ est vraie pour tout objet x d'un ensemble construit par induction, on utilise le raisonnement suivant :

THÉORÈME 2.5: **Raisonnement par induction structurelle**
Soit $\mathcal{P}(x)$ un prédicat sur un ensemble $\mathcal{I}$ défini par induction. Si :

1. $\mathcal{P}(x)$ est vrai pour tout élément x de $\mathcal{B}$;
2. Pour tout constructeur $\mathbf{C}$ d'arité k, pour tout k-uplet d'éléments $(x_1,\ldots,x_k)$ de I :

$$\Big(\forall i \in [1,k], \mathcal{P}(x_i) \text{ vraie}\Big) \Rightarrow \Big(\mathcal{P}\big(\mathbf{C}(x_1,\ldots,x_k)\big) \text{ vraie } \Big)$$

Alors la propriété $\mathcal{P}$ est vraie pour tout élément de I.

En d'autres termes, il suffit de :

- Vérifier la propriété sur les objets de base ;
- Vérifier que la propriété est encore vraie après application de chaque règle de production.

2.4 Listes

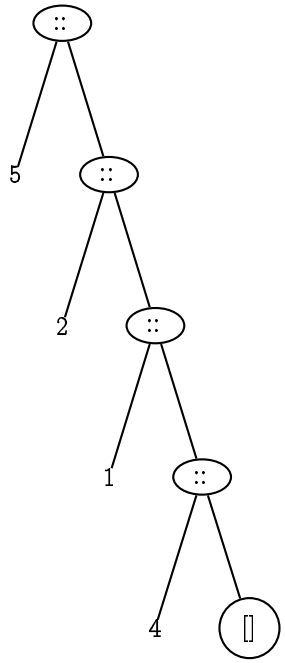
Pour définir un vecteur, il faut préciser sa taille lors de la création, et celle-ci ne changera plus. Les listes chaînées sont une structure de données homogènes plus souple : elles servent à garder en mémoire un nombre fini d'éléments de même type, mais leur taille peut varier dynamiquement. Une liste d'éléments de type `'a` se définit inductivement avec :

- un seul élément atomique, la liste vide notée `NIL` (`[]` en CAML) ;
- une collection de constructeurs paramétrés par les éléments x de type `'a` : `CONS_x` d'arité 1 qui prennent en argument une liste q et produisent une nouvelle liste `CONS_x(q)` notée `x :: q` en CAML.

Par exemple, on produit les listes suivantes d'entiers par applications successives de constructeurs :

1. `l0 = [];`
2. `l1 = CONS_4(l0) = 4 :: [];`
3. `l2 = CONS_1(l1) = 1 :: 4 :: [];`
4. `l3 = CONS_2(l2) = 2 :: 1 :: 4 :: [];`
5. `l4 = CONS_5(l3) = 5 :: 2 :: 1 :: 4 :: [].`

La liste $l4$ se note `[5; 2; 1; 4]` en CAML, et peut être représentée par l'arbre suivant :



Formellement le type liste peut se décrire par la formule :

$$L = NIL + element \times L$$

Le type liste est prédéfini en CAML ainsi que les fonctions `tl` (« tail ») et `hd` (« head ») qui renvoient la queue et la tête d'une liste :

CAML
```
#let l = [0; 2; 3] ;;
l : int list = [0; 2; 3]
#tl l;;
- : int list = [2; 3]
#hd l;;
- : int = 0
#4 :: l;;
- : int list = [4; 0; 2; 3]
```

Nous n'utiliserons pas ces fonctions, mais plutôt le filtrage sur les listes. Le raisonnement récursif typique pour écrire une fonction sur les listes est le suivant ; Une liste est soit :

- vide ;
- de la forme `x :: q` où q est la queue (une liste) et x la tête (un élément).

Le squelette d'une fonction récursive typique sur une liste est donc :

CAML
```
let rec f l =
match l with
| [] -> ...
| x :: q -> ...;;
```

La preuve de correction et de terminaison d'une fonction récursive de ce type sur une liste se fait par induction :

THÉORÈME 2.6: **Fonctions sur les listes**

1. On vérifie que la fonction renvoie le bon résultat pour la liste vide ;
2. On suppose que la fonction renvoie le bon résultat pour une liste q et on montre que la fonction renvoie le bon résultat pour la liste `x :: q`.

Alors la fonction se termine et renvoie le bon résultat pour toute liste.

C'est en écrivant la preuve par induction que l'on trouve comment écrire les fonctions sur les listes.
Une fonction qui compte le nombre d'éléments d'une liste :

CAML
```
let rec longueur l =
  match l with
    | [] -> 0
    | x :: q -> 1 + longueur q;;
```

Une fonction qui concatène deux listes :

CAML
```
#let rec concatene l1 l2 =
  match l1 with
    | [] -> l2
    | x :: q -> x :: concatene q l2;;

concatene : 'a list -> 'a list -> 'a list = <fun>
```

Cette fonction est prédéfinie en CAML par l'opérateur infixe `@` :

CAML
```
#let l1 = [1; 2]
and l2 = [3; 4]
in l1 @ l2;;

- : int list = [1; 2; 3; 4]
```

Exercice 2-7
Déterminer le nombre $T(n_1,n_2)$ d'appels récursifs de la fonction précédente, si $|l1| = n_1$ et $|l_2| = n_2$.

Une fonction qui retourne l'image miroir d'une liste l :

CAML
```
  let rec insere_fin a l =
  match l with
    | [] -> [a]
    | x :: q -> x :: (insere_fin a q);;

let rec miroir l =
  match l with
    | [] -> []
    | x :: q -> insere_fin x (miroir q);;
```

Si n désigne le nombre d'éléments de la liste l (*longueur* de l), et si $S(n)$ est le nombre d'appels récursifs de la fonction `insere_fin a l` on a la relation de récurrence :

$$\begin{aligned} S(0) &= 0 \\ \forall n \geq 1 \quad S(n) &= 1 + S(n-1) \end{aligned}$$

d'où l'on tire $S(n) = n$.
Si l'on note $T(n)$ le nombre d'appels récursifs pour calculer `miroir l` (une mesure de la complexité temporelle), on a la relation de récurrence :

$$\begin{aligned} T(0) &= 0 \\ \forall n \geq 1 \quad T(n) &= 1 + S(n-1) + T(n-1) \end{aligned}$$

d'où

$$\begin{aligned} T(n) &= 1 + (n-1) + T(n-1) \\ &= n + (n-1) + T(n-2) \\ &\vdots \\ &= n + (n-1) + \cdots + 1 + T(0) \\ &= n + (n-1) + \cdots + 1 \\ &= \frac{n(n+1)}{2} \\ &= \Theta(n^2) \end{aligned}$$

Si l'on note $C(n)$ le nombre d'utilisations du constructeur `::` (une mesure de la complexité spatiale), on a la relation de récurrence :

$$\begin{aligned} C(0) &= 0 \\ \forall n \geq 1 \quad C(n) &= 1 + C(n-1) \end{aligned}$$

d'où l'on tire $C(n) = n$.
On peut également utiliser d'autre types de filtrage, en distinguant la liste vide, une liste à un élément et le cas général :

CAML
```
let f l =
match l with
| [] -> ...
| [x] -> ...
| x :: y :: q -> ... ;;
```

Un filtrage sur un couple de listes distingue les cas suivants :

- les deux listes sont vides ;
- la première liste est vide, mais pas la deuxième ;
- la deuxième liste est vide, mais pas la première ;
- aucune des deux listes n'est vide.

Le squelette de la fonction correspondante est :

CAML
```
let f l1 l2 =
match (l1, l2) with
| [], [] -> ...
| [], _ -> ...
| _, [] ->
| x1 :: q1, x2 :: q2 -> ... ;;
```

Nous verrons des exemples en TD.

2.5 Implémentation de listes dans d'autres langages

Dans les langages de programmation traditionnels (C ou Pascal), une liste s'implémente à l'aide de *pointeurs*. Un pointeur est une adresse mémoire d'une taille fixée (un peu comme les références en CAML). On peut définir une liste chaînée par des cellules, formées d'un élément x et d'un pointeur vers une autre cellule. Il existe un pointeur particulier qui ne pointe vers « rien », le pointeur NIL. Une liste est alors représentée par un pointeur qui pointe vers la première cellule. En CAML, le programmeur n'a pas à s'occuper de la gestion des pointeurs, et les fonctions écrites sur les listes sont beaucoup plus claires.

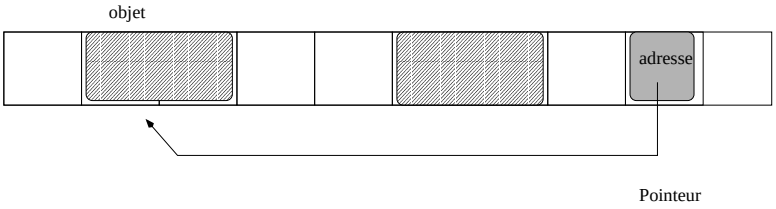


FIG. 2.2 – *Pointeur*

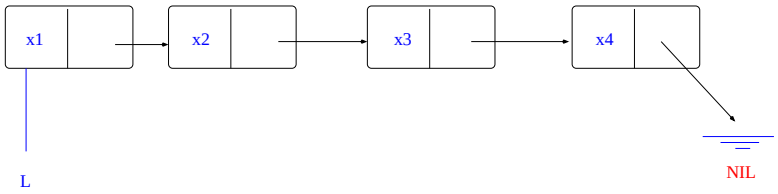


FIG. 2.3 – *Liste*

2.6 Programmation fonctionnelle contre programmation impérative

Un théorème de la théorie de la calculabilité nous dit que la programmation impérative et la programmation récursive calculent les mêmes choses. On voit que l'on peut écrire la fonction `factorielle` de façon impérative, mais est-ce aussi simple pour la fonction `hanoi` ?
En programmation impérative, on utilise souvent des données « persistantes », par exemple une référence dont le contenu est modifié par des instructions du programme (affectations ...). Une « procédure » est une suite de modifications de l'état de la machine pour arriver à un état final.
En programmation fonctionnelle, à chaque appel récursif, l'environnement de la fonction est vierge et il n'est pas possible d'utiliser de « mémoire ». On utilise plutôt le passage d'arguments, et des fonctions auxiliaires.
Dans certains cas, le choix d'une fonction récursive est catastrophique vu sa complexité ; par exemple, la fonction suivante calcule le nième nombre de fibonacci défini par :

$$F_0 = F_1 = 1 \text{ et } \forall n \geq 2, F_n = F_{n-1} + F_{n-2}$$

CAML
```
let rec fibo n =
match n with
| 0 -> 1
| 1 -> 1
| _ -> fibo (n - 1) + fibo (n - 2);;
```

CAML
```
#trace "fibo";;
La fonction fibo est dorénavant tracée.
- : unit = ()
#fibo 5;;
fibo <-- 5
fibo <-- 3
fibo <-- 1
fibo --> 1
fibo <-- 2
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo --> 3
fibo <-- 4
fibo <-- 2
```

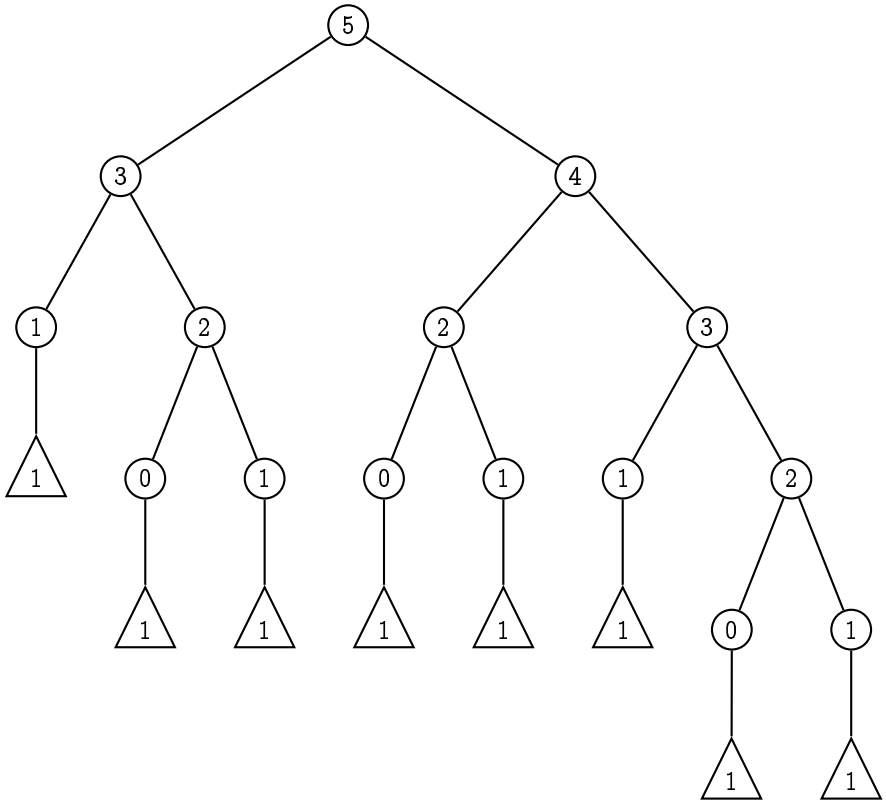


FIG. 2.4 – *Appels récursifs de fibo* 5

```
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo <-- 3
fibo <-- 1
fibo --> 1
fibo <-- 2
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo --> 3
fibo --> 5
fibo --> 8
- : int = 8
```

Cette fonction effectue plusieurs appels récursifs inutiles pour calculer `fibo 3`. La figure 2.4 résume sous forme d'arbre les différents appels récursifs.

2.6.1 Récursivité terminale

Au niveau du microprocesseur, il faut comprendre comment est évaluée une fonction récursive. La fonction suivante calcule $n!$:

CAML

```
let rec fact n =
  match n with
```

```
    | 0 -> 1
    | n -> n * fact (n - 1);;
```

A chaque appel de fonction, un nouvel emplacement mémoire est crée qui contient la valeur des arguments, et un endroit pour placer la valeur de retour de la fonction. Si l'on demande d'évaluer `fact 3`, une cellule `(3, _)` est crée, mais pour remplir la valeur de retour, il faut connaître la valeur de retour de `fact 2`. Un nouvel emplacement `(2,_)` est donc crée et ainsi de suite (voir figure 2.5). Ces cellules sont successivement « empilées », puis « dépilées » jusqu'à pouvoir évaluer `fact 3`. A chaque appel récursif, la hauteur de la pile augmente, et il se peut que la place mémoire disponible soit dépassée. On a droit alors à un message de *dépassement de pile* (stack overflow). Il existe une classe de fonctions récursives qui peuvent s'évaluer avec

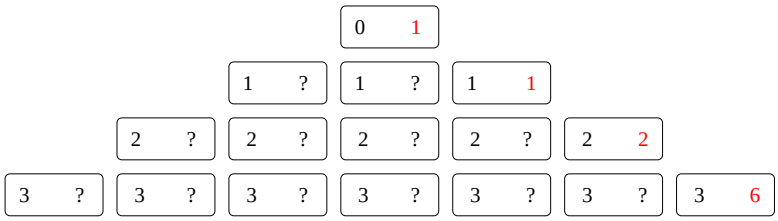


FIG. 2.5 – *Calcul de* 3!

un espace mémoire constant : les fonctions récursives terminales.

DÉFINITION 2.3: **Fonctions récursives terminales**
On dit qu'une fonction récursive est *terminale* si le seul appel récursif qu'elle fait se trouve en dernière position de la fonction et si cet appel est simple.

La fonction suivante est récursive terminale :

CAML
```
let rec terminale n =
    match n with
     | 0 -> print_newline()
     | _ -> print_int n; terminale (n - 1);;
```

alors que celle-ci ne l'est pas :

CAML
```
  let rec  non_terminale n =
  match n with
    | 0 -> print_newline();
    | _ ->  non_terminale (n - 1); print_int n;;
```

Les fonctions récursives terminales sont intéressantes car elles ne nécessitent qu'un espace mémoire constant lors de leur exécution (la hauteur de la pile d'exécution ne croît pas avec la taille de la donnée); en effet, on peut « oublier » la valeur des arguments lors de l'appel récursif.
Il existe une transformation automatique qui permet de remplacer une fonction récursive terminale par un programme impératif utilisant une boucle et une référence. Supposons que le schéma d'une fonction récursive terminale f (à un argument x) soit de la forme suivante :

```
let rec f x =
  match cas_de_base x with
  | true -> f_base x;
  | false  -> action x; f (sigma x);;
```

où `cas_de_base` désigne un prédicat correspondant à l'ensemble des cas terminaux, `f_base` une certaine fonction, `action x` une série d'effets de bords dépendant de la valeur de x, et `sigma x` un argument strictement inférieur à x (pour un ordre bien fondé). Alors la procédure schématique suivante effectue les mêmes calculs que f :

```
let proc x =
```

```
let arg = ref x in
while not (cas_de_base !arg) do
   action !arg;
   arg := sigma !arg
done;
f_base !arg;;
```

Par exemple, la fonction récursive terminale de l'exemple ci-dessus se dérécursifie de la façon suivante :

CAML
```
let terminale_imp n =
  let arg = ref n in
    while !arg <> 0 do
      print_int !arg;
      arg := !arg - 1
    done;
    print_newline();;
```

Les compilateurs fonctionnels modernes reconnaissent et optimisent automatiquement les fonctions récursives terminales.
Plus généralement, il existe une théorie de la *dérécursification* qui permet de transformer une fonction récursive quelconque en une procédure impérative équivalente. De nombreux algorithmes complexes s'expriment naturellement de façon récursive, et l'écriture d'une fonction récursive correspond à la description de ces algorithmes. En dérécursifiant ces fonctions, on obtient des programmes plus efficaces, que l'on peut ensuite encore optimiser.

2.6.2 Accumulateurs

La fonction factorielle écrite sous la forme usuelle n'est pas récursive terminale ; il faut connaître l'évaluation de `fact (n - 1)` avant de pouvoir retourner le résultat de `fact n` qui utilise la valeur n gardée en suspens.
On peut programmer cette fonction dans le style impératif en utilisant une référence :

CAML
```
let fact_i n =
  let r = ref 1 in
    for i = n downto 1 do
      r := !r * i
    done;
    !r;;
```

Une optimisation classique en programmation fonctionnelle consiste à transformer une fonction récursive non-terminale en une fonction récursive terminale équivalente. Pour cela, on utilise un *accumulateur*, c'est à dire un argument supplémentaire. Écrivons une fonction auxiliaire `fact_t` à deux arguments telle que `fact_t n x` calcule $x \times n \times (n-1) \times \cdots \times 1$:

CAML
```
let rec fact_t n x =
  match n with
    | 0 -> x
    | n -> fact_t (n - 1) (x * n);;
```

Il est alors simple de définir la factorielle à l'aide de cette fonction auxiliaire, il suffit d'appeler `fact_t` avec $x = 1$:

CAML
```
let fact n = fact_t n 1;;
```

Écrivons notre nouvelle fonction factorielle en définissant localement la fonction auxiliaire `fact_t` :

CAML
```
let fact n =
  let rec fact_t n x =
    match n with
```

```
      | 0 -> x
      | n -> fact_t (n - 1) (x * n)
  in
    fact_t n 1;;
```

Remarque 6. Cette écriture est inefficace, à cause de la dernière instruction `fact_t n 1`. A chaque calcul de `fact n`, une nouvelle fonction `fact_t` est évaluée (puisqu'elle dépend du paramètre n), ce qui nécessite un temps de calcul inutile. On peut éviter cela en écrivant :

CAML
```
let fact  =
  let rec fact_t  x n =
    match n with
      | 0 -> x
      | n -> fact_t (x * n) (n - 1)
  in
    fact_t 1 ;;
fact : int -> int = <fun>
```

La fonction `fact_t` est évaluée une seule fois et renvoie une fonction.
Il est possible d'améliorer la complexité de la fonction `miroir` (qui calculait l'image miroir d'une liste) en utilisant une fonction auxiliaire, `miroir_aux` qui prend deux listes pour arguments, une liste $l = [l_0; \ldots; l_n]$ et une liste accumulateur $accu = [a_0; \ldots; a_p]$ et renvoie la liste $[l_n; ...; l_0; a_0; ...; a_p]$. Il suffit alors d'appeler cette fonction avec un accumulateur vide :

CAML
```
  let rec miroir l =
  let rec miroir_aux l accu =
    match l with
      | [] -> accu
      | x :: q -> miroir_aux q (x :: accu)
  in
    miroir_aux l [];;
```

On peut éviter d'évaluer `miroir_aux` à chaque appel de `miroir l` en utilisant l'application partielle :

CAML
```
let  miroir  =
  let rec miroir_aux accu l =
    match l with
      | [] -> accu
      | x :: q -> miroir_aux (x :: accu) q in
    miroir_aux [];;
```

Pour évaluer la complexité de cette fonction, notons $T(n,p)$ le nombre d'appels récursifs nécessaires pour calculer `miroir_aux l accu` lorsque $|l| = n$ et $|accu| = p$. On obtient la relation de récurrence

$$\forall p \in \mathbb{N}, \quad T(0,p) = 0$$
$$\forall n \geq 1, \forall p \in \mathbb{N} \quad T(n,p) = 1 + T(n-1,p+1)$$

qui donne immédiatement $T(n,p) = n + T(0,p+n) = n$. Par conséquent, `miroir l` nécessite n appels récursifs alors que la première version que nous avions écrite nécessitait de l'ordre de n^2 appels récursifs.

Exercice 2-8

Ecrire une fonction `bits` qui renvoie la liste des chiffres en base 2 d'un entier n :

$$n = a_p.2^p + \cdots + a_1.2 + a_0 \mapsto [a_p; \ldots; a_0]$$

Chapitre 3

Analyse mathématique des algorithmes

3.1 Complexité

L'analyse de complexité d'un algorithme consiste à évaluer :

1. Son temps d'exécution (la complexité *temporelle*) ;
2. La place mémoire nécessaire (la complexité *spatiale*).

Ces deux quantités dépendent d'une « taille » des données à traiter, souvent exprimée à l'aide d'un entier n. Ce peut être le nombre d'éléments d'un tableau ou d'une liste, le nombre de bits d'un entier ... Un calcul exact du temps de calcul d'un algorithme dépend du matériel utilisé et est souvent compliqué. On calcule en général le nombre d'exécutions d'une opération particulière significative, par exemple le nombre de comparaisons pour le tri d'un tableau, ou le nombre d'appels récursifs pour une fonction récursive.

Les algorithmes sont souvent sensibles à d'autres facteurs qu'à la taille des données. Il est intéressant de définir trois notions de complexité. Notons D_n l'ensemble des données du problème de taille n, et $C(d)$ le coût de l'algorithme pour une donnée $d \in D_n$.

- La complexité *dans le pire des cas* :

$$\mathrm{Cmax}(n) = \max\{C(d)\,;\, d \in D_n\}$$

- La complexité *dans le meilleur des cas* :

$$\mathrm{Cmin}(n) = \min\{C(d)\,;\, d \in D_n\}$$

- La complexité *en moyenne* (une notion plus difficile, qui fait intervenir des probabilités) :

$$\mathrm{Cmoy}(n) = \sum_{d \in D_n} p(d)C(d)$$

où $p(d)$ représente la probabilité d'apparition de la donnée d parmi toutes les données de taille n.

Dans un algorithme de contrôle d'une centrale nucléaire, c'est la complexité dans le cas le pire qui est intéressante (on ne désire pas que le programme de contrôle réagisse après l'explosion). Dans un algorithme de recherche de pages web comme Alta Vista, c'est la complexité moyenne qui est la plus intéressante : on s'attend à ce que l'on effectue de nombreuses recherches ; tant pis si quelques unes n'aboutissent pas immédiatement, ce qui compte c'est la surcharge moyenne du serveur.

Les formules exactes de complexité sont en général compliquées, et on préfère donner une estimation asymptotique lorsque la taille des données est grande. Les notations de Landau sont bien adaptées pour cela :

DÉFINITION 3.1: On considère deux suites (u_n) et (v_n) de réels positifs.

- On dit que (u_n) est *dominée* par (v_n) et l'on note

$$u_n = O(v_n)$$

lorsque :

$$\exists M > 0, \exists n_0 \in \mathbb{N}, \text{ tq } \forall n \geq n_0, \quad u_n \leq M v_n$$

- On dit que (u_n) et (v_n) sont de *même ordre* et l'on note

$$u_n = \Theta(v_n)$$

lorsque :

$$u_n = O(v_n) \text{ et } v_n = O(u_n)$$

- On dit que (u_n) *domine* (v_n) lorsque $v_n = 0(u_n)$ et l'on note

$$u_n = \Omega(v_n)$$

Exemple 5.
a) Si $u_n = 4n^2 + n + 1$,

$$u_n = O(n^2) \quad u_n = O(n^3) \quad u_n = \Theta(n^2)$$

b) Si $u_n = 2n \log_2 n + n$, *alors*

$$u_n = \Theta(n . \log n)$$

Remarque 7. Bien qu'adaptées à l'analyse mathématique des algorithmes, les notations de Landau peuvent cacher des « coûts occultes ». Par exemple, si la complexité d'un algorithme est $T(n) = C.n$, il est considéré comme étant linéaire, donc efficace. Mais si la constante C est de l'ordre de 10^{10}, il va de soi, que ce n'est pas le cas.

On distingue traditionnellement les classes de complexité suivantes :

- complexité *logarithmique* : en $\Theta(\log n)$ (très efficace) ;
- complexité *linéaire* : en $\Theta(n)$ (efficace) ;
- complexité *quasi-linéaire* : en $O(n \ln n)$ mais pas en $O(n)$ (efficace) .
- complexité *polynômiale* : en $O(n^k)$ (moyennement efficace lorsque k est inférieur à 3, inefficace sinon). Les algorithmes matriciels sont dans cette classe) ;
- complexité *exponentielle* : $T(n) \geq a^n$ avec $a > 1$ (totalement inefficaces sauf pour de petites données).

Le tableau suivant donne une estimation de la complexité temporelle sur un ordinateur pouvant effectuer 10^6 opérations par secondes. On note ∞ lorsque la valeur dépasse 10^{100} a (a : années).

Classe de complexité / Taille	$\log_2 n$	n	$n.log_2 n$	n^2	n^3	2^n
$n = 10^2$	6,6 μs	0,1 ms	0,6 ms	10 ms	1 s	4×10^{16} a
$n = 10^3$	9,9 μs	1 ms	9,9 ms	1 s	16,6 min	∞
$n = 10^4$	13,3 μs	10 ms	0,1 s	100 s	11,5 j	∞
$n = 10^6$	19,9 μs	1 s	19,9 s	11,5 j	$31,7 \times 10^3$ a	∞

Remarque 8. Pour résoudre des problèmes compliqués, il n'est pas toujours possible de trouver un algorithme de complexité linéaire. Cependant, il existe assez souvent des algorithmes de complexité quasi-linéaire (basé en général sur une stratégie « diviser pour régner »). Ces algorithmes se comportent en pratique comme des algorithmes linéaires. En effet, pour des valeurs de n « humaines » ($< 10^{15}$), $log_2 n$ ne dépasse jamais quelques dizaines (< 35), et peut donc être considéré comme étant constant.

Par exemple, en traitement d'images, on utilise la *transformée de Fourier*. Pour traiter une image de taille $n = 1024 \times 768 = 7{,}810^5$,

- En utilisant la transformée de Fourier en n^2, le temps de calcul est de l'ordre de 172 h ;
- En utilisant la transformée de Fourier rapide en $n \log n$, le temps de calcul est de l'ordre de 15,4 s.

Remarque 9. Le tableau suivant résume la taille maximale des données que l'on peut traiter en un temps imparti, en fonction des classes d'algorithmes :

Classe de complexité Temps maximal	$\log_2 n$	n	$n.log_2 n$	n^2	n^3	2^n
1 s	∞	10^6	63×10^3	10^3	100	19
1 min	∞	6×10^7	28×10^5	77×10^2	390	25
1 h	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 j	∞	86×10^9	27×10^8	29×10^4	44×10^2	36

Remarque 10. La loi de Moore n'a pas été démentie jusqu'à présent : tous les 18 mois, la puissance de calcul des ordinateurs double. Même si cette loi continue à être vérifiée, dans 50 ans, les algorithmes exponentiels resteront toujours aussi inefficaces ! En effet, voici résumé dans le tableau suivant comment varient les limites de taille lorsque la puissance de l'ordinateur est multipliée par un facteur α (n représente la taille maximale des données du tableau précédent).

Complexité	$\log_2 n$	n	$n.log_2 n$	n^2	n^3	2^n
Taille maximale	n^α	$\alpha \times n$	$\sim \alpha \times n$	$\sqrt{\alpha} \times n$	$\sqrt[3]{\alpha} \times n$	$n + \log_2 \alpha$

Par exemple, si la vitesse de calcul de l'ordinateur est multipliée par 100, en 1 jour de calcul, cet ordinateur ne pourra pas résoudre un problème de complexité 2^n de taille supérieure à $n = 42 \ldots$

3.2 Etude d'un cas : recherche d'un élément dans un tableau

Recherche linéaire

On recherche si un élément x est dans un tableau t. Considérons la fonction CAML suivante :

CAML
```
let cherche x t =
  let n = vect_length t and i = ref 0 and trouve = ref false in
    while (!i < n && not !trouve) do
      trouve :=  t.(!i) = x ;
      i := !i + 1
      (* INV : trouve <==> x est dans {t.(0) ... t.(i-1)} *)
    done;
    !trouve;;

cherche : 'a -> 'a vect -> bool = <fun>
```

On note n la longueur du tableau t. La complexité est évaluée en comptant le nombre d'accès au tableau t. Dans le pire des cas cette complexité est $c(n) = n$. Dans le meilleur des cas cette complexité est $c(n) = 1$.

THÉORÈME 3.1: **Complexité en moyenne de la recherche linéaire**
On suppose que les tableaux considérés contiennent des éléments distincts, et que l'élément x recherché se trouve dans le tableau avec une probabilité q. On suppose également que si x est dans le tableau, il peut se trouver dans n'importe quel case avec la même probabilité $\frac{1}{n}$ Alors le nombre moyen d'accès pour chercher x dans un tableau de taille n vaut :

$$\text{Cmoy}(n) = (1-q)n + \frac{q}{2}(n+1)$$

En particulier, si l'on sait que l'élément x se trouve dans le tableau, le nombre moyen d'accès nécessaires vaut $\frac{n+1}{2}$.

Exercice 3-1
Modifier la fonction `recherche` pour qu'elle renvoie l'indice i lorsque $x = t.(i)$ et -1 si x n'est pas dans le

tableau.

Exercice 3-2

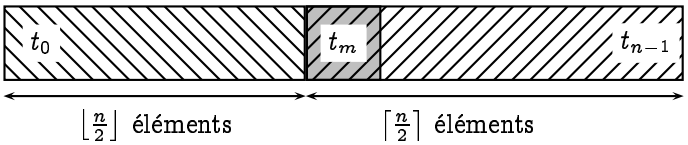
On suppose que les éléments du tableau sont des entiers compris entre 1 et p, et qu'un même élément peut apparaître à plusieurs endroits dans le tableau. Montrer que sous ces hypothèses, le nombre moyen de comparaisons pour chercher un élément $x \in [1,p]$ dans un tableau de taille n vaut

$$C_{moy}(n) = p - \frac{(p-1)^n}{p^{n-1}}$$

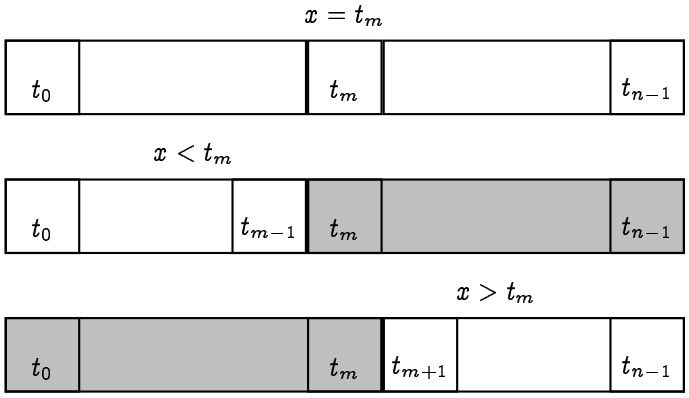
On pourra déterminer le nombre T_i de tableaux tels que l'élément x apparaisse pour la première fois à la position i et le nombre T_n de tableaux qui ne contiennent pas x.

Recherche dichotomique

On suppose le tableau t trié par ordre croissant. On cherche si un élément x est dans le tableau.
Si le tableau est de taille supérieure à 1, on compare x avec l'élément t_m situé au milieu du tableau :



1. Si $x = t_m$, on a trouvé x ;
2. Si $x < t_m$, x ne peut se trouver à droite de m et il suffit de chercher x parmi les $\lfloor \frac{n}{2} \rfloor$ éléments se trouvant avant t_m ;
3. Si $x > t.(m)$, x ne peut se trouver à gauche de m et il suffit de chercher x parmi les $\lceil \frac{n}{2} \rceil - 1$ éléments après t_m.



Remarque 11. Faisons une remarque utile pour les calculs qui vont suivre. Si n et p sont deux entiers, avec $p \neq 0$, alors

$$n/p = \left\lfloor \frac{n}{p} \right\rfloor$$

CAML

```
let recherche_dicho x t =
  let rec cherche_entre g d =
    if d < g  then false
    else
      let m = (g + d) / 2 in
        if x = t.(m) then true
        else
          if x < t.(m) then
            cherche_entre g (m - 1)
          else
```

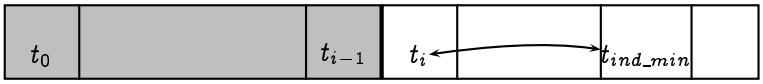


FIG. 3.1 – *tri sélection : les i premiers éléments sont déjà placés*

```
            cherche_entre (m + 1) d
  in
    cherche_entre 0 (vect_length t - 1);;
```

THÉORÈME 3.2: **Complexité de la recherche dichotomique**
Le nombre de comparaisons lors d'une recherche dichotomique est majoré par $2\lfloor \log_2 n \rfloor + 2$; la complexité de la recherche dichotomique est donc en en $O(\log n)$ dans le pire des cas.

Exercice 3-3

Que pensez-vous de la fonction `recherche_dicho` si l'on remplace le cas de terminaison

`if d < g  then false`

par

`if d = g then x = t.(g)`

Exercice 3-4

Ecrire une version impérative de `recherche_dicho`. Prouvez votre fonction grâce à un invariant de boucle.

Exercice 3-5

On veut trouver l'indice de la *première* occurrence de x s'il est présent dans le tableau. Comment modifier la recherche dichotomique pour cela? Prouver la terminaison de votre fonction, et déterminer le nombre maximal de comparaisons.

3.3 Tris élémentaires

On considère une série de données. A chaque donnée correspond une « clé », et l'on dispose d'un ordre sur les clés. Par exemple, une donnée pourrait être un enregistrement qui contient un nom, prénom ... d'un élève ainsi que sa moyenne annuelle en mathématiques. La clé d'une donnée serait la moyenne de l'élève.
Les données sont stockées dans un tableau, et nos algorithmes doivent modifier le tableau pour qu'à la fin, les données soient rangées par ordre croissant sur les clés.
Les algorithmes que nous allons voir n'utilisent que les comparaisons entre deux éléments du tableau, et effectuent des *transferts* d'éléments du tableau.
Remarque 12. On dit qu'un algorithme de tri est *stable* lorsqu'il ne modifie pas la place de deux éléments ayant une même clé. Par exemple, si le tableau contient des données d'élèves, initialement trié par ordre alphabétique sur les noms, après tri en fonction de la note de math, on souhaite que les élèves ayant même note apparaissent encore par ordre alphabétique (pour ne pas faire de jaloux) !
Nous allons voir quelques algorithmes classiques de tris. Pour simplifier, nous supposerons que les clés sont les éléments et que ce sont des entiers. Ces fonctions se généralisent facilement en passant en argument un prédicat de comparaison des clés.

3.3.1 Le tri par sélection

On cherche le plus petit élément de `t.(0)` à `t.(n-1)` que l'on échange avec `t.(0)`, puis le plus petit élément de `t.(1)` à `t.(n-1)` que l'on échange avec `t.(1)` et ainsi de suite.

CAML

```
let tri_selection t =
  let echange  i j =
    let temp = t.(i) in
```

```
    t.(i) <- t.(j);
    t.(j) <- temp
in
let n = vect_length t in
let ind_min = ref 0 in
  for i = 0 to n - 2 do
    ind_min := i;
    for j = i + 1  to n - 1 do
      if t.(j) < t.(!ind_min) then
          ind_min := j;
      (* INV t.(ind_min) = min{t.(i), ... , t.(j)} *)
    done;
    (* t.(ind_min) = min{t.(i) ... t.(n-1)} *)
    echange  i !ind_min;
    (* INV i < k < n ==> t.(0) <= ... <= t.(i) <= t.(k) *)
  done;;
```

THÉORÈME 3.3: **Analyse du tri par sélection**

- Le nombre de comparaisons du tri par sélection pour un tableau de taille n vaut :

$$C_{min}(n) = C_{moy}(n) = C_{max}(n) = n(n-1)/2 = \Theta(n^2)$$

- Le nombre de transferts du tri par sélection vaut :

$$T_{min}(n) = T_{moy}(n) = T_{max}(n) = 3(n-1) = \Theta(n)$$

- Le tri par sélection n'est pas stable.

Preuve. Cherchons le nombre de comparaisons nécessaires pour trier un tableau t à n éléments par cette méthode. A chaque passage dans la boucle

`for j = i + 1 to n - 1`

on effectue une comparaison. Par conséquent, le nombre de comparaisons vaut

$$C(t) = \sum_{i=0}^{n-1}(n-1-i) = \sum_{k=1}^{n-1} k = n(n-1)/2$$

Cherchons le nombre de transferts nécessaires. Chaque échange nécessite 3 transferts, et chaque passage dans la boucle

`for i = 0 to (n-2)`

nécessite un échange. Au total, il y a

$$T(t) = 3(n-1)$$

transferts. □

Remarque 13. Si le tableau à trier vaut $t = [|1_1; 1_2; 0|]$, après tri par sélection, le tableau trié est $t = [|0; 1_2; 1_1|]$ ce qui montre que ce tri n'est pas stable.

3.3.2 Le tri bulle

L'idée de cet algorithme est la suivante : le tableau `t` est trié si pour tout indice i entre 0 et $n-2$, on a

$$t.(i) \leq t.(i+1)$$

On va donc parcourir le tableau en faisant varier l'indice i du début à la fin (on dit qu'on fait une passe) en échangeant les éléments `t.(i)` et `t.(i+1)` chaque fois que $t.(i) > t.(i+1)$. On fait des passes tant que le tableau n'est pas trié.

On montre facilement qu'après le premier passage, le plus grand élément se trouve à la fin du tableau et donc qu'au second passage, on peut se contenter de parcourir le tableau de l'indice 0 à $n-3$, et ainsi de suite. Ainsi programmé on a

CAML
```
let tri_bulle t =
  let echange i j =
    let temp = t.(i) in
      t.(i) <- t.(j);
      t.(j) <- temp
  in
  let n = vect_length t  in
    for k = n-2 downto 0 do
      (* parcourt le tableau de 0 à k
         et échange deux éléments consécutifs
         s'ils ne sont pas dans le bon ordre *)
      for i = 0 to k do
        if t.(i) > t.(i+1) then
          echange i (i+1)
      done;
      (* INV: t.(0) ... t.(k) <= t.(k + 1)  *)
    done;;
```

Dans l'analyse d'un tri, il est clair que la valeur des clés n'est pas importante ; ce qui compte, c'est la position relative des enregistrements les uns par rapport aux autres. On peut associer à un tableau t, une permutation $\sigma = \left(\begin{smallmatrix} 1 & 2 & \cdots & n \\ \sigma(1) & \sigma(2) & \cdots & \sigma(n) \end{smallmatrix}\right) \in S_n$ représentant l'ordre des éléments dans le tableau. Par exemple, si

`t = [|10; 15; 5; 41; 20; 2|]`

on lui associe la permutation

$$\sigma = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 2 & 6 & 5 & 1 \end{smallmatrix}\right)$$

Chaque échange dans le tableau t se traduit par une transposition de deux éléments de la permutation associée.

DÉFINITION 3.2: **Inversions d'une permutation**
Soit une permutation $\sigma \in S_n$. On dit qu'un couple d'indices $(i,j) \in [1,n]^2$ avec $i < j$ est une *inversion* de σ lorsque $\sigma(i) > \sigma(j)$. Nous noterons $\mathrm{INV}(\sigma)$ le nombre d'inversions d'une permutation σ.

Par exemple, les inversions du tableau précédent sont les couples (1,3), (1,6), (2,5), (2,6), (3,6), (4,5), (4,6) et (5,6). $\mathrm{INV}(t) = 8$.
Le tri bulle échange deux éléments t_i et t_{i+1} successifs du tableau si et seulement si $(i,i+1)$ est une inversion de la permutation associée. Le nombre d'inversions décroît donc d'une unité à chaque échange. Comme à la fin le tableau est trié (donc sans inversions), le nombre total d'échanges effectués par le tri bulle est égal au nombre d'inversions du tableau initial. Pour déterminer le nombre d'échanges moyen du tri bulle, nous utilisons le résultat suivant :

THÉORÈME 3.4: **Nombre moyen d'inversions**
Le nombre moyen d'inversions parmi les permutations de S_n vaut $n(n-1)/4$.

Preuve. Soit une permutation $\sigma \in S_n$. Associons-lui sa « permutation miroir » σ' définie par

$$\sigma' = \begin{pmatrix} 1 & 2 & \dots & n \\ \sigma(n) & \sigma(n-1) & \dots & \sigma(1) \end{pmatrix}$$

Il est clair que (i,j) est une inversion de σ si et seulement si $(n+1-j, n+1-i)$ n'est pas une inversion de σ'. Le nombre d'inversions de σ et σ' est donc le nombre de couples (i,j) avec $i < j$ distincts : il y en a $\binom{n}{2} = n(n-1)/2$.
On peut séparer l'ensemble des permutations en deux classes disjointes A : les permutations σ avec $\sigma(1) < \sigma(n)$ et les autres B. L'application $\sigma \mapsto \widetilde{\sigma}$ est une bijection de A vers B.

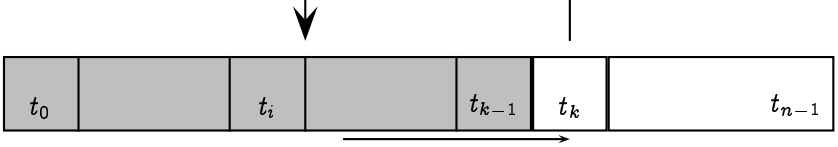


FIG. 3.2 – *tri insertion : insérer t_k à sa place et décaler les éléments*

En groupant les permutations de A avec leurs images miroir (dans B), on détermine le nombre moyen d'inversions :

$$\mathrm{INV}_{moy} = \frac{1}{n!}\sum_{\sigma\in S_n}\mathrm{INV}(\sigma) = \frac{1}{n!}\sum_{\sigma\in A}\mathrm{INV}(\sigma)+\mathrm{INV}(\sigma') = \frac{1}{n!}\binom{n}{2}|A| = \frac{1}{n!}\frac{n(n-1)}{2}\frac{n!}{2} = \frac{n(n-1)}{4}$$

□

THÉORÈME 3.5: **Analyse du tri bulle**

- Dans tous les cas, le nombre de comparaisons du tri bulle vaut :

$$C_{min}(n) = C_{moy}(n) = C_{max}(n) = n(n-1)/2 = \Theta(n^2)$$

- Le nombre de transferts du tri bulle vaut :
  - $C_{min}(n) = 0$ dans le meilleur cas (le tableau est trié),
  - $C_{moy}(n) = 3n(n-1)/4 = \Theta(n^2)$ dans le cas moyen,
  - $C_{max}(n) = 3n(n-1)/2 = \Theta(n^2)$ dans le cas le pire.
- Le tri bulle est stable.

Preuve. Cherchons le nombre de comparaisons pour trier un tableau t à n éléments par cette méthode. Chaque passage dans la boucle

`for i = 0 to k`

nécessite 1 comparaison. Il y a donc au total

$$C(t) = \sum_{k=0}^{n-2}(k+1) = \sum_{j=1}^{n-1} j = n(n-1)/2$$

comparaisons.
Déterminons maintenant le nombre de transferts. Chaque échange effectué diminue d'une unité le nombre d'inversions dans le tableau t. Il y a donc $\mathrm{INV}(t)$ échanges.
Dans le meilleur des cas (lorsque le tableau initial est trié), l'algorithme n'effectue aucun échange.
Dans le cas moyen, le nombre d'échanges vaut le nombre moyen d'inversions, c'est à dire $n(n-1)/4$.
Dans le pire des cas, lorsque le tableau est trié par ordre décroissant, le nombre d'inversions vaut $\binom{n}{2} = n(n-1)/2$. □

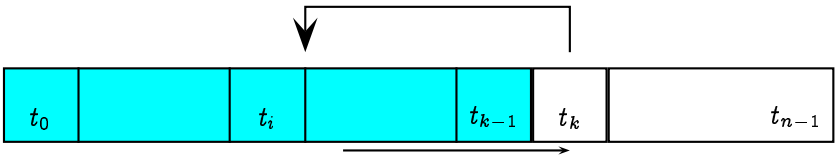
3.3.3 tri par insertion

C'est ainsi qu'un joueur de cartes trie sa main, en ajoutant les cartes une à une dans le jeu déjà trié.
Si le tableau t est trié de `t.(0)` à `t.(k-1)`, alors on insère l'élément `t.(k)` à la bonne place dans le tableau $[t_0;\ldots;t_{k-1}]$; ainsi le tableau sera trié de `t.(0)` à `t.(k)`.

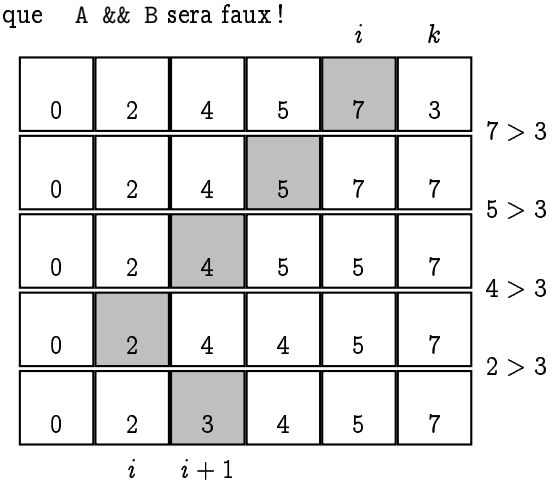
CAML

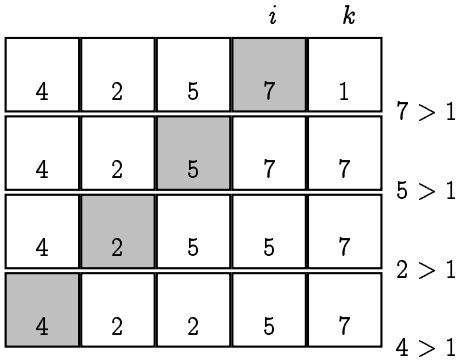
```
let tri_insertion t =
  let n = vect_length t
  and i = ref(1)
  and temp = ref 0 in
    for k = 1 to n - 1 do
```

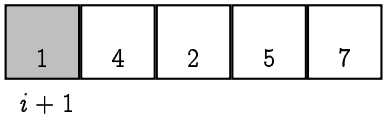
```
    (* insère t.(k) à sa place dans
       le sous-tableau trié t.(0) ... t.(k-1) *)
    temp := t.(k);
    i := k - 1;
    while (!i >= 0) && (t.(!i) > !temp) do
      t.(!i + 1) <- t.(!i);
      i := !i - 1;
      (* INV : -1 <= i
         && t(0) <= ... <= t.(i) <= t.(i+1) = t.(i+2) <= ... <= t.(k)
         && temp < t.(i+1)  *)
    done;
    (* (i = -1 && temp < t.(0) = t.(1) <= ... <= t.(k)
       ou alors
       i >= 0
       && t.(0) <= ... <= t.(i) <= t.(i+1) =t.(i+2) <= ... <= t.(k)
       && t.(i) <= temp < t.(i+1) *)
    t.(!i + 1) <- !temp;
    (* t.(0) <= ... <= t.(i) <= temp < t.(i+2) <= ... <= t.(k) *)
  done;;
```



Remarque 14. Si l'on avait inversé l'ordre des deux conditions `(t.(!i) > !temp) && (!i >= 0)` le programme aurait été incorrect : si i arrive à -1, au passage suivant dans la boucle, il faudrait évaluer $t.(-1)$ ce qui provoque un débordement.
Le programme précédent est correct grâce à l'*évaluation paresseuse* d'un booléen `A && B` : si A est faux, CAML n'évalue pas B, car il sait déjà que `A && B` sera faux !







Pour éviter à chaque passage dans la boucle `for k` de vérifier si $i >= 0$, on peut utiliser une *sentinelle*. On trie le tableau $t.(1)...t.(n-1)$ en plaçant en $t.(0)$ un élément qui est plus petit que tous les autres :

CAML

```
let tri_insertion_sentinelle t =
  let n = vect_length t and
    i = ref 1 and temp = ref 0 in
    for k = 2 to n - 1 do
      temp := t.(k);
      i := k - 1;
      while t.(!i) > !temp do
        t.(!i + 1) <- t.(!i);
        i := !i - 1;
      done;
      t.(!i + 1) <- !temp;
    done;;
```

THÉORÈME 3.6: **Analyse du tri par insertion**

- Le nombre de comparaisons du tri par insertion (avec sentinelle) pour trier un tableau de taille n vaut :
  - Dans le meilleur cas, $C_{min}(n) = n - 1 = \Theta(n)$,
  - Dans le cas moyen, $C_{moy}(n) = (n-1)(n+4)/4 = \Theta(n^2)$,
  - Dans le cas le pire, $C_{max}(n) = (n-1)(n+2)/2 = \Theta(n^2)$;
- Le nombre de transferts du tri par insertion vaut :
  - Dans le meilleur des cas, $T_{min}(n) = 2(n-1)$,
  - Dans le cas moyen, $T_{moy}(n) = (n-1)(n+8)/4 = \Theta(n^2)$,
  - Dans le cas le pire, $T_{max}(n) = (n-1)(n+4)/2 = \Theta(n^2)$;
- Le tri par insertion est stable.

Preuve. On fait l'analyse du tri par insertion sur un tableau $[|t_0; \ldots; t_{n-1}|]$ de longueur n, avec une sentinelle placée en t_{-1}. Pour insérer l'élément t_k à sa place parmi $t_0, \ldots, t_{k-1}$, il faut autant de comparaisons que d'éléments parmi $t_0, \ldots t_{k-1}$ qui sont strictement supérieurs à t_k, plus une comparaison supplémentaire —avec le premier élément $t_i \leq t_k$ (lorsque l'on n'utilise pas de sentinelle, il n'y a pas toujours cette comparaison supplémentaire). En notant I_k le nombre d'indices i avec $i < k$ et $t_i > t_k$, on obtient au total

$$C(t) = \sum_{k=1}^{n-1} (I_k + 1) = (n-1) + \sum_{k=1}^{n-1} I_k$$

Mais $\sum_{k=1}^{n-1} I_k$ représente le nombre de couples (i,k) avec $i < k$ tels que $t_i < t_k$, c'est à dire le nombre d'inversions de la permutation associée à t. Par conséquent, le nombre de comparaisons du tri par insertion pour un tableau t vaut

$$C(t) = (n-1) + \mathrm{INV}(\sigma)$$

Pour insérer l'élément t_k à sa place parmi $t_0, \ldots, t_{k-1}$, il faut à chaque fois un premier transfert ($temp \leftarrow t_k$). Ensuite, à chaque comparaison effectuée, on fait un transfert, et à la fin un transfert supplémentaire $t_{i+1} \leftarrow t_i$. Par conséquent, le nombre de transferts vérifie :

$$T(t) = \sum_{k=1}^{n-1} (2 + I_k) = 2(n-1) + INV(t) = (n-1) + C(t)$$

Le nombre de transferts diffère du nombre de comparaisons d'un facteur linéaire. □

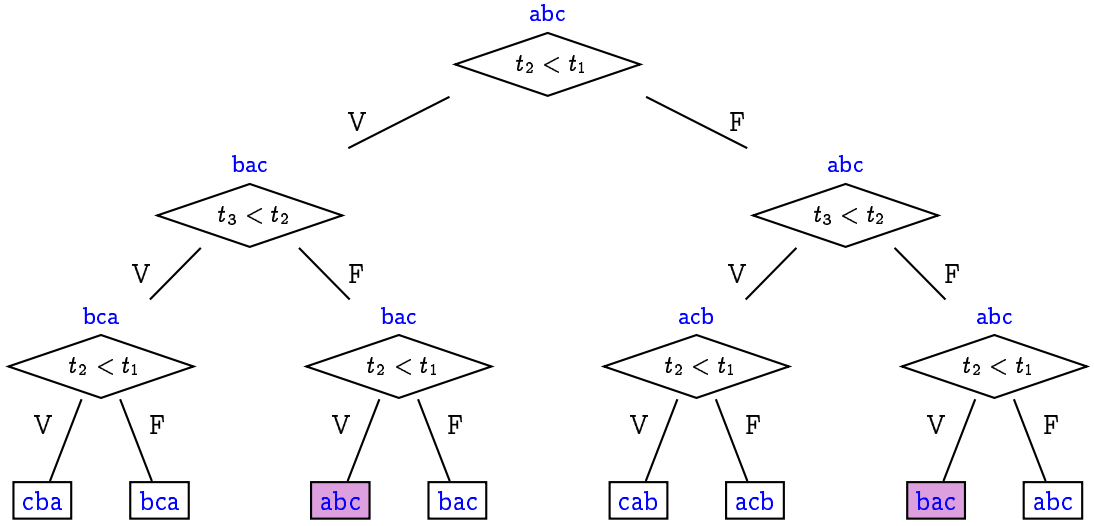


FIG. 3.3 – *Un arbre de décision pour le tri bulle de* 3 *éléments*

3.3.4 Comparaison des tris

Le tableau suivant donne le nombre de comparaisons et de transferts dans le meilleur des cas, en moyenne et dans le cas le pire pour les trois tris. Il dit également si les tris sont stables.

	Comparaisons			Transferts			Stabilité
Tri	min	moy	max	min	moy	max	
Sélection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	non
Bulle	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	0	$\Theta(n^2)$	$\Theta(n^2)$	oui
Insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	oui

Ces trois tris élémentaires permettent de trier des tableaux de quelques centaines d'éléments, mais leur efficacité devient mauvaise dès que n dépasse le millier. Par exemple, si l'on suppose qu'une comparaisons entre clés nécessite 10^{-6} secondes, pour un tableau de taille $n = 10^6$, on a $n^2 = 10^{12}$ et donc il faut de l'ordre de 10^6 secondes (environ 280 heurs) uniquement pour les comparaisons du tri bulle ! Le tri par sélection est le plus performant lorsqu'il s'agit de trier des enregistrements importants, car le nombre de transferts est linéaire. Dans le cas de tableaux presque triés, le tri par insertion est efficace, car le nombre de comparaisons et de transferts pour ce tri est étroitement lié au nombre d'inversions du tableau. Le tri à bulle donne de moins bons résultats expérimentalement que les deux autres tris et est donc à proscrire.
On peut se demander s'il existe des méthodes de tris plus efficaces que les trois méthodes élémentaires que nous venons d'examiner. En particulier, on peut se demander s'il existe un algorithme de tri de complexité linéaire. La réponse est négative comme le montre le théorème suivant :

THÉORÈME 3.7: **Borne inférieure d'un algorithme de tri**
Tout algorithme de tri qui n'utilise que les comparaisons entre éléments d'un tableau de taille n effectue dans le cas le pire au moins $\lceil \log_2(n!) \rceil$ comparaisons.
Comme $log_2(n!) \sim n \log_2 n$, le nombre de comparaisons d'un tel algorithme de tri dans le cas le pire est en

$$\Omega(n.\log n)$$

Preuve. La démonstration utilise les *arbres de décision*. Etant donné un tableau t quelconque de taille n, l'algorithme commence par comparer deux éléments. Selon le résultat du test, il exécutera une série d'instructions ou une autre. Graphiquement, le test est représenté par un noeud d'un arbre. Si le test s'avère positif, l'exécution de l'algorithme se poursuit dans la branche gauche de l'arbre, sinon dans la branche droite. La figure 3.3.4 représente l'arbre de décision associé au tri bulle de trois éléments. Après un nombre fini de comparaisons, l'algorithme s'arrête, et on représente dans une feuille de l'arbre le tableau modifié. Puisque l'algorithme doit trier un tableau quelconque, et qu'il y a $n!$ permutations possibles des éléments d'un tableau, le nombre de

feuilles de l'arbre de décision doit être supérieur à $n!$. Mais si l'algorithme effectue h comparaisons, il y a au maximum 2^h feuilles dans l'arbre. Par conséquent, on doit avoir l'inégalité

$$2^h \geq n!$$

d'où l'on tire $h \geq \lceil \log_2 n! \rceil$. Pour estimer asymptotiquement la quantité $\log_2 n!$, on peut écrire

$$\log_2 n! = \sum_{k=2}^{n} \log_2 k$$

et utiliser la croissance de la fonction $\log_2$ pour comparer $\log_2 k$ avec une intégrale :

$$\int_{k-1}^{k} \log_2 t \, \mathrm{d}t \leq \log_2 k \leq \int_{k}^{k+1} \log_2 t \, \mathrm{d}t$$

d'où en sommant (ajouter $0 = \log_2(1)$ pour la majoration en ne pas majorer $\log_2 n$) :

$$\int_{1}^{n} \log_2 t \, \mathrm{d}t \leq \log_2 n! \leq \int_{1}^{n} \log_2 t \, \mathrm{d}t + \log_2 n$$

L'intégrale se calcule par parties :

$$\int_{1}^{n} \log_2(t) \, \mathrm{d}t = \frac{1}{\ln 2}\left(n \ln n - n + 1\right)$$

On obtient alors l'encadrement :

$$n \log_2 n - \frac{n}{\ln 2} + \frac{1}{\ln 2} \leq \log_2(n!) \leq n \log_2 n - \frac{n}{\ln 2} + \frac{1}{\ln 2} + \log_2 n$$

qui montre que $\log_2 n! \sim n \log_2 n$. □

Remarque 15. Aucune des trois méthodes de tri élémentaires ne fournit cette borne en $n.\log n$. Nous verrons au prochain chapitre un algorithme de tri de complexité $\Theta(n.\log n)$. Il sera en quelque sorte « optimal » au vu du théorème précédent.

Remarque 16. On montre également que la complexité *moyenne* d'un algorithme de tri se basant uniquement sur la comparaison des clés est toujours en $\Omega(n \log n)$.

Remarque 17. Il existe d'autres algorithmes plus performants qui utilisent d'autres informations sur les clés que les simples comparaisons. Par exemple, si les clés sont des entiers compris entre 0 et 10^p, on peut utiliser les chiffres de ces entiers pour imaginer une méthode plus efficace.

Exercice 3-6

On dispose d'un tableau t rempli de n éléments distincts dont les clés sont les entiers $0, \ldots, n-1$.

a) Montrer que la fonction suivante trie sur place le tableau :

CAML

```
let trie t =
  let echange i j =
    let temp = t.(i) in
      t.(i) <- t.(j);
      t.(j) <- temp
  in
  let n = vect_length t in
    for i = 0 to n - 1 do
      while  t.(i) <> i do
        echange i t.(i);
      done;
    done;;
```

b) Déterminer le nombre maximal d'échanges et de comparaisons qu'effectue cette fonction.

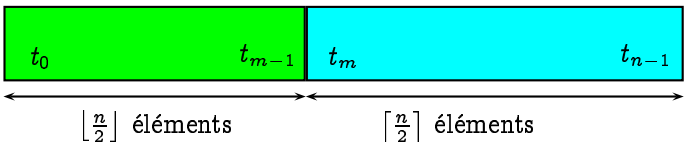
Chapitre 4

Stratégie diviser pour régner

4.1 La stratégie diviser pour régner

Pour traiter un problème de taille n :

1. Partager les données en deux parties de taille approximativement égales ;
2. Traiter séparément une ou les deux parties ;
3. Fusionner les résultats des deux parties.



4.2 Un cas modèle

DÉFINITION 4.1: Si $x \in \mathbb{R}$, on note :

- $\lfloor x \rfloor$ la partie entière de x : le plus grand entier inférieur ou égal à x.
- $\lceil x \rceil$ la partie entière supérieure de x : le plus petit entier supérieur ou égal à x

Ces entiers sont caractérisés par les inégalités :

$$\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$$

$$\lceil x \rceil - 1 < x \leq \lceil x \rceil$$

Remarque 18. Si $n \in \mathbb{N}$, alors :

- n pair :

$$\left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil = \frac{n}{2} = n/2$$

- n impair :

$$\left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2} = n/2$$

$$\left\lceil \frac{n}{2} \right\rceil = \frac{n+1}{2} = n/2 + 1$$

Dans les deux cas, on a toujours :

$$\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n$$

Exercice 4-1

Supposons que dans un algorithme « diviser pour régner » le coût de la division et de la fusion soit linéaire de la forme Cn avec $C > 0$ et qu'il faille traiter récursivement les deux sous-problèmes de taille $\left\lfloor \frac{n}{2} \right\rfloor$ et $\left\lceil \frac{n}{2} \right\rceil$. Le coût d'une donnée de taille 1 vaut 0.
a) Ecrire la relation de récurrence vérifiée par le coût $T(n)$ de l'algorithme.
b) On suppose que $n = 2^p$ est une puissance de 2. En notant $u_p = T(2^p)$, déterminer la relation de récurrence vérifiée par la suite (u_p).

c) En déduire l'expression de $T(n)$ en fonction de n lorsque n est une puissance de 2.
d) Montrer que $T(n)$ est une suite croissante.
e) Montrer que $T(n) = \Theta(n \log n)$.

Remarque 19. On voit donc qu'il est intéressant de trouver des algorithmes linéaires pour diviser et fusionner des données, car alors il est possible de trouver un algorithme quasi-linéaire pour traiter le problème qui nous intéresse.

4.3 Résolution générale des récurrences « diviser pour régner »

Pour traiter un problème de taille n en utilisant l'idée « diviser pour régner », on utilise la démarche suivante :

1. *Diviser* le problème en deux sous-problèmes de taille $\left\lfloor \frac{n}{2} \right\rfloor$ et $\left\lceil \frac{n}{2} \right\rceil$. Coût du partage : $f_1(n)$.
2. *Résoudre récursivement* les deux sous-problèmes.
3. *Fusionner* les résultats. Coût de la fusion : $f_2(n)$.

Si l'on note $T(n)$ le coût pour traiter le problème de taille n, on a donc la relation de récurrence suivante :

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n)$$

où $f(n) = f_1(n) + f_2(n)$.
Nous admettrons le résultat suivant :

THÉORÈME 4.1: **Complexité d'un algorithme « diviser pour régner »**

- Si $f(n) = O(n^\beta)$ avec $\beta < 1$, alors $T(n) = \Theta(n)$;
- Si $f(n) = \Theta(n)$, alors $T(n) = \Theta(n \log n)$;
- Si $f(n) = \Theta(n^\beta)$ avec $\beta > 1$, alors $T(n) = \Theta(n^\beta)$.

Retenons que :

1. Si le coût total de la division et de la fusion d'un problème de taille n est négligeable devant n, alors l'algorithme « diviser pour régner » est de complexité linéaire.
2. Si le coût total de la division et de la fusion d'un problème de taille n est linéaire, l'algorithme « diviser pour régner » est de complexité $n \log n$ c'est à dire quasi-linéaire.
3. Si le coût total de la division et de la fusion d'un problème de taille n est plus que linéaire, la complexité de l'algorithme « diviser pour régner » est du même ordre que le coût de la fusion-division.

Le deuxième cas est le plus souvent rencontré en pratique pour les algorithmes intéressants comme les tris.
Certains algorithmes « diviser pour régner » ne traitent pas également les deux sous-problèmes (la recherche dichotomique par exemple). On a le théorème plus général suivant :

THÉORÈME 4.2: **Complexité générale d'un algorithme « diviser pour régner »**
Soit $(a,b) \in \mathbb{N}^2$ avec $a + b \geq 1$. Posons

$$\alpha = \log_2(a + b)$$

On considère une suite $T(n)$ vérifiant la relation de récurrence :

$$T(n) = aT\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + bT\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n)$$

- Si $f(n) = O(n^\beta)$ avec $\beta < \alpha$, alors $T(n) = \Theta(n^\alpha)$;
- Si $f(n) = \Theta(n^\alpha)$ alors $T(n) = \Theta(n^\alpha \log n)$;
- Si $f(n) = \Theta(n^\beta)$ avec $\beta > \alpha$ alors $T(n) = \Theta(n^\beta)$.

Exercice 4-2
Déterminer, dans le pire des cas, le nombre asymptotique de comparaisons dans la recherche dichotomique.

4.4 L'algorithme d'exponentiation rapide

Il s'agit de calculer x^n lorsque $x \in \mathbb{R}$ et $n \in \mathbb{N}$ (n grand). Le premier programme qui vient à l'esprit est le suivant :

CAML
```
let exponentiation x n =
  let p = ref(1.) in
    for i = 1 to n do
      p := !p *. x
    done;
    !p;;
```

Il nécessite n multiplications.
Le deuxième algorithme se base sur la remarque suivante :

$$x^n = \begin{cases} \left(x^{n/2}\right) * \left(x^{n/2}\right) & \text{si n est pair} \\ x * \left(x^{n/2}\right) * \left(x^{n/2}\right) & \text{si n est impair} \end{cases}$$

CAML
```
let rec expo_rapide x n =
  match n with
    | 0 -> 1.;
    | 1 -> x;
    | _ -> let y = expo_rapide x (n / 2) in
        if (n mod 2 = 0) then y *. y
        else x *. y *. y;;
```

Théorème 4.3 : Complexité de l'exponentiation rapide

1. Lorsque $n = 2^p$, l'algorithme d'exponentiation rapide nécessite $\log_2(n)$ multiplications.
2. Lorsque n est quelconque, l'algorithme d'exponentiation rapide nécessite $\Theta(\log_2 n)$ multiplications dans le pire des cas.

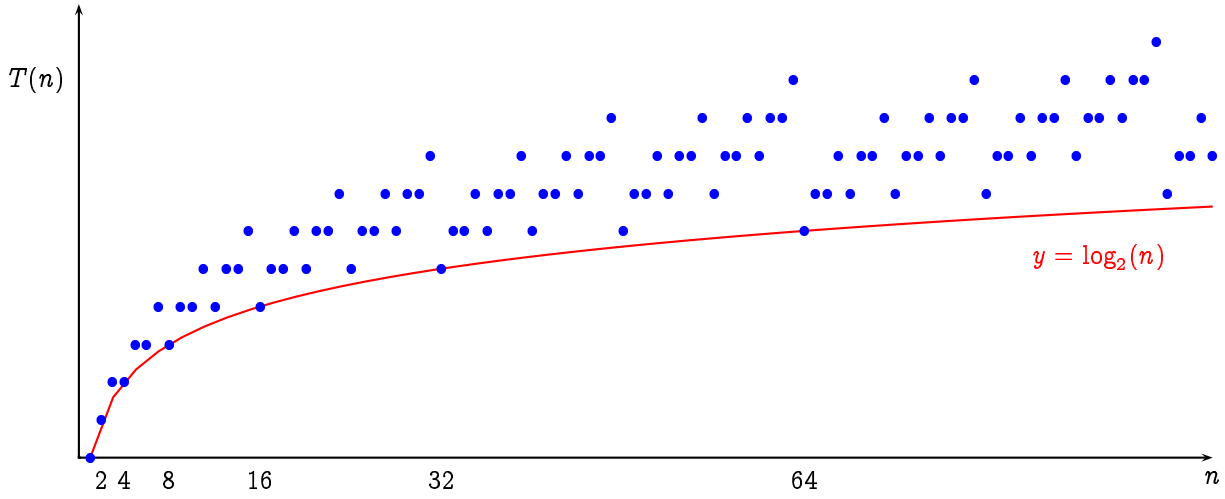


Fig. 4.1 – *Nombre de multiplications pour l'exponentiation rapide*

4.5 Le tri fusion (mergesort)

On veut trier par ordre croissant un tableau t de n éléments (pour nous des entiers). La remarque fondamentale est que l'on peut fusionner deux tableaux triés en un seul tableau trié en un temps linéaire nécessitant $|t_1| + |t_2|$ comparaisons

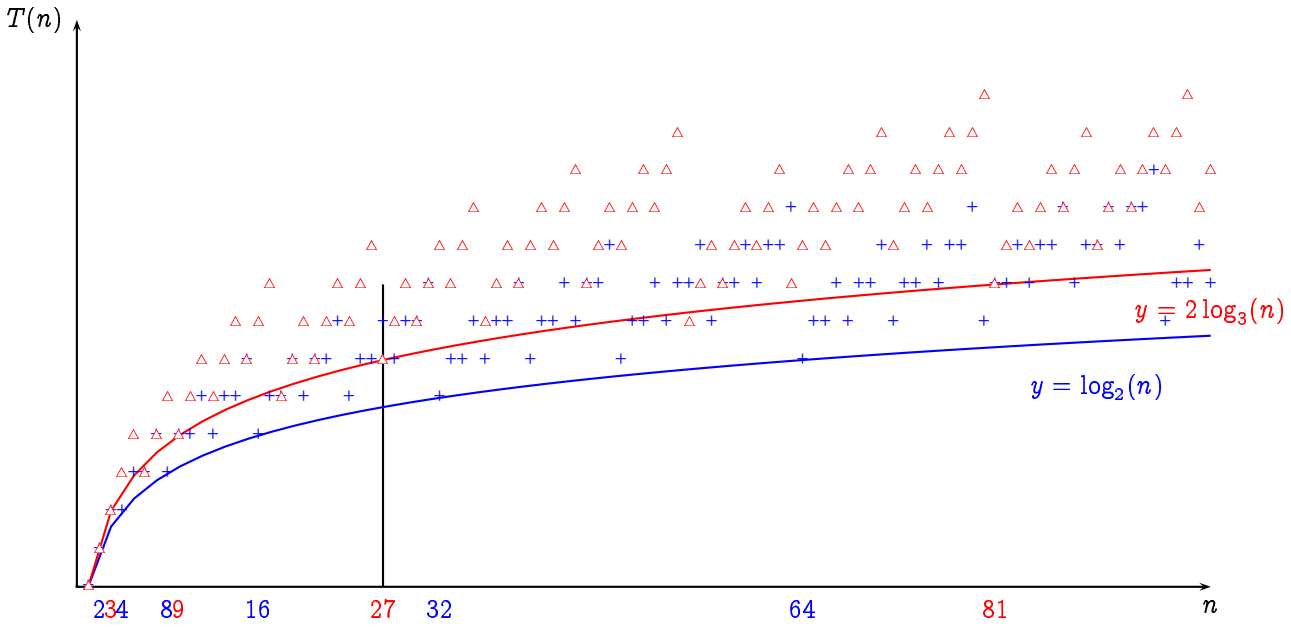


FIG. 4.2 – $T(n)$ *et* $S(n)$

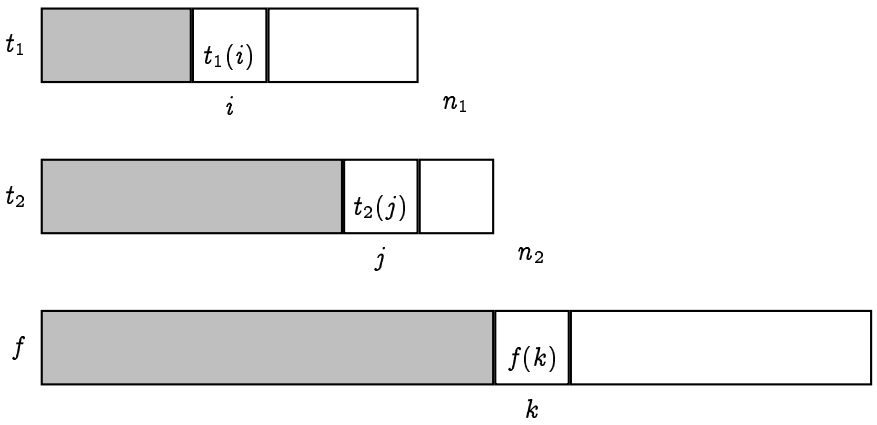


FIG. 4.3 – *Fusion linéaire de deux tableaux*

CAML

```
let fusion t1 t2 =
  let n1 = vect_length t1 in
  let n2 = vect_length t2 in
  let f = make_vect (n1 + n2) 0  in
  let i = ref 0 in
  let j = ref 0 in
    (* remplit le tableau f *)
    for k = 0 to (n1 + n2 - 1) do
      if (!i < n1) then (* s'il reste des éléments dans t1 *)
        begin
          if (!j < n2) then (* et dans t2 *)
            begin
              if ( t1.(!i) <= t2.(!j)) then
                     begin
                       f.(k) <- t1.(!i);
                       i := !i + 1;
                     end
                 else
                   begin
                     f.(k) <- t2.(!j);
                     j := !j + 1;
                   end
              end
            else (* s'il n'y a plus d'éléments dans t2 *)
              begin
                f.(k) <- t1.(!i);
                i := !i + 1;
              end
        end
      else (* s'il n'y a plus d'éléments dans t1 *)
        begin
          f.(k) <- t2.(!j);
          j := !j + 1;
        end
    done;
    f;;
```

Le tri fusion consiste à

1. *diviser* le tableau t en deux tableaux t_1, t_2 de tailles $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$
2. *trier récursivement* les deux tableaux t_1 et t_2.
3. *fusionner* les deux tableaux triés.

CAML

```
let rec tri_fusion t =
  let n = vect_length t in
    if (n < 2) then t else
      begin
        let m = n / 2 in
          let t1 = tri_fusion (sub_vect t 0 m) and
        let t2 = tri_fusion (sub_vect t m (n - m)) in
          fusion t1 t2
      end;;
```

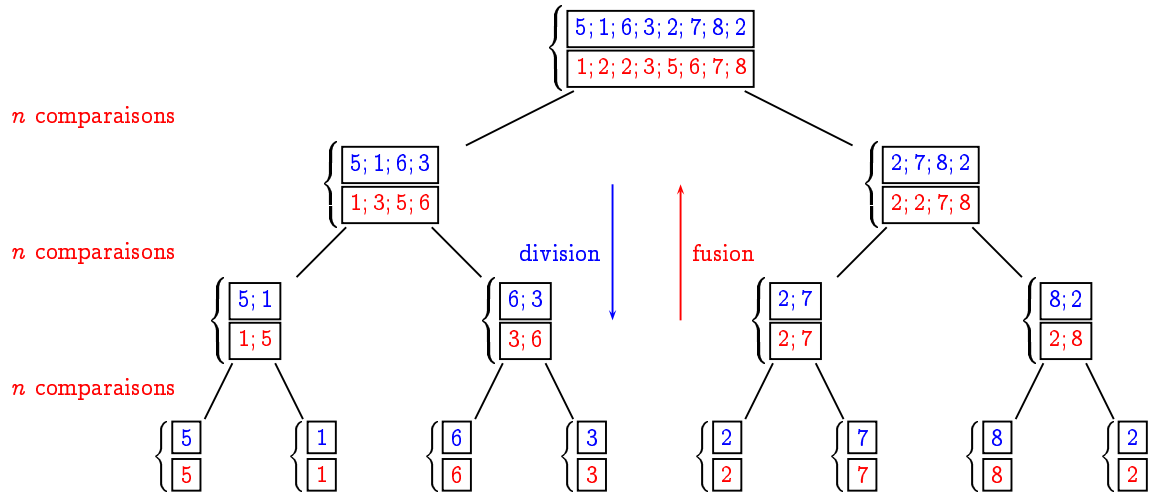


FIG. 4.4 – *Tri fusion. Hauteur de l'arbre :* $\log_2(n)$, *nombre de comparaisons :* $n \log_2(n)$

THÉORÈME 4.4: Complexité du tri fusion

1. Si le tableau t est de taille $n = 2^p$, le tri fusion nécessite $n \log_2 n$ comparaisons.
2. Si le tableau t est de taille n quelconque, le tri fusion nécessite $\Theta(n \log_2 n)$ comparaisons.

Le programme que nous avons écrit n'est pas efficace, car il nécessite trop de place en mémoire : A chaque appel récursif de `tri_fusion`, on crée trois tableaux auxiliaires. Il est possible de modifier la fonction en travaillant uniquement sur les indices du tableau initial :

CAML

```
let tri_fusion t =
  let n = vect_length t in
    (* fusion du sous tableau
       [|t.(debut) .. t.(milieu)|]
       avec [|t.(milieu+1) ... t.(fin) |] *)
  let fusion debut milieu fin =
    let f = make_vect (fin - debut + 1) 0 in
    let i = ref debut in
    let j = ref (milieu + 1) in
      for k = 0 to (fin - debut) do
        if (!i <= milieu) then
          begin
            (* s'il y a des éléments dans
               les deux sous-tableaux*)
            if (!j <= fin) then
              begin
                if t.(!i) <= t.(!j) then
                  begin
                    f.(k) <- t.(!i);
                    i := !i + 1;
                  end
                else
                  begin
                    f.(k) <- t.(!j);
                    j := !j + 1;
                  end
              end
            else
              (* il n'y a plus d'éléments dans le
```

```
                sous-tableau droit *)
              begin
                      f.(k) <- t.(!i);
                      i := !i + 1;
              end
          end
        else
          (* il n'y a plus d'éléments dans le
             sous-tableau gauche *)
          begin
            f.(k) <- t.(!j);
            j := !j + 1;
          end
      done;
      (* recopie de f dans t *)
      for k = debut to fin  do
        t.(k) <- f.(k-debut);
      done;
in
  (* tri du sous-tableau [|t.(i) ... t.(j) |] *)
let rec tri_rec i j =
  if (j - i) > 0 then
    let m = (i + j) /2 in
      tri_rec i m;
      tri_rec (m + 1)  j;
      fusion i m j
in
  tri_rec 0 (n-1);;
```

Remarque 20. L'inconvénient majeur du tri fusion est qu'il nécessite un tableau auxiliaire lors de l'étape de fusion et donc qu'il consomme de l'espace mémoire inutile.

4.6 Le tri rapide (quicksort)

On veut trier par ordre croissant un tableau t. L'idée de cet algorithme est de prendre un élément pivot a quelconque du tableau (par exemple le premier), et de réorganiser le tableau en

`[| t.(0); ... ;t.(m); ... ; t.(n-1) |]`

de telle sorte que $t.(m) = a$, $t.(0), \ldots, t.(m-1) \leq a$ et $t.(m+1), \ldots, t.(n-1) > a$.

Il suffit ensuite de réutiliser récursivement cet algorithme sur le sous-tableau

`[|t.(0); ... ; t.(m-1) |]`

et sur le sous-tableau

`[|t.(m+1); ... ; t.(n-1) |]`

pour obtenir à la fin un tableau trié.

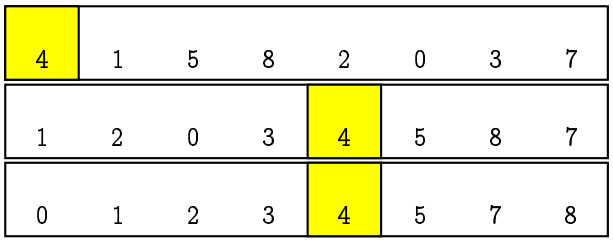


FIG. 4.5 – *Idée du tri rapide*

Il n'y a pas d'étape de fusion dans le tri rapide. Par contre, l'étape de partition est cruciale.

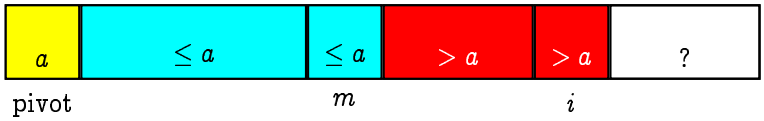


FIG. 4.6 – *Invariant de boucle du tri rapide*

Pour effectuer la partition d'un tableau $t = [|t_0; \ldots; t_{n-1}|]$, nous allons effectuer une boucle for indexée par $i \in [0,n-1]$ et maintenir l'invariant de boucle suivant (voir la figure 4.6) :

- $1 \leq m \leq i < n$;
- $\forall k \in [1,m]$, $t_k \leq a$;
- $\forall k \in [m+1,i]$, $t_k > a$.

A la fin de la boucle for, on aura trouvé un indice m et partagé le tableau. Il suffira d'échanger $t.(0)$ avec $t.(m)$ (voir la figure 4.7).

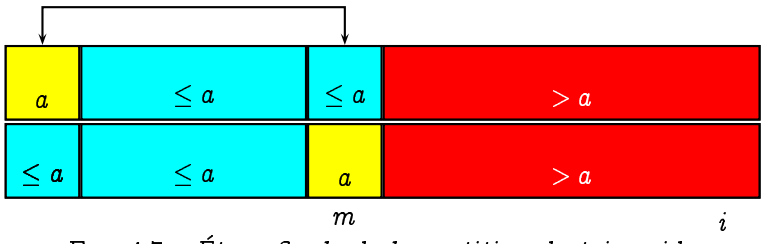


FIG. 4.7 – *Étape finale de la partition du tri rapide*

CAML
```
let partition t  =
  let echange i j =
    let temp = t.(i) in
      t.(i) <- t.(j);
      t.(j) <- temp
  in
  let m = ref 0 in
    for i = 1  to vect_length t - 1 do
      if t.(i) <= t.(0) then
        begin
          m := !m + 1;
          echange i !m ;
        end
        (* INV : 1 <= m <= i < n
           && 1 <= k <= m => t.(k) <= t.(0)
           && m < k <= i => t.(k) > t.(0) *)
    done;
    (* 0< k <= m => t.(k) <= t.(0)
       && m < k <n => t.(k) > t.(0) *)
    echange 0 !m;
    !m;;
```

Il suffit de modifier cette fonction pour qu'elle partitionne un sous-tableau $[|t_d; \ldots; t_f|]$ et de la rendre locale. Le tri rapide s'écrit alors :

CAML
```
let tri_rapide t =
  let n = vect_length t in
  let echange i j =
    let temp = t.(i) in
      t.(i) <- t.(j);
      t.(j) <- temp
  in
```

```
let partition d f =
  let m = ref d in
    for i = d + 1 to f do
      if t.(i) <= t.(d) then
        begin
          echange i (!m + 1);
          m := !m + 1;
        end
    done;
    echange d !m;
    !m
in
  (* trie le sous-tableau [|t.(d) .. t.(f) |] *)
let rec tri_rapide_rec d f =
  if (f - d > 0) then
    begin
          let m = partition d f in
        tri_rapide_rec d (m - 1);
        tri_rapide_rec (m + 1) f;
    end
in
  tri_rapide_rec 0 (n - 1);;
```

THÉORÈME 4.5: **Complexité du tri rapide**
Pour trier un tableau de taille n, le tri rapide nécessite

1. Dans le cas le pire, $O(n^2)$ comparaisons ;
2. Dans le cas moyen, $O(n \log_2 n)$ comparaisons.

Preuve. Lorsque le tableau t est déjà trié, après l'étape de partition, le pivot reste en place. A l'étape suivante, il faudra trier deux tableaux de longueur 0 et $(n-1)$ encore triés. En notant $C(n)$ le nombre de comparaisons dans ce cas, on a la relation de récurrence

$$C(n) = n - 1 + C(n-1)$$

ce qui montre que $C(n) = \Theta(n^2)$. On peut montrer que c'est le pire des cas.
Pour déterminer le nombre moyen de comparaisons du tri rapide, faisons l'hypothèse que le pivot a la même probabilité $1/n$ de se trouver placé après l'étape de partition à la case p, $(0 \leq p \leq n-1)$. Il faudra alors trier deux tableaux $[|t_0; \ldots; t_{p-1}|]$ et $[|t_{p+1}; \ldots; t_{n-1}|]$ de longueurs p et $n-1-p$. Donc le nombre $C_{moy}(n)$ de comparaisons vérifie la relation de récurrence :

$$C_{moy}(n) = (n-1) + \frac{1}{n}\sum_{p=0}^{n-1}\big(C_{moy}(p) + C_{moy}(n-1-p)\big) = (n-1) + \frac{2}{n}\sum_{k=0}^{n-1} C_{moy}(k)$$

En multipliant cette relation par n, et en l'écrivant également à l'ordre $(n-1)$, on trouve

$$nC_{moy}(n) = n(n-1) + 2\sum_{k=0}^{n-1} C_{moy}(k)$$

$$(n-1)C_{moy}(n) = (n-1)(n-2) + 2\sum_{k=0}^{n-2} C_{moy}(k)$$

En soustrayant ces deux relations pour éliminer la plupart des termes de la somme, on aboutit à

$$nC_{moy}(n) = 2(n-1) + (n+1)C_{moy}(n-1)$$

En divisant par $n(n+1)$, on trouve

$$\frac{C_{moy}(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{C_{moy}(n-1)}{n}$$

Posons alors $D_n = \dfrac{C_{moy}(n)}{n+1}$. On obtient la relation de récurrence suivante sur D_n :

$$D_n = \frac{2(n-1)}{n(n+1)} + D_{n-1} = -\frac{2}{n} + \frac{4}{n+1} + D_{n-1}$$

d'où l'on tire puisque $D_1 = C_{moy}(1) = 0$, que

$$D_n = -2\sum_{k=2}^{n} \frac{1}{k} + 4\sum_{k=2}^{n} \frac{1}{k+1}$$

et en introduisant les *nombres harmoniques*

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

$$D_n = 2H_n - 4 + \frac{4}{n+1}$$

d'où finalement,

$$C_{moy}(n) = 2(n+1)H_n - 4 + \frac{4}{n+1}$$

Mais en comparant la somme H_n avec les intégrales $\int_a^b \dfrac{dt}{t}$, on montre que $H_n \sim \ln n$. Par conséquent,

$$C_{moy}(n) \sim 2n \ln n$$

□

Remarque 21. Le tri rapide est souvent utilisé en pratique, car il n'utilise pas de mémoire supplémentaire contrairement au tri fusion. En moyenne, sa complexité temporelle est asymptotiquement optimale.
Remarque 22. Lorsque le tableau t est déjà trié, le tri rapide n'est pas efficace, car la partition selon le premier élément donne deux tableaux déséquilibrés. Une variante qui permet d'éviter ce problème consiste à choisir avant la partition trois éléments du tableau : le premier, le dernier et l'élément du milieu, et de partitionner le tableau selon l'élément médian de ces trois candidats. Une autre amélioration, toujours dans le but d'éviter une partition déséquilibrée, consiste à choisir le pivot aléatoirement.

4.7 Multiplication de polynômes

On représente un polynôme de degré p

$$P = a_0 + a_1 X + \cdots + a_p X^p$$

par un vecteur de longueur $(p+1)$ `[|a_0; ... ; a_p|]` Pour simplifier, on supposera les coefficients entiers. Le problème qui nous intéresse est de multiplier efficacement deux polynômes

$$P = a_0 + a_1 X + \cdots + a_n X^p$$

$$Q = b_0 + b_1 X + \cdots + b_q X^q$$

$$R = PQ = c_0 + c_1 X + \cdots + c_{p+q} X^{p+q}$$

La formule qui donne le coefficient c_k est :

$$\forall k \in [0,p+q], \quad c_k = \sum_{\substack{0 \le i \le p \\ 0 \le j \le q \\ i+j=k}} a_i b_j$$

d'où la première fonction qui vient à l'esprit :

CAML

```
let multiplie p q =
  let dp = vect_length p - 1 (* degré de p*)
```

```
  and dq = vect_length q - 1 in (* degré de q*)
  let r = make_vect (dp + dq + 1) 0 in
    (* remplit les coefficients du produit*)
    for i = 0 to dp do
      for j = 0 to dq do
        r.(i + j) <- r.(i + j) + p.(i) * q.(j)
      done;
    done;
    r;;
```

Il est clair que la fonction précédente utilise une multiplication à chaque passage dans la double boucle, soit au total, $(dp+1)*(dq+1)$ multiplications. Lorsque P et Q sont de degré n, le nombre de multiplications scalaires est en $\Theta(n^2)$.
D. Knuth a proposé un algorithme utilisant la méthode « diviser pour régner ». On suppose que les deux polynômes sont de degré inférieur à $2n-1$. L'idée est d'écrire :

$$P = \left(a_0 + a_1 X + \cdots + a_{n-1}X^{n-1}\right) + X^n\left(a_n + a_{n+1}X + \cdots + a_{2n-1}X^{n-1}\right) = P_0 + X^n P_1$$

$$Q = \left(b_0 + b_1 X + \cdots + b_{n-1}X^{n-1}\right) + X^n\left(b_n + b_{n+1}X + \cdots + b_{2n-1}X^{n-1}\right) = Q_0 + X^n Q_1$$

Alors le produit s'écrit :

$$\begin{aligned} R &= (P_0 + P_1X^n) * (Q_0 + Q_1X^n) \\ &= P_0 * Q_0 + (P_0 * Q_1 + P_1 * Q_0)\, X^n + P_1 * Q_1 X^{2n} \\ &= P_0 * Q_0 + ((P_0 + P_1) * (Q_0 + Q_1) - P_0 * Q_0 - P_1 * Q_1)\, X^n + P_1 * Q_1 X^{2n} \\ &= \alpha_0 + (\alpha_1 - \alpha_0 - \alpha_2)\, X^n + \alpha_2 X^{2n} \end{aligned}$$

Sous la dernière forme, on voit qu'il n'est nécessaire que de calculer *trois* multiplications (à la place de 4 à priori) de polynômes P_0,P_1,Q_0,Q_1 de degré $n-1$:

$$\begin{aligned} \alpha_0 &= P_0 * Q_0 \\ \alpha_1 &= (P_0 + P_1) * (Q_0 + Q_1) \\ \alpha_2 &= P_1 * Q_1 \end{aligned}$$

Comme l'addition de deux polynômes nécessite un temps bien moins long que la multiplication, on gagnera en complexité temporelle.

CAML

```
(* Multiplication de knuth de deux polynômes.
   On suppose que |p|  = |q| = m =  2^k,
   c'est à dire deg(p) = deg(q) = 2^k - 1.
   La fonction retourne le vecteur r de longueur
   2*m - 1 avec  les coefficients du produit pq *)

let rec knuth p q =
  let m = vect_length p  in
  let r = make_vect (2 * m - 1) 0 in
    (* cas terminal, p=a_0, q = b_0 *)
    if m = 1 then
      r.(0) <-  p.(0) * q.(0)
    else
      begin
        (* découpage *)
        let n = m / 2 in
        let p0 = sub_vect p 0 n
```

```
        and q0 = sub_vect q 0 n
        and p1 = sub_vect p n n
        and q1 = sub_vect q n n in
          (* calcul de p01 = p0 + p1
             et de q01 = q0 + q1 *)
        let p01 = make_vect n 0
        and q01 = make_vect n 0 in
          for i = 0 to n - 1 do
            p01.(i) <- p0.(i) + p1.(i);
            q01.(i) <- q0.(i) + q1.(i)
          done;
          (* calcul récursif des trois multiplications *)
          let alpha0 = knuth p0 q0 (* p0q0 *)
          and alpha1 = knuth p01 q01 (* (p0 + p1)(q0 + q1) *)
          and alpha2 = knuth p1 q1 in (* p1 q1 *)
            (* remplit les coefficients de r*)
            for i = 0 to 2 * n - 2 do
              r.(i) <- r.(i) + alpha0.(i);
              r.(i + n) <- r.(i + n ) + alpha1.(i)
              - alpha0.(i) - alpha2.(i);
              r.(i + 2 * n) <- r.(i + 2 * n) + alpha2.(i)
            done;
      end;
      r;;
```

THÉORÈME 4.6: **Nombre de multiplications de l'algorithme de Knuth**
On suppose que les deux polynômes P et Q sont de degré $m = 2^k - 1$. (Les polynômes P et Q ont donc m coefficients). En notant $M(k)$ le nombre de multiplications scalaires, on a la formule de récurrence:

$$\begin{aligned} & M(0) = 1 \\ \forall k \geq 1 \quad & M(k) = 3M(k-1) \end{aligned}$$

et on obtient alors

$$M(k) = 3^k$$

d'où le nombre total de multiplications qui vaut

$$m^{\log_2 3} \approx m^{1.58}$$

Remarque 23. On utilise l'idée de cet algorithme pour effectuer la multiplication des grands entiers, dont le chiffres en base b sont stockés dans un tableau. Si un entier n s'écrit en base b : $n = a_p.b^p + \cdots + a_1.b + a_0$, on le représente par un vecteur $P_n = [|a_0; \ldots; a_p|]$. La multiplication de deux entiers longs n et p revient à calculer le produit de deux polynômes P_n et P_p. Il suffit de gérer la contrainte sur les chiffres qui doivent être compris entre 0 et b en propageant des retenues.

Remarque 24. Il existe un algorithme en $\mathcal{O}(n \log n)$ qui permet de multiplier deux polynômes à coefficients complexes : la transformée de Fourier rapide (FFT).

Exercice 4-3

Si l'on n'utilise pas l'astuce qui réduit le nombre de multiplications de 4 à 3 dans l'algorithme de Knuth, déterminer le nombre $T(m)$ de multiplications nécessaires dans l'algorithme « diviser pour régner » correspondant. (On supposera que $m = 2^k$).

Exercice 4-4

Trouver un équivalent du nombre $A(m)$ d'additions scalaires nécessaires pour effectuer le produit de deux polynômes à m coefficients (donc de degré $m - 1$)

- Pour la multiplication naïve `mult` ;
- Pour la multiplication de Knuth. (Supposer que $m = 2^k$).

4.8 Multiplication de matrices : algorithme de Strassen

Il s'agit de multiplier deux matrices carrées $n \times n$, avec n grand.

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix}$$

La matrice produit $C = AB = ((c_{ij}))_{1 \leq i,j \leq n}$ a pour coefficient générique :

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

Il est facile d'écrire une fonction qui effectue la multiplication matricielle en n^3 multiplications scalaires.
L'idée de Strassen est d'effectuer des produits par blocs. Si n est pair, on partage les matrices A et B en 4 matrices de taille $n/2$:

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \quad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

et alors

$$C = \begin{pmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{pmatrix}$$

L'idée de Strassen est qu'il suffit de 7 multiplications matricielles pour calculer C (à la place de 8 comme on pourrait le croire). A titre de curiosité, voici les formules de Strassen :

$$C = \begin{pmatrix} -P_2 + P_4 + P_5 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 - P_3 + P_5 - P_7 \end{pmatrix}$$

avec

$$\begin{aligned} P_1 &= A_1 * (B_2 - B_4) \\ P_2 &= (A_1 + A_2) * B_4 \\ P_3 &= (A_3 + A_4) * B_1 \\ P_4 &= A_4 * (B_3 - B_1) \\ P_5 &= (A_1 + A_4) * (B_1 + B_4) \\ P_6 &= (A_2 - A_4) * (B_3 + B_4) \\ P_7 &= (A_1 - A_3) * (B_1 + B_2) \end{aligned}$$

Exercice 4-5

Déterminer le nombre de multiplications scalaires nécessaires dans l'algorithme de Strassen.

Exercice 4-6

Si dans l'algorithme précédent, on utilise 8 multiplications au lieu de 7, déterminer le nombre de multiplications nécessaires.

Chapitre 5

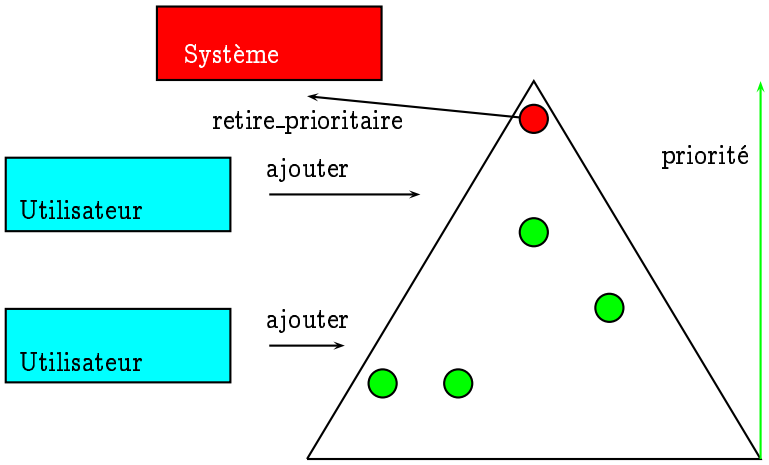
Piles, expressions algébriques

5.1 Types de données abstraits

La conception d'un programme compliqué se fait toujours en plusieurs étapes qui correspondent à des raffinements successifs. La première étape consiste à analyser le problème à traiter de façon abstraite, en définissant des opérations de haut niveau sur les données à traiter, sans se préocupper dans un premier temps de l'implémentation de ces opérations.

Prenons un problème classique. Un système d'exploitation doit traiter plusieurs tâches qui sont en attente. À chaque tâche est associée une *priorité*. Une fois que le système a traité une tâche, il doit rechercher la tâche la plus prioritaire pour la traiter. De plus, les utilisateurs peuvent ajouter à tout moment de nouvelles tâches. Une structure de donnée adaptée pour ce problème doit permettre de retirer en temps constant la tâche la plus prioritaire, et d'ajouter rapidement une tâche à la structure. Une telle structure s'appelle une *file de priorité*. Les opérations correspondantes peuvent être définies formellement par :

- `retire_prioritaire : file_priorité -> 'a` : retire l'élément de plus haute priorité ;
- `ajouter : 'a -> int -> file_priorité -> unit` : ajoute un élément de type `'a` de priorité $i \in \mathbb{N}$.



Le programmeur s'engage à n'utiliser que ces opérations, et n'a donc pas besoin de connaître le fonctionnement interne de la structure de données. Il est donc possible d'implanter cette structure de donnée dans une bibliothèque, et même de changer par la suite l'implémentation, sans que les programmes utilisant cette bibliothèque en souffrent. On peut dans un premier temps décider d'implémenter une file de priorité par une liste de couples `('a, int)` triée par ordre décroissant selon les priorités, dans laquelle on place les éléments ajoutés à leur place. L'élément de plus haute priorité se trouve ainsi toujours en début de liste. Le retrait de l'élément de plus grande priorité nécessite un temps constant, alors que l'ajout d'un nouvel élément se fait en temps $O(n)$. Il est possible ensuite de changer l'implémentation d'une file de priorité (en utilisant des arbres) de telle sorte que l'ajout d'un élément se fasse en temps $O(\log n)$.

5.2 Pile

Une structure de données classique, omniprésente en informatique est la *pile LIFO* (Last In, First Out), appelée *stack* en Anglais.
Les opérations traditionnellement associées à une pile sont :

- `new : unit -> 'a pile` : crée une nouvelle pile vide ;
- `push : 'a -> 'a pile -> unit` : ajoute un élément au sommet de la pile ;
- `pop : 'a pile -> 'a` : retire l'élément au sommet de la pile et le renvoie.
- `is_empty : 'a pile -> bool` : un prédicat qui teste si la pile est vide.

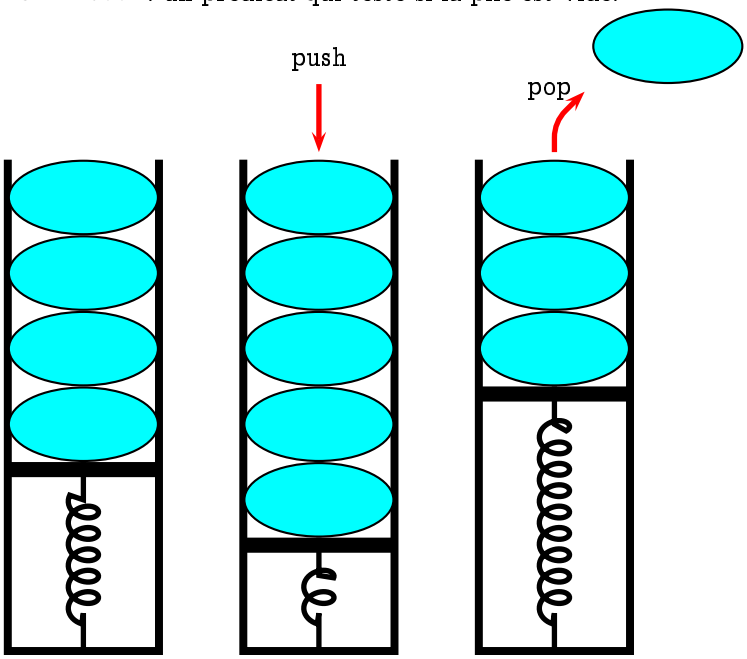


FIG. 5.1 – *Représentation d'une pile LIFO et des opérations push et pop*

Une façon commode de visualiser une pile est d'imaginer une pile d'assiettes dans un restaurant. Le cuisinier peut empiler des assiettes (après les avoir lavées), et peut à tout moment retirer une assiette propre de la pile pour servir un repas. Le nom « LIFO » s'explique car le dernier élément empilé sera le premier à être retiré de la pile. Il n'y a pas de limitation théorique au nombre d'éléments que peut contenir une pile. En pratique, la mémoire est limitée et si l'on essaie d'empiler plus d'éléments que le maximum autorisé, on a droit à un message de dépassement de pile (*stack overflow*).

`Exemple 6.`

- *On se sert d'une pile pour implémenter la récursivité dans un langage de programmation. L'environnement d'une fonction est stockée dans un pile lors d'un appel récursif. Une fois la valeur de retour connue, le programme reprend son exécution et dépile l'environnement de la fonction.*
- *Un problème d'analyse syntaxique consiste à vérifier un bon parenthésage d'un code source : est-ce que toutes les parenthèses, accolades... sont bien appariées ? On utilise une pile pour résoudre ce problème (voir TD).*

Exercice 5-1

Ecrire une fonction
`rotation` de type : `'a pile ->unit`
qui effectue une « rotation d'un cran » de la pile : l'élément qui était au fond de la pile se retrouve au sommet et les autres éléments sont décalés vers le bas.

Remarque 25. Les opérations sur les piles peuvent différer légèrement de celles présentées ici, (en particulier pour tester si une pile est vide) mais les deux opérations principales *push* et *pop* sont communes à toutes les implémentations de piles.

Remarque 26. Une pile est un objet mémoire « mutable », utilisé principalement en programmation impérative. Si une fonction récursive doit utiliser une pile, il faut créer auparavant la pile à l'extérieur de la fonction (si on la crée à l'intérieur, une nouvelle pile vide sera définie à chaque appel récursif). La fonction va ensuite modifier la pile par *effet de bord*.
On peut implémenter une pile de plusieurs façons. Nous utilisons ici un type enregistrement *mutable* (voir appendice A) :

```
type 'a pile = {mutable contenu : 'a list};;

let new () =
  {contenu = []};;

let push x p =
  p.contenu <- x :: p.contenu;;

let pop p =
  match p.contenu with
    | [] -> failwith "pile vide"
    | x :: q -> p.contenu <- q; x;;

let is_empty p =
  match p.contenu with
    | [] -> true
    | _ -> false;;
```

5.3 Files

Une autre structure abondamment utilisée en informatique est une *file d'attente*. C'est une structure « FIFO » (First In First Out). Elle fonctionne comme une queue à un guichet : on peut ajouter un élément à une file, et retirer le premier élément introduit.
Les méthodes associées sont :

- `new_file : unit -> 'a file` : crée une nouvelle file vide ;
- `ajout_file : 'a -> 'a file -> unit` : ajoute un élément à une file ;
- `retire_file : 'a file -> 'a` : retire le premier élément de la file et le retourne.
- `file_vide : 'a file -> bool` : un prédicat qui teste si une file est vide ;

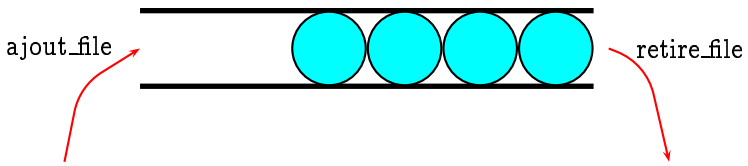


FIG. 5.2 – *Une file d'attente*

`Exemple 7.` *Un système d'exploitation utilise une file pour gérer les « tampons » (buffers) d'entrée-sortie : par exemple les caractères tapés au clavier ou l'écriture des données sur le disque.*

5.4 Expressions algébriques

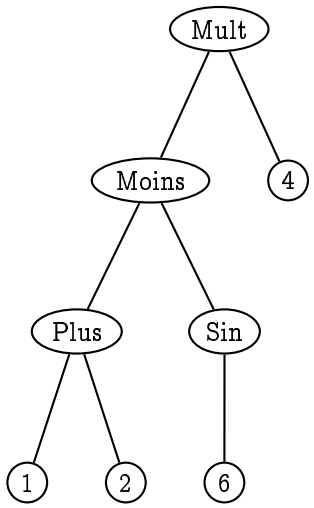
Considérons

1. des variables atomiques (de type float) ;
2. deux constructeurs d'arité 1 : `Sin`, `Exp` ;
3. quatre constructeurs d'arité 2 : `Plus`, `Moins`, `Mult`, `Div`.

Nous appellerons « expression algébrique » (simplifiée) l'ensemble des objets définis inductivement à partir de ces constructeurs et des variables atomiques.
L'expression mathématique $((1+2)-(sin(6)))*4$ correspond à l'expression algébrique définie par :

- 1,2 sont des atomes, donc $e1$ = `Plus(1, 2)` est une expression ;
- 6 est un atome donc $e2$ = `Sin(6)` est une expression ;
- $e1$,$e2$ sont des expressions, donc $e3$ = `Moins(e1, e2)` en est une aussi ;
- $e3$ est une expression, ainsi que 4, donc `Mult(e3, 4)` est une expression.

On peut représenter la construction précédente par un *arbre* (la dernière règle appliquée est placée au sommet de l'arbre). Lorsqu'un constructeur est d'arité k, il est représenté par un noeud de l'arbre qui possède k fils (sous-arbres).



En CAML, on définit le type d'une expression algébrique en utilisant les types somme :

CAML
```
type exp_alg =
    Variable of float          (* arité 0*)
  | Sin of exp_alg             (* Fonctions : arité 1*)
  | Exp of exp_alg
  | Plus of exp_alg * exp_alg  (* Opérateurs : arité 2*)
  | Moins of exp_alg * exp_alg
  | Divise of exp_alg * exp_alg
  | Mult of exp_alg * exp_alg;;
```

On obtient la *valeur* d'une expression algébrique de la façon suivante :

CAML
```
let rec evalue e =
  match e with
    | Variable x -> x
    | Sin e1 -> sin (evalue e1)
    | Exp e1 -> exp (evalue e1)
    | Plus (e1, e2) -> (evalue e1) +. (evalue e2)
    | Moins (e1, e2) -> (evalue e1) -. (evalue e2)
    | Divise (e1, e2) -> (evalue e1) /. (evalue e2)
    | Mult (e1, e2) ->  (evalue e1) *. (evalue e2);;
```

Il n'est pas agréable de définir une expression algébrique sous cette forme. On préfère une notation *linéaire*. Il en existe trois :

1. La notation *infixe* ;
2. La notation *préfixe* ;
3. La notation *postfixe* ;

que l'on obtient par un parcours en profondeur de l'arbre représentant une expression algébrique. Les fonctions d'affichage linéaire en CAML s'écrivent simplement :

CAML
```
#let rec prefixe e =
  match e with
    | Variable x -> print_float x; print_string " "
    | Plus (e1, e2) -> print_string "+ "; prefixe e1; prefixe e2
    | Moins (e1, e2) -> print_string "- "; prefixe e1; prefixe e2
    | Divise (e1, e2) -> print_string "/ "; prefixe e1; prefixe e2
    | Mult (e1, e2) ->  print_string "* "; prefixe e1; prefixe e2
    | Sin e1 -> print_string "sin "; prefixe e1
    | Exp e1 -> print_string "exp "; prefixe e1;;
prefixe : exp_alg -> unit = <fun>
```

CAML
```
#let rec postfixe e =
  match e with
    | Variable x -> print_float x; print_string " "
    | Plus (e1, e2) ->  postfixe e1; postfixe e2; print_string "+ "
    | Moins (e1, e2) -> postfixe e1; postfixe e2; print_string "- "
    | Divise (e1, e2) -> postfixe e1; postfixe e2; print_string "/ "
    | Mult (e1, e2) ->  postfixe e1; postfixe e2;  print_string "* "
    | Sin e1 -> postfixe e1; print_string "sin "
    | Exp e1 -> postfixe e1;  print_string "exp ";;
postfixe : exp_alg -> unit = <fun>
```

CAML
```
let rec infixe e =
match e with
    | Variable x -> print_float x; print_string " "
    | Plus (e1, e2) ->
        print_string "(";
        infixe e1;
        print_string "+ ";
        infixe e2;
        print_string ")"
    | Moins (e1, e2) ->
        print_string "(";
        infixe e1;
        print_string "- ";
        infixe e2;
        print_string ")"
    | Divise (e1, e2) ->
        print_string "(";
        infixe e1;
        print_string "/ ";
        infixe e2;
        print_string ")"
    | Mult (e1, e2) ->
        print_string "(";
        infixe e1;
        print_string "* ";
        infixe e2;
        print_string ")";
    | Sin e1 ->
        print_string "sin( ";
        infixe e1;
        print_string ")"
```

```
    | Exp e1 ->
        print_string "exp( ";
        infixe e1;
        print_string ")";;
```

CAML

```
#let e = Mult(
  Moins(
    Plus(Variable 1., Variable 2.),
    Sin (Variable 6.)),
  Variable 4.);;
e : exp_alg =
 Mult
  (Moins (Plus (Variable 1.0, Variable 2.0), Sin (Variable 6.0)),
   Variable 4.0)

#prefixe e;;
* - + 1.0 2.0 sin 6.0 4.0 - : unit = ()

#postfixe e;;
1.0 2.0 + 6.0 sin - 4.0 * - : unit = ()

#infixe e;;
(((1.0 + 2.0 )- sin( 6.0  ))* 4.0 )- : unit = ()
```

La notation linéaire infixe courante est ambigüe, car elle nécessite des parenthèses ou alors des hypothèses implicites de priorité des opérateurs. Par exemple, comment interpréter la notation linéaire suivante :

`2 + 3 * 4 :  (2 + 3) * 4  ou 2 + (3 * 4) ?`

On peut montrer que la notation postfixe est non-ambigüe: on peut reconstituer l'arbre de syntaxe correspondant à une notation linéaire postfixe. C'est la notation utilisée dans les calculatrices Hewlett-Packard, appelée souvent notation polonaise inversée.

5.4.1 Évaluation d'une expression arithmétique postfixée à l'aide d'une pile

Nous nous intéressons ici à deux problèmes distincts. Étant donnée une notation linéaire quelconque :

- Est-elle algébrique postfixée? (*syntaxe* correcte)
- Évaluer cette expression (*sémantique*).

Idéalement, l'utilisateur entre une expression passée comme une chaîne de caractères de la forme

`"2  4 + sin  3 *"`

et notre fonction retourne la valeur correspondante à cette expression. Il est assez difficile d'analyser directement une telle chaîne de caractères (il faut éliminer les espaces, convertir les parties de la chaîne correspondant à des nombres en flottants ...). Le rôle d'un *analyseur lexical* est de transformer une telle chaîne en une suite de *lexèmes* (unités lexicales), en séparant les nombres, des opérateurs et des fonctions. Nous n'aborderons pas cette question et nous supposerons que la notation linéaire est déjà fournie sous forme d'une liste de lexèmes :

CAML

```
type opérateurs = L_plus | L_moins | L_mult | L_div;;
type fonctions = L_sin | L_exp;;
type lexème = L_var of float | L_fonc of fonctions | L_op of  opérateurs;;
```

L'expression $e = 1\ 2\ +\ 6\ \sin\ -\ 4\ *$ sera alors représentée par la liste :

```
let l = [L_var 1.; L_var 2.; L_op L_plus; L_var 6.; L_fonc L_sin;
L_op L_moins; L_var 4.; L_op L_mult];;
```

On peut résoudre simultanément les deux problèmes à l'aide d'une pile, en lisant une EAP de gauche à droite. On part d'une pile vide.

- Si on lit un lexème représentant une variable, on empile la valeur de la variable ;

– Si on lit un lexème représentant une fonction, on dépile la valeur au sommet de pile, on applique la fonction correspondante et on empile le résultat ;
– Si on lit un lexème représentant un opérateur, on dépile deux valeurs de la pile, on applique l'opérateur à ces deux valeurs et on empile le résultat.

On montre que

Théorème 5.1: Évaluation d'une EAP à l'aide d'une pile

1. Si l'expression est une EAP syntaxiquement correcte, lors du parcours gauche-droite, la pile contient toujours au moins un élément et à la fin du parcours elle contient un seul élément.
2. La valeur correspondant à l'évaluation de l'EAP est l'unique élément qui se trouve dans la pile à la fin du parcours.

Voici sur un dessin l'évolution de la pile pour notre expression modèle $e = 1\ 2\ +\ 6\ \sin\ -\ 4\ *$. Au dessous de la pile est représenté le lexème lu :

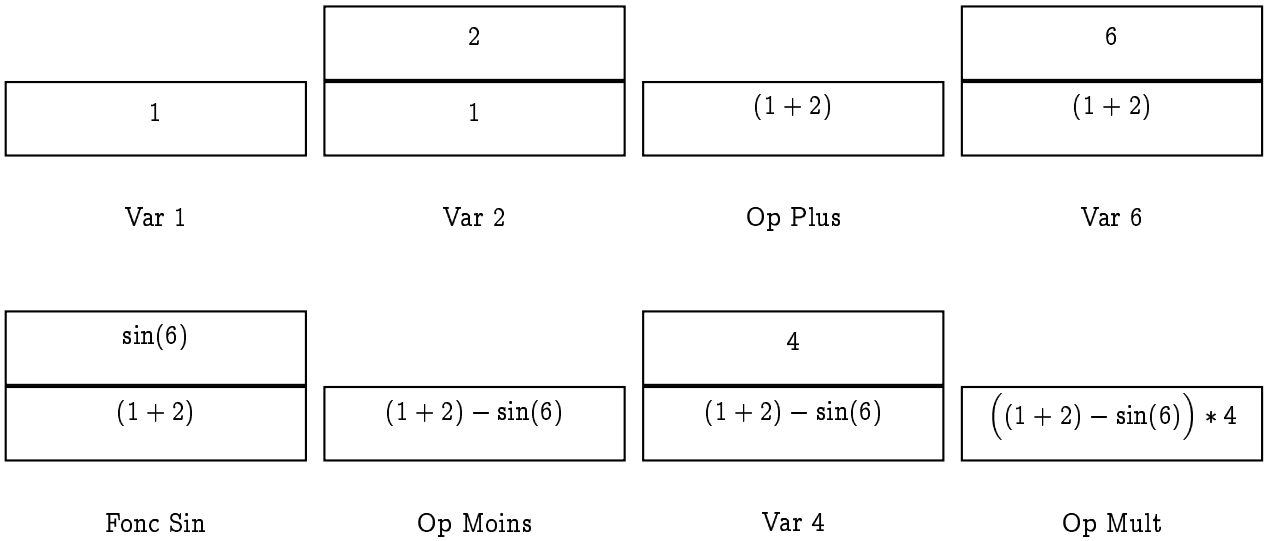


Fig. 5.3 – *Évaluation d'une EAP à l'aide d'une pile*

Voici une façon de programmer l'évaluation d'une EAP en Caml. On définit une pile initialement vide, et plusieurs fonctions qui vont manipuler cette pile. On commence par les opérations sur la pile correspondant à une variable, une fonction ou un opérateur. Ensuite une fonction `traite_lexème` qui effectue l' opération correspondant au type de lexème lu, enfin une fonction `parse` qui parcourt la liste de lexèmes et applique les opérations correspondant à chaque lexème rencontré.

CAML
```
let evalue expression =
  let p = new() in
  let traite_var v =
    push v p
  and traite_fonc f =
    match f with
      | L_sin -> let x = pop p in push (sin x) p
      | L_exp -> let x = pop p in push (exp x) p
  and traite_op o =
    let x = pop p in
    let y = pop p in
      match o with
        | L_plus -> push (x +. y) p
        | L_moins -> push (x -. y) p
        | L_mult -> push (x *. y) p
        | L_div -> push (x /. y) p
  in
```

```
let traite_lexème lex =
  match lex with
    | L_var v -> traite_var v
    | L_fonc f -> traite_fonc f
    | L_op o -> traite_op o
in
let rec parse l =
  match l with
    | [] -> pop p
    | lex :: q -> traite_lexème lex; parse q
in
  parse expression;;
```

Si l'expression est syntaxiquement incorrecte, il se produit une exception « pile vide » ou alors la pile à la fin contient plus d'un élément. On pourrait modifier la fonction en utilisant des exceptions pour vérifier si une expression est bien formée tout en l'évaluant.

Exercice 5-2

En adaptant la fonction précédente, écrivez une fonction qui renvoie l'arbre de syntaxe abstrait de type `exp_alg` d'une expression algébrique à partir de sa forme postfixe.

Chapitre 6

Logique

6.1 Propositions logiques

Une proposition est phrase non-ambigüe à laquelle on peut attribuer une valeur de vérité `vrai` ou `faux`.
Par exemple les phrases suivantes définissent des propositions

- p : « il pleut ce soir » ;
- q : « il existe une infinité de nombres premiers » ;
- r: « l'informatique est une science exacte ».

La vérité d'une telle proposition dépend du « contexte » dans lequel elle est formulée. Par exemple, la proposition p dépend du lieu où l'on se trouve, et r dépend de la personne concernée.
Néanmoins, il est possible de faire des raisonnements logiques en ne connaissant pas à priori les valeurs de vérité des propositions. Par exemple :

- A : « S'il pleut, je prends mon imperméable » ;
- B : « Si j'ai mon imperméable, je ne suis pas mouillé » ;
- C : « S'il ne pleut pas, je ne suis pas mouillé ».

Si ces trois propositions sont vraies, on peut en déduire par un raisonnement logique que je ne suis jamais mouillé. Pour formaliser ce raisonnement, considérons les trois propositions suivantes :

- p : « Il pleut » ;
- q : « J'ai mon imperméable » ;
- r : « Je suis mouillé ».

Les trois propositions A,B,C s'écrivent alors en calcul propositionnel :

- $A = p \Rightarrow q$;
- $B = q \Rightarrow \overline{r}$;
- $C = \overline{p} \Rightarrow \overline{r}$.

A,B,C sont appelées *propositions logiques* et p,q,r sont appelées *variables propositionnelles*. Si l'on peut par des transformations formelles déduire que sous les hypothèses que les propositions A, B et C sont vraies, la proposition $\overline{r}$ est vraie, alors on aura montré que je ne suis jamais mouillé.
Mais on aura démontré aussi que si, dans une certaine arithmétique on a les hypothèses suivantes :

- (α) : Si $1 = 0$ alors $1 + 1 = 1$
- (β) : Si $1 + 1 = 1$, alors $1 + 1 \neq 0$
- (γ) : Si $1 \neq 0$, alors $1 + 1 \neq 0$

alors on aura automatiquement $1 + 1 \neq 0$. Il suffit pour cela de poser

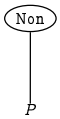
- p : «$1 = 0$»
- q : «$1 + 1 = 1$»
- r : «$1 + 1 = 0$»

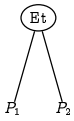
et le raisonnement précédent donnera le résultat.
Nous allons dans un premier temps définir formellement la *syntaxe des propositions logiques*, et dans un second temps étudier la *sémantique* des propositions logiques (leur affecter une *valeur*).

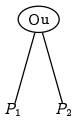
6.2 Syntaxe des propositions logiques

6.2.1 Définition inductive des propositions logiques

On peut définir inductivement l'ensemble des propositions logiques en utilisant les constructeurs `Non` (d'arité 1), `Et` (d'arité 2) , `Ou` (d'arité 2) de la façon suivante :

- Les éléments atomiques sont les variables propositionnelles (appartenant à un certain ensemble) et deux constantes V et F. Ils sont représentés par une feuille d'un arbre $\textcircled{v}$;
- Si P est une proposition, Non(P) est une proposition, représentée par l'arbre 

;
- Si P_1 et P_2 sont deux propositions, Et(P_1,P_2) est une proposition, représentée par l'arbre 

;
- Si P_1 et P_2 sont deux propositions, Ou(P_1,P_2) est une proposition, représentée par l'arbre 

.

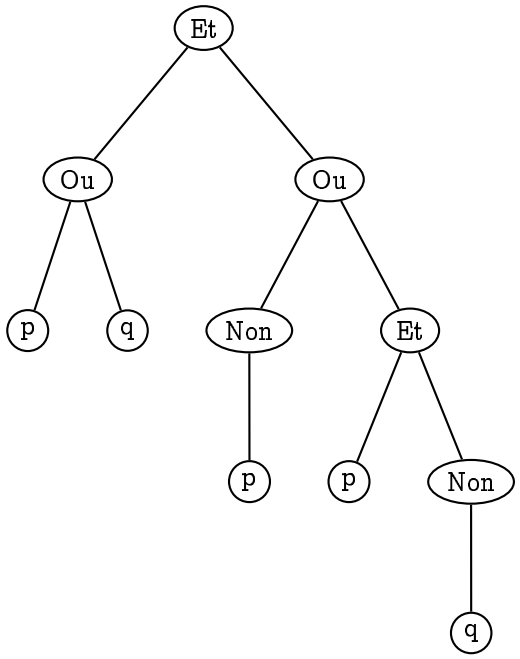
Remarque 27.

- On trouve également dans les livres une définition des propositions sans les constantes V et F.
- Certains auteurs appellent *propositions* nos variables propositionnelles, et *formules logiques*, nos propositions logiques.

Si A est une proposition obtenue par application d'un nombre fini de ces règles inductives, on peut la représenter par un arbre dont les feuilles correspondent aux éléments « atomiques » (c'est à dire des variables propositionnelles), et les noeuds aux constructeurs utilisés. Par exemple :

```
A = Et(
Ou(p, q),
Ou(
   Non(p),
   Et(p, Non(q))
  )
)
```

est représentée par l'arbre suivant :



Définissons le type CAML correspondant :

CAML

```
type proposition =
  V
```

```
  | F
  | Var of char
  | Non of  proposition
  | Et of proposition * proposition
  | Ou of proposition * proposition;;
```

La proposition ci-dessus se définit alors en CAML par :

CAML

```
let p = Et(
  Ou(Var 'p', Var 'q'),
  Ou(
    Non(Var 'p'),
    Et(Var 'p', Non(Var 'q'))
  )
);;
p : proposition =
 Et (Ou (Var 'p', Var 'q'), Ou (Non (Var 'p'), Et (p, Non (Var 'q'))))
```

6.2.2 Notation linéaire d'une proposition logique

La notation avec constructeurs est lourde. On préfère noter les propositions logiques sous une forme linéaire infixe en utilisant un parenthésage. A une proposition logique P, on associe la chaîne de caractère $\widetilde{P}$ définie inductivement par :

- Si $P = v$ où v est une variable (ou une constante V, F), $\widetilde{P}$ est la chaîne de caractères correspondant au nom de la variable v ;
- Si $P = \text{Non}(P_1)$, $\widetilde{P} = \neg\widetilde{P_1}$;
- Si $P = \text{Et}(P_1,P_2)$, $\widetilde{P} = (\widetilde{P_1} \wedge \widetilde{P_2})$;
- Si $P = \text{Ou}(P_1,P_2)$, $\widetilde{P} = (\widetilde{P_1} \vee \widetilde{P_2})$.

La fonction d'affichage en CAML est immédiate. (On a remplacé les caractères $\neg,\wedge,\vee$ par des chaînes de caractères ASCII).

CAML

```
let rec linéaire p =
  match p with
    | V -> print_char 'V'
    | F -> print_char 'F'
    | Var c ->  print_char c
    | Non p1 -> print_string " non "; linéaire p1
    | Et (p1, p2) ->
        print_char '(';
        linéaire p1;
        print_string " et ";
        linéaire p2;
        print_char ')'
    | Ou (p1, p2) ->
        print_char '(';
        linéaire p1;
        print_string " ou ";
        linéaire p2;
        print_char ')';;
```

```
#linéaire p;;
((p ou q) et ( non p ou (p et  non q)))- : unit = ()
```

Pour reconstituer l'arbre d'une propoposition à partir de la notation linéaire, les parenthèses sont nécessaires (comme pour les expressions algébriques), mais elles deviennent vite envahissantes. On peut en supprimer

certaines s'il n'y a pas d'ambiguité. On notera par exemple $\neg p \wedge (q \vee r)$ à la place de $(\neg p \wedge (q \vee r))$.
Remarque 28. On pourrait utiliser également une notation postfixe, comme pour les expressions algébriques et se passer alors de parenthèses.

6.3 Sémantique

6.3.1 Évaluation d'une proposition logique

Considérons des propositions logiques construites à partir de variables propositionnelles. Chacune des variables ne peut prendre que deux valeurs logiques, 1 (correspondant à « vrai ») ou 0 (correspondant à « faux »).
Fixons une valeur $\sigma(p)$ à chacune des variables propositionnelles (σ est une fonction de l'ensemble des variables propositionnelles vers $\{0,1\}$, et s'appelle une *distribution de vérité*). 0n définit inductivement la valeur $V_\sigma(P)$ d'une proposition P quelconque de la façon suivante :

- Si $P = V$, $V_\sigma(P) = 1$, si $P = F$, $V_\sigma(P) = 0$;
- Si $P = v$, $V_\sigma(P) = \sigma(v)$;
- Si $P = \text{Non}(P_1)$, $V_\sigma(P) = 1 - V_\sigma(P_1)$;
- Si $P = \text{Et}(P_1,P_2)$, $V_\sigma(P) = \min\big(V_\sigma(P_1),V_\sigma(P_2)\big)$;
- Si $P = \text{Ou}(P_1,P_2)$, $V_\sigma(P) = \max\big(V_\sigma(P_1),V_\sigma(P_2)\big)$.

Si une proposition A s'exprime à l'aide de n variables propositionnelles $(v_1,\ldots,v_n)$, il y a 2^n valeurs possibles pour le n-uplet $\big(\sigma(v_1),\ldots,\sigma(v_n)\big)$. On résume l'évaluation d'une proposition A en fonction des valeurs de ses variables dans une *table de vérité* à 2^n lignes. Par exemple, pour la proposition $A = p \vee q$:

p	q	$p \vee q$
1	1	1
1	0	1
0	1	1
0	0	0

6.3.2 Propositions logiquement équivalentes

DÉFINITION 6.1: **Propositions logiquement équivalentes**
On dit que deux propositions logiques P et Q sont *logiquement équivalentes* si et seulement si pour toute distribution de vérité σ, $V_\sigma(P) = V_\sigma(Q)$.
Lorsque deux propositions logiques P et Q sont logiquement équivalentes, on note $P \equiv Q$.

On vérifie par exemple que les propositions $P = \neg\neg p$ et $Q = p$ sont logiquement équivalentes : il suffit d'examiner leur table de vérité. De même, vérifions que les propositions $A = (p \wedge q) \vee r$ et $B = (p \vee r) \wedge (q \vee r)$ sont logiquement équivalentes :

p	q	r	$p \wedge q$	$p \vee r$	$q \vee r$	A	B
1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1
1	0	1	0	1	1	1	1
1	0	0	0	1	0	0	0
0	1	1	0	1	1	1	1
0	1	0	0	0	1	0	0
0	0	1	0	1	1	1	1
0	0	0	0	0	0	0	0

Remarque 29. Deux propositions logiquement équivalentes peuvent être très différentes syntaxiquement. Par exemple, les propositions suivantes sont toutes logiquement équivalentes :

- $\neg(p \iff q)$

- $(p \vee q) \wedge \neg(p \wedge q)$
- $(p \wedge \neg q) \vee (q \wedge \neg p)$

Remarque 30. Soient P et Q sont deux propositions formées à partir des variables $(a_1, \ldots, a_n)$. Si P et Q sont logiquement équivalentes, on en déduit de nouvelles proposisions logiquement équivalentes en remplaçant les variables $(a_1, \ldots, a_n)$ par des propositions quelconques. Par exemple :

$$(p \vee q) \wedge \neg(p \wedge q) \equiv (p \wedge \neg q) \vee (q \wedge \neg p)$$

$$\big((a \wedge b) \vee (\neg a \wedge c)\big) \wedge \neg\big((a \wedge b) \wedge (\neg a \wedge c)\big) \equiv \big((a \wedge b) \wedge \neg(\neg a \wedge c)\big) \vee \big((\neg a \wedge c) \wedge \neg(a \wedge b)\big)$$

On peut étendre la syntaxe des propositions en définissant de nouveaux connecteurs et en précisant leur sémantique. Voici les plus courants à connaître. Leur évaluation se déduit de la table de vérité suivante :

p	q	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \Rightarrow q$	$p \iff q$	p NAND q	p NOR q
1	1	1	1	0	1	1	0	0
1	0	0	1	1	0	0	1	0
0	1	0	1	1	1	0	1	0
0	0	0	0	0	1	1	1	1

Remarque 31. Le connecteur $\oplus$ s'appelle « ou exclusif ». Soient (P,Q) deux propositions. La valeur de vérité de $V_\sigma(P \oplus Q)$ vaut 1 si et seulement si $V_\sigma(P)$ et $V_\sigma(Q)$ sont différents. On vérifie par induction structurelle que

$$P \text{ NAND } Q \equiv \neg(P \wedge Q)$$

$$P \text{ NOR } Q \equiv \neg(P \vee Q)$$

Remarque 32. La proposition $p \Rightarrow q$ n'est pas intuitive. La proposition $p \Rightarrow q$ est vraie sauf si p est vrai et q faux. En particulier, si p est faux, $p \Rightarrow q$ est vrai. On montre avec les tables de vérité que

$$\boxed{P \Rightarrow Q \equiv \neg P \vee Q}$$

DÉFINITION 6.2: **Système de connecteurs complet**
On dit qu'un système de connecteurs S est *complet* lorsque toute proposition est logiquement équivalente à une proposition formée uniquement à l'aide des connecteurs de S. Par définition de la syntaxe des propositions, le système $\{\neg, \wedge, \vee\}$ est complet.

Exercice 6-1
Montrer que $S = \{$ NAND $\}$ est un système de connecteurs complet.

6.3.3 Tautologies

DÉFINITION 6.3: **Satisfiabilité**
On dit qu'une proposition P est *satisfiable* s'il existe une distribution de vérité σ telle que $V_\sigma(P) = 1$. On dit alors que σ *satisfait* P.

Remarque 33. Le problème qui consiste à vérifier si une proposition P faisant intervenir n variables est satisfiable est l'un des problèmes fondamentaux en théorie de la complexité. On peut résoudre ce problème en temps $O(2^n)$ en examinant toutes les lignes de la table de vérité, mais on ne connaît pas à ce jour d'algorithme polynômial capable de le résoudre. Par contre, étant donné une distribution de vérité σ, il est facile de vérifier en temps polynômial si $V_\sigma(P) = 1$. On dit que c'est un problème NP. Ce problème est souvent noté SAT, et de nombreux problèmes d'optimisation difficiles (comme le problème du voyageur de commerce) se ramènent à la résolution de SAT par une transformation polynômiale.

DÉFINITION 6.4: **Tautologie**
Soit P une proposition. Si pour toute distribution de vérité σ, la valeur $V_\sigma(P)$ vaut 1, on dit que A est une *tautologie*.

Remarque 34. On ne connaît pas à ce jour d'algorithme qui vérifie qu'une proposition est une tautologie en temps polynômial. C'est un problème fondamental d'algorithmique considéré comme étant difficile.

Quelques exemples de tautologies :

- $(p \wedge q) \Rightarrow p$;
- $p \Rightarrow (p \vee q)$;
- $\big(p \wedge (p \Rightarrow q)\big) \Rightarrow q$;

Remarque 35. Deux propositions A et B sont logiquement équivalentes si et seulement si $A \Longleftrightarrow B$ est une tautologie.

Remarque 36. Si A est une tautologie, on peut remplacer les variables propositionnelles de A par des propositions quelconques, et obtenir de nouvelles tautologies.

Nous allons maintenant utiliser des notations plus simples pour les propositions logiques. Nous noterons $\overline{p}$ à la place de $\neg p$, $p.q$ (ou même pq) à la place de $p \wedge q$ et $p + q$ à la place de $p \vee q$.

Voici quelques tautologies classiques (on le montre en écrivant les tables de vérité) :

1. $p \Longleftrightarrow p$;
2. $\overline{\overline{p}} \Longleftrightarrow p$;
3. $F \Rightarrow p$;
4. $p \Rightarrow V$;
5. $p + \overline{p}$ (tiers exclu) ;
6. $\big(p.(p \Rightarrow q)\big) \Rightarrow q$ (modus ponens) ;
7. $\big((p \Rightarrow q).(q \Rightarrow r)\big) \Rightarrow (p \Rightarrow r)$ (transitivité de $\Rightarrow$) ;
8. $\big(p.(\overline{p} + q)\big) \Longleftrightarrow (p.q)$.
9. $p.p \equiv p$;
10. $p + p \equiv p$ (idempotence) ;
11. $\overline{\overline{p}} \equiv p$;
12. $(p.q).r \equiv p.(q.r)$ (associativité de $\wedge$) ;
13. $(p + q) + r \equiv p + (q + r)$ (associativité de $\vee$) ;
14. $p + q \equiv q + p$; (commutativité de $\vee$)
15. $p.q \equiv q.p$ (commutativité de $\wedge$) ;

Voici ensuite quelques équivalences logiques à connaître :

1. $(p.q) + r \equiv (p + r).(q + r)$ (distributivité de $\vee$ par rapport à $\wedge$) ;
2. $(p + q).r \equiv (p.r) + (q.r)$ (distributivité de $\wedge$ par rapport à $\vee$) ;
3. $p + \overline{p} \equiv V$ (tiers exclu) ;
4. $p.\overline{p} \equiv F$;
5. $p + F \equiv p$;
6. $p + V \equiv V$;
7. $p.V \equiv p$;
8. $p.F \equiv F$;
9. $p + (p.q) \equiv p$;
10. $p.(p + q) \equiv p$;
11. $p \Rightarrow q \equiv \overline{q} \Rightarrow \overline{p}$ (raisonnement par contraposition) ;
12. $p \Longleftrightarrow q \equiv (p \Rightarrow q).(q \Rightarrow p)$;
13. $p \Rightarrow q \equiv (\overline{p} + q)$ (expression de $\Rightarrow$ en fonction de la négation et du ou).

THÉORÈME 6.1: **Lois de De Morgan**
Pour toutes propositions p, q :

- $\overline{p.q} \equiv \overline{p} + \overline{q}$;
- $\overline{p + q} \equiv \overline{p}.\overline{q}$.

Remarque 37. Il faut être à l'aise avec le calcul propositionnel. Voici quelques applications courantes des équivalences logiques précédentes :

1. $a.b + c \equiv (a + c).(b + c)$;
2. $a.b + c + d \equiv (a + c + d).(b + c + d)$;
3. $(a + a.b) \equiv a$;

4. $(a + \overline{a} + b) \equiv V$;
5. $(a + \overline{a} + b).c \equiv c$;
6. ...

6.3.4 Formes conjonctives et disjonctives

DÉFINITION 6.5: **Littéral**
On appelle *littéral*, tout proposition logique de la forme $P = v$ ou $P = \overline{v}$ où v est une variable propositionnelle.

DÉFINITION 6.6: **Formes conjonctives et disjonctives**
Soit une proposition logique P formée à partir de variables propositionnelles $v_1, \ldots, v_k$. On dit que P est une *forme conjonctive* si P s'écrit

$$P = P_1 \wedge P_2 \wedge \ldots \wedge P_n$$

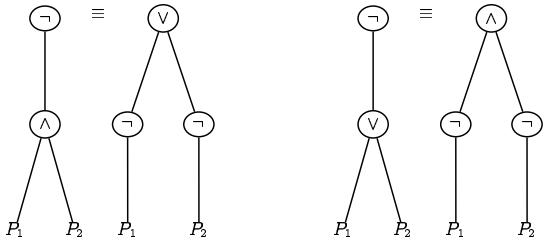
où les P_i sont des *clauses*, c'est à dire une disjonction de littéraux (par exemple $P_i = v_1 \vee \overline{v_3} \vee v_5$).
On dit que P est une *forme disjonctive* si P s'écrit

$$P = P_1 \vee P_2 \vee \ldots \vee P_n$$

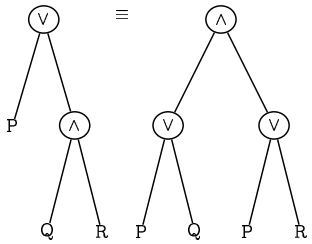
où chaque P_i est une conjonction de littéraux (par exemple $P_i = \overline{v_2} \wedge v_3 \wedge \overline{v_4}$).

Il est possible de transformer tout proposition P en une forme conjonctive logiquement équivalente en utilisant les étapes suivantes :

1. Remplacer tous les connecteurs apparaissant dans P en fonction des connecteurs basiques $\neg$, $\wedge$ et $\vee$;
2. Utiliser les lois de De Morgan pour faire descendre au maximum les connecteurs $\neg$: $\neg(P_1 \wedge P_2) \equiv \neg P_1 \vee \neg P_2$ et $\neg(P_1 \vee P_2) \equiv \neg P_1 \wedge \neg P_2$.



3. Utiliser la distributivité de $\vee$ par rapport à $\wedge$ pour faire descendre dans l'arbre les $\vee$: $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$.



Par exemple, considérons la proposition

$$P = (a \Rightarrow b) \Rightarrow (\overline{c} \wedge b)$$

L'algorithme précédent donne :

1. $P \equiv \neg(\neg a \vee b) \vee (\neg c \wedge b)$
2. $P \equiv (a \wedge \neg b) \vee (\neg c \wedge b)$
3. $P \equiv [a \vee (\neg c \wedge b)] \wedge [\neg b \vee (\neg c \wedge b)]$
4. $P \equiv [(a \vee \neg c) \wedge (a \vee b)] \wedge [(\neg b \vee \neg c) \wedge (\neg b \vee b)]$
5. $P \equiv (a \vee \neg c) \wedge (a \vee b) \wedge (\neg b \vee \neg c) \wedge (\neg b \vee b)$

Remarque 38. En échangeant le rôle de $\wedge$ et $\vee$, l'algorithme précédent permet de trouver une forme disjonctive d'une proposition quelconque.

Exercice 6-2

Mettre sous une forme disjonctive puis conjonctive la proposition

$$P = (p \iff q) \text{ NAND } (q \Rightarrow r)$$

Exercice 6-3

Un coffre-fort est muni de n serrures et peut être ouvert uniquement lorsque ces n serrures sont simultanément ouvertes. Cinq personnes : a, b, c, d et e doivent recevoir des clés correspondant à certaines serrures. Chaque clé peut être disponible en autant d'exemplaires qu'on le souhaite. On cherche n et une répartition des clés entre les cinq personnes, de telle manière que le coffre puisse être ouvert si et seulement si on se trouve dans une au moins des situations suivantes :

- présence simultanée de a et b ;
- présence simultanée de a, c et d ;
- présence simultanée de b, d et e.

On considère cinq variables propositionnelles notées a, b, c, d et e. La variable x vaut 1 si et seulement si x est présent.

1. Exprimer la proposition « le coffre-fort peut-être ouvert » à l'aide d'une proposition logique A sous forme disjonctive.
2. Transformer A en une forme conjonctive équivalente.
3. Déterminer une valeur de n et une répartition possible.

DÉFINITION 6.7: **Mintermes, maxtermes**

- On appelle *minterme*, constitué de k variables propositionnelles $(v_1, \ldots, v_k)$, une conjonction de littéraux dans laquelle figure exactement une fois chacune des k variables. Par exemple, $p.q$, $\overline{p}.q$, $p.\overline{q}$ et $\overline{p}.\overline{q}$ sont tous les mintermes que l'on peut former en les deux variables (p,q).
- On appelle *maxterme* une disjonction de littéraux dans laquelle figure exactement une fois chacune des k variables. Les maxtermes formés à l'aide de deux variables sont $p+q$, $\overline{p}+q$, $p+\overline{q}$ et $\overline{p}+\overline{q}$.

DÉFINITION 6.8: **Formes normales conjonctives et disjonctive**

Soit P une proposition logique formée à l'aide de k variables $v_1, \ldots, v_k$. On appelle :

- *Forme normale conjonctive* de P, toute proposition logiquement équivalente à P qui s'écrit comme une conjonction de maxtermes.
- *Forme normale disjonctive* de P, toute proposition logiquement équivalente à P qui s'écrit comme une disjonction de mintermes.

THÉORÈME 6.2: **Mise sous forme normale disjonctive d'une proposition**

Toute proposition logique formée à l'aide de k variables propositionnelles est logiquement équivalente à une proposition sous forme normale disjonctive (respectivement conjonctive). De plus, cette forme normale est unique à l'ordre près.

L'algorithme s'appuie sur les tables de vérité. Par exemple, si une proposition P dans les trois variables p,q,r a pour table de vérité

p	q	r	P
1	1	1	0
1	1	0	1
1	0	1	1
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	0
0	0	0	1

En examinant les lignes de la table de vérité contenant 1, on trouve la forme normale disjonctive de P :

$$P \equiv (p.q.\overline{r}) + (p.\overline{q}.r) + (\overline{p}.q.r) + (\overline{p}.\overline{q}.\overline{r})$$

Pour obtenir la forme normale conjonctive de P, on commence par écrire la forme normale disjonctive de $\overline{P}$. Il suffit pour cela d'examiner les lignes de la table de vérité contenant 0 :

$$\overline{P} \equiv (p.q.r) + (p.\overline{q}.\overline{r}) + (\overline{p}.q.\overline{r}) + (\overline{p}.\overline{q}.r)$$

En utilisant ensuite d'utiliser les lois de De Morgan on obtient la forme normale conjonctive :

$$P \equiv (\overline{p} + \overline{q} + \overline{r}).(\overline{p} + q + r).(p + \overline{q} + r).(p + q + \overline{r})$$

Exercice 6-4

1. Montrer qu'il existe un unique connecteur logique ϕ à trois variables $\phi(p,q,r)$ tel que pour toute variable s :
$$\phi(s,\overline{s},s) \equiv \phi(s,F,F) \equiv V \text{ et } \phi(s,s,\overline{s}) \equiv \phi(s,V,V) \equiv F$$
2. Donner la forme normale disjonctive de la proposition $A = \phi(p,q,r)$, puis sa forme normale conjonctive ;
3. Existe-t-il une forme disjonctive plus simple équivalente à A ?

6.4 Méthode de résolution

DÉFINITION 6.9: **Conséquence logique**
Une proposition Q est une *conséquence logique* d'une proposition P lorsque pour toute distribution de vérité σ telle que $V_\sigma(P) = 1$, on a $V_\sigma(Q) = 1$.

Remarque 39. Q est une conséquence logique de P si et seulement si $P \Rightarrow Q$ est une tautologie.
On dispose d'un nombre fini d'hypothèses (H_1), ..., (H_n) : ce sont des propositions dont on suppose que la valeur logique vaut 1. On veut montrer une *conclusion*, c'est à dire montrer que la valeur d'une proposition C vaut 1 également. En d'autres termes, il s'agit de montrer que la proposition

$$(H_1 \wedge \ldots \wedge H_n) \Rightarrow C$$

est une tautologie. Une *preuve* consiste à utiliser le calcul des propositions pour dériver une série de conséquences logiques intermédiaires $E_1, \ldots, E_k$ avec $E_k = C$.
On pourrait former la table de vérité de la proposition $(H_1 \wedge \ldots \wedge H_n) \Rightarrow C$ pour vérifier que c'est une tautologie, mais cette méthode serait de complexité exponentielle. On utilise une autre méthode dont la complexité dans le pire des cas est également exponentielle, mais qui s'avére plus rapide dans de nombreux cas.
Cette technique de résolution se base sur l'application de la tautologie

$$\big((p \vee q) \wedge (\neg p \vee r)\big) \Rightarrow (q \vee r) \tag{6.1}$$

qui permet d'obtenir une nouvelle conséquence logique où l'on a éliminé la variable p.

Exemple 8. *Reprenons l'exemple de l'introduction. À partir des affirmations :*
- « *S'il pleut, je prends mon imperméable* » ;
- « *Si j'ai mon imperméable, je ne suis pas mouillé* » ;
- « *S'il ne pleut pas, je ne suis pas mouillé* ».

Montrons que je ne serai jamais mouillé. Considérons les trois variables propositionnelles
1. p : « *il pleut* » ;
2. q : « *je prends mon imperméable* » ;
3. r : « *je suis mouillé* ».

Notons H_1, H_2 et H_3 les trois propositions logiques correspondant aux affirmations de l'énoncé :
1. $H_1 = p \Rightarrow q$;

2. $H_2 = q \Rightarrow \overline{r}$;
3. $H_3 = \overline{p} \Rightarrow \overline{r}$.

Formons les conséquences logiques suivantes :

1. $E_1 = \overline{p} + q$ *(équivalence logique de* H_1*)* ;
2. $E_2 = \overline{q} + \overline{r}$ *(équivalence logique de* H_2*)* ;
3. $E_3 = p + \overline{r}$ *(équivalence logique de* H_3*)* ;
4. $E_4 = q + \overline{r}$ *(règle 6.1 de* E_1 *et* E_3*)* ;
5. $E_4 = \overline{r} + \overline{r} \equiv \overline{r}$ *(règle 6.1 de* E_2 *et* E_4*)*.

On a donc prouvé que $\overline{r}$ *était vrai et par conséquent, je ne suis pas mouillé.*

Une autre technique pour montrer qu'une proposition C est une conclusion logique d'une série d'hypothèses $H_1, \ldots, H_n$, consiste à raisonner par l'absurde en montrant que $Faux$ est une conséquence logique des hypothèses $H_1, \ldots, H_n, \neg C$.

La technique complète de résolution, pour montrer qu'une proposition C est une conséquence logique d'hypothèses $H_1, \ldots, H_n$ est la suivante :

1. Mettre les hypothèses et $\neg C$ sous forme conjonctive (pas forcément normale) : $H_1 = C_{11} \wedge \ldots \wedge C_{1p_1}, \ldots, H_n = C_{n1} \wedge \ldots \wedge C_{np_n}$;
2. On obtient alors $p_1 \times \cdots \times p_n$ hypothèses H'_i qui sont des *clauses* (disjonctions de littéraux) ;
3. Utiliser systématiquement la règle d'élimination 6.1 pour obtenir des étapes E_k, en éliminant des variables.
4. Aboutir à la conséquence logique F.

Exercice 6-5

Vous êtes perdu sur une piste dans le désert. Vous arrivez à une bifurcation. Chacune des deux pistes est gardée par un sphinx que vous pouvez interroger. Les pistes peuvent conduire à une oasis, soit se perdre dans un désert profond (au mieux elles conduisent toutes deux à une oasis, au pire elles se perdent toutes les deux).

A . Le sphinx de droite vous répond : **Une au moins des deux pistes conduit à une oasis**.

B . Le sphinx de gauche vous répond : **La piste de droite se perd dans le désert**.

C . Vous savez que les deux sphinx disent tous les deux la vérité, ou bien mentent tous les deux.

Si D est la proposition « Il y a une oasis au bout de la route de droite » et G est la proposition « Il y a une oasis au bout de la route de gauche » ,alors

1. Exprimer par une formule de la logique de propositions les affirmations **A** et **B** ;
2. Exprimer alors la connaissance **C** ;
3. Résoudre l'énigme en utilisant les tables de vérité ;
4. Résoudre l'énigme en utilisant le calcul des propositions ;

Exercice 6-6

On interroge un logicien qui dit toujours la vérité sur sa vie sentimentale. À la question : « Est-il vrai que si vous aimez Marie, alors vous aimez Anne? », il a répondu :

- Si c'est vrai, alors j'aime Marie ;
- Si j'aime Marie, alors c'est vrai.

Qu'en concluez-vous?

6.5 Circuits logiques

6.5.1 Fonctions booléennes

DÉFINITION 6.10: **Fonction booléenne** On appelle *fonction booléenne* de k variables, une fonction $f : \{0,1\}^k \mapsto \{0,1\}$.

À Toute proposition logique P à k variables $(v_1, \ldots, v_k)$, on associe une fonction booléenne f_P, que l'on définit à partir de la table de vérité de P. Réciproquement, étant donnée une fonction booléenne f à k variables, on

peut lui associer plusieurs propositions P dépendant de k variables booléennes telles que $f_P = f$. Un moyen simple de déterminer une proposition associée à une fonction booléenne est d'écrire la fonction f à l'aide d'une table et d'utiliser la forme normale disjonctive (ou conjonctive) associée :

Par exemple, à la fonction booléenne de deux variables f donnée par la table :

p	q	f(p, q)
1	1	0
1	0	1
0	1	0
0	0	1

on peut associer la proposition logique

$$P = (p.\overline{q}) + (\overline{p}.\overline{q})$$

6.5.2 Circuits logiques

Un ordinateur est constitué de circuits électriques dans lesquels passe ou ne passe pas du courant. À chaque points d'entrée $P_1, \ldots, P_k$ du circuit est porté une certaine tension (0 V ou 5 V) et à la sortie Q , on récupère une certaine tension :

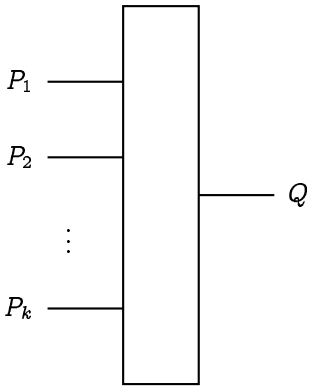


FIG. 6.1 – *Un circuit électrique à k entrées et une sortie*

Un circuit *câble* une fonction booléenne $f : \{0,1\}^k \mapsto \{0,1\}$ telle que $f(p_1, \ldots, p_k)$ représente la valeur logique de la sortie si les points d'entrée du circuit sont affectés des valeurs logiques $p_1, \ldots, p_k$. Dans ce qui suit, on négligera le temps nécessaire à un circuit pour propager les signaux. On parle alors de circuits *combinatoires*, par opposition aux circuits *séquentiels* (pour lesquels il est possible de connecter la sortie d'un circuit à son entrée).

Il existe des circuits préfabriqués à l'aide de transistors, appelés *portes logiques*. Ils correspondent aux connecteurs logiques usuels, et leurs représentation schématique est donnée à la figure 6.2.

On s'intéresse au problème de câblage d'une fonction booléenne donnée.

`Exemple 9.` *On veut concevoir un afficheur à cristaux liquides qui permet d'afficher des chiffres en base* 10. *Pour simplifier le problème, on décide d'afficher des entiers codés en base* 2 *sur* 3 *bits (donc des entiers compris entre* 0 *et* 7*). Notre afficheur est constitué de* 7 *diodes que l'on peut allumer ou non. Sur la figure 6.3, on a affiché le chiffre* 4 *en allumant les diodes* d_2, d_3, d_4 *et* d_6.

La fonction booléenne qui correspond à l'affichage de la diode $d1$ *est donnée par la table suivante :* b_1, b_2, b_3 *correspondent aux trois chiffres de l'entier* n *à afficher :* $n = (b_1 b_2 b_3)_2$.

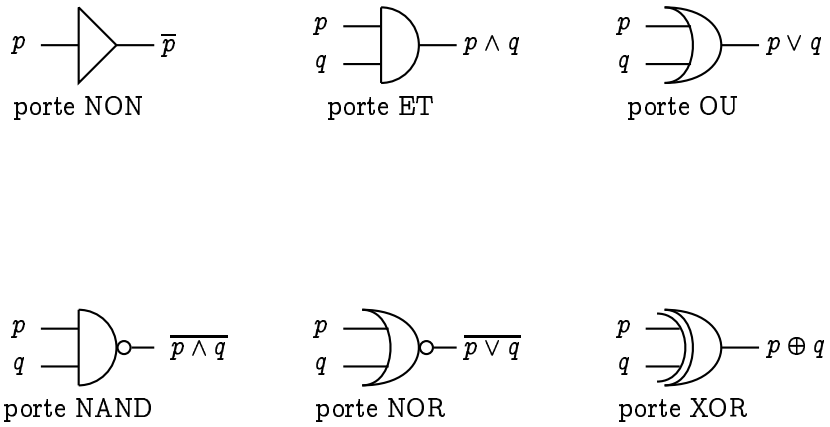


FIG. 6.2 – *Portes logiques élémentaires*

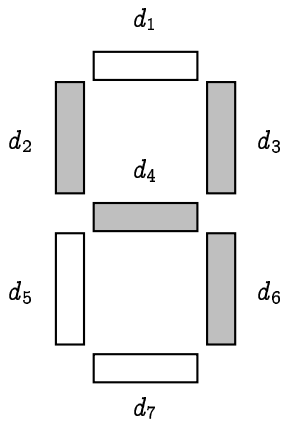


FIG. 6.3 – *Un afficheur LCD*

a	b	c	n	$f_3(n)$
1	1	1	7	1
1	1	0	6	0
1	0	1	5	0
1	0	0	4	1
0	1	1	3	1
0	1	0	2	1
0	0	1	1	1
0	0	0	0	1

Déterminons la forme normale conjonctive de la proposition correspondant à cette fonction booléenne :

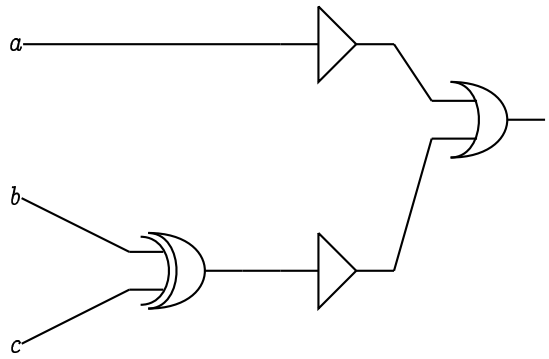
$$\overline{P} \equiv (a.b.\overline{c}) + (a.\overline{b}.c)$$

$$P \equiv (\overline{a} + \overline{b} + c).(\overline{a} + b + \overline{c})$$

En factorisant $\overline{a}$, on peut simplifier cette proposition :

$$P \equiv \overline{a} + \left[(\overline{b} + c).(b + \overline{c})\right] \equiv \overline{a} + \overline{b \oplus c}$$

Voici un câblage qui réalise cette fonction booléenne :



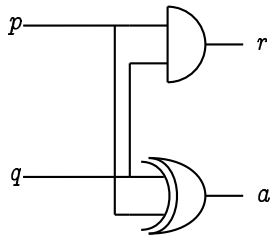
Remarque 40. Un problème important dans la conception des circuits logiques consiste à minimiser le nombre de portes logiques élémentaires utilisées. C'est un problème délicat d'optimisation.

6.5.3 Additionneurs

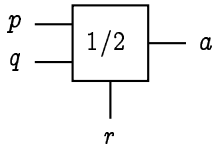
Une opération fondamentale, câblée dans les microprocesseurs consiste à réaliser l'addition de deux entiers codés sur 32 ou 64 bits. On utilise pour cela des *additionneurs*.
Commençons par un circuit simple qui permet d'additionner deux entiers p, q codés sur 1 bit. La sortie du circuit donne le bit de poids le plus faible de l'entier $p + q$, noté a et la retenue éventuelle r. Voici la table de la fonction booléenne correspondante :

p	q	a	r
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

Une proposition correspondant à a est simplement le « ou exclusif ». Une formule correspondant à r est $p.q$. Voici un circuit qui câble ces deux fonctions.



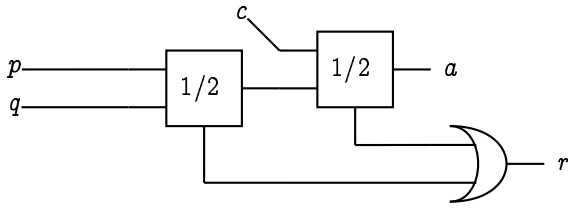
Ce circuit s'appelle un *demi-additionneur*. Notons le



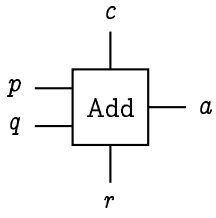
Réalisons maintenant un additionneur complet de deux bits p et q en tenant compte d'une retenue précédente c. Le résultat est formé de la somme $a = p + q + c$ et d'une nouvelle retenue r. Commençons par former la somme (a_1,r_1) de p et q à l'aide d'un demi-additionneur, et ensuite formons la somme (a_2,r_2) de $a_1 + c$ à l'aide d'un autre demi-additionneur. Voici la table des fonctions logiques correspondantes :

p	q	c	a_1	r_1	a_2	r_2	a	r
1	1	1	0	1	1	0	1	1
1	1	0	0	1	0	0	0	1
1	0	1	1	0	0	1	0	1
1	0	0	1	0	1	0	1	0
0	1	1	1	0	0	1	0	1
0	1	0	1	0	1	0	1	0
0	0	1	0	0	1	0	1	0
0	0	0	0	0	0	0	0	0

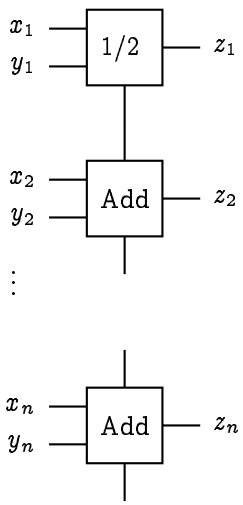
Cette table montre que $a = a_2$ et que $r = r_1 \vee r_2$. Voici donc le circuit correspondant à un additionneur complet 1-bit :



Notons-le



Pour réaliser un additionneur complet n-bits, qui calcule la somme des entiers $(x_n \dots x_1)_2 + (y_n \dots y_1)_2 = (z_n \dots z_1)_2$, il suffit de mettre en série plusieurs additionneurs pour propager les retenues :



Remarque 41. Ce circuit n'est pas satisfaisant si l'on prend en compte les délais de propagation qui sont proportionnels au nombre de bits. On peut améliorer le circuit en utilisant une méthode « diviser pour régner ».

Annexe A

Types construits en CAML

A.1 Types produit

Il est possible de définir en CAML de nouveaux types. Par exemple, pour représenter un point dans le plan, on pourrait utiliser un couple de flottants, mais cela conduirait à des programmes confus. Il est préférable de dire qu'un point est constitué d'une abscisse x et d'une ordonnée y. On utilise pour cela un *type produit* (qui correspond mathématiquement à un produit cartésien de deux types existants). On représente l'abscisse et l'ordonnée d'un point en utilisant des *étiquettes* x et y :

CAML
```
#type point = {x : float; y : float};;
Le type point est défini.
```

On peut désormais définir des objets de type `point`, et accéder à leurs abscisses et ordonnées :

CAML
```
#let p1 = {x = 0.5; y = 4.1};;
p1 : point = {x = 0.5; y = 4.1}

#p1.x;;
- : float = 0.5
#p1.y;;
- : float = 4.1
```

et définir des fonctions qui manipulent des points :

CAML
```
#type vecteur = {dx : float; dy : float};;
Le type vecteur est défini.

#let translate p v  =
  let x1 = p.x +. v.dx  and y1 = p.y +. v.dy in
    {x = x1; y = y1};;
translate : point -> vecteur -> point = <fun>
```

Remarquez le typage de la fonction qui est bien plus clair que si l'on avait utilisé des couples !
Il n'est pas possible de modifier un point une fois qu'il est défini. Pour remédier à cela, CAML permet de rendre certains champs *mutables* dans un type produit :

CAML
```
#type point = {mutable x : float; y : float};;
Le type point est défini.

#let p = {x = 0.3; y = 2.};;
p : point = {x = 0.3; y = 2.0}

#p.x <- 2.;;
- : unit = ()
```

```
#p.y <- 5.;;
Entrée interactive:
>p.y <- 5.;;<EOF>
>^^^^^^^^^
L'étiquette y n'est pas mutable.
```

Voici comment écrire la fonction translation pour qu'elle *modifie* les points :

CAML

```
#type point = {mutable x : float; mutable y : float};;
Le type point est défini.

#let p = {x = 0.3; y = 2.};;
p : point = {x = 0.3; y = 2.0}

#type vecteur = {dx : float; dy : float};;
Le type vecteur est défini.

#let v = {dx = 1.; dy = 2.};;
v : vecteur = {dx = 1.0; dy = 2.0}

#let translate p v =
    p.x <- p.x +. v.dx
    p.y <- p.y +. v.dy;;
translate : point -> vecteur -> unit = <fun>

#translate p v;;
- : unit = ()

#p;;
- : point = {x = 1.3; y = 4.0}
```

A.2 Types somme

1. Types énumérés.
   On peut définir un nouveau type avec un nombre fini d'éléments à l'aide de *constructeurs* d'arité zéro (des constantes). Par exemple, pour représenter les figures d'un jeu de cartes à l'aide des constantes `Valet, Dame, Roi, As`, on définit le

   CAML

   ```
   type figure = Valet | Dame | Roi | As;;
   ```

   La barre verticale s'interprète ici comme « ou ». Pour écrire des fonctions qui manipulent les éléments de ce nouveau type, on utilise le filtrage :

   CAML

   ```
   let valeur_figure f =
     match f with
       | Valet -> 2;
       | Dame -> 3;
       | Roi -> 4;
       | As -> 5;;

   valeur_figure : figure -> int = <fun>
   ```

   L'usage veut qu'en CAML, les noms des constructeurs commencent par une majuscule.

2. Types Somme

   On peut construire des types correspondant mathématiquement à l'union disjointe de deux types. On utilise des constructeurs paramétrés pour accéder aux types d'origine :

CAML
```
type nombre = Entier of int | Reel of float;;
```

   Écrivons une fonction qui additionne deux nombres, en utilisant le filtrage :

CAML
```
let addition_nombres x y =
  match (x, y) with
    | Entier p, Entier q -> Entier (p + q)
    | Reel p, Reel q -> Reel (p +. q)
    | Entier p, Reel q -> Reel (float_of_int p +. q)
    | Reel p, Entier q -> Reel (p +. float_of_int q);;

addition_nombres : nombre -> nombre -> nombre = <fun>
```

   Voici un autre exemple pour faire de l'arithmétique dans $\overline{\mathbb{R}}$:

CAML
```
type reel_bar = Reel of float | MInfini | PInfini ;;
Le type reel_bar est défini.

#let rec  addition x y =
  match (x, y) with
    | Reel x, Reel y -> Reel (x +. y)
    | Reel x, MInfini -> MInfini
    | Reel x, PInfini -> PInfini
    | MInfini, PInfini -> failwith "operation non definie"
    | _ -> addition y x;;
addition : reel_bar -> reel_bar -> reel_bar = <fun>

#addition (Reel 3.)  (Reel 4.);;
- : reel_bar = Reel 7.0

#addition (Reel 4.) PInfini;;
- : reel_bar = PInfini

#addition PInfini MInfini;;
Exception non rattrapée: Failure "operation non definie"
```

3. Types récursifs

   C'est la définition de type la plus intéressante pour nous, qui est très proche de la théorie de l'induction structurelle. Par exemple, nous avons défini une liste d'entiers comme étant :

   - Soit la liste vide, représentée par la constante `NIL` ;
   - Soit de la forme `Cons(i, q)` où i est un entier et q une liste. Le constructeur paramétré `Cons` est d'arité 1 (il prend un objet de type liste comme argument).

   En caml, cela donne :

CAML
```
#type liste = NIL | Cons of int * liste;;
Le type liste est défini.
```

CAML
```
#let l = Cons (1, Cons(2, Cons(3, NIL)));;
l : liste = Cons (1, Cons (2, Cons (3, NIL)))
```

et la fonction qui calcule la longueur d'une telle liste s'écrit alors :

CAML
```
#let rec longueur l =
  match l with
    | NIL -> 0
    | Cons (x, q) -> 1 + longueur q;;

longueur : liste -> int = <fun>
```

Il est également possible de définir récursivement une liste d'éléments d'un type arbitraire (`'a`) :

CAML
```
#type 'a liste = NIL | Cons of 'a * 'a liste;;
Le type liste est défini.
```

Définissons maintenant un type abstrait pour représenter les expressions algébriques (avec uniquement les opérateurs `+`, `*`, `-`, `/`, les fonctions `sin`, `exp` et les variables de type `float`). Nous utilisons les constructeurs :

- `Variable` d'arité zéro pour représenter les *constantes* ;
- `Plus`, `Moins`, `Divise`, `Mult` d'arité 2 pour représenter les *opérateurs* ;
- `Sin`, `Exp` d'arité 1 pour représenter les *fonctions*.

CAML
```
type exp_alg =
    Variable of float
  | Plus of exp_alg * exp_alg
  | Moins of exp_alg * exp_alg
  | Divise of exp_alg * exp_alg
  | Mult of exp_alg * exp_alg
  | Sin of exp_alg
  | Exp of exp_alg;;

Le type exp_alg est défini.
```

Pour écrire une fonction qui imprime une expression algébrique en notation postfixe, il suffit d'utiliser le filtrage :

CAML
```
let rec postfixe e =
  match e with
    | Variable x -> print_float x; print_char ' '
    | Plus (e1, e2) -> postfixe e1; postfixe e2; print_string " + "
    | Moins (e1, e2) -> postfixe e1; postfixe e2; print_string " - "
    | Divise (e1, e2) -> postfixe e1; postfixe e2; print_string " / "
    | Mult (e1, e2) -> postfixe e1; postfixe e2; print_string " * "
    | Sin e1 -> postfixe e1; print_string " sin "
    | Exp e1 -> postfixe e1; print_string " exp ";;

postfixe : exp_alg -> unit = <fun>

#let e = Plus(Sin( Mult(Variable 3., Variable 2. )) ,
              Divise(Variable 2., Moins( Variable 5., Variable 1.)));;
e : exp_alg =
 Plus
  (Sin (Mult (Variable 3.0, Variable 2.0)),
   Divise (Variable 2.0, Moins (Variable 5.0, Variable 1.0)))

#postfixe e;;
3.0 2.0  *  sin 2.0 5.0 1.0  -  /  + - : unit = ()
```

Il est simple d'écrire une fonction qui évalue une expression algébrique :

CAML
```
let rec evalue e =
  match e with
    | Variable x -> x
    | Plus (e1, e2) -> (evalue e1) +. (evalue e2)
    | Moins (e1, e2) -> (evalue e1) -. (evalue e2)
    | Divise (e1, e2) -> (evalue e1) /. (evalue e2)
    | Mult (e1, e2) ->  (evalue e1) *. (evalue e2)
    | Sin e1 -> sin (evalue e1)
    | Exp e1 -> exp (evalue e1);;

evalue : exp_alg -> float = <fun>
```

Nous rencontrons ici un problème bien connu en théorie de la compilation. Les algorithmes qui manipulent les expressions algébriques, s'écrivent facilement lorsque les expressions algébriques sont données sous forme abstraite. Mais on ne peut pas demander à un utilisateur de rentrer une expression algébrique sous cette forme. Il serait bien plus agréable d'entrer l'expression sous la forme $(2+3)*\sin(2.23)+2/5.3$ par exemple (syntaxe concrète), et de laisser à la machine le rôle de traduire cette entrée en un objet de type `exp_alg` (*syntaxe abstraite*). Ce procédé de transformation se fait en général en deux étapes appelées *analyse lexicale* et *analyse syntaxique*.
Ce problème est central en théorie de la compilation : comment passer d'un code source défini dans un langage de programmation (syntaxe concrète) à un exécutable binaire. Une première étape effectuée par les compilateurs consiste à transformer le code source d'entrée en un arbre de syntaxe abstrait qui permet de vérifier la *syntaxe* d'un programme. L'arbre de syntaxe est plus facile à manier par la suite pour le transformer en code machine. Un compileur est traditionnellement divisé en trois parties :

(a) Un *analyseur lexical* qui lit le flux de caractères de l'entrée présentée sous une forme concrète et le sépare en unités lexicographiques appelées *lexèmes* Son rôle consiste à ignorer les espaces, les commentaires, reconnaître des nombres, les mots-clés du langage...) ;
(b) Un *analyseur syntaxique* qui prend la sortie de l'analyseur lexical (un flux de lexèmes) et qui la transforme en *arbre de syntaxe* (syntaxe abstraite) ;
(c) La partie consacrée à la *sémantique* : interprètation de l'arbre de syntaxe abstraite pour produire du code intermédiaire, l'optimiser ensuite pour enfin produire un exécutable en code machine.

Cette démarche permet de bien séparer la partie *syntaxe* (qui correspond à la *grammaire* du langage de programmation), de la partie *sémantique* (qui correspond au *sens* du programme).

Annexe B

Aide mémoire CAML

1. Entiers

`succ n`	Successeur de n
`pred n`	Prédécesseur de n
`mod`	Reste de la division euclidienne
`/`	Quotient de la division euclidienne
`abs`	valeur absolue
`int_of_float`	conversion réel vers entier
`string_of_int`	conversion entier vers chaîne
`int_of_string`	conversion chaîne vers entier

2. Manipulation de bits

`land n p`	Et logique bit à bit
`lor n p`	Ou logique bit à bit
`lxor n p`	Ou exclusif bit à bit
`lnot n`	complément bit à bit
`n lsl m`	Décale n de m bits à gauche
`n lsr m`	Décale n de m bits à droite

3. Flottants

`+. -. *. /.`	Opérateurs sur les réels
`x ** n`	Exponentiation
`<. >. <=. ...`	Opérateurs de comparaison
`exp log sqrt sin cos`	fonctions mathématiques
`tan asin acos atan`	fonctions mathématiques
`abs_float`	Valeur absolue
`ceil floor`	Partie entière inférieure-supérieure (retourne un réel)
`sin, exp, atan`	Fonctions mathématiques
`float_of_int`	conversion entier vers réel
`int_of_float`	conversion réel vers entier

4. Opérations sur les booléens.

`A && B`	Et booléen (évaluation paresseuse)
`A || B`	Ou booléen (évaluation paresseuse)
`not A`	Non booléen
`< <= = <> >= >`	Comparaisons

5. Caractères

`‘a‘`	Définition d'un caractère (entre accents graves)
`char_of_int`	Conversion caractère vers code ASCII
`int_of_char`	Conversion code ASCII vers caractère

6. Chaînes de caractères

`let s = "Bonjour"`	Définition d'une chaîne
`s.[i]`	Accès au ième caractère d'une chaîne
`s.[i] <- 'a'`	Remplacement du ième caractère
`s ^ t`	Concaténation de chaînes
`string_length`	Longueur d'une chaîne
`make_string n 'a'`	Création d'une nouvelle chaîne de longueur n remplie de a
`sub_string debut longueur`	Sous-chaîne
`print_string "Hello"`	Impression d'une chaîne
`print_newline ()`	Impression d'un retour à la ligne

7. Références

`let x = ref 0`	Définition d'une référence
`!x`	contenu de la référence
`x := val`	Modification du contenu de la référence

8. Vecteurs

`[|0; 1; 2|]`	syntaxe d'un vecteur
`t.(i)`	ième élément d'un vecteur
`t.(i) <- val`	modification d'un élément du vecteur
`vect_length t`	longueur d'un vecteur
`make_vect l v`	Crée un nouveau vecteur de longueur l rempli de `v`
`sub_vect i long`	renvoie le sous-vecteur `[|t.(i); ... ;t.(i+long-1)|]`
`concat_vect t1 t2`	Concaténation de deux vecteurs
`let t = copy_vect t1`	Copie d'un vecteur
`vect_of_list`	conversion liste vers vecteur

9. Listes

`[1; 2; 3]`	Définition d'une liste
`[ ]`	Liste vide
`x :: l`	Ajout d'un élément en tête de liste
`x :: q`	motif de filtrage sur les listes
`hd l`	Retourne la tête de liste
`tl l`	Retourne la queue de liste
`l1 @ l2`	Concaténation de deux listes
`list_length`	Longueur d'une liste
`mem x l`	Appartenance d'un élément x à la liste l
`map f l`	Applique la fonction f à tous les éléments de la liste
`list_of_vect`	Conversion liste vers vecteur
`rev l`	Image miroir d'une liste

10. Entrées-sorties

`print_int`	Affiche un entier
`print_float`	Affiche un réel
`print_char`	Affiche un caractère
`print_string`	Affiche une chaîne de caractères
`print_newline()`	Passe à la ligne

11. Boucles

```
for i = 0 to n do
  expression_1;
  ...
  expression_n
done;

while condition do
   expression_1;
   ...
   expression_n
done;
```

12. Tests

Test simple :

```
if condition then expression;
```

Test conditionnel :

```
if condition then expression_1
   else expression_2;
```

A la place d'une expression, on peut en placer plusieurs dans un bloc :

```
if condition then begin
   expression_1;
   ...
   expression_n
end;
```

```
if condition then begin
   expression_1;
   ...
   expression_n
end else begin
   expression_1;
   ...
   expression_n
end;
```

Annexe C

Conseils de présentation des programmes

Voici quelques conseils pour que vos programmes soient lisibles. Ils sont inspirés de la page web de l'auteur de caml, Pierre Weiss et sont disponibles à l'adresse suivante:

`http://caml.inria.fr/FAQ/pgl-fra.html`

C.1 Espaces

- Insérez un espace *avant* et *après* chaque opérateur infixe (un argument à gauche et un argument à droite) (`+ - := :: ...`):

```
let a=a+1;;  (* mauvais *)
let a = a + 1;; (* bon *)

| x::q->(f x) +1 (* mauvais *)
| x :: q -> (f x) + 1 (* bon *)

r :=1+!r ; (* mauvais *)
r := 1 + !r; (* bon *)
```

- Insérez un espace *après* chaque séparateur (`, ;`), mais pas avant:

```
let couple = (1,2);; (* mauvais *)
let couple = (1, 2);; (* bon *)

let l = [1;2;3;4] ;; (* mauvais *)
let l = [1; 2; 3; 4];; (* bon *)
```

C.2 Indentation des blocs

- Les point virgules sont des *séparateurs* d'instructions dans une séquence:

CAML
```
for i = 0 to n do
  a := !a + 1;
  b := 2 * b;
  print_int !a;
  print_int !b
done
```

– Pour les blocs `begin ... end`, placer le begin à la fin de la première ligne, et le `end` indenté au niveau de l'instruction.

CAML
```
if x > 2 then begin
  a := !a + 1;
  b := !b + 1
end
else begin
  a := !a - 2;
  b := !b - 1;
  end;
```

C.3 Filtrage

Respectez l'indentation suivante :

CAML
```
let f l =
  match l with
  | 0 -> 1
  | 1 -> 2
  | _ -> 0;;
```

Lorsque le membre droit d'un filtrage est long, passer à la ligne :

CAML
```
let rec f x y=
  match x with
  | 0 -> 0
  | _ ->
      let z = f (x - 1) y in
        if z > y then 1
        else 2;;
```

C.4 Parenthèses

Les parenthèses servent à définir une priorité inhabituelle. Ne pas abuser de parenthèses inutiles. Pour cela, il faut connaître les priorités des opérateurs du langage.

C.5 Commentaires

– Un commentaire placé avant une fonction décrit ce que cette fonction calcule, sous réserve que les hypothèses de validité (décrites dans le commentaire) soient satisfaites ;
– un commentaire dans le corps d'une fonction explique *comment* la fonction arrive au résultat (la personne qui lit le corps de la fonction veut savoir comment elle fonctionne). Dans un programme impératif, les meilleurs commentaires sont les invariants de boucles ;
– un commentaire qui explique ce que fait une instruction est nuisible (la meilleure explication est l'instruction du langage elle-même) ;
– ne pas placer de commentaires dans une fonction récursive. Justifier la fonction par une phrase.

Annexe D

Solutions des exercices

Exercice 1-1

```
let max t =
  (* Ne fonctionne qu'avec des tableaux non-vides *)
  let n = vect_length t in
  let m = ref t.(0) in
    for i = 1 to n - 1 do
      if t.(i) > !m then m := t.(i)
    done;
    !m;;
```

Exercice 1-3

Considérons l'invariant de boucle placé en commentaire :

```
let horner p x =
  let n = vect_length p in
  let y = ref p.(n - 1) in
    for k = n - 2 downto 0 do
      y := !y *. x +. p.(k)
    (* INV : y = p.(n-1) x^{n-k-1} + p.(n-2) x^{n-k-2} + ...
      + p.(k) *)
    done;
    !y;;
```

On considère un tableau $p = [|p_0; \ldots; p_{n-1}|]$ correspondant au polynôme $P(X) = p_0 + p_1 X + \cdots + p_{n-1}X^{n-1}$. Avant le premier passage ($k = n-1$), $y = p.(n-1)$ et donc l'invariant est vérifié. Si on suppose l'invariant vérifié, après le passage k, $y = p_{n-1}x^{n-k-1} + \cdots + p_k$, après le passage suivant correspondant à $k+1$, y contient $x \times (p_{n-1}x^{n-k-1} + \cdots + p_k) + p_{k+1}$, c'est à dire $y = p_{n-1}x^{n-1-(k+1)} + \cdots + p_k x + p_{k+1}$ et l'invariant est bien maintenu. Après le dernier passage, correspondant à $k = 0$, la référence y contient donc $y = p_{n-1}x^{n-1} + \cdots + p_1 x + p_0$ qui est bien la valeur $P(x)$ cherchée.

Exercice 1-4

```
let multiplie x y =
  let m = ref 0
  and a = ref x
  and b = ref y in
    while !b > 0 do
      if !b mod 2 = 1 then m := !m + !a;
      a := !a * 2;
      b := !b / 2
    (* INV a * b + m = x * y *)
    done;
    !m;;
```

Prouvons que la propriété placée en commentaire est un invariant de boucle. Avant le premier passage dans la boucle, puisque $m = 0$, $a = x$ et $b = y$, l'invariant est bien vérifié. Supposons l'invariant vrai après le ième passage dans la boucle et montrons qu'il est encore vrai au passage suivant. Étudions deux cas :

- b impair : $b = 2p + 1$. Après la première instruction, la nouvelle valeur de m est $m' = m + a$. Après les deux instructions suivantes, les nouvelles valeurs de a et b sont $a' = 2a$ et $b' = p$. Alors $a'b' + m' = (2a)p + (m + a) = a \times (2p + 1) + m = ab + m = xy$.
- b est pair : $b = 2p$. La valeur de m n'est pas modifiée : $m' = m$ et $a' = 2a$, $b' = b/2 = p$. Alors $a'b' + m' = (2a)p + m = ab + m = xy$.

Dans les deux cas, l'invariant est maintenu. Montrons la terminaison de l'algorithme. Puisque $b' = b/2$, lorsque $b > 0$, $b' < b$ et donc la valeur de la référence b (qui est un entier) décroit strictement à chaque passage. Il ne peut y avoir qu'un nombre fini de passages dans la boucle while.
Montrons la correction de la fonction. Après le dernier passage dans la boucle while, d'après l'invariant, $ab+m = xy$ et puisqu'on est sorti de la boucle, $b = 0$ ce qui montre que $m = xy$.
Déterminons le nombre de passages dans la boucle while. Si nous écrivons l'entier $b = y$ en base 2 :

$$b = b_p 2^p + \cdots + b_1 2 + b_0 \quad b_i \in \{0,1\}$$

puisque $0 \leq b_0 < 2$, on écrit la division euclidienne de b par 2 :

$$b = (b_p 2^{p-1} + \cdots + b_1) \times 2 + b_0$$

par conséquent, l'entier $b/2$ s'écrit avec les chiffres $(b_p \ldots b_1)$ en base 2. Le nombre de passages dans la boucle while vaut donc p, le nombre de chiffres de y en base 2. Mais si l'entier y s'écrit avec les chiffres $b_p \neq 0, \ldots b_0$ en base 2, un simple encadrement montre que

$$2^p \leq y \leq 1 + 2 + \cdots + 2^p = 2^{p+1} - 1$$

et donc $2^p \leq y < 2^{p+1}$ et en prenant le logarithme en base 2, $p \leq \log_2 y < p + 1$ ce qui montre que $p = \lfloor \log_2 y \rfloor$ (partie entière inférieure). L'algorithme nécessite donc $2 \lfloor \log_2 y \rfloor$ décalages de bits (multiplications et divisions par 2) et au plus $\lfloor \log_2 y \rfloor$ additions (pour mettre à jour la référence m).

Exercice 1-5

```
let drapeau t =
  let n = vect_length t in
  let i = ref (-1)
  and j = ref 0
  and k = ref n in
    while !j < !k do
      if t.(!j) = 0 then
        j := !j + 1
      else if t.(!j) = 1 then begin
        k := !k - 1;
        echange t !j !k
      end else begin
        i := !i + 1;
        echange t !i !j;
        j := !j + 1;
      end
    (* INV : pour 0 <= l <= n - 1,
           l <= i ==> t.(l) = -1
       et l <= i < j ==> t.(l) = 0
       et l >= k ==> t.(l) = +1
    *)
    done;;
```

Exercice 1-6

Considérons la suite $r_0 = a, r_1 = b > r_2 > \cdots > r_n = \text{pgcd}(a,b)$ de l'algorithme d'Euclide. L'idée pour trouver

un invariant est de définir un couple (u_k,v_k) « intermédiaire » de telle façon que $\forall k \in [\![0,n]\!]$, $u_k a + v_k b = r_k$. Pour cela, posons

$$\begin{cases} u_0 &= 1 \\ v_0 &= 0 \end{cases} \begin{cases} u_1 &= 0 \\ v_1 &= 1 \end{cases}$$

Pour que l'invariant soit vérifié au départ. Supposons l'invariant vrai pour $k = 0 \dots k$. On a donc $r_{k-1} = au_{k-1} + bv_{k-1}$ et $r_k = au_k + bv_k$. Puisque r_{k+1} est le reste de la division euclidienne de r_{k-1} par r_k,

$$r_{k-1} = q_k r_k + r_{k+1}$$

et donc

$$r_{k+1} = r_{k-1} - q_k r_k = (au_{k-1} + bv_{k-1}) - q_k(au_k + bv_k) = a(u_{k-1} - q_k u_k) + b(v_{k-1} - q_k v_k)$$

Pour maintenir l'invariant, il suffit donc de poser

$$\begin{cases} u_{k+1} &= u_{k-1} - q_k u_k \\ v_{k+1} &= v_{k-1} - q_k v_k \end{cases}$$

D'où la fonction :

```
let bezout a b =
  let r1 = ref a
  and r2 = ref b
  and u1 = ref 1
  and u2 = ref 0
  and v1 = ref 0
  and v2 = ref 1
  and temp = ref 0
  and q = ref 0 in
    while !r2 > 0 do
      q := !r1 / !r2;
      temp := !r1;
      r1 := !r2;
      r2 := !temp mod !r2;
      temp := !u1;
      u1 := !u2;
      u2 := !temp - !q * !u2;
      temp := !v1;
      v1 := !v2;
      v2 := !temp - !q * !v2
    (* INV : a * u1 + b * v1 = r1*)
    done;
    (* r2 = 0, r1 = pgcd(a, b), a * u1 + b * v1 = pgcd(a, b) *)
    (!u1, !v1);;
```

Exercice 2-1

```
mystere 5;;
5 4 3 2 1
1 2 3 4 5 - : unit = ()

#mystere "BONJOUR" 0;;
BONJOUR
ruojnob- : unit = ()
```

Exercice 2-3

La première fonction utilise la propriété d'additivité :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

```
let rec binomial n p =
  match (n, p) with
    | _, 0 -> 1
    | 0, _ -> 0
    | _ -> (binomial (n - 1) (p - 1)) + (binomial (n -1) p);;
```

et la deuxième la propriété de factorisation :

$$\binom{n}{p} = (n \times \binom{n-1}{p-1})/p$$

```
let rec binomial n p =
  match (n, p) with
    | _, 0 -> 1
    | 0, _ -> 0
    | _ -> n * (binomial (n - 1) (p - 1)) / p;;
```

Exercice 2-4

Par induction sur n. On considère l'ordre usuel sur $\mathbb{N}$, Pour l'élément minimal $n = 0$, $x^0 = 1$ par définition. Soit $n > 0$. D'après l'hypothèse d'induction, puisque $(n-1) < n$, `exp x (n - 1)` calcule x^{n-1}. Alors $x^n = x \times x^{n-1}$.

Exercice 5-1

```
let rotation p =
  let q = new () in
    while not is_empty p do
      push (pop p) q
    done;
    let temp = pop q in
      while not is_empty q do
        push (pop q) p;
      done;
      push temp p;;
```

Table des figures

1.1 Référence . . . 9
1.2 Vecteur . . . 9
1.3 Drapeau hollandais . . . 14

2.1 Ordre de la divisibilité sur $\mathbb{N} \setminus \{0,1\}$ . . . 21
2.2 Pointeur . . . 29
2.3 Liste . . . 29
2.4 Appels récursifs de fibo 5 . . . 30
2.5 Calcul de 3! . . . 31

3.1 tri sélection : les i premiers éléments sont déjà placés . . . 38
3.2 tri insertion : insérer t_k à sa place et décaler les éléments . . . 41
3.3 Un arbre de décision pour le tri bulle de 3 éléments . . . 44

4.1 Nombre de multiplications pour l'exponentiation rapide . . . 48
4.2 $T(n)$ et $S(n)$ . . . 49
4.3 Fusion linéaire de deux tableaux . . . 49
4.4 Tri fusion. Hauteur de l'arbre : $\log_2(n)$, nombre de comparaisons : $n \log_2(n)$ . . . 51
4.5 Idée du tri rapide . . . 52
4.6 Invariant de boucle du tri rapide . . . 53
4.7 Étape finale de la partition du tri rapide . . . 53

5.1 Représentation d'une pile LIFO et des opérations push et pop . . . 60
5.2 Une file d'attente . . . 61
5.3 Évaluation d'une EAP à l'aide d'une pile . . . 65

6.1 Un circuit électrique à k entrées et une sortie . . . 77
6.2 Portes logiques élémentaires . . . 78
6.3 Un afficheur LCD . . . 78

Index

Symboles

$O()$ 33
$\Theta()$ 33
+. 3
;; 3

B

begin 9
Blocs 9
Boucles
  for 10
  while 10

C

Chaînes
  concaténation 6
char 5
Circuits logiques 75
Complexité 33
  dans le cas le pire 33
  moyenne 33, 35
  ordres de grandeur 34
Correction d'une fonction récursive 21

D

Diviser pour régner 45

E

end 9
Exponentiation rapide 47
  complexité 47
Expression algébrique 64, 84
  postfixée 84
Expressions algébriques 60

F

float 5
Fonctions 7
  récursives 17
  récursives terminales 30

H

Hanoï 22

I

if 9
induction 21
Induction structurelle 24
Invariant de boucle 10

K

Knuth 56

L

let in 4
Listes 24
  image miroir 26
Logique
  proposition 66
Lois de Morgan 71

M

Multiplication
  de matrices 57
  de polynômes 56

O

Opérateurs logiques 10
Ordre bien fondé 19

P

Parenthèses 4
permutations
  inversions 39
  nombre moyen d'inversions 39
Piles 59, 64
Polynômes 56
Programmation
  fonctionnelle 14
  impérative 7

R

Récursivité
  preuve de correction 21
  terminaison 21
Références 8
Recherche
  dichotomique 36
  linéaire 35

S

Strassen (algorithme) 57

T

Tableaux 8
Tautologie 70
Terminaison d'une fonction récursive 21
Test 9
Tri 37
  borne inférieure 43

bulle 38
fusion 47
par insertion 40
par sélection 37
rapide 51
stabilité 37
Type 5
Type de données 58
Types
énumérés 82
contraintes 15
somme 83
somme récursifs 83

V

Vecteurs 8