

Lycée du Parc – MP2I

INFORMATIQUE

2021 – 2022

JEAN-BAPTISTE BIANQUIS
pro@bianquis.org

LISTE DES CHAPITRES

Cours	6
1 Introduction au langage OCaml	6
2 Aspects fonctionnels de OCaml	13
3 Aspects impératifs de OCaml	43
4 Correction, terminaison	63
5 Introduction à la complexité	75
6 Structures de données	92
7 Piles et files	101
8 Gestion de la mémoire	110
9 Représentation des données	136
10 Arbres	158
11 Dictionnaires	180
12 Tas et files de priorité	221
13 Familles d'algorithmes	235
14 Graphes : aspects théoriques	266
15 Algorithmes sur les graphes	295
16 Logique	333
Travaux pratiques	351
I Initiation à OCaml	351
II Manipulation de listes	361
III Tableaux, ordre supérieur	370
IV Tri insertion, tri fusion	379
V Analyse d'algorithmes	391
VI Exercices d'entraînement	400
VII Ensembles et combinatoire	404
VIII Écrivons des programmes !	414
IX Petits problèmes d'algorithmique	418

X	Piles, Files	422
XI	Introduction au langage C	427
XII	Boucles et tableaux statiques	441
XIII	Un peu de dessin	445
XIV	Pointeurs	463
XV	Pile d'appel, ensemble de Mandelbrot	469
XVI	Tableaux dynamiques	481
XVII	Exceptions en OCaml	495
XVIII	Programmation d'un allocateur mémoire	507
XIX	Tri radix	513
XX	Expressions arithmétiques	524
XXI	Promenade sylvestre	532
XXII	Structures de données chaînées en C	539
XXIII	Ce duc y parle	555
XXIV	Listes doublement chaînées	569
XXV	Listes à accès direct	575
XXVI	Arbres binaires de recherche en C	587
XXVII	Arbres binaires de recherche en OCaml	595
XXVIII	Arbres rouge-noir en OCaml	606
XXIX	Seam carving	614
XXX	Arbres de Braun	627
XXXI	Adressage ouvert	632
XXXII	Autour du tri rapide	646
XXXIII	Deux exemples de diviser pour régner	653
XXXIV	Programmation dynamique : le problème du sac à dos	662
XXXV	Maximisation d'une somme	665
XXXVI	Plus longue sous-séquence croissante	668
XXXVII	Graphes eulériens	678
XXXVIII	Parcours de graphes en OCaml	685
XXXIX	Clôture transitive	697
XL	Fichiers	705
XLI	Manipulation de formules logiques	712

LISTE DES CHAPITRES

XLII Autour de Dijkstra	718
Table des matières	730

Cours

INTRODUCTION AU LANGAGE OCAML

Présentation du langage OCaml

Le langage OCaml a été développé à l'INRIA¹ à partir des années 80. L'installation est assez simple si l'on travaille sous Linux ou sur un Mac, un peu plus délicate sous Windows : je vous donnerai rapidement accès à un environnement en ligne permettant de travailler directement dans votre navigateur Web.

Sans faire d'historique ni de typologie détaillée des langages de programmation, on peut donner rapidement quelques caractéristiques de OCaml. Il s'agit d'un langage :

- *compilé* : les fichiers source doivent d'abord être traduits en langage machine avant d'être exécutés, comme en C, C++, Java. Python, au contraire, est *interprété* : le code est (plus ou moins) lu ligne par ligne à l'exécution par un programme appelé *interpréteur*. OCaml est cependant doté d'une REPL (*Read Eval Print Loop*) qui permet de travailler comme vous en avez sans doute l'habitude en Python. Nous utiliserons les deux approches cette année.
- *typé statiquement* : une variable ne peut pas changer de type au cours de l'exécution. Python est typé dynamiquement, ce qui est assez caractéristique des langages de script (Python, Ruby, Perl, JavaScript...) ; la plupart des langages compilés (C, Java, C++) sont typés statiquement, avec quelques subtilités ;
- *fortement typé*, c'est-à-dire que toutes les erreurs de type seront détectées à la compilation. En Python, ces erreurs seront détectées à l'exécution ; en C, elles ne seront pas forcément détectées.
- *fonctionnel*, c'est-à-dire que les fonctions sont des valeurs comme les autres, qui peuvent être renvoyées comme résultat d'un calcul, prises en argument... C'est également possible en Python, mais c'est moins pratique et moins « idiomatique ». Un langage fonctionnel met également l'accent sur la récursion plus que sur l'itération et privilégie les valeurs non mutables (essentiellement, fonctions et variables sont utilisées d'une manière proche de ce que l'on fait en mathématiques et pas comme des cases mémoire dans lesquelles on écrit à notre gré). Python n'est pas du tout fonctionnel en ce sens là, les exemples classiques sont plutôt LISP (dans une certaine mesure), Standard ML (très proche de OCaml) et Haskell.
- *impur* : la mutation (modification de la valeur d'une variable) est quand même possible dans certains cas. Haskell est l'exemple le plus connu de langage (presque) pur. Notez quand même qu'en comparaison de Python, OCaml est une vestale...
- *strict* : les arguments d'une fonction sont évalués avant de faire l'appel. Si vous n'avez jamais fait de Haskell (ni de λ -calcul), vous n'avez sûrement jamais réalisé que l'on pouvait faire autrement... Haskell (et son ancêtre Miranda) sont les seuls langages un tant soit peu connus à être *paresseux* (le contraire de strict dans ce contexte).
- *orienté objet*, ou plutôt « doté d'une système d'objets dont quasiment personne ne se sert » : c'est de là que vient le « O » de OCaml. Dans la plupart des langages actuellement populaires, les objets jouent un rôle important (Python, C++, C#, Java...), dans les autres, la notion n'existe pas (C, Haskell).

OCaml est un langage « presque confidentiel mais pas tout-à-fait » : il est principalement utilisé dans l'univers de la recherche et pour créer des outils manipulant des programmes éventuellement écrits dans un autre langage (compilateurs, analyse statique, génération automatique de code...). On peut citer deux variantes assez populaires de OCaml :

- F#, qui a été créé par Microsoft et fait partie de l'environnement .Net : il y a quelques différences par rapport à OCaml mais il est immédiat d'apprendre l'un si l'on connaît l'autre ;

1. Institut National de Recherche en Informatique et Automatique, un établissement public de recherche français comme le CNRS, l'INSERM...

- Reason, créé assez récemment par Facebook. Ici, on ne peut même pas vraiment parler de variante : le langage est exactement le même (et utilise le même compilateur), seule la syntaxe a été modifiée de manière à être moins déroutante pour des développeurs habitués à JavaScript ou PHP (qui sont les langages les plus utilisés chez Facebook).

Plusieurs langages et *frameworks* populaires ont été assez nettement influencés par OCaml (et par les idées de la programmation fonctionnelle plus généralement). On peut citer Rust (qui vise à fournir une alternative plus sûre à C/C++) et React (le *framework* JavaScript le plus populaire aujourd’hui, créé par l’auteur de ReasonML).

I Opérateurs de base

Les types les plus simples en OCaml sont **int** (type des entiers), **float** (type des nombres à virgule flottante, l’« équivalent » informatique des réels), **bool** (les booléens, c’est-à-dire **true** et **false**), **char** (type des caractères) et **string** (type des chaînes de caractères).

On dispose des opérateurs arithmétiques usuels sur les entiers :

```
# 3 + 4;;
- : int = 7
# 12 - 3 * 7;;
- : int = -9
# 5 / 2 ;;
- : int = 2
# 17 mod 5;;
- : int = 2
```

et sur les flottants :

```
# 3.1 +. 5.4;;
- : float = 8.5
# 5. /. 2. ;;
- : float = 2.5
# 2. ** 8. ;; (* puissance, défini uniquement sur les flottants *)
- : float = 256.
```

Il faut bien remarquer qu’il y a deux variantes distinctes des opérateurs, l’une pour les entiers et l’autre pour les flottants. Si l’on essaie de mélanger les deux types, on obtiendra une erreur :

```
# 2.5 + 3;;
Characters 0-3:
2.5 + 3;;
^ ^
Error: This expression has type float but an expression was expected of type
int
# 2.5 +. 3;;
Characters 7-8:
2.5 +. 3;;
^ ^
Error: This expression has type int but an expression was expected of type
float
```

Les opérateurs de comparaison sont, eux, *polymorphes* : on utilise le même opérateur pour comparer deux entiers, deux flottants...²

2. mais pas un entier et un flottant.

```
# 2 < 3;;
- : bool = true
# 3.4 >= 2.5;;
- : bool = true
# (4, 8) = (4, 9);;
- : bool = false
# (4, 8) <> (4, 9);;
- : bool = true
```

Sur les booléens, les opérations de base sont le « et », le « ou » et la négation :

```
# true && false;; (* "et" logique *)
- : bool = false
# true || false;; (* "ou" logique *)
- : bool = true
# not false;; (* négation *)
- : bool = true
```

Tout texte compris entre (* et *) est ignoré : on parle de *commentaires*.

2 Variables

On peut définir des variables *globales* (qui existent dans l'ensemble du programme) :

```
# let x = 3;;
val x : int = 3
# x + x;;
- : int = 6
# let y = x;;
val y : int = 3
# let x = 4;;
val x : int = 4
# y;;
- : int = 3
```

et également des variables *locales* (dont la portée est limitée) :

```
# let a = 4 in a * a + 1;;
- : int = 17
# a;;
Characters 0-1:
a;;
^
Error: Unbound value a
```

Une variable locale peut *masquer* une variable globale :

```
# let a = 5;;
val a : int = 5
# let a = 4 in a * a;;
- : int = 16
# a;;
- : int = 5
```

Très souvent, on enchaînera les définitions de variables locales (mais normalement ce sera à l'intérieur d'une définition de fonction, pas dans la boucle interactive) :

```
# let a = 4 in let b = 7 in a * b;;
- : int = 28
# let a = 4 in let b = a * a in a + b;;
- : int = 20
```

En OCaml, une variable « normale » ne *change jamais de valeur* une fois définie (on dit qu'elle n'est *pas mutable*). C'est assez différent de ce qui se passe dans la plupart des langages, mais très similaire à la notion mathématique de variable³. Nous aurons l'occasion de revenir largement sur ce point dans la suite du cours.

3 Définitions de fonctions

Un exemple minimal :

```
# let carre x = x * x;;
val carre : int -> int = <fun>
```

Comme l'opérateur * ne marche que sur les entiers, OCaml sait que x doit être un entier, et comme le résultat de la multiplication de deux entiers est un entier, OCaml sait que le résultat sera un entier. Il en déduit le type de la fonction : `int -> int`. On dit que OCaml a *inféré le type* de la fonction `carre`.

On notera immédiatement une différence importante avec Python : il n'y a pas de `return`. En OCaml, tout est une expression, et en particulier la valeur de retour d'une fonction est simplement la valeur de l'expression définissant la fonction une fois que l'on a substitué les valeurs concrètes des arguments formels.

En OCaml, l'application de la fonction f à l'argument x se note simplement `f x` (et non `f(x)` comme en mathématiques). Si l'on veut appeler cette fonction, il suffit donc de taper

```
# carre 10;;
- : int = 100
```

Attention aux priorités :

```
# carre 3 + 5;;
- : int = 14
# carre (3 + 5);;
- : int = 64
```

En OCaml, l'application de fonction est prioritaire sur tout le reste : `f x y` signifie « `f(x)`, le tout appliqué à `y` ».

Quand une fonction prend plusieurs arguments, la technique « naturelle » en OCaml est d'en écrire une version *curryfiée*⁴ :

```
# let ajoute x y = x + y;;
val ajoute : int -> int -> int = <fun>
```

Si l'on veut ajouter le carré de 4 et celui de $2+3$ (en utilisant les fonctions que nous venons de définir), il faut alors écrire :

```
# ajoute (carre 4) (carre (2 + 3));;
- : int = 41
```

On peut définir des variables locales à l'intérieur d'une fonction (c'est même le principal intérêt des variables locales...) :

3. « Let $x = \dots$ » est la traduction anglaise de « Soit $x = \dots$ », et ce choix n'a pas été fait par hasard.

4. rien à voir avec les épices : c'est en référence à Haskell Curry, logicien (qui a également donné son nom au langage Haskell).

```
# let f a b =
  let x = carre (a + b) in
  let y = carre (a - b) in
  carre (x + y) - carre (x - y);;
val f : int -> int -> int = <fun>
# f 3 7;;
- : int = 6400
```

4 Fonctions récursives

En informatique, une fonction est dite *récursive* si elle s'appelle elle-même. Il y a beaucoup de choses à dire sur la récursivité, et nous y reviendrons longuement, mais pour aujourd'hui on peut se contenter de quelques exemples très simples.

On rappelle la définition classique de la factorielle :

$$0! = 1 \quad \text{et} \quad \forall n \in \mathbb{N}, (n+1)! = (n+1) \times n!$$

En mathématiques, on dirait qu'il s'agit d'une définition *par récurrence*; en informatique, on parlera plutôt de *définition inductive*, voire *récursive*.

Pour changer temporairement de langage, il y a en Python deux manières naturelles d'écrire une fonction « factorielle » :

```
#####
# Code en Python !!! #
#####

def fact_iter(n):
    f = 1
    for k in range(1, n + 1):
        f = f * k
    return f

def fact_rec(n):
    if n == 0:
        return 1
    return n * fact_rec(n - 1)

#####
# Fin du code en Python #
#####
```

La première version est dite *itérative*: elle utilise une boucle (et ce qui serait en OCaml une variable *mutable* f). La deuxième version, très proche de la définition mathématique, est dite récursive.

Ces deux versions peuvent être traduites en OCaml, mais la première, qui utilise les aspects impératifs de OCaml, ne nous intéresse pas pour l'instant.

```
(* Vous pouvez sauter cette partie pour l'instant. *)
let fact_iter n =
  let f = ref 1 in
  for k = 1 to n do
    f := !f * k
  done;
  !f
(* Fin de la partie à ignorer. *)

let rec fact_rec n =
  if n = 0 then 1
  else n * fact_rec (n - 1)
```

On notera qu'en OCaml, une définition récursive utilise **let rec** au lieu de **let**.

5 Listes

Le type **list** est un type prédéfini en OCaml. Il permet de représenter des listes de valeurs de type arbitraire mais homogène. Cela signifie que l'on peut définir une liste d'entiers ou une liste de booléens, mais pas une liste contenant à la fois des entiers et des booléens.

Attention : le type **list** de OCaml est très différent du type **list** de Python. Ce que l'on appelle liste en Python ressemble plutôt à ce que l'on appellera tableau en OCaml (et dans la plupart des langages).

On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points-virgules :

```
# let l = [];
val l : 'a list = []
# let m = [1];
val m : int list = [1]
# let s = [3; 5; 0];
val s : int list = [3; 5; 0]
```

Étant donnée une liste, on peut en construire une nouvelle en ajoutant un élément en tête :

```
# let k = 3 :: l ;;
val k : int list = [3]
# let j = 7 :: k ;;
val j : int list = [7; 3]
```

Tous les éléments d'une liste doivent avoir le même type (uniquement des entiers, ou alors uniquement des flottants, ou alors...):

```
# let w = [3.2; 5.1];
val w : float list = [3.2; 5.1]
# let t = [3.2 ; 5];
Characters 15-16:
let t = [3.2 ; 5];
^
Error: This expression has type int but an expression was expected of type
float
```

Essentiellement, il y a deux types de listes : la liste vide, et les autres. Une liste non vide est constituée d'un premier élément, appelé *tête* ou *head* et de la liste (éventuellement vide) de ses autres éléments, appelée *queue* ou *tail*. Le *pattern matching* (qu'on traduit parfois en français par *filtrage*, ou *filtrage par motif*) est la manière standard d'opérer sur des listes :

```
let rec somme l =
  match l with
  | [] -> 0
  | x :: xs -> x + somme xs
```

On voit ici le schéma fondamental de filtrage sur les listes : soit la liste est vide (c'est ici, comme souvent, le cas de base de la récursion), soit elle s'écrit `head :: tail`, avec `head` un élément de type '`a`' et `tail` une liste de type '`a list`'.

Il arrive quand même parfois que ces deux cas ne suffisent pas. Ici, on souhaite calculer le maximum des éléments d'une liste, en utilisant la fonction prédéfinie `max` (qui permet d'obtenir le maximum de deux nombres).

```
let rec max_liste l =
  match l with
  | [] -> failwith "liste vide"
  | x :: [] -> x
  | x :: xs -> max x (max_liste xs)
```

Dans ce filtrage, les motifs ne sont pas mutuellement exclusifs : une liste de la forme `x :: []` est également de la forme `x :: xs` avec `xs = []`. Cependant, les motifs sont essayés dans l'ordre : ainsi, une liste à un seul élément sera acceptée par la deuxième clause et n'arrivera pas jusqu'à la troisième. En revanche, une liste contenant au moins deux éléments sera refusée par les deux premiers motifs et sera donc traitée par la troisième clause.

Remarque

Le motif `x :: []` peut être remplacé par `[x]`, ce que l'on fera systématiquement (plus concis, plus lisible).

ASPECTS FONCTIONNELS DE OCAML

I Types de base

I.I Types numériques

I.I.a Type int

Le type **int** est celui des entiers (*integer* en anglais).

Opérateur	Signification	Exemples
+	Addition d'entiers	$2 + 3 = 5$
*	Multiplication d'entiers	$2 * 3 = 6$
-	Soustraction / opposé	$3 - 7 = -4$ ou $-(2 + 3) = -5$
/	« Quotient » de la division euclidienne	$17 / 5 = 3$ $(-17) / 5 = -3$
mod	« Reste » de la division euclidienne	$17 \text{ mod } 5 = 2$ $-17 \text{ mod } 5 = -2$
abs	Valeur absolue	$\text{abs } (-2) = 2$

TABLE 2.1 – Opérations sur les entiers

Remarque

Pour la division, a / b fait un arrondi vers 0, et ne correspond donc à la division euclidienne mathématique que si a et b sont de même signe. $a \text{ mod } b$ est défini de telle manière que l'identité $a = (a / b) * b + (a \text{ mod } b)$ soit systématiquement vérifiée, ce qui signifie que $a \text{ mod } b$ est négatif si a est négatif.

La plupart de ces opérateurs sont *infixes* (il faut les placer entre leurs deux arguments). Cependant, l'addition par exemple est en fait une fonction de type **int** → **int** → **int**. On peut y accéder en plaçant l'opérateur entre parenthèses :

```
# (+);;
- : int -> int -> int = <fun>
# (+) 12 23;;
- : int = 35
```

Les entiers sont représentés sur 32 ou 64 bits (suivant la version de OCaml), mais OCaml utilise un de ces bits comme un « drapeau » : les entiers représentables varient donc entre -2^{30} et $2^{30} - 1$ (sur une machine 32 bits) ou -2^{62} et $2^{62} - 1$ (sur une machine 64 bits). Ces valeurs sont stockées dans les constantes **min_int** et **max_int** (ici sur une machine 64 bits) :

```
# min_int;;
- : int = -4611686018427387904
# max_int;;
- : int = 4611686018427387903
# 1 lsl 62 - 1 (* 2^62 - 1 *);;
- : int = 4611686018427387903
```

Tous les calculs sont effectués modulo 2^{31} ou 2^{63} suivant les cas :

```
# max_int + 1 = min_int;;
- : bool = true
```

Remarque

Nous reviendrons plus en détail sur ces points quant nous traiterons la représentation des entiers.

I.I.b Type float

Le type des nombres dits *à virgule flottante* (ou flottants pour faire simple), représentés sur 64 bits. La représentation interne d'un flottant et toutes les questions liées à la précision des calculs seront traitées ultérieurement.

Opérateur	Signification	Exemples
+. . .	Addition de flottants	2.1 +. 3. = 5.1
*. . .	Multiplication de flottants	2.1 *. 3. = 6.2
-. . .	Soustraction / opposé	3. -. 7.1 = -4.1 -.(2. +. 3.2) = -5.2
/. . .	Division de flottants	17. /. 5. = 3.4
** . . .	Puissance	2.3 ** 4.1 = 30.41...
exp . . .	Exponentielle	exp 1. = 2.718...
log . . .	Logarithme népérien	log 2. = 0.69...
floor . . .	« Partie entière » ¹	floor (-2.3) = -3.
ceil . . .	Arrondi par excès	ceil 3.1 = 4.
abs_float . . .	Valeur absolue	abs_float (-3.2) = 3.2

TABLE 2.2 – Opérations sur les flottants

Les fonctions trigonométriques usuelles (directes, inverses, hyperboliques) sont aussi fournies.

```
# asin (cos 0.);;
- : float = 1.57079632679489656
(* arcsin(cos(0)) = pi / 2 *)
```

I.I.c Coercitions

Contrairement à la plupart des langages, OCaml ne fait jamais de conversion automatique des entiers vers les flottants (ni évidemment des flottants vers les entiers). Les fonctions de conversion de type, aussi appelées *coercitions*, sont :

Fonction	Type	Comportement	Exemples
float_of_int	int -> float	Injection canonique	float_of_int 4 = 4.
int_of_float	float -> int	Arrondi vers 0	int_of_float 3.7 = 3 int_of_float (-3.7) = -3

TABLE 2.3 – Conversion entiers ↔ flottants

Remarque

Comme `float_of_int` sert relativement souvent, elle a un *alias* plus court : `float : int -> float`.

1. Attention, le résultat est un flottant!

1.2 Booléens

Le type booléen ne contient que deux valeurs : `true` et `false`.

Opérateur	Signification	Exemples
<code>not</code>	Négation	<code>not true = false</code>
<code>&&</code>	« Et » séquentiel	<code>true && false = false</code>
<code> </code>	« Ou » séquentiel	<code>true false = true</code>

TABLE 2.4 – Opérateurs booléens

Remarque

Les opérateurs `&&` et `||` sont *séquentiels*, ce qui signifie qu'ils n'évaluent leur second argument que si nécessaire. Ainsi, `true || x` renvoie `true` immédiatement sans évaluer l'expression `x`, ce qui peut être important si déterminer la valeur de `x` demande un calcul long, et crucial si le calcul de `x` ne termine pas ou résulte en une erreur :

```
# (2 < 3) || (1 / 0 = 0);;
- : bool = true
# (2 < 3) && (1 / 0 = 0);;
Exception: Division_by_zero.
# (1 / 0 = 0) || (2 <> 3);;
Exception: Division_by_zero.
```

```
# (2 > 3) || (1 / 0 = 0);;
Exception: Division_by_zero.
# (2 > 3) && (1 / 0 = 0);;
- : bool = false
```

On dispose également d'opérateurs de comparaison polymorphes (qui permettent de comparer deux valeurs d'un même type, quel que soit ce type) :

Opérateur	Signification	Exemples
<code>=</code>	Égalité structurelle	<code>[[3]; [1; 2]] = [[3]; [1; 2]]</code> renvoie <code>true</code>
<code><></code>	Différence structurelle	<code>[[3]; [1; 2]] <> [[3]; [1; 4]]</code> renvoie <code>true</code>
<code><</code>	Inférieur strict	<code>[[3]; [1; 2]] < [[3]; [1; 4]]</code> donne <code>true</code>
<code><=</code>	Inférieur ou égal	<code>[[3]; [1; 2]] <= [[4]; [1; 1]]</code> donne <code>true</code>

TABLE 2.5 – Comparaisons

Tester l'égalité structurelle de deux valeurs simples (entiers, flottants...) se fait en temps constant. Ce n'est bien sûr plus le cas si, par exemple, on compare deux listes : on peut avoir à parcourir toute la structure.

Attention : les opérateurs `==` et `!=` existent également, mais leur sens est différent (égalité *physique*, c'est-à-dire égalité des pointeurs). Nous ne nous en servirons pour ainsi dire jamais.

1.3 Caractères et chaînes de caractères

En OCaml, les types caractère (`char`) et chaîne de caractère (`string`) sont séparés. Une chaîne de caractère se comporte essentiellement comme un tableau de `char` (sauf qu'une chaîne n'est pas mutable). Pour définir une constante littérale de type `char`, on mettra le caractère entre apostrophes, et pour une constante de type `string` entre guillemets.

```
# 'a';
- : char = 'a'
# "a";
- : string = "a"
```

```
# "abc";
- : string = "abc"
```

Les fonctions d'affichage sont par essence impératives, mais si vous en avez besoin :

Fonction	Signification	Exemples
<code>print_string</code>	Afficher une chaîne	<code>print_string "abc"</code> affiche "abc"
<code>print_int</code>	Afficher un entier	<code>print_int (2 + 3)</code> affiche "5"
<code>print_float</code>	Affiche un flottant	<code>print_float (acos (-1.))</code> affiche 3.14159265359

TABLE 2.6 – Fonctions d'affichage

On dispose également de `int_of_string`, `string_of_int`, `float_of_string` et `string_of_float` :

```
# float_of_string "2.3e5";;
- : float = 230000.
# string_of_float (2.**10.);;
- : string = "1024."
# int_of_string "12.34";;
Exception: Failure "int_of_string".
```

1.4 Listes

Nous avons déjà utilisé le type `list`, et nous reviendrons dessus de manière plus poussée ultérieurement. Voilà quand même une table de référence pour quelques fonctions usuelles.

Fonction	Type	Signification
<code>@</code> (infixe)	<code>'a list -> 'a list -> 'a list</code>	Concaténation
<code>List.hd</code>	<code>'a list -> 'a</code>	Renvoie l'élément de tête
<code>List.tl</code>	<code>'a list -> 'a list</code>	Renvoie la queue
<code>List.rev</code>	<code>'a list -> 'a list</code>	Renvoie la liste « miroir »
<code>List.length</code>	<code>'a list -> int</code>	Renvoie la longueur de la liste
<code>List.map</code>	<code>('a -> 'b) -> 'a list -> 'b list</code>	cf TD
<code>List.fold_left</code>	<code>('a -> 'b -> 'a) -> 'a -> 'b list -> 'a</code>	cf TD
<code>List.fold_right</code>	<code>('a -> 'b -> 'b) -> 'a list -> 'b -> 'b</code>	cf TD
<code>List.iter</code>	<code>('a -> unit) -> 'a list -> unit</code>	cf TD

TABLE 2.7 – Fonctions prédéfinies sur les listes

Remarques

- Nous n'avons pas mentionné `::` car il s'agit d'un constructeur et non d'une fonction. C'est cependant (et de loin) ce qu'on utilise le plus souvent.
- Notez bien que les fonctions `List.hd` et `List.tl` sont très rarement utiles : on préférera le pattern matching.
- Attention, la fonction `List.length` a une complexité linéaire ; nous y reviendrons.

1.5 Type `unit`

Le type `unit` est (essentiellement) celui des *actions*. Il n'y a qu'une valeur de ce type : `()`. Nous reviendrons en détail sur ce sujet lorsque nous évoquerons les aspects impératifs de OCaml.

2 Types structurés

2.1 Types produit

2.1.a Couples

Si '`a`' et '`b`' sont des types, alors '`a * b`' est un type, dit *produit* de '`a`' et de '`b`'. C'est le type des couples dont la première composante est de type '`a`' et la deuxième de type '`b`'.

```
# let a = (1.2, 2.5);;
val a : float * float = (1.2, 2.5)
# let b = (2, [1; 7]);;
val b : int * int list = (2, [1; 7])
```

Deux fonctions sont prédefinies pour accéder aux composantes :

- `fst` : '`a * b` -> '`a`
- `snd` : '`a * b` -> '`b`

```
# fst a;;
- : float = 1.2
# snd b;;
- : int list = [1; 7]
```

Le pattern matching marche comme on peut s'y attendre : les fonctions `norme` et `norme2` définies ci-dessous sont parfaitement équivalentes.

```
let norme vecteur = match vecteur with
| (x,y) -> sqrt (x**2. +. y**2.)

let norme2 vecteur =
  sqrt ((fst vecteur)**2. +. (snd vecteur)**2.)
```

Cependant, il est possible (et plus agréable ici) de le faire directement sur les arguments :

```
let norme3 (x, y) = sqrt (x**2. +. y**2.)
```

De manière plus générale, on peut très facilement « déconstruire » un couple pour récupérer ses composantes :

```
# let x, y = a;;
val x : float = 1.2
val y : float = 2.5
```

2.1.b tuples

Les produits de plus de deux types fonctionnent exactement de la même manière que les couples, à ceci près qu'on ne dispose pas de fonctions prédefinies pour accéder aux différents éléments : il faut utiliser le pattern matching (éventuellement implicite).

```
# let somme_triplet (x, y, z) = x + y + z;;
val fst_triplet : 'a * 'b * 'c -> 'a = <fun>
# let a, b, c = 2.3, 5, [];;
val a : float = 2.3
val b : int = 5
val c : 'a list = []
```

2.2 Types flèches

Si `'a` et `'b` sont des types, `'a -> 'b` est le type des fonctions qui prennent un argument de type `'a` et renvoient un argument de type `'b`.

2.2.a Curryfication

Les types `'a -> ('b -> 'c)` et `('a -> 'b) -> 'c` sont complètement différents :

- dans le cas `'a -> ('b -> 'c)`, on prend en argument un objet de type `'a` et l'on renvoie une fonction de type `'b -> 'c` en résultat;
- dans le cas `('a -> 'b) -> 'c`, on prend en argument une fonction de type `'a -> 'b` et l'on renvoie un objet de type `'c` comme résultat.

Comme nous allons le voir, le cas `'a -> ('b -> 'c)` est beaucoup plus fréquent que l'autre en OCaml. Pour limiter le nombre de parenthèses, on choisit donc la convention suivante :

$$'a -> 'b -> 'c \stackrel{\text{def}}{=} 'a -> ('b -> 'c)$$

Pourquoi est-il si fréquent de rencontrer des fonctions de type `'a -> 'b -> 'c`? Parce que l'on peut voir une telle fonction comme une fonction de *deux paramètres*, qui prend ces paramètres *l'un après l'autre*.

Exemple 2.1

Considérons la fonction suivante :

```
let rec puissance x n =
  if n = 0 then 1.
  else x *. puissance x (n - 1)
```

Cette fonction prend en entrée un flottant `x` et un entier `n` supposé positif ou nul, et renvoie le flottant x^n . On la voit donc naturellement comme une fonction de deux variables, et on l'utilisera le plus souvent comme cela :

```
# puissance 2.5 4;;
- : float = 39.0625
```

Pourtant son type est :

```
puissance : float -> int -> float
```

En réalité, c'est donc une fonction qui prend en entrée un flottant et qui renvoie une fonction. Par exemple, `puissance 2.5` est un objet bien défini : il s'agit de la fonction qui à un entier `n` associe `puissance 2.5 n`:

```
# let appli_partielle = puissance 2.5;;
val appli_partielle : int -> float = <fun>
# appli_partielle 4;;
- : float = 39.0625
# appli_partielle 1;;
- : float = 2.5
```

C'est le principe de la *curryfication* (en l'honneur du logicien Haskell Curry) qui est la manière standard de définir des fonctions de plusieurs arguments en OCaml.

2.2.b Définition de fonctions

Pour définir une fonction en OCaml, on dispose de trois possibilités :

```
let f = (fun x_1 ... x_n -> expression)

let f = function
| pattern_1 -> expression_1
| pattern_2 -> expression_2
...
| pattern_n -> expression_n

let f x_1 ... x_n = expression
```

La première est utile pour définir des fonctions *anonymes* (donc sans la partie **let** f =) que l'on utilise immédiatement (souvent comme argument à une fonction d'ordre supérieur) :

```
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let f = compose (fun x -> x * x) (fun x -> x + 1);;
val f : int -> int = <fun>

(* On vient de définir la composée de la fonction carré avec
   la fonction  $x \mapsto x+1$  sans avoir à donner de nom à ces
   fonctions *)

# f 4;;
- : int = 25
```

La deuxième permet de gagner quelques caractères quand on définit une fonction d'un seul argument qui commence immédiatement par un **match** sur l'argument, mais on s'en passe très bien :

```
let rec longueur = function
| [] -> 0
| x :: xs -> 1 + longueur xs

(* parfaitement équivalent à : *)

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

La troisième forme est celle que nous utiliserons le plus souvent. Remarquez qu'une fonction récursive doit nécessairement être définie à l'aide de **let rec**.

2.2.c Parenthésage

S'habituer au parenthésage de OCaml peut prendre un peu de temps, mais les règles sont en fait très logiques. Considérons les quelques exemples suivants :

```
(* 1. *) f x
(* 2. *) f (x y)
(* 3. *) f x y
(* 4. *) f (u x) (v y)
(* 5. *) f (u x) (v (t1 t2) w)
```

Dans chaque cas, la première chose à faire est de compter le nombre de « paquets » auxquels on applique la fonction f. Chacun de ces paquets doit être vu comme un argument que l'on passe à f.

1. C'est le cas le plus simple : on applique f à x.

```
# sqrt 2. ;;
- : float = 1.41421356237309515
```

Bien sûr, si f attend plus d'un argument, le résultat peut encore être une fonction (il s'agit alors d'une *application partielle*) :

```
# max 32 ;;
- : int -> int = <fun>
# let g = max 32;;
val g : int -> int = <fun>
# g 12;;
- : int = 32 (* car max 32 12 = 32 *)
# g 45;;
- : int = 45 (* car max 32 45 = 45 *)
```

2. On ne passe toujours qu'un seul argument à f. Cet argument, ($x \ y$), est lui-même le résultat de l'application de x à y : x doit donc être une fonction. On demande tout simplement de calculer $(f \circ x)(y) = f(x(y))$. Un exemple concret simple :

```
# log (exp 12.);;
- : float = 12.
```

3. On passe deux arguments x et y à f :

```
# max 12 34;;
- : int = 34
```

4. On passe encore deux arguments à f : ($u \ x$) et ($v \ y$). u doit nécessairement être une fonction (d'au moins un argument), tout comme v.

```
(* renverse la liste, puis ajoute 5 à chaque élément *)
# List.map ((+) 5) (List.rev [1; 2; 3; 4]);
- : int list = [9; 8; 7; 6]
```

5. On passe toujours deux arguments à f (ce sont ($u \ x$) et ($v \ (t1 \ t2) \ w$)), un argument à u, un argument à t1 et deux arguments à v.

```
# max (log 2.) (min (exp 1.) 1.5);;
- : float = 1.5
(* max(ln(2), min(exp(1), 1.5)) = max(ln(2), 1.5) = 1.5 *)
```

3 Filtrage

Le *filtrage par motif*, que l'on appelle plus couramment *pattern matching*, fournit une syntaxe extrêmement pratique pour effectuer une disjonction de cas sur la **forme** d'une expression. Cette technique est utilisée constamment lorsqu'on programme en OCaml, en particulier lorsqu'on opère sur des types sommes (voir plus loin).

3.1 Forme générale d'un **match**

```
match expr with
| filtre_1 -> valeur_1
| ...
| filtre_k -> valeur_k
```

- **expr** est une expression quelconque, d'un certain type '**a**'. En pratique, ce sera le plus souvent quelque chose de simple :
 - soit une variable : **match x with**;
 - soit un couple (ou triplet, ou...) de variables : **match x, y with**;
 - soit le résultat d'une application de fonction : **match f x with**;
 - soit parfois une combinaison des cas précédents : **match x, f y, g y with**.
- **filtre_1, ..., filtre_k** suivent la syntaxe des filtres définie ci-dessous, et doivent bien sûr tous correspondre au type '**a**' de **expr**.
- **valeur_1, ..., valeur_k** sont des expressions (quelconques) d'un même type '**b**'. L'expression **valeur_i** peut faire intervenir les variables présentes dans **filtre_i**.
- L'expression complète **match expr with | filtre_1 -> ... | filtre_k -> valeur_k** est de type '**b**' (le type commun à **valeur_1, ..., valeur_k**).

Les filtres sont essayés un par un, dans l'ordre, et le premier qui accepte l'expression **expr** est utilisé : à ce moment, on « passe à droite de la flèche », et la valeur du **match** est obtenue en remplaçant dans **valeur_i** les variables du filtre par ce qui a été capturé.

3.2 Forme des filtres

Un filtre peut faire intervenir :

- des *constructeurs de type*. Nous verrons plus bas de quoi il s'agit, mais visuellement c'est très simple : c'est quelque chose qui commence par une majuscule (plus les cas particuliers du **:** pour les listes et de la virgule pour les tuples).
- des *constantes littérales* comme **1, 72, 3.14, "z", true, [], "toto", [4; 8; 6]**;
- des *variables*, qui seront **systématiquement considérées comme fraîches**. Autrement dit, si le filtre contient une variable **x** et qu'il existe déjà une variable portant ce nom, **ces deux variables n'ont rien à voir entre elles**. Attention, un filtre est nécessairement *linéaire*, ce qui signifie qu'une même variable ne peut apparaître qu'une seule fois dans un filtre donné.
- des **_ (underscore)**, qui acceptent tout (comme des variables) mais ne capturent rien (et ne peuvent donc être utilisés à droite de la flèche).

3.3 Test d'exhaustivité

Le compilateur émet systématiquement un *warning* lorsqu'il n'est pas capable d'assurer qu'un **match** est exhaustif : c'est une fonctionnalité extrêmement puissante, qui permet d'éliminer un nombre incalculable d'erreurs dans les programmes. Il peut arriver que le cas « oublié » soit en fait un cas d'erreur, pour lequel aucune réponse n'est valable (maximum d'une liste vide, par exemple). Dans ce cas, il est conseillé de rajouter un cas d'erreur explicite, en utilisant **failwith**.

```
let rec max_liste u =
  match u with
  | [] -> failwith "max non défini"
  | [x] -> x
  | x :: xs -> max x (max_liste xs)
```

3.4 Clause gardée

On peut rajouter un *garde* à un filtre, avec la syntaxe `filtre when condition -> valeur`. Il faut que condition soit un booléen : s'il s'évalue en `true` (et si le filtre accepte l'expression de départ), alors on « passe à droite de la flèche », sinon on passe au filtre suivant.

Remarque

On peut toujours éviter l'utilisation de `when` à l'aide de `if...then...else` à droite de la flèche, mais cela donne parfois un code moins clair. Les deux codes suivants sont presque équivalents :

```
match expr with
| filtre when condition -> valeur
| filtre when not condition -> valeur'
```

```
match expr with
| filtre ->
  if condition then valeur
  else valeur'
```

La seule différence est que OCaml ne peut pas garantir l'exhaustivité si toutes les clauses correspondant à un motif sont gardées. Ainsi, le code suivant générera un *warning* :

```
let rec nb_pos u =
  match u with
  | [] -> 0
  | x :: xs when x >= 0 -> 1 + nb_pos xs
  | x :: xs when x < 0 -> nb_pos xs
```

On préférera dans ce cas utiliser l'une des deux solutions suivantes :

```
let rec nb_pos u =
  match u with
  | [] -> 0
  | x :: xs when x >= 0 -> 1 + nb_pos xs
  | x :: xs (* x < 0 ici *) -> nb_pos xs

let rec nb_pos u =
  match u with
  | [] -> 0
  | x :: xs ->
    if x >= 0 then 1 + nb_pos xs
    else nb_pos xs
```

3.5 Clause multiple

On peut regrouper plusieurs clauses en une :

```
match expr with
| filtre_1 | filtre_2 -> ...
| filtre_3 -> ...
| ...
```

Attention, dans ce cas, toutes les clauses regroupées doivent contenir exactement les mêmes variables.

3.6 match avec un seul cas

Il est possible d'écrire un **match** avec une seule clause, mais dans ce cas on préférera systématiquement le remplacer par un simple **let**. Par exemple, le code suivant :

```
match couple with
| a, b -> a + b
```

est parfaitement équivalent à :

```
let a, b = couple in a + b
```

On privilégiera la deuxième solution.

3.7 Exemples et erreurs à éviter

Exemple 2.2

- La fonction suivante prend en entrée deux listes $[x_0; \dots; x_n]$ et $[y_0; \dots; y_n]$, de même type et de même longueur, et renvoie $[\max x_0 y_0; \dots; \max x_n y_n]$:

```
(* 'a list -> 'a list -> 'a list *)
let rec max2 u v =
  match u, v with
  | [], [] -> []
  | x :: xs, y :: ys -> (max x y) :: max2 xs ys
  | _ -> failwith "longueurs différentes"
```

- La fonction suivante fait la même chose mais ne suppose plus que les listes sont de même longueur. Lorsqu'une liste est épuisée, on prend simplement les éléments de l'autre.

```
(* 'a list -> 'a list -> 'a list *)
let rec max2_bis u v =
  match u, v with
  | [], [] -> []
  | x :: xs, [] -> x :: max2_bis xs v
  | [], y :: ys -> y :: max2_bis u ys
  | x :: xs, y :: ys -> (max x y) :: max2_bis xs ys
```

- Les fonctions suivantes sont d'autres versions de `max2_bis` (exactement la même spécification), avec un code un peu plus concis.

```
let rec max2_bis u v =
  match u, v with
  | [], _ -> v
  | _, [] -> u
  | x :: xs, y :: ys -> (max x y) :: max2_bis xs ys

let rec max2_bis u v =
  match u, v with
  | [], w | w, [] -> w
  | x :: xs, y :: ys -> (max x y) :: max2_bis xs ys
```

Exemple 2.3

Il est parfois nécessaire d'imbriquer plusieurs **match**; dans ce cas, il faut délimiter le **match** interne par un **begin...end**. Par exemple, la fonction suivante prend en argument une liste de listes $[u_0; \dots; u_n]$ et renvoie la liste obtenue en prenant à chaque fois le premier élément de u_i si cette liste est non vide et en l'ignorant si elle est vide :

```
(* a' list list -> 'a list *)
let rec tetes v =
  match v with
  | u :: us ->
    begin match u with
    | [] -> tetes us
    | x :: xs -> x :: tetes us
    end
  | [] -> []
```

Ces **match** imbriqués sont un peu lourds, et on peut très souvent les éviter. Ici, on écrirait plutôt :

```
let rec tetes v =
  match v with
  | [] :: us -> tetes us
  | (x :: xs) :: us -> x :: tetes us
  | [] -> []
```

Exemple 2.4

On donne ici des exemples d'erreurs classiques, et les manières de les corriger.

Les variables sont fraîches. Considérons la fonction suivante, censée déterminer si l'élément x apparaît dans la liste u :

```
(* CODE FAUX !!! *)
let rec appartient x u =
  match u with
  | [] -> false
  | x :: xs -> true
  | y :: xs -> appartient x xs
(* FIN DU CODE FAUX *)
```

À la compilation, on obtient un *warning* nous indiquant que le dernier cas ne sera jamais utilisé. En effet, la variable x apparaissant dans le deuxième cas n'a rien à voir avec l'argument x de la fonction. Le deuxième cas se lit essentiellement : « si la liste est de la forme $x :: xs$ pour un certain x et un certain xs , alors **true** », or toute liste non vide est de cette forme.

Dans un cas comme celui-ci, on utilisera soit une clause gardée (avec un **when**), soit un **if...then...else** à droite de la flèche :

```
let rec appartient x u =
  match u with
  | [] -> true
  | y :: xs when x = y -> true
  | y :: xs -> appartient x xs
```

```
let rec appartient x u =
  match u with
  | [] -> true
  | y :: xs -> if x = y then true else appartient x xs
```

Dans la deuxième version, le `if...then...else` est en fait maladroit et peut être simplifié. On obtient alors :

```
let rec appartient x u =
  match u with
  | [] -> []
  | y :: xs -> x = y || appartient x xs
```

Un entier n'a pas de forme. En mathématiques, il est tout à fait raisonnable d'écrire : « si n est de la forme $2p$ avec p entier, alors... ». En revanche, informatiquement, un entier n'a essentiellement pas de forme : c'est juste un entier. Par exemple, si l'on souhaite compter le nombre d'entiers pairs dans une liste d'entiers :

```
(* CODE FAUX !!! *)
let rec nombre_pairs u =
  match u with
  | [] -> 0
  | (2 * p) :: xs -> 1 + nombre_pairs xs
  | _ :: xs -> nombre_pairs xs
(* FIN DU CODE FAUX *)

(* CODE CORRECT *)
let rec nombre_pairs u =
  match u with
  | [] -> 0
  | n :: xs when n mod 2 = 0 -> 1 + nombre_pairs xs
  | _ :: xs -> nombre_pairs xs

(* ou bien *)
let rec nombre_pairs u =
  match u with
  | [] -> 0
  | n :: xs ->
    if n mod 2 = 0 then 1 + nombre_pairs xs
    else nombre_pairs xs

(* ou encore *)
let rec nombre_pairs u =
  match u with
  | [] -> 0
  | n :: xs -> nombre_pairs xs + if n mod 2 = 0 then 1 else 0
```

Comme un entier n'a pas de structure, il est assez rarement intéressant de faire un `match` sur un entier. Par exemple, la fonction suivante renvoie $-1, 0$ ou 1 suivant le signe de son argument, et elle est correcte, mais elle est inutilement lourde et l'on préférera un `if...then...else` :

```
(* CODE CORRECT MAIS MALADROIT *)
let signe n =
  match n with
  | n when n > 0 -> 1
  | n when n = 0 -> 0
  | _ -> -1

(* code à privilégier *)
let signe n =
  if n > 0 then 1
  else if n = 0 then 0
  else -1
```

4 Définition de types

4.1 Syntaxe

Pour nommer un nouveau type en OCaml, on utilise le mot-clé **type** :

```
type p3D = float * float * float
type v3D = float * float * float
type sphere = p3D * float
```

Pour des types comme ceux-ci, qui « existent déjà », l'intérêt peut sembler limité. Cependant la lisibilité du code peut se trouver grandement améliorée :

```
let vect3D ((x1, y1, z1) : p3D) ((x2, y2, z2) : p3D) : v3D =
  ((x2 -. x1), (y2 -. y1), (z2 -. z1))

let norme3D ((x, y, z) : v3D) = sqrt (x**2. +. y**2. +. z**2.)

let dist3D (a : p3D) (b : p3D) = norme3D (vect3D a b)

let in_sphere (a : p3D) ((centre, r) : sphere) =
  (dist3D a centre) <= r
```

Remarquez les *annotations de type* : on peut quand on déclare une fonction utiliser la syntaxe

```
let f (x_1 : t_1) ... (x_n : t_n) : u = ...
```

pour spécifier que les arguments x_1, \dots, x_n doivent respectivement être de type t_1, \dots, t_n et que le résultat doit être de type u . Ici, le type inféré pour `in_sphere` est :

```
# in_sphere;;
- : p3D -> sphere -> bool = <fun>
```

à comparer avec (si l'on n'annote rien) :

```
# in_sphere_2;;
- : float * float * float -> (float * float * float) * float -> bool = <fun>
```

4.2 Types sommes

Les types produits correspondent à un produit cartésien $A \times B$ et les types flèches à un ensemble d'applications $A \rightarrow B$ (ou B^A). La troisième opération algébrique fondamentale sur les types est l'union (disjointe) : les types correspondant à cette opération sont appelés *types sommes*, ou *types union*.

4.2.a Types énumérés finis

Un type ne contenant qu'un nombre fini de valeur (dit *type énuméré*) peut être défini avec la syntaxe suivante :

```
type enumeration = Constructeur_1 | Constructeur_2 | ... | Constructeur_n
```

Remarque

Les *constructeurs de type* doivent obligatoirement commencer par une majuscule, et ce sont les seuls noms (avec les modules) qui peuvent commencer par une majuscule en OCaml.

On peut par exemple définir un type signe à trois valeurs :

```
type signe = Negatif | Nul | Positif
```

On traduit ensuite la règle sur le signe d'un produit :

```
let signe_produit s1 s2 =
  match s1, s2 with
  | Nul, _ -> Nul
  | _, Nul -> Nul
  | Positif, _ -> s2
  | Negatif, Positif -> Negatif
  | Negatif, Negatif -> Positif
```

On peut ensuite utiliser cette fonction pour déterminer le signe d'un produit d'entiers :

```
let signe_of_int n =
  if n = 0 then Nul
  else if n < 0 then Negatif
  else Positif

let signe_produit_int n p =
  signe_produit (signe_of_int n) (signe_of_int p)
```

```
# signe_produit Negatif Negatif;;
- : signe = Positif
# signe_produit_int 10_000_000_000 10_000_000_000;;
- : signe = Positif
# signe_of_int (10_000_000_000 * 10_000_000_000);;
- : signe = Negatif (* Pourquoi ? *)
```

On peut remarquer que l'inférence de type et le pattern matching fonctionnent sans problème sur un type défini de cette manière.

Parmi les types de base de OCaml, l'un est essentiellement un type énuméré et peut être ré-implémenté sans aucune difficulté : le type **bool**

```
type booleen = Vrai | Faux
```

```
let et a b =
  match a, b with
  | Vrai, Vrai -> Vrai
  | _ -> Faux
```

Attention, la définition de la fonction et n'est en fait **pas équivalente** à celle du `&&` de OCaml puisqu'elle évalue systématiquement ses deux arguments, alors qu'on a vu que l'opérateur `&&` est séquentiel. *Idem* pour ou.

4.2.b Constructeurs paramétriques

Si l'on veut définir un type contenant une infinité de valeurs, il faut pouvoir paramétriser les constructeurs par des valeurs d'un type préexistant. Par exemple, définissons un type permettant de représenter certains sous-ensembles du plan². Les sous-ensembles que nous considérerons sont :

- l'ensemble réduit à l'origine (qui ne contient donc qu'un seul point) ;
- les droites passant par l'origine ;
- le plan en entier.

Pour le premier et troisième cas, on aura un simple constructeur puisqu'à chaque fois il n'y a qu'un seul ensemble possible. En revanche, pour le deuxième cas, il faut préciser de quelle droite on parle : pour cela, on peut par exemple donner un vecteur directeur sous la forme d'un couple de flottants. On aura donc un *constructeur paramétrique* (ici, un constructeur paramétré par deux flottants).

On obtient le type suivant :

```
type sous_ensemble =
| Zero
| Droite of float * float
| Plan
```

On peut ensuite définir une fonction inter calculant l'intersection de deux de ces sous-ensembles :

```
let inter e f =
  match e, f with
  | Plan, _ -> f
  | _, Plan -> e
  | Droite (x1, y1), Droite (x2, y2) when x1 *. y2 -. x2 *. y1 = 0. -> e
  | _ -> Zero
```

Remarque

Attention : nous avons utilisé un test d'égalité sur les flottants (pour déterminer si les deux vecteurs directeurs étaient collinaires). Il ne faut en fait **jamais faire cela** pour des raisons liées à la précision finie des flottants ; nous y reviendrons ultérieurement dans l'année.

4.2.c Types paramétriques

Certains types de OCaml sont dits *paramétriques* : par exemple, une '`a` **list**' contient des éléments qui ont tous le même type, mais ce type est quelconque.

Nous aurons souvent à définir de tels types, typiquement lorsque nous définirons de nouvelles structures de données (susceptibles de contenir des données de type quelconque). Pour l'instant, nous nous contenterons de mentionner un type paramétrique prédéfini que nous utiliserons suivant.

4.2.d Type '`a` option

Le type option est prédéfini en OCaml par

```
type 'a option = None | Some of 'a
```

C'est un type très utile pour définir des fonctions qui peuvent ne pas avoir de résultat à renvoyer. Attention, une valeur de type '`a` option' n'est pas de type '`a`' : il faut « déconstruire » l'option si l'on veut récupérer le '`a`'.

2. Les *sous-espaces vectoriels* du plan, comme vous le verrez cette année en mathématiques.

Par exemple, les deux fonctions suivantes renvoient :

- **Some** si *i* est le premier indice d'apparition de l'élément *elt* dans la liste *u*;
- **None** si *elt* n'apparaît pas dans *u*.

```
let index elt liste =
  let rec aux elt u n =
    match u with
    | [] -> None
    | x :: xs when x = elt -> Some n
    | x :: xs -> aux elt xs (n + 1) in
  aux elt liste 0
```

```
let rec index2 elt u =
  match u with
  | [] -> None
  | x :: xs when x = elt -> Some 0
  | x :: xs ->
    begin
      match index2 elt xs with
      | None -> None
      | Some n -> Some (n + 1)
    end
```

Quand on utilise ces fonctions, il faut « déballer » l'option :

```
(* minimum de (index x u) et (index x v) (ou None) *)
let premier_x x u v =
  match (index x u, index x v) with
  | Some n, Some p -> Some (min n p)
  | Some n, None -> Some n
  | None, Some p -> Some p
  | None, None -> None
```

```
# premier_x 7 [1; 3; 5; 7] [2; 3; 1; 4; 8; 7];
- : int option = Some 3
# premier_x 8 [1; 3; 5; 7] [2; 3; 1; 4; 8; 7];
- : int option = Some 4
# premier_x (-12) [1; 3; 5; 7] [2; 3; 1; 4; 8; 7];
- : int option = None
```

4.3 Types enregistrement

Un type enregistrement (*record* en anglais) est essentiellement un type produit dans lequel chaque composante a un *nom*. La syntaxe est

```
type enregistrement = {nom_1 : type_1 ; ... ; nom_n : type_n};;
```

et, si *x* est de type *enregistrement*, on accède ensuite à la valeur du champ *nom_1* de *x* par *x.nom_1*.

Remarques

- Les règles qui déterminent ce qui se passe quand plusieurs types enregistrement différents ont des champs qui portent le même nom sont un peu complexes : le plus simple est d'éviter cette situation.
- L'ordre des champs n'a pas d'importance, seul le nom compte. Par exemple, dans l'exemple qui suit (sur les nombres complexes), `{re = 1. ; im = 0.}` et `{im = 0. ; re = 1.}` sont indiscernables.

4.3.a Exemple élémentaire : nombres complexes

Si l'on veut définir un type pour représenter des complexes, il est assez logique de procéder ainsi :

```
type complexe = {re : float ; im : float}

let conjugue z = {re = z.re ; im = -.z.im}
```

```
# let i = {re = 0.; im = 1.};;
val i : complexe = {re = 0.; im = 1.}
# let j = {im = (sqrt 3.) /. 2.; re = 0.5};;
val j : complexe = {re = 0.5; im = 0.866025403784438597}
(* L'ordre dans lequel on remplit les champs n'a pas d'importance :
   OCaml regarde les noms *)
# conjugue i;;
- : complexe = {re = 0.; im = -1.}
# conjugue j;;
- : complexe = {re = 0.5; im = -0.866025403784438597}
```

On peut aussi faire du pattern matching sur des enregistrements (ce n'est qu'assez rarement utile) :

```
let norme z =
  match z with
  {re = x ; im = y} -> sqrt (x**2. +. y**2.)
```

```
# norme j = 1.;;
- : bool = false
# norme j;;
- : float = 0.99999999999999889
(* Les tests d'égalité sur les flottants, ça ne marche pas ! *)
```

Dans ce cas, tout comme pour les types produits, il est plus pratique de « déconstruire » l'objet directement :

```
let norme {re = x; im = y} =
  sqrt (x**2. +. y**2.)
```

En réalité, le mieux est souvent de ne pas déconstruire mais d'accéder aux champs quand on en a besoin :

```
let norme z =
  sqrt (z.re ** 2. +. z.im ** 2.)
```

4.3.b Un exemple classique : le jeu de cartes

On veut définir un type permettant de représenter des cartes à jouer.

- Une carte est caractérisée par sa *couleur* et sa *valeur*.
- Les couleurs possibles sont pique, cœur, carreau et trèfle.
- Les valeurs possibles sont as, roi, dame, valet ainsi que les entiers de 2 à 10.

Une possibilité pour le type :

```
type couleur = Pique | Coeur | Carreau | Trefle
type valeur = As | Roi | Dame | Valet | Mineure of int
type carte = {co : couleur ; va : valeur}
(* Le 8 de trefle sera donc {co = Trefle ; va = Mineure 8}
   et le roi de cœur {co = Coeur ; va = Roi} *)
```

Et deux exemples de fonctions très simples opérant sur ce type :

```
let pique_ou_as carte =
  match carte with
  | {co = Pique} -> true (* on n'est pas obligé de "matcher" tous les champs *)
  | {va = As} -> true
  | _ -> false

(* Ici, le pattern matching n'apporte pas grand-chose : *)
let trefle_ou_8 carte =
  (carte.co = Trefle) || (carte.va = Mineure 8)
```

Vous écrirez quelques autres fonctions en TD.

Exercices Supplémentaires

I Version récursive des boucles `while`

Exercice 2.5

p. 37

Pour $n \geq 0$, on définit la *somme harmonique* :

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

- Écrire une fonction harmonique : `int` -> `float` prenant en entrée un entier n supposé supérieur ou égal à zéro et renvoyant H_n .
- On admet (vous le montrerez en cours de mathématiques) que $H_n \rightarrow +\infty$ quand $n \rightarrow \infty$. Définir mathématiquement ce que renvoie la fonction suivante :

```
let premier_n x =
  let rec aux k =
    if harmonique k >= x then k
    else aux (k + 1) in
  aux 0
```

- Si `premier_n x` vaut n , combien d'additions vont être effectuées lors du calcul (en fonction de n) ? Raisonnement, combien devrait-on en effectuer ?
- Combien de temps prend le calcul de `premier_n 7.` ? de `premier_n 9.` ? de `premier_n 11.` ?
Vous pouvez mesurer très approximativement « à la main » ou utiliser la fonction `Sys.time` si vous voulez être plus précis.
- Écrire une version plus efficace de `premier_n` et comparer les temps de calcul expérimentaux.
On pourra utiliser une fonction auxiliaire `aux k s` en maintenant l'invariant $s = \sum_{i=1}^k \frac{1}{i}$.

Exercice 2.6

p. 37

Tout entier strictement positif n peut être encadré entre deux puissances de 2 consécutives : $2^k \leq n < 2^{k+1}$. Dans ce cas, on dira que le *logarithme discret* $\lfloor \log_2 n \rfloor$ de n est égal à k . Cela revient à définir :

$$\lfloor \log_2 n \rfloor := \max\{k \in \mathbb{N} \mid 2^k \leq n\}$$

- Écrire une fonction `deux_puissance` : `int` -> `int` telle que `deux_puissance k` renvoie l'entier 2^k pour $k \geq 0$.
On pourrait ici utiliser l'opérateur de décalage, mais nous en parlerons plus tard.
- En utilisant cette fonction, écrire une fonction `log2` : `int` -> `int` prenant en entrée un entier n supposé strictement positif et renvoyant son logarithme discret.
On pourra compléter le squelette suivant :

```

let log2 n =
  (* aux k renvoie le plus grand k' tel que
   *  $2^{k'} \leq n$ , en supposant  $2^k \leq n$ . *)
  let rec aux k =
    ...
    ... in
  aux ...

```

3. Montrer que si $n \geq 2$, alors $\lfloor \log_2 n \rfloor = 1 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor$.
4. En déduire une version beaucoup plus simple de la fonction log2.

2 Entrainement sur les listes

Exercice 2.7

p. 38

Écrire une fonction double : '`a list` -> '`a list` qui à une liste `[a1; ...; ap]` associe `[a1; a1; a2; a2, ..., ap ;ap]`.

Exercice 2.8

p. 38

Écrire une fonction repete : '`a list` -> `int` -> '`a list` telle que repete `u n` soit égal à `u @ u @ ... @ u (n fois)`.

```

utop[75]> repete [1; 2; 3] 4;;
- : int list = [1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3]

```

Exercice 2.9

p. 38

Écrire une fonction monotone : '`a list` -> `bool` qui détermine si une liste est monotone. Le plus simple est d'effectuer deux parcours de la liste, mais on peut s'en sortir avec un seul.

Exercice 2.10

p. 39

Écrire une fonction partition : (`'a -> bool`) -> '`a list` -> (`'a list * 'a list`) qui prend un *prédictat* (une fonction renvoyant un booléen) et une liste `u` et renvoie le couple `(v,w)` où `v` est la liste des éléments de `u` vérifiant le prédictat et `w` celle des éléments de `u` ne le vérifiant pas. L'ordre des éléments sera conservé.

```

utop[77]> partition (fun x -> (x > 0)) [2; -1; -5; 4; 6; 0; -4];;
- : int list * int list = ([2; 4; 6], [-1; -5; 0; -4])

```

Exercice 2.11

p. 39

1. Écrire une fonction zip : '`a list` -> '`b list` -> (`'a * 'b`) `list` qui prend deux listes `[x_1; ... ; x_n]` et `[y_1; ... ; y_n]` et renvoie `[(x_1, y_1); ... ; (x_n, y_n)]`. La fonction lèvera l'exception `Failure "longueurs differentes"` (via `failwith "..."`) si les listes n'ont pas la même longueur.

```
# zip [1; 2; 4] ["a"; "c"; "b"];;
- : (int * string) list = [(1, "a"); (2, "c"); (4, "b")]
# zip [1; 2; 4; 5] ["a"; "c"; "b"];;
Exception: Failure "longueurs differentes"
```

2. Écrire une fonction `unzip` telle que `unzip (zip u v)` renvoie `(u, v)`.

```
# unzip (zip [1; 2; 4] ["a"; "c"; "b"]);
- : int list * string list = ([1; 2; 4], ["a"; "c"; "b"])
```

Exercice 2.12

p. 39

1. Écrire une fonction `sommes_cumulees : int list -> int list` qui renvoie $[x_1; x_1 + x_2; \dots; x_1 + \dots + x_n]$ quand on l'appelle sur $[x_1; \dots; x_n]$:

```
# sommes_cumulees [2; 3; 5; 31];
- : int list = [2; 5; 10; 41]
```

2. Estimer le nombre d'additions effectuées par votre fonction si la liste est de longueur n .
 3. Si ce nombre n'est pas de l'ordre de n , ré-écrire la fonction pour que ce soit le cas.

Exercice 2.13

p. 40

On rappelle que la variance de (x_1, \dots, x_n) est définie par $V = \frac{1}{n} \sum_{k=1}^n (x_i - \bar{x})^2$, où \bar{x} est la moyenne de (x_1, \dots, x_n) .

1. Écrire une fonction `moyenne : float list -> float option` (qui renverra `None` si la liste est vide).
 2. Écrire une fonction `variance : float option -> float option` (qui renverra `None` si la liste est vide).

```
# variance [];
- : float option = None
# variance [1.];
- : float option = Some 0.
# variance [1.; 3.; 5.];
- : float option = Some 2.66666666666666652
```

Exercice 2.14

p. 40

Écrire une fonction `make_index : (int * string list) list -> (string * int list) lista` qui fonctionne comme sur l'exemple suivant (pensez à l'index d'un livre) :

```
# make_index [(1, ["ab"; "ac"]); (2, ["ab"; "ad"]); (3, ["ab"; "zz";
→ "ac"])];
- : (string * int list) list =
[("ac", [3; 1]); ("ab", [3; 2; 1]); ("ad", [2]); ("zz", [3])]
```

La chaîne `"ac"` apparaît aux « pages » 1 et 3, `"ab"` apparaît à chaque page, `"ad"` seulement en page 2...

Remarque

Il y a des solutions efficaces à ce problème (classique), utilisant des structures de données plus sophistiquées que les listes. Nous aurons l'occasion d'y revenir, mais l'idée ici est simplement de s'entraîner à la manipulation de listes en OCaml.

a. le type que vous obtiendrez sera plus général, ce qui n'est pas gênant

3 Approfondissement

Exercice 2.I5 – Produit cartésien

p. 40

Écrire une fonction `produit` : `'a list -> 'b list -> ('a * 'b) list` qui calcule le « produit cartésien » de deux listes. Autrement dit, `produit u v` doit renvoyer la liste des couples (x, y) où x est un élément de u et y un élément de v (l'ordre d'apparition des couples est quelconque). D'éventuelles répétitions dans l'une des deux listes donneront des répétitions dans le produit.

```
utop[2]> produit [1; 2; 3] ["a"; "b"; "c"];
- : (int * string) list =
[(1, "a"); (1, "b"); (1, "c"); (2, "a"); (2, "b"); (2, "c"); (3, "a");
 (3, "b"); (3, "c")]
utop[3]> produit [1; 2; 1] [12; 24];
- : (int * int) list = [(1, 12); (1, 24); (2, 12); (2, 24); (1, 12); (1,
→ 24)]
```

Exercice 2.I6 – Combinations avec répétitions

p. 40

Si vous disposez de n objets identiques à ranger dans m tiroirs, et que le nombre d'objets par tiroir n'est pas limité, une manière possible de ranger les objets correspond exactement à un m -uplet d'entiers positifs ou nuls dont la somme vaut n : la valeur de la première composante indique combien il y a d'objets dans le premier tiroir, la deuxième composante le nombre d'objets dans le deuxième tiroir et ainsi de suite. Par exemple, pour $n = 2$ et $m = 3$, on obtient $(2, 0, 0)$, $(1, 1, 0)$, $(1, 0, 1)$, $(0, 2, 0)$, $(0, 1, 1)$ et $(0, 0, 2)$.

Vous verrez plus tard dans l'année (en cours de mathématiques) combien de possibilités distinctes existent. Écrire une fonction `enumere` : `int -> int -> int list` renvoyant la liste de toutes ces possibilités.

```
# enumere 2 4;;
- : int list list =
[[0; 0; 0; 2]; [0; 0; 1; 1]; [0; 0; 2; 0]; [0; 1; 0; 1]; [0; 1; 1; 0];
 [0; 2; 0; 0]; [1; 0; 0; 1]; [1; 0; 1; 0]; [1; 1; 0; 0]; [2; 0; 0; 0]]
```

Exercice 2.I7 – Une élimination plus poussée

p. 41

On considère le processus suivant :

- on part d'une liste (d'entiers, par exemple);
- à chaque fois que l'on trouve deux éléments consécutifs égaux, on les supprime tous les deux;
- si l'on crée ainsi de nouvelles opportunités de suppression (car deux éléments égaux se retrouvent désormais côté à côté), on itère le procédé;
- on s'arrête quand il n'y a plus d'éléments consécutifs égaux.

Notez que le résultat final ne dépend pas de l'ordre dans lequel on choisit de faire les suppressions.

Un exemple pour illustrer le processus (en partant deux fois de la même liste mais en faisant les éliminations dans un ordre différent) :

- | | |
|-------------------------------|-------------------------------|
| ■ 1,2,2,3,1,3,1,4,4,2,1,1,2,1 | ■ 1,2,2,3,1,3,1,4,4,2,1,1,2,1 |
| ■ 1,3,1,3,1,4,4,2,1,1,2,1 | ■ 1,2,2,3,1,3,1,2,1,1,2,1 |
| ■ 1,3,1,3,1,2,1,1,2,1 | ■ 1,2,2,3,1,3,1,2,2,1 |
| ■ 1,3,1,3,1,2,2,1 | ■ 1,2,2,3,1,3,1,1,1 |
| ■ 1,3,1,3,1, | ■ 1,2,2,3,1, |
| ■ 1,3,1,3 | ■ 1,3,1,3 |

Écrire une fonction `elimine` : `'a list -> 'a list` qui renvoie le résultat final (en essayant si possible d'être efficace).

Solutions

Correction de l'exercice 2.5 page 32

- Il faut juste faire attention aux conversions entiers-flottants :

```
let rec harmonique n =
  if n = 0 then 0.
  else 1. /. float n +. harmonique (n - 1)
```

- Cette fonction renvoie le plus petit entier $n \geq 0$ tel que $H_n \geq x$.
- Un appel à `harmonique k` effectue k additions (et k calculs d'inverse). Si `premier_n x` renvoie n , alors on a effectué des appels pour $k = 0 \dots n$, ce qui fait un total de $\sum_{k=0}^n k = \frac{n(n+1)}{2}$ additions.
Si on faisait le calcul « à la main » (en s'aidant d'une calculatrice, disons), on ne ferait que n additions : en effet, on ne recommencerait pas le calcul à partir de zéro à chaque étape.
- Sur ma machine, j'obtiens environ 5 millisecondes pour $x = 7$ (qui renvoie 616), 230 millisecondes pour $x = 9$ (qui renvoie 4550) et 18 secondes pour $x = 11$ (qui renvoie 33617).
- Cette version n'effectue que n additions (ou plutôt, *de l'ordre de* n additions) si elle renvoie n :

```
let premier_n x =
  let rec aux k s =
    if s >= x then k
    else aux (k + 1) (s +. 1. /. float (k + 1)) in
  aux 0 0.
```

Le calcul de `premier_n 11.` est ici presque instantané (une milliseconde).

Correction de l'exercice 2.6 page 32

- Cette fonction ne pose pas de problème :

```
let rec deux_puissance k =
  if k = 0 then 1
  else 2 * deux_puissance (k - 1)
```

Comme dit dans l'énoncé, l'opérateur de décalage à gauche `lsl` permet de faire directement ce calcul sans écrire de fonction, mais nous en parlerons quand nous nous intéresserons à la représentation des entiers.

- On complète ce qui est fourni :

```
let log2 n =
  let rec aux k =
    if deux_puissance (k + 1) > n then k
    else aux (k + 1) in
  aux 0
```

À chaque étape, on a $2^k \leq n$. Si $2^{k+1} > n$, alors k est la valeur cherchée, sinon on ré-essaie avec $k + 1$.

3. Soit $n \geq 2$ et $k = \lfloor \log_2 n \rfloor$. Notons que $k \geq 1$ (puisque $n \geq 2$). On a $2^k \leq n < 2^{k+1}$, donc $2^{k-1} \leq n/2 < 2^k$.

- Comme la fonction partie entière est croissante, on a $\lfloor 2^{k-1} \rfloor \leq \lfloor n/2 \rfloor$, et donc $2^{k-1} \leq \lfloor n/2 \rfloor$ ($k \geq 1$ donc 2^{k-1} est entier).

- Comme $\lfloor x \rfloor \leq x$ pour tout $x \in \mathbb{R}$, on a aussi $\lfloor n/2 \rfloor < 2^k$.

On en déduit $2^{k-1} \leq \lfloor n/2 \rfloor < 2^k$, et donc par définition $\lfloor \log_2 n/2 \rfloor = k - 1$, ce qui permet de conclure.

4. Avec la relation que l'on vient de prouver, l'écriture de la fonction est immédiate :

```
let rec log2 n =
  if n <= 1 then 0
  else 1 + log2 (n / 2)
```

Correction de l'exercice 2.7 page 33

```
let rec double u =
  match u with
  | [] -> []
  | x :: xs -> x :: x :: double xs
```

Correction de l'exercice 2.8 page 33

```
let rec repete u n =
  if n = 0 then []
  else u @ repete u (n - 1)
```

Correction de l'exercice 2.9 page 33

```
(* On peut utiliser deux fonctions croissante et decroissante. *)
let rec croissante = function
  | [] | [_] -> true
  | x :: y :: xs -> (x <= y) && croissante (y :: xs)

(* Une petit raffinement syntaxique (loin d'être indispensable) :
   le "as" dans un pattern matching pour capturer une expression. *)
let rec decroissante = function
  | [] | [_] -> true
  | x :: (y :: xs as v) -> (x >= y) && decroissante v

let monotone_1 u = croissante u || decroissante u
```

La première version peut faire deux parcours de la liste. On peut aussi s'en sortir en un seul parcours (ça n'apporte pas grand chose) :

```

let monotone u =
  (* Le paramètre crois vaut truessi le préfixe déjà lu est croissant,
   * decr vaut truessi le préfixe déjà lu est décroissant. *)
  let rec aux u crois decr =
    match u with
    | [] | [_] -> true
    | x :: y :: xs when x < y -> crois && aux (y :: xs) true false
    | x :: y :: xs when x > y -> decr && aux (y :: xs) false true
    | x :: y :: xs -> aux (y :: xs) crois decr in
  aux u true true

```

Correction de l'exercice 2.I0 page 33

```

let rec partition pred u =
  match u with
  | [] -> ([], [])
  | x :: xs -> let (oui, non) = partition pred xs in
    if pred x then (x :: oui, non)
    else (oui, x :: non)

```

Correction de l'exercice 2.II page 33

1.

```

let rec zip u v =
  match u, v with
  | [], [] -> []
  | x :: xs, y :: ys -> (x, y) :: (zip xs ys)
  | _ -> failwith "longueurs différentes"

```

2.

```

let rec unzip w =
  match w with
  | [] -> ([], [])
  | (x, y) :: tail -> let u, v = unzip tail in (x :: u, y :: v)

```

Correction de l'exercice 2.I2 page 34

On donne directement une version efficace, ne parcourant qu'une seule fois la liste (et faisant un nombre d'opérations proportionnel à la longueur de celle-ci) :

```

let sommes_cumulees u =
  (* s est la somme du préfixe déjà traité, u le suffixe restant
   * à traiter. *)
  let rec aux u s =
    match u with
    | [] -> []
    | x :: xs -> (x + s) :: (aux xs (x + s)) in
  aux u 0

```

Correction de l'exercice 2.13 page 34

1. J'utilise ici un *fold* pour la fonction somme, mais cela n'a bien sûr rien d'obligatoire.

```
let somme m = List.fold_left ( +. ) 0. m

let moyenne m = match List.length m with
| 0 -> None
| n -> Some ((somme m) /. float_of_int n)
```

2. Le *map* est bien pratique ici, même si l'on peut faire sans :

```
let variance u = match moyenne u with
| None -> None
| Some m -> moyenne (List.map (fun x -> (x -. m) ** 2.) u)
```

Correction de l'exercice 2.14 page 34

La fonction est légèrement désagréable à écrire, les *fold* permettent de rester concis.

```
let creer_index pages =
(* ajouter_item n index s renvoie index dans lequel on a ajouté une
entrée page n pour la chaîne s *)
let rec ajouter_item n index s =
match index with
| [] -> [(s, [n])]
| (t, numeros) :: reste when t = s -> (t, n :: numeros) :: reste
| x :: reste -> x :: ajouter_item n reste s in
(* renvoie index dans lequel on a ajouté une entrée page n pour chaque
élément de items *)
let ajouter_page index (n, items) =
List.fold_left (ajouter_item n) index items in
(* On part d'un index vide et on ajoute toutes les pages *)
List.fold_left ajouter_page [] pages
```

Correction de l'exercice 2.15 page 35

Pour faire le produit de $x :: xs$ et v , on met d'abord tous les couples de la forme (x, y) avec $y \in v$, puis les éléments du produit de xs et de v . Le cas de base ($u = []$) est immédiat.

```
let rec produit u v =
match u with
| [] -> []
| x :: xs ->
let avec_x = List.map (fun y -> (x, y)) v in
avec_x @ produit xs v
```

Correction de l'exercice 2.16 page 35

Ça devient nettement plus délicat, surtout à cette période de l'année. Savoir écrire cette fonction n'est pas un objectif dans l'immédiat.

```

(* range a b = [a; a + 1; ...; b - 1] *)
let rec range a b =
  if a >= b then []
  else a :: (range (a + 1) b)

(* On va utiliser List.flatten, qui prend une liste de listes
   et en renvoie la concaténation.
   Si l'on souhaite la redéfinir soi-même, c'est facile : *)
let rec concat = function
  | [] -> []
  | liste :: autres_listes -> liste @ concat autres_listes
(* ou, pour faire court : *)
let concat_2 = List.fold_left (@) []

(* Et maintenant l'énumération des combinaisons avec répétitions : *)
let rec enumere n m =
  (* prefixe : 'a -> 'a list list -> 'a list list
     prefixe x [l_1; l_2; l_3...] = [x::l_1; x::l_2; x::l_3...]*)
  let prefixe x = List.map (fun u -> x :: u) in
  (* reccur p renvoie la liste de toutes les répartitions de n objets
     dans m tiroirs contenant p objets dans le premier tiroir *)
  let reccur p = prefixe p (enumere (n - p) (m - 1)) in
  (* 0 objet, 0 tiroir : une unique possibilité (la combinaison vide) *)
  if m = 0 && n = 0 then [[]]
  (* au moins 1 objet, 0 tiroir : aucune possibilité *)
  else if m = 0 then []
  else List.flatten (List.map reccur (range 0 (n + 1)))

```

Correction de l'exercice 2.17 page 35

Pour commencer, on peut essayer de traduire le plus simplement possible le processus décrit :

- la fonction `une_passe` parcourt la liste en éliminant tous les couples d'éléments successifs égaux;
- on rappelle cette fonction tant qu'il se passe quelque chose.

```

let rec une_passe = function
  | [] -> []
  | x :: y :: xs when x = y -> une_passe xs
  | x :: xs -> x :: une_passe xs

let rec elimine_1 u =
  let u' = une_passe u in
  if u = u' then u else elimine_1 u'

```

Cette solution n'est pas très satisfaisante car on peut être amené à effectuer de nombreux parcours de la liste. Par exemple, si l'on considère la liste $u = v @ [2] @ v$ avec $v = [1; 2; 1; 2; \dots; 1; 2]$, on élimine seulement deux éléments à chaque appel à `une_passe`, alors que le résultat final est $[2]$. Si u contient n éléments, cela donne donc un nombre total d'opérations de l'ordre de n^2 .

On peut faire bien mieux, en n'effectuant qu'une seule passe au total :

```
let elimine u =
  (* vus : les éléments déjà vus et non éliminés (plus récent en tête)
   reste : les éléments qui restent à voir *)
  let rec aux vus reste =
    match vus, reste with
    | _, [] -> List.rev vus
    | x :: xs, y :: ys when x = y -> aux xs ys
    | _, y :: ys -> aux (y :: vus) ys in
  aux [] u
```

En prenant `u = (repete [1; 2] 10_000) @ [2] @ (repete [1; 2] 10_000)`, on obtient `elimine_1 u = [2]` en environ 30 secondes, et `elimine u = [2]` instantanément.

ASPECTS IMPÉRATIFS DE OCAML

I Type `unit`

`unit` est l'un des types de base de OCaml : celui des *actions*. Il n'y a qu'une seule valeur de type `unit`, c'est `()`.

```
# let x = ();
val x : unit = ()
```

Une fonction de type `t -> unit`, où `t` est un type, ne renvoie rien¹. La fonction `print_string`, par exemple, est de type `string -> unit` : si on l'applique à une chaîne en écrivant `print_string "Hello!"`, on n'obtient pas une « valeur » qui pourrait être ré-utilisée dans un calcul ultérieur mais une *action* : l'affichage de `Hello!` à l'écran².

Une fonction peut aussi être de type `unit -> 'a` pour un certain type `'a`. Cela signifie que la fonction ne prend pas de paramètre (significatif) en entrée. Par exemple, la fonction `print_newline : unit -> unit` qui affiche une ligne vide³ :

```
# print_newline;;
- : unit -> unit = <fun>
# print_newline ();;
- : unit = ()
(*
  On remarquera que même si print_newline ne prend "pas d'argument", il faut
  quand même lui passer l'argument () pour effectuer l'appel.
*)
```

Il arrive souvent que l'on souhaite exprimer quelque chose comme « dans ce cas, fais cela, dans les autres cas, ne fais rien ». C'est tout-à-fait possible, en utilisant `()` pour dire « ne fais rien » :

```
let affiche_si_pair n =
  match n mod 2 with
  | 0 -> print_int n
  | _ -> ()

(* ou bien *)
let affiche_si_pair n =
  if n mod 2 = 0 then
    print_int n
  else
    ()
```

Ce dernier cas se présente assez souvent : c'est pour cela que dans la plupart des langages, la branche `else` est optionnelle dans un `if...then...else`. En OCaml, elle est en général obligatoire, sauf si le type de retour est `unit`.

1. Techniquement, une telle fonction renvoie `()`, un peu comme une fonction sans `return` en Python qui renvoie `None`.
2. En C (et dans de nombreux langages inspirés du C), une telle fonction aurait `void` comme type de retour.
3. Et qui sert donc surtout à revenir à la ligne entre deux affichages.

```
(* CODE FAUX *)
let abs n = if n < 0 then -n
(* erreur de type : c'est logique, qu'est censé valoir abs 3 ? *)

(* CODE CORRECT *)
let affiche_si_negatif n = if n < 0 then print_int n
(* strictement équivalent à :
let affiche_si_pair n = if n < 0 then print_int n else () *)
```

L'idée est simplement que `if cond then exp1 else exp2` est une expression dotée d'une valeur et donc d'un type. Ce type est celui, **nécessairement commun**, de `exp1` et `exp2`. Dans le cas où `exp1` est de type `unit`, on dispose d'une valeur par défaut (`()`) si `exp2` n'est pas spécifié; si `exp1` a un autre type, ce n'est pas le cas.

2 Variables mutables

En mathématiques, si l'on écrit « soit x l'unique réel vérifiant... » ou « soit f la fonction qui à un réel x associe... », il est sous-entendu que l'objet ainsi défini ne changera jamais de valeur. On peut éventuellement réutiliser le nom f pour désigner ultérieurement une autre fonction (une fois qu'on n'aura plus besoin de la première fonction f), mais il s'agira alors d'un autre objet : les occurrences de f qui suivent la première définition mais précèdent la deuxième feront toujours référence à la première définition.

Il en est de même en OCaml, comme le montre l'exemple suivant :

```
# let x = 3;;
val x : int = 3
# let y = x + 2;;
val y : int = 5
# let f n = x * n;;
val f : int -> int = <fun>
# let x = 7;;
val x : int = 7
# y;;
- : int = 5
(* Pas très surprenant, le résultat serait le même dans la plupart des langages :
 * quand "let y = x + 2" a été executé, x a été remplacé par sa valeur.
 * La valeur de y n'est pas affectée par une redéfinition ultérieure de x. *)
# f 3;;
- : int = 9
(* Plus surprenant : le "x" dans la définition de f se réfère
 * toujours à la définition de x en vigueur au moment du "let f =". *)
```

En Python, c'est un peu différent :

```
x = 3

def f(n):
    return x * n

print(f(3))
x = 7
print(f(3))
```

Affichage obtenu :

```
9
21
```

Il est cependant possible de définir des variables *mutables* (pouvant changer de valeur) en OCaml : il faut juste le spécifier.

2.1 Champs mutables dans un enregistrement

Quand on définit une type enregistrement, il est possible de spécifier que certains champs sont modifiables après la création à l'aide du mot-clé **mutable**. Définissons par exemple un type **compteur** qui contiendra trois champs entiers : **debut** et **fin** contiendront respectivement les valeurs initiales et finales du compteur, alors que **courant** contiendra la valeur courante. Seul ce dernier champ est susceptible de changer durant la « vie » du compteur, ce qui nous conduit à la définition de type suivante :

```
type compteur = {debut : int ; fin : int ; mutable courant : int}
```

Une fois créé un compteur, on peut modifier la valeur de son champ **courant** avec l'opérateur `<-` :

```
# let c = {debut = 1; fin = 5; courant = 1};;
val c : compteur = {debut = 1; fin = 5; courant = 1}
# c.courant <- 4;;
- : unit = ()
# c;;
- : compteur = {debut = 1; fin = 5; courant = 4}
# c.debut <- 2;;
Characters 0-12:
  c.debut <- 2;;
~~~~~
Error: The record field label debut is not mutable
```

On peut maintenant définir une fonction **incremente** : **compteur** → **unit** qui tente d'incrémenter le compteur et lève une exception si le compteur est déjà à sa valeur finale.

```
let incremente c =
  if c.courant < c.fin then
    c.courant <- c.courant + 1
  else
    failwith "valeur maximale atteinte"
```

Une telle fonction est dite *impure*, ou *avec effets de bord*. La différence fondamentale avec les fonctions *pures* vues jusqu'à maintenant est que **incremente** modifie l'« univers » quand on l'appelle. En particulier, cela signifie que deux appels successifs à **incremente** avec le même argument peuvent avoir un comportement différent.

```
# let test = {debut = 1 ; fin = 5 ; courant = 3};;
val test : compteur = {debut = 1; fin = 5; courant = 3}
# incremente test;;
- : unit = ()
# test;;
- : compteur = {debut = 1; fin = 5; courant = 4}
# incremente test;;
- : unit = ()
# test;;
- : compteur = {debut = 1; fin = 5; courant = 5}
# incremente test;;
Exception: Failure "valeur maximale atteinte".
```

2.2 Une syntaxe allégée : les références

Très souvent, on souhaite simplement créer une variable « mutable » d'un certain type **t** pré-existant. Il est bien sûr possible définir un type enregistrement *ad hoc* :

```
type 'a modifiable = {mutable valeur : 'a}
```

On peut alors procéder comme suit :

```
# let x = {valeur = 3};;
val x : int modifiable = {valeur = 3}
# x.valeur <- 17;;
- : unit = ()
# x.valeur <- x.valeur * 2;;
- : unit = ()
# x;;
- : int modifiable = {valeur = 34}
# x.valeur;;
- : int = 34
```

Cependant, cette manière de procéder est quelque peu lourde d'un point de vue syntaxique. OCaml propose donc un *syntactic sugar* : le type prédéfini '`a ref`'. Il s'agit du même type (aux noms près) que le '`a modifiable`' défini plus haut :

```
type 'a ref = {mutable contents : 'a}
```

L'avantage est que l'on dispose de deux opérateurs prédéfinis ! (préfixe) et `:=` (infixe) :

Opérateur	Type	Signification	Syntaxe
<code>ref</code>	<code>'a -> 'a ref</code>	Création d'une référence	<code>ref x</code> équivaut à <code>{contents = x}</code>
<code>!</code>	<code>'a ref -> 'a</code>	Dé-référencement	<code>!x</code> équivaut à <code>x.contents</code>
<code>:=</code>	<code>'a ref -> 'a -> unit</code>	Affectation	<code>x := y</code> équivaut à <code>x.contents <- y</code>

FIGURE 3.1 – Opérations sur les références

Un exemple d'utilisation :

```
# let x = ref 3. ;;
val x : float ref = {contents = 3.}
# x +. 1.2;;
Characters 0-1:
  x +. 1.2;;
  ^
Error: This expression has type float ref
       but an expression was expected of type float
# !x +. 1.2;;
- : float = 4.2
# x := (!x)**3. ;;
- : unit = ()
# x;;
- : float ref = {contents = 27.}
# !x;;
- : float = 27.
```

Si `x` est une référence et que l'on écrit `let y = x`, on fait « pointer » `y` vers la même case mémoire que `x` (on parle d'égalité *physique*). Cela signifie que toute modification ultérieure du contenu de `x` affectera `y`, et inversement.

```

# let x = ref 0;;
val x : int ref = {contents = 0}
# let y = x;;
val y : int ref = {contents = 0}
# x := 2;;
- : unit = ()
# y;;
- : int ref = {contents = 2}
# y := -5;;
- : unit = ()
# x;;
- : int ref = {contents = -5}
# let x = ref 2;;
val x : int ref = {contents = 2}
(* On définit un nouveau x qui ne pointe plus vers la même case *)
# y;;
- : int ref = {contents = -5}
(* Le nouveau x et y ne sont donc plus liés. *)

```

3 L'opérateur ;

En OCaml, le ; est un opérateur de *séquencement* : on peut considérer que expr₁ ; expr₂ signifie « évaluer l'instruction 1, puis l'expression 2 ». Plus précisément, expression₁ ; expression₂ signifie :

- évaluer expression₁ (ce calcul peut comporter des effets de bord et donc résulter en un affichage, la modification d'une variable mutable...);
- jeter le résultat de ce calcul s'il y en a un;
- évaluer expression₂ (ici aussi, il peut y avoir des effets de bord); le résultat de cette évaluation donne la valeur de l'expression complète expression₁; expression₂.

Techniquement, ; est un opérateur infixé de type 'a -> 'b -> 'b. Cependant, le type 'a sera presque toujours **unit** (OCaml émet d'ailleurs un *warning* si ce n'est pas le cas). Un exemple typique d'utilisation (on réutilise le type compteur défini plus haut) :

```

let incremente c =
  if c.courant < c.fin then
    begin
      c.courant <- c.courant + 1;
      print_string "La nouvelle valeur est ";
      print_int c.courant
    end
  else
    failwith "valeur maximale atteinte"

```

```

# c;;
- : compteur = {debut = 1; fin = 5; courant = 3}
# incremente c;;
La nouvelle valeur est 4- : unit = ()
# c;;
- : compteur = {debut = 1; fin = 5; courant = 4}

```

Remarquez que le **begin...end** est **indispensable**⁴ : le code **if a then b; c else d** est lu (**if a then b; c else d**), ce qui résulte en une erreur de syntaxe. La question de savoir ce qui fait ou non partie du corps d'un **then** ou d'un **else** est une **source d'erreurs récurrentes**; un bon réflexe est

4. On peut cependant remplacer le **begin** par une parenthèse ouvrante et le **end** par une parenthèse fermante.

de demander régulièrement à l'éditeur (VS Code, Emacs...) de ré-indenter le code, ce qui permet de mettre en évidence les différences éventuelles entre ce que l'on voulait dire et ce que le compilateur va comprendre :

```
(* Code mal indenté : ne fait pas ce qu'on pourrait croire *)
if condition then
  instruction_1
else
  instruction_2;
instruction_3

(* Code correctement indenté : on voit que instruction_3 ne fait pas partie
   du if/then/else et sera exécutée dans tous les cas. *)
if condition then
  instruction_1
else
  instruction_2;
instruction_3
```

4 Boucles

4.1 Boucle `for`

OCaml propose deux variantes de la boucle `for`, selon que l'indice croît ou décroît :

```
for compteur = start to finish do
  expression
done
```

```
for compteur = start downto finish do
  expression
done
```

Remarques

- `start` et `finish` doivent être de type `int` et expression de type `unit`.
- `compteur` est considéré comme une variable de type `int` locale à la boucle et ne peut pas être modifiée « à la main » dans le corps de la boucle.
- Notez bien que les deux bornes de la boucle sont **incluses**.
- OCaml ne possède pas d'équivalents de `break`, `continue` et autres : une boucle `for` est forcément exécutée en entier (et l'on ne sort d'une boucle `while` que quand la condition devient fausse).

Calculons par exemple la somme des entiers et des carrés de `deb` à `fin` (inclus) :

```
let sommes deb fin =
  let s = ref 0 in
  let t = ref 0 in
  for i = deb to fin do
    s := !s + i;
    t := !t + i * i
  done;
  (!s, !t)

(* val sommes : int -> int -> int * int = <fun>
 * # sommes 1 5;;
 * - : int * int = (15, 55) *)
```

Exercice 3.1

Écrire une fonction pour calculer chacune des sommes suivantes (« bêtement », c'est-à-dire sans chercher à simplifier mathématiquement la somme) :

1. $\sum_{i=1}^n \frac{1}{i+n}$;
2. $\sum_{i=1}^n \sum_{j=1}^n ij$;
3. $\sum_{1 \leq i < j \leq n} ij$.

4.2 Boucle while

La syntaxe d'une boucle **while** en OCaml est :

```
while condition do
  expression
done;
```

Remarque

condition est de type **bool** et expression de type **unit**.

Par exemple, une fonction `inverse_fact : int -> int` telle que `inverse_fact n` renvoie le premier entier k tel que $k! \geq n$:

```
let inverse_fact n =
  let k = ref 0 in
  let f = ref 1 in
  while !f < n do
    k := !k + 1;
    f := !f * !k
  done;
  !k
```

Exercice 3.2

Écrire une fonction `puissance_inf : int -> int` prenant en entrée un entier n supposé strictement positif et renvoyant la plus grande puissance de 2 inférieure ou égale à n :

```
# puissance_inf 1;;
- : int = 1
# puissance_inf 3;;
- : int = 2
# puissance_inf 7;;
- : int = 4
# puissance_inf 8;;
- : int = 8
```

4.3 Itération sur les éléments d'une structure de données

En Python, il est souvent bien plus pratique d'itérer directement sur les éléments d'une liste plutôt que de passer par les indices ; on peut même en profiter pour déconstruire les éléments le cas échéant :

```
#####
## Code en Python !!! ##
#####

def affiche(liste):
    s = 0
    for (x, y) in liste:
        print("Somme : ", x + y)
        s = s + x + y
    print("Total : ", s)

#####
## Fin du code en Python ##
#####
```

Cette fonction prend une liste de couples de nombres et affiche la somme de chacun des couples puis la somme totale.

La manière typique d'écrire cela en OCaml serait :

```
let affiche liste =
  let s = ref 0 in
  let affiche_couple (x, y) =
    print_endline ("Somme : " ^ string_of_int (x + y));
    s := !s + x + y in
  List.iter affiche_couple liste;
  print_endline ("Total : " ^ string_of_int !s)
```

La boucle `for (x, y) in liste:` a été remplacée par un appel à `List.iter`. La spécification de cette fonction est :

- `List.iter : ('a -> unit) -> 'a list -> unit`
 - `List.iter f [a1; ...; an]` équivaut à `begin f a1; ...; f an; () end`.
- Autrement dit, `List.iter` appelle la fonction `f` successivement sur chacun des éléments de la liste, ce qui revient exactement à une boucle `for` dont le corps serait constitué de `f`.

Notez que le code est plus lourd en OCaml, mais principalement à cause de la méthode choisie pour l'affichage. En réalité, on écrirait plutôt :

```
open Printf
(* Pour écrire g plutôt que Printf.g pour les fonctions du module Printf *)

let affiche_bis liste =
  let s = ref 0 in
  let f (x, y) = printf "Somme : %d\n" (x + y); s := !s + x + y in
  List.iter f liste;
  printf "Total : %d\n" !s
```

5 Tableaux

5.1 Type 'a array

Les tableaux sont la principale alternative aux listes pour stocker un nombre quelconque d'éléments d'un même type. Pour un tableau `t` de `n` éléments, les indices varient de 0 à `n - 1` et l'on peut accéder directement à l'élément d'indice `i` par `t.(i)`. De plus, un tableau est une structure mutable dont on peut modifier l'élément d'indice `i` par `t.(i) <- nouvelle_valeur`.

```
# let t = [| 2.3 ; 3.4 ; -2.1 |];
val t : float array = [|2.3; 3.4; -2.1|]
# t.(0);
- : float = 2.3
# t.(1) <- 5.72;;
- : unit = ()
# t;;
- : float array = [|2.3; 5.72; -2.1|]
```

Les fonctions de manipulation des tableaux se trouvent dans le module `Array` de la bibliothèque standard, on y accède par `Array.nom_de_la_fonction`. Certaines de ces fonctions sont données ici, vous êtes invités à consulter le manuel pour découvrir les autres.

Fonction	Type	Utilisation
<code>Array.make</code>	<code>int -> 'a -> 'a array</code>	<code>Array.make n x = [x;..;x]</code> (taille n)
<code>Array.length</code>	<code>'a array -> int</code>	Donne le nombre d'éléments d'un tableau.
<code>Array.of_list</code>	<code>'a list -> 'a array</code>	Convertit une liste en un tableau.
<code>Array.to_list</code>	<code>'a array -> 'a list</code>	Convertit un tableau en une liste.
<code>Array.init</code>	<code>int -> (int -> 'a) -> 'a array</code>	<code>Array.init n f = [f 0;..;f (n-1)]</code>
<code>Array.map</code>	Voir plus bas	Comme pour les listes
<code>Array.iter</code>	Voir plus bas	Comme pour les listes
<code>Array.fold_left</code>	Voir plus bas	Comme pour les listes
<code>Array.fold_right</code>	Voir plus bas	Comme pour les listes

FIGURE 3.2 – Opérations de base sur le type array

Remarques

- La fonction `Array.length` s'exécute en temps constant (contrairement à la fonction `List.length`).
- Les fonctions `Array.map`, `Array.iter`, `Array.fold_left` et `Array.fold_right` sont des équivalents exacts des fonctions correspondantes du module `List` :
 - `Array.map` : `('a -> 'b) -> 'a array -> 'b array`, telle que
`Array.map f t` renvoie `[| f t.(0); ...; f t.(n - 1) |]`;
 - `Array.iter` : `('a -> unit) -> 'a array -> unit`, telle que
`Array.iter f t` équivaut à `begin f t.(0); ...; f t.(n - 1) end`;
 - `Array.fold_left` : `('a -> 'b -> 'a) -> 'a -> 'b array -> 'a`, telle que
`Array.fold_left f init t` renvoie `f (... (f (f init t.(0)) t.(1)) ...) t.(n-1)`;
 - `Array.fold_right` : `('b -> 'a -> 'a) -> 'b array -> 'a -> 'a`, telle que
`Array.fold_right f t` renvoie `f t.(0) (f t.(1) (... (f t.(n-1) init) ...))`.

Exercice 3.3

p. 56

- Écrire une fonction `double` : `int array -> int array` qui prend en entrée un tableau `[|x1; ...; xn|]` d'entiers et renvoie le tableau `[|2 * x1; ...; 2 * xn|]`.
- Écrire une fonction `mul` : `int -> int array -> int array` qui prend en entrée un entier `a` et un tableau `[|x1; ...; xn|]` et renvoie le tableau `[|a * x1; ...; a * xn|]`.
- Écrire une fonction `affiche` : `int array -> unit` qui prend en entrée un tableau d'entiers et affiche ces entiers dans l'ordre du tableau, un entier par ligne.

5.2 Aliasing

Attention : quand `t` est un tableau, `let u = t` fait pointer `u` vers la même zone mémoire que `t`. Ainsi, toute modification de l'un entraîne une modification de l'autre.

```
# let t = [|3; 7|];;
val t : int array = [|3; 7|]
# let u = t;;
val u : int array = [|3; 7|]
(* u et t sont *physiquement* égaux
   *)
# u.(0) <- 5;;
- : unit = ()
(* On modifie u. *)
# t;;
- : int array = [|5; 7|]
(* t a également été modifié. *)
```

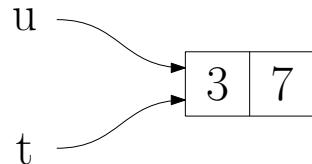


FIGURE 3.3 – Schéma-mémoire après `let u = t`.

Le problème se pose souvent quand on souhaite créer un tableau de tableaux :

```
# let t = [|0 ; 0 |];;
val t : int array = [|0; 0|]
# let u = Array.create 4 t;;
val u : int array array = [| [|0; 0|]; [|0; 0|]; [|0; 0|]; [|0; 0|] |]
(* u est maintenant constitué de 4 pointeurs vers le même tableau t. *)
# t.(1) <- 5;;
- : unit = ()
(* On modifie t. *)
# u;;
- : int array array = [| [|0; 5|]; [|0; 5|]; [|0; 5|]; [|0; 5|] |]
(* Tous les éléments de u ont été modifiés *)
# u.(2).(0) <- -1;;
- : unit = ()
(* On modifie l'un des éléments de u en place. *)
# u;;
- : int array array = [| [-1; 5|]; [| -1; 5|]; [| -1; 5|]; [| -1; 5|] |]
(* Les autres ont aussi été modifiés. *)
```

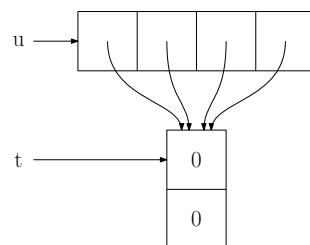


FIGURE 3.4 – Schéma-mémoire après `let u = Array.create 4 t`.

Pour initialiser correctement une matrice, il faut que chaque ligne pointe vers un tableau différent :

```
(* Renvoie une matrice n * p dont tous les éléments valent init *)
let make_matrix n p init =
  let t = Array.create n [| |] in
  for k = 0 to n - 1 do
    t.(k) <- Array.create p init
  done;
  t
```

Cette fonction existe déjà dans la bibliothèque standard : c'est
`Array.make_matrix : int -> int -> 'a -> 'a array array.`

5.3 Parcours de tableaux

Même si ce n'est pas une règle absolue, on privilégie souvent les boucles pour parcourir les tableaux de la même manière que l'on privilégie la récursion pour parcourir les listes⁵.

Deux exemples :

- la fonction `somme` : `int array -> int` calcule tout simplement la somme des éléments d'un `int array`;
- la fonction `mini` : `'a array -> 'a * int` renvoie un couple x, i où x est le minimum du tableau fourni en entrée et i l'indice de sa première occurrence.

```
let somme t =
  let s = ref 0 in
  let n = Array.length t in
  for i = 0 to n - 1 do
    s := !s + t.(i)
  done;
  !s
```

```
let mini t =
  let indice_mini = ref 0 in
  let valeur_mini = ref t.(0) in
  for i = 0 to Array.length t - 1 do
    if t.(i) < !valeur_mini then begin
      valeur_mini := t.(i);
      indice_mini := i
    end
  done;
  (!valeur_mini, !indice_mini)
```

```
# somme [|0; 4; 0; -2; 3; 1; -2|];
- : int = 4
# mini [|0; 4; 0; -2; 3; 1; -2|];
- : int * int = (-2, 3)
```

Exercice 3.4

p. 57

Écrire une version de `indice_mini` n'utilisant qu'une seule référence.

Remarque

Les cases d'un tableau sont numérotées de 0 à $n-1$ comme en Python, mais dans `for i = deb to fin`, les deux bornes sont incluses (contrairement à un `range` Python) : on écrira donc très souvent des boucles `for i = 0 to n - 1`.

5. Attention, beaucoup d'algorithmes sur les tableaux sont essentiellement récursifs. Pour un simple parcours, en revanche, une boucle est plus naturelle.

Exercices

Entraînement sur les tableaux

Exercice 3.5

p. 57

Écrire trois versions de la fonction appartient : '`a -> 'a array -> bool`' telle que `appartient x t` renvoie `true` si et seulement si l'élément `x` apparaît dans le tableau `t` :

- une utilisant une boucle `for` et parcourant le tableau jusqu'à la fin dans tous les cas ;
- une utilisant une boucle `while` et s'arrêtant dès qu'on a trouvé la valeur ;
- une récursive, n'utilisant aucune référence, et s'arrêtant dès qu'on a trouvé la valeur.

Remarque

En règle générale, on privilégiera la deuxième solution mais la troisième convient très bien aussi (et dans certains cas, c'est la plus simple à écrire).

Exercice 3.6

p. 59

Écrire une fonction `croissant` : '`a array -> bool`' qui détermine si le tableau passé en argument est croissant (au sens large). On arrêtera le parcours dès que la réponse est connue, à l'aide d'une boucle `while` (ou d'une fonction auxiliaire récursive).

Exercice 3.7

p. 60

Écrire une fonction `swap` : '`a array -> int -> int -> unit`' telle que, après l'appel `swap t i j`, les éléments de `t` d'indices `i` et `j` aient été échangés.

Exercice 3.8

p. 60

Pour une séquence $s = s_0, \dots, s_{n-1}$, on définit le *miroir* de s par $\bar{s} = s_{n-1}, \dots, s_0$. Une séquence est un *palindrome* si $s = \bar{s}$. Écrire une fonction `est_palindrome` : '`a array -> bool`' qui détermine si le tableau passé en argument est un palindrome. On essaiera de limiter au maximum le nombre de comparaisons effectuées.

```
# est_palindrome [|2; 1; 1; 3; 1; 1; 2|];
- : bool = true
# est_palindrome [|2; 1; 1; 3; 3; 1; 1; 2|];
- : bool = true
# est_palindrome [|2; 1; 1; 3; 3; 1; 2; 2|];
- : bool = false
```

Exercice 3.9

p. 60

Écrire une fonction `renverse` : '`a array -> unit`' telle que, si l'on a initialement $t = [| t_0; \dots; t_{k-1} |]$, alors, après l'appel `renverse t`, on a $t = [| t_{k-1}; \dots; t_0 |]$. On travaillera « en place » sans créer de tableau auxiliaire, et l'on fera bien attention à ce que la fonction soit correcte indépendamment de la parité de la longueur du tableau considéré.

```
# let t = [|4; 1; 2; 5|];;
val t : int array = [|4; 1; 2; 5|]
# renverse t;;
- : unit = ()
# t;;
- : int array = [|5; 2; 1; 4|]
```

Tableaux et listes

Exercice 3.10

p. 61

Écrire une fonction `occurrences` (`x : 'a`) (`t : 'a array`) : `int list` renvoyant la liste (éventuellement vide) des indices `i` tels que `t.(i) = x`. On fera en sorte que la liste renvoyée soit en ordre croissant.

```
# occurrences 7 [|0; 2; 7; 1; 7; 1; 0; 7|];
- : int list = [2; 4; 7]
```

Exercice 3.11

p. 61

1. Écrire une fonction `mini` : `'a array -> 'a` renvoyant le minimum du tableau (supposé non vide) passé en argument.
2. À l'aide de la fonction `mini` et de la fonction `occurrences`, écrire une fonction `indices_mini` : `'a array -> int list` telle que `indices_mini t` renvoie la liste des indices de `t` tels que `t.(i) = min t`, dans l'ordre croissant.
3. Écrire une nouvelle version de `indices_mini` n'effectuant qu'un seul parcours du tableau.

```
# indices_mini [|10; 2; 7; 1; 7; 1; 3; 7|];
- : int list = [3; 5]
```

Exercice 3.12

p. 62

Écrire une fonction `filtre` (`pred : 'a -> bool`) (`t : 'a array`) -> `'a list` qui renvoie la liste des `t.(i)` tels que `pred t.(i) = true`, classée dans l'ordre des `i` croissants.

```
# filtre (fun n -> n mod 2 = 0) [|1; 4; 2; 5; 7; 8; 7; 10; 4; 5|];
- : int list = [4; 2; 8; 10; 4]
```

Solutions

Correction de l'exercice 3.1 page 49

1.

```
let somme1 n =
  let s = ref 0. in
  for i = 1 to n do
    s := !s +. 1. /. float (i + n)
  done;
!s
```

2.

```
let somme2 n =
  let s = ref 0 in
  for i = 1 to n do
    for j = 1 to n do
      s := !s + i * j
    done;
  done;
!s
```

3.

```
let somme3 n =
  let s = ref 0 in
  for i = 1 to n do
    for j = i + 1 to n do
      s := !s + i * j
    done;
  done;
!s
```

Correction de l'exercice 3.2 page 49

```
let puissance_inf n =
  let puissance = ref 1 in
  while 2 * !puissance <= n do
    puissance := 2 * !puissance
  done;
!puissance
```

Correction de l'exercice 3.3 page 51

1. La version qui est sans doute la plus naturelle :

```
let double t = Array.map (fun x -> 2 * x) t
```

En fait, cette définition est de la forme `let f x = g x` et peut donc être simplifiée (ce n'est en rien indispensable) :

```
let double = Array.map (fun x -> 2 * x)
```

2. C'est à peine plus compliqué :

```
let mul a t = Array.map (fun x -> a * x) t
```

Ou en plus concis :

```
let mul a = Array.map (fun x -> a * x)
```

3. Pas de difficulté particulière :

```
let affiche t =
  let f i =
    print_int i;
    print_newline () in
  Array.iter f t
```

Si on tient à faire plus concis (ce que je ne conseille pas ici) :

```
let affiche = Array.iter (fun i -> print_int i; print_newline ())
```

En réalité, on utiliserait `printf` pour faire l'affichage, mais on en parlera quand on fera du C :

```
let affiche = Array.iter (Printf.printf "%d\n")
```

Correction de l'exercice 3.4 page 53

`valeur_mini` peut facilement être recalculé si l'on connaît `indice_mini`, on peut donc écrire :

```
let mini t =
  let indice_mini = ref 0 in
  for i = 0 to Array.length t - 1 do
    if t.(i) < t.(!indice_mini) then indice_mini := i
  done;
  (t.(!indice_mini), !indice_mini)
```

Il faut noter qu'il n'y a pas de moyen immédiat de calculer `indice_mini` à partir de `valeur_mini`, donc c'est bien `indice_mini` qu'il faut garder.

Correction de l'exercice 3.5 page 54

Pour la version avec une boucle `for`, on utilise une `bool ref` qui indique si on a trouvé l'élément :

```
let appartient x t =
  let vu = ref false in
  for i = 0 to Array.length t - 1 do
    if t.(i) = x then vu := true
  done;
!vu
```

Pour la version avec une boucle **while**, on peut adapter l'idée :

```
let appartient x t =
  let vu = ref false in
  let i = ref 0 in
  while not !vu && !i < Array.length t do
    i := !i + 1
  done;
!vu
```

En réalité, la variable vu n'est pas utile :

```
let appartient x t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n && t.(!i) <= x do
    i := !i + 1
  done;
!i < n
```

En sortie de boucle, si l'on a parcouru tout le tableau sans trouver x, alors !i vaut n et l'on renvoie donc **false**; sinon, on est sorti de la boucle « avant la fin » parce que l'on a trouvé x, et l'on renvoie **true**.

On peut aussi procéder récursivement, en définissant une fonction auxiliaire dont l'unique argument correspond à l'indice de l'élément à examiner :

```
let appartient x t =
  let n = Array.length t in
  let rec aux i =
    if i = n then false
    else if t.(i) = x then true
    else aux (i + 1)
  aux 0
```

On écrirait en fait plutôt :

```
let appartient x t =
  let n = Array.length t in
  let rec aux i =
    i < n && (t.(i) = x || aux (i + 1))
  aux 0
```

À ce moment de l'année, ce code n'est pas forcément évident à comprendre. La correction de l'exercice 3.6 utilise une technique similaire et l'explique plus en détail.

Correction de l'exercice 3.6 page 54

Version avec une boucle **while** (le plus standard) :

```
let croissant t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n - 1 && t.(!i) <= t.(!i + 1) do
    incr i (* équivaut à i := !i + 1 *)
  done;
  !i >= n - 1
```

Remarques

- La condition est $i < n - 1$ (et pas $i < n$) car le dernier test doit porter sur $t.(n - 2)$ et $t.(n - 1)$.
- On renvoie le résultat du test $i \geq n - 1$: en effet, si ce booléen vaut **true**, c'est qu'on est sorti de la boucle à cause de la première condition, et que le tableau est donc croissant. S'il vaut **false**, c'est qu'on a trouvé une valeur de i telle que $t.(!i) > t.(!i + 1)$, et donc que le tableau n'est pas croissant.
- Si l'on remplace la dernière ligne par $i = n - 1$, la fonction renvoie un résultat incorrect dans un cas particulier : lequel ?

Version récursive, qui est presque plus simple :

```
let croissant t =
  let n = Array.length t in
  let rec aux i =
    if i >= n - 1 then true
    else if t.(i) > t.(i + 1) then false
    else aux (i + 1) in
  aux 0
```

On peut en fait être plus concis :

```
let croissant t =
  let n = Array.length t in
  let rec aux i =
    i >= n - 1 || (t.(i) <= t.(i + 1) && aux (i + 1)) in
  aux 0
```

Il n'est pas forcément évident de prime abord que cette fonction fait bien ce que l'on veut, et encore moins qu'elle le fait de la manière souhaitée, c'est-à-dire en s'arrêtant dès que possible. C'est pourtant bien le cas, et cela tient au caractère séquentiel des opérateurs **||** et **&&** :

- si $i \geq n - 1$, alors le premier opérande du **||** s'évalue à **true** et la fonction **aux** renvoie donc **true** sans évaluer le reste ;
- sinon, si $t.(i) > t.(i + 1)$, alors le premier opérande du **&&** s'évalue à **false** et **aux** renvoie donc **false** sans évaluer le deuxième opérande, et donc sans effectuer d'appel récursif (on arrête donc le parcours) ;
- sinon, on effectue l'appel **aux (i + 1)** (autrement dit, on continue le parcours du tableau).

Correction de l'exercice 3.7 page 54

```
let swap t i j =
  let tmp = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- tmp
```

Correction de l'exercice 3.8 page 54

On parcourt le tableau par les deux bouts tant que les éléments correspondent, si les deux extrémités « se rencontrent » c'est qu'on a bien un palindrome :

```
let est_palindrome t =
  let i = ref 0 in
  let j = ref (Array.length t - 1) in
  while !i < !j && t.(!i) = t.(!j) do
    incr i;
    decr j (* équivaut à j := !j - 1 *)
  done;
  !i >= !j
```

En fait, une seule référence suffit puisque dans le code ci-dessus on a un invariant $j = n - 1 - i$ (avec n la longueur du tableau) :

```
let est_palindrome t =
  let i = ref 0 in
  let n = Array.length t in
  while !i < n - 1 - !i && t.(!i) = t.(n - 1 - !i) do
    incr i;
  done;
  !i >= n - 1 - !i
```

Honnêtement, cette version est moins lisible : mieux vaut privilégier la première.

Correction de l'exercice 3.9 page 54

On peut écrire une boucle **for** mais il faut être très attentif aux bornes :

```
let renverse t =
  let n = Array.length t in
  for i = 0 to n / 2 - 1 do
    swap t i (n - 1 - i)
  done
```

Honnêtement, il est moins risqué d'utiliser une boucle **while**, ou une version récursive :

```
let renverse t =
  let rec aux deb fin =
    if deb < fin then begin
      swap t deb fin;
      aux (deb + 1) (fin - 1)
    end in
  aux 0 (Array.length t - 1)
```

Correction de l'exercice 3.I0 page 55

Les tableaux se prêtent bien à la programmation impérative, les listes à la programmation fonctionnelle : quand on mélange les deux, il faut faire un choix.

- Version impérative :

```
let occurrences x t =
  let liste_occs = ref [] in
  for i = Array.length t - 1 downto 0 do
    if t.(i) = x then liste_occs := i :: !liste_occs
  done;
  !liste_occs
```

On remarquera que l'on effectue la boucle « à l'envers » de manière à ce que la liste soit construite naturellement dans le bon ordre. Il serait tout à fait acceptable d'utiliser une boucle **for** *i* = 0 **to** ... et de renvoyer **List.rev** !list_occs; en revanche, la solution consistant à effectuer la boucle dans le sens croissant et à faire un **liste_occs := !liste_occs @ [i]** dans son corps est à proscrire, pour des raisons d'efficacité (la complexité est alors de l'ordre de n^2 au lieu d'être de n).

- Version récursive :

```
let occurrences x t =
  let rec aux i =
    if i = Array.length t then []
    else if t.(i) = x then i :: aux (i + 1)
    else aux (i + 1) in
    aux 0
```

Il est important de bien comprendre pourquoi la liste est ici construite dans l'ordre souhaité.

Correction de l'exercice 3.II page 55

1.

```
let mini t =
  let m = ref t.(0) in
  for i = 1 to Array.length t - 1 do
    m := min !m t.(i)
  done;
  !m
```

2. Il s'agit juste de combiner les deux fonctions :

```
let indices_mini t =
  occurrences (mini t) t
```

3.

Correction de l'exercice 3.12 page 55

```
let filtre pred t =
  let rec aux k =
    if k = Array.length t then []
    else if pred t.(k) then t.(k) :: aux (k + 1)
    else aux (k + 1) in
  aux 0
```

CORRECTION, TERMINAISON

Beware of bugs in the above code; I have only proved it correct, not tried it.

DONALD KNUTH

I Spécification d'une fonction

La *spécification* d'une fonction, c'est le contrat qu'elle doit respecter. On peut le découper ainsi (au moins dans les cas simples) :

- Entrées :

- **Nombre et types des arguments** : par exemple, « cette fonction prend en entrée une liste t de nombres et un nombre x ». En OCaml, cette information fait partie du type de la fonction ; c'est aussi le cas (dans une moindre mesure) en C, mais en Python cela doit être spécifié séparément.
- **Préconditions** (éventuellement) : une ou des conditions qui doivent être vérifiées par les entrées pour que la fonction s'exécute correctement. Par exemple, « les éléments de la liste doivent être distincts », « la liste doit être triée par ordre croissant », « l'entier passé en argument est positif »... Si ces préconditions ne sont pas vérifiées, la fonction a un comportement non déterminé : autrement dit, elle fait ce qu'elle veut (par exemple renvoyer un résultat arbitraire, planter...¹).

- Sorties :

- **Type du résultat** : par exemple, « cette fonction renvoie un nombre », ou « cette fonction renvoie un couple formé d'un nombre et d'une liste », ou « cette fonction ne renvoie rien ».
- **Valeur du résultat** : si la fonction renvoie une « vraie » valeur (son type de retour n'est pas **unit** ou son équivalent **void** en C ou **NoneType** en Python), que doit vérifier cette valeur ? Par exemple, « la fonction renvoie n! (ou n est l'argument) » ou « la fonction renvoie une liste contenant les mêmes éléments que l'argument mais triée par ordre croissant ».
- **Effets secondaires** (éventuellement) : tous les effets que la fonction a sur le monde, en dehors du renvoi de son résultat. Typiquement, modification de l'argument ou d'une variable globale, affichage, suppression de toutes les données de l'ordinateur, envoi de missiles intercontinentaux... Par exemple : « après l'exécution de la fonction, le tableau passé en argument est trié et contient les mêmes éléments qu'au départ ».

Remarques

- On peut regrouper « effets secondaires » et « valeur du résultat » en *postcondition*.
- Une fonction ne doit pas avoir d'effets secondaires autres que ceux apparaissant dans sa spécification. Par exemple, la fonction suivante n'est pas une manière acceptable de calculer le maximum d'un tableau :

```
let apres_moi_le_deluge t =
  let n = Array.length t in
  for i = 1 to n - 1 do
    t.(i) <- max t.(i) t.(i - 1)
  done;
  t.(n - 1)
```

- En pratique, on essaie souvent d'éviter les comportements « non définis » ou très problématiques. S'il y a un moyen simple et efficace de vérifier que les préconditions sont remplies, on peut donc préférer faire le test pour détecter un éventuel problème le plus tôt possible. Pour cela, la plupart des langages

1. formater le disque dur, envoyer la totalité des images présentes sur votre ordinateur à tous vos contacts de messagerie...

(dont OCaml, C et Python) permettent de rajouter des *assertions*. En OCaml, **assert** est une fonction de type **bool** -> **unit** ayant le comportement suivant :

- si condition s'évalue en **true**, alors **assert** condition renvoie juste () (et ne fait rien, donc);
- si condition s'évalue en **false**, alors **assert** condition lève une exception **Assert_failure**. Cette exception peut éventuellement être *rattrapée*² (nous verrons plus tard comment faire), mais en tout cas elle signale immédiatement le problème et évite de continuer l'exécution du programme dans un état non défini.

Par exemple, pour une fonction calculant la factorielle, on préférera écrire :

```
let rec factorielle n =
  assert (n >= 0);
  let f = ref 1 in
  for i = 1 to n do
    f := !f * i
  done;
  !f
```

Ainsi, un appel sur un $n < 0$ sera immédiatement détecté comme une erreur :

```
# factorielle (-4);;
Exception: Assert_failure ("//toplevel//", 2, 2).
```

Sans le **assert**, la fonction aurait renvoyé un résultat dénué de sens (1, en l'occurrence) sans se plaindre, ce qui aurait rendu l'erreur beaucoup plus difficile à détecter.

- Attention cependant, ce conseil n'est pas toujours possible à appliquer (la précondition peut être trop coûteuse, voire impossible, à vérifier), et il s'applique surtout à du code destiné à être utilisé « en production », par opposition à ce que l'on vous demande d'écrire dans un cadre scolaire.
Ainsi, si un énoncé vous demande d'écrire une fonction prenant en entrée « un tableau supposé trié par ordre croissant », vous n'avez pas à vérifier que c'est effectivement le cas.

2 Correction partielle, correction totale

Définition 4.1 – Terminaison d'une fonction

On dit qu'une fonction *termine* si elle renvoie un résultat en un nombre fini d'étapes quelles que soient les valeurs de ses paramètres.

Remarques

- Les valeurs admissibles des paramètres peuvent être limitées par des préconditions : une fonction prenant en entrée un **int** peut n'être définie que pour les entiers positifs, une fonction cherchant une occurrence d'un élément x dans un tableau t peut se limiter au cas où x apparaît dans t ...
- Le nombre d'étapes de calcul ne sera en général pas *borné*, puisqu'il dépend de la valeur des paramètres.

Exemple 4.1

La fonction suivante calcule $n!$ pour $n \geq 0$:

```
let rec factorielle n =
  if n = 0 then 1
  else n * factorielle (n - 1)
```

On considère qu'elle termine, car il y a une pré-condition $n \geq 0$. Cependant, si on l'appelle sur

2. Même si **Assert_failure** est typiquement une exception qu'on ne rattrape pas.

un $n < 0$, on rentre dans une boucle infinie.

Définition 4.2 – Correction partielle

Une fonction est dite *partiellement correcte* (par rapport à sa spécification) si, quelles soient les valeurs des paramètres vérifiant les préconditions :

- soit elle renvoie un résultat conforme à la spécification;
- soit elle ne termine pas.

Définition 4.3 – Correction totale

Une fonction est dite *totalement correcte* (ou simplement *correcte*) si elle est partiellement correcte et qu'elle termine.

Exemple 4.2

1. La fonction suivante est partiellement correcte, vis à vis de n'importe quelle spécification (aucun risque qu'elle ne renvoie un résultat incorrect...):

```
let rec f x = f x
```

2. La fonction suivante est censée calculer x^n pour $x \in \mathbb{Z}$ et $n \in \mathbb{N}$:

```
let rec puissance x n =
  if n = 1 then x
  else x * puissance x (n - 1)
```

Elle est partiellement correcte, mais pas totalement correcte : en effet, $\text{puissance } x \ 0$ ne termine pas.

3 Programmes itératifs

3.1 Terminaison

La terminaison d'un programme n'utilisant que des boucles **for** est immédiate (à condition bien sûr qu'il s'agisse de « vraies » boucles **for**, ce qui est le cas en OCaml). En présence de boucles **while**, prouver la terminaison peut être arbitrairement compliqué : en particulier, nous montrerons ultérieurement que ce problème est *indécidable*, c'est-à-dire qu'il n'existe pas d'algorithme permettant de déterminer si un programme quelconque termine. Cela n'empêche bien sûr pas de montrer la terminaison dans de nombreux cas particuliers.

Définition 4.4 – Variant de boucle

Un *variant de boucle* est une quantité :

- entière;
- minorée;
- qui décroît **strictement** à chaque passage dans une boucle.

Propriété 4.5

Si une boucle admet un variant de boucle, alors elle ne peut être infinie. Par conséquent, un programme dans lequel toutes les boucles admettent des variants termine nécessairement.

Remarque

Une quantité entière, *majorée* et qui *croît* strictement fait aussi l'affaire.

Exemple 4.3

Considérons la fonction suivante :

```
let rec log2 n =
  let i = ref 0 in
  let x = ref n in
  while !x > 1 do
    x := !x / 2;
    incr i (* équivaut à i := !i + 1 *)
  done;
!i
```

x est un variant de boucle :

- c'est un entier;
- il est minoré par 1 (tant qu'on est dans la boucle);
- en notant x' la valeur en fin d'itération, on a $x' = \lfloor x/2 \rfloor \leq x/2 < x$ puisque $x \geq 1$, donc il est strictement décroissant.

Cette fonction termine donc.

Remarque

Attention, si l'on remplace la condition de la boucle par $!x \geq 0$, la terminaison n'est plus assurée car la décroissance n'est plus stricte.

Exercice 4.4

p. 73

On considère la fonction suivante :

```
let disjoints u v =
  let iu = ref 0 in
  let iv = ref 0 in
  let nu = Array.length u in
  let nv = Array.length v in
  while !iu < nu && !iv < nv && u.(!iu) <> v.(!iv) do
    if u.(!iu) < v.(!iv) then incr iu else incr iv
  done;
  !iu = nu || !iv = nv
```

1. iu est-il un variant de boucle ? Même question pour iv .
2. Identifier un variant de boucle.
3. Quelle pré-condition doit être vérifiée par u et v pour que cette fonction soit « correcte » (il faut bien sûr commencer par préciser ce que *correcte* signifie ici, en s'aidant du nom de la fonction).

Les variants de boucle ne sont pas le seul outil : fondamentalement, tout type de raisonnement peut être utilisé pour prouver la terminaison d'une fonction.

Exemple 4.5

Considérons la fonction ci-contre.
Une récurrence simple montre qu'après k passages dans la boucle, i vaut k et f vaut $k!$. Comme $k! \rightarrow +\infty$, il est alors « clair » que ce programme termine pour toute valeur de n .

```
let inv_fact n =
  let i = ref 0 in
  let f = ref 1 in
  while !f <= n do
    incr i; f := !f * !i
  done;
  !i
```

On peut éventuellement remarquer que, en plus d'être clair, c'est faux : si n vaut `max_int`, il est clair (et vrai, cette fois) que le programme ne terminera pas. Cependant, on supposera sauf mention explicite du contraire que l'on travaille avec un modèle idéal des entiers dans lequel on n'a jamais de problème de dépassement de capacité.

Exemple 4.6

Considérons maintenant le programme :

```
let syracuse n =
  let k = ref n in
  let i = ref 0 in
  while !k <= 1 do
    i := !i + 1;
    if !k mod 2 = 0 then k := !k / 2
    else k := 3 * !k + 1
  done;
  !i
```

Si vous arrivez à montrer qu'il termine pour toute valeur de n (en faisant comme si le type `int` correspondait exactement aux entiers mathématiques), faites-moi signe.

3.2 Correction

Les preuves de correction simples de programmes itératifs reposent sur le principe d'*invariant de boucle*, idée très similaire à celle d'une récurrence mathématique.

Définition 4.6 – Invariant de boucle

Un *invariant de boucle* est une propriété (un *prédicat*) qui :

- est vraie avant la première itération d'une boucle initialisation
- reste vraie à la fin d'une itération, en supposant qu'elle soit vraie au début. héritage

Remarques

- On notera souvent x' la valeur de la variable x en fin d'itération et I' l'invariant en fin d'itération.
- Pour une boucle *conditionnelle* (boucle `while`) avec une condition P et un invariant I , l'héritage revient à prouver $P \wedge I \Rightarrow I'$ (« si la condition de boucle et l'invariant sont vérifiés en début d'itération, alors l'invariant est vérifié en fin d'itération »). En sortie de boucle, on aura alors $\neg P \wedge I$ (la condition de boucle est fausse et l'invariant est vrai).
- Pour une boucle *inconditionnelle* (boucle `for`), il y a une gestion implicite de l'indice de boucle. On considérera alors que :
 - l'initialisation se fait avec la valeur de départ de l'indice ;
 - l'indice est mis à jour juste avant de passer à l'itération suivante ;

- en sortie de boucle, l'invariant est vérifié pour la valeur de l'indice suivant immédiatement la borne supérieure (si la boucle va jusqu'à $i = n - 1$, l'invariant est vérifié pour $i = n$).
- Comme pour la terminaison, les preuves de correction peuvent être arbitrairement difficiles (la correction d'un programme peut dépendre d'une conjecture mathématique, par exemple).

Exemple 4.7

On souhaite montrer que le programme suivant renvoie bien un indice du minimum de t . Pour cela, on considère l'invariant de boucle $P(i)$: « $m = \min(t_0, \dots, t_{i-1})$ et $t.(!ind) = m$ ». On note n la longueur de t .

```
let ind_min t =
  let ind = ref 0 in
  let m = ref t.(0) in
  for i = 1 to Array.length t - 1 do
    if t.(i) < !m then begin
      m := t.(i);
      ind := i
    end
  done;
!ind
```

- $P(1)$ est vérifié car, au début de la première itération, on a $ind = 0$ et $m = t_0 = \min(t_0)$.
- En supposant $P(i)$ (pour $1 \leq i \leq n - 1$), on a deux cas :
 - si $t_i < \min(t_0, \dots, t_{i-1}) = m$, alors on aura en fin d'itération $m' = t_i$ et $ind' = i$, ce qui est correct puisqu'alors $\min(t_0, \dots, t_i) = t_i$;
 - sinon, on a $\min(t_0, \dots, t_i) = \min(t_0, \dots, t_{i-1}) = m = t_{ind}$. Or on ne fait rien dans ce cas et l'on a donc $m' = m$ et $ind' = ind$, ce qui est bien correct.

À la fin de l'exécution, on a donc $P(n)$ vraie, c'est-à-dire $m = \min(t_0, \dots, t_{n-1}) = \min t$ et $t.(ind) = m$, et ind est donc bien un indice du minimum de t .

En pratique, dans un cas aussi simple, on se contentera (au mieux) de donner l'invariant de boucle sans démonstration. Dans des cas plus compliqués, en revanche, l'invariant de boucle est indispensable.

3.3 Exemples fondamentaux

Les deux algorithmes présentés ici sont à connaître absolument.

► Exercice 4.8 – Exponentiation rapide, version itérative

p. 73

On considère la fonction `expo` : `int -> int -> int`, dont la spécification est :

Précondition $n \geq 0$

Résultat $\text{expo } a \ n = a^n$

```
let expo a n =
  let p = ref n in
  let x = ref 1 in
  let b = ref a in
  while !p <> 0 do
    if !p mod 2 = 1 then x := !x * !b;
    b := !b * !b;
    p := !p / 2
  done;
!x
```

1. Montrer que $x \cdot b^p = a^n$ est un invariant de boucle.
2. En déduire la correction partielle de la fonction.

3. Montrer la correction totale de la fonction.

► Exercice 4.9 – Recherche dichotomique

p. 73

On considère l'algorithme suivant :

Entrées un tableau $t = (t_0, \dots, t_{n-1})$ et une valeur x

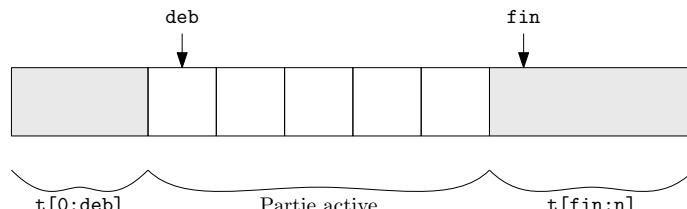
Précondition t est trié par ordre croissant

Résultat un indice $i \in [0 \dots n - 1]$ tel que $t_i = x$ s'il en existe un, n sinon.

Algorithme 1 Recherche dichotomique dans un tableau trié

```

fonction RECHERCHE( $x, t$ )
    deb  $\leftarrow 0$ 
    fin  $\leftarrow n$ 
    tant que fin – deb  $> 0$  faire
        milieu  $\leftarrow (\text{deb} + \text{fin})/2$                                  $\triangleright$  Division entière
        si  $t_{\text{milieu}} = x$  alors
            renvoyer milieu
        sinon si  $t_{\text{milieu}} < x$  alors
            deb  $\leftarrow \text{milieu} + 1$ 
        sinon
            fin  $\leftarrow \text{milieu}$ 
        renvoyer n
    
```



1. Montrer que cet algorithme termine.
2. Montrer que si l'algorithme renvoie un indice $i \neq n$, alors ce résultat est correct.
3. On note $t[a : b] = (t_a, t_{a+1}, \dots, t_{b-1})$ (attention, on va jusqu'à l'indice b exclu). Montrer qu'on a l'invariant suivant : « $x \notin t[0 : \text{deb}] \cup t[\text{fin} : n]$ ».
4. En déduire la correction de l'algorithme.
5. L'algorithme reste-t-il correct (totalement ou partiellement, on précisera) si :
 - a. on remplace la ligne $\text{deb} \leftarrow \text{milieu} + 1$ par $\text{deb} \leftarrow \text{milieu}$?
 - b. on remplace la ligne $\text{fin} \leftarrow \text{milieu}$ par $\text{fin} \leftarrow \text{milieu} - 1$?

4 Programmes récursifs

Une première remarque s'impose : il n'y a pas de différence fondamentale entre un programme écrit à l'aide de boucles ou de manière récursive. En effet, il est facile de remplacer toutes les boucles **while** par des récursions, et toujours possible (mais pas facile, en règle générale) de remplacer la récursion par une boucle **while**. En un certain sens, tout ce qui a été dit sur les programmes itératifs s'applique donc aux programmes récursifs (en particulier le caractère arbitrairement difficile des preuves de terminaison).

4.1 Principe général

En première approche, la terminaison d'un programme récursif repose sur l'existence d'un certain nombre (potentiellement infini) de cas de base et sur la certitude que toute suite d'appels finit par arriver

sur l'un de ces cas de base.

Le cas le plus simple, et le plus fréquent, est celui d'un programme ayant une définition de ce type :

- $f(0)$ donné ;
- $\forall n \in \mathbb{N}, f(n + 1) = g(n, f(n))$ où g est une fonction que l'on sait calculer.

Il est alors immédiat de prouver la terminaison par récurrence, et souvent possible de prouver simultanément la correction.

Exercice 4.10

p. 74

Montrer que le programme suivant termine et calcule $n!$, pour tout entier $n \geq 0$.

```
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
```

Fondamentalement, ce qui rend possible la récurrence est la décroissance stricte de l'argument au cours des appels récursifs. Comme \mathbb{N} n'admet pas de suite infinie strictement décroissante, on peut alors conclure que la suite des appels se termine. En pratique, on justifiera cette décroissance et l'on ne rédigera pas la récurrence (évidemment, si l'exemple est aussi simple que celui qui précède, on ne justifiera rien du tout). C'est l'équivalent d'un variant de boucle dans un programme impératif.

La correction d'un programme récursif est parfois nettement plus simple à prouver que celle d'un programme itératif équivalent. En effet, la correspondance entre le code et les identités mathématiques qui le justifient peut apparaître beaucoup plus clairement dans le cas récursif :

► Exemple 4.II – Exponentiation rapide, version récursive

```
let rec expo x n =
  if n = 0 then 1
  else if n mod 2 = 0 then expo (x * x) (n / 2)
  else x * expo x (n - 1)
```

- $0 \leq n - 1 < n$ et $0 \leq n/2 < n$ (si $n > 0$), donc n décroît strictement au cours des appels jusqu'à être nul : la fonction termine.
- Les identités $x^0 = 1$, $x^{2k} = (x^2)^k$ et $x^n = x \cdot x^{n-1}$ si $n \geq 1$ sont correctes, donc la fonction aussi.

Il arrive souvent que la suite des arguments ne soit pas strictement décroissante, soit parce que cela n'a pas de sens (si ce sont des listes par exemple), soit parce que c'est simplement faux. Dans ce cas, on peut chercher une fonction φ de l'ensemble des arguments dans \mathbb{N} dont les valeurs décroissent strictement au cours des appels successifs. Autrement dit, on cherche une fonction φ à valeurs dans \mathbb{N} telle que, dès que le programme f contient un appel du type $f(x) \rightarrow f(y)$, on ait $\varphi(y) < \varphi(x)$.

Exercice 4.12

p. 74

Justifier la terminaison de la fonction suivante :

```
let rec f n =
  if (n mod 2 <> 0) || (n = 0) then n
  else f (3 * n / 2)
```

4.2 Fonctions mutuellement récursives

Les définitions données jusqu'à présent peuvent reposer sur plusieurs récursions, mais ces récursions sont définies successivement. Ainsi, on peut par exemple définir une première fonction f récursive, puis définir g en posant $g(n) = f(g(n - 1))$ (et les cas de base adéquats). Il est également possible de définir

des fonctions *mutuellement récursives* : l'exemple le plus simple serait

$$\begin{cases} f(0) = 1 \\ g(0) = 0 \\ \forall n \in \mathbb{N}, f(n+1) = g(n) \\ \forall n \in \mathbb{N}, g(n+1) = f(n) \end{cases}$$

Pour programmer un tel couple de fonctions en OCaml, on utilise le mot-clé **and** :

```
let rec f n =
  if n = 0 then 1 else g (n - 1)
and g n =
  if n = 0 then 0 else f (n - 1)
```

En remplaçant les 0 et les 1 par des booléens, on peut être plus concis :

```
let rec f n = (n = 0) || g (n - 1)
and g n = (n <> 0) && f (n - 1)
```

Exercice 4.13

De quelles fonctions élémentaires f et g sont-elles des implémentations tordues et inefficaces ?

Nous n'aurons pas très souvent besoin de définir des fonctions mutuellement récursives, mais c'est quand même nécessaire de temps en temps.

Exercices Supplémentaires

Solutions

Correction de l'exercice 4.4 page 66

1. Lors d'un passage dans la boucle, iu peut rester inchangé (si $u.(!iu) \geq v.(!iv)$), donc iu n'est ni strictement croissant ni strictement décroissant : ce n'est pas un variant de boucle. De même pour iv .
2. À chaque passage dans la boucle, soit iu soit iv est incrémenté et l'autre est inchangé, donc $iu + iv$ augmente de 1 : cette quantité est strictement croissante. De plus, $iu + iv$ est majoré par $nu + nv$, donc $iu + iv$ est un variant de boucle.
3. Il faut que u et v soient triés par ordre croissant. Si c'est le cas, alors `disjoint u v` renvoie `true` si et seulement si u et v n'ont aucun élément en commun.

Correction de l'exercice 4.8 page 68

1. **Initialisation** Au départ, on a $x \cdot b^p = 1 \cdot a^n = a^n$, c'est bon.

Hérité On suppose $x \cdot b^p = a^n$ et l'on distingue les cas :

- si p est pair, $p = 2k$, alors

$$\begin{cases} x' = x \\ b' = b^2 \\ p' = k \end{cases}$$

On a donc $x' \cdot b'^{p'} = x \cdot (b^2)^k = x \cdot b^{2k} = a^n$, c'est bon.

- si p est impair, $p = 2k + 1$, alors

$$\begin{cases} x' = xb \\ b' = b^2 \\ p' = k \end{cases}$$

On a donc $x' \cdot b'^{p'} = xb \cdot (b^2)^k = xb^{2k+1} = a^n$, c'est bon.

Donc $x \cdot b^p = a^n$ est un invariant de boucle.

2. En sortie de boucle, on a $p = 0$ (condition de boucle) et $x \cdot b^p = a^n$ (invariant). On en déduit $x \cdot b^0 = a^n$, c'est-à-dire $x = a^n$. Or la fonction renvoie x , elle est donc partiellement correcte.

3. p est un variant de boucle. En effet :

- p est un entier ;
- au départ, $p = n \geq 0$ d'après la précondition ;
- si la condition de boucle est vérifiée, alors $p > 0$ et on a donc $p' = \lfloor p/2 \rfloor < p$, et $p' \geq 0$;
- p est donc entier, minoré par zéro et strictement décroissant : c'est un variant de boucle.

Ainsi la fonction termine, donc d'après la question précédente elle est totalement correcte.

Correction de l'exercice 4.9 page 69

1. Montrons que $fin - deb$ est un variant de boucle.

- C'est bien un entier.
- La condition de boucle assure $fin - deb > 0$ tant qu'on est dans la boucle, donc il est minoré.
- On a donc $deb \leq fin - 1$, d'où $deb < \frac{deb+fin}{2} < \frac{2fin-1}{2}$ puis $deb \leq \lfloor \frac{deb+fin}{2} \rfloor \leq fin - 1$ en appliquant la fonction partie entière qui est croissante (attention, les inégalités

deviennent larges). Ainsi, on a $\text{deb} \leq \text{milieu} < \text{fin}$. Or :

- soit on sort de la boucle ;
- soit $\text{deb}' = \text{milieu} + 1 > \text{deb}$ et $\text{fin}' = \text{fin}$, d'où $\text{fin}' - \text{deb}' < \text{fin} - \text{deb}$;
- soit $\text{deb}' = \text{deb}$ et $\text{fin}' = \text{milieu} < \text{fin}$, d'où encore $\text{fin}' - \text{deb}' < \text{fin} - \text{deb}$.

$\text{fin} - \text{deb}$ est donc bien un variant de boucle, et la fonction termine.

2. Le seul moyen de renvoyer autre chose que n est de passer le test $t_{\text{milieu}} = x$ et de renvoyer milieu immédiatement, ce qui est évidemment correct.

3. Initialisation Avant la première itération, on a $\text{deb} = 0$ et $\text{fin} = n$, d'où $t[0 : \text{deb}] = t[\text{fin} : n] = \emptyset$: l'invariant est trivialement vérifié.

Héritéité On suppose $x \notin t[0 : \text{deb}] \cup t[\text{fin} : n]$.

- Si $t[\text{milieu}] = x$, alors on sort de la fonction et la question ne se pose plus.
- Si $t[\text{milieu}] < x$, alors $x \notin t[\text{deb} : \text{milieu} + 1]$ puisque le tableau est trié. Donc $x \notin t[0 : \text{deb}] \cup t[\text{deb} : \text{milieu} + 1] \cup t[\text{fin} : n] = t[0 : \text{deb}'] \cup t[\text{fin} : n]$.
- Si $t[\text{milieu}] > x$, alors $x \notin t[\text{milieu} : \text{fin}]$ (le tableau est trié). Donc $x \in t[0 : \text{deb}] \cup t[\text{milieu} : \text{fin}] \cup t[\text{fin} : n] = t[0 : \text{deb}] \cup t[\text{fin}' : n]$.

La propriété est donc bien un invariant de boucle.

4. Si l'on sort de la boucle sans être passé par la ligne « renvoyer milieu », alors on a $\text{fin} \geq \text{deb}$ et $x \notin t[0 : \text{deb}] \cup t[\text{fin} : n]$, donc $x \notin t$. On renvoie n , c'est correct. De plus, on a vu qu'un résultat autre que n était forcément correct et que la fonction terminait : la fonction est donc correcte.

5. a. L'algorithme reste partiellement correct après cette modification (on réduit moins l'espace de recherche, ce qui ne peut pas causer de résultat faux). En revanche, il ne termine plus, par exemple pour $x = 3$ et $t = (2)$.

b. Cette fois, l'algorithme termine (on réduit davantage l'espace de recherche à chaque itération). En revanche, il n'est plus correct : `RECHERCHE(20, (10, 20, 30, 40, 50))` renvoie 5 (il ne trouve pas l'élément) au lieu de 1.

Correction de l'exercice 4.10 page 70

Posons $H(n)$: « `fact n` termine et renvoie $n!$ ».

Initialisation `fact 0` termine en renvoyant 1, et $0! = 1$, c'est bon.

Héritéité On suppose $H(n)$ pour un certain $n \geq 0$. On a alors `fact (n + 1) = n * fact n` d'après le code. Or `fact n` termine et renvoie $n!$, donc `fact (n + 1)` termine et renvoie $(n + 1) \cdot n! = (n + 1)!$.

On conclut donc que pour tout $n \geq 0$, l'appel `fact n` termine et renvoie $n!$.

Correction de l'exercice 4.12 page 70

Il faut considérer l'exposant de 2 dans la décomposition en facteurs premiers de n (ce qu'on appelle la *2-valuation* de n), que l'on va noter $\varphi(n)$.

- Si n est impair ou nul, la fonction termine immédiatement.
- Sinon, on a $n = 2k$ avec $\varphi(k) = \varphi(n) - 1$ et $\varphi(\frac{3n}{2}) = \varphi(3k) = \varphi(k) = \varphi(n) - 1$.
- Ainsi, $\varphi(n)$ est un entier naturel qui décroît strictement au cours des appels et la fonction termine quand il atteint zéro : on peut conclure que la fonction termine.

INTRODUCTION À LA COMPLEXITÉ

I Notations mathématiques

Définition 5.1

Soient u et v deux suites réelles. On note :

- $u = O(v)$ s'il existe $A > 0$ et $n_0 \in \mathbb{N}$ tels que $|u_n| \leq A |v_n|$ pour $n \geq n_0$.
- $u = \Omega(v)$ s'il existe $A > 0$ et $n_0 \in \mathbb{N}$ tels que $|u_n| \geq A |v_n|$ pour $n \geq n_0$.
- $u = \Theta(v)$ s'il existe $A, B > 0$ et $n_0 \in \mathbb{N}$ tels que $A |v_n| \leq |u_n| \leq B |v_n|$ pour $n \geq n_0$.

Remarques

- $u = \Theta(v)$ équivaut à $(u = O(v))$ et $(v = O(u))$.
- Si l'on ne considère que des suites à valeurs strictement positives (restriction qui ne pose aucun problème pour les études de complexité), alors :
 - $u_n = O(v_n)$ si et seulement si $\exists A \in \mathbb{R} \forall n \in \mathbb{N}, u_n \leq A v_n$
 - $u_n = \Omega(v_n)$ si et seulement si $\exists A \in \mathbb{R} \forall n \in \mathbb{N}, u_n \geq A v_n$
 - $u_n = \Theta(v_n)$ si et seulement si $\exists A, B \in \mathbb{R}_+^* \forall n \in \mathbb{N}, A v_n \leq u_n \leq B v_n$
- Remarquez qu'il n'y a plus de « à partir d'un certain rang » ; dans les calculs, ces formulations sont souvent plus pratiques à manipuler (et moins sujettes à des erreurs de raisonnement).
- Le symbole « égal » n'a pas vraiment sa signification habituelle ici, et il serait plus logique de noter $u \in O(v)$ (mais on le fait rarement). En particulier, si $u = O(w)$ et $v = O(w)$, on ne peut *rien conclure* sur u par rapport à v .
- Il faut bien retenir les interprétations intuitives :
 - $u = O(v)$ signifie « u est majorée par v à une constante multiplicative près » ou « u est au plus de l'ordre de grandeur de v » ;
 - $u = \Theta(v)$ signifie « u et v sont du même ordre de grandeur ».
- La relation grand- Θ est *symétrique* : $u = \Theta(v)$ si et seulement si $v = \Theta(u)$. Ce n'est pas du tout le cas de grand- O .

Exemple 5.1

- $\ln n = \Theta(\log_2 n)$
- $12n^2 + 3n \log n - n = O(n^2)$
- $12n^2 = O(n^{17})$
- $12n^2 = \Theta(n^2)$
- $12n^2$ n'est pas un grand- Θ de n^{17}
- $17 + (-1)^n + \frac{\log n}{n} = O(1)$

2 Types de ressources, niveau de détail

Étudier la complexité d'un algorithme, c'est s'intéresser aux ressources qu'il consomme pour effectuer sa tâche, et plus précisément à la manière dont cette consommation évolue quand la taille des données augmente.

Les principales ressources auxquelles on peut s'intéresser sont :

- **le temps et l'espace**, les deux seules qui nous concerteront ;
- **l'énergie**, qui prend une importance de plus en plus grande à cause de son impact sur l'autonomie (principalement dans les téléphones) et sur le coût monétaire des calculs (principalement à l'échelle d'un *datacenter*) ;
- **les données échangées sur le réseau**, qui peuvent être le facteur limitant si l'on fait du calcul distribué... .

L'autre axe suivant lequel varie la notion de complexité est celui du niveau de détail souhaité.

- Tout d'abord, l'étude de la complexité se fait le plus souvent de manière **asymptotique**, c'est-à-dire en faisant tendre la taille des données vers l'infini. De nombreuses questions intéressantes en pratique ne rentrent pas dans ce cadre (même si l'étude asymptotique peut orienter sur la bonne voie) :
 - est-ce que je vais arriver à produire une image toutes les 16ms sur un matériel précis (une PS5, disons) ?
 - est-ce que je peux *prouver* qu'il ne s'écoulera jamais plus de 50ms entre l'acquisition des données par les capteurs et l'envoi du signal de commande aux actionneurs de mon nouvel avion ?
 - je reçois des transactions à traiter par paquets de mille, comment traiter un tel paquet le plus efficacement possible ?
 - Même en se limitant à l'étude asymptotique, il reste beaucoup de variations.
 - En théorie de la complexité, des questions typiques sont :
 - cet algorithme est-il en temps polynomial ? De ce point de vue, le tri rapide, le tri fusion et le tri par insertion se valent... .
 - cet algorithme est-il en temps élémentaire (c'est-à-dire majoré par une tour d'exponentielles de hauteur bornée) ? Si la réponse est non, il est temps de s'inquiéter... .
 - Si l'on veut plus de détails, on peut simplement ignorer toutes les constantes multiplicatives. On considère alors que toutes les opérations élémentaires se valent (ajouter deux entiers machine, prendre le logarithme d'un flottant, accéder à une case d'un tableau...), et on cherche l'*ordre de grandeur* du nombre total d'opérations effectuées en fonction de la taille n des données. On dira alors que le tri fusion a une complexité temporelle en $\Theta(n \log n)$ et le tri insertion en $\Theta(n^2)$ (dans le pire des cas).
- C'est le point de vue du programme, et ce sera donc le nôtre.**
- On peut bien sûr donner des résultats asymptotiques plus précis. Typiquement, on comptera quelques types d'opérations élémentaires et l'on cherchera un équivalent, voire mieux. Par exemple, on peut prouver qu'en moyenne, sur une permutation aléatoire de $[0 \dots n - 1]$, le tri insertion effectue $\frac{n^2}{4} + \frac{3n}{4} - \ln n + O(1)$ comparaisons.

Si la complexité temporelle (ou spatiale) d'un algorithme est en $\Theta(f(n))$, on dira que l'algorithme est en temps (ou en espace) :

- **constant** si $f(n) = O(1)$ (c'est-à-dire si $f(n)$ est majoré par une constante) ;
- **logarithmique** si $f(n)$ est de l'ordre de $\log n$;
- **poly-logarithmique** si $f(n)$ est de l'ordre de $P(\log n)$ où P est un polynôme de degré supérieur ou égal à 2 ;
- **linéaire** si $f(n)$ est de l'ordre de n ;
- **quasi-linéaire** si $f(n)$ est en $O(n^{1+\varepsilon})$ pour tout $\varepsilon > 0$ (par exemple si $f(n)$ est de l'ordre de $n \log n$) ;
- **quadratique** si $f(n)$ est de l'ordre de n^2 ;
- **polynomial** si $f(n)$ est majoré par n^k pour un certain k ;

3 Complexité en temps

Quand on s'intéresse à la complexité temporelle d'un algorithme, on cherche à évaluer comment son temps d'exécution évolue quand on fait tendre la taille des données vers l'infini. Comme le temps de calcul dépend de nombreux facteurs difficiles à contrôler (implémentation, machine, langage...), on se concentre sur le *nombre d'instructions élémentaires* exécutées. Il faut donc :

- déterminer (ou estimer, ou majorer...) le nombre de fois que chaque instruction (chaque « ligne » du programme) est exécutée;
- déterminer si chacune de ces instructions est *élémentaire* ou pas;
- pour les instructions qui ne sont pas élémentaires, estimer leur complexité;
- sommer toutes ces complexités;
- tenter éventuellement d'en tirer des conclusions sur le temps de calcul.

Supposons pour l'instant que nous avons réussi à traiter le premier point, et intéressons-nous aux autres.

3.1 Opérations élémentaires

Ce qui caractérise une opération élémentaire, c'est qu'elle s'exécute en temps constant. Quelques exemples qu'il faut connaître en OCaml :

Opérations élémentaires

- récupérer la tête et la queue d'une liste (en faisant un `match`, ou en utilisant `List.hd` et `List.tl`);
- accéder à (ou modifier) l'élément `i` d'un `array` par `t.(i)`;
- obtenir la longueur d'un `array` par `Array.length t`;
- ajouter, soustraire, multiplier, comparer... deux flottants;
- de même pour deux entiers (machine);
- effectuer un appel de fonction (on ne parle que du coût de l'appel, pas de celui de l'exécution de la fonction appelée).

Opérations non élémentaires

- accéder à l'élément `i` d'une liste (temps proportionnel à `i`);
- concaténer deux listes par `u @ v` (temps proportionnel à $|u|$);
- obtenir la longueur d'une liste par `List.length u` (temps proportionnel à $|u|$).

3.2 Complexité dans le pire des cas

On donne toujours la complexité d'un algorithme en fonction de la taille n de l'entrée. Pourtant, la plupart des algorithmes peuvent avoir un temps d'exécution très variable entre deux entrées de même taille. La fonction suivante, par exemple, s'exécute en temps constant si le premier élément de `u` vaut `x`, en temps proportionnel à $|u|$ si `x` n'apparaît pas dans `u` :

```
let rec appartient x u =
  match u with
  | [] -> false
  | y :: ys -> (x = y) || appartient x ys
```

Par défaut, nous nous intéresserons toujours à la complexité *dans le pire des cas* : autrement dit, le $T(n)$ cherché est le nombre maximum d'opérations élémentaires pour traiter une entrée de taille n (et la fonction précédente a une complexité en $O(|u|)$).

3.3 Complexité en moyenne

Certains algorithmes peuvent être très lents dans une petite proportion de cas « pathologiques » et très efficaces sur les autres. Il peut alors être intéressant de calculer la *complexité en moyenne* de l'algorithme, c'est-à-dire l'espérance du temps de calcul pour une certaine loi de probabilité sur les données.

Ce type de complexité est plus délicat à étudier que la complexité dans le pire des cas, et ce pour deux raisons :

- il n'est pas toujours évident de définir une loi de probabilité sur les données, et encore moins une loi de probabilité pertinente. Dans l'exemple précédent, quelle peut bien être la probabilité que le premier élément soit égal à x ?
- une fois que l'on a fixé la loi de probabilité, les calculs sont en général beaucoup plus délicats que pour le pire cas. Il peut même être nécessaire de faire des vraies mathématiques (majorer des queues de distribution, estimer la croissance asymptotique de coefficients de séries génératrices...).

L'exemple le plus connu, et sans doute le plus important, d'algorithme pour lequel il est pertinent de faire une analyse en moyenne est celui du **tri rapide** : en effet, il est en $\Theta(n^2)$ dans le pire des cas mais en $\Theta(n \log n)$ en moyenne si le tableau d'entrée est dans un ordre aléatoire. De plus, il est en pratique *plus efficace* que le tri fusion (qui est en $\Theta(n \log n)$ dans le pire cas). Nous traiterons cet exemple en détail ultérieurement.

3.4 Complexité amortie

Le type **list** du langage Python ressemble davantage au type **array** de OCaml qu'au type **list**. En particulier, il permet l'accès en temps constant à l'élément d'indice i . Cependant, il propose deux opérations qui sont impossibles sur un tableau OCaml (ou C) :

- `t.append(x)` modifie la **list** `t` en ajoutant l'élément `x` à la fin (la taille de la liste augmente de un);
- `t.pop()` renvoie le dernier élément de la **list** et le supprime de la liste (la taille de la liste diminue de un).

Si l'on s'intéresse à la complexité de ces opérations, on a la situation suivante (en notant n la longueur de `t`) :

- une opération `t.append` ou `t.pop` peut prendre un temps proportionnel à n (rarement) ou constant (le plus souvent);
- une série de N opérations prendra au total un temps $O(N)$, et donc un temps $O(1)$ par opération.

On dit que `append` et `pop` ont une complexité $O(n)$ dans le pire cas, mais $O(1)$ *amortie* : en effet, il est possible d'amortir dans le temps le coût des opérations lentes (car une opération lente garantit la présence d'un certain nombre d'opérations rapides). Cette notion de complexité amortie est souvent pertinente pour l'analyse des structures de données : nous aurons l'occasion d'y revenir (en particulier quand nous réaliserons des structures équivalentes aux **list** Python en OCaml et en C).

4 Calculs de complexité

Dans cette partie, on considère que l'on connaît la complexité de toutes les fonctions prédéfinies qu'on utilise (on sait donc que `Array.length` est en $\Theta(1)$ et que `List.length` est en $\Theta(n)$, par exemple), et que l'on souhaite calculer la complexité temporelle dans le pire cas d'une fonction. Autrement dit, on cherche une estimation asymptotique du nombre $T(n)$ d'opérations effectuées dans le pire cas, sous la forme d'un Θ (idéalement), ou d'un « bon » grand-O.

Remarque

Si vous prouvez rigoureusement que la complexité du tri fusion est en $O(2^{n!})$, vous n'avez certes pas commis d'erreur mais vous n'avez pas non plus obtenu de points...

4.1 Algorithmes itératifs

4.1.a Boucles `for`

Il faut simplement sommer le nombre d'opérations pour chacune des itérations de la boucle. Par exemple :

```

1 let exemple (t : int array) =
2   let n = Array.length t in
3   let f i = ... in
4   for i = 0 to n - 1 do
5     let x = f i in
6     t.(i) <- x + i
7   done

```

La ligne 2 n'est exécutée qu'une seule fois, et prend un temps unitaire. Les lignes 5 et 6 (le *corps* de la boucle) sont exécutées n fois chacune. La ligne 6 est une opération élémentaire, mais la ligne 5 contient un appel à une fonction inconnue f avec le paramètre i : le nombre total d'instructions exécutées dans cette boucle est donc $\Theta\left(\sum_{i=0}^{n-1}(1 + \varphi(i))\right)$, où $\varphi(i)$ correspond au nombre d'instructions pour le calcul de $f i$.

- Si f s'exécute en temps constant, par exemple `let f i = i + 1`, alors chaque itération est en $\Theta(1)$ et on obtient au total $\Theta(n)$.
- Le temps d'exécution de $f i$ pourrait aussi dépendre de n , tout en restant indépendant de i . Par exemple :

```
let f i =
  let x = ref 0 in
  for j = 0 to n - 1 do
    if t.(j) < t.(i) then incr x
  done;
!x
```

Dans ce cas, chacun des termes de la somme vaut n (ou plutôt, est un $\Theta(n)$), et l'on obtient donc :

$$\Theta\left(\sum_{i=0}^{n-1}(1 + n)\right) = \Theta(n^2)$$

- Considérons maintenant la fonction f suivante, qui s'exécute en temps $\Theta(i)$:

```
let f i =
  let x = ref 0 in
  for j = 0 to i - 1 do
    if t.(j) < t.(i) then incr x
  done;
!x
```

— Le calcul de la complexité totale ne pose pas de problème :

$$\Theta\left(\sum_{i=0}^{n-1}i\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2)$$

— Comme on a toujours $i < n$, la fonction f s'exécute en temps $O(n)$. Si l'on souhaite seulement une *majoration* de la complexité totale, on peut donc écrire :

$$O\left(\sum_{i=0}^{n-1}n\right) = O(n^2)$$

Ici, on a obtenu une majoration précise : notre « grand-O » est en fait un Θ .

- Ce ne sera pas toujours le cas. Par exemple, si la fonction f s'exécute en temps $\Theta(2^i)$, alors une majoration brutale donnerait :

$$O\left(\sum_{i=0}^{n-1}2^n\right) = O(n2^n)$$

C'est correct (c'est bien une *majoration*), mais c'est grossier. En réalité, on a :

$$\Theta\left(\sum_{i=0}^{n-1}2^i\right) = \Theta(2^n - 1) = \Theta(2^n)$$

Remarques

- Il faut faire bien attention à ne pas multiplier des complexités alors qu'on devrait les ajouter. Essentiellement, on fait toujours des sommes de complexité, qui peuvent éventuellement se transformer en produits quand tous les termes sommés sont égaux (par exemple dans le cas où f était en $\Theta(n)$, ici). En particulier, si vous obtenez une complexité en $n!$, vous avez **presque certainement fait une erreur** en remplaçant une $\sum_{i=1}^n i$ par un $\prod_{i=1}^n i$.
- Attention également à bien voir la différence entre des boucles *imbriquées* et des boucles *successives*.

```

for i = 1 to n do
    instruction      (* on passe n fois ici *)
done;
for i = 1 to n do
    insruction      (* et n fois ici *)
done;
for i = 1 to n do
    for j = 1 to n do
        instruction    (* on passe n² fois ici *)
    done
done

```

Dans un cas on a deux sommes, dans l'autre une somme double.

4.1.b Boucles **while**

La seule différence avec une boucle **for** est qu'on ne connaît pas *a priori* le nombre d'itérations. Le plus souvent, on sera amené à le majorer mais il faudra faire attention à ne pas être trop grossier.

Remarque

Attention, il ne faut pas oublier de prendre en compte le temps nécessaire à évaluer la condition de la boucle. Pour prendre un exemple un peu idiot, la fonction (Python) suivante s'exécute en temps $\Theta(|u|^2)$:

```

## Code Python
def f(u, borne):
    i = 0
    while i < len(u) and sum(u[:i]) < borne: # Évaluation en O(i) !!!
        i += 1                                # Corps de la boucle en O(1)
    return i

```

Exemple 5.2

On considère les deux fonctions suivantes, et l'on note $n = |t|$:

```

let nb_zeros t i =
  let j = ref 0 in
  while i + !j < Array.length t && t.(i + !j) = 0 do
    incr j
  done;
  !j

let bloc_max_zeros t =
  let m = ref 0 in
  let i = ref 0 in
  while i < Array.length t do
    let j = nb_zeros t !i in
    m := max !m j;
    i := !i + j + 1
  done;
  !m

```

- Pour la fonction `nb_zeros`, la condition et le corps de la boucle sont en $O(1)$, et l'on fait au plus $n - i$ itérations, donc la complexité est en $O(n - i)$. Le nombre exact d'itérations dépend de la position du premier zéro, mais notre majoration est optimale dans le pire des cas.
- Pour la fonction `bloc_max_zeros`, la complexité de chaque itération est dominée par celle de l'appel à `nb_zeros`. Comme i augmente d'au moins un à chaque itération, on peut majorer grossièrement et obtenir pour la complexité totale :

$$O\left(\sum_{i=0}^{n-1} (n - i)\right) = O(n^2)$$

C'est *correct*, mais bien trop *imprécis*!

- En réalité, le temps d'exécution de `nb_zeros t !i` est proportionnel à la valeur renvoyée. Ainsi, en notant $0 = i_0, \dots, i_p = n$ les valeurs successives de i , la k -ème itération prend un temps proportionnel à $i_k - i_{k-1}$. Au total, on obtient donc :

$$O\left(\sum_{k=1}^p (i_k - i_{k-1})\right) = O(i_p - i_0) = O(n)$$

- En pratique, on rédigerait de manière moins formelle : `nb_zeros t !i` parcourt un morceau de t de taille j et temps $O(j)$, et la ligne `i := !i + j + 1` assure qu'on ne parcourt jamais deux fois la même case. Au total, on effectue donc un unique parcours de t , en temps $O(n)$.

4.2 Algorithmes récursifs

Pour la complexité temporelle, la principale différence avec le cas itératif est qu'on obtient naturellement une formule de récurrence pour la complexité, et qu'il y a donc une étape supplémentaire pour passer de cette formule de récurrence à une forme close (quand c'est possible).

Exemple 5.3

On considère le problème suivant : étant donnée une liste (u_0, \dots, u_{n-1}) , renvoyer la liste $(u_0 + n - 1, u_1 + n - 1, \dots, u_{n-1})$.

Première version

```
let rec f = function
| [] -> []
| x :: xs -> x + List.length xs :: f xs
```

Si $|u| = n$, alors un appel `f u` donne :

- un appel à `f` sur une liste de longueur $n - 1$;
- un appel à `List.length` sur une liste de longueur $n - 1$;
- des opérations en temps constant.

On en déduit que $T(n) \leq T(n - 1) + An$, avec A une constante. Il vient alors $T(n) = O(n^2)$ (en faisant apparaître une somme télescopique, par exemple), et ce O est en fait un Θ .

Deuxième version

```
let g u =
let rec aux u k =
match u with
| [] -> []
| x :: xs -> x + k :: aux xs (k - 1) in
aux u (List.length u - 1)
```

Ici, on a :

- pour aux, en notant $n = |u|$, $T(n) = T(n - 1) + A$ et donc $T(n) = \Theta(n)$;
- pour g, un appel à `List.length` en $\Theta(n)$ et un appel à aux en $\Theta(n)$, donc au total $\Theta(n)$.

En réalité, on se contenterait de dire que g effectue deux parcours « simples » de la liste (un pour aux et un pour `List.length`), et est donc en temps linéaire.

4.3 Rapport avec le temps de calcul

Après avoir analysé la complexité temporelle d'un algorithme, on dispose d'une information du type $T(n) = \Theta(n \log n)$ (par exemple). En théorie, cela ne nous dit absolument rien du temps de calcul réel pour une valeur donnée de n , mais en pratique on peut (heureusement !) en tirer quelques informations.

Valeur de la constante cachée Pour un algorithme raisonnablement simple, on peut supposer que la constante multiplicative cachée dans le Θ est de l'ordre de 10, voire de 100. Pour certains algorithmes très sophistiqués, elle peut être gigantesque et rendre l'algorithme inutilisable en pratique.

For any instance $G = (V, E)$ that one could fit into the known universe, one would easily prefer $|V|^{70}$ to even constant time, if that constant had to be one of Robertson and Seymour's.

David Johnson

Traduction d'une opération élémentaire Certaines des opérations élémentaires vues plus haut (ajouter deux flottants, par exemple) correspondent directement à une instruction processeur. D'autres sont plus complexes et correspondront à plusieurs instructions (deux ou trois pour lire la case i d'un tableau en OCaml, une bonne centaine pour faire un append en Python).

Cycle processeur Vu de l'extérieur, l'état d'un processeur évolue de manière discrète : il est dans un certain état à l'instant t_n , dans un certain état à l'instant $t_{n+1} = t_n + h$ et dans un état non défini entre ces deux instants. Ce h est la durée d'un *cycle*, c'est-à-dire l'inverse de la *fréquence d'horloge* que les constructeurs communiquent. Comme vous le savez peut-être, la fréquence d'un processeur actuel varie entre 1GHz et 5GHz, et un cycle prend donc de 0.2ns à 1ns.

Temps pour exécuter une instruction Exécuter une instruction prend au moins un cycle¹, mais peut prendre beaucoup plus longtemps. Quelques exemples :

- ajouter deux entiers, comparer deux flottants : 1 cycle ;
- une division entière : quelques dizaines de cycles ;
- accès mémoire : entre 1 et 500 cycles... .

Une recette de cuisine En étant optimiste :

- la constante cachée vaut 2 (très optimiste) ;
- chaque opération élémentaire donne 5 instructions (optimiste) ;
- on a un IPC (nombre d'instructions exécutées par cycle d'horloge) de 2 (très raisonnable) ;
- un cycle prend 0.2ns (optimiste) ;

On arrive alors à un temps de calcul de $f(n)$ nano-secondes si la complexité est en $\Theta(f(n))$. C'est possible si par exemple on programme (bien) une multiplication matricielle en C (on peut même faire mieux).

Si au contraire on travaille dans un langage « lent » et si l'algorithme se prête moins à un traitement efficace en machine, on peut facilement être mille fois plus lent. On pourra donc retenir la recette suivante :

Théorème 5.2 – Ceci n'est pas un théorème

Si la complexité temporelle est en $\Theta(f(n))$, alors le temps de calcul sera le plus souvent compris entre $f(n)$ nano-secondes et $f(n)$ micro-secondes (pour des valeurs de n « pas trop petites »).

1. Mais un processeur peut exécuter plusieurs instructions en parallèle (sur un même cœur).

	10	100	1 000	10 000	1 000 000	10^9
$\Theta(\log n)$	1ns	10ns	10ns	10ns	10ns	100ns
$\Theta(\sqrt{n})$	1ns	10ns	100ns	100ns	1μs	1ms
$\Theta(n)$	10ns	100ns	1μs	10μs	1ms	1s
$\Theta(n \log n)$	10ns	100ns	10μs	100μs	10ms	10s
$\Theta(n^2)$	100ns	10μs	1ms	100ms	10min	10 ans
$\Theta(n^3)$	1μs	1ms	1s	10 min	10 ans	∞
$\Theta(2^n)$	1μs	∞	∞	∞	∞	∞

FIGURE 5.1 – Ordre de grandeur du temps de calcul pour quelques valeurs de n et complexités usuelles.

On est ici très optimiste : il faut par exemple comprendre que pour une complexité en $\Theta(n \log n)$ avec $n = 10^6$, on prendra **au mieux quelques dizaines de milli-secondes**. Pour une version pessimiste, il faut essentiellement tout multiplier par mille (et dans les cas pathologiques par 100!!!!!, mais c'est rare).

4.4 Quelques résultats expérimentaux

Tri rapide en OCaml Le tri rapide est un algorithme de tri, que nous étudierons plus tard et qui a une complexité de $\Theta(n \log n)$ en moyenne si le tableau à trier est aléatoire. En prenant une implémentation raisonnablement efficace de cet algorithme en OCaml, on obtient pour un tableau de un million d'éléments :

Nombre d'instructions environ 450 millions, donc le A de $An \log n$ vaut à peu près 20;

Instructions par cycle environ 1.3 ;

Durée d'un cycle environ 0.3ns (ne dépend que de l'ordinateur);

Temps d'exécution environ 100ms (pourrait se déduire des trois points précédents).

Sachant que $n \log n$ vaut environ 20 millions pour $n = 10^6$, la « recette de cuisine » prédisait entre 20 millisecondes et 20 secondes : on est dans la partie basse de la fourchette.

Les graphiques ci-dessous permettent de visualiser ce qui se passe pour différentes valeurs de n : le nombre d'instructions est toujours d'environ $20n \log n$.

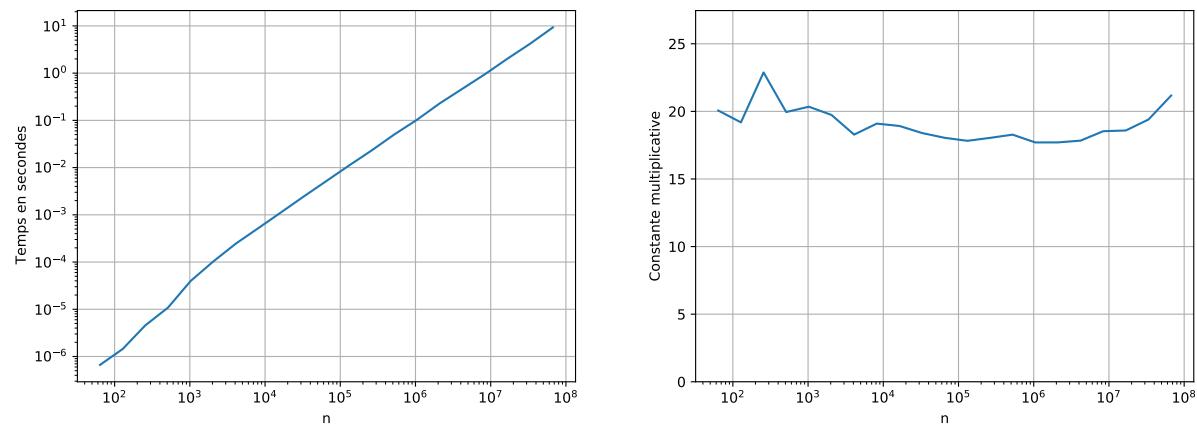


FIGURE 5.2 – Tri rapide en OCaml (raisonnablement optimisé).

Dichotomie en OCaml Pour la recherche dichotomique dans un tableau trié, on trouve bien un nombre d’instructions en $A \log n$ avec $A \simeq 20$, mais le nombre d’instructions par cycle varie significativement avec n (quand le tableau devient trop grand, il ne tient plus dans le *cache*, qui est plus ou moins la partie « rapide » de la mémoire).

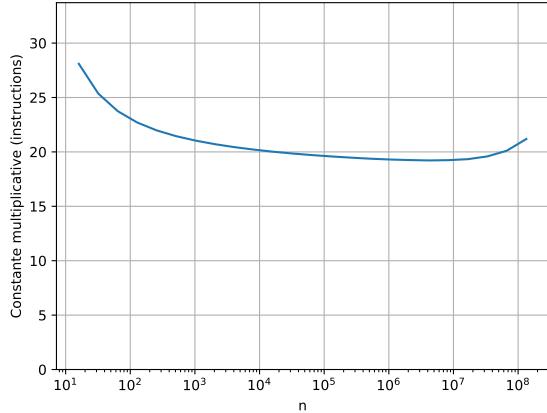


FIGURE 5.3 – Le nombre d’instructions par recherche vaut environ $20 \log n$.

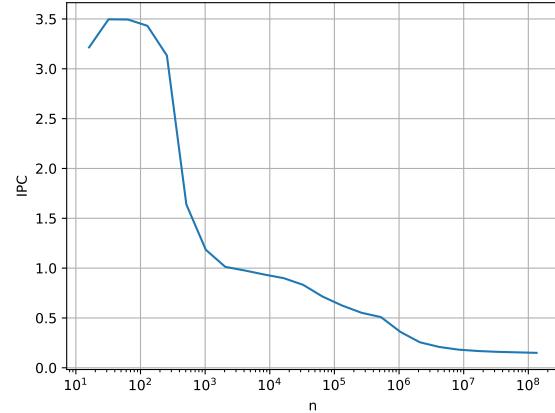


FIGURE 5.4 – Le nombre d’instructions par cycle décroît fortement avec n .

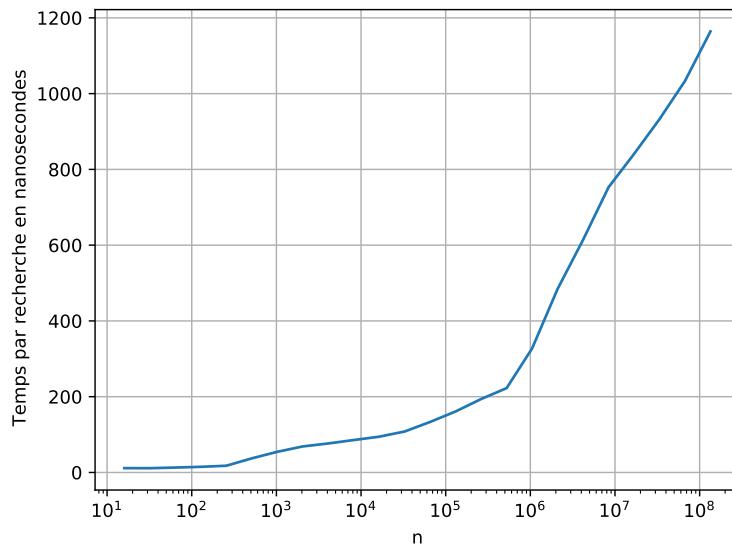


FIGURE 5.5 – Si le temps par recherche était proportionnel à $\log n$, on obtiendrait une droite.

Tri insertion en OCaml et tri fusion en Python Pour observer l’impact relatif des constantes multiplicatives et des complexités asymptotiques, on peut comparer un tri insertion bien écrit en OCaml et un tri fusion très naïf en Python, c’est-à-dire un algorithme en $A n^2$ et un en $B n \log n$ avec $A \ll B$.

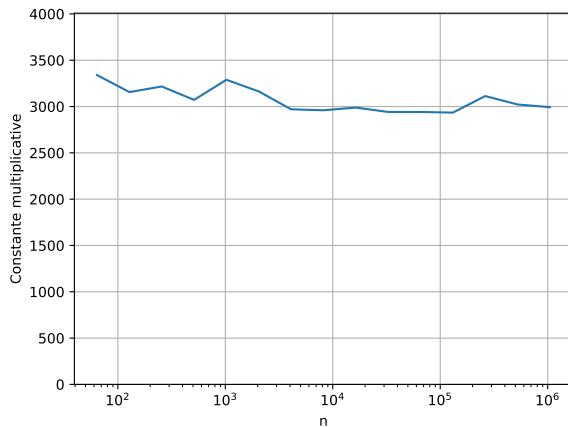


FIGURE 5.6 – Le nombre d’instructions exécutées par cette variante du tri fusion en Python est de l’ordre de $3000n \log n$.

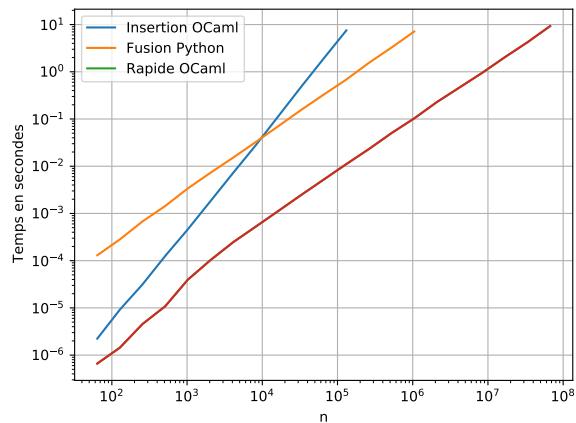


FIGURE 5.7 – Comparaison de temps de calcul.

5 Complexité en espace

La complexité en espace mesure la quantité de mémoire de travail utilisée par l’algorithme. On ne compte pas la taille des données, ni normalement la taille du résultat.

Exemple 5.4

- La fonction suivante calcule le n -ème nombre de Fibonacci :

```
let fibo n =
  let t = Array.make (n + 1) 0 in
  t.(1) <- 1;
  for i = 2 to n do
    t.(i) <- t.(i - 1) + t.(i - 2)
  done;
  t.(n)
```

Elle a une complexité en espace en $O(n)$ puisqu’elle alloue un tableau de taille $n + 1$ pour réaliser ses calculs (sa complexité en temps est également linéaire).

- La fonction suivante effectue le même calcul :

```
let fibo n =
  let u = ref 0 in
  let v = ref 1 in
  for i = 1 to n do
    let tmp = !u in
    u := !v;
    v := !v + tmp
  done;
  !u
```

Elle a également une complexité temporelle linéaire, mais sa complexité spatiale est constante, puisqu’elle utilise uniquement un nombre borné de variables entières.

De nombreux algorithmes « échangent de l'espace contre du temps » : pour obtenir une meilleure complexité temporelle, on accepte une moins bonne complexité spatiale. C'est par exemple le cas de tous les algorithmes de programmation dynamique, que nous étudierons ultérieurement. C'est souvent efficace en pratique, mais il y a un certain nombre de seuils qu'il est coûteux de dépasser (les données ne rentrent plus dans le cache, les données ne rentrent plus en mémoire vive, les données ne rentrent plus dans le stockage de masse local...).

Complexité spatiale et récursivité

Il y a une difficulté supplémentaire quand on s'intéresse à la complexité spatiale d'une fonction récursive, liée à une consommation « cachée » de mémoire sur la pile. Nous aborderons cette question quand nous nous intéresserons à la manière dont sont effectivement exécutées les fonctions récursives.

Taille des données et capacités des mémoires

Il faut avoir en tête quelques ordres de grandeur (simplifiés) :

- un entier, un flottant ou un pointeur prennent 8 octets en mémoire;
- un ordinateur typique a quelques giga-octets de mémoire vive;
- ce même ordinateur peut disposer de quelques téra-octets de mémoire de stockage, mais si l'on doit utiliser cette mémoire pour les calculs les performances peuvent facilement être divisées par mille (voire un million...).

Exercices

Exercice 5.5 – Tri par sélection

p. 90

On considère la fonction suivante :

```
let indice_mini t debut =
  let i_mini = ref debut in
  for i = debut + 1 to Array.length t - 1 do
    if t.(i) < t.(!i_mini) then i_mini := i
  done;
  !i_mini

let tri_selection t =
  for i = 0 to Array.length t - 2 do
    echange t i (indice_mini t i)
  done
```

1. Donner le type de `indice_mini` et de `tri_selection`.
2. Donner la spécification de la fonction `indice_mini`.
3. On considère le tableau `t = [| 2; 1; 4; 5; 0; 1 |]` et l'appel `tri_selection t`. Donner l'état du tableau `t` à la fin de chacune des itérations de la boucle de la fonction `tri_selection`.
4. Déterminer la complexité de la fonction `tri_selection` en fonction de la longueur n du tableau passé en argument. On supposera que les comparaisons entre éléments se font en temps unitaire.

Exercice 5.6 – Complexité de la recherche dichotomique

p. 90

On considère la fonction suivante, qui effectue une recherche dichotomique dans un tableau `t`, de longueur n , supposé trié.

```
let cherche_dicho t x =
  let rec aux deb fin =
    if deb >= fin then None
    else
      let mil = deb + (fin - deb) / 2 in
      if t.(mil) = x then Some mil
      else if t.(mil) < x then aux (mil + 1) fin
      else aux deb mil in
  aux 0 (Array.length t)
```

1. Montrer que si $\text{aux } d \ f \rightarrow \text{aux } d' \ f'$ (si l'appel `aux d f` résulte en un appel récursif aux `d' f'`), alors $f' - d' \leq (f - d)/2$.
2. En supposant que la comparaison entre deux éléments du tableau se fait en temps constant, en déduire que la fonction `cherche_dicho` est de complexité $O(\log n)$.

Exercice 5.7 – Complexité de l'exponentiation rapide

p. 91

On considère la fonction suivante :

```
let rec expo a n =
  if n = 0 then 1
  else if n mod 2 = 0 then expo (a * a) (n / 2)
  else a * expo a (n - 1)
```

- On note $M(n)$ le nombre de multiplications effectuées lors de l'appel `expo a n` (pour $n \geq 0$). Exprimer $M(2n+1)$ et $M(2n)$ en fonction de $M(n)$.
- En déduire que $M(n) \leq 1 + 2 \log_2 n$, pour $n \geq 1$.
- Quelle est la complexité de la fonction `expo` ?
- Expérimentalement, en traduisant cette fonction **en Python** et en mesurant le temps d'exécution de `expo(3, n)`, on obtient les courbes suivantes :

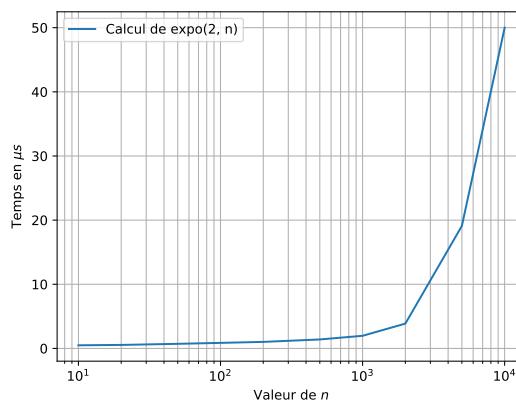


FIGURE 5.8 – Graphique semilog pour l'exponentiation rapide d'un entier.

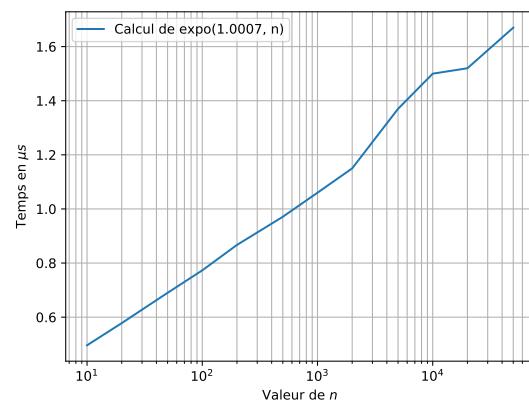


FIGURE 5.9 – Graphique semilog pour l'exponentiation rapide d'un flottant.

Comment expliquer ce phénomène ?

Exercice 5.8

p. 91

On veut calculer les termes de la suite définie par

$$u_0 = 1 \text{ et } \forall n \in \mathbb{N}^*, u_n = \frac{u_{n-1}}{1} + \frac{u_{n-2}}{2} + \dots + \frac{u_0}{n}.$$

Pour cela, on peut utiliser la fonction suivante :

```
let rec suite n =
  if n = 0 then 1.
  else
    begin
      let s = ref 0. in
      for k = 1 to n do
        s := !s +. suite (n - k) /. float k
      done;
      !s
    end
```

- On note $D(n)$ le nombre de divisions qui seront effectuées au total lors de l'appel `suite n` (en comptant les appels récursifs). Donner une relation de récurrence reliant $D(n)$ et les $D(k)$ pour $0 \leq k < n$.

2. En déduire la valeur de $D(n)$, puis la complexité de la fonction suite.
3. Écrire une fonction (non récursive) effectuant le calcul en un temps plus raisonnable, au prix de l'utilisation d'un stockage auxiliaire. On déterminera les complexités temporelle et spatiale de cette nouvelle fonction.

Remarque

Nous verrons plus tard que la première version de `suite` a en fait une complexité spatiale en $\Theta(n)$, ce qui signifie que la deuxième version est strictement meilleure (même complexité spatiale, complexité temporelle incomparablement meilleure).

Solutions

Correction de l'exercice 5.5 page 87

1. On a `indice_mini` : '`a array` -> `int` -> `int` et `tri_selection` : '`a array` -> `unit`.
2. Entrées : un '`a array` `t`, un `int` `début`.
Précondition : $0 \leqslant \text{début} < |t|$
Résultat : un indice $i \in [\text{début} \dots |t| - 1]$ tel que $t_i = \min t[\text{début} \dots |t| - 1]$.
3. On obtient successivement :
 - 0, 1, 4, 5, 2, 1
 - 0, 1, 4, 5, 2, 1
 - 0, 1, 1, 5, 2, 4
 - 0, 1, 1, 2, 5, 4
 - 0, 1, 1, 2, 4, 5
4. Si $|t| = n$, la fonction `indice_mini` a une complexité en $O(n - \text{début})$. La fonction `tri_selection` fait des appels à `indice_mini` pour des valeurs de `début` variant de 0 à $n - 2$, on obtient donc

$$O\left(\sum_{i=0}^{n-2} (n - i)\right) = O(n^2)$$

Correction de l'exercice 5.6 page 87

1. On a $m = d + \lfloor \frac{f-d}{2} \rfloor$, donc

$$(1) \quad m \leqslant d + \frac{f-d}{2} < m+1.$$

S'il y a un appel récursif, il se fera :

- soit avec $d' = m + 1$ et $f' = f$, d'où

$$\begin{aligned} f' - d' &= f - (m + 1) \\ &< f - \left(d + \frac{f-d}{2}\right) && \text{d'après (1)} \\ &= \frac{f-d}{2} \end{aligned}$$

- soit avec $d' = d$ et $f' = m$, d'où

$$\begin{aligned} f' - d' &= m - d \\ &\leqslant d + \frac{f-d}{2} - d && \text{d'après (1)} \\ &= \frac{f-d}{2} \end{aligned}$$

2. Après une série de k appels, on a donc $f_k - d_k \leqslant \frac{f_0 - d_0}{2^k} = \frac{n}{2^k}$. Or la fonction termine dès que $f_k - d_k = 0$, c'est-à-dire dès que $f_k - d_k < 1$ (ou avant, si l'on trouve l'élément). Au plus tard, on s'arrête donc quand $\frac{n}{2^k} < 1$, c'est-à-dire $2^k > n$ ou $k > \log_2 n$. Le nombre d'appels est donc en $O(\log n)$, et comme chaque appel s'effectue en temps constant on en déduit que la fonction a une complexité temporelle en $O(\log n)$.

Correction de l'exercice 5.7 page 87

1. La lecture du code fournit immédiatement, pour $n \geq 1$:

$$\begin{cases} M(2n) = 1 + M(n) \\ M(2n+1) = 1 + M(2n) = 2 + M(n) \end{cases}$$

2. On procède par récurrence forte sur $n \geq 1$.

Initialisation : pour $n = 1$, on a d'après le code $M(1) = 1$, et $1 + 2 \log_2 1 = 1$, la propriété est initialisée.

Héritéité : soit $n > 1$, supposons la propriété vérifiée pour $1 \leq k < n$.

- Si n est pair, $n = 2k$ avec $1 \leq k < n$, alors :

$$\begin{aligned} M(n) &= M(2k) \\ &= 1 + M(k) \\ &\leq 2 + 2 \log_2 k && \text{d'après } H_k \\ &= 2(1 + \log_2 k) \\ &= 2 \log_2(n) && n = 2k \\ &< 1 + 2 \log_2 n \end{aligned}$$

- Si n est impair, $n = 2k + 1$ avec $1 \leq k < n$, alors :

$$\begin{aligned} M(n) &= M(2k + 1) \\ &= 2 + M(k) \\ &\leq 3 + 2 \log_2 k && \text{d'après } H_k = 1 + 2(1 + \log_2 k) \\ &= 1 + 2 \log_2(2k) \\ &\leq 1 + 2 \log_2(2k + 1) \\ &= 1 + 2 \log_2 n \end{aligned}$$

On a donc bien $M(n) \leq 1 + 2 \log_2 n$ pour $n \geq 1$.

3. La complexité de la fonction est proportionnelle au nombre de multiplications, c'est donc un $O(\log n)$.

Ce grand- O est en fait un grand- Θ : on pourrait facilement montrer que $M(n) \geq 1 + \log_2 n$.

4. La majoration du nombre de multiplications reste bien sûr valable, c'est donc la proportionnalité de la complexité à ce nombre de multiplications qui doit être fausse dans le cas où a est entier. Effectivement, les entiers en Python ne sont pas des entiers machine, mais peuvent croître arbitrairement : par exemple, `expo(3, 10000)` a plus de 4 000 chiffres en base 10. Les multiplications deviennent donc de plus en plus coûteuses, et la complexité n'est plus logarithmique.

Ce problème ne se pose pas pour les flottants, et on obtient alors bien une droite dans le graphique semilog (ce qui correspond à une complexité de la forme $A \log n$).

Correction de l'exercice 5.8 page 88

STRUCTURES DE DONNÉES

I Type abstrait et implémentation

1.1 Types abstraits liste et vecteur

Un type de donnée est avant tout spécifié par une liste d'opérations élémentaires permettant de créer (ou éventuellement de modifier) un objet du type en question ainsi que d'accéder aux éléments qu'il contient. Par exemple, on appellera usuellement « liste » et « vecteur » une structure de données fournissant les opérations élémentaires suivantes :

Opération	Type	Effet
<code>[]</code>	<code>'a list</code>	Liste vide
<code>cons x m</code>	<code>'a -> 'a list -> 'a list</code>	Rajoute un élément en tête
<code>tail m</code>	<code>'a list -> 'a list</code>	Renvoie la liste sauf son premier élément
<code>head m</code>	<code>'a list -> 'a</code>	Renvoie le premier élément de la liste

FIGURE 6.1 – Signature d'un type « liste ».

Opération	Type	Effet
<code>create n x</code>	<code>int -> 'a -> 'a array</code>	Renvoie un vecteur de taille n initialisé à x
<code>get t i</code>	<code>'a array -> int -> 'a</code>	Renvoie l'élément d'indice i
<code>set t i x</code>	<code>'a array -> int -> 'a -> unit</code>	Modifie l'élément d'indice i en place

FIGURE 6.2 – Signature d'un type « vecteur ».

Ces « signatures » ne précisent pas la manière dont est réalisée la structure, ce qui est voulu : tant que la spécification est respectée, la réalisation concrète peut être changée sans que le code utilisant la structure n'ait à être réécrit. Cependant, on précise normalement la complexité des opérations élémentaires : une structure qui ne permet pas l'accès à un élément arbitraire en temps constant ne sera généralement pas considéré comme un vecteur. On obtiendrait alors :

Opération	Coût
<code>cons x m</code>	$O(1)$
<code>tail m</code>	$O(1)$
<code>head m</code>	$O(1)$

Opération	Coût
<code>create n x</code>	$O(n)$
<code>get t i</code>	$O(1)$
<code>set t i x</code>	$O(1)$

FIGURE 6.3 – Coût des opérations sur une « liste ».

FIGURE 6.4 – Coût des opérations sur un « vecteur ».

Si l'on connaît les opérations élémentaires fournies et leur coût, l'implémentation sous-jacente n'a que peu d'importance pour l'« utilisateur » de la structure de donnée : c'est tout l'intérêt du concept de type abstrait de données. Cependant, vous êtes censés savoir comment ces structures sont réalisées en OCaml (listes simplement chaînées et tableaux, respectivement).

1.2 Modèle mémoire

On présente ici un modèle extrêmement simplifié. Nous rentrerons (un peu) plus dans les détails lorsque nous programmerons en C.

On peut voir la mémoire (vive) d'un ordinateur comme un immense tableau dont les cases sont numérotées de 0 à $N - 1$, où N est de l'ordre de quelques milliards (ou nettement plus).

- Le numéro d'une case est appelé *adresse mémoire*. Il s'agit d'un entier de taille fixée. Cette taille est celle d'un *mot machine*, et correspond également à la taille des *entiers machine*, c'est-à-dire des entiers que le processeur est capable de traiter en une opération (pour faire une addition par exemple). La taille d'un mot machine fait 64 bits sur la grande majorité des ordinateurs actuels (32 bits parfois).
 - Les cases ont toutes la même taille (elles peuvent toutes contenir la même quantité de données) : un octet, c'est-à-dire 8 bits. Cependant, nous considérerons pour simplifier que chaque case fait un mot machine – sur un ordinateur actuel, cela revient à regrouper les cases par paquets de 8. Ainsi, une case peut contenir l'adresse d'une autre case (on parle de « pointeur »).
 - Accéder (en lecture ou en écriture) à une case d'adresse donnée est une opération élémentaire qui se fait en temps constant.

1.3 Réalisation concrète en OCaml

Tableau

- Un `'a array` est représenté en OCaml par une série de cases mémoire consécutives. La « valeur » d'un tableau `u` est en fait l'adresse de la première de ces cases :
 - quand on passe un `t` : `'a array` comme argument à une fonction, on passe en fait l'*adresse* du tableau `t` ;
 - si l'on fait `let v = t in ...`, alors le nom `v` est associé à la même adresse que le nom `t`.
 - Pour obtenir l'adresse de la case `t.(i)`, il suffit de prendre l'adresse de `t` et de lui ajouter `i` (à quelques détails près) : on peut donc bien accéder à une case quelconque en temps constant (une addition et une lecture mémoire).
 - Pour que cette méthode fonctionne, il est indispensable que toutes les cases du tableau fassent la même taille : en OCaml, ce sera systématiquement un mot machine.
 - Si un objet de type `'a` ne tient pas dans un mot machine, les cases d'un `'a array` contiendront donc en fait des pointeurs vers les éléments du tableau.
 - La taille du tableau est stockée juste avant les données ; cela permet d'avoir une fonction `Array.length` (en temps constant, de plus) et de détecter quand un accès à `t.(i)` est « hors bornes » (en comparant `i` à la longueur de `t`).

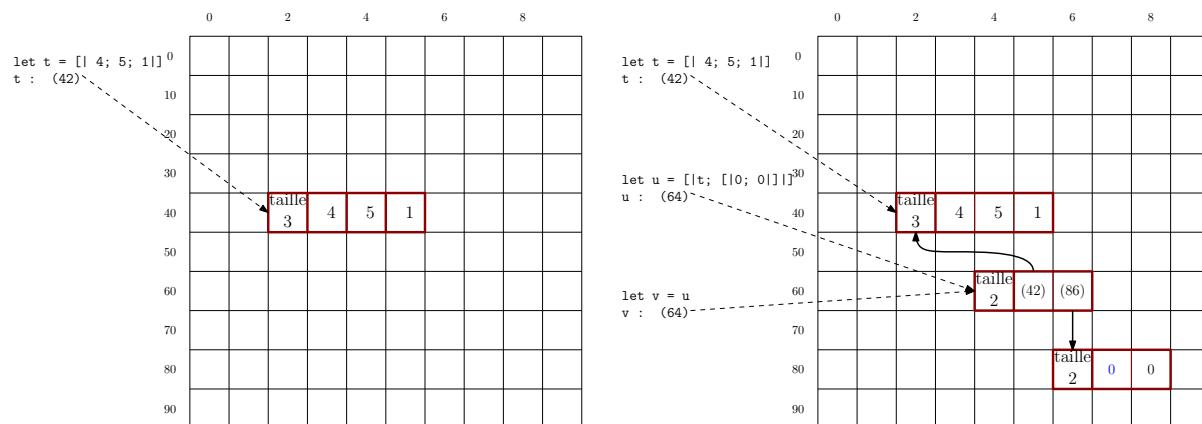


FIGURE 6.5 – Représentation mémoire des **array** en OCaml.

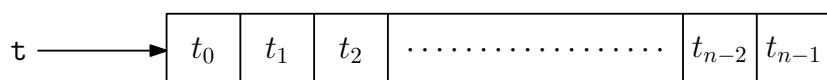


FIGURE 6.6 – Représentation simplifiée d'un **array**.

Remarques

- On peut être plus précis sur le calcul de l'adresse de la i -ème case d'un tableau t : sachant que chaque case d'un tableau OCaml occupe 8 octets et qu'il y a un bloc de 8 octets au début du tableau contenant (entre autres) la taille du tableau, on a :

$$\text{adresse}(t_i) = \text{adresse}(t) + 8(i + 1)$$

- En C, on ne sera pas limité à des « cases » de 8 octets : cela ne pose pas de problème (il faut juste remplacer le 8 dans le calcul ci-dessus) mais il est indispensable que toutes les cases d'un tableau donné fassent la même taille.
- Être capable de détecter les accès « hors bornes » à un tableau est très important (et il est nécessaire que le tableau « connaisse sa taille » si l'on veut pouvoir garantir cette détection). Si on ne le fait pas, alors un accès à la 10^{ème} case d'un tableau de taille 5 résulte en un accès mémoire à une case n'ayant rien à voir avec le tableau. Plusieurs choses peuvent alors se produire :
 - si l'on a de la chance, cet accès est interdit par le système car la zone mémoire en question n'appartient pas au programme : on a donc un plantage immédiat avec une *segmentation fault* ;
 - si l'on a moins de chance, la lecture ou l'écriture a lieu et l'on a donc une corruption silencieuse du programme : il finira sans doute par planter on se sait trop quand, après avoir fait on ne sait trop quoi ;
 - si l'on a vraiment pas de chance, cet accès illégal n'a pas été fait par hasard mais bien pour accéder à des données qui devraient être protégées : on peut ainsi par exemple modifier l'adresse de retour d'une fonction pour exécuter du code arbitraire. C'est l'exemple le plus classique de faille de sécurité : on parle de *buffer overflow*.

Liste simplement chaînée

- Une liste est constituée de *cellules*, et chaque cellule est constituée de deux cases : l'une contenant l'élément de la liste, l'autre l'adresse de la cellule suivante.
- S'il n'y a pas de cellule suivante (parce qu'on est au bout de la liste), on mettra une valeur particulière dans la deuxième case (zéro, en pratique).
- La « valeur » de la liste est l'adresse de la cellule de tête.
- Si u est une 'a **list** et x : 'a, la liste $x :: u$ est obtenue en créant une nouvelle cellule constituée de l'élément x et d'un pointeur vers la liste u (c'est-à-dire vers la cellule de tête de u).

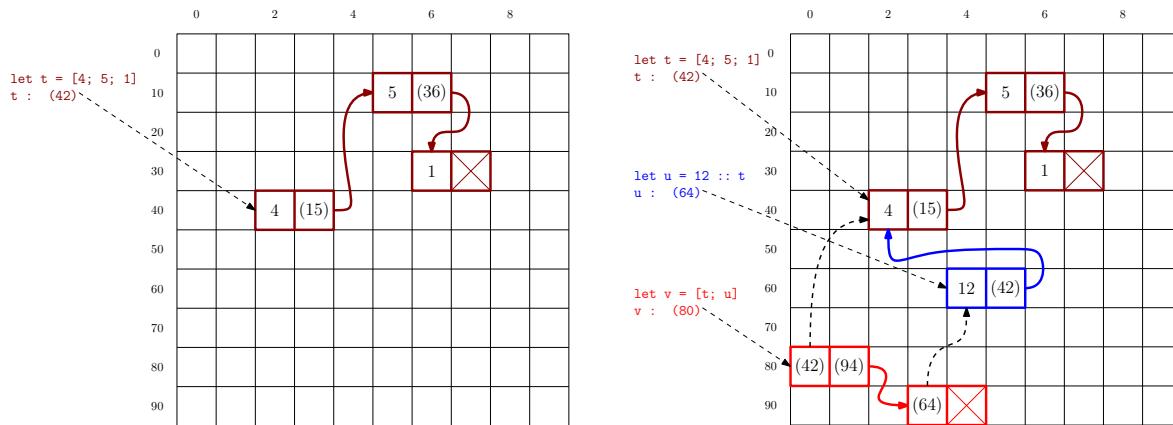


FIGURE 6.7 – Représentation mémoire des **list** en OCaml.

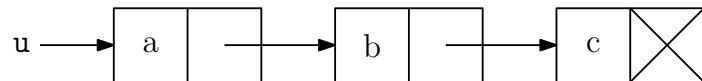


FIGURE 6.8 – Représentation simplifiée d'une liste $u = [a; b; c]$.

Remarques

- Connaître une liste u , c'est simplement connaître l'adresse de sa première cellule : par conséquent, le seul moyen d'accéder au i -ème élément d'une liste est de suivre i pointeurs, ce qui prendra un temps proportionnel à i .
- De même, le calcul de la longueur prend un temps proportionnel à la longueur de la liste : il faut suivre les pointeurs jusqu'à arriver à la liste vide.
- Une liste consomme plus d'espace qu'un tableau contenant les mêmes éléments puisqu'il faut stocker un pointeur supplémentaire par élément. Cependant, cela ne changera rien aux complexités spatiales puisqu'on négligera les facteurs constants.
- En réalité, une cellule de liste occupe *trois* cases en OCaml, la troisième étant à usage interne (elle sert au *garbage collector*¹ pour libérer la mémoire quand elle n'est plus utilisée).

Une remarque sur les « listes » de Python La réalisation de ces deux structures en OCaml est très classique, mais c'est loin d'être la seule possible. En particulier, le type `list` de Python ressemble à la fois à ce que l'on a ici appelé « liste » et « vecteur », mais son implémentation est significativement différente de celle des `array` de OCaml (et n'a rien à voir avec celle des `list`). On parle généralement de *tableau dynamique* pour désigner le type de données abstrait correspondant aux `list` de Python (et aussi aux `std::vector` du C++, entre autres).

2 Structures de données fonctionnelles et impératives

Une structure de donnée est dite *fonctionnelle* (ou *persistante*) si elle ne peut pas être modifiée *en place* ; dans le cas contraire, elle est dite *impérative*, ou *mutable*. La spécification donnée plus haut pour les listes est fonctionnelle : les opérations `cons` et `tail` ne modifient pas la liste passée en entrée, mais renvoient de nouvelles listes. C'est tout le contraire pour la spécification « vecteur ».

Remarque

Le type `list` de Python satisfait simultanément les spécifications « liste » et « vecteur », à ceci près qu'il est impératif (et que l'ajout des éléments se fait en queue) : si on exécute `m.append(x)`, `m` est modifiée.

Les structures de données impératives ont un intérêt majeur : il est souvent plus facile d'écrire des algorithmes *efficaces* en utilisant ces structures. En particulier, il n'existe pas de réalisation fonctionnelle de la structure de « vecteur » permettant l'accès à une case arbitraire en temps constant, ce qui impose parfois (assez rarement en fait) un facteur $\log n$ supplémentaire pour la complexité si l'on ne souhaite utiliser que des structures fonctionnelles.

L'intérêt des structures fonctionnelles est double :

- il est souvent plus facile d'écrire des programmes *corrects* si l'on limite au maximum les effets de bord (en particulier, on n'a plus de problème d'*aliasing*) ;
- plusieurs objets peuvent partager une partie de leur représentation mémoire ;
- en particulier, on peut garder une trace des précédentes versions de la structure pour un coût moindre que dans le cas mutable : si l'on fait un `append` en Python, l'ancienne version de la liste est perdue. Le seul moyen d'en garder une trace (ce qui est assez souvent utile, comme nous aurons l'occasion de le voir) est d'en faire une copie, pour un coût en $O(n)$ tant en temps qu'en espace. En Caml, un `let u = x :: v` laisse `v` accessible tout en ayant un coût unitaire.²

Exercice 6.1

p. 98

On considère deux listes $u = [1; 2; 3]$ et $v = [21; 22; 23]$. Faire les schémas mémoire (simplifiés) correspondant à :

1. `let w = 0 :: u;`
2. `let a = w @ v;`
3. `let b = u @ a;`
4. `let c = [a; b].`

1. Ramasse-miettes en français

2. L'équivalent Python, `u = v + [x]`, a un coût linéaire en la taille de `v`.

Exercice 6.2

p. 98

On considère la fonction suivante :

```
let rec f u =
  match u with
  | [] -> []
  | x :: xs -> u :: f xs
```

1. Quel est le type de `f` ?
2. Que renvoie cette fonction si on l'appelle sur `[1; 2; 3]` ?
3. Quelle est la complexité en temps de `f`, en fonction de $|u|$?
4. Faire un schéma mémoire correspondant au résultat renvoyé pour $u = [1; 2; 3]$.
5. Combien d'espace le résultat renvoyé occupe-t-il en mémoire en fonction de la longueur $|u|$ de l'argument (on supposera que chaque élément de la liste de départ u occupe un espace unitaire) ? En quoi cela peut-il être surprenant ?

Dans le cas d'une structure de données fonctionnelle, le fait que plusieurs objets partagent tout ou partie de leur représentation mémoire ne pose jamais de problème. En revanche, pour une structure impérative, il convient d'être très prudent !

Exercice 6.3

p. 99

On souhaite écrire une fonction nulle : `int -> int -> float array array` permettant de créer une matrice nulle dont les dimensions sont passées en argument.

1. On propose le code suivant :

```
let nulle n p =
  let ligne = Array.make p 0. in
  Array.make n ligne
```

- a. Quelle est la complexité de cette fonction ?
 - b. Faire un schéma de la représentation mémoire de `nulle 2 3`.
 - c. Quel est le problème ?
2. Reprendre la question précédente dans chacun des cas suivants :

```
let nulle_2 n p =
  Array.make n (Array.make p 0.)

let nulle_3 n p =
  let t = Array.make n [| |] in
  for i = 0 to n - 1 do
    t.(i) <- Array.make p 0.
  done;
  t

let nulle_4 n p =
  Array.init n (fun i -> Array.make p 0.)
```

Remarque

Ce problème se présentant assez souvent, OCaml propose une fonction prédéfinie `Array.make_matrix : int -> int -> 'a -> 'a array array` permettant de créer et initialiser proprement une « matrice ».

3 Définition en OCaml

Il n'est pas possible de redéfinir le type **array** en OCaml (il faut avoir accès à la représentation mémoire des données). En revanche, le type **list** se définit très simplement :

```
type 'a liste =
| Vide
| Cons of 'a * 'a liste

(* Pour définir l'équivalent de [1; 3; 7] : *)
let u = Cons(1, Cons(3, Cons(7, Vide)))

let rec somme u =
  match u with
  | Vide -> 0
  | Cons (x, xs) -> x + somme xs

# somme u;;
- : int = 11
```

C'est un type *récursif* : il apparaît des deux côtés du signe = dans sa définition. Ce sera le cas à chaque fois que l'on définira un nouveau type de données (permettant de stocker un nombre arbitraire d'éléments) sans réutiliser les types **list** ou **array**.

Remarque

Le type '**a** liste' défini ci-dessus est **rigoureusement** équivalent au type '**a** list' prédéfini : après compilation, on obtient exactement les mêmes instructions machine.

4 Exemple de spécification plus complexe

On donne ici une spécification possible pour un type « ensemble » (*a priori* fonctionnel). Nous avons déjà vu en TP deux réalisations possibles de ce type de données (à l'aide de listes, ordonnées ou non) ; nous verrons plus tard dans l'année une réalisation plus efficace utilisant une table de hachage, et l'année prochaine une autre réalisation efficace utilisant un arbre binaire de recherche.

Opération	Type	Effet
vide	'a set	Ensemble vide
ajouter x s	'a -> 'a set -> 'a set	Renvoie $s \cup \{x\}$
egal s t	'a set -> 'a set -> bool	Égalité ensembliste
est_dans x s	'a -> 'a set -> bool	Test d'appartenance
union s t	'a set -> 'a set -> 'a set	Renvoie $s \cup t$
inter s t	'a set -> 'a set -> 'a set	Renvoie $s \cap t$
diff s t	'a set -> 'a set -> 'a set	Renvoie $s \setminus t$

FIGURE 6.9 – Signature (incomplète) d'un type set fonctionnel.

Remarque

Il manque (au moins) une opération pour que ce type soit utilisable : laquelle ?

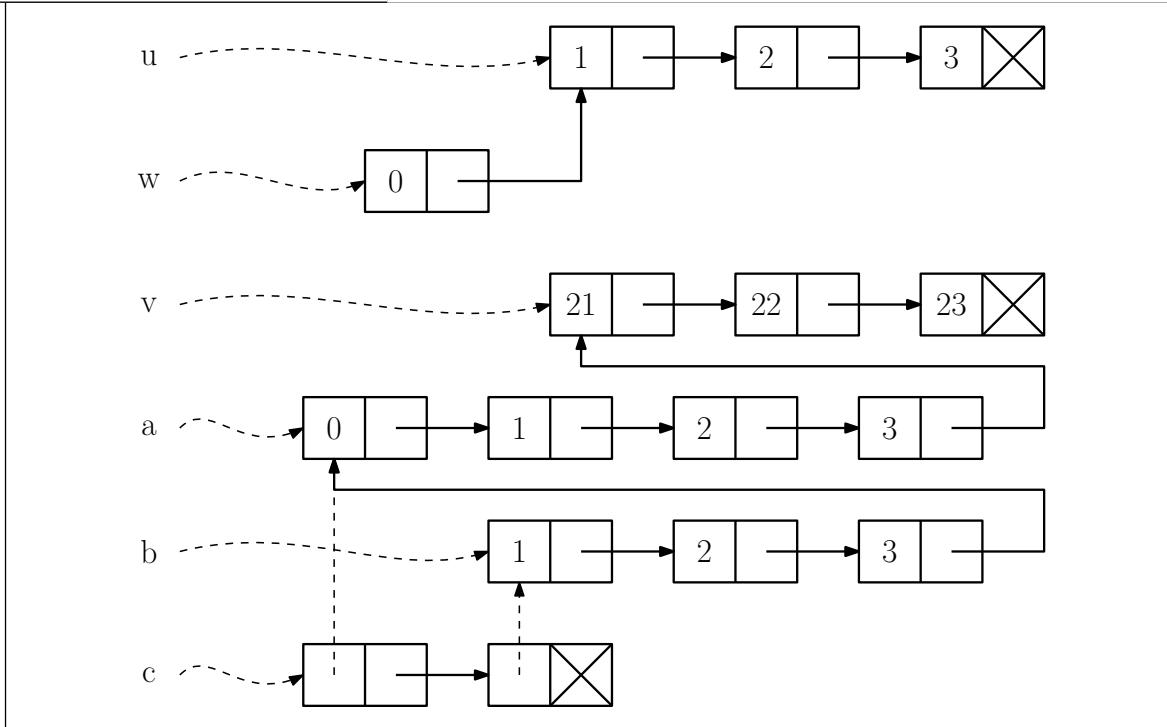
Exercice 6.4

p. 99

En supposant que le type '**a** set' a été défini avec la signature ci-dessus, écrire une fonction **est_sans_doublon** : '**a** list -> bool' qui renvoie **true** si et seulement si les éléments de la liste passée en argument sont deux à deux distincts.

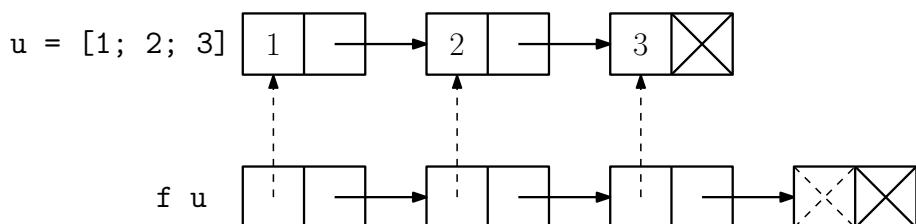
Solutions

Correction de l'exercice 6.1 page 95



Correction de l'exercice 6.2 page 96

1. `f : 'a list -> 'a list list`
2. `f [1; 2; 3] = [[1; 2; 3]; [2; 3]; [3]; []]`
3. Une opération en temps unitaire pour chaque élément de u , donc la complexité est en $O(|u|)$.
- 4.



5. On consomme un espace mémoire proportionnel à $|u|$. C'est un peu surprenant si l'on regarde le résultat de la fonction : on renvoie une liste de listes, et la somme de la longueur de ces listes est proportionnel à $|u|^2$. Le schéma mémoire explique cette situation : les différentes listes partagent en fait leurs données.

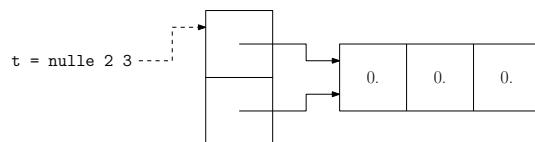
Remarque

La complexité temporelle étant en $O(|u|)$, il est de toute façon impossible de consommer une quantité quadratique de mémoire : chaque écriture prend au moins un temps unitaire !

Correction de l'exercice 6.3 page 96

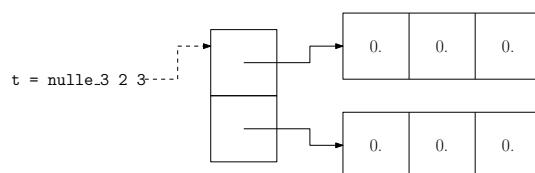
1. a. La première ligne s'exécute en temps $O(p)$ et la deuxième en temps $O(n)$, donc $O(n+p)$ au total.

b.



c. Toutes les lignes de notre « matrice » sont **physiquement** égales : si l'on en modifie une, elles seront toutes affectées.

2. ■ `nulle_2` se comporte exactement comme `nulle` (c'est exactement la même fonction).
■ Pour `nulle_3`, la première ligne est en $O(n)$ puis on passe n fois dans une boucle dont le corps est en $O(p)$, donc $O(np)$ pour la boucle et aussi $O(np)$ au total.



- `nulle_4` se comporte comme `nulle_3` (l'appel `Array.make p 0.` est fait n fois).

Correction de l'exercice 6.4 page 97

On utilise une fonction auxiliaire qui prend en argument une liste u et un ensemble s et renvoie `true` si et seulement si :

- la liste u est sans doublons ;
- et aucun élément de u n'appartient à l'ensemble s .

```
let est_sans_doublon u =
  let rec aux liste ensemble =
    match liste with
    | [] -> true
    | x :: xs ->
      if est_dans x ensemble then false
      else aux xs (ajouter x ensemble) in
        aux u vide
```

Il est plus élégant d'éliminer le if/then/else :

```
let est_sans_doublon u =
  let rec aux liste ensemble =
    match liste with
    | [] -> true
    | x :: xs ->
      not (est_dans x ensemble) && aux xs (ajouter x ensemble) in
        aux u vide
```

Remarque

Pour la complexité de cette fonction (en notant $n = |u|$), on fait :

- n opérations élémentaires sur les listes ;
- au plus n appels à `est_dans` sur des ensembles de taille au plus n ;
- au plus n appels à `ajouter` sur des ensembles de taille au plus n .

La complexité des opérations `est_dans` et `ajouter` dépendra de la réalisation concrète de la structure d'ensemble : ce sera typiquement du $O(1)$ ou $O(\log n)$ pour les réalisations efficaces, et du $O(n)$ pour les réalisations naïves, ce qui donnera une complexité de $O(n)$, $O(n \log n)$ ou $O(n^2)$ pour `est_sans_doublon` suivant les cas.

PILES ET FILES

I Piles

1.1 Principe

Une pile (*stack* en anglais) est une structure de données dite LIFO (*last in, first out*, ce qu'on traduit le plus souvent par *premier arrivé, dernier servi*), qui doit son nom à l'analogie avec une pile d'assiettes. Les deux opérations élémentaires sont :

- rajouter une assiette au sommet de la pile ;
- récupérer l'assiette se trouvant actuellement au sommet (qui est donc la dernière à avoir été empilée). Bien évidemment, cette opération échouera si la pile est vide.

Dans la pile ci-dessous, l'élément au sommet (3) est le dernier ajouté, et sera le prochain à être retiré (à condition qu'on n'en ajoute pas d'autre avant de faire une extraction).

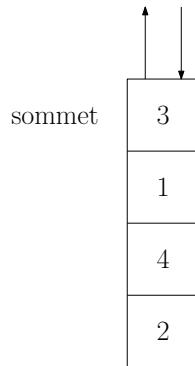


FIGURE 7.1 – Un exemple de pile contenant quatre entiers.

1.2 Pile fonctionnelle

Opération	Type	Signification
empty_stack	'a stack	Pile vide
is_empty	'a stack -> bool	Teste si la pile est vide
push	'a -> 'a stack -> 'a stack	Rajoute un élément au sommet
pop	'a stack -> 'a * 'a stack	Récupère le sommet et le reste de la pile

FIGURE 7.2 – Interface d'une pile fonctionnelle

Remarque

On se convainc facilement que cette interface est parfaitement équivalente à celle du type abstrait *liste* vue au chapitre précédent : push a la même spécification que cons et pop u renvoie le couple (head u, tail u). Il n'y a donc rien de nouveau à dire, et l'on s'utilisera simplement des '*a* **list**' OCaml pour réaliser cette structure.

1.3 Pile impérative

1.3.a Interface

Opération	Type	Signification
empty_stack	<code>unit</code> -> ' <code>a</code> stack	Crée une nouvelle pile vide
is_empty	' <code>a</code> stack -> <code>bool</code>	Teste si la pile est vide
push	' <code>a</code> -> ' <code>a</code> stack -> <code>unit</code>	Empile un élément (modifie la pile)
pop	' <code>a</code> stack -> ' <code>a</code>	Dépile un élément (modifie la pile)

FIGURE 7.3 – Interface d'une pile mutable

Remarques

- Attention à la fonction `pop` : son type de retour est '`a` (et pas `unit`) mais elle a bien un effet de bord (l'élément est retiré de la pile). Ainsi, deux appels successifs `pop` s'enverront en général deux valeurs différentes.
 - Dans le cas d'une pile impérative, `empty_stack` est une fonction de type `unit` -> '`a` stack, et pas juste une constante de type '`a` stack comme dans le cas fonctionnel. Pourquoi ?
- On peut alors écrire une fonction `somme` :

```
let somme pile =
  let s = ref 0 in
  while not (is_empty pile) do
    let x = pop pile in
    s := !s + x
  done;
  !s
```

Remarques

- Cette fonction a un **énorme** inconvénient : lequel ?
- En réalité, on n'utilisera que très rarement les piles de cette manière (parcours de toute la pile). Il sera bien plus fréquent d'avoir des insertions et des extractions entrelacées, comme nous aurons l'occasion de le voir dans les chapitres suivants.

1.3.b Implémentation

Version à base de liste On peut très facilement réaliser une pile impérative en utilisant une '`a` `list` ref :

```
type 'a pile = 'a list ref
```

```
let empty_stack () = ref []
let push x s = (s := x :: !s)
```

```
let pop s =
  match !s with
  | [] -> failwith "pop from empty stack"
  | x :: xs -> s := xs; x
```

Si l'on souhaite un schéma-mémoire, il faut juste comprendre qu'une '`a` `list` ref est simplement une case mémoire qui contient un pointeur **mutable** vers une liste :

FIGURE 7.4 – Pile réalisée par une `int list` ref, et résultat d'une opération `push 3 f`.

Pile dans un tableau Pour la deuxième version, il faut, à la création de la pile, définir une capacité maximale. On crée alors un tableau ayant cette taille, et l'on maintient à jour l'indice courant du sommet actuel :

- pour ajouter un élément, on vérifie que la pile n'est pas pleine, on incrémente courant et l'on écrit le nouvel élément;
 - pour extraire un élément, on vérifie que la pile n'est pas vide, on récupère l'élément à l'indice courant et l'on décrémente courant.

En OCaml, on pourrait utiliser le type suivant :

```
type 'a pile = {donnees : 'a array; mutable courant : int}
```

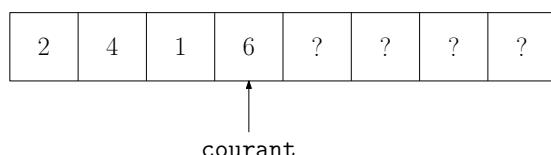


FIGURE 7.5 – Une pile de capacité 8 contenant quatre éléments. courant vaut 3, donc le sommet de la pile est données . $(3) = 6$.

Remarques

- Les valeurs des cases situées à droite de courant n'ont aucune importance.
 - Si l'on utilise le type proposé plus haut, il faut disposer d'un élément de type 'a au moment de la création (on parle de *témoin de type*) car il est nécessaire d'initialiser le tableau. Une solution à ce problème est d'utiliser un tableau de 'a option (que l'on initialisera alors à **None**).
 - Il est possible de se libérer de la contrainte de capacité bornée en utilisant un tableau *dynamique* (c'est-à-dire redimensionnable); c'est ce qui est fait dans le type **list** de Python, et nous aurons l'occasion de réaliser cette structure en C et en OCaml cette année.

2 Files

2.1 Principe et interface

Une file (*queue* en anglais) est une structure FIFO (*first in, first out*, qu'on pourrait traduire par *premier arrivé, premier servi*). On peut l'imaginer horizontale : on rajoute les éléments par la gauche, on les enlève par la droite. Cette fois, l'analogie est avec une file d'attente : les clients arrivent par la gauche, le guichet est à droite. Le prochain client servi sera celui qui attend depuis le plus longtemps.

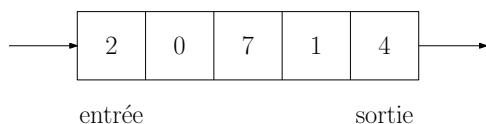


FIGURE 7.6 – Une file contenant cinq éléments. Le dernier arrivé est 2, le prochain à sortir sera 4.

Opération	Type	Signification
empty_queue	'a queue	File vide
is_empty	'a queue -> bool	Teste si la file est vide
push	'a -> 'a queue -> 'a queue	Rajoute un élément
pop	'a queue -> 'a * 'a queue	Récupère l'élément le plus ancien et le reste de la file

FIGURE 7.7 – Interface d'une file fonctionnelle

Opération	Type	Signification
empty_queue	<code>unit</code> -> <code>'a queue</code>	Crée une file vide
is_empty	<code>'a queue</code> -> <code>bool</code>	Teste si la file est vide
push	<code>'a</code> -> <code>'a queue</code> -> <code>unit</code>	Rajoute un élément
pop	<code>'a queue</code> -> <code>'a</code>	Récupère l'élément le plus ancien (et l'enlève)

FIGURE 7.8 – Interface d'une file mutable

Remarques

- Les noms des opérations sont moins fixés que pour les piles : on peut par exemple trouver enqueue pour celle que j'ai appelée push et dequeue pour celle que j'ai appelée pop.
- À nouveau, attention à la version impérative de pop qui renvoie une valeur **et** a un effet de bord.
- Sauf mention contraire, on écrira (x_1, \dots, x_n) pour une file dont l'élément le plus ancien est x_n .

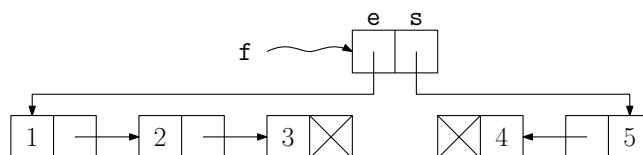
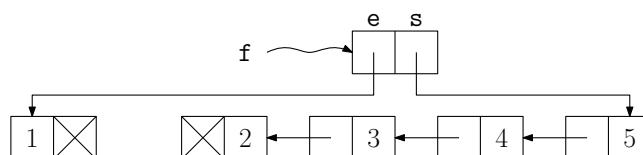
2.2 File fonctionnelle

On peut implémenter naïvement une file à l'aide d'une liste, mais soit `push_left` soit `pop_right` sera alors en $O(n)$, suivant si l'on fait les insertions en queue ou en tête de liste. La solution classique est d'utiliser deux listes :

- une que l'on appellera e pour « entrant » et qui correspondra à la partie « gauche » de la file ;
- une que l'on appellera s pour « sortant » et qui correspondra à la partie « droite » de la file, à l'**envers**.

Le couple $(e, s) = ([x_1; \dots; x_n], [y_1; \dots; y_p])$ correspondra alors à la file $\rightarrow [x_1 | \dots | x_n | y_p | \dots | y_1] \rightarrow$

- Pour ajouter un élément à la file, il suffit de le rajouter en tête de e : `push x (e, s) = (x :: e, s)`
- Pour extraire un élément :
 - si s est non vide, on extrait l'élément de tête de s : `pop (e, x :: xs) = (x, (e, xs))`
 - si s est vide et e non vide, on se ramène au cas précédent en remplaçant s par le miroir de e et e par la liste vide : `pop ([x_1; \dots; x_n], []) = pop ([], [x_n; \dots; x_1]))`
 - si les deux listes sont vides, l'opération échoue.

FIGURE 7.9 – La file $(1, 2, 3, 4, 5)$ représentée par le couple $([1; 2; 3], [5; 4])$.FIGURE 7.10 – La même file $(1, 2, 3, 4, 5)$ représentée par le couple $([1], [5; 4; 3; 2])$.**Exercice 7.1**

p. 108

1. Donner une série d'opérations permettant d'aboutir à situation de la figure 7.9 en partant d'une file vide.
2. Donner une série d'opérations permettant d'aboutir à situation de la figure 7.10 en partant d'une file vide.

3. Compléter le tableau suivant :

Instruction	e	s	x
	[1; 2; 3]	[5; 4]	
<code>let x, f = pop f</code>			
<code>let x, f = pop f</code>			
<code>let f = push 6 f</code>			
<code>let x, f = pop f</code>			
<code>let x, f = pop f</code>			
<code>let f = push 7 f</code>			
<code>let x, f = pop f</code>			

L'intérêt de cette réalisation est qu'une opération coûteuse (l'extraction d'un élément alors que la liste s est vide et que la liste e contient n éléments) est forcément compensée par une série d'opérations peu coûteuses. C'est le principe de la *complexité amortie* : au lieu de s'intéresser au coût maximal d'une opération (qui peut ici être proportionnel à la taille de la file), on majore le coût total d'une série de n opérations.

Théorème 7.1 – Complexité amortie pour une file fonctionnelle

Avec la réalisation décrite ci-dessus, le coût total d'une série de n opérations (insertions et extractions) sur une file initialement vide est en O(n).

Autrement dit, lors d'une série de n opérations, le coût moyen d'une opération est en O(1).

Démonstration

Se référer au TD. ■

2.3 File impérative

Tableau circulaire On fixe une capacité maximale pour la file à la création et l'on adapte la réalisation vue pour les piles. Il y a deux modifications à apporter :

- les *deux* extrémités de la file sont « actives ». Au lieu d'avoir un indice courant indiquant le sommet de la pile, on aura donc deux indices, disons *entrée* et *sortie*, pour indiquer les deux extrémités;
- si l'on fait une série de « insertion + extraction » le nombre d'éléments dans la file va rester constant mais la zone dans laquelle ils sont stockés va se déplacer dans le tableau. Pour éviter de se retrouver bloqué, on rend le tableau *circulaire* en considérant les indices modulo sa taille.

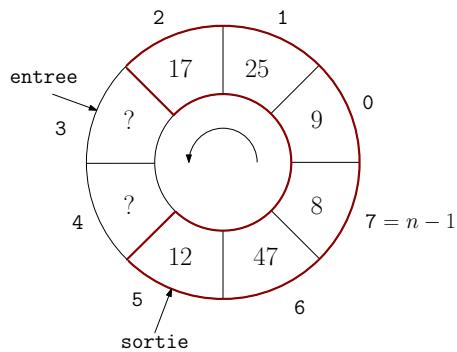


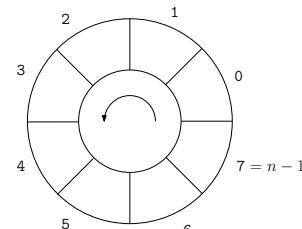
FIGURE 7.11 – La file $f = (17, 25, 9, 8, 47, 12)$ dans un tableau de capacité 8.

Exercice 7.2

p. 108

On exécute les instructions suivantes ; compléter le schéma.

```
push 10 f; print_int (pop f);
print_int (pop f); push 20 f
```



Liste mutable On adapte la réalisation des piles impératives avec des ‘a **list** ref. Comme les deux extrémités de la file sont « actives », la file contiendra deux pointeurs mutables : un vers la première cellule et un vers la dernière. Les cellules seront très similaires à celles d’une liste (un élément et un pointeur vers la cellule suivante), à ceci près que ce pointeur next sera *mutable*.

- Pour l’opération `pop`, on suit le pointeur `deb` pour accéder à la première cellule `c`, puis :
 - on récupère la valeur de l’élément `c.elt`;
 - on modifie le pointeur `deb` de la file pour qu’il pointe vers la cible de `c.next`.
- Pour l’opération `push`, on suit le pointeur `fin` pour accéder à la dernière cellule `d`, puis :
 - on crée une nouvelle cellule avec l’élément voulu et une valeur particulière « vide » pour `next`;
 - on modifie le pointeur `next` de la cellule `d` pour qu’il pointe vers cette nouvelle cellule;
 - on modifie le pointeur `fin` de la file pour qu’il pointe vers cette nouvelle cellule.

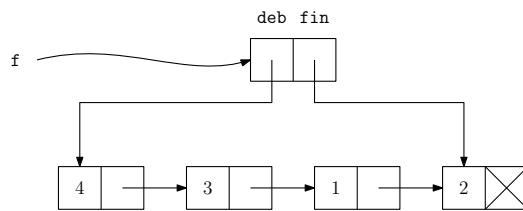


FIGURE 7.12 – La file $\leftarrow (4, 3, 1, 2) \leftarrow$.

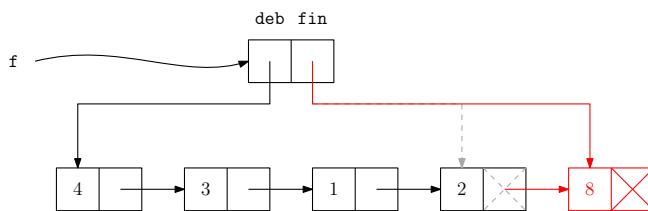


FIGURE 7.13 – Ajout de l’élément 8.

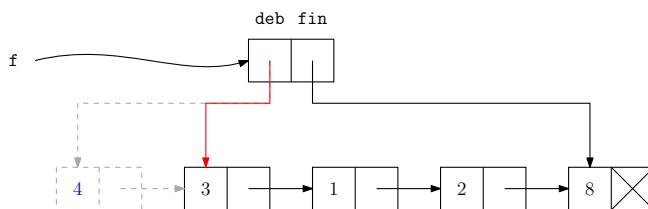


FIGURE 7.14 – Extraction d’un élément.

Remarques

- Cette réalisation de la structure de file est celle utilisée par le module **Queue** de OCaml, à quelques détails près.
- On a passé sous silence quelques difficultés (file vide, file ne contenant qu’un seul élément, choix des types OCaml).

Exercice 7.3

p. 109

1. Expliquer pourquoi on pourrait facilement modifier la structure de donnée de manière à disposer d’une fonction `length` en temps constant, et pourquoi cette solution ne s’applique pas vraiment au type **list** de OCaml.
2. On a choisi de faire les extractions en « tête de liste » et les insertions en « queue ». Expliquer pourquoi c’est en fait la seule possibilité.

3 File de priorité

Une file de priorité est une structure dans laquelle chaque élément est inséré avec une certaine *priorité* (typiquement un entier ou un flottant, mais en tout cas un élément d'un ensemble totalement ordonné). Quand on extrait un élément, on ne récupère pas le plus ancien ou le plus récent mais **celui ayant la priorité la plus forte** (ou celle qui a la priorité la plus forte, selon les variantes).

Opération	Type	Signification
empty_pq	unit -> ('a, 'b) pq	Crée une file vide
insert	('a * 'b) -> ('a, 'b) pq -> unit	Insère un élément avec une priorité
extract_max	('a, 'b) pq -> ('a * 'b)	Récupère l'élément de priorité maximale et l'enlève de la file

FIGURE 7.15 – Interface d'une file de priorité mutable

Il existe de nombreuses réalisations (fonctionnelles ou impératives) d'une file de priorité offrant des complexités en $O(\log n)$ (ou mieux) pour l'insertion et l'extraction du maximum. Nous aurons l'occasion d'utiliser cette structure, et d'en étudier des réalisations possibles, cette année.

4 Deque

Les *deques* (*Double ended queues*) sont une généralisation des files dans lesquelles on peut insérer et extraire par les deux extrémités :

Opération	Type	Signification
empty_deque	unit -> 'a deque	Crée une deque vide
is_empty	'a deque -> bool	Teste si la deque est vide
push_left	'a -> 'a deque -> unit	Rajoute un élément à gauche
push_right	'a -> 'a deque -> unit	Rajoute un élément à droite
pop_left	'a deque -> 'a	Récupère l'élément de gauche et l'enlève
pop_right	'a deque -> 'a	Récupère l'élément de droite et l'enlève

FIGURE 7.16 – Interface d'une deque mutable

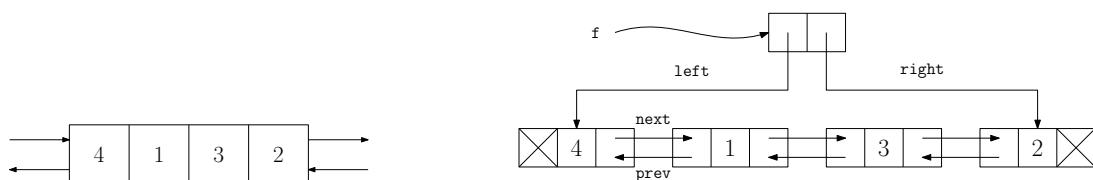


FIGURE 7.17 – La deque (4, 1, 3, 2) et sa réalisation sous forme de liste doublement chaînée.

- Pour réaliser une deque impérative, il y a deux possibilités :
 - adapter la version « tableau circulaire » des files impératives (c'est immédiat) ;
 - adapter la version « liste mutable » des files impératives, ce qui demande d'utiliser des listes *doublement chaînées* (où chaque cellule contient un pointeur vers la cellule suivante et un vers la cellule précédente). L'idée est simple, mais il faut être soigneux pour la mettre en pratique.
- On peut réaliser une deque fonctionnelle efficace en adaptant la réalisation fonctionnelle des files à l'aide de deux listes, mais c'est (très) loin d'être évident.

Solutions

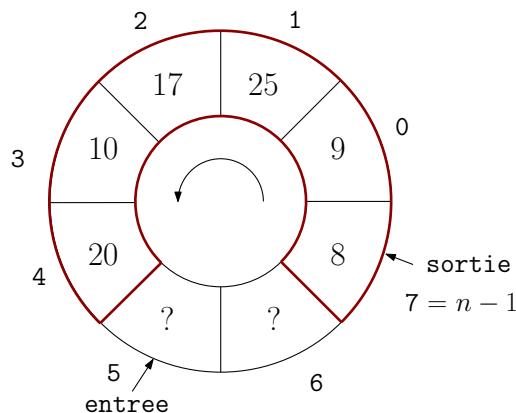
Correction de l'exercice 7.1 page 104

1. Il y a une infinité de possibilités, l'une des plus simples est :
 - on ajoute 6;
 - on ajoute 5;
 - on ajoute 4 (on obtient ([4; 5; 6], []));
 - on fait un pop, on récupère 6 et la nouvelle file est ([], [5; 4]);
 - on ajoute 3, puis 2, puis 1, on obtient ([1; 2; 3], [5; 4]).
2. À nouveau, on donne une solution simple parmi d'autres :
 - on ajoute successivement 6, 5, 4, 3 et 2, on obtient ([2; 3; 4; 5; 6], []);
 - on fait un pop, on récupère 6 et la file ([], [5; 4; 3; 2]);
 - on ajoute 1, on obtient ([1], [5; 4; 3; 2]).
- 3.

Instruction	e	f	x
	[1; 2; 3]	[5; 4]	
<code>let x, f = pop f</code>	[1; 2; 3]	[4]	5
<code>let x, f = pop f</code>	[1; 2; 3]	[]	4
<code>let f = push 6 f</code>	[6; 1; 2; 3]	[]	
<code>let x, f = pop f</code>	[]	[2; 1; 6]	3
<code>let x, f = pop f</code>	[]	[1; 6]	2
<code>let f = push 7 f</code>	[7]	[1; 6]	
<code>let x, f = pop f</code>	[7]	[6]	1

Correction de l'exercice 7.2 page 105

On obtient l'affichage 12 47 (on a mis une espace pour la lisibilité, il n'y en aurait pas en réalité), et la situation finale est :



Correction de l'exercice 7.3 page 106

1. On peut stocker un entier mutable `length` avec les deux pointeurs mutables `deb` et `fin`, et le mettre à jour quand on ajoute ou enlève un élément. Pour une liste, il faudrait stocker la longueur dans chaque cellule, ce qui serait certes possible mais très coûteux.
2. On peut sans difficulté faire une insertion « en tête de liste » : il suffit de créer une nouvelle cellule, de la faire pointer vers la valeur actuelle de `deb` puis de modifier `deb` pour qu'il pointe vers cette nouvelle cellule.
En revanche, faire une extraction « en queue de liste » est problématique : il faudrait faire pointer `fin` vers l'*avant-dernière* cellule (et modifier cette cellule). Le problème, c'est qu'on dispose seulement d'un pointeur vers la *dernière* cellule, et qu'on ne peut pas revenir à l'*avant-dernière* puisque il n'y a pas de pointeur d'une cellule vers la précédente. On serait donc obligé de parcourir toute la liste depuis le début.

GESTION DE LA MÉMOIRE

Sur les points touchant au fonctionnement réel d'un ordinateur et d'un système d'exploitation, ce chapitre est truffé de demi-vérités, d'approximations, de généralisations abusives... C'est sans importance pour nous : il faut juste avoir une idée, même très vague, de comment les choses se passent. Les points sur lesquels il faut des connaissances plus précises sont ceux touchant directement à la programmation.

1 Mémoire d'un ordinateur

1.1 Mémoire volatile et non volatile

Une mémoire est dite *volatile* si son contenu est perdu à chaque mise hors tension, *non volatile* dans le cas contraire. On parle aussi de *stockage de masse* pour la mémoire non volatile.

Les deux types de mémoire non volatile les plus couramment utilisés aujourd'hui sont la mémoire Flash (téléphones, SSD) et les disques durs. La capacité de cette mémoire varie typiquement entre quelques dizaines de giga-octets et quelques téra-octets (si on reste sur des ordinateurs personnels).

La mémoire volatile principale d'un ordinateur (ou d'un téléphone) est aussi appelée *mémoire vive*. Sa capacité varie typiquement entre quelques giga-octets et quelques dizaines de giga-octets.

1.2 Hiérarchie mémoire

Si l'on rentre plus dans les détails, un ordinateur possède en fait plusieurs types de mémoire volatile, qui constituent, avec la mémoire de masse, la *hiérarchie mémoire*.

- Le premier niveau de la hiérarchie mémoire est constitué des *registres* du processeur. C'est en quelque sorte la mémoire interne au processeur, et, pour simplifier, c'est là que les calculs sont effectivement effectués. Par exemple, pour additionner deux entiers situés quelque part en mémoire, il faut d'abord les charger dans des registres, puis effectuer l'opération. Le résultat est placé dans un registre et peut ensuite être écrit en mémoire.

Chaque registre a une taille fixée (disons 64 bits pour simplifier), et le nombre de registres est de l'ordre de quelques dizaines.

- Il y a ensuite plusieurs niveau de *cache* (trois, typiquement). Le premier niveau fait quelques dizaines de kilo-octets et contient une copie des données situées en mémoire vive auxquelles on a accédé très récemment : il permet d'y accéder très rapidement (quelques cycles). Les autres niveaux fonctionnent de la même manière, chacun ayant une capacité plus grande et une vitesse d'accès plus faible par rapport à celui qui précède.
- On passe ensuite à la mémoire vive proprement dite. Il faut plusieurs centaines de cycles pour accéder à une donnée se trouvant en mémoire vive mais n'étant présente dans aucun des caches.
- Le dernier niveau de la hiérarchie est constitué de la mémoire non volatile. S'il n'y a plus de place disponible en mémoire vive, le système d'exploitation peut utiliser le stockage de masse pour déplacer des données situées en mémoire vive qui n'ont pas été utilisées depuis longtemps. Si on veut ensuite accéder à ces données, il faudra commencer par les recharger en mémoire (en faisant de la place si nécessaire), ce qui est très long (en comparaison d'un accès mémoire normal).

La gestion de la hiérarchie mémoire n'est pas directement du ressort du programmeur :

- l'utilisation des registres est gérée par le compilateur (sauf si on écrit du code en assembleur);
- la gestion des caches est faite par le processeur;
- la gestion du *swap* (utilisation du stockage de masse) est faite par le système d'exploitation, dans le cadre de la gestion de la mémoire virtuelle.

1.3 Mémoire virtuelle

On oublie à présent les caches (qui sont gérés directement par la machine) et les registres (qui ont un rôle différent). Chaque octet de la mémoire vive d'un ordinateur possède une *adresse physique*, qui est tout simplement un entier positif, entre zéro et le nombre d'octets de la mémoire pour simplifier. C'est cette adresse qui, physiquement, va être placée sur des fils pour demander au module de mémoire de la lire et de la transmettre au processeur. Cependant, dans un système moderne (au sens très large), ces adresses physiques sont complètement invisibles pour les programmes, et ne sont gérées que par le matériel et le système d'exploitation.

Les processus (les programmes, essentiellement) qui s'exécutent ne manipulent que des *adresses virtuelles*. Ces adresses virtuelles sont toujours des entiers, mais elles peuvent prendre n'importe quelle valeur entre 0 et $2^{64} - 1$ (sur une machine 64 bits, en passant beaucoup de choses sous le tapis), ce qui est essentiellement infini en pratique. Cela a plusieurs avantages :

- une adresse virtuelle peut correspondre à une adresse physique en mémoire vive, à des données qui se trouvent physiquement sur le disque (parce qu'on n'a plus de place en mémoire, par exemple), ou à rien du tout (ce qui, à un moment donné, sera le cas de l'immense majorité des adresses virtuelles) ;
- la liaison entre adresses physiques et adresses virtuelles est différente pour chaque processus. Ainsi, chaque processus vit comme s'il était seul en mémoire, ce qui simplifie les choses et, surtout, améliore la sécurité.

2 Organisation de la mémoire d'un processus

Schématiquement, la mémoire virtuelle d'un processus est organisée ainsi :

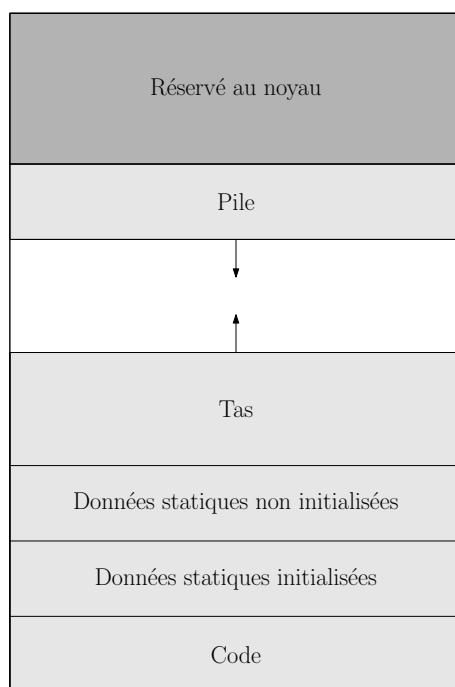


FIGURE 8.1 – Organisation (très simplifiée) de la mémoire virtuelle.

- Toute une partie de l'espace d'adresses est réservé au noyau (au « cœur » du système d'exploitation) et le processus n'y a pas directement accès.
- Les trois sections situées en bas sont créées au lancement du programme, et leur taille est fixée à ce moment là.
 - La section « Code » (qui s'appelle souvent `.text` en réalité) contient le programme à proprement parler, c'est-à-dire les instructions à exécuter par le processeur. Cette section est normalement en *lecture seule* (elle ne peut pas être modifiée durant l'exécution).
 - La section pour les données initialisées contient les variables globales du programme qui ont été initialisées lors de leur définition. Elle est en réalité séparée en deux, car certaines de ces variables peuvent ensuite être modifiées et d'autres non (variables notées `const` en C).

- La section pour les données non initialisées contient les variables globales qui ont été définies, mais pas initialisées. En C, ces variables sont automatiquement initialisées à zéro.
- Le *tas* servira à stocker les données *allouées dynamiquement*, c'est-à-dire pendant l'exécution du programme. En C, on alloue de la mémoire sur le tas à l'aide de la fonction `malloc` et on la libère avec `free`. La taille du tas évolue au cours de l'exécution du programme.
- La *pile*, ou *pile d'appel* est une structure constituée de *blocs d'activation* (on parle plus couramment de *stack frames*). À tout moment, il y a un bloc d'activation sur la pile pour chaque appel de fonction actif (déjà commencé, pas encore terminé). Chaque bloc contient les (ou plutôt certaines des) variables locales de la fonction, ainsi que des informations permettant de continuer l'exécution du programme une fois l'appel terminé. La pile est donc aussi de taille variable.

Considérons un petit programme en C :

```
#include <stdio.h>

const int N = 3;
int P = 7;
double X;

void f(int n){
    printf("%d\n", n + N);
}

int main(void){
    X = 3.14;
    printf("%f %d\n", X, P);
    f(12);
    P = 9;
    printf("%d\n", P);
    return 0;
}
```

On peut le compiler (première ligne de commande), puis étudier la structure du fichier créé :

```
$ gcc -o memoire -c memoire.c
$ objdump -s memoire
memoire:      file format elf64-x86-64

Contents of section .text:
0000 554889e5 4883ec10 897dfcba 03000000  UH..H.....}.....
0010 8b45fc01 d089c648 8d050000 00004889  .E.....H.....H.
0020 c7b80000 0000e800 00000090 c9c35548 .....UH
0030 89e5f20f 10050000 0000f20f 11050000 .....
0040 00008b15 00000000 488b0500 00000089 .....H.....
0050 d666480f 6ec0488d 05000000 004889c7 .fH.n.H.....H..
0060 b8010000 00e80000 0000bf0c 000000e8 .....
0070 00000000 c7050000 00000900 00008b05 .....
0080 00000000 89c6488d 05000000 004889c7 .....H.....H..
0090 b8000000 00e80000 0000b800 0000005d .....
00a0 c3          .
Contents of section .data:
0000 07000000      ...
Contents of section .rodata:
0000 03000000 25640a00 25662025 640a0000 ....%d..%f %d...
0010 1f85eb51 b81e0940      ...Q...@
```

Plus des informations non pertinentes pour nous

La section `.text` est vraiment complètement illisible : il s'agit de code machine. On peut éventuellement la *désassembler* pour obtenir *l'assembleur* ; ici, on va juste demander de le faire pour la fonction `f`.

```
$ objdump --disassembly= f memoire

memoire:      file format elf64-x86-64

Disassembly of section .init:
Disassembly of section .plt:
Disassembly of section .text:

0000000000001139 <f>:
1139:   55          push    %rbp
113a: 48 89 e5      mov     %rsp,%rbp
113d: 48 83 ec 10  sub    $0x10,%rsp
1141: 89 7d fc      mov     %edi,-0x4(%rbp)
1144: ba 03 00 00 00  mov    $0x3,%edx
1149: 8b 45 fc      mov    -0x4(%rbp),%eax
114c: 01 d0          add    %edx,%eax
114e: 89 c6          mov    %eax,%esi
1150: 48 8d 05 b5 0e 00 00  lea    0xeb5(%rip),%rax
1157: 48 89 c7      mov    %rax,%rdi
115a: b8 00 00 00 00  mov    $0x0,%eax
115f: e8 cc fe ff ff  call   1030 <printf@plt>
1164: 90              nop
1165: c9              leave
1166: c3              ret
```

3 Portée d'un identifiant

Dans tout langage de programmation, un identifiant (un « nom de variable ») a une *portée*, c'est-à-dire une partie du programme dans lequel il a un sens et où on peut y faire référence. Certains langages utilisent une portée *dynamique*, qui ne peut être connue qu'à l'exécution, mais ils sont aujourd'hui extrêmement rares : C et OCaml (et Python, même si c'est un peu différent) utilisent une portée *statique*, déterminée à la compilation.

Pour faire simple, la portée peut être :

- *globale*, c'est-à-dire tout le fichier. Dans ce cas, elle commence à la ligne où le nom est défini et s'arrête à la fin du fichier.
- *locale* à un *bloc*. Les cas les plus courants sont ceux d'un nom local à une fonction (c'est en particulier le cas des paramètres) et d'un nom local à une boucle.

Dans un langage compilé à portée statique (comme C ou OCaml), faire référence à un nom en dehors de sa portée est une erreur qui sera détectée à la compilation. En Python c'est également une erreur, mais elle ne sera détectée qu'à l'exécution.

```

1 #include <stdio.h>
2
3 int n;
4
5 double pi = 3.14;
6
7 int fact(int n){
8     int f = 1;
9     for (int i = 1; i <= n; i = i + 1){
10         f = f * i;
11     }
12     return f;
13 }
14
15 int main(void){
16     int p = 10;
17     n = 3;
18     printf("%d! = %d\n", n, fact(n));
19     printf("%d! = %d\n", p, fact(p));
20     return 0;
21 }
```

Ici :

- la portée de `pi` commence ligne 5 et va jusqu'au bout du programme;
- la portée de `f` commence à la ligne 8 et s'arrête à l'accolade fermante ligne 13;
- la portée de `i` commence à sa définition ligne 9 et s'arrête à l'accolade fermante ligne 11;
- la portée de `p` commence à la ligne 16 et s'arrête à l'accolade fermante ligne 21;
- le nom `n` a deux portées distinctes : une portée globale qui commence ligne 3 et va jusqu'à la fin du fichier, et une portée locale qui commence ligne 7 et s'arrête à l'accolade fermante ligne 13. Dans ce cas, il y a un phénomène de *masquage* : une référence au nom `n` désignera toujours l'objet correspondant à la portée la plus interne. Autrement dit, comme on pouvait s'y attendre, à l'intérieur de la fonction `f` le nom `n` correspond au paramètre de la fonction, mais à l'extérieur il fait référence à la variable globale.

Portée et initialisation En C, on peut séparer la définition d'un identifiant de l'initialisation de la variable qui lui correspond. Par exemple, on aurait pu remplacer la ligne 16 par `int p;` sans rien modifier d'autre, et le programme aurait compilé. Cependant, c'est à proscrire : le comportement du programme est alors non défini et quand on lit `p` à la ligne 19, on peut obtenir n'importe quelle valeur. C'est un peu différent pour une variable globale (qui est automatiquement initialisée à zéro), mais cela reste une mauvaise idée.

Blocs Toute série d'instructions entourée par des accolades (en particulier le corps d'une boucle, mais pas seulement) définit un *bloc*, et un nom défini à l'intérieur de ce bloc est local à ce bloc. Par exemple :

```

int i = 3;
int j = 2;
while (j > 0){
    int i = 0;
    j = j - 1;
}
printf("%d", i); // i vaut 3 ici
```

De même, le code suivant est faux (il ne compile même pas) :

```
1 if (x < 0){
2     int signe = -1;
3 } else {
4     int signe = 1;
5 }
printf("%d", signe); // signe n'existe pas !!!
```

Fonctions récursives En OCaml, il faut utiliser **let rec** pour définir une fonction récursive, car sinon la portée de **f** ne contient pas sa définition.

```
1 let f x = x + 1
2
3 let f n =
4   if n = 0 then 17 else 2 * f (n / 2)
5
6 let p = f 3
```

Le code ci-dessus est très étrange, mais il est accepté :

- à la ligne 4, **f** désigne la fonction définie ligne 1 puisque on n'est pas encore dans la portée du **f** défini ligne 3;
- en revanche, à la ligne 6, le **f** désigne la fonction définie à la ligne 11;
- finalement, **p** vaudra 4.

Avec un **let rec**, la portée de l'identifiant inclut sa définition :

```
1 let f x = x + 1
2
3 let rec f n =
4   if n = 0 then 17 else 2 * f (n / 2) (* appel récursif *)
5
6 let p = f 3 (* p vaut 68 *)
```

En C, le comportement par défaut est celui obtenu avec **let rec** en OCaml : la portée inclut la définition. Il n'y a donc pas de syntaxe particulière pour définir une fonction récursive :

```
int fact(int n){
    if (n == 0) return 1;
    return n * fact(n - 1);
}
```

Notez (mais oubliez-le au plus vite!) que cela rend « légal » une définition **int n = n**. Cela n'a en revanche aucun sens : la valeur de **n** est quelconque.

4 Pointeurs en C

Comme nous l'avons vu, OCaml, comme essentiellement tous les langages, utilise des *pointeurs* : par exemple, une cellule de liste est constituée de la valeur (qui peut éventuellement être un pointeur) et d'un pointeur vers la cellule suivante (avec un cas particulier si on est en bout de liste). Cependant, ces pointeurs sont « cachés » : le programmeur ne peut pas les manipuler explicitement.

En C, en revanche, il est possible de récupérer l'adresse d'un objet et de créer un autre objet qui aura pour valeur cette adresse. On dira alors que ce deuxième objet est un *pointeur* vers le premier objet.

Il y a deux opérateurs fondamentaux :

- l'opérateur *address-of*, noté **&**. **&a** renvoie un pointeur vers l'objet **a**.

- l'opérateur de *déréférencement* (ou d'*indirection*), noté * : si p est un pointeur, alors *p permet d'accéder à la valeur vers laquelle il pointe.

Si t est un type (par exemple `int`, `double`...), alors *t est le type « pointeur vers t ». Ainsi :

- si x est de type t, alors &t est de type t*;
- si p est de type t*, alors *p est de type t.

Exemple 8.1

```
#include <stdio.h>

int main(void){
    int x = 12;
    int y = 100;
    int *p = &x;
    printf("x = %d\n", x);
    printf("p = %lld\n", p);
    printf("p = %p\n", p);
    printf("&x = %p\n", &x);
    *p = 15;
    printf("x = %d\n", x);
    printf("p = %p\n", p);
    x = 7;
    printf("x = %d\n", x);
    printf("p = %p\n", p);
    p = &y;
    printf("x = %d\n", x);
    printf("p = %p\n", p);
    printf("*p = %d\n", *p);
    return 0;
}
```

```
$ ./pointeurs
x = 12
p = 140724646564568
p = 0x7ffd0291d2d8
&x = 0x7ffd0291d2d8
x = 15
p = 0x7ffd0291d2d8
x = 7
p = 0x7ffd0291d2d8
x = 7
p = 0x7ffd0291d2dc
*p = 100
```

Passage par valeur En C, les paramètres d'une fonction sont toujours passés *par valeur*. Autrement dit, quand on passe un entier x à une fonction, c'est la *valeur* de x qui lui est transmise, et pas son adresse : cela signifie que les éventuelles modifications apportées à x par la fonction ne seront pas visibles de l'extérieur :

Exemple 8.2

```
#include <stdio.h>

int f(int x){
    x = x - 1;
    return x;
}

int main(void){
    int x = 4;
    int y = f(x);
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    return 0;
}
```

```
$ ./pass_by_value
x = 4
y = 3
```

Cependant, rien n’empêche la valeur du paramètre passé à la fonction d’être un pointeur ! Dans ce cas, il devient possible de modifier la valeur pointée.

Exemple 8.3

```
#include <stdio.h>

void echange(int* x, int* y){
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void){
    int a = 3;
    int b = 7;
    echange(&a, &b);
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    return 0;
}
```

```
$ ./echange
a = 7
b = 3
```

5 Durée de vie d'un objet

Un *objet* en C (on pourra simplement parler de *variable*) est une zone mémoire et possède un type. Un objet a une adresse fixe tout au long de sa vie. Sa *durée de vie* est la période durant laquelle il existe en mémoire. Cette durée de vie peut être :

- *statique* si l’objet existe en mémoire durant toute l’exécution du programme. C’est le cas des variables globales (elles existent même en mémoire avant que leur portée ne débute) ;
- *automatique* si l’objet n’existe en mémoire que pendant l’exécution d’un bloc (d’une fonction, le plus souvent) ;
- *allouée* pour les objets alloués et désalloués sur le tas à la demande (appels à `malloc` et `free`) .

Si l’on accède à un objet alors qu’il n’est plus « vivant », le comportement du programme est indéfini. La manière la plus simple de faire cela, c’est d’écrire une fonction qui renvoie un pointeur vers l’un de ses objets locaux (qui ne vivent que jusqu’à la fin de l’appel).

Exemple 8.4

Si l’exécution du programme suivant transforme votre ordinateur en dragon à trois têtes, ne venez pas vous plaindre : *Undefined Behavior*, ça veut dire *undefined*.

```
#include <stdio.h>
int* f(void){
    int x = 3;
    return &x;
}

int main(void){
    int *p = f();
    printf("p = %p\n", p);
    printf("*p = %d\n", *p);
    return 0;
}
```

```
$ ./lifetime
p = (nil)
[1] 484869 segmentation fault
```

6 Pile d'appels

Dans cette partie, on utilisera autant OCaml que C pour illustrer les concepts : certaines questions ne se posent qu'en C, mais la plupart sont les mêmes dans tous les langages.

6.1 Principe et nécessité

Considérons les deux fonctions suivantes¹ :

```

1  let g n =
2    n * n
3
4  let f n =
5    let s = ref 0 in
6    let k = ref 1 in
7    while !k <= n do
8      s := !s + g !k; (* .L104 *)
9      incr k
10   done;
11   !s

```

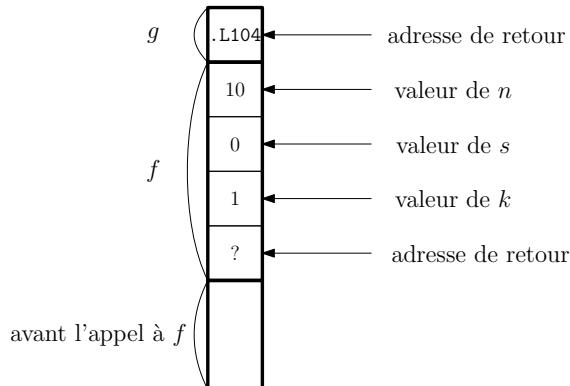


FIGURE 8.2 – État de la pile d'appels lors du premier passage ligne 8 dans le calcul de $f\ 10$.

Que se passe-t-il « en vrai » quand on calcule $f\ 2?$ f crée des variables locales, leur donne une valeur, puis appelle g avec l'argument 1. Pour cela, elle cède le contrôle d'exécution à g (et l'exécution « saute » au début du code de g). Quand g aura fini son calcul, il faudra qu'elle rende la main à f : mais l'exécution de f doit reprendre là où elle s'est arrêtée, et dans l'environnement qui était valable à ce moment (s doit valoir 0, k doit valoir 1...). Il faut donc que f sauvegarde un certain nombre d'informations avant d'appeler g ².

Cette sauvegarde d'informations se fait sur ce qu'on appelle la *pile d'appels*. Vu de loin, un élément (on parle de *stack frame* ou *bloc d'activation*) de la pile contient toutes les informations relatives à un appel donné d'une fonction donnée :

- le stockage local de la fonction ;
- le point auquel l'exécution du programme devra reprendre quand l'appel sera terminé (c'est le plus important!) ;
- éventuellement, tout ou partie des arguments de la fonction (qui peuvent être passés soit sur la pile, soit dans des registres).

Quand une fonction est appelée, elle crée une *stack frame* au sommet de la pile, qu'elle supprimera juste avant de renvoyer son résultat et de passer le contrôle au point qui était sauvegardé. À tout moment de l'exécution du programme, la hauteur de la pile (en nombre de *frames*) est donc égale au nombre de fonctions actives (c'est-à-dire ayant été appelées et n'ayant pas encore retourné leur résultat).

Une remarque sur la terminologie : le nom de « pile » n'est bien sûr pas anodin, il y a bien une très forte analogie avec la structure de données que nous avons vu au chapitre précédent. On empile au moment des appels, on dépile au moment des retours. Seul le bloc d'activation situé au sommet de la pile est actif (la fonction n'a pas accès aux variables locales de la fonction appelante!). En revanche, on peut accéder sans problème (et sans avoir besoin de dépiler) à n'importe quelle variable située dans ce bloc au sommet.

6.2 Un coup d'œil sous le capot

On peut demander à OCaml de nous donner l'assembleur généré à la compilation d'un fichier. Dans les multiples transformations que subit un programme avant de pouvoir être exécuté, celle qui fournit l'assembleur est essentiellement l'avant-dernière : le programme est encore lisible par un humain (avec un peu de bonne volonté), mais sa traduction en langage machine est « triviale ».

1. On a utilisé un **while** au lieu d'un **for** car l'assembleur généré est plus facilement lisible.

2. Tout ou partie de cette sauvegarde peut être faite par g plutôt que par f , mais cela ne change rien au problème.

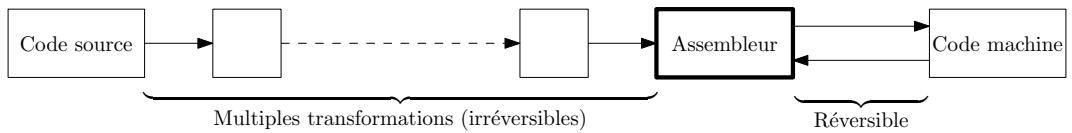


FIGURE 8.3 – Étapes de la compilation d'un programme OCaml.

```

0000000000417050 <camlAsm1_g_8>:
417050: 48 89 c3          mov    rbx,rax
417053: 48 d1 fb          sar    rbx,1
417056: 48 ff c8          dec    rax
417059: 48 0f af c3          imul   rax,rbx
41705d: 48 ff c0          inc    rax
417060: c3                  ret

```

FIGURE 8.4 – Récupération de l'assembleur (à droite) à partir du code machine (au milieu).

Voyons donc ce que produit OCaml sur cet exemple (l'assembleur a été nettoyé et simplifié) :

```

Fonction_g:
.L100:
    movq    %rax, %rbx      ; On élève l'argument au carré
    ...
    imulq   %rbx, %rax      ; (c'est un peu compliqué pour des
                           ; raisons techniques).
    ret                 ; On renvoie et l'on saute au point de retour sauvegardé.

Fonction_f:
    subq    $24, %rsp      ; On réserve de la place sur la pile pour 3 entiers.
.L103:
    ...
                           ; Je saute l'initialisation.

.L102:
    movq    (%rsp), %rdi      ; On charge n dans %rdi.
    cmpq    %rdi, %rax      ; On compare n et k.
    jg     .L101            ; Si k > n, on sort de la boucle.
    call    Fonction_g        ; Sinon, on appelle g (en sauvegardant le point de retour)
.L104:
    movq    8(%rsp), %rbx      ; On restaure s.
    addq    %rax, %rbx      ; On ajoute à s le résultat de g.
    ...
    movq    %rbx, 8(%rsp)      ; On sauvegarde s (pour le prochain appel).
    movq    16(%rsp), %rax      ; On restaure k.
    addq    $2, %rax       ; On incrémente k.
    movq    %rax, 16(%rsp)      ; On sauvegarde k (pour le prochain appel).
    jmp     .L102            ; On retourne au début de la boucle.

.L101:
    movq    %rbx, %rax      ; On prépare le résultat.
    addq    $24, %rsp      ; On libère la pile.
    ret                 ; On renvoie et l'on saute au point de retour sauvegardé.

```

6.3 Pile et fonctions récursives

Pour l'instant, on a considéré le cas d'une fonction *f* qui appelle une fonction *g*. Mais rien n'empêche bien sûr *f* de s'appeler elle-même³ : ce sera le cas si *f* est récursive. Dans ce cas, il y aura *un bloc d'activation par appel actif de f*. C'est logique, puisque chacun de ses appels possède ces propres variables locales, et son propre compteur de programme (chacun des appels en est à un point différent de son exécution).

Pour visualiser ce type de situation, il est plus intéressant de réfléchir en termes d'*arbres d'appels*. Considérons une fonction calculant les termes de la suite de Fibonacci de manière récursive et naïve⁴ :

```
let rec fib n = if n <= 1 then n else fib (n - 2) + fib (n - 1)
```

Un appel à *fib 5* produit un appel à *fib 4* et un à *fib 3*, qui produisent eux-mêmes d'autres appels... La situation est très bien résumée par l'arbre d'appels suivant :

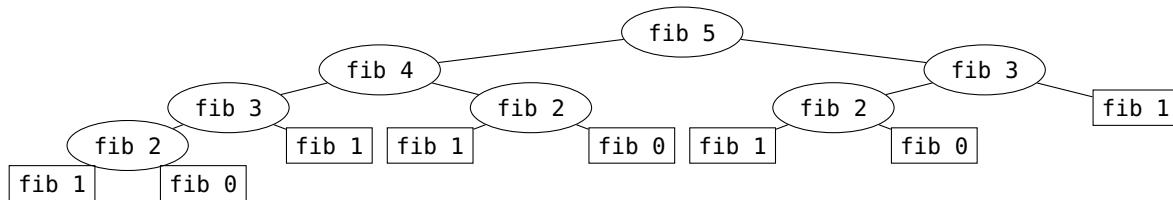


FIGURE 8.5 – Arbre d'appel pour *fib 5*

```

fib 5
fib 5 | fib 4
fib 5 | fib 4 | fib 3
fib 5 | fib 4 | fib 3 | fib 2
fib 5 | fib 4 | fib 3 | fib 2 | fib 1
fib 5 | fib 4 | fib 3 | fib 2
fib 5 | fib 4 | fib 3 | fib 2 | fib 0
fib 5 | fib 4 | fib 3 | fib 2
fib 5 | fib 4 | fib 3
fib 5 | fib 4 | fib 3 | fib 1
fib 5 | fib 4 | fib 3
fib 5 | fib 4
fib 5 | fib 4 | fib 2
fib 5 | fib 4 | fib 2 | fib 1
--> fib 5 | fib 4 | fib 2
fib 5 | fib 4 | fib 2 | fib 0
fib 5 | fib 4 | fib 2
fib 5 | fib 4
fib 5
fib 5 | fib 3
fib 5 | fib 3 | fib 2
fib 5 | fib 3 | fib 2 | fib 1
fib 5 | fib 3 | fib 2
fib 5 | fib 3 | fib 2 | fib 0
fib 5 | fib 3 | fib 2
fib 5 | fib 3
fib 5 | fib 3 | fib 1
fib 5 | fib 3
fib 5

```

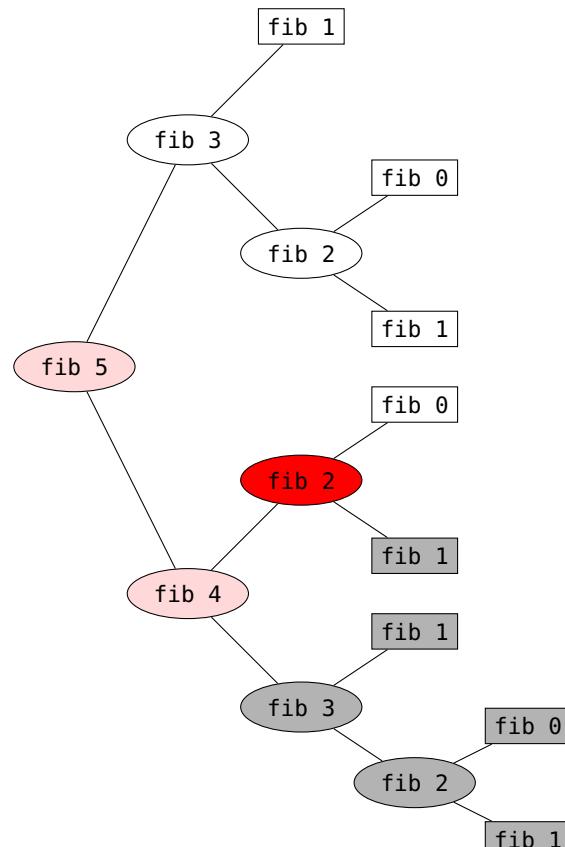


FIGURE 8.7 – Appels en cours et terminés pour la ligne marquée de la figure 8.6.

FIGURE 8.6 – Évolution de la pile d'appels lors du calcul de *fib 5*.

3. Directement ou indirectement : *f* peut appeler *g* qui appelle *f*.

4. Ici, « naïve » est à comprendre au sens de *scandaleusement mauvais et d'ailleurs contraire aux bonnes mœurs*.

Il faut avoir de cet arbre une vision dynamique : on en effectue un parcours en profondeur. Quand on est en train de visiter un nœud, les appels actifs sont ceux correspondant au nœud actuel ainsi qu'à tous ses ancêtres (*i.e.* tous les nœuds situés sur le chemin qui le relie à la racine). C'est ce qui est illustré par les figures 8.6 et 8.7.

Notons que l'on peut très facilement obtenir une « trace » des appels à une fonction dans la boucle interactive de Caml. Une fois que l'on a défini une fonction (disons fibo) il suffit d'exécuter #trace fibo (avec le #!) pour qu'un appel à fibo affiche tous les appels engendrés.

```
# #trace fibo;;
fibo is now traced.
# fibo 4;;
fibo <-> 4 # appel à fibo 4
fibo <-> 3 # appel à fibo 3
fibo <-> 2 # appel à fibo 2
fibo <-> 1 # appel à fibo 1
fibo --> 1 # fibo 1 -> 1
fibo <-> 0
fibo --> 0
fibo --> 1
fibo <-> 1
fibo --> 1
```



```
fibo --> 2 # fibo 3 -> 2
fibo <-> 2 # appel à fibo 2
fibo <-> 1 # ...
fibo --> 1 # ...
fibo <-> 0 # ...
fibo --> 0 # ...
fibo --> 1 # qui finit par renvoyer 1
fibo --> 3 # l'appel initial renvoie
→ 3
- : int = 3
# #untrace fibo;;
fibo is no longer traced.
# fibo 4;;
- : int = 3
```

La hauteur maximale de la pile est égale à la hauteur de l'arbre. Ici, cette hauteur est de l'ordre de n alors que le temps d'exécution est exponentiel. Cependant, si l'on s'intéresse à une fonction ayant une structure d'appels plus simple, les choses peuvent être différentes :

```
let rec somme n =
  if n = 0 then 0
  else n + somme (n - 1)
```

Ici, l'arbre d'appels obtenu est en fait linéaire :



FIGURE 8.8 – Arbre d'appels pour somme 6.

Le problème, qui n'est pas forcément évident quand on regarde le code, est qu'un appel à somme n demande un espace mémoire proportionnel à n : c'est la hauteur maximale de la pile d'appels. De plus, l'espace disponible sur la pile est bien plus limité que l'espace mémoire total. Le résultat :

```
# somme 1_000_000;;
Stack overflow during evaluation (looping recursion?).
```

Notons que le temps de calcul n'est pas le problème : le *stack overflow* (dépassement de pile) se produit presque instantanément. Ce qu'il faut retenir :

Propriété 8.1

La complexité en espace d'une fonction récursive est au moins égale à sa profondeur maximale de récursion (*c'est-à-dire* à la profondeur de son arbre d'appels).

Remarques

- Elle peut bien sûr être supérieure à cette profondeur, typiquement si la fonction crée des listes ou des tableaux auxiliaires.

- Par défaut, l'espace disponible sur la pile est assez faible (de l'ordre de quelques méga-octets). On peut donc faire un *stack overflow* bien avant d'épuiser la mémoire disponible.

Ici, cette complexité linéaire en espace est problématique : une fonction itérative aura clairement une utilisation mémoire constante (sauf si l'on s'amuse à stocker tous les résultats intermédiaires...) :

```
let somme_iterative n =
  let s = ref 0 in
  for k = 1 to n do
    s := !s + k
  done;
  !s
```

Et le problème disparaît :

```
# somme_iterative 1_000_000;;
- : int = 500000500000
```

6.4 Récursion terminale

6.4.a Principe

En Python, on s'arrêterait là : si une fonction a une structure d'appel aussi simple, il est facile de la « dé-récurser », et on évite les problèmes de *stack overflow*. Cependant, la programmation fonctionnelle, et donc le OCaml, se prête plus facilement à un style récursif qu'itératif. Par conséquent, le compilateur OCaml procède à une optimisation appelée *élimination de la récursion terminale*.

Remarque

gcc et clang procèdent également tous les deux à cette optimisation. Cependant, elle n'est pas garantie par le standard C, et il n'est pas *a priori* impossible que le compilateur la « manque » dans certains cas, par exemple parce qu'il a d'abord procédé à une autre optimisation (je ne sais pas si cela arrive en pratique). En OCaml, l'élimination est garantie.

Si l'on regarde le code de la fonction somme, on voit qu'il est indispensable de sauvegarder le contexte avant de procéder à l'appel récursif. En effet, le résultat de cet appel devra être ajouté à n (dont il faut donc se rappeler la valeur) avant d'être renvoyé par la fonction appelante. En revanche, considérons la fonction suivante :

```
let somme_terminale n =
  let rec somme_aux n acc =
    if n > 0 then somme_aux (n - 1) (acc + n)
    else acc
  in
  somme_aux n 0
```

La fonction somme_terminale n'est pas récursive (et n'est appelée qu'une fois), on peut donc l'ignorer. La fonction somme_aux, qui effectue véritablement le calcul, n'a qu'un seul appel récursif, **dont elle renvoie immédiatement le résultat**. Comme elle n'aura aucun travail à effectuer une fois l'appel récursif terminé, il est totalement inutile de sauvegarder son contexte. Le compilateur le remarque, et elle s'exécute maintenant en espace constant :

```
# somme_terminale 1_000_000;;
- : int = 500000500000
```

Définition 8.2

Une fonction est dite *récursive terminale* (ou *tail recursive*) si les seuls appels récursifs qu'elle contient sont placés en position finale, c'est-à-dire si elle renvoie immédiatement le résultat sans faire d'opération dessus.

En Caml, une telle fonction sera automatiquement compilée en une version non récursive et ne pourra donc jamais donner lieu à un dépassement de pile.

Remarques

- La plupart des langages fonctionnels font la même optimisation que Caml, car la récursion y est souvent utilisée pour faire ce genre de calculs (dans certains langages, c'est même la seule possibilité). En revanche, dans un langage comme Python, le fait qu'une fonction récursive soit terminale ou pas ne change rien : si l'on veut écrire une fonction `max` qui fonctionne sur des listes « longues », on est donc obligé de le faire avec une boucle.
- Une fonction récursive terminale n'est en quelque sorte « pas vraiment récursive » : il est en effet très facile de la transformer en une fonction non récursive utilisant juste un accumulateur et une boucle `while` (c'est ce que le compilateur fait automatiquement).
- Toute fonction récursive peut être rendue terminale (et donc dé-récurcifiée), mais le plus souvent cela revient simplement à programmer « à la main » la pile d'appels, ce qui a pour principal intérêt de rendre le code complètement incompréhensible.

6.4.b Encore un peu d'assembleur

On reprend les deux fonctions récursives calculant la somme des n premiers entiers :

```
let rec somme n =
  if n > 0 then
    n + somme (n - 1)
  else
    0
```

```
let somme_terminale n =
  let rec somme_aux n acc =
    if n > 0 then
      somme_aux (n - 1) (acc + n)
    else
      acc
  in
  somme_aux n 0
```

On considère aussi une version itérative, et l'assembleur généré par OCaml pour cette version :

```
let somme_iter n =
  let s = ref 0 in
  let k = ref n in
  while !k > 0 do
    s := !s + !k;
    decr k
  done;
  !s
```

Fonction somme_iter:

```
.L108:
  movq $1, %rbx          ; s <- 0
.L107:
  cmpq $1, %rax          ; si k <= 0
  jle .L106               ; aller en L106
  leaq -1(%rbx,%rax), %rbx ; sinon, s <- s + k
  addq $-2, %rax          ; k <- k - 1
  jmp .L107                ; aller en L107
.L106:
  movq %rbx, %rax          ; on se prépare à renvoyer s
  ret                      ; on renvoie
```

Les deux fonctions récursives sont compilées très différemment, mais le point crucial est que la version récursive terminale est compilée exactement de la même manière que la version itérative.

```

Fonction somme:
    subq    $8, %rsp           ; on réserve une place sur la pile
.L102:
    cmpq    $1, %rax          ; si n <= 0
    jle     .L101             ; alors on va en L101
    movq    %rax, (%rsp)      ; sinon, on sauvegarde n
    addq    $-2, %rax         ;
    call    somme             ; puis on appelle somme (n - 1)
.L100:
    movq    (%rsp), %rbx      ; on restaure n
    leaq    -1(%rbx,%rax), %rax ; on l'ajoute au résultat de l'appel
    addq    $8, %rsp          ; on libère la pile
    ret                 ; et on renvoie
.L101:
    movq    $1, %rax          ; on se prépare à renvoyer 0
    addq    $8, %rsp          ; on libère la pile
    ret                 ; on renvoie

Fonction somme_aux:
.L113:
    cmpq    $1, %rax          ; si n <= 0
    jle     .L112             ; on va en L112
    leaq    -1(%rbx,%rax), %rbx ; sinon, acc <- acc + n
    addq    $-2, %rax         ; n <- n - 1
    jmp     .L113             ; on va en L113
.L112:
    movq    %rbx, %rax        ; on se prépare à renvoyer acc
    ret                 ; on renvoie

Fonction somme_term:
.L110:
    movq    $1, %rbx          ; acc <- 0
    jmp     L113              ; on appelle somme_aux (appel terminal)

```

Les deux manières de calculer la somme sont compilées très différemment. En revanche, la version terminale est rigoureusement identique à ce qu'on obtient en compilant la version itérative de la fonction :

6.4.c Intérêts de la récursion terminale

En règle générale, une version récursive terminale d'une fonction ne s'exécutera pas plus vite (même pas par un facteur constant) qu'un version non terminale. Sauf demande explicite, il est donc inutile de chercher absolument à trouver une telle version, sauf si l'on est dans l'un des cas suivants :

- on trouve cela plus naturel (cela dépend des gens et des fonctions, mais cela peut par exemple arriver pour le maximum ou la longueur d'une liste);
- la version récursive terminale a une complexité inférieure à la version non terminale (miroir d'une liste, par exemple);
- on sait qu'on risque d'avoir des profondeurs de récursion très grandes en pratique (concaténation de deux listes dont la première est très longue, par exemple);
- la complexité en espace de la version non terminale n'est pas satisfaisante. Il s'agit essentiellement d'une reformulation du point précédent : calculer la somme des n premiers entiers par une fonction récursive « naïve » demande un espace de travail proportionnel à n (alors que les versions itératives et terminales sont toutes deux en espace constant).

6.4.d Inconvénients de la récursion terminale

- Le plus souvent, la version non terminale est plus simple à écrire (et à lire!).
- Très souvent, une fonction récursive terminale qui renvoie une liste construit celle-ci « à l'envers », et nécessite donc un [List.rev](#) avant de renvoyer son résultat. Elle peut donc être plus lente (par un facteur constant) que la version non terminale. Cela n'a pas d'importance pour nous (les facteurs constants ne nous intéressent pas), mais peut en avoir dans certaines applications.

7 Allocation dynamique

7.1 Fonction malloc

La fonction `malloc` a le prototype suivant :

```
void* malloc(size_t size);
```

- L'unique paramètre est la taille du bloc à allouer, **en octets**. On reparlera du type `size_t` un peu plus tard, mais ce n'est pas très important pour l'instant : il faut passer un entier positif, c'est tout ce qu'il y a à retenir.
- On obtient en retour un pointeur vers le début du bloc alloué. Pour l'instant, ce pointeur est de type `void*`, ce qui signifie essentiellement qu'on n'a pas encore choisi le type des données qui vont se trouver dans le bloc.
- La taille du bloc est passée en argument à `malloc`, mais c'est ensuite à nous de nous en souvenir : il n'est pas possible de la récupérer si on ne dispose que du pointeur renvoyé par `malloc`.
- La fonction `malloc` est déclarée dans le fichier `stdlib.h`. Il faudra donc bien penser à mettre une ligne `#include <stdlib.h>` dans l'entête du fichier (mais on le fera systématiquement de toute façon).

Exemple 8.5

Un `double` prend 8 octets en mémoire. Pour allouer un bloc permettant de stocker un objet de ce type on peut faire :

```
double* p = malloc(8);
```

Le pointeur renvoyé par `malloc` a ici été automatiquement *transtypé* en `double*` (on parle plus souvent de *cast* pour le transtypage, et le verbe *caster* est, il faut bien le reconnaître, d'usage assez courant chez les informaticiens). On peut rendre ce transtypage explicite :

```
double* p = (double*)malloc(8);
```

En C++, ce transtypage explicite est obligatoire ; en C, il n'est pas très idiomatique mais le programme de MP2I suggère de le faire.

Supposons maintenant qu'on ait le code suivant (qui n'a aucun intérêt, c'est juste pour avoir un exemple minimal) :

```
double* f(double x){
    double* p = (double*)malloc(8);
    *p = x;
    return p;
}
```

L'objet `p` est local à la fonction : il vit sur la pile. En revanche, l'objet *pointé* par `p` vit sur le tas. Cet objet pointé par `p` continuera donc à vivre jusqu'à ce qu'on le libère manuellement : il est tout à fait légal de renvoyer `p`. On peut donc par exemple écrire :

```
int main(void){
    double* p = f(3.14);
    // p est un pointeur vers un bloc sur le tas.
    // Ce bloc fait 8 octets, et contient actuellement
    // le double 3.14.
    printf("%f\n", *p); // affiche 3.140000
    return 0;
}
```

7.2 Pointeurs et tableaux

En C, un tableau et un pointeur c'est *presque* la même chose⁵. Plus précisément, un tableau se comporte le plus souvent comme un pointeur vers son premier élément. Pour commencer, il est sans doute plus simple de voir les choses dans l'autre sens : un pointeur, c'est un tableau (d'une certaine taille, qu'on a intérêt à connaître parce que le pointeur ne la connaît pas, lui). En particulier, le code suivant est parfaitement valide :

```
double x = 3.14;
double* p = &x;
printf("%f\n", p[0]);
```

Autrement dit, les expressions `p[0]` et `*p` sont parfaitement équivalentes pour un pointeur `p`. Évidemment, cela appelle une question : que signifie `p[12]` ?

```
// CODE FAUX !
double x = 3.14;
double* p = &x;
printf("%f\n", p[12]); // Apocalypse, c'est malencontreux.
```

Le problème, c'est que la définition `double x = 3` a réservé un espace mémoire suffisant pour stocker *un* double, pas pour en stocker *treize*. `p[12]` n'est pas du tout une expression problématique *en soi*, mais on fait ici un accès hors-bornes. En revanche, il est parfaitement légal (et c'est même quelque chose que l'on va faire des centaines de fois dans l'année) d'écrire cela :

```
int n = 20;
double* t = (double*)malloc(8 * n); // Bloc permettant de stocker 20 doubles
// ATTENTION, ce bloc n'est pas initialisé.
// Il est donc illégal de lire t[0] (ou t[3]) pour l'instant.
for (int i = 0; i < n; i++) { t[i] = 1.5 * i; }
// Notre « tableau » est désormais initialisé.
printf("t[%d] = %f\n", 5, t[5]); // Affiche "t[5] = 7.5".
```

Exemple 8.6

La fonction suivante renvoie un « tableau » de `n` doubles, initialisé à la valeur `x` fournie.

```
double* array_make(int n, double x){
    double* t = (double*)malloc(8 * n);
    for (int i = 0; i < n; i++){
        t[i] = x;
    }
    return t;
}
```

On peut par exemple l'utiliser ainsi :

```
int main(void){
    int n = 10;
    double* t = array_make(n, 1.0);
    for (int i = 1; i < n; i++) { t[i] = t[i - 1] * 1.5; }
    // On a maintenant t[i] = 1.5i pour 0 ≤ i < n.
}
```

5. Tout est dans le « presque », malheureusement.

7.3 Opérateur `sizeof`

Spécifier manuellement la taille des objets est une très mauvaise idée, pour diverses raisons. On dispose en fait d'un opérateur `sizeof` qui prend en argument un type⁶ et renvoie la taille (en octets) des objets de ce type. Cet opérateur est évalué statiquement (à la compilation), et on a par exemple :

- `sizeof(double)` qui vaut 8;
- `sizeof(float)` qui vaut 4;
- `sizeof(char)` qui vaut 1;
- `sizeof(int)` qui dépend de la machine et du compilateur⁷;
- `sizeof(int*)` qui vaut 4 sur une machine 32 bits et 8 sur une machine 64 bits (c'est la taille d'un pointeur vers un `int`, pas la taille d'un `int`).

Exemple 8.7

On remplacerait le code de l'exemple ci-dessus par :

```
float* array_make(int n, double x){
    double* t = (double*)malloc(n * sizeof(double));
    for (int i = 0; i < n; i++){
        t[i] = x;
    }
    return t;
}
```

Exemple 8.8 – Tableaux bidimensionnels « à plat »

Considérons le code suivant :

```
int rows = 3;
int cols = 4;
int* matrix = (int*)malloc(rows * cols * sizeof(int));
```

On a alloué un bloc mémoire permettant de stocker `rows * cols` entiers. On peut considérer que ce bloc représente une matrice de `rows` lignes et `cols` colonnes, sous cette forme :

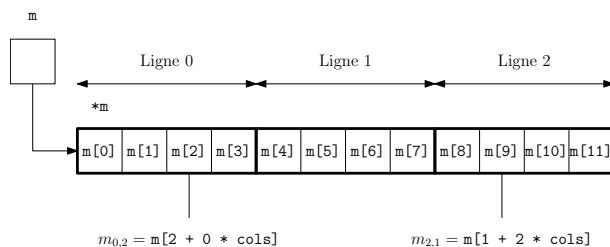


FIGURE 8.9 – Tableau bidimensionnel à plat, format *row major*.

Comme indiqué sur la figure, on peut désormais accéder à la case en *i*-ème ligne, *j*-ème colonne par `m[j + i * cols]`.

Remarques

- Nous reviendrons sur les différentes manières de stocker un tableau bidimensionnel (ou tridimensionnel, ou...).
- L'ordre de stockage décrit (d'abord la première ligne, puis la deuxième...) est dit *row major*. C'est la manière standard de procéder en C, mais certains langages préfèrent l'ordre *column major* (d'abord la première colonne, puis la deuxième...).

6. Ou une expression en fait, mais on l'utilisera avec des types.

7. En pratique c'est toujours 4.

7.4 Fonction free

La fonction `free` permet de libérer un bloc mémoire précédemment alloué par `malloc` pour qu'il puisse être réutilisé. Son prototype est le suivant :

```
void free(void* p)
```

- L'argument de `free` est un *pointeur générique* `void*`. Cela signifie qu'elle accepte un pointeur de n'importe quel type (`int*`, `double*`, `char**`...).
- Le pointeur passé à `free` doit **obligatoirement avoir été créé par un appel à `malloc`.**

Parfois, le bloc sera libéré dans la fonction même où il a été créé :

Exemple 8.9

La fonction suivante prend en entrée un tableau d'entiers entre 0 et 255 (ainsi que la taille de ce tableau), et renvoie l'entier le plus fréquent :

```
uint8_t plus_frequent(uint8_t* t, int len){
    int max_value = 255;
    int* counts = (int*)malloc((max_value + 1) * sizeof(int));
    // On initialise counts à zéro
    for (int i = 0; i <= max_value; i++) { counts[i] = 0; }

    // Après cette boucle, counts[x] sera le nombre d'occurrences de x dans t.
    for (int i = 0; i < len; i++){
        counts[t[i]] = counts[t[i]] + 1;
    }

    // On détermine l'élément le plus fréquent.
    int plus_frequent = 0;
    for (int x = 1; x <= max_value; x++){
        if (counts[x] > counts[plus_frequent]){
            plus_frequent = x;
        }
    }

    // ON N'OUBLIE PAS DE LIBÉRER counts.
    free(counts);

    return (uint8_t)plus_frequent;
}
```

Si on enlève la ligne `free(counts);`, la fonction a une *fuite de mémoire* (ou *memory leak*). En effet, chaque appel à `plus_frequent` allouerait alors un bloc `counts` (qui est de taille 1024 octets en supposant que les `int` sont sur 32 bits) sans le désallouer. Comme la fonction ne renvoie pas de pointeur vers ce bloc, il est ensuite parfaitement impossible de le désallouer : chaque appel consomme donc 1Ko de mémoire de manière définitive (jusqu'à la fin de l'exécution du programme).

Définition 8.3 – Fuite de mémoire

On dit qu'un programme a une *fuite de mémoire* s'il ne désalloue pas (et surtout s'il n'est pas capable de désallouer) la totalité de la mémoire qu'il a alloué sur le tas.

Remarques

- La distinction entre « ne pas désallouer » et « ne pas être capable de désallouer » vient du fait que toute la mémoire est automatiquement libérée quand le programme finit de s'exécuter. Dans des « vrais » programmes, il n'est donc pas rare qu'un bloc qui doit rester vivant jusqu'à la fin du programme ne soit jamais libéré : on peut laisser le système d'exploitation s'en charger. Cependant, il est absolument crucial d'être *capable* de le libérer, c'est-à-dire de disposer d'un pointeur sur lequel on peut appeler `free`. En pratique, le bon moyen de vérifier cela, c'est de :
 - libérer absolument tout;
 - et compiler avec l'option `-fsanitize=address`, ce qui aura pour effet (entre autres) de signaler tous les blocs alloués et non libérés à la fin de l'exécution.
- Nous appliquerons systématiquement la règle suivante : **tout ce qui a été alloué doit être libéré, et on utilise `fsanitize=address` pour vérifier que c'est bien le cas.**
- Les fuites de mémoire sont des bugs extrêmement courants dans les programmes écrits en C (et dans une mesure moindre en C++). Traditionnellement, ils étaient très difficiles à détecter et à isoler : on remarque juste que la consommation mémoire du programme semble être une fonction strictement croissante du temps, et que, par exemple, notre traitement de texte consomme 200Mo quand on ouvre un document mais 1Go au bout d'une heure de travail sur ce document. L'option `-fsanitize=address` a changé la vie de beaucoup de programmeurs.

Exemple 8.10

Considérons le programme suivant, qui alloue deux blocs et ne libère rien :

```
#include <stdlib.h>

int main(void){
    int* tab1 = (int*)malloc(6 * sizeof(int));
    {
        int* tab2 = (int*)malloc(2 * sizeof(int));
        //
        // Le programme a une fuite de mémoire : il n'y a plus
        // aucun moyen de libérer le bloc pointé par tab2 (puisque
        // nous sommes sortis de la portée de tab2).
        //
        // On pourrait en revanche libérer celui pointé par tab1.
        return 0;
    }
}
```

```
$ gcc -o leak -fsanitize=address leak.c
$ ./leak

=====
==844098==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 24 byte(s) in 1 object(s) allocated from:
#0 0x7f4ff414c279 in __interceptor_malloc /build/gcc/src/gcc/...
#1 0x564083d7116a in main (/home/jb/boulot/mp2i/git/...)
#2 0x7f4ff3ef0b24 in __libc_start_main (/usr/lib/libc.so.6+0x27b24)

Direct leak of 8 byte(s) in 1 object(s) allocated from:
#0 0x7f4ff414c279 in __interceptor_malloc /build/gcc/src/gcc/...
#1 0x564083d71178 in main (/home/jb/boulot/mp2i/git/...)
#2 0x7f4ff3ef0b24 in __libc_start_main (/usr/lib/libc.so.6+0x27b24)

SUMMARY: AddressSanitizer: 32 byte(s) leaked in 2 allocation(s).
```

On signale bien deux fuites pour des objets alloués dans la fonction `main` :

- une de 24 octets qui correspond au tableau `tab1` de 6 entiers;
 - une de 8 octets qui correspond au tableau `tab2` de 2 entiers.
-

Il sera bien sûr très courant qu'un bloc alloué dans une fonction soit libéré dans une autre. Si l'on écrit par exemple une fonction qui crée un tableau et qui le renvoie, ce n'est certainement pas elle qui va le libérer ! Toute la difficulté est de ne pas perdre de vue *qui* est responsable de la libération. L'exemple ci-dessous est assez simple ; nous serons parfois confrontés à des situations plus complexes.

Exemple 8.II

Le programme suivant prend un argument en ligne de commande : un entier n . Il génère ensuite un tableau de n entiers aléatoires entre 0 et 99 $t = t_0, \dots, t_{n-1}$, calcule le tableau des *sommes cumulées croissantes* (ou *sommes préfixes*) associées $ps = 0, t_0, t_0 + t_1, \dots, t_0 + \dots + t_{n-1}$. Pour finir, il affiche t et ps .

```
#include <stdio.h>
#include <stdlib.h>

int* rand_array(int len){
    int* t = (int*)malloc(len * sizeof(int));
    for (int i = 0; i < len; i++){
        t[i] = rand() % 100;
    }
    return t;
}

int* prefix_sum(int* t, int len){
    int* ps = (int*)malloc(len * sizeof(int));
    int prev_sum = 0;
    for (int i = 0; i < len; i++){
        ps[i] = prev_sum;
        prev_sum = prev_sum + t[i];
    }
    return ps;
}

void print_array(int* t, int len){
    for (int i = 0; i < len; i++){
        printf("%d ", t[i]);
    }
    printf("\n");
}

int main(int argc, char* argv[]){
    int n = atoi(argv[1]);
    int* t = rand_array(n);
    int* ps = prefix_sum(t, n);
    print_array(t, n);
    free(t);           // t ne servira plus : on peut libérer
    print_array(ps, n);
    free(ps);          // ps ne servira plus : on peut libérer
    return 0;
}
```

Pas d'insulte de la part du *sanitizer*, tout va bien !

```
$ gcc -o prefix -Wall -Wextra -fsanitize=address prefix_sum.c
$ ./prefix 10
83 86 77 15 93 35 86 92 49 21
0 83 169 246 261 354 389 475 567 616
```

7.5 Les erreurs courantes

Memory leak

C'est ce que nous venons de voir.

Use after free

Il est illégal d'accéder à une case mémoire appartenant à un bloc qui a déjà été libéré. L'erreur est parfois évidente :

```
int* t = malloc(...);
...
free(t);
printf("%d\n", t[0]); // ILLÉGAL, on vient d'appeler free(t) !
```

Parfois, ça peut être en tout petit peu plus subtil :

```
// On alloue une « matrice » (à plat).
int* mat = malloc(rows * cols * sizeof(int));
...
// On récupère un pointeur vers la ligne 3 :
int* ligne3 = &mat[3 * cols]

// On peut maintenant accéder à m_(3, 2) ainsi :
int m32 = ligne3[2];
...
// On libère le bloc
free(mat);

// Le pointeur ligne3 n'est plus valide !
printf("%d\n", ligne3[1]); // ERREUR : USE AFTER FREE
```

Double free

Il est illégal de libérer deux fois le même bloc, y compris en utilisant deux alias différents :

```
int* p = malloc(...);
free(p); // OK
free(p); // ERREUR : DOUBLE FREE
```

```
int* p = malloc(...);
int* q = p;
free(q); // OK
free(p); // ERREUR : DOUBLE FREE
```

Cette erreur peut sembler stupide, mais elle n'est en fait pas si facile à éviter : c'est ce qui se passe quand deux « personnes » pensent être responsables de la libération d'un même objet (sans nécessairement se rendre compte qu'il s'agit du même objet).

Appel de `free` sur un pointeur qui ne vient pas de `malloc`

Ce n'est pas parce qu'un pointeur pointe sur le tas qu'on peut le libérer en utilisant `free` ! Il faut absolument que ce pointeur soit exactement celui qui a été renvoyé par `malloc` (ce qui revient à dire qu'il doit nécessairement pointer au **début** du bloc alloué).

```
// On alloue une « matrice » (à plat).
int* mat = malloc(rows * cols * sizeof(int));

// On récupère un pointeur vers (le début de) la ligne 3
int* ligne3 = &mat[3 * cols]

free(ligne3); // ERREUR : ligne3 ne vient pas de malloc
free(&mat[0]); // OK : &mat[0] est égal à mat
```

Exercices

I Récursion terminale

Exercice 8.12

- Écrire une fonction récursive terminale calculant le maximum d'une liste d'entiers.
*Comme on se limite aux **int list**, on pourra prendre **min_int** pour le maximum d'une liste vide.*
- Écrire une fonction récursive terminale calculant (simultanément) le maximum et le minimum d'une liste d'entiers.

Exercice 8.13

- Écrire une fonction se comportant comme le **range** de Python (sans le troisième argument optionnel). On appellera cette fonction (`<|>`), ce qui permettra de l'utiliser comme un opérateur infixé.

```
# 3 <|> 7;;
- : int list = [3; 4; 5; 6]
```

- Que donne `1 <|> 1_000_000`?
- Si la réponse à la question précédente est `stack overflow`, recommencer.

Exercice 8.14

- La fonction de concaténation prédéfinie sur les listes (*i.e.* l'opérateur `@`) est-elle récursive terminale? Comment s'en rendre compte?
- Définir un opérateur de concaténation `@|` qui soit récursif terminal.
- Comment expliquer le choix fait dans la bibliothèque standard de OCaml?

Exercice 8.15 – Fonctionnelles sur les listes : **map** et **iter**

On rappelle la spécification et l'implémentation « standard » des fonctions **List.map** et **List.iter** de la bibliothèque standard :

```
(** List.map : ('a -> 'b) -> 'a list -> 'b list
List.map f [x1; ... ; xn] renvoie [f x1; ... ; f xn] *)
let rec map f u =
  match u with
  | [] -> []
  | x :: xs -> let y = f x in y :: map f xs
```

```
(** List.iter : ('a -> unit) -> 'a list -> unit
List.iter f [x1; ... ; xn] équivaut à f x1; ... ; f xn; () *)
let rec iter f u =
  match u with
  | [] -> ()
  | x :: xs -> f x; iter f xs
```

Pour chacune de ces fonctions :

1. déterminer si l'implémentation proposée est terminale;
2. si ce n'est pas le cas, écrire une version terminale;
3. expliquer ce que l'on gagne et ce que l'on perd en utilisant la version terminale.

Exercice 8.16 – Fonctionnelles sur les listes : `fold_left` et `fold_right`

On rappelle la définition des fonctions `List.fold_left` et `List.fold_right` :

```
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
(* List.fold_left f a [b1; ...; bn]
   renvoie
   f (... (f (f a b1) b2) ...) bn. *)

List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
(* List.fold_right f [a1; ...; an] b
   renvoie
   f a1 (f a2 (... (f an b) ...)). *)
```

1. Écrire de la façon la plus simple possible ces deux fonctions.
2. L'une est naturellement récursive terminale, laquelle ? Est-il possible d'écrire l'autre de manière terminale ? À quel prix ?

Exercice 8.17 – Utilisation de fonctionnelles

Cet exercice n'est pas directement lié au contenu du chapitre. On rappelle qu'il existe des fonctions `Array.map`, `Array.iter`, `Array.fold_left` et `Array.fold_right` équivalentes à celles sur les listes (sauf qu'elles sont toutes itératives ou récursives terminales).

1. Écrire une fonction `produit_liste : int list -> int` renvoyant le produit des éléments d'une liste, en utilisant les fonctionnelles sur les listes.
2. Écrire une fonction `nb_positifs : int list -> int` renvoyant le nombre d'éléments positifs ou nuls d'une liste, toujours en s'aidant des fonctionnelles.
3. La fonction `List.flatten` prend en entrée une liste de listes (d'un même type) et renvoie leur concaténation :

```
# List.flatten [[2; 1; 4]; [3]; []; [7; 10; 2]];
- : int list = [2; 1; 4; 3; 7; 10; 2]
```

Écrire une version de `List.flatten` utilisant `List.fold_left` et une utilisant `List.fold_right`. Déterminer la complexité de chacune des implémentations.

Exercice 8.18

On souhaite écrire une fonction `partitionne` prenant en entrée une `'a list` et une fonction `pred : 'a -> bool` et renvoyant un couple (`oui, non`) de `'a list` tel que :

- les éléments de `oui` sont exactement ceux de la liste d'entrée pour lesquels `pred` s'évalue à `true`, dans le même ordre que dans la liste d'entrée;
- ceux de `non` sont ceux pour lesquels `pred` s'évalue à `false`, toujours dans le même ordre que la liste d'entrée.

Écrire une version récursive terminale et une version récursive non terminale de cette fonction (les deux doivent avoir la même complexité). En général, laquelle préférera-t-on ?

Exercice 8.19 – Tri fusion

L'implémentation la plus simple du tri fusion est la suivante :

```
let rec separe = function
| [] -> ([], [])
| [x] -> ([x], [])
| x :: y :: xs -> let a, b = separe xs in (x :: a, y :: b)

let rec fusionne u v =
match u, v with
| [], _ -> v
| _, [] -> u
| x :: xs, y :: ys ->
  if x <= y then x :: fusionne xs v
  else y :: fusionne u ys

let rec tri_fusion = function
| [] -> []
| [x] -> [x]
| u ->
  let a, b = separe u in
  fusionne (tri_fusion a) (tri_fusion b)
```

1. Préciser si chacune des trois fonctions est récursive terminale ou non.
 2. Si l'on appelle `separe` sur une liste de longueur n , quelle sera la profondeur de l'arbre d'appels ?
 3. Si l'on appelle `fusionne` sur deux listes de longueur n chacune, donner un encadrement de la profondeur de l'arbre d'appels.
 4. Dessiner l'allure de l'arbre d'appels de `tri_fusion` sur une liste de longueur 2^k .
 5. Expliquer pourquoi il serait difficile^a d'écrire une version terminale de `tri_fusion`.
 6. On souhaite pouvoir traiter de « longues » listes (disons dix millions d'éléments). Est-il nécessaire de modifier `separe`? `fusionne`? `tri_fusion`?
 7. Faire les modifications nécessaires, et vérifier qu'on arrive bien à traiter une liste de dix millions d'entiers tirés aléatoirement entre 1 et `10_000_000`.
- a. Ce qui n'est nullement synonyme de *impossible*.

REPRÉSENTATION DES DONNÉES

I Entiers

1.1 Entiers positifs en base b

Théorème 9.1 – Écriture en base b

Soit $b \geq 2$ un entier. Tout entier $n \in \mathbb{N}$ peut s'écrire sous la forme $n = \sum_{i=0}^p a_i b^i$ avec les $a_i \in \{0, \dots, b-1\}$.

Si l'on impose $a_p \neq 0$, cette écriture est unique. On écrira $n = \overline{a_p \dots a_0}^b$.

Remarques

- On parle de *décomposition en base b de n*. Les a_i sont appelés *chiffres de n en base b*.
- Les chiffres qui correspondent aux plus grandes puissances de b sont dits *plus significatifs* (ou de poids fort), ceux qui correspondent aux petites puissances de b sont *moins significatifs* (ou de poids faible).
- La condition rajoutée pour avoir l'unicité est donc que le chiffre le plus significatif doit être non nul. Il y a un problème si $n = 0$, mais l'on considère par convention que 0 a une écriture vide quelle que soit la base.
- On peut aussi représenter des nombres en base 1, mais c'est un peu différent et présente un intérêt limité.
- Quand on écrit $n = 237$ sans préciser la base, il faut bien évidemment comprendre $n = \overline{237}^{10}$.
- En base 2, les valeurs possibles pour un chiffre sont 0 et 1. On parlera indifféremment de *bit* ou de *chiffre en base 2*.
- Si la base est supérieure à dix, on a un problème pour l'écriture : il y a moins de chiffres (usuels) que de chiffres (de la base). Le seul cas que l'on rencontre en pratique est celui de la base 16 (dite *hexadécimal*) : la convention est d'utiliser les lettres de A à F pour représenter les chiffres de 10 à 15.
- Les bases 2, 16 et (dans une moindre mesure) 8 sont couramment utilisées en informatique. Par conséquent, on peut dans la plupart des langages rentrer directement des nombres dans ces bases. En OCaml :

```
# 0b10010;; (* binaire *)
- : int = 18
# 0xff;;      (* hexadécimal *)
- : int = 255
# 0o77;;      (* octal *)
- : int = 63
```

En C, la syntaxe est la même pour l'hexadécimal, mais elle diffère pour l'octal (et il n'y a pas de littéraux en binaire en C99) :

```
int n = 0xff // hexadécimal (n vaut 255)
int p = 77 // décimal
int q = 077 // octal (q vaut 63)
```

Démonstration

Cette démonstration est importante car elle fournit l'algorithme. On démontre l'existence et l'unicité de la représentation canonique par récurrence forte.

- **Initialisation :** pour $n = 0$, la représentation est vide par convention.
- **Héritéité :** on prend $n > 0$ et l'on suppose existence et unicité pour tous les $k < n$.
On divise euclidiquement n par b : $n = bq + r$ avec $0 \leq r < b$.
 - Existence : comme $b > 1$, on a $q < n$ et l'on peut donc écrire $q = \sum_{i=0}^p \alpha_i b^i$ (avec $\alpha_p \neq 0$). On a alors :

$$\begin{aligned} n &= b \sum_{i=0}^p \alpha_i b^i + r \\ &= \sum_{i=0}^p \alpha_i b^{i+1} + r \\ &= \sum_{i=0}^{p+1} \beta_i b^i \quad \text{avec } \beta_i = \begin{cases} r & \text{si } i = 0 \\ \alpha_{i-1} & \text{si } i > 0 \end{cases} \end{aligned}$$

Les β_i sont bien tous dans $[0 \dots b - 1]$, et $\beta_{p+1} \neq 0$.

- Unicité : supposons $n = \sum_{i=0}^m \gamma_i b^i$, avec $\gamma_m \neq 0$. Il faut prouver $m = p + 1$ et $\gamma_i = \beta_i$ pour $0 \leq i \leq p + 1$. On peut écrire :

$$\begin{aligned} n &= \gamma_0 + b \sum_{i=1}^m \gamma_i b^{i-1} \\ &= \gamma_0 + b \underbrace{\sum_{i=0}^{m-1} \gamma_{i+1} b^i}_{q'}, \text{ avec } 0 \leq \gamma_0 < b. \end{aligned}$$

On a donc nécessairement $\gamma_0 = r = \alpha_0$ et $q' = q$ (reste et quotient de la division euclidienne).

On a $\gamma_m \neq 0$, donc l'unicité de la décomposition de q (hypothèse de récurrence) fournit $m - 1 = p$ et $\gamma_{i+1} = \alpha_i$ pour $0 \leq i \leq p$.

■

Exemple 9.1

- Si l'on veut obtenir les chiffres de 137 en base 10, on commence par écrire que $137 = 13 \times 10 + 7$ (division euclidienne). Le chiffre de poids faible a_0 est donc 7, et avant cela on a les chiffres de 13.
De même, $13 = 1 \times 10 + 3$, donc $a_1 = 3$, et on continue en remplaçant 13 par 1.
 $1 = 0 \times 10 + 1$, donc $a_2 = 1$, on remplace 1 par 0 et on a terminé car 0 n'a « pas de chiffre ».
On a donc ^a $137 = \overline{137}^{10}$.
- Si au contraire on dispose de la liste des chiffres en base b et que l'on souhaite obtenir le nombre, le plus efficace est de remarquer ^b que $\sum_{i=0}^p a_i b^i = a_0 + b(a_1 + b(a_2 + \dots (a_{p-1} + b a_p)))$.

Pour 137 en base 10, cela donne $7 + 10(3 + 10 \times 1) = 137$.

En écrivant les calculs dans l'ordre (pour se rapprocher d'un algorithme), on aurait plutôt :

$$0 \xrightarrow[\times 10]{} 0 \xrightarrow[+1]{} 1 \xrightarrow[\times 10]{} 10 \xrightarrow[+3]{} 13 \xrightarrow[\times 10]{} 130 \xrightarrow[+7]{} 137$$

a. c'est rassurant à défaut d'être spectaculaire

b. on appelle cette méthode *l'algorithme de Horner*

Exercice 9.2

1. Donner l'écriture binaire de 59 et de 31. Quel phénomène (général) peut-on remarquer dans le deuxième cas ?
2. Calculer $\overline{1000110}^2$ et $\overline{C7}^{16}$.
3. Comment s'écrit $\overline{102}^3$ en base 5 ?

► Exercice 9.3

p. 157

1. Écrire en OCaml une fonction `eval_msd : int -> int list -> int` telle que nombre b [$a_{n-1}; \dots; a_0$] renvoie l'entier dont l'écriture en base b est $\overline{a_{n-1} \dots a_0}^b$. On pourra supposer que $b > 1$, que les a_i sont dans $[0 \dots b - 1]$, et que le résultat tient dans un `int`.
2. Écrire une fonction `eval_lsd` qui a la même spécification, sauf que la liste des chiffres est donnée avec le chiffre le plus significatif en premier.
3. Écrire une fonction `digits_msd : int -> int -> int list` telle que `digits_msd b n` renvoie la liste des chiffres de n en base b , chiffre le plus significatif en premier. On supposera $n \geq 0$ et $b > 1$.
4. Écrire une fonction `digits_lsd` ayant la même spécification sauf que la liste des chiffres est renvoyée chiffre le moins significatif en premier.

Propriété 9.2

Soit $b \geq 2$ un entier.

- Un nombre $n > 0$ s'écrit (canoniquement) avec $1 + \lfloor \log_b(n) \rfloor$ chiffres en base b .
- Avec N chiffres (au maximum) en base b , on peut écrire b^N nombres différents : ce sont exactement les entiers de 0 à $b^N - 1$.

Exercice 9.4

- Un *octet* est un bloc de huit bits.^a Quelles sont les valeurs représentables sur un octet ?
- Sachant que $2^{10} = 1\,024 \simeq 10^3$, quel est l'ordre de grandeur du plus grand entier représentable sur 32 bits ? sur 64 bits ?
 - a. Et constitue généralement la plus petite quantité de mémoire qu'un ordinateur peut gérer de manière indépendante.

Les opérations arithmétiques élémentaires dans une base autre que dix ne posent pas de problème – à condition d'avoir bien compris comment on les fait en base 10...

Exercice 9.5

1. Effectuer l'addition $\overline{100110}^2 + \overline{1011}^2$ entièrement en base 2 (c'est-à-dire sans jamais convertir un nombre en base 10).
2. Effectuer la multiplication $\overline{100110}^2 \times \overline{1011}^2$ entièrement en base 2.^a
3. Écrire les tables de multiplication en base 3.
4. Effectuer la multiplication $\overline{1022}^3 \times \overline{221}^3$ entièrement en base 3.

a. Si vous ressentez à présent un brin d'amertume envers vos maîtres et maîtresses d'école primaire qui vous ont imposé – au nom de normes sociales d'un autre âge – cette abomination connue sous le nom de « base dix », c'est bien compréhensible.

1.2 Entiers de taille fixée

Quand on parle d'entiers en informatique, il s'agit presque toujours d'entiers *de taille fixée*. Chaque machine a une taille d'entiers maximale sur laquelle elle est capable d'opérer « naturellement » : aujourd'hui, c'est presque toujours 64 bits sur un ordinateur, mais les machines 32 bits sont encore d'usage courant (micro-contrôleurs...). Ici, *opérer naturellement* signifie qu'une opération arithmétique sur des entiers de cette taille est une opération élémentaire pour le processeur.

Ensuite, un langage peut proposer des types entiers différents, en faisant varier :

- la largeur du type (le nombre de bits utilisés pour coder les entiers) ;
- le caractère signé ou non du type (voir plus loin).

Pour chacun de ces types, il faut également définir¹ ce qui se passe en cas de *dépassement de capacité* (ou *overflow*). En effet, si a et b tiennent sur k bits, il n'y a aucune raison que ce soit le cas de $a + b$, et encore moins de $a \cdot b$.

Entiers en OCaml Il n'y a essentiellement qu'un type entier en OCaml : le type **int**, qui est :

- signé ;
- de largeur *largeur naturelle moins un*, donc typiquement 63 bits.

Les valeurs représentables varient de $\text{min_int} = -2^{62}$ à $\text{max_int} = 2^{62} - 1$, et l'*overflow* a un comportement bien défini : les calculs se font modulo 2^{63} puis sont ramenés dans l'intervalle $[\text{min_int} \dots \text{max_int}]$.

```
# min_int - 1;;
- : int = 4611686018427387903
# max_int;;
- : int = 4611686018427387903
# -min_int;;
- : int = -4611686018427387904
# min_int;;
- : int = -4611686018427387904
```

Entiers en C En C, c'est nettement plus compliqué. Pour commencer, il y a une série de types définis dans `stdint.h` :

Type	Largeur	Signé ?	Valeurs
uint8_t	8 bits	Non	0 .. 255
int8_t	8 bits	Oui	-128 .. 127
uint16_t	16 bits	Non	0 .. 65 535
int16_t	16 bits	Oui	-32 768 .. 32 767
uint32_t	32 bits	Non	0 .. 4 294 967 295
int32_t	32 bits	Oui	-2 147 483 648 .. 2 147 483 647
uint64_t	64 bits	Non	0 .. $2^{64} - 1$
int64_t	64 bits	Oui	- $2^{63} \dots 2^{63} - 1$

Ces types ont l'avantage d'être clairement définis, indépendamment de l'architecture de la machine, du système d'exploitation, du compilateur... Cependant, ils sont relativement « récents »² et l'on rencontre encore beaucoup d'« anciens » types. Il est même à peu près impossible de les éviter complètement :

- un certain nombre de fonctions (dont `main`, par exemple) renvoient ou prennent en argument des **int** ou **long int** ou...
- quand on définit un littéral, c'est par défaut un **int** (on peut utiliser un suffixe pour choisir un autre type). Ensuite, les règles (byzantines) de promotion et conversion automatiques s'appliquent, ce qui peut souvent donner des surprises.

1. Ou *indéfinir*, suivant le langage...

2. Tout est relatif : ils ont été introduits dans le standard C99, qui date de 1999...

```
#include <stdio.h>
#include <stdint.h>

int main(void){
    // Ne fait pas ce qu'on croit :
    uint64_t n = 4 * 1000 * 1000 * 1000 * 1000;

    // n est interprété comme un int pour être affiché,
    // ça n'a aucun sens :
    printf("%d\n", n);

    // Pas la « bonne » manière d'afficher n, mais le
    // problème ne vient pas de là.
    printf("%llu\n", n);
    return 0;
}
```

```
$ ./litteraux.out
-1454759936
18446744072254791680
```

Type	Signification usuelle	Littéraux
int	int32_t	3
unsigned int	uint32_t	3u
long int	int32_t ou int64_t	3l
unsigned long int	uint32_t ou uint64_t	3ul
long long int	int64_t	3ll
unsigned long long int	uint64_t	3ull

FIGURE 9.1 – Les types entiers *implementation defined* que nous rencontrerons et leur valeur usuelle.

On peut au moins définir notre grand n :

```
#include <stdio.h>
#include <stdint.h>

int main(void){
    // Ne fait pas ce qu'on croit :
    uint64_t n = 2ull * 1000ull * 1000ull * 1000ull * 1000ull;

    // Pas la « bonne » manière d'afficher n,
    // mais à part le warning tout va bien.
    printf("%llu\n", n);
    return 0;
}
```

```
$ ./litteraux2.out
20000000000000
```

1.3 Entiers signés

Dans toute la partie mathématique, les entiers considérés étaient supposés positifs, mais nous venons de voir que C et OCaml proposent (bien évidemment) des types *signés*, qui permettent de représenter des valeurs négatives.

D'une manière ou d'une autre, il est clair que le signe nous « coûtera » un bit : il faut stocker une information {+,-}. Il est assez naturel d'imaginer la stratégie suivante :

- le bit le plus significatif détermine le signe : un 1 signifie que le nombre est positif, un 0 qu'il est négatif
- la valeur absolue du nombre est stockée de manière standard sur les autres bits.

Remarque

Attention : implicitement, nous considérons ici que l'on travaille avec des entiers d'une certaine largeur fixée. Ainsi, l'expression *bit le plus significatif* désigne le bit de poids maximal dans cette largeur (celui qui correspond à 2^{32} , ou à 2^{64} , par exemple), ce qui explique pourquoi il peut être égal à zéro.

Bien que cette méthode paraisse raisonnable (c'est d'ailleurs celle que l'on utilise pour les flottants), elle a deux défauts :

- le nombre 0 a deux représentations : 00...0 et 10...0. Cela a aussi pour conséquence de ne pouvoir stocker que $2^N - 1$ valeurs différentes : les entiers de $-(2^{N-1} - 1)$ à $2^{N-1} - 1$ (on perd une place puisque zéro en prend deux) ;
- les opérations arithmétiques usuelles ne sont pas très naturelles : essentiellement, pour ajouter deux nombres, on est obligé de regarder leur bit de poids fort pour déterminer leur signe et de distinguer les cas.

En réalité, aucun ordinateur n'utilise cette représentation pour les entiers signés. L'immense majorité utilise la représentation par *complément à deux*³.

Définition 9.3 – Représentation par complément à deux

On se fixe une largeur w . La *valeur en complément à deux* de la suite de bits (b_{w-1}, \dots, b_0) est :

$$-b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

Remarques

- Autrement dit, le bit de poids fort a un poids négatif.
- Une même suite de bits (b_0, \dots, b_{w-1}) , dans une même largeur w , correspondra donc à deux entiers différents : $\sum_{i=0}^{w-1} b_i 2^i$ en *non signé* et $-b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$ en *signé* par complément à deux.

Exemple 9.6

Fixons $w = 4$, et notons $v_4(\text{bits})$ (respectivement $vs_4(\text{bits})$) la valeur non signée (respectivement signée) associée à une suite de bits.

- $vs_4(0000) = v_4(0000) = 0$
- $vs_4(0100) = v_4(0100) = 4$
- $vs_4(1100) = -8 + 4 = -4$ et $v_4(1100) = 8 + 4 = 12$
- $vs_4(1111) = -8 + 4 + 2 + 1 = -1$ et $v_4(1111) = 8 + 4 + 2 + 1 = 15$

Exercice 9.7

On fixe $w = 8$.

1. Quel est le plus grand entier non signé représentable (c'est-à-dire la plus grande valeur $v_8(\text{bits})$ que l'on peut obtenir) ?
2. Quels sont les plus petits et plus grands entiers signés représentables ?

3. Et les quelques autres le complément à un.

3. Quelle suite de bits donne $vs_8(\text{bits}) = 0 ? 127 ? -1 ? -128 ?$

Exercice 9.8 – Complément à deux et complément à un

Si $n \in \llbracket 0, 2^w - 1 \rrbracket$ est un entier codé en binaire sur w bits, son *complément à un* est l'entier obtenu en inversant tous les bits de n (les 1 deviennent des 0 et inversement). Justifier que le complément à deux de n peut être obtenu en ajoutant un au complément à un de n .

Propriété 9.4 – Valeurs maximales et minimales en complément à deux

Pour une largeur de w :

- le plus grand entier signé représentable est $2^{w-1} - 1$;
- le plus petit entier signé représentable est -2^{w-1} .

Propriété 9.5

Pour n'importe quelle largeur w et suite de bits $\text{bits} = (b_{w-1}, \dots, b_0)$, on a :

$$v_w(\text{bits}) \equiv vs_w(\text{bits}) \pmod{2^w}$$

Remarque

Cette propriété est la raison d'être de la représentation par complément à deux, comme la suite va le montrer.

Exercice 9.9 – Addition en complément à deux

1. Poser les additions binaires suivantes :

$$\begin{array}{r} 1011 \\ + 0111 \\ \hline \end{array} \quad \begin{array}{r} 1001 \\ + 0011 \\ \hline \end{array} \quad \begin{array}{r} 1001 \\ + 1011 \\ \hline \end{array} \quad \begin{array}{r} 0101 \\ + 0011 \\ \hline \end{array}$$

2. Interpréter ces additions comme des opérations sur des entiers non signés de 4 bits (on ne gardera donc que les quatre bits les moins significatifs du résultat). Mathématiquement, cela revient à faire quoi ?
3. Reprendre la question précédente en faisant cette fois une interprétation signée (en complément à deux) sur quatre bits.
4. Quel critère simple, portant sur les deux bits de retenue de poids les plus forts, peut-on utiliser pour déterminer s'il y a eu un « vrai » dépassement de capacité ? Par *vrai dépassement de capacité*, on entend une situation dans laquelle le résultat mathématique de l'opération n'est pas représentable avec la largeur fixée.

Remarque

Nous n'allons pas rentrer dans les détails (qui ne nous intéressent pas vraiment), mais l'avantage principal de la représentation en complément à deux, illustré par l'exercice ci-dessus, est qu'on peut utiliser essentiellement le même circuit physique pour les opérations arithmétiques sur les entiers signés et non signés.

Dépassement de capacité Dès qu'on travaille sur des entiers de largeur fixée, le problème du *dépassement de capacité* (ou *overflow*) se pose : si a et b tiennent sur w bits, il n'y a aucune raison que ce soit le cas de $a + b$, et encore moins de ab .

Exemple 9.10 – La minute culturelle

L'année 2012 a été marquée par une addition majeure au patrimoine culturel de l'humanité : la vidéo *Gangnam Style*. À cette époque, le nombre de vues d'une vidéo Youtube était codé en

interne sur un entier 32 bits signé (un `int`, tout simplement). Bien évidemment, ce choix, qui limite le nombre de vues à 2 147 483 647, n'était absolument pas adapté à un chef d'œuvre de cette ampleur. Début 2014, il est devenu évident qu'on allait bientôt avoir un dépassement de capacité, ce qui aurait eu pour conséquence de faire passer le nombre de vues à $-2\ 147\ 483\ 648^a$. Fort heureusement, Youtube a apporté la modification nécessaire à temps : les vues sont maintenant codées sur un entier signé de 64 bits, ce qui laisse un peu de marge (la valeur maximale étant 9 223 372 036 854 775 807).

a. Une autre conséquence possible étant d'ailleurs le début de la troisième guerre mondiale en raison d'un *Undefined Behavior*, voir plus bas.

Exemple 9.11 – Un exemple plus trivole

Le vol inaugural de la fusée Ariane 5 a eu lieu le 4 juin 1996. Comme le montre l'illustration ci-dessous, il s'est terminé, un peu moins de 37 secondes après le décollage, par ce que nous appellerons pudiquement un *RUD*^a.



FIGURE 9.2 – Dommage.

La raison de cet échec tient en une ligne de code, dans laquelle on convertit en entier 16 bits une valeur trop grande pour y rentrer :

```

L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M LSB_BV) *
                                 G_M_INFO_DERIVE(T_ALG.E_BV));
if L_M_BV_32 > 32767 then
  P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
  P_M_DERIVE(T_ALG.E_BV) := .16#8000#;
else
  P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M_
end if;

50 | P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                ((1.0/C_M LSB_BH) *
                                 G_M_INFO_DERIVE(T_ALG.E_BH)))
end LIRE_DERIVE;

```

FIGURE 9.3 – Extrait du code source (en Ada) d'Ariane 5. On peut voir un certain nombre de conversions entier 32 bits vers entier 16 bits avec protection contre les dépassements de capacité, et, soulignée en rouge, une conversion flottant vers entier 16 bits non protégée.

a. *Rapid Unplanned Dissassembly*

Dépassement de capacité en OCaml Comme dit plus haut, les calculs sur les **int** de OCaml sont effectués modulo 2^{63} puis le résultat est ramené dans l'intervalle $[-2^{62} \dots 2^{62} - 1]$.

Dépassement de capacité en C En C les choses sont un peu différentes :

- pour les types non signés, les calculs sont effectués modulo 2^w , où w est la largeur du type;
- pour les types signés, le dépassement de capacité a un comportement non défini.

```
uint8_t n = 0xFF; // n vaut 255
n++;           // n vaut maintenant 0 (garanti)

int n = INT_MAX; // n vaut 231 - 1
int p = n + 1;  // comportement non défini
```

1.4 Opérations bit à bit

En plus des opérations arithmétiques usuelles en mathématiques (addition, multiplication, division...), il est naturel de définir des opérations *bit à bit* (ou *bitwise*) qui opèrent directement sur la représentation binaire : en effet, pour un processeur, ces opérations sont très simples à réaliser (beaucoup plus qu'une division par exemple).

Opération	Notation C	Notation OCaml	Exemple
Conjonction	&	land	$\begin{array}{r} 0001101 \\ \& 0101001 \\ \hline 0001001 \end{array}$
Disjonction		lor	$\begin{array}{r} 0001101 \\ 0101001 \\ \hline 0101101 \end{array}$
Ou exclusif	^	lxor	$\begin{array}{r} 0001101 \\ ^ 0101001 \\ \hline 0100100 \end{array}$
Négation	~	lnot	$\begin{array}{r} 0101001 \\ \sim \\ 1010110 \end{array}$
Décalage à gauche	n << i	n lsl i	$0b1001 \text{ lsl } 3 = 0b1001000$
Décalage à droite	n >> i	n lsr i	$0b1001011 \text{ lsr } 3 = 0b1001$

FIGURE 9.4 – Opérations bit à bit

Remarques

- L'argument i des décalages doit obligatoirement être positif ou nul (et inférieur ou égal à la largeur du type).
- L'opération $n << i$ décale la représentation binaire de n de i chiffres vers la gauche en rajoutant des zéros à droite. S'il n'y a pas de dépassement de capacité, cela revient à multiplier n par 2^i .
- L'opération $n >> i$ décale la représentation binaire de n de i bits vers la droite. Si n est un entier non signé, ou signé et positif, cela revient à calculer $\lfloor n/2^i \rfloor$. Si n est négatif, nous n'utiliserons jamais cette opération.
- Évitez de remplacer tous les $n = n * 2$ par des $n = n << 1$: le compilateur est assez grand pour faire ça tout seul.
- Le résultat d'une négation dépend de la largeur de l'entier ! Il faut être très prudent si l'on utilise cette opération (ce qui ne devrait pas être notre cas).

► Exercice 9.12 – Propriétés du XOR

On se place dans l'ensemble \mathbb{U} des nombres binaires non signés sur N bits, avec $N \geq 1$ fixé. On note \wedge l'opération XOR, qui est une loi de composition interne sur \mathbb{U} .

1. \wedge est-elle commutative ? associative ?
2. \wedge a-t-elle un élément neutre ?
3. Que vaut $x \wedge x$?
4. Que peut-on dire de l'ensemble (\mathbb{U}, \wedge) ?
5. **One-time pad** Supposons qu'Alice et Bob veuillent communiquer de manière chiffrée un message de longueur $n \leq N$ et qu'ils disposent d'une suite de N bits, partagée, qu'ils sont les seuls à connaître. Comment peuvent-ils procéder ?

► Exercice 9.13 – Manipulations diverses

On considère un entier non signé n , codé sur N bits (par exemple un `uint64_t`, avec dans ce cas $N = 64$).

1. Comment déterminer si n est pair ?
2. Comment obtenir un entier p dont la représentation binaire est $\overline{1\dots 1}^2$ (k bits à 1, avec $k \leq N$) ?
3. Comment déterminer si un entier est divisible par 2^k ?
4. Comment récupérer le quotient de la division de n par 2^k ? le reste ?
5. Comment calculer $i2^k$, où $i2^k \leq n < i2^{k+1}$?
6. Comment extraire le k -ème bit de n (le bit de poids 2^k) ?
7. Comment extraire les bits de poids $2^k, \dots, 2^{k+l-1}$ de n (où $1 \leq l \leq N - k$) ?

Exercice 9.14 – Popcount

On considère toujours un entier n non signé sur N bits.

1. Comment déterminer si n est une puissance de 2 en faisant un nombre d'opérations constant (indépendant de N) ?
2. On définit $\text{popcount}(n)$ comme le nombre de bits à 1 dans la représentation de n . Comment calculer $\text{popcount}(n)$ en utilisant un nombre d'opérations proportionnel à $\text{popcount}(n)$ (et non à N) ?
3. Écrire une fonction popcount ayant le prototype suivant :

```
int popcount(uint64_t n);
```

Exercice 9.15 – Bit Scan Reverse

Pour un entier $n = \overline{x_{N-1} \dots x_0}^2$ non nul et non signé codé sur N bits, on définit $\text{bsr}(n)$ le plus grand i tel que $x_i = 1$.

1. Proposer une fonction (simple) de prototype

```
int bsr(uint64_t n);
```

2. On suppose qu'on a précalculé un tableau `int t_bsr[256]` tel que $t_{bsr}[i]$ contienne $\text{bsr}(i)$ (pour $0 \leq i < 256$). Ré-écrire la fonction bsr pour qu'elle effectue au maximum trois décalages.

2 Caractères et chaînes de caractères

2.1 Code ASCII

Le code ASCII associe un caractère à chaque entier entre 0 et 127 (ce qui correspond à 7 bits non signés). Ces caractères peuvent être classés en trois grandes catégories.

Caractères alphanumériques : les chiffres et les lettres minuscules et majuscules. Seules les lettres utilisées en anglais font partie du code ASCII, donc pas de « é », de « ñ », de « ß »...

Autres caractères imprimables : les signes de ponctuation, quelques symboles (+, *, {...}) et l'espace.

Caractères non imprimables la tabulation, les différents caractères correspondant à un retour à la ligne, le caractère nul...

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30			□	!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

Comme les caractères étaient presque systématiquement codés sur 8 bits, les codes 128 à 255 étaient « libres » : ils ont pendant très longtemps été utilisés pour coder les caractères spécifiques aux différentes langues (signes diacritiques, lettres supplémentaires...). Cependant ces extensions posaient deux problèmes :

- elles n'étaient pas standardisées, puisqu'elles différaient d'une langue à l'autre et qu'il y avait même plusieurs extensions concurrentes pour une même langue⁴;
- elles permettaient plus ou moins de gérer les langues européennes (ou au moins les langues basées sur l'alphabet latin), mais étaient totalement inappropriées au chinois, au japonais...

2.2 Unicode

Le standard Unicode a été développé à partir de la fin des années 1980. À l'heure actuelle, il définit des codes pour 144 697 caractères, ce qui permet de gérer l'ensemble des langues, ainsi que de nombreux caractères supplémentaires (comme les Emojis, par exemple). Ce standard définit trois représentations binaires UTF-8, UTF-16 et UTF-32, et il est assez complexe : nous ne rentrerons pas dans les détails. Dans tous les cas, les *codepoints* (l'entier associé à un caractère) ne sont pas modifiés pour les caractères appartenant au code ASCII.

2.3 Caractères et chaînes de caractères en OCaml

Le type `char`

- Le type `char` est codé sur 8 bits.
- Les littéraux de type `char` sont de la forme '`a`' pour les caractères ASCII imprimables :

```
utop # 'Z';;
- : char = 'Z'
utop # ' ';
- : char = ' '
```

4. En pratique, jusqu'à la fin des années 2000, il y avait une chance sur deux que tous les caractères accentués contenus dans un mail soient remplacés par une bouillie infâme avant de parvenir au destinataire.

- Ce n'est pas un type entier (on ne peut pas écrire 'b' + 3), mais on dispose de deux fonctions de conversion :
 - `int_of_char : char -> int` qui associe à chaque caractère un entier entre 0 et 255;
 - `char_of_int : int -> char` qui prend un entier **entre 0 et 255** et renvoie le caractère correspondant.

```

utop # char_of_int 117;;
- : char = 'u'
utop # char_of_int 180;;
- : char = '\180'
utop # char_of_int 500;;
Exception: Invalid_argument "char_of_int".
utop # int_of_char 'x';
- : int = 120
utop # int_of_char ' ';
- : int = 32
utop # char_of_int (int_of_char 'a' + 4);;
- : char = 'e'

```

Le type `string`

Une chaîne de caractères est stockée de manière contiguë en mémoire, et elle connaît sa taille. Elle se comporte essentiellement comme un tableau de caractères, sauf que :

- la syntaxe pour accéder à un élément est légèrement différente : `s.[i]` (des crochets au lieu de parenthèses);
- elle n'est pas mutable (on ne peut pas faire `s.[i] <- ...`).

Les opérations les plus courantes sur les `string` :

Accès à un caractère : `s.[i]`, les éléments sont indexés à partir de zéro, et l'accès se fait en temps O(1).

Longueur : `String.length : string -> int`, en temps O(1).

Concaténation : `s ^ t` renvoie une nouvelle chaîne formée des caractères de `s` suivis de ceux de `t`. Complexité O(|s| + |t|).

2.4 Caractères et chaînes de caractères en C

Le type `char`

- En C, le type `char` est un type entier, mais **nous ne nous en servirons pas ainsi**.
- Il est codé sur 8 bits (en pratique), et il peut être signé ou non signé (ce qui justifie d'éviter de l'utiliser comme un entier).
- Nous l'utiliserons donc comme le type `char` de OCaml :

```

char c = 'z';
int n = (int)c; // conversion char -> int
char c = (char)50; // conversion int -> char

```

Les chaînes en C

Il n'y a pas de type chaîne en C : une chaîne est simplement un tableau de caractères non nuls, terminé par un caractère nul. Nous aurons l'occasion d'y revenir, mais c'est extrêmement piégeux.

```

char* s = "Hello!";
char t[6] = {'H', 'e', 'l', 'l', 'o', '!'}; // Ceci n'est pas une chaîne.
int n = strlen(s);                         // OK
printf("%s : longueur %d\n", s, n);        // OK (Hello! : longueur 6)
int p = strlen(t);                         // PLANTE !!!
printf("%s : longueur %d\n", t, p);         // PLANTE !!!

```

3 Nombres en virgule flottante

3.1 Précision relative, précision absolue

Définition 9.6

On considère un réel x (non nul) et une approximation \tilde{x} de ce réel.

- L'*erreur absolue* est la quantité $|x - \tilde{x}|$ (on parle aussi de *précision absolue* de l'approximation).
- L'*erreur relative* est la quantité $\frac{|x - \tilde{x}|}{x}$ (on parle aussi de *précision relative*).

Remarque

Il est bien évident que dans l'immense majorité des cas, l'erreur relative est bien plus pertinente que l'erreur absolue : des mesures à un mètre près de la distance Terre-Soleil et de la taille d'une personne ne sont pas considérées comme aussi précises l'une que l'autre...

3.2 Flottants en base 10

3.2.a Nombres décimaux

Définition 9.7

On appelle *nombre décimal* un nombre de la forme $n \cdot 10^p$ où $n, p \in \mathbb{Z}$.

Remarques

- Autrement dit, les décimaux sont exactement les réels pouvant s'écrire avec un nombre fini de chiffres après la virgule (*i.e.* dont l'écriture décimale est finie).
- Les décimaux forment un sous-ensemble strict de \mathbb{Q} : ce sont exactement les rationnels dont le dénominateur (sous forme irréductible) n'a pas d'autre diviseur premier que 2 et 5.

3.2.b Nombres à virgule fixe

Pour représenter informatiquement un nombre décimal (positif pour simplifier), on peut imaginer stocker deux entiers : l'un correspondant à la partie entière, l'autre à la partie décimale. En réalité, il suffit de stocker un entier si l'on sait de combien il faut « décaler la virgule » : un nombre à virgule fixe n'est jamais qu'un nombre entier multiplié par une constante de la forme 10^{-k} . Si l'on choisit $k = 3$ et que l'on prend par exemple des entiers sur 32 bits, on peut représenter 327,124 ou 12,1, mais pas 0,000 12 : c'est un peu dommage...

En pratique, cette représentation est surtout utilisée pour les calculs monétaires (qui doivent se faire au centime près).

3.2.c Nombres à virgule flottante

En physique, on utilise généralement l'écriture scientifique : les nombres sont mis sous la forme $m \times 10^e$ avec $1 \leq m < 10$ et $e \in \mathbb{Z}$. Chaque décimal a une unique représentation de ce type : 124,421 s'écrit $1,244\ 21 \times 10^2$ et 0,003 15 s'écrit $3,15 \times 10^{-3}$. Si l'on se fixe une taille maximale pour m et e , et que l'on rajoute un bit de signe pour coder les nombres négatifs, on obtient un format utilisable :

$$\begin{array}{c} \boxed{-1} \quad \boxed{2} \quad , \quad \boxed{3} \quad \boxed{2} \\ \text{signe} \quad \text{mantisse} \quad \text{exposant} \end{array} = -230$$

$$\begin{array}{c} \boxed{+1} \quad \boxed{4} \quad , \quad \boxed{0} \quad \boxed{-2} \\ \text{signe} \quad \text{mantisse} \quad \text{exposant} \end{array} = 0,04$$

Bien sûr, la virgule n'est pas stockée explicitement : on stocke un entier $\overline{m_0 \dots m_t}^{10}$ dans la mantisse, et l'on sait qu'il faut l'interpréter comme s'il y avait une virgule après le premier chiffre, donc comme le nombre $m_0 + \frac{m_1}{10} + \dots + \frac{m_t}{10^t}$.

En notant s le signe et e l'exposant, $\boxed{s} \boxed{\overline{m_0 \dots m_t}} \boxed{e}$ représente alors $s \cdot \left(\sum_{i=0}^t m_i \times 10^{-i} \right) \cdot 10^e$, et la contrainte $m \in [1, 10[$ correspond à $m_0 \neq 0$. Il faut alors traiter différemment le nombre 0 (pour lequel on ne peut pas imposer $m_0 \neq 0$).

La contrainte sur le premier chiffre est nécessaire si l'on veut l'unicité de la représentation :

$$\begin{array}{c} \boxed{+1} \quad \boxed{0} \quad , \quad \boxed{4} \quad \boxed{-1} \\ \text{signe} \qquad \text{mantisse} \qquad \text{exposant} \end{array} = 0,04$$

Cependant, cette contrainte est gênante dans certains cas : lesquels ?

Exercice 9.16

Si l'on prend $e_{\min} = -10$, $e_{\max} = 10$ et $t = 1$, on peut représenter le nombre 0 et ceux de la forme

$$\pm \left(m_0 + \frac{m_1}{10} \right) 10^e, \text{ avec } e \in [-10 \dots 10], m_0 \in [1, 9] \text{ et } m_1 \in [0, 9]$$

Donner alors :

1. le plus grand et le plus petit nombre représentable;
2. le plus petit nombre strictement positif représentable;
3. le plus petit nombre strictement supérieur à 10 représentable.

Exercice 9.17

1. On prend $e_{\min} = -2$, $e_{\max} = 2$ et $t = 3$.
 - a. Combien de nombres peut-on représenter ?
 - b. Quel est le plus grand nombre représentable ?
 - c. Quel est le plus petit nombre strictement positif représentable ?
 - d. Quel est le deuxième plus grand nombre représentable ?
2. On prend $e_{\max} = -e_{\min} = E \in \mathbb{N}^*$ et $t \in \mathbb{N}^*$.
 - a. Quel est le plus grand nombre représentable ?
 - b. Quel est le plus petit nombre strictement positif représentable ?
 - c. Quel est le deuxième plus grand nombre représentable ?

On donne ci-dessous la liste complète des nombres décimaux représentables avec $t = 0$, $e_{\min} = -1$ et $e_{\max} = 1$:

```
-90, -80, -70, -60, -50, -40, -30, -20, -10,
-9, -8, -7, -6, -5, -4, -3, -2, -1,
-0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1,
0,
0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 20, 30, 40, 50, 60, 70, 80, 90
```

On peut faire plusieurs remarques importantes :

- l'écart entre deux nombres successifs croît avec la valeur absolue des nombres : ils ne sont pas du tout équirépartis ;
- on a une symétrie parfaite par rapport à zéro ;
- les opérations arithmétiques élémentaires ne sont pas des lois de composition interne. Autrement dit, la somme (par exemple) de deux nombres représentables n'a aucune raison d'être représentable :
 - on peut avoir un dépassement de capacité (*overflow*), par exemple dans le calcul de $60 + 80$ ou de $-50 - 90$: c'est un problème déjà rencontré pour les entiers ;
 - on peut aussi avoir un problème de perte de précision : $8 + 0,2$ ou $3 + 20$ ne sont pas représentables... .

Pour « régler » le dernier problème, la seule solution est d’arrondir :

Définition 9.8

Notons \mathcal{F} l’ensemble des nombres représentables (pour une taille de mantisse et d’exposant données). Si x est un réel, on note \bar{x} l’élément de \mathcal{F} le plus proche de x , avec la convention qu’on arrondit vers 0 en cas d’équidistance.

On peut alors définir l’addition sur les flottants :

Définition 9.9

On définit une loi de composition interne $\bar{+}$ sur \mathcal{F} par :

$$x \bar{+} y = \overline{x + y}$$

Remarques

- On définit de même les autres opérations élémentaires.
- On a ignoré les dépassements de capacité. Nous en reparlerons, mais c’est une question moins importante que celle de l’arrondi.
- On notera souvent $+$ au lieu de $\bar{+}$ (mais ces opérations sont pourtant bien différentes).

Exercice 9.18

Donner le résultat des opérations (flottantes) suivantes :

1. Avec $t = 0$, $e_{\min} = -1$ et $e_{\max} = 1$:
 - a. $1 + 3$
 - b. $20 + 30$
 - c. $20 + 3$
 - d. $20 + 7$
2. Avec $t = 2$, $e_{\min} = -3$ et $e_{\max} = 3$:
 - a. $1,27 + 12,3$
 - b. $1\,270 + 1,47$

Il est immédiatement apparent d’après la définition que l’addition est commutative. Malheureusement, pour le reste...

Exercice 9.19

1. Avec $t = 2$, $e_{\min} = -3$ et $e_{\max} = 3$, calculer (en flottant) $1270 - (1270 - 1)$ et $(1270 - 1270) + 1$.
2. Avec $t = 0$, $e_{\min} = -1$ et $e_{\max} = 1$, calculer (en flottant) $(1 + 0,1) + (-1)$ et $1 + (0,1 - 1)$.

Exercice 9.20

Avec $t = 1$, $e_{\min} = -1$ et $e_{\max} = 1$, calculer $\frac{a+b}{2}$ et $a + \frac{b-a}{2}$ avec $a = 9,7$ et $b = 9,9$.

3.2.d Précision machine

Définition 9.10 – ε -machine

Notons $x = 1 + \varepsilon$ le plus petit flottant représentable strictement supérieur à 1. On appelle alors ε machine la quantité ε .

En notant $t + 1$ le nombre de chiffres de la mantisse, on a

$$\varepsilon = \left(\frac{1}{10^0} + \frac{1}{10^t} \right) \times 10^0 - 1 = 10^{-t}$$

Remarques

- Ce ε n'est pas du tout le plus petit flottant strictement positif! La valeur du plus petit flottant représentable dépend principalement des valeurs possibles de l'exposant, alors que la valeur de ε ne dépend que de la largeur de la mantisse.
- On peut aussi définir $1 + \varepsilon$ comme le plus grand réel dont la représentation flottante vaut 1. Cette définition n'est pas tout-à-fait équivalente : elle donne un ε deux fois plus petit.
- On peut tout de suite noter (nous y reviendrons) que sur une « vraie » machine, en utilisant des flottants sur 64 bits, le ε machine vaut environ 10^{-16} .

Théorème 9.II

Pour tout réel x , en l'absence d'*overflow* et d'*underflow*, on a

$$\frac{|\bar{x} - x|}{x} \leq \frac{\varepsilon}{2}$$

Remarques

- Le ε machine fournit donc une borne sur l'erreur relative due à un arrondi. Cette borne ne dépend pas de l'ordre de grandeur du réel que l'on approche, à condition que sa valeur absolue soit dans l'intervalle $[10^{\varepsilon_{\min}}, 10^{\varepsilon_{\max}}]$ (sinon on a un dépassement de capacité et le problème est différent).
- Bien entendu, l'erreur absolue dépend, elle, très fortement du réel que l'on approche.
- **Attention :** lorsqu'on enchaîne les calculs, les erreurs d'arrondi s'ajoutent les unes aux autres. On ne peut donc pas du tout garantir que l'erreur relative finale sera inférieure à ε .

3.3 Représentation machine : flottants en base 2

3.3.a Nombres dyadiques

Tout réel a une⁵ écriture décimale : un nombre fini de chiffres avant la virgule, puis un nombre généralement infini de chiffres après la virgule. Par exemple :

$$246.17 = 2 \cdot 10^2 + 4 \cdot 10^1 + 6 \cdot 10^0 + 1 \cdot 10^{-1} + 7 \cdot 10^{-2}$$

$$0.90909090\dots = \sum_{k=0}^{+\infty} \frac{9}{10^{2k+1}} = \frac{10}{11}$$

On peut faire la même chose en base 2 et obtenir une écriture binaire d'un réel :

$$\overline{1011.0011}^2 = 2^3 + 2^1 + 2^0 + 2^{-3} + 2^{-4} = \frac{179}{16} = \overline{11.1875}^{10}$$

Exercice 9.21

1. Donner l'écriture décimale de $\overline{10.101}^2$.
2. Donner l'écriture binaire de $\overline{4.125}^{10}$.

Définition 9.I2

On appelle nombre *dyadique* un réel de la forme $\frac{n}{2^p}$, où $n \in \mathbb{Z}$ et $p \in \mathbb{N}$.

Remarques

- Les nombres dyadiques sont les équivalents en base 2 des nombres décimaux pour la base 10 : ce sont les réels qui s'écrivent avec un nombre fini de chiffres après la virgule en base 2.
- Les nombres dyadiques forment un sous-ensemble strict des nombres décimaux. Ce sont exactement les rationnels dont le dénominateur, sous forme irréductible, est une puissance de deux.
- Un rationnel non dyadique ne peut jamais être représenté de manière exacte par un flottant en base 2, indépendamment de la taille de la mantisse.

5. ou des, en fait

Exemple 9.22

Les nombres $\frac{1}{5}$ et $\frac{1}{10}$ ne sont pas dyadiques. On voit donc apparaître des erreurs d'arrondi dans des calculs à des endroits surprenants si l'on pense en base 10 :

```
# (1. /. 5.) *. 3.;;
- : float = 0.600000000000000089
# 3. /. 5.;;
- : float = 0.6
# 0.2 *. 0.1;;
- : float = 0.0200000000000000039
# 0.2 /. 10.;;
- : float = 0.02
# 3. /. 5. = (1. /. 5.) *. 3.;;
- : bool = false
```

Si l'on remplace mentalement 0.2 par $\overline{0.00110011001100\dots}^2$, il n'y a plus rien de surprenant...

Question : comment peut-on obtenir cette écriture binaire infinie (et, ce faisant, prouver qu'elle est ultimement périodique) ?

3.3.b Particularité de la base 2

On reprend un bit de signe s , une mantisse m et un exposant e . L'exposant est compris entre e_{\min} et e_{\max} , et la mantisse est codée sur $t + 1$ chiffres en binaire. On a donc :

$$\boxed{s \ | \ m_0 \dots m_t \ | \ e} = (-1)^s \left(m_0 + \frac{m_1}{2} + \dots + \frac{m_t}{2^t} \right) 2^e = (-1)^s \overline{m_0.m_1 \dots m_t}^2 2^e$$

Pour les décimaux, on avait imposé que, dans la représentation standard d'un nombre, le premier chiffre de la mantisse soit non nul. On fait de même pour les flottants en binaire, mais cela revient à imposer que $m_0 = 1$: il est donc inutile de stocker ce chiffre. La représentation devient donc :

$$\boxed{s \ | \ 1m_1 \dots m_t \ | \ e} = (-1)^s \left(1 + \frac{m_1}{2} + \dots + \frac{m_t}{2^t} \right) 2^e = (-1)^s \overline{1.m_1 \dots m_t}^2 2^e$$

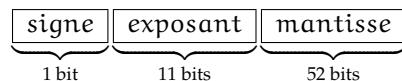
Pour stocker une mantisse avec t chiffres après la virgule, on n'utilisera donc que t bits.

3.3.c Les formats flottants standards

Il existe une norme définissant les nombres flottants, qui porte le nom de IEEE 754 (ce nom est assez connu mais n'est pas à retenir). Cette norme définit des standards pour les nombres flottants codés sur 32 et 64 bits⁶ :

- en OCaml (et en Python) le seul type de base (qui s'appelle **float** dans les deux cas) est celui des flottants en *double précision* sur 64 bits ;
- en C, le type **float** est sur 32 bits et le type **double** sur 64 bits.

Flottants en double précision Le format en double précision utilise un total de 64 bits. La répartition est la suivante⁷ :



11 bits d'exposant permettent de représenter 2048 exposants différents. La représentation choisie utilise un décalage : les 11 bits codent un entier entre 0 et 2047, auquel on soustrait 1023 pour obtenir l'exposant réel.

On devrait donc pouvoir coder les exposants entre -1023 et +1024, mais les deux valeurs extrêmes sont en fait utilisées pour des cas particuliers : les exposants standard vont de -1022 à +1023.

Comme vu plus haut, la mantisse a un chiffre de poids fort implicite qui vaut 1 : les bits $m_1 \dots m_{52}$ codent donc $\overline{1.m_1 \dots m_{52}}^2$, sauf dans un cas particulier vu plus bas.

6. Et quelques autres formats rarement utilisés.

7. On a mis les bits de l'exposant entre celui du signe et ceux de la mantisse car c'est le cas en mémoire ; nous aurons peut-être l'occasion de voir que cette décision n'a pas été prise au hasard.

Finalement, en notant $-1023 \leq e \leq 1024$ l'exposant obtenu après décalage et s le bit de signe :

- si $-1022 \leq e \leq 1023$, le nombre vaut $(-1)^s \times 2^e \times \overline{1.m_1 \dots m_{52}}^2$;
- si $e = -1023$ et $m = 0$ (tous les bits de la mantisse sont nuls), on a $+0$ ou -0 suivant le bit de signe;
- si $e = -1023$ et $m \neq 0$, on a un nombre dénormal, qui vaut $(-1)^s \times 2^{-1022} \times \overline{0.m_1 \dots m_{52}}^2$.
- si $e = 1024$ et $m = 0$, on a $+\infty$ ou $-\infty$ suivant le bit de signe;
- si $e = 1024$ et $m \neq 0$, on a NaN : une valeur particulière qui signifie que le résultat du calcul n'est pas défini.

Tout ceci n'est pas à retenir par cœur! Ce qu'il faut en revanche savoir, c'est que :

- le ε machine vaut $2^{-52} \simeq 2 \cdot 10^{-16}$, et l'erreur relative maximale pour un arrondi vaut donc environ 10^{-16} (on peut retenir 10^{-16} pour la valeur de ε);
- autrement dit, on a environ 16 chiffres décimaux de précision;
- les nombres représentables de manière standard vont de (environ) 10^{-308} à 10^{308} : vous pouvez retenir $10^{\pm 300}$;
- (optionnel) : on peut représenter des nombres un peu plus petit (jusqu'à environ 10^{-324}) de manière dénormale, mais l'on perd alors de la précision (puisque il faut des zéros au début de la mantisse).

```

(* Divisions par zéro : *)

# 1. /. 0.;
- : float = infinity
# 1. /. -0.;
- : float = neg_infinity
# 0. /. 0.;
- : float = nan

(* Calcul de arcsin(2) : *)

# asin 2.;;
- : float = nan

(* Calcul de ln(0) *)

# log 0.;;
- : float = neg_infinity

(* Calcul de arctan de moins l'infini *)
# atan neg_infinity;;
- : float = -1.57079632679489656

(* Underflows *)
# 0.1 ** 1000.;;
- : float = 0.
# -1. /. (10. ** 1000.);;
- : float = -0.

(* Overflows *)
# 10. ** 1000.;;
- : float = infinity

(* NaN et infinity *)
# infinity -. infinity;;
- : float = nan
# infinity /. 2. = infinity;;
- : bool = true
# nan = nan;;
- : bool = false

```

Flottants en simple précision Le principe est exactement le même, mais il y a 8 bits d'exposants et 23 bits de mantisse. Cela donne :

- des valeurs absolues qui vont de environ 10^{-38} à 10^{38} pour les nombres représentables ;
- une précision machine de $2^{-23} \simeq 10^{-7}$.

Littéraux flottants en C Par défaut, un littéral flottant est un double : si l'on veut définir un littéral en simple précision, il faut utiliser le suffixe f. Comme pour les entiers, il y a des règles de promotion automatique (si l'un des opérandes est un **double** et l'autre un **float**, le **float** est promu en **double** avant l'opération), et, comme pour les entiers, je ne vous embêterai pas trop avec cela.

```

#include <stdio.h>

int main(void){
    // Définition d'un float
    float x = 3.14f;

```

Lycée du Parc – MP2I
 // Définition d'un double
double y = 3.14;

// Le calcul a lieu en double, puis

```
$ ./flottants.out
x = 3.140000
y = 3.140000
z = 0.000000000100
zero = 0.000000000000
```

3.4 Quelques pièges du calcul numérique

3.4.a Annulation de chiffres

Exercice 9.23 – Je vous assure que tout ceci a un intérêt.

À votre grand bonheur, vous avez reçu pour Noël une balance d'excellente qualité : elle offre trois chiffres (décimaux) de précision, et ce autant pour des masses de l'ordre du gramme que de l'ordre de la tonne. Vous décidez d'utiliser cette balance pour mesurer la masse m_h de votre hamster h. On suppose pour simplifier que m_h est de l'ordre de 100 grammes (un gros hamster, d'après Wikipedia).

- Si h accepte de monter docilement sur la balance et d'y rester le temps qu'elle fasse sa mesure, avec quelle précision obtiendrez-vous m_h ?
- h (qui est d'une intelligence assez rare pour un rongeur) vous soupçonne, à tort ou à raison, de vouloir utiliser cette pesée pour justifier une mise au régime. Il descend donc immédiatement de la balance à chaque fois que vous l'y posez, et ce avant que la mesure n'ait été faite. Vous décidez alors de le peser indirectement : vous vous pesez une première fois avec h dans la main, puis une deuxième fois sans h, et vous faites la différence. Avec quelle précision obtenez-vous m_h ?

► **Exercice 9.24**

On considère deux flottants a et b connus chacun avec k chiffres significatifs (en base 10). On suppose que $|\frac{a-b}{a}| \simeq 10^{-d}$.

- Que peut-on dire des d chiffres les plus significatifs de a et de b ?
- Combien y a-t-il de chiffres significatifs dans $a - b$?

Exemple 9.25

On a :

$$\frac{1 - \cos(x)}{x^2} = \frac{1}{2} \left(\frac{\sin(\frac{x}{2})}{\frac{x}{2}} \right)^2 \xrightarrow{x \rightarrow 0} \frac{1}{2}$$

et même plus précisément

$$\left| \frac{1 - \cos x}{x^2} - \frac{1}{2} \right| \underset{x \rightarrow 0}{\sim} \frac{x^2}{24} \simeq 0,0417 \cdot x^2.$$

Mais les deux expressions se comportent très différemment lors d'un calcul en virgule flottante :

x	$\left \frac{1 - \cos(x)}{x^2} - \frac{1}{2} \right $	$\left \frac{1}{2} \left(\frac{\sin(\frac{x}{2})}{\frac{x}{2}} \right)^2 - \frac{1}{2} \right $
1e-01	4.17e-04	4.17e-04
1e-02	4.17e-06	4.17e-06
1e-03	4.17e-08	4.17e-08
1e-04	3.04e-09	4.17e-10
1e-05	4.14e-08	4.17e-12
1e-06	4.45e-05	4.17e-14
1e-07	4.00e-04	4.44e-16
1e-08	5.00e-01	0.00e+00

3.4.b Calcul de sommes

Exemple 9.26

Calcul de $\sum_{k=2}^{10^7} \frac{1}{k(k-1)}$:

```
let s1 = ref 0. in
for k = 2 to 10_000_000 do
  let kf = float k in
  let den = kf *. (kf -. 1.) in
  s1 := !s1 +. 1. /. den
done;
Printf.printf "s1 = %.16f\n" !s1
```

s1 = 0.999998999998053

Calcul de $\sum_{k=2}^{10^7} \frac{1}{k(k-1)}$, à l'envers :

```
let s1 = ref 0. in
for k = 10_000_000 downto 2 do
  let kf = float k in
  let den = kf *. (kf -. 1.) in
  s1 := !s1 +. 1. /. den
done;
Printf.printf "s1 = %.16f\n" !s1
```

s1 = 0.9999989999999999

D'où vient la différence (et quelle est ici la meilleure méthode) ?

Solutions

Correction de l'exercice 9.3 page 138

1.

```
let eval_msd b digits =
  let rec aux digits acc =
    match digits with
    | [] -> acc
    | x :: xs -> aux xs (b * acc + x) in
  aux digits 0
```

2.

```
let rec eval_lsd b digits =
  match digits with
  | [] -> 0
  | x :: xs -> x + b * eval_lsd b xs
```

3.

```
let digits_msd b n =
  let rec aux n acc =
    if n = 0 then acc
    else aux (n / b) (n mod b :: acc) in
  aux n []
```

4.

```
let rec digits_lsd b n =
  if n = 0 then []
  else n mod b :: digits_lsd b (n / b)
```

ARBRES

I Introduction

1.1 Premier exemple

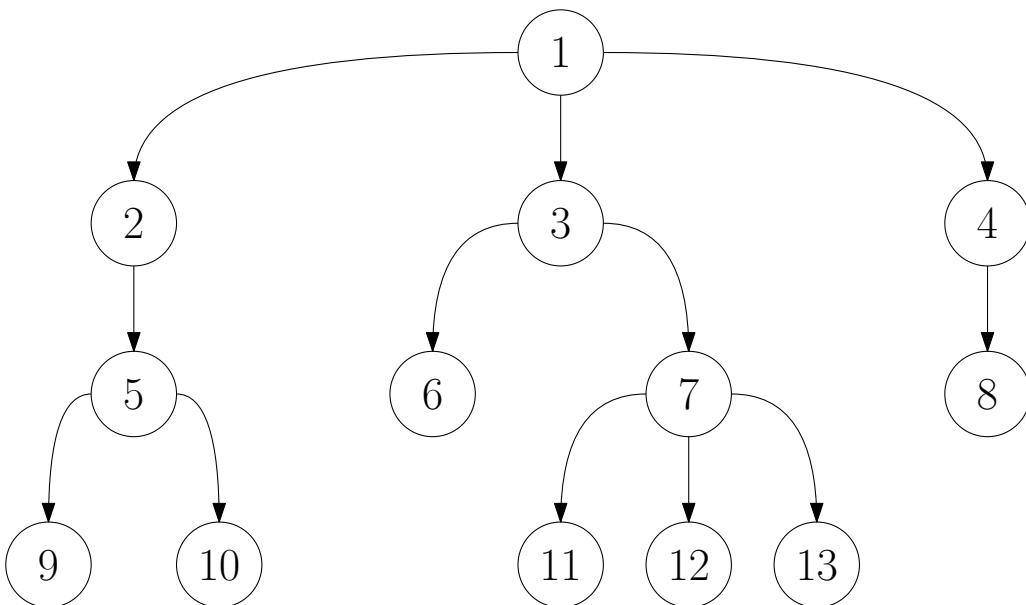


FIGURE 10.1 – L’arbre T

- Un *arbre* est une structure hiérarchique constituée de *nœuds* qui peuvent éventuellement porter des *étiquettes*.
- L’arbre ci-dessus contient 13 nœuds, étiquetés avec les entiers de 1 à 13.
- Chaque nœud a un certain nombre (éventuellement nul) d'*enfants*. Ici :
 - les enfants du nœud 1 sont les nœuds 2, 3 et 4;
 - le seul enfant du nœud 2 est le nœud 5;
 - le nœud 9 n’a pas d’enfant.
- L’*arité* d’un nœud est son nombre d’enfants. Le nœud 1 est d’arité 3, le nœud 2 d’arité 1 et le nœud 9 d’arité 0.
- Un nœud d’arité 0 est appelé *feuille*; un nœud d’arité non nulle est appelé *nœud interne*.
- Si x est un enfant de y, on dit que y est le *père* de x.
- Dans un arbre, il existe un unique nœud qui n’a pas de père : on appelle ce nœud la *racine* de l’arbre.
- Les *ancêtres* du nœud 7 sont les nœuds 3 et 1 : son père, le père de son père, et ainsi de suite jusqu’à la racine de l’arbre.
- Les enfants d’un même père sont dit *frères* : par exemple, 11 et 13 sont les frères de 12.
- La *profondeur* d’un nœud est la longueur du chemin qui relie la racine à ce nœud. La racine est à profondeur 0, ses enfants sont à profondeur 1, ses petits-enfants à profondeur 2...
- La *hauteur* de l’arbre est la profondeur maximum d’un nœud. Ici, la hauteur vaut 3.

1.2 Expression arithmétique

Une expression arithmétique se voit naturellement comme un arbre ; il serait même plus approprié de dire qu'une expression arithmétique *est* un arbre, que l'on a pour habitude de représenter « à plat » en s'aidant de parenthèses.

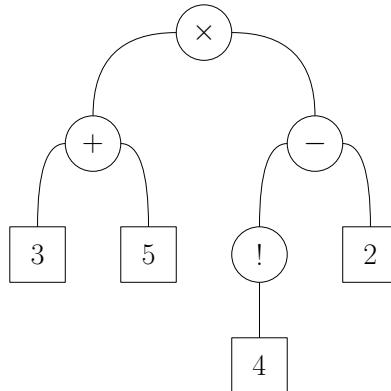


FIGURE 10.2 – L'arbre de l'expression $(3 + 5) \times (4! - 2)$.

1.3 Trie

Dans la variante ci-dessous, les étiquettes sont portées par les arêtes et non par les nœuds. Implicitement, un nœud est étiqueté par le mot que l'on lit en lisant les caractères situés sur le chemin depuis la racine. L'arbre représente un ensemble de mots : l'étiquette d'un nœud appartient à cet ensemble si le nœud correspondant est représenté par un carré.

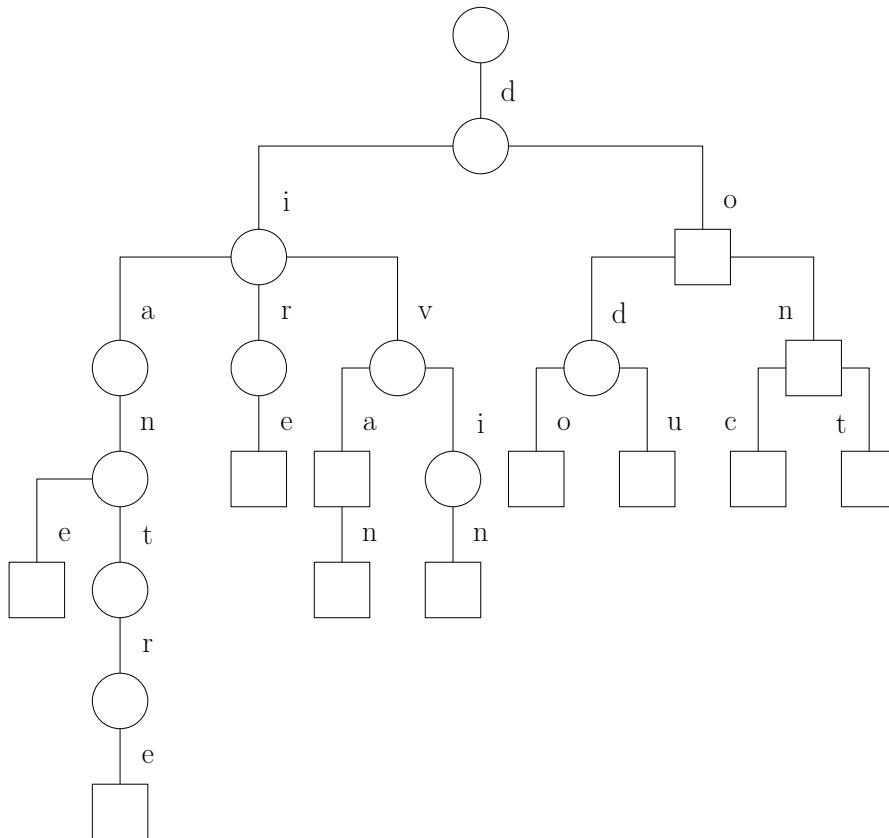


FIGURE 10.3 – Un *trie* pour l'ensemble {diane, diantre, dire, diva, divin, do, dodo, dodu, don, donc, dont}.

1.4 Arbre binaire de recherche

Un *arbre binaire de recherche* est une structure de données permettant de réaliser les types abstraits ENSEMBLE et DICTIONNAIRE (entre autres). La propriété fondamentale d'un tel arbre est que, pour chaque nœud x , tous les nœuds du sous-arbre gauche de x portent une étiquette inférieure à celle de x et tous ceux du sous-arbre droit une étiquette supérieure à celle de x .

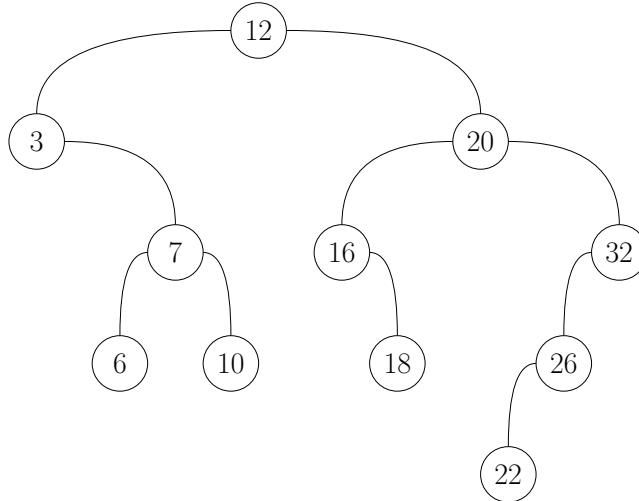


FIGURE 10.4 – Un ABR pour l'ensemble $\{3, 6, 7, 10, 12, 16, 18, 20, 22, 26, 32\}$.

On observe une structure d'arbre un peu différente ici : certains nœuds n'ont qu'un seul fils, mais ce fils est un fils gauche ou un fils droit (dans les deux exemples précédents, un nœud d'arité 1 avait juste un fils qui n'était ni un fils droit ni un fils gauche).

1.5 Arbre syntaxique

De même que l'on peut faire apparaître la structure d'une expression arithmétique en rendant explicite l'arbre qui lui est sous-jacent, on peut faire apparaître la *structure syntaxique* d'une « phrase » à l'aide d'un arbre. La « phrase » en question peut aussi bien être une vraie phrase, issue d'une langue naturelle, qu'un fragment généré par une grammaire formelle (par exemple un extrait de code OCaml ou C).

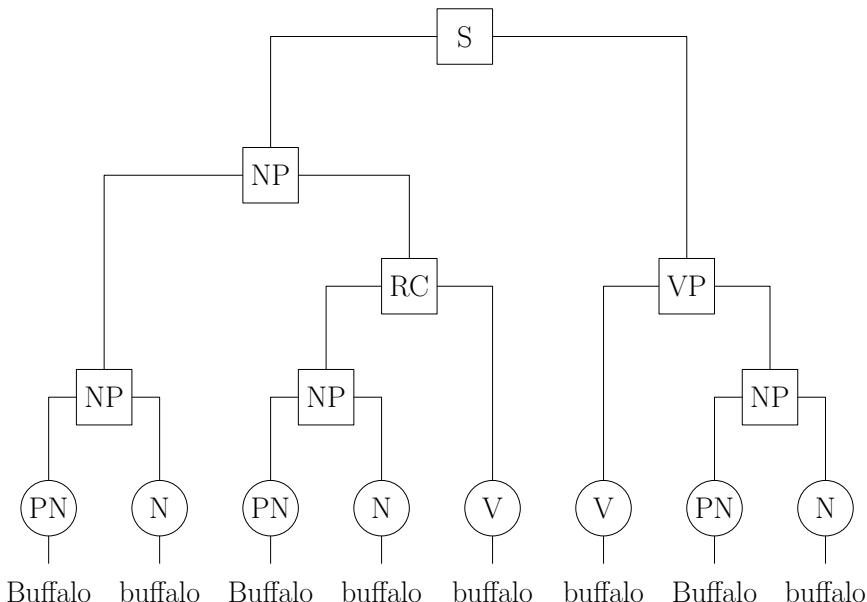


FIGURE 10.5 – Un exemple typique d'arbre syntaxique, qui permet de rendre limpide une phrase qui pouvait sembler quelque peu déroutante de prime abord.

2 Définitions

Remarque importante Il est très courant que les définitions varient légèrement d'une fois sur l'autre, parce que l'on choisit ce qui est le plus adapté à un problème donné. Par conséquent, il faudra être très attentif à bien lire les sujets (de TD, de TP, de devoir, de concours) pour identifier la définition précise avec laquelle on travaille.

2.1 Arbre binaire strict

Définition 10.1 – Arbre binaire strict

On considère un ensemble \mathcal{F} (ensemble des étiquettes des feuilles) et un ensemble \mathcal{N} (ensemble des étiquettes des nœuds internes). On peut alors définir l'ensemble $\mathcal{A}(\mathcal{N}, \mathcal{F})$ des arbres binaires stricts comme suit :

- les feuilles sont des arbres : $\mathcal{F} \subset \mathcal{A}(\mathcal{N}, \mathcal{F})$;
- si g et d sont deux arbres et si $x \in \mathcal{N}$, alors $N(x, g, d) \in \mathcal{A}(\mathcal{N}, \mathcal{F})$.

On a une correspondance immédiate avec une définition de type en OCaml :

```
type ('n, 'f) arbre_bin =
| Feuille of 'f
| Nœud of 'n * ('n, 'f) arbre_bin * ('n, 'f) arbre_bin
```

```
let arbre_exemple =
  Nœud(1,
    Feuille 0.7,
    Nœud(4,
      Nœud(3,
        Feuille 1.4,
        Feuille 3.1),
      Feuille 8.2))
```

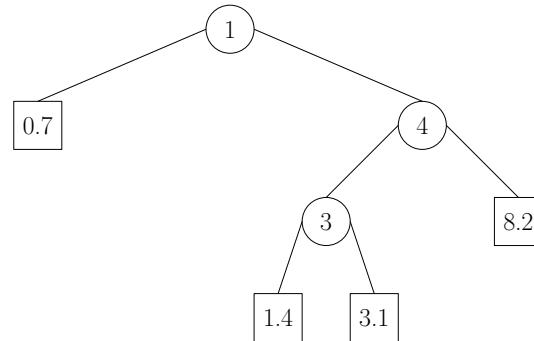


FIGURE 10.6 – Un exemple d'arbre binaire strict de type int, float.

Remarques

- Dans la définition ci-dessus, il faut comprendre implicitement que $\mathcal{A}(\mathcal{N}, \mathcal{F})$ est *le plus petit ensemble* (au sens de l'inclusion) vérifiant les deux conditions.
- Autrement dit, un arbre binaire strict est un arbre dans lequel tous les nœuds internes sont d'arité 2.

Induction structurelle L'ensemble des arbres binaires stricts est un cas particulier d'*ensemble inductif* : tout arbre est soit une feuille, soit de la forme $N(x, g, d)$ où g et d sont des arbres plus « simples ». Nous étudierons un peu la théorie des ensembles inductifs ultérieurement, mais pour l'instant il faut comprendre les implications pratiques.

Preuve par induction structurelle Si P est un prédictat sur les arbres tel que :

- pour tout $f \in \mathcal{F}$, $P(f)$;
 - pour tous $x \in \mathcal{N}$ et $g, d \in \mathcal{A}(\mathcal{F}, \mathcal{N})$, $(P(g) \text{ et } P(d)) \Rightarrow P(N(x, g, d))$,
- alors P est vérifié pour tout arbre de $\mathcal{A}(\mathcal{F}, \mathcal{N})$.

Définition par induction On peut définir une fonction φ sur $\mathcal{A}(\mathcal{F}, \mathcal{N})$ en :

- donnant sa valeur sur les feuilles (cas de base);
- définissant $\varphi(N(x, g, d))$ en fonction de x , $\varphi(g)$ et $\varphi(d)$.

2.2 Vocabulaire

Définition 10.2 – Profondeur, hauteur

La *hauteur* d'un arbre binaire strict est définie récursivement par :

- $h(T) = 0$ si T est une feuille ;
- $h(N(n, g, d)) = 1 + \max(h(g), h(d))$

La *profondeur* d'un nœud x dans un arbre binaire est la longueur du chemin qui relie la racine à ce nœud.

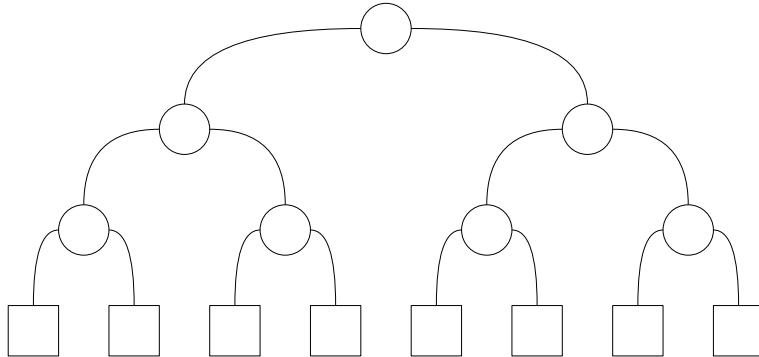
Remarque

La hauteur d'un arbre est donc la longueur maximale d'un chemin reliant la racine à une feuille.

Définition 10.3 – Arbre binaire parfait

Un *arbre binaire parfait* est :

- soit une feuille ;
- soit de la forme $N(x, g, d)$, où g et d sont deux arbres binaires parfaits de même hauteur.



Propriété 10.4

Un arbre binaire strict est parfait si et seulement si toutes ses feuilles sont à la même profondeur.

Démonstration

On procède par induction structurelle.

- Si T est une feuille, c'est évident.
- Si $T = N(x, g, d)$, alors g et d sont deux arbres binaires parfaits de même hauteur h , par définition. Cela signifie que la feuille la moins profonde de g est à profondeur h dans g , et donc, par hypothèse d'induction, que toutes les feuilles de g sont à profondeur h dans g , et donc à profondeur $h + 1$ dans T . Il en va de même pour les feuilles de d , or les feuilles de T sont exactement celles de g et celles de d : elles sont donc toutes à la même profondeur $h + 1$.

2.3 Arbre binaire non strict

Définition 10.5 – Arbre binaire non strict

On considère un ensemble \mathcal{E} (ensemble des étiquettes des nœuds). L'ensemble $\mathcal{A}(\mathcal{E})$ des arbres binaires non stricts étiquetés par \mathcal{E} est défini inductivement par :

- l'arbre vide \perp appartient à $\mathcal{A}(\mathcal{E})$;
- si g et d appartiennent à $\mathcal{A}(\mathcal{E})$ et $x \in \mathcal{E}$, alors l'arbre $N(x, g, d)$ appartient à $\mathcal{A}(\mathcal{E})$.

Remarques

- Quand on parle d'*arbre binaire*, sans préciser, cela peut signifier arbre binaire strict ou non strict suivant le contexte...
- Un arbre binaire non strict est donc un arbre dans lequel :
 - les nœuds sont d'arité zéro, un ou deux;
 - l'unique fils d'un nœud d'arité un est soit un fils gauche, soit un fils droit.

FIGURE 10.7 – Deux arbres binaires non stricts *differents*.

Le type OCaml qui correspond naturellement à cette définition est le suivant :

```
type 'a arbre_binaire_non_strict =
| V
| N of 'a * 'a arbre_binaire_non_strict * 'a arbre_binaire_non_strict
```

Définis ainsi, les arbres binaires non stricts peuvent en fait être vus comme un cas particulier d'arbre binaire strict : il suffit de considérer la définition des arbres binaires stricts et de choisir comme ensemble d'étiquettes pour les feuilles un singleton $\{\perp\}$. En particulier, on peut remarquer que le type '`'a arbre_binaire_non_strict`' défini ci-dessus est isomorphe au type `('n, 'f) arbre_bin` défini plus dans lequel on a fixé `'f = unit` (ou n'importe quel type à un seul élément).

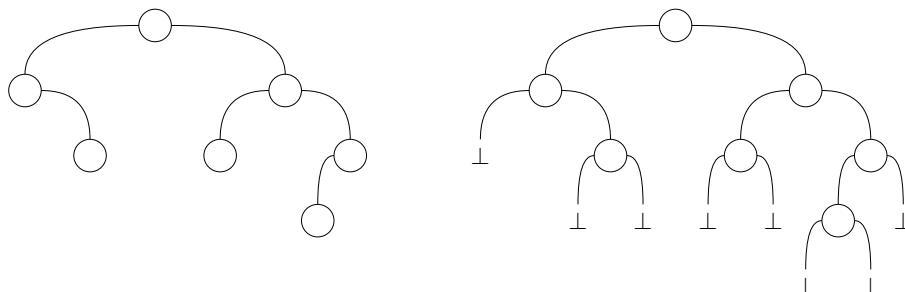


FIGURE 10.8 – Un arbre binaire non strict et sa représentation stricte.

Définition 10.6

Dans un arbre binaire non strict :

- l'arité d'un nœud est son nombre d'enfants *non vides*;
- une feuille est un nœud d'arité zéro, autrement dit un nœud de la forme $N(x, \perp, \perp)$;
- la hauteur d'un arbre réduit à une feuille vaut 0, ce qui pousse à définir la hauteur de l'arbre vide comme valant -1 .

Remarque

Ces définitions ne correspondent pas à ce qu'on obtiendrait en considérant les arbres binaires non stricts comme des cas particuliers d'arbres binaires stricts. À nouveau, il peut y avoir des variations : il faut être très attentif aux définitions du sujet.

3 Parcours d'arbres binaires

Parcourir un arbre, c'est décrire l'ensemble de ses nœuds (internes ou non) dans un certain ordre, typiquement pour effectuer sur chacun d'entre eux une opération (qui peut tout simplement être d'afficher l'étiquette à l'écran).

D'une certaine manière, un parcours commence nécessairement à la racine (du point de vue informatique, il n'y a pas vraiment de différence entre l'arbre et sa racine), mais cela ne signifie pas pour autant que l'on souhaite effectuer l'opération en premier sur la racine... Tous les parcours que nous allons décrire se font de gauche à droite.

3.1 Parcours en profondeur

Dans un parcours en profondeur, on commence par descendre le plus profondément possible (jusqu'à une feuille, donc) avant de remonter pour parcourir le reste de l'arbre.

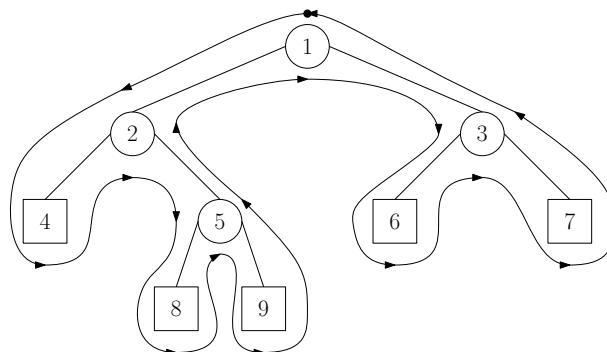


FIGURE 10.9 – Principe du parcours en profondeur.

On peut faire deux remarques sur le parcours en profondeur :

- il est naturellement récursif, puisqu'il s'agit de traiter la racine et de parcourir récursivement les sous-arbres gauche et droit (les feuilles étant les cas de base);
- on passe trois fois par chaque nœud interne :
 - avant d'explorer son fils gauche;
 - entre l'exploration du fils gauche et celle du fils droit;
 - après l'exploration du fils droit.

Cette deuxième remarque induit trois ordres différents sur les noeuds :

ordre préfixe : on classe les nœuds par l'instant de leur première visite¹. Si l'on considère que l'on effectue un traitement sur chaque nœud, cela revient à traiter (récursivement) :

- d'abord la racine;
- puis le sous-arbre gauche;
- puis le sous-arbre droit.

L'ordre induit sur les étiquettes de l'exemple est 1, 2, 4, 5, 8, 9, 3, 6, 7.

ordre infixé : on traite d'abord le sous-arbre gauche, puis le nœud, puis le sous-arbre droit. Cela revient à classer les nœuds dans l'ordre de leur deuxième visite, et l'ordre induit sur les étiquettes de l'exemple est 4, 2, 8, 5, 9, 1, 6, 3, 7

ordre postfixe : on traite d'abord le sous-arbre gauche, puis le sous-arbre droit, puis le nœud. Cela revient à classer les nœuds dans l'ordre de leur troisième visite, et l'ordre induit sur les étiquettes de l'exemple est 4, 8, 9, 5, 2, 6, 7, 3, 1

Remarque

La notion de parcours en profondeur n'est pas limitée aux arbres binaires : en particulier, l'arbre d'appels d'une fonction récursive est parcouru en profondeur lors de l'exécution (et il n'a aucun raison d'être binaire en général). De même, la notion d'ordre préfixe et postfixe garde un sens en présence de nœuds d'arité quelconque (mais pas celle d'ordre infixé).

1. Pour les feuilles, on considère que les trois visites se font simultanément.

3.2 Parcours en largeur

Dans un parcours en largeur, on parcourt les noeuds par profondeur croissante (et de gauche à droite pour une profondeur donnée). Ce parcours est un peu moins utilisé que les variantes du parcours en profondeur.

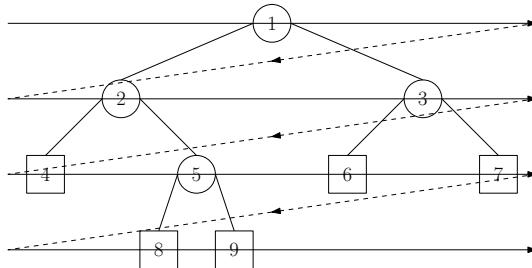


FIGURE 10.10 – Principe du parcours en largeur.

L'ordre induit sur les étiquettes de l'exemple est 1, 2, 3, 4, 5, 6, 7, 8, 9

3.3 Unicité

S'il on dispose de l'énumération des étiquettes dans un ordre défini, on peut se demander s'il est possible de reconstruire l'arbre correspondant (essentiellement, s'il y a un unique arbre donnant cette énumération).

Si, comme c'est le cas dans les exemples ci-dessus, les noeuds internes ne sont pas différenciés des feuilles dans l'énumération, la réponse est non. Cependant, c'est possible dans tous les ordres **sauf l'ordre infixé** si les noeuds internes sont distingués ($[1, 2, 4, 5, 8, 9, 3, 6, 7]$ par exemple). Nous ferons la démonstration dans un cas plus général, mais l'on peut déjà considérer l'exemple suivant :

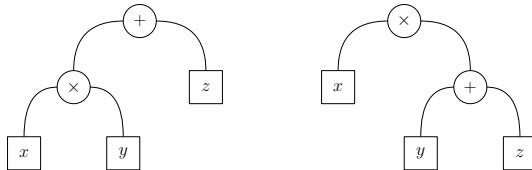


FIGURE 10.11 – Les deux arbres ci-dessus admettent exactement le même parcours en profondeur infixé : $x \times y + z$. En revanche, leurs parcours préfixes (ou suffixes) sont distincts.

3.4 Programmation en OCaml

Travailler avec des arbres est extrêmement facile en OCaml : c'est l'une des tâches pour lesquelles le langage est particulièrement bien adapté.

```

let rec nb_feuilles arbre =
  match arbre with
  | Feuille _ -> 1
  | Noeud (_, g, d) -> nb_feuilles g + nb_feuilles d

let rec affiche_prefixe arbre =
  match arbre with
  | Feuille x -> Printf.printf "F %d " x
  | Noeud (x, ga, dr) ->
    Printf.printf "N %d " x;
    affiche_prefixe ga;
    affiche_prefixe dr

(* affiche_prefixe ex2 affiche N 1 N 2 F 4 N 5 F 8 F 9 N 3 F 6 F 7 *)
  
```

4 Un peu de dénombrement

4.1 Relations entre hauteur, nombre de nœuds, nombre de feuilles

Propriété 10.7

En notant n le nombre de nœuds internes d'un arbre binaire strict et f le nombre de feuilles, on a $f = n + 1$.

Démonstration

On procède par *induction structurelle* sur l'arbre A :

- si A est une feuille, alors $f = 1$, $n = 0$ et la propriété est vérifiée ;
- sinon, A est de la forme (x, A_g, A_d) .
On a alors $f_g = n_g + 1$ et $f_d = n_d + 1$ par hypothèse d'induction, $f = f_g + f_d$ et $n = n_g + n_d + 1$.
On en déduit :

$$f = n_g + 1 + n_d + 1 = (n_g + n_d + 1) + 1 = n + 1.$$

■

Remarque

Cette relation est fausse en général pour un arbre binaire non strict.

Exercice 10.1 – Extension au cas k-aire

On appelle *arbre k-aire strict* un arbre dont tous les nœuds internes ont exactement k fils (où k est une constante entière fixée). Trouver une relation entre le nombre $n(\mathcal{T})$ de nœuds internes d'un arbre k-aire strict \mathcal{T} et son nombre $f(\mathcal{T})$ de feuilles, et la prouver.

Propriété 10.8

Un arbre binaire strict de hauteur h a au plus $2^{h+1} - 1$ nœuds (et $2^h - 1$ nœuds internes). Le cas d'égalité est celui des arbres binaires parfaits.

Remarque

De nombreux algorithmes sur des arbres ayant n nœuds ont une complexité proportionnelle à la hauteur de l'arbre. D'après ce qui précède, si l'arbre est parfait, cette complexité vaut essentiellement $\log_2 n$. Bien sûr, dans le cas d'un arbre très déséquilibré, cette hauteur peut être de l'ordre de n .

Exercice 10.2 – Relation hauteur-taille pour un arbre complet

On appelle (au moins pour cette exercice) *arbre binaire complet de hauteur h* un arbre binaire non strict dont toutes les feuilles sont de profondeur h ou $h - 1$. Donner (et justifier) pour un tel arbre :

1. un encadrement du nombre total de nœuds (internes ou non) en fonction de la hauteur ;
2. un encadrement de la hauteur en fonction du nombre total de nœuds.

Exercice 10.3 – Arbres rouges-noirs

On considère un arbre binaire dont chaque nœud est colorié soit en rouge, soit en noir et qui vérifie les propriétés suivantes :

- un nœud rouge ne peut avoir de fils rouge ;
- tous les chemins partant de la racine et descendant jusqu'à une feuille contiennent le même nombre de nœuds noirs.

Montrer qu'il existe une constante A telle que $h \leq A \log n$ (où h est la hauteur de l'arbre et n son nombre total de nœuds).

4.2 Nombre d'arbres binaires de taille fixée

Définition 10.9

Les *nombres de Catalan* sont définis par :

- $c_0 = 1$
- $c_{n+1} = \sum_{k=0}^n c_k c_{n-k}$ pour $n \in \mathbb{N}$.

Remarques

- Ces nombres apparaissent naturellement dans de nombreux problèmes de combinatoire.
- La manière « naturelle » de calculer récursivement les nombres de Catalan est très inefficace. Comment faire mieux ?

Propriété 10.10

Il y a c_n arbres binaires stricts non étiquetés possédant n nœuds internes.

Démonstration

On procède par récurrence sur le nombre n de nœuds de l'arbre.

- Il y a exactement un arbre binaire sans nœud interne : celui réduit à une feuille.
- Un arbre binaire à $n + 1$ nœuds internes est constitué de :
 - une racine (non étiquetée) ;
 - un sous-arbre gauche ayant k nœuds internes (avec $0 \leq k \leq n$), choisi librement : par hypothèse de récurrence, on a c_k choix ;
 - un sous-arbre droit ayant $n - k$ nœuds internes, choisi librement et indépendamment du sous-arbre gauche : c_{n-k} choix.

À k fixé, on a donc $c_k c_{n-k}$ arbres à $n + 1$ nœuds internes, donc au total $\sum_{k=0}^n c_k c_{n-k} = c_{n+1}$ arbres. ■

Théorème 10.11

Pour $n \in \mathbb{N}$, on a :

$$(n+2)c_{n+1} = 2(2n+1)c_n \quad (1)$$

$$c_n = \frac{1}{n+1} \binom{2n}{n} \quad (2)$$

Démonstration

- Démontrons le deuxième point par récurrence, en admettant temporairement le premier :

Initialisation Pour $n = 0$, on a bien $\frac{1}{1} \binom{0}{0} = 1 = c_0$.

Hérédité Supposons la propriété vraie au rang n .

$$(n+2)c_{n+1} = 2(2n+1) \cdot c_n \quad \text{d'après (1)}$$

$$= \frac{2(2n+1)}{n+1} \cdot \binom{2n}{n} \quad \text{d'après H}_n$$

$$= \frac{2(2n+1)}{n+1} \cdot \frac{(n+1)^2}{(2n+1)(2n+2)} \cdot \binom{2n+2}{n+1}$$

$$= \frac{2(n+1)}{2n+2} \cdot \binom{2n+2}{n+1}$$

$$= \binom{2n+2}{n+1}$$

On a donc H_{n+1} , ce qui achève la récurrence.

- Pour le premier point, on remarque que :

- le membre de gauche correspond au nombre de couples (A, f) où A est un arbre à $n + 1$ nœuds internes et f est une feuille de A ;
- le membre de droite correspond au nombre de triplets (B, x, ε) où B est un arbre à n nœuds internes, x un nœud (interne ou non) de B et ε est choisi dans l'ensemble $\{\leftarrow, \rightarrow\}$.

On considère alors l'application *efface* qui à un couple (A, f) associe le triplet (B, x, ε) où :

- B est l'arbre obtenu à partir de A en supprimant la feuille f ainsi que son père (le frère de f se trouve alors remonté d'un cran);
- x est le nœud remonté (l'ancien frère de f);
- ε vaut \leftarrow si f était un fils gauche, \rightarrow si f était un fils droit.

Cette application est une bijection, de réciproque *insere* qui à un triplet (B, x, ε) associe le couple (A, f) où :

- A est obtenu à partir de B en créant un nouveau nœud à l'emplacement de x , ayant pour fils gauche (si $\varepsilon = \leftarrow$) une feuille et comme fils droit x (avec le sous-arbre associé). Les fils sont inversés si $\varepsilon = \rightarrow$;
- f est la feuille nouvellement créée.

On en déduit l'égalité des cardinaux, et donc le point (1).

■
p. 174

Exercice 10.4 – Équivalent de c_n

On rappelle ^a la formule de Stirling : $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$. En déduire un équivalent simple de c_n .

a. Rappel qui n'est sans doute pas inutile, puisque j'ai eu l'occasion de constater à de multiples reprises que les élèves ont une fâcheuse tendance à ne pas se souvenir de ce qu'ils verront plus tard en cours de mathématiques...

Exemple 10.5

Pour rendre effectives les définitions ci-dessus, il faut choisir une numérotation des nœuds et des feuilles : on choisit l'ordre préfixe. On obtient alors :

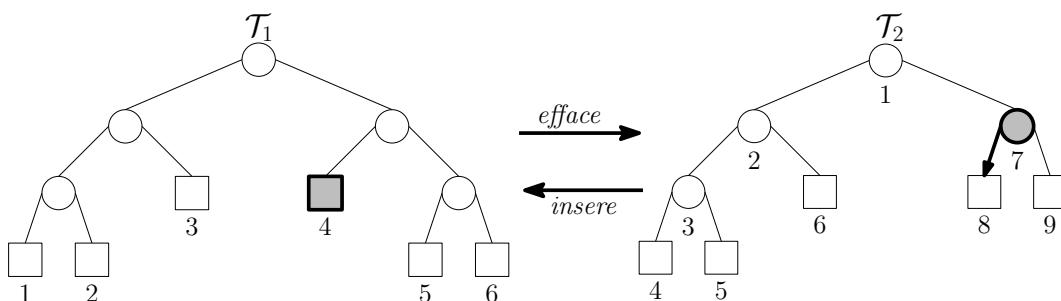


FIGURE 10.12 – $\text{efface}(\mathcal{T}_1, 4) = (\mathcal{T}_2, 7, \leftarrow)$ et $\text{insere}(\mathcal{T}_2, 7, \leftarrow) = (\mathcal{T}_1, 4)$.

Exercice 10.6

En reprenant les arbres ci-dessus, calculer :

1. $\text{efface}(\mathcal{T}_1, 2)$;
2. $\text{insere}(\mathcal{T}_2, 6, \rightarrow)$;
3. $\text{efface}(\mathcal{T}_2, 5)$;
4. $\text{insere}(\mathcal{T}_1, 1, \leftarrow)$.

Attention, les numéros correspondent à des feuilles quand on applique *efface* et à des nœuds (internes ou non) quand on applique *insere*.

5 Arbres non binaires

5.1 Lecture unique

On s'intéresse à des arbres dont les nœuds peuvent avoir une arité quelconque : chaque nœud porte une étiquette et possède une liste d'enfants. Une feuille est simplement un nœud dont la liste d'enfants est vide. Un type permettant de représenter de tels arbres pourrait être :

```
type 'a arbre = N of 'a * 'a arbre list
```

On définit le parcours préfixe d'un tel arbre par :

- $\text{prefix}(N(x, [])) = (x, 0)$
- $\text{prefix}(N(x, [a_1, \dots, a_k])) = (x, k), \text{prefix}(a_1), \dots, \text{prefix}(a_k)$

Remarque

Il faut comprendre les virgules ci-dessus comme des concaténations : (x, k) , suivi de la représentation préfixe de a_1 , suivie de la représentation préfixe de $a_2 \dots$

Définition 10.12 – Suite équilibrée

- On définit le *poids* d'une suite finie $u = (x_1, k_1), \dots, (x_n, k_n)$ par :

$$p((x_1, k_1), \dots, (x_n, k_n)) = \sum_{i=1}^n (k_i - 1)$$

- La suite vide a donc un poids nul.
- Un *préfixe* de u est une suite de la forme $(x_1, k_1), \dots, (x_j, k_j)$ avec $0 \leq j \leq n$.
- Le préfixe est dit *strict* si $j < n$.
- Un *facteur* de u est une suite $(x_i, k_i), (x_{i+1}, k_{i+1}), \dots, (x_{j-1}, k_{j-1})$ avec $1 \leq i \leq j \leq n$.
- La suite u est dite *équilibrée* si :
 - pour tout préfixe strict v de u , $p(v) \geq 0$;
 - $p(u) = -1$.

Propriété 10.13

Pour tout arbre t , $\text{prefix}(t)$ est équilibré.

Propriété 10.14

En tout point d'une suite équilibrée commence un unique facteur équilibré.

Propriété 10.15

Une suite u est équilibrée si et seulement si il existe un arbre t tel que $u = \text{prefix}(t)$. Dans ce cas, t est unique.

Remarques

- Autrement dit, un arbre est uniquement défini par son parcours préfixe, à condition que ce parcours contienne l'arité de chacun des nœuds.
- Dans le cas d'un arbre binaire strict, zéro et deux sont les seules valeurs possibles de l'arité : il suffit donc de noter pour chaque nœud s'il s'agit d'une feuille ou d'un nœud interne.
- Le résultat correspondant pour le parcours postfixe se prouve exactement de la même manière.

5.2 Transformation en arbre binaire

Il existe une manière standard de transformer un arbre d'arité quelconque en arbre binaire, de manière réversible (une bijection de l'ensemble des arbres dans l'ensemble des arbres binaires, essentiellement) : c'est la représentation LCRS (*Left child, right sibling*, c'est-à-dire fils gauche, frère droit).

Une manière de comprendre cette représentation est d'imaginer qu'un nœud de l'arbre initial est constitué d'une étiquette et d'une liste chaînée d'enfants (éventuellement vide). Dans ce cas :

- chaque nœud contient un lien vers la liste chaînée de ses enfants, c'est-à-dire vers une cellule contenant son fils le plus à gauche ;
- et chaque nœud, sauf la racine, appartient à l'une de ces listes chaînées. Dans cette liste, la cellule contenant le nœud contient également un lien vers le « frère droit » du nœud (qui est dans la cellule suivante de la liste).

Exemple 10.7

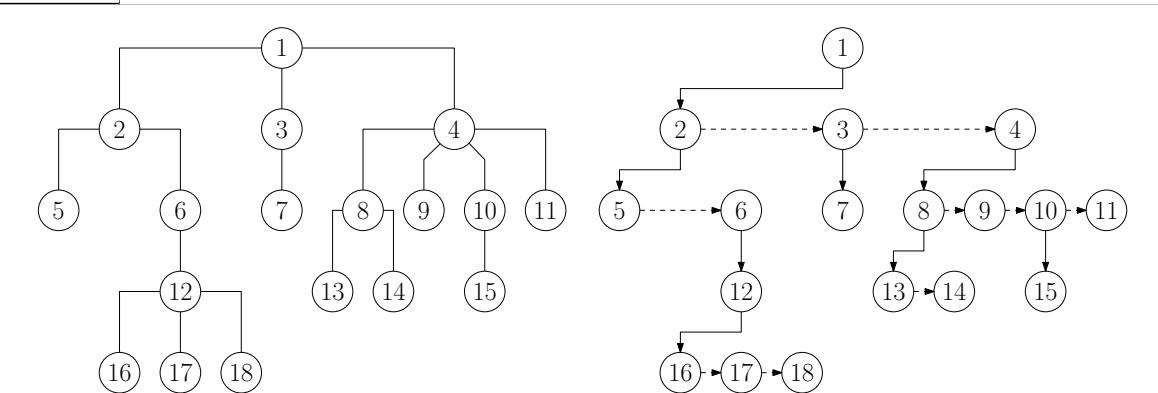


FIGURE 10.13 – Un arbre d'arité quelconque.

FIGURE 10.14 – Représentation avec liste chaînée d'enfants.

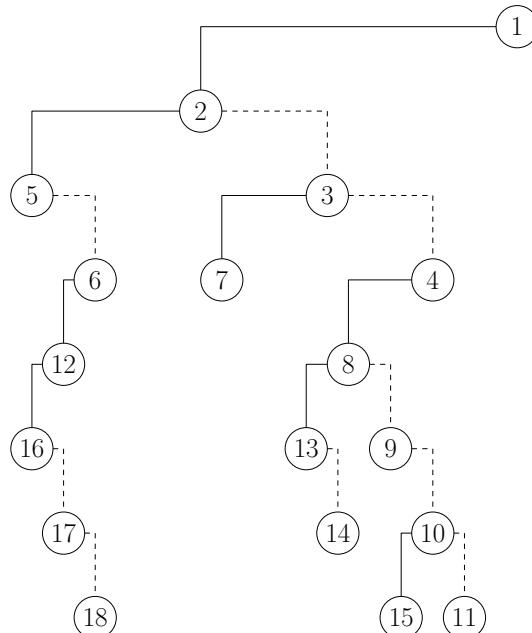
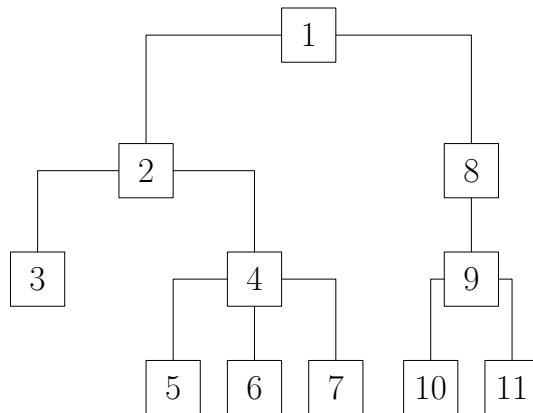


FIGURE 10.15 – Transformation en arbre binaire.

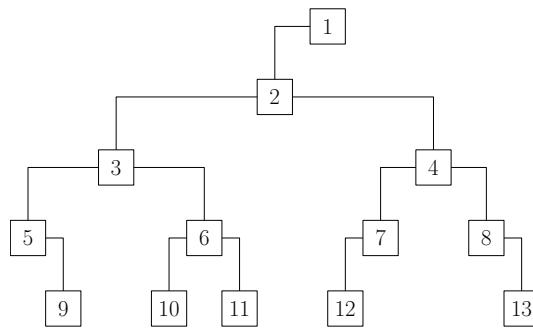
Exercice 10.8

p. 174

1. Appliquer la transformation décrite ci-dessus à l’arbre suivant :



2. Appliquer la transformation inverse à l’arbre suivant :



Nous aurons l’occasion d’étudier les avantages et les inconvénients de la représentation sous forme d’arbre binaire, qui dépendent des applications.

Exercice 10.9

p. 175

On considère les deux types suivants :

- le type '`a arbre`' représente un arbre d’arité quelconque (une feuille correspond à un nœud de la forme `Nn (x, [])`) :

```

type 'a arbre =
| Nn of 'a * 'a arbre list
  
```

- le type '`a binaire`' correspond à un arbre binaire (non strict) :

```

type 'a binaire =
| V
| N of 'a * 'a binaire * 'a binaire
  
```

1. Écrire une fonction `vers_binaire` réalisant la transformation de la partie 5.2 dans le sens « arité quelconque vers binaire ».
2. Écrire une fonction `vers_naire` réalisant la transformation inverse.

Chacune de ces fonctions peut s’écrire en une demi-douzaine de lignes, mais c’est loin d’être évident. Ne pas hésiter à demander des indications si nécessaire.

Exercices

Exercice 10.10 – Quelques petites fonctions

p. 175

On considère des arbres binaires de type :

```
type 'a strict = F of 'a | N of 'a * 'a strict * 'a strict
```

1. Écrire une fonction profondeur_min : 'a strict -> int qui renvoie la profondeur minimale d'une feuille de l'arbre.
2. Écrire une fonction diff_max : 'a strict -> int qui renvoie la différence maximale entre la profondeur de deux feuilles de l'arbre.
3. Écrire une fonction feuille_basse : 'a strict -> 'a qui renvoie l'étiquette de la feuille de l'arbre située le plus à gauche, parmi celles de profondeur maximale.
On essaiera d'écrire une fonction efficace.
4. Écrire une fonction arbre_hauteurs : 'a strict -> int strict qui prend en entrée un arbre t et renvoie un arbre ayant exactement la même forme que t, mais dans lequel l'étiquette de chaque nœud a été remplacée par la hauteur du sous-arbre correspondant.
On essaiera d'écrire une fonction efficace.

Exercice 10.11 – Arbres d'arité quelconque

p. 176

On considère des arbres d'arité quelconque :

```
type 'a arbre = N of 'a * 'a arbre list
```

1. Écrire une fonction est_binaire_strict : 'a arbre -> bool qui détermine si son argument est un arbre binaire strict (dont tous les nœuds sont d'arité 0 ou 2).
2. Écrire une fonction somme : int arbre -> int qui renvoie la somme des étiquettes de l'arbre passé en argument.

Exercice 10.12

p. 177

On s'intéresse à des arbres ayant pour type :

```
type 'a binaire =
| V
| N of 'a * 'a binaire * 'a binaire
```

- On définit la *taille* d'un arbre comme son nombre total de nœuds (non vides).
 - Un nœud N (x, g, d) est dit *lourd* si la taille de d est strictement supérieure à celle de g, *léger* sinon.
 - Un nœud vide E est considéré comme léger.
 - Le *poids* d'un arbre est défini comme son nombre de nœuds lourds.
1. Écrire une fonction taille : 'a binaire -> int.
 2. En utilisant cette fonction, écrire une fonction poids : 'a binaire -> int (la plus simple possible).
 3. Donner un exemple d'arbre de taille n pour lequel poids s'exécute en temps quadratique.
 4. Proposer une version plus efficace de poids.

Exercice 10.13 – Constructions d'arbres

p. 177

On utilise à nouveau le type binaire de l'exercice précédent.

- *peigne gauche* un arbre dont tous les nœuds internes sont de la forme $N(x, \text{ gauche}, V)$;
- *peigne droit* un arbre dont tous les nœuds internes sont de la forme $N(x, V, \text{ droit})$;
- *arbre complet* un arbre dont toutes les feuilles sont à profondeur n ou $n - 1$ pour un certain n (on ne demande pas que le dernier niveau soit rempli de gauche à droite) ;
- *arbre parfait* un arbre dont toutes les feuilles sont à profondeur n pour un certain n .

1. Écrire une fonction `peigne_gauche : int -> int binaire` prenant en entrée un entier n et renvoyant un peigne gauche à n nœuds internes dont les étiquettes, lues dans l'ordre préfixe, forment la liste $[n, \dots, 1]$.
2. Écrire une fonction `peigne_droit : int -> int binaire` avec la même spécification (mais renvoyant un peigne droit).
3. Écrire une fonction `complet : int -> int binaire` ayant encore la même spécification (mais renvoyant un arbre complet).
4. Écrire une fonction `parfait : int -> int binaire` renvoyant un arbre parfait à $2^n - 1$ nœuds internes dont les nœuds situés à profondeur k portent l'étiquette $n - k$.
5. Quelle est la complexité temporelle de la fonction `parfait`? Combien d'espace l'arbre renvoyé occupe-t-il en mémoire?

Exercice 10.14 – Parcours en largeur et reconstruction

p. 178

On garde toujours le type binaire et l'on s'intéresse cette fois au parcours en largeur.

1. Écrire une fonction `liste_lergeur : 'a binaire -> 'a list` renvoyant la liste des étiquettes d'un arbre, dans l'ordre du parcours en largeur.
Indication : on pourra utiliser le module **Queue** et définir une fonction auxiliaire de type `'a binaire Queue.t -> 'a list` réalisant le parcours en largeur d'une forêt (d'abord toutes les racines, puis tous les nœuds situés à profondeur 1 dans les différents arbres, puis...).
2. Écrire une fonction `numerote_largeur : 'a binaire -> ('a * int) arbre` renvoyant un arbre ayant exactement la même forme que celui passé en argument mais dans lequel on a adjoint à chaque étiquette le numéro du noeud correspondant dans le parcours en largeur.

Remarque

Cette question est assez difficile (même si le code attendu est très court).

Solutions

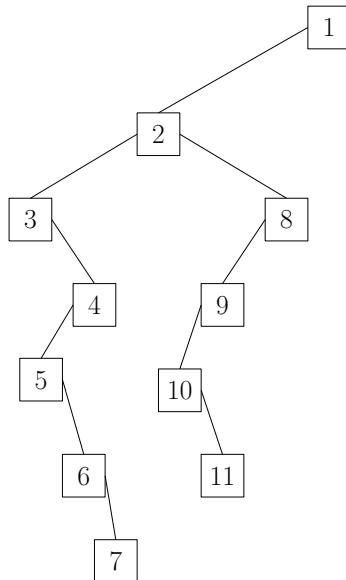
Correction de l'exercice 10.4 page 168

On calcule :

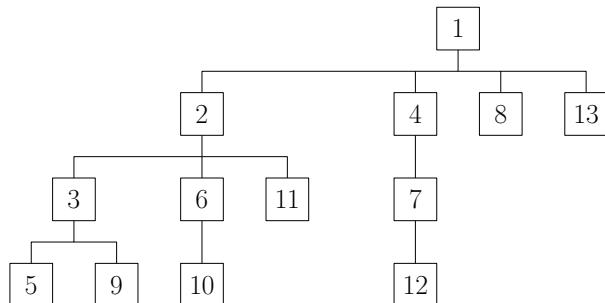
$$\begin{aligned}c_n &= \frac{1}{n+1} \binom{2n}{n} \\&\sim \frac{1}{n} \cdot \frac{(2n)!}{(n!)^2} \\&\sim \frac{1}{n} \cdot \frac{\sqrt{4\pi n}}{2\pi n} \cdot \left(\frac{2n}{e}\right)^{2n} \cdot \left(\frac{e}{n}\right)^{2n} \\&\sim \frac{1}{n^{3/2} \sqrt{\pi}} \cdot 2^{2n} \\&\sim \frac{4^n}{n^{3/2} \sqrt{\pi}}\end{aligned}$$

Correction de l'exercice 10.8 page 171

1.



2.



Correction de l'exercice 10.9 page 171

1.

```
let vers_binaire arbre =
  let rec aux courant freres =
    match courant, freres with
    | Nn (x, []), [] -> N (x, V, V)
    | Nn (x, e :: es), [] -> N (x, aux e es, V)
    | Nn (x, []), f :: fs -> N (x, V, aux f fs)
    | Nn (x, e :: es), f :: fs -> N (x, aux e es, aux f fs) in
      aux arbre []
```

2.

```
let vers_naire arbre =
  let rec vers_liste = function
    | V -> []
    | N (x, g, d) -> Nn (x, vers_liste g) :: vers_liste d in
  match vers_liste arbre with
  | [arbre_naire] -> arbre_naire
  | _ -> failwith "argument invalide"
```

Correction de l'exercice 10.10 page 172

1. Une seule solution raisonnable :

```
let rec profondeur_min arbre =
  match arbre with
  | F _ -> 0
  | N (_, g, d) -> 1 + min (profondeur_min g) (profondeur_min d)
```

2. Diverses manières de procéder, mais le plus simple est clairement de faire deux parcours de l'arbre : un pour trouver la profondeur minimale d'une feuille, et un pour trouver la profondeur maximale d'une feuille (c'est-à-dire la hauteur de l'arbre) :

```
let rec hauteur = function
  | F _ -> 0
  | N (_, g, d) -> 1 + max (hauteur g) (hauteur d)

let diff_max arbre =
  hauteur arbre - profondeur_min arbre
```

3. Le plus simple est de procéder ainsi :

```
let rec feuille_basse_naif arbre =
  match arbre with
  | F x -> x
  | N (_, g, d) ->
    if hauteur g >= hauteur d then feuille_basse_naif g
    else feuille_basse_naif d
```

Cependant, cette fonction n'a pas une complexité satisfaisante : si l'arbre est un peigne, par exemple, elle est en temps quadratique en la taille de l'arbre. Une version plus efficace

utilise une fonction auxiliaire qui calcule la hauteur de l’arbre en même temps qu’elle trouve la feuille qui nous intéresse :

```
let feuille_basse arbre =
  let rec aux = function
    | F x -> (x, 0)
    | N (_, g, d) ->
        let (xg, hg) = aux g in
        let (xd, hd) = aux d in
        if hg <= hd then (xg, 1 + hg)
        else (xd, 1 + hd) in
      let x, _ = aux arbre in x
```

4. La version la plus naturelle (parmi celles qui ont la bonne complexité) :

```
let lire_racine arbre =
  match arbre with
  | F x -> x
  | N (x, _, _) -> x

let rec arbre_hauteurs arbre =
  match arbre with
  | F _ -> F 0
  | N (_, g, d) ->
    let g' = arbre_hauteurs g in
    let d' = arbre_hauteurs d in
    let h = 1 + max (lire_racine g') (lire_racine d') in
    N (h, g', d')
```

Correction de l’exercice 10.11 page 172

1. Une seule solution raisonnable :

```
let rec est_binaire_strict arbre =
  match arbre with
  | N (_, []) -> true
  | N (_, [a; b]) -> est_binaire_strict a && est_binaire_strict b
  | _ -> false
```

2. La manière la plus générale de traiter ce genre de choses : on applique la fonction qui nous intéresse à chaque enfant (c’est un **List.map**), puis l’on calcule le résultat (en faisant la somme de la liste, ici) :

```
let rec somme_liste liste =
  match liste with
  | [] -> 0
  | x :: xs -> x + somme_liste xs

let rec somme (N (x, enfants)) =
  x + somme_liste (List.map somme enfants)
```

Ici, on pouvait procéder autrement :

```
let rec somme_bis arbre =
  match arbre with
  | N (x, []) -> x
  | N (x, e :: es) -> somme_bis e + somme_bis (N (x, es))
```

Correction de l'exercice 10.12 page 172

1. À savoir faire les yeux fermés :

```
let rec taille = function
| V -> 1
| N (_, g, d) -> 1 + taille g + taille d
```

2. On traduit directement la définition :

```
let rec poids = function
| V -> 0
| N (_, g, d) ->
  if taille g < taille d then 1 + poids g + poids d
  else poids g + poids d
```

3. Posons $t_0 = V$ et $t_{n+1} = N(0, V, t_n)$. On a $|t_n| = n$, et si l'on note $\varphi(n)$ le nombre d'opérations pour le calcul de $poids_{t_n}$, on a :

$$\varphi(t_n) = \varphi(t_{n-1}) + \Omega(n)$$

puisque l'appel `taille d` se fait en temps proportionnel à n . On en déduit immédiatement $\varphi(t_n) = \Omega(n^2)$.

4. Il faut calculer *simultanément* la taille et le poids :

```
let poids_efficace arbre =
(* aux a renvoie le couple (poids a, taille a) *)
let rec aux = function
| V -> (0, 0)
| N (_, g, d) ->
  let (pg, tg) = aux g in
  let (pd, td) = aux d in
  if pg < pd then (1 + pg + pd, 1 + tg + td)
  else (pg + pd, 1 + tg + td) in
let p, _ = aux arbre in
p
```

Correction de l'exercice 10.13 page 173

1. Pas de problème :

```
let rec peigne_gauche n =
  if n = 0 then V
  else N (n, peigne_gauche (n - 1), E)
```

2. Idem :

```
let rec peigne_droit n =
  if n = 0 then V
  else N (n, V, peigne_droit (n - 1))
```

3. C'est plus délicat (et la solution n'est pas unique, puisque la forme de l'arbre n'est pas entièrement spécifiée) :

```
let complet n =
  (* renvoie un arbre complet avec noeuds i, ..., j-1 *)
  let rec aux i j =
    if i >= j then V
    else
      let nb_droit = (j - i) / 2 in
      let premier_gauche = i + nb_droit in
      N (j - 1, aux premier_gauche (j - 1), aux i premier_gauche) in
    aux 1 (n + 1)
```

4. Les sous-arbres gauche et droit sont identiques, on ne fait donc qu'un seul appel récursif :

```
let rec parfait n =
  if n = 0 then V
  else
    let a = parfait (n - 1) in
    N (n, a, a)
```

5. À la lecture du code, on voit immédiatement que la complexité vérifie $\varphi(n) = \varphi(n-1) + C$, où C est une constante, et donc $\varphi(n) = O(n)$. Cela peut sembler étrange puisque l'arbre que l'on construit contient $2^n - 1$ nœuds (non vides) : il faut comprendre qu'en mémoire, on a ici un partage maximal, ce qui fait que l'arbre occupe une place proportionnelle à n .

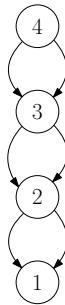


FIGURE 10.17 – Schéma mémoire simplifié pour `parfait 4`.

Correction de l'exercice 10.14 page 173

- On propose ici une version récursive utilisant une pile impérative (vous trouverez dans la correction du dernier TP une version itérative utilisant également une file impérative).

```

let liste_largeur arbre =
  let file = Queue.create () in
  Queue.push arbre file;
  let rec loop () =
    if Queue.is_empty file then []
    else
      match Queue.pop file with
      | V -> loop ()
      | N (x, g, d) ->
        Queue.push g file;
        Queue.push d file;
        x :: loop () in
  loop ()

```

2. C'est faisable en itératif sans trop de problème : essentiellement, il suffit de produire la liste des noeuds dans l'ordre du parcours en largeur, puis reconstruire l'arbre en ajoutant les numéros. On propose ici une solution entièrement fonctionnelle, assez subtile en fait :

```

type 'a file = 'a list * 'a list

let push x (entree, sortie) = (x :: entree, sortie)

let rec pop file =
  match file with
  | [], [] -> failwith "pop sur file vide"
  | entree, x :: xs -> (x, (entree, xs))
  | entree, [] -> pop ([], List.rev entree)

let file_vide = ([], [])

let est_vide file =
  file = ([], [])

let numerote_largeur a =
  let rec aux i foret =
    if est_vide foret then file_vide
    else match pop foret with
    | V, xs -> push V (aux i xs)
    | N (x, g, d), xs ->
      let foret' = xs |> push g |> push d |> aux (i + 1) in
      let d', foret' = pop foret' in
      let g', foret'' = pop foret' in
      push (N ((x, i), g', d')) foret'' in
  let initial = push a file_vide in
  let arbre, _ = pop (aux 1 initial) in
  arbre

```

DICTIONNAIRES

I Ensembles, dictionnaires

1.1 Type abstrait SET

La structure de données abstraite SET correspond à la notion mathématique d'ensemble : il n'y a pas de répétition, et les éléments ne sont pas ordonnés. Une signature possible est donnée ci-dessous, pour des ensembles **fonctionnels** :

Opération	Type	Commentaire
<i>Opérations caractéristiques</i>		
member	'a -> 'a set -> bool	Test d'appartenance
add	'a -> 'a set -> 'a set	
remove	'a -> 'a set -> 'a set	
<i>Opérations complémentaires</i>		
empty_set	'a set	
is_empty	'a set -> bool	
iter	('a -> unit) -> 'a set -> unit	« Boucle for »
equal	'a set -> 'a set -> bool	Égalité ensembliste

FIGURE 11.1 – Signature minimale pour un type abstrait SET fonctionnel

Remarque

Cette signature est suffisante, mais il manque bien sûr de nombreuses fonctions très utiles que l'on ajouterait si l'on écrivait une « vraie » bibliothèque.

La spécification des différentes fonctions est intuitivement évidente. Cependant, il n'est jamais inutile de formaliser notre intuition ; pour ce faire, on associe à tout objet s : ' a set un ensemble (au sens mathématique du terme) $\varphi(s)$ et l'on exige :

- $\varphi(\text{empty_set}) = \emptyset$
- $\varphi(\text{add } x \ s) = \varphi(s) \cup \{x\}$
- $\varphi(\text{remove } x \ s) = \varphi(s) \setminus \{x\}$
- $\text{member } x \ s$ si et seulement si $x \in \varphi(s)$
- $\text{equal } s \ s'$ si et seulement si $\varphi(s) = \varphi(s')$
- si $\varphi(s) = \{x_1, \dots, x_n\}$, alors $\text{iter } f \ s$ a le même effet que `List.iter f [x_1; ...; x_n]` (attention, l'ordre des éléments est arbitraire).

Exercice II.1 – Nombre d'éléments distincts

p. 208

On suppose que l'on dispose d'un type ' a set doté des opérations définies ci-dessus. Écrire une fonction `cardinal_liste` : ' a list -> int qui renvoie le nombre d'éléments **distincts** de son argument.

1.2 Type abstrait MAP

Le type abstrait MAP (ou Dict) correspond à un dictionnaire, c'est-à-dire à une application partielle d'un ensemble A dans un ensemble B. Les éléments de A sont appelés *clés* et ceux de B *valeurs*. On parle généralement de *dictionnaire* ou de *tableau associatif*.

Ici, on présente une signature fonctionnelle :

Opération	Type	Commentaire
<i>Opérations caractéristiques</i>		
get	'a -> ('a, 'b) map -> 'b option	None si la clé est absente
set	'a -> 'b -> ('a, 'b) map -> ('a, 'b) map	Ajout ou remplacement
remove	'a -> ('a, 'b) map -> ('a, 'b) map	
<i>Opérations complémentaires</i>		
empty_map	('a, 'b) map	
iter	('a * 'b -> unit) -> ('a, 'b) map -> unit	

FIGURE 11.2 – Signature minimale pour un type abstrait MAP fonctionnel

Formellement, on associe à tout $d : ('a, 'b) \text{ map}$ une relation fonctionnelle $\varphi(d) \subset A \times B$, c'est-à-dire un ensemble de couples vérifiant $((x, y) \in \varphi(d) \text{ et } (x, y') \in \varphi(d)) \Rightarrow y = y'$.

On demande alors :

- $\varphi(\text{empty_map}) = \emptyset$
- $\text{get } x \text{ m} = \text{Some } y \text{ si } (x, y) \in \varphi(\text{m})$ (y est nécessairement unique)
- $\text{get } x \text{ m} = \text{None}$ s'il n'y a pas de couples de la forme (x, y) dans $\varphi(\text{m})$
- $\varphi(\text{remove } x \text{ m}) = \varphi(\text{m}) \setminus \{(x, y) \mid y \in B\}$
- $\varphi(\text{set } x \text{ y m}) = \varphi(\text{remove } x \text{ m}) \cup \{(x, y)\}$

Exercice II.2 – Listes d'associations

p. 208

La manière la plus simple de réaliser le type abstrait Dict est d'utiliser une liste de couples (*clé, valeur*) telles que les clés soient deux à deux distinctes. On suppose (uniquement dans cet exercice) que l'on choisit cette réalisation :

```
type ('a, 'b) dict = ('a * 'b) list
```

Écrire en OCaml des fonctions get et set respectant la spécification donnée ci-dessus, et donner leur complexité.

Exercice II.3 – Fonctions de conversion

p. 208

En général, on choisira des réalisations plus efficaces que les listes d'associations. Cependant, il est toujours utile de pouvoir convertir entre un objet de type Dict et une liste d'associations, et les fonctions présentes dans la signature suffisent pour réaliser cette conversion. Écrire en OCaml les deux fonctions suivantes (en supposant que l'on dispose de toutes les fonctions de la signature pour le type ('a, 'b) dict) :

1. `of_list` : ('a * 'b) list -> ('a, 'b) map (on pourra supposer que la liste fournie ne contient pas deux associations différentes pour la même clé);
2. `to_list` : ('a, 'b) map -> ('a * 'b) list

Remarque

Dans cet exercice, on ne sait *absolument pas* comment est réalisé le type ('a, 'b) dict (en particulier, il n'est pas *a priori* égal à ('a * 'b) list).

Exercice II.4 – Listes d'antécédents

p. 208

On considère un 'a array de longueur n que l'on voit comme une application $f : [0 \dots n] \rightarrow A$. Écrire une fonction antecedents : 'a array -> ('a, int list) map qui renvoie un dictionnaire m dont les clés sont les $y \in f([0 \dots n - 1])$ et où la valeur associée à un y est la liste des $x \in [0 \dots n]$ tels que $f(x) = y$.

2 Arbres binaires de recherche

2.1 Définitions

Les *arbres binaires de recherche* fournissent une réalisation naturelle (et efficace) des types abstraits SET et DICT. Nous nous concentrerons d'abord sur le type SET, et nous verrons ultérieurement que l'on peut facilement adapter au type DICT.

Définition II.1 – Arbre binaire de recherche

Soit (A, \leq) un ensemble totalement ordonné. On définit l'ensemble $\mathcal{T}(A)$ par :

- $\perp \in \mathcal{T}(A)$
- si $g \in \mathcal{T}(A)$, $d \in \mathcal{T}(A)$ et $x \in A$, alors $(g, x, d) \in \mathcal{T}(A)$.

Si $t \in \mathcal{T}(A)$, on définit $\varphi(t)$ par :

- $\varphi(\perp) = \emptyset$
- $\varphi(g, x, d) = \{x\} \cup \varphi(g) \cup \varphi(d)$

L'ensemble $ABR(A)$ des *arbres binaires de recherche* sur A est défini par :

- $\perp \in ABR(A)$;
- $(g, x, d) \in ABR(A)$ si et seulement si
 - g et d sont dans $ABR(A)$;
 - $\max(\varphi(g)) < x < \min(\varphi(d))$.

Remarques

- Dans un tel arbre, une feuille est un noeud de la forme (\perp, x, \perp) . Quand on représente graphiquement l'arbre, on omet systématiquement les fils vides.
- On prend la convention $\max \emptyset < x < \min \emptyset$ pour tout $x \in A$.
- Le fait qu'on ait choisi des inégalités strictes dans la définition d'un ABR garantit que les étiquettes sont deux à deux distinctes. On manipule donc des ensembles (et non des multi-ensembles).

Exemple II.5 – Arbre binaire de recherche

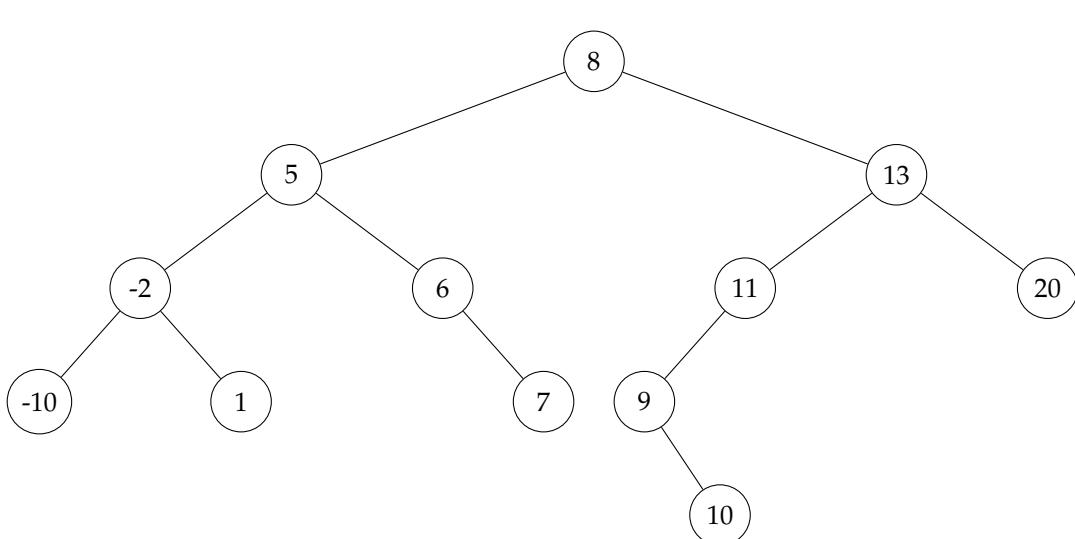


FIGURE 11.3 – Un arbre binaire de recherche pour l'ensemble $\{-10; -2; 1; 5; 6; 7; 8; 9; 10; 11; 13; 20\}$

Attention, la propriété d'ABR n'est pas « locale » : il ne suffit pas de vérifier que chaque noeud a une étiquette plus grande que celle de son fils gauche et plus petite que celle de son fils droit.

Exemple II.6 – Violation de la propriété d'ABR

Ceci n'est pas un ABR : il y a deux violations de la propriété (qui ont été mises en évidence). Si l'on se limitait à comparer les étiquettes des pères avec celles de leurs fils, on ne détecterait pas de problème.

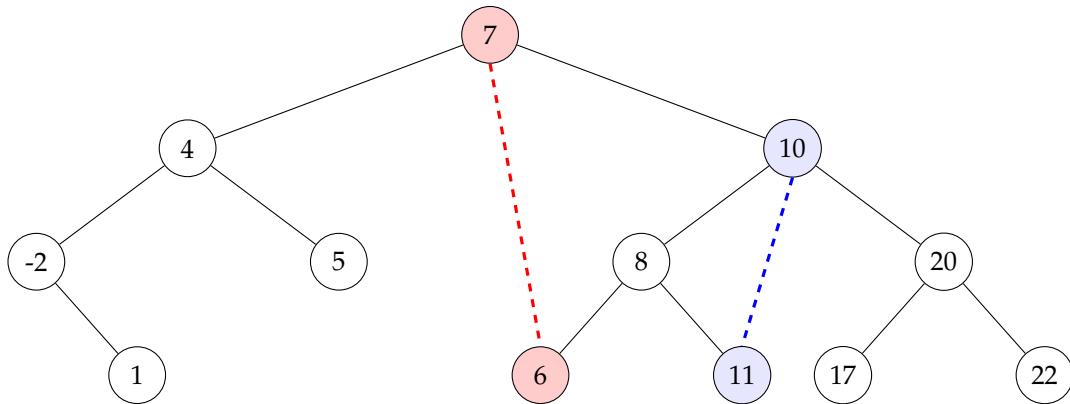


FIGURE 11.4 – Propriété d'ABR non vérifiée

Pour un ensemble donné d'étiquettes, de nombreux ABR sont possibles et leur forme peut varier énormément.

Exercice II.7

Donner un ABR de hauteur maximale et deux (distincts !) de hauteur minimale pour l'ensemble $\{1, 2, 3, 4, 5, 6\}$.

Théorème II.2

t est un ABR si et seulement si $\text{infixe}(t)$ est triée par ordre croissant (où $\text{infixe}(t)$ désigne la liste des étiquettes de t dans l'ordre infixé).

Démonstration

Par induction structurelle :

- si l'arbre est vide, il n'y a rien à prouver ;
- sinon, l'arbre est de la forme $t = (g, x, d)$. En notant l_g, l_d, l_t la liste des étiquettes de g, d, t dans l'ordre infixé, on a par définition $l_t = l_g, x, l_d$ (où la virgule représente une concaténation).
 - Supposons que t soit un ABR. On a alors :
 - g et d sont des ABR par définition, donc l_g et l_d sont croissantes par hypothèse d'induction ;
 - $\max \varphi(g) < x < \min \varphi(d)$ (avec les conventions usuelles pour $\max \emptyset$ et $\min \emptyset$) puisque la condition d'ordre est vérifiée à la racine de t , et donc $\max l_g < x < \min l_d$.

On en déduit que l_t est croissante.

- Inversement, supposons que l_t soit croissante. On a alors :
 - l_g et l_d croissantes, donc par hypothèse d'induction g et d sont des ABR.
 - $\max l_g < x < \min l_d$, donc $\max \varphi(g) < x < \min \varphi(d)$ et la condition d'ordre est vérifiée à la racine de t .

Donc t est un ABR.

Remarques

- À chaque niveau de l'arbre, les étiquettes lues de gauche à droite forment une suite croissante.
- En fait, si l'on représente « proprement » l'arbre, alors la lecture des étiquettes de gauche à droite, indépendamment du niveau, correspond à un parcours infixé (et donne donc une suite croissante).
- Le minimum est obtenu en descendant à gauche depuis la racine jusqu'à arriver à la fin de la branche.
- De même pour le maximum en descendant à droite.

2.2 Algorithmes élémentaires

2.2.a Représentation en OCaml

Pour représenter en OCaml un arbre binaire dans lequel un nœud peut n'avoir qu'un seul fils, on utilise le type suivant :

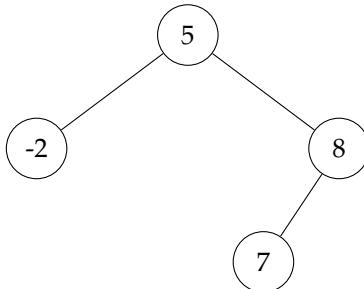
```
type 'a abr =
| V
| N of 'a abr * 'a * 'a abr
```

Remarques

- On rappelle que le **of** n'est présent que dans la définition de type (et pas quand on utilise ultérieurement le type).
- Le constructeur de type **N**, en revanche, ne doit pas être omis.

Exercice 11.8

1. Comment représente-t-on une feuille avec ce type ?
2. Définir en OCaml l'arbre suivant :



3. Dessiner l'arbre défini par :

```
N (N (V, 0, V),
      1,
      N (N (V, 2, N (V, 4, V)),
            5,
            N (V, 6, V)))
```

2.2.b Représentation en C

En C, on représentera un nœud d'un ABR par une **struct** :

```
typedef int item; // Pour des clés entières

struct BST {
    item key;
    struct BST *left;
    struct BST *right;
};

typedef struct BST bst;
```

L'arbre vide est représenté par un pointeur nul, et la fonction suivante permet donc de créer une nouvelle feuille :

```

bst* new_node(item x){
    bst *node = malloc(sizeof(bst));
    node->key = x;
    node->left = NULL;
    node->right = NULL;
    return node;
}

```

Dans la suite, on supposera toujours que les nœuds d'un ABR ont été créés par des appels à cette fonction `new_node`.

► Exercice II.9

p. 209

Écrire une fonction `bst_free` qui libère la mémoire occupée par un ABR.

```
void bst_free(bst* t);
```

2.2.c Recherche

Pour rechercher un élément dans un ABR, on descend le long d'une branche depuis la racine jusqu'à trouver l'élément ou arriver à un sous-arbre vide. Tout l'intérêt de la propriété d'ABR est de permettre de n'explorer **qu'une seule branche**.

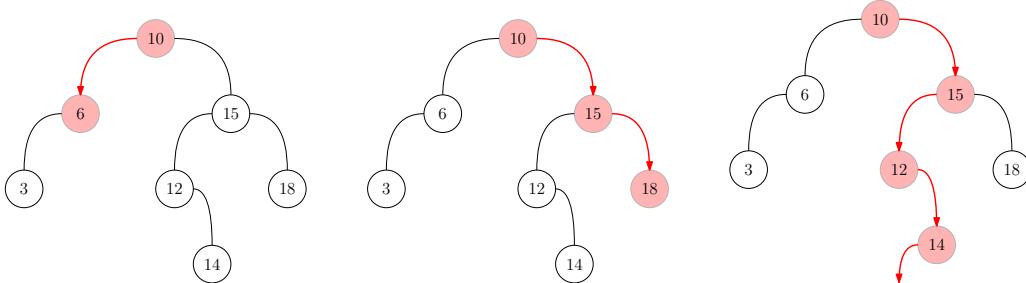


FIGURE 11.5 – Recherche de 6, de 18 et de 13 dans un ABR

On notera la grande similarité entre la recherche dans un ABR et la **recherche dichotomique dans un tableau trié**.

► Exercice II.10

p. 209

1. Écrire une fonction OCaml `appartient` : `'a abr -> 'a -> bool`.
2. Montrer la correction de cette fonction.

► Exercice II.11

p. 209

Écrire une fonction `minimum` : `'a abr -> 'a` renvoyant la plus petite étiquette d'un ABR. On lèvera une exception si l'arbre est vide.

► Exercice II.12

p. 209

Écrire la version C de ces deux fonctions :

```

bool bst_member(bst *t, item x);
item bst_minimum(bst *t);

```

2.3 Insertion

Les insertions se font toujours en bas de l'arbre (l'élément ajouté sera une feuille). On descend donc le long d'une branche, comme lors d'une recherche, de manière à insérer l'élément au bon endroit. Si l'élément est déjà présent, il sera nécessairement rencontré lors du parcours : dans ce cas, on renvoie l'arbre inchangé (si l'on réalise la structure SET).

Remarque

Tout l'intérêt d'un ABR par rapport à un tableau trié vient de la possibilité de réaliser efficacement des insertions (et des suppressions).

Exercice II.13

p. 210

1. Écrire une fonction `insere` : '`a abr -> 'a -> 'a abr` tel que `insere a x` renvoie un ABR contenant les mêmes éléments que `a`, ainsi que l'élément `x`. Si `x` était déjà présent dans `a`, on renverra l'arbre inchangé.
2. Quelle modification faut-il apporter au code précédent si l'on souhaite qu'un élément puisse avoir plusieurs occurrences (structure `MULTISSET`) ?

En C, même si le principe reste le même, il va falloir procéder un peu différemment : la structure d'arbre que nous utilisons est mutable. La fonction `bst_insert` peut être programmée ainsi :

```
bst* bst_insert(bst* t, item x){
    if (t == NULL) {
        return new_node(x);
    }
    if (t->key < x) {
        t->right = bst_insert(t->right, x);
    } else if (t->key > x) {
        t->left = bst_insert(t->left, x);
    }
    return t;
}
```

Exercice II.14

p. 210

1. Ré-écrire la fonction `bst_insert` de manière itérative.
2. Est-il encore nécessaire que cette fonction renvoie un `bst*` (ou peut-elle renvoyer `void`) ? Si oui, pourquoi ?

Exercice II.15

p. 211

1. Écrire une fonction `construit` : '`a list -> 'a abr` renvoyant un ABR dont les étiquettes sont les éléments de la liste passée en argument.
2. Cette fonction est-elle récursive terminale ? Peut-on facilement en écrire une version terminale ?

Cette fonction appelle `insere` qui n'est pas récursive terminale, mais ce n'est pas la question : on s'intéresse juste à la fonction `construit` elle-même.
3. Préciser l'ordre dans lequel les insertions sont faites, dans les deux versions.
4. Prévoir (puis vérifier) l'arbre obtenu à partir des listes suivantes, en insérant les éléments dans leur ordre d'apparition dans la liste :
 - a. [1; 4; 2; 3; 0; 7; 5]
 - b. [0; 1; 2; 3; 4; 5]
 - c. [5; 4; 3; 2; 1; 0]

Exercice II.16

p. 212

Écrire une fonction C build, prenant en entrée un tableau d'objets de type item et sa taille, et renvoyant l'ABR obtenu en insérant les éléments du tableau, dans l'ordre, dans un arbre initialement vide.

```
bst *build(item t[], int len);
```

2.3.a Partage

Il faut bien comprendre que les structures que nous avons définies en OCaml et en C, bien qu'elles aient essentiellement la même représentation en mémoire, sont complètement différentes. En effet, on a une structure immuable en OCaml et mutable en C : dans un cas, l'insertion d'un élément dans un arbre crée un nouvel arbre, sans modifier l'arbre initial, alors que dans l'autre cas on a une mutation.

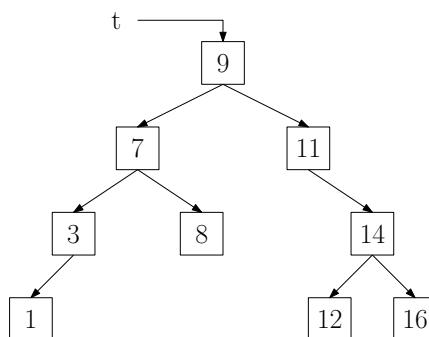
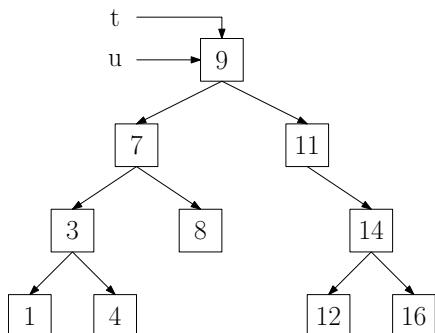
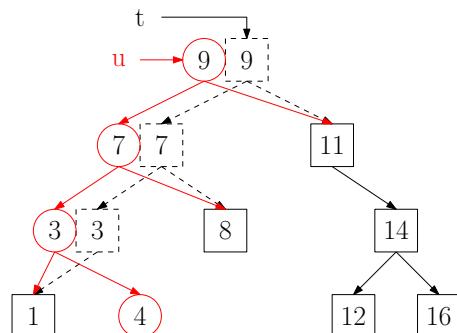


FIGURE 11.6 – Un ABR t (en C ou en OCaml).

FIGURE 11.7 – Insertion en C : résultat de l'opération `bst *u = bst_insert(t, 4)`. Les pointeurs u et t sont égaux.FIGURE 11.8 – Insertion en OCaml : résultat de l'opération `let u = insere t 4`. La partie pointillée appartient à t mais pas à u, la partie rouge à u mais pas à t, le reste est partagé.

Il est tout-à-fait possible de définir une structure d'ABR mutable en OCaml, mais l'intérêt est assez limité (le code est nettement moins élégant, en particulier). En revanche, définir et utiliser de manière sûre une structure immuable d'arbre en C est réellement problématique (on en vient rapidement à écrire un *garbage collector*...).

2.3.b Suppression

Étant donnés un ABR `t` représentant un ensemble $\varphi(t)$ et un élément `x`, on souhaite renvoyer un ABR `t'` représentant $\varphi(t) \setminus \{x\}$. Il y a trois cas simples :

- si l'élément à supprimer n'apparaît pas dans l'arbre, il n'y a rien à faire ;
- si l'élément est une feuille, il suffit de la supprimer ;
- si l'élément n'a qu'un seul fils non vide, il suffit de supprimer l'élément et de remonter son fils.

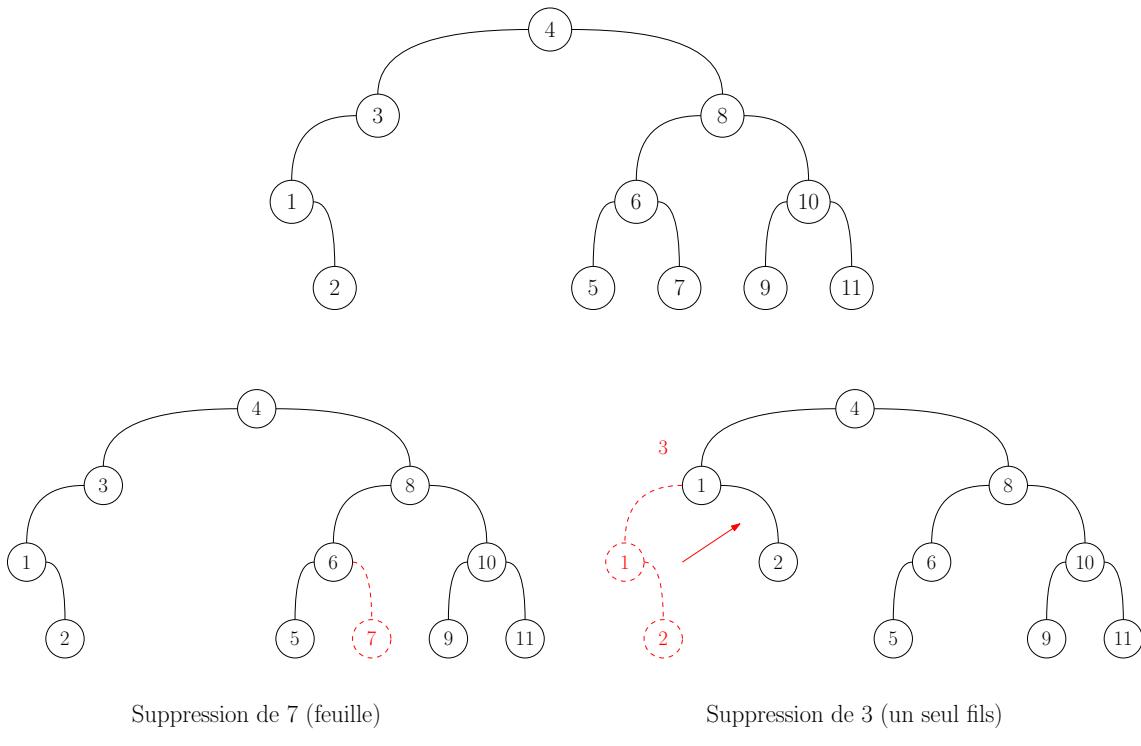


FIGURE 11.9 – Cas simples pour la suppression

On considère le code OCaml suivant, que l'on va compléter :

```

1 let rec supprime arbre elt =
2   match arbre with
3   | V ->
4   | N (V, x, dr) when x = elt ->
5   | N (ga, x, V) when x = elt ->
6   | N (ga, x, dr) when elt < x ->
7   | N (ga, x, dr) when elt > x ->
8   | N (ga, x, dr) ->

```

Exercice II.17 – Suppression d'un élément, cas simples

p. 212

Compléter les lignes 3 à 7 de la fonction supprime.

Dans le cas où le noeud à supprimer possède deux fils, c'est un peu plus compliqué. Une possibilité assez simple (mais **non satisfaisante !**) est la suivante :

- on remonte le sous-arbre droit d de l'élément x à supprimer;
- le sous-arbre gauche g de x devient le fils gauche du minimum de d .

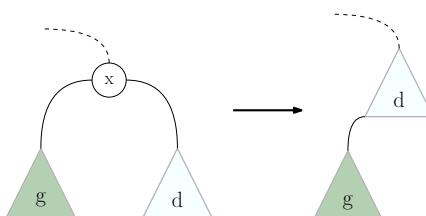


FIGURE 11.10 – Suppression d'un élément, méthode naïve.

Exercice II.18

p. 212

Justifier la correction de la méthode ci-dessus, et expliquer pourquoi elle n'est pas satisfaisante.

Pour faire mieux, il faut d'abord remarquer que le minimum d'un ABR ne peut avoir de fils gauche. Par conséquent, la suppression du minimum rentre toujours dans l'un des cas simples traités plus haut. On peut donc procéder ainsi :

- on considère le sous-arbre droit d de l'élément x à supprimer (d est non vide, sinon on serait dans l'un des cas simples) ;
- on récupère le minimum m de d ;
- on calcule d' , résultat de la suppression de m dans d ;
- on renvoie l'arbre (g, m, d') (autrement dit, on remplace x par le minimum de d , et l'on supprime ce minimum dans d).

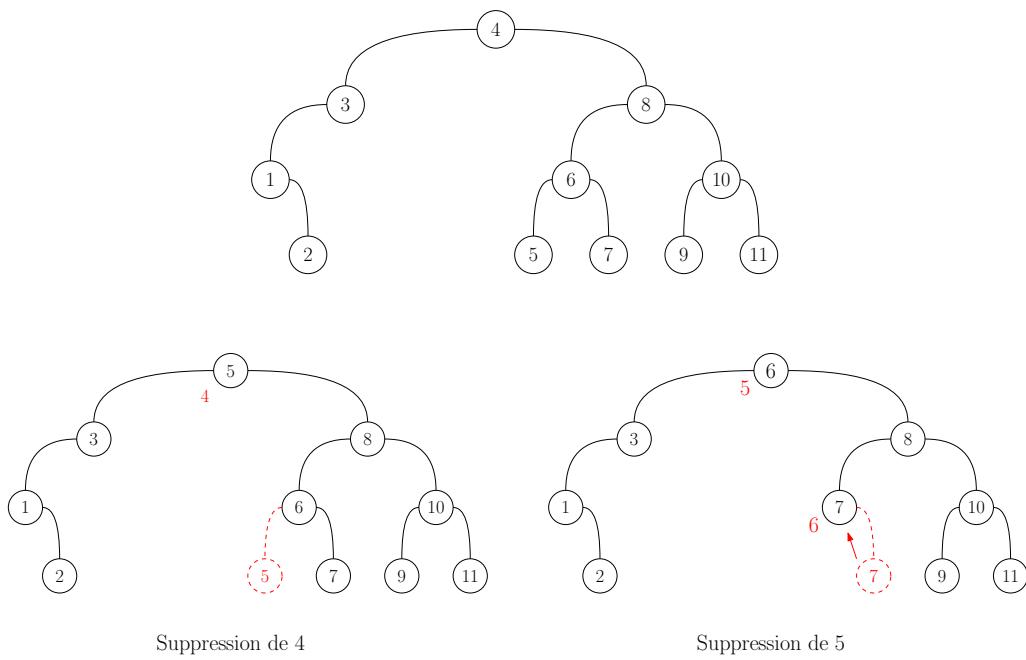


FIGURE 11.11 – Suppression d'un élément ayant deux fils

Exercice II.19 – Suppression d'un élément, cas général

p. 212

1. Écrire une fonction `supprime_min : 'a abr -> 'a abr` qui prend un ABR t supposé non vide et renvoie l'ABR t' obtenu en supprimant le minimum de t .
2. Compléter la ligne 8 de la fonction `supprime`.
3. Supposons que l'on dispose d'un ABR a ainsi que d'un élément x de a , et que l'on définisse :

```
let b = insere (supprime a x) x
```

A-t-on, en général, $a = b$?

Exercice II.20 – Suppression d'un élément, langage C

p. 213

1. Écrire une fonction `bst_delete_min` supprimant le minimum d'un ABR supposé non vide.

```
bst *delete_min(bst *t);
```

2. Écrire une fonction `bst_delete` supprimant un élément d'un ABR. Si l'élément n'apparaît pas dans l'ABR, ce dernier sera renvoyé inchangé.

```
bst *bst_delete(bst *t, item x);
```

2.3.c Complexité des opérations

Définition II.3 – Hauteur d'un ABR

La *hauteur* $h(t)$ d'un ABR t est définie par :

- $h(\perp) = -1$;
- $h(g, x, d) = 1 + \max(h(g), h(d))$

Remarques

- Fixer la hauteur de l'arbre vide à -1 plutôt qu'à 0 est une convention ; elle n'est pas universelle, mais c'est celle du programme.
- Autrement dit, la hauteur d'un ABR est le nombre maximal d'arêtes traversées le long d'un chemin reliant la racine à une feuille de l'arbre (où une feuille est un nœud de la forme **N** (**V**, **x**, **V**)).

Propriété II.4

Soit t un ABR de hauteur h . Une opération de recherche, d'insertion ou de suppression dans t effectue au plus $h + 1$ comparaisons entre clés.

Démonstration

Pour chacune de ces fonctions, on effectue au plus une comparaison par niveau de l'arbre, puisque l'on ne compare l'élément à rechercher (insérer, supprimer) qu'avec des éléments situés sur un même chemin depuis la racine. ■

Théorème II.5

Soit t un ABR de hauteur h . En supposant que la comparaison de deux clés peut se faire en temps constant, les opérations de recherche, d'insertion et de suppression peuvent s'effectuer en temps $O(h)$.

Si le nombre d'étiquettes de t est $n \geq 1$, on a

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Démonstration

Pour la recherche et l'insertion, on descend le long d'un chemin depuis la racine en effectuant des opérations en temps constant pour chaque nœud rencontré.

Pour la suppression telle que nous l'avons écrite, certains nœuds peuvent être vus deux fois : une fois l'élément à supprimer trouvé, on commence par déterminer le minimum de son sous-arbre droit (dans l'un des cas) avant de le supprimer. Cependant, on reste bien en $O(h)$. ■

Remarques

- Si les clés sont des entiers ou des flottants, les comparaisons sont bien en temps constant. Si on a par exemple des chaînes de caractères, c'est plus compliqué puisque la comparaison prend un temps proportionnel à la longueur du plus grand préfixe commun : en pratique, on utilise des variantes spécialisées dans ce cas. On peut bien sûr toujours affirmer qu'on fera au plus h comparaisons.
- Le pire cas est celui d'un *peigne* pour lequel les performances sont catastrophiques (temps $\Theta(n)$).
- En insérant les étiquettes par ordre croissant, on obtient justement un peigne...
- Si l'arbre est le résultat d'une suite aléatoire d'insertions et de suppressions, on aura en moyenne du $\Theta(\log n)$ (nous le verrons en exercice).

3 Réalisation d'un dictionnaire par un ABR

Les ABR permettent de réaliser de manière fonctionnelle (persistante) la structure de dictionnaire, à condition que les clés soient choisies dans un ensemble totalement ordonné. Pour ce faire, on stocke dans les nœuds les couples (clé, valeur) et toutes les comparaisons se font uniquement sur les clés.

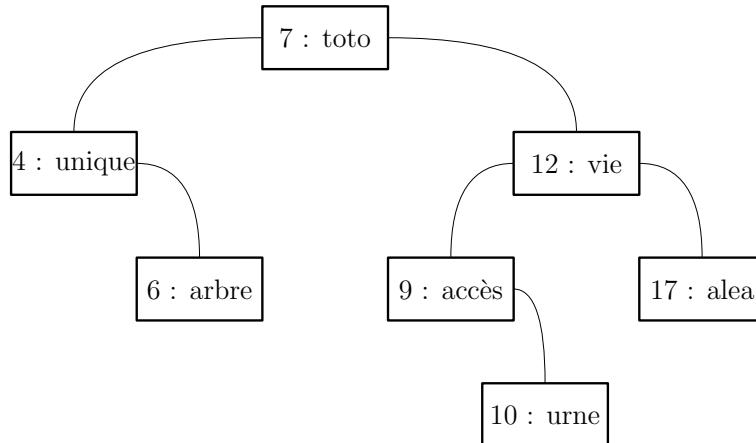


FIGURE 11.12 – Exemple de (`int, string`) dict.

On remarquera que dans l'exemple ci-dessus, la propriété d'ordre des ABR s'applique uniquement aux clés (entières). Les valeurs n'ont d'ailleurs pas de raison d'être choisies dans un type ordonné.

Exercice II.21 – Opérations élémentaires sur les dictionnaires fonctionnels

p. 214

On considère le type suivant :

```

type ('k, 'v) dict =
| Vide
| Noeud of 'k * 'v * ('k, 'v) dict * ('k, 'v) dict
  
```

Écrire les fonctions suivantes (dont les spécifications sont données en-dessous de la figure 11.2) :

```

get : 'k -> ('k, 'v) dict -> 'v option
set : 'k -> 'v -> ('k, 'v) dict -> ('k, 'v) dict
  
```

On obtient ainsi des complexités en $O(h)$ pour get, set et remove, ce qui est satisfaisant si l'arbre est raisonnablement équilibré (car on a alors $h = O(\log n)$).

Remarque

Pour un (`string, int`) dict (un dictionnaire dont les clés sont des chaînes de caractères), il est possible d'utiliser un ABR (l'ensemble des chaînes de caractères est totalement ordonné par l'ordre lexicographique), mais c'est rarement la bonne solution. On préférera utiliser une variante de *trie* (comme vu en travaux pratiques) ou une table de hachage (dont nous parlerons à la fin de ce chapitre). En effet, la comparaison de deux chaînes de caractères est coûteuse : elle prend un temps proportionnel à leur plus grand préfixe commun.

4 Arbres auto-équilibrés

4.1 Arbres 2-3

4.1.a Structure d'un arbre 2-3

Définition II.6 – Arbres 2-3

Un *arbre 2-3* est :

- soit l'arbre vide ;
- soit de la forme $N(g, x, d)$ où x est une étiquette et g, d sont des arbres 2-3, vérifiant la condition

$$\max g < x < \min d$$

- soit de la forme $N(g, x, m, y, d)$, où x, y sont des étiquettes et g, m, d des arbres 2-3, vérifiant la condition

$$\max g < x < \min m \leq \max m < y < \min d$$

On impose de plus une condition d'*équilibrage parfait* : tous les nœuds vides sont à la même profondeur.

Remarque

À nouveau, on a pris des inégalités strictes ici (sauf pour $\min m$ et $\max m$, l'arbre m pouvant avoir zéro ou un élément), ce qui impose l'unicité des clés et correspond typiquement à une structure d'ensemble. On a également gardé la convention $\min \emptyset = +\infty$ et $\max \emptyset = -\infty$.

Exemple II.22

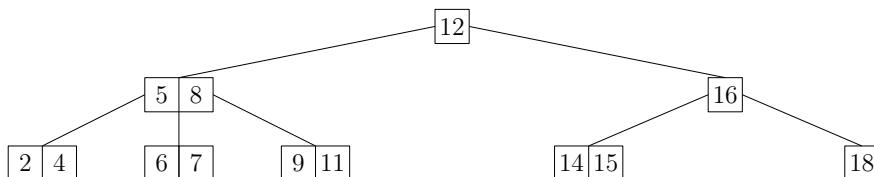


FIGURE 11.13 – Un exemple d'arbre 2-3 ; les nœuds vides ne sont pas représentés.

Exercice II.23

p. 214

Dessiner tous les arbres 2-3 ayant comme ensemble d'étiquettes :

- | | |
|-----------|---------------|
| 1. 1,2; | 3. 1,2,3,4; |
| 2. 1,2,3; | 4. 1,2,3,4,5. |

Propriété II.7 – Hauteur d'un arbre 2-3

En notant $h(t)$ la hauteur de t et $|t|$ son nombre d'étiquettes, on a :

$$h(t) = O(\log |t|)$$

Démonstration

Tous les nœuds vides sont à distance $h(t) + 1$ de la racine (en prenant $h(V) = -1$), donc il y a au moins 2^k nœuds à profondeur k . Il y a donc au moins 2^h étiquettes à profondeur h , d'où $2^h \leq n$ et donc $h \leq \log_2 n$.

■

Remarque

Plus précisément, on a $\lfloor \log_3(n+1) \rfloor - 1 \leq h \leq \lfloor \log_2(n+1) \rfloor - 1$.

4.1.b Recherche dans un arbre 2-3

Imaginons que l'on représente en OCaml un arbre 2-3 à l'aide du type suivant :

```
type 'a arbre23 =
| V
| B of 'a arbre23 * 'a * 'a arbre23
| T of 'a arbre23 * 'a * 'a arbre23 * 'a * 'a arbre23
```

On peut alors très facilement adapter l'algorithme de recherche dans un ABR :

```
let rec appartient x t =
match t with
| V -> false
| B (g, y, d) ->
  if x < y then appartient x g
  else if x > y then appartient x d
  else x = y
| T (g, y, m, z, d) ->
  if x < y then appartient x g
  else if x = y then true
  else if x < z then appartient x m
  else if x = z then true
  else appartient x d
```

Exercice II.24

p. 215

Ré-écrire cette fonction en remplaçant tous les **if .. then .. else** par des expressions booléennes construites avec **||** et **&&** (ce n'est pas forcément une bonne idée ici, la version donnée ci-dessus est sans doute plus facilement lisible).

Propriété II.8

La recherche d'une clé dans un arbre 2-3 contenant n clés se fait en temps $O(\log n)$ (sous l'hypothèse que les comparaisons entre clés se font en temps constant).

Démonstration

On descend le long d'une branche de l'arbre en faisant des opérations en temps constant (comparaison de clés, déconstruction) sur chaque noeud traversé. La complexité est donc en $O(h)$, c'est-à-dire en $O(\log n)$ d'après la propriété 11.7. ■

4.1.c Insertion dans un arbre 2-3

L'insertion d'un nouvel élément dans un arbre 2-3 débute comme l'insertion dans un arbre binaire : on descend dans l'arbre jusqu'à trouver l'endroit approprié. Par exemple, pour insérer 17 dans l'arbre de la figure 11.13, on commence par effectuer la descente figurée en gras sur le schéma suivant :

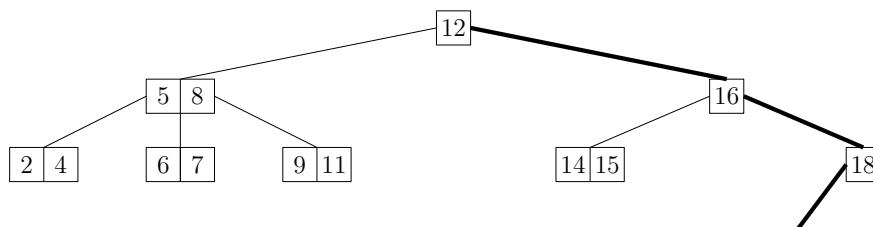


FIGURE 11.14 – Insertion de 17, recherche de l'emplacement.

Tout le problème est de maintenir la condition d'équilibrage parfait : on ne peut pas ajouter 17 comme fils gauche de 18 comme on le ferait pour un ABR. Ici, le dernier noeud rencontré est un noeud binaire, et il suffit donc de le transformer en noeud ternaire :

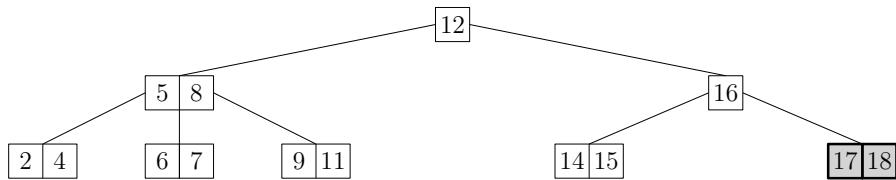


FIGURE 11.15 – Insertion de 17.

Si l'on décide ensuite d'insérer 13, on accepte de créer temporairement un nœud quaternaire :

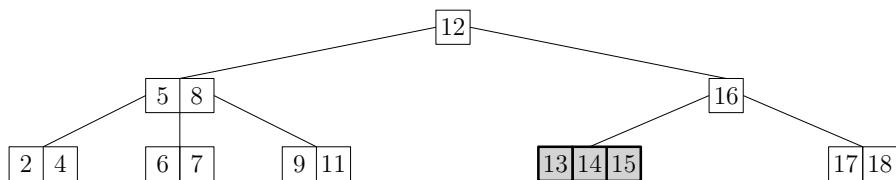


FIGURE 11.16 – Insertion de 13, étape intermédiaire.

On se débarrasse ensuite de ce nœud quaternaire en remontant l'une de ses étiquettes au niveau supérieur et en réarrangent l'arbre de manière à préserver la propriété d'ordre :

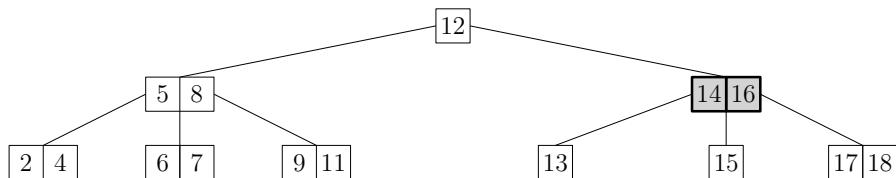


FIGURE 11.17 – Insertion de 13, suppression du nœud quaternaire.

Comme le père était un nœud binaire, il se transforme en nœud ternaire, ce qui ne pose pas de problème. Si l'on insère maintenant 10 dans l'arbre, il y aura une étape supplémentaire :

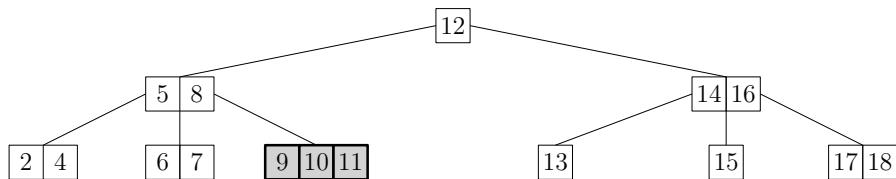


FIGURE 11.18 – Insertion de 10, étape initiale.

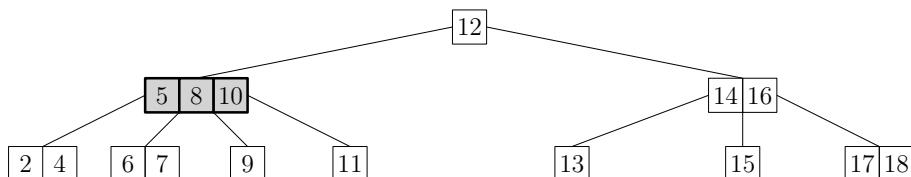


FIGURE 11.19 – Insertion de 10, étape intermédiaire.

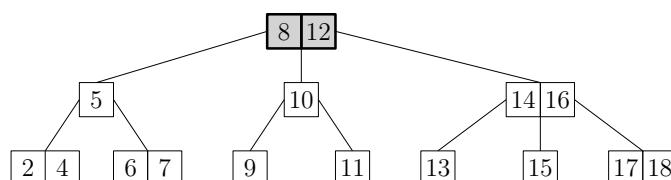


FIGURE 11.20 – Insertion de 10, état final.

Ce processus, consistant à remonter l'étiquette centrale d'un nœud quaternaire vers son père, à réorganiser l'arbre en conséquence et à recommencer sur le père, peut s'arrêter pour deux raisons :

- le père était un nœud binaire, qui devient un nœud ternaire ;
- il n'y a plus de père car on est remonté jusqu'à la racine.

Nous n'avons pas encore rencontré le deuxième cas jusqu'à présent. Insérons donc 22 dans le dernier arbre obtenu :

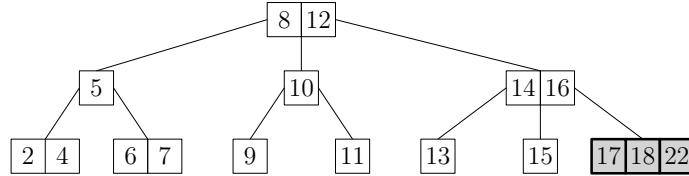


FIGURE 11.21 – Insertion de 22, première étape.

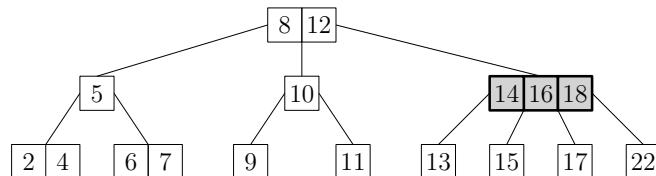


FIGURE 11.22 – Insertion de 22, deuxième étape.

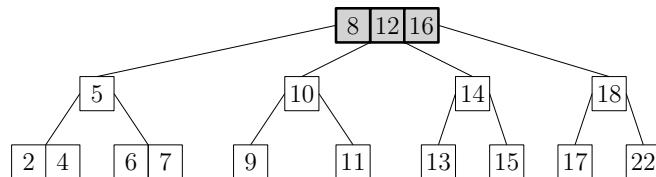


FIGURE 11.23 – Insertion de 22, troisième étape, racine quaternaire.

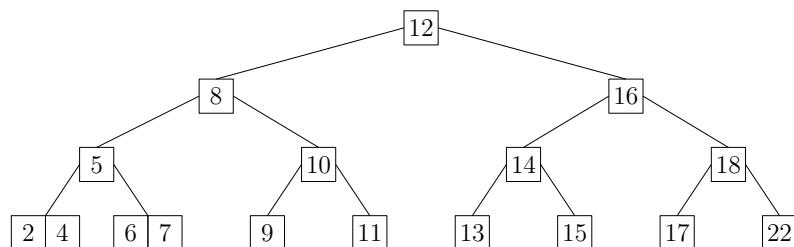


FIGURE 11.24 – Insertion de 22, état final après scission de la racine.

On notera que c'est le seul cas où la hauteur de l'arbre a augmenté, et que dans ce cas *tous* les nœuds vides ont vu leur profondeur augmenter de un (puisque la transformation a eu lieu à la racine). On a donc bien conservé la propriété d'équilibrage parfait.

On peut récapituler les différents cas pour faire remonter les nœuds quaternaires vers la racine (ou les éliminer) :

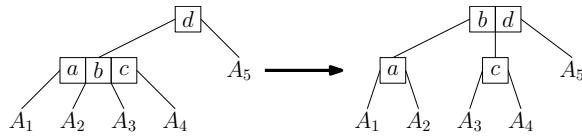


FIGURE 11.25 – Fils gauche d'un nœud binaire.

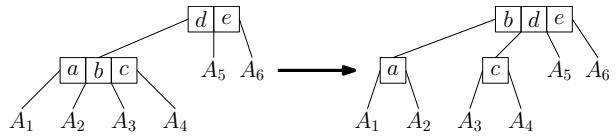


FIGURE 11.26 – Fils gauche d'un nœud ternaire.

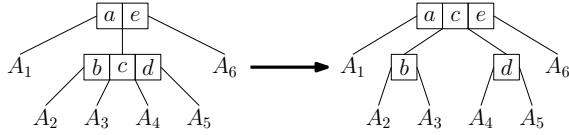


FIGURE 11.27 – Fils central d'un nœud ternaire.

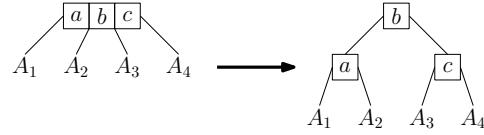


FIGURE 11.28 – Racine.

Exercice 11.25

p. 215

1. Justifier que les transformations décrites conservent les propriétés d'ordre et d'équilibrage parfait.
2. Compléter les deux cas manquants (fils droit d'un noeud binaire et fils droit d'un nœud ternaire).

Implémentation des arbres 2-3 Autant la recherche dans un arbre 2-3 se programme facilement, comme nous l'avons vu plus haut, autant l'insertion est un peu délicate. En effet, on peut être amené à définir un type intermédiaire autorisant les nœuds quaternaires, et, surtout, le nombre de cas à considérer est assez important.

4.2 Arbres rouge-noir

4.2.a Définition, correspondance avec les arbres 2-3

Les *arbres rouge-noir* (ou arbres *bicolores*) sont essentiellement une version « binarisée » des arbres 2-3, qui permet une implémentation plus simple. L'idée est de remplacer tous les nœuds ternaires par deux nœuds binaires, en marquant en rouge l'arête nouvellement créée (ainsi que le nœud auquel cette arête aboutit) :

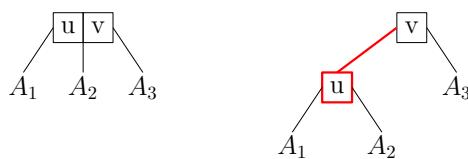


FIGURE 11.31 – Transformation d'un nœud ternaire vers deux nœuds binaires.

La propriété d'ordre des arbres 2-3 avant la transformation correspond exactement à la propriété d'ordre des ABR après la transformation :

$$\max A_1 < u < \min A_2 \leq \max A_2 < v < \min A_3$$

La propriété d'équilibrage totale est en revanche un peu modifiée : les nœuds vides des arbres A_1 et A_2 ont vu leur profondeur augmenter de un, alors que celle des nœuds de l'arbre A_3 est inchangée. On définit donc la *profondeur noire* d'un nœud comme le nombre d'arêtes noires dans le chemin reliant la racine à ce nœud. On voit alors que la propriété d'équilibrage parfait de l'arbre 2-3 initial se traduit par l'égalité des profondeurs noires de tous les nœuds vides.

Enfin, il est impossible d'obtenir deux arêtes rouges consécutives : en effet, cela correspondrait à un nœud quaternaire dans l'arbre initial. De plus, les arêtes rouges « penchent à gauche » dans notre version : autrement dit, un nœud rouge sera forcément le fils gauche d'un nœud noir.

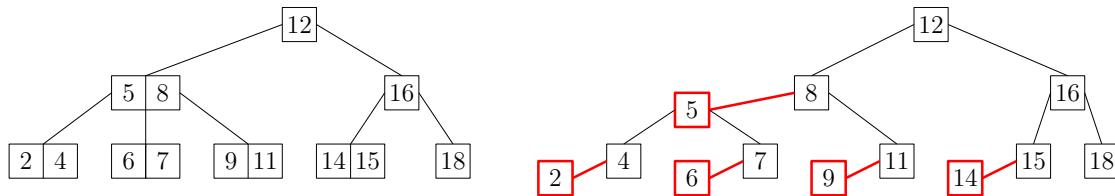


FIGURE 11.32 – Transformation d'un arbre 2-3 en arbre rouge-noir gauche.

Définition II.9 – Arbre rouge-noir gauche

Un *arbre rouge-noir gauche* est un arbre binaire de recherche dans lequel chaque nœud possède une couleur (rouge ou noir, les nœuds vides étant considérés comme noirs) et qui vérifie les conditions suivantes :

- aucun nœud rouge ne possède de fils rouge;
- tous les chemins reliant la racine à un nœud vide contiennent le même nombre de nœuds noirs;
- le fils droit d'un nœud est nécessairement noir;
- la racine est noire.

Remarques

- Les deux dernières conditions ne sont pas indispensables mais sont naturelles dans le cadre de la correspondance *arbre 2-3 \leftrightarrow arbre rouge-noir gauche*.
- On a choisi de placer les couleurs sur les nœuds plutôt que sur les arêtes (c'est plus pratique pour l'implémentation). Un nœud est rouge si et seulement si l'arête qui le relie à son père est rouge.

Propriété II.10 – Hauteur d'un arbre rouge-noir

Si t est un arbre rouge-noir, alors $h(t) = O(\log |t|)$.

Démonstration

Partons d'un arbre rouge-noir de taille n et de hauteur h , et transformons le en arbre 2-3. On obtient un arbre de taille n , et de hauteur $h' \geq h/2$ (les liens rouges ont été supprimés, et au plus la moitié des liens sont rouges sur un chemin reliant la racine à un nœud vide). D'après la propriété 11.7, on a donc $h \leq 2h' = O(\log n)$. ■

4.2.b Réalisation pratique

En C comme en OCaml, le plus simple est d'ajouter un « champ » couleur à chaque nœud non vide (les nœuds vides seront implicitement noirs). Dans les deux cas, ce n'est pas le plus efficace mais c'est sans importance pour nous.

```
type color = R | B

type 'a rb =
| E
| N of color * 'a rb * 'a rb * 'a rb
```

```
struct Node {
    struct Node *left;
    struct Node *right;
    datatype key;
    bool red;
};

typedef struct Node node;

struct RB {
    node *root;
};

typedef struct RB rb;
```

Exercice II.26

Écrire en OCaml et en C une fonction permettant de vérifier si un arbre vérifie bien les conditions de la définition 11.9.

```
check_rb : 'a rn -> bool
```

```
bool check_rb(rb *t);
```

4.2.c Recherche dans un arbre rouge-noir

La recherche dans un arbre rouge-noir se passe exactement comme dans un ABR classique : il suffit d'ignorer les indications de couleur.

Propriété II.11

La recherche dans un arbre rouge-noir contenant n nœuds se fait en temps $O(\log n)$, si la comparaison entre clés est en temps constant.

4.2.d Insertion dans un arbre rouge-noir

On peut voir l'insertion dans un arbre rouge-noir comme une simple traduction de celle dans un arbre 2-3 :

- on commence par insérer tout en bas, en transformant un nœud binaire en nœud ternaire ou un nœud ternaire en nœud quaternaire ;
- cela se traduit par la création d'une arête rouge vers notre nœud (qui sera donc rouge) ;
- la création de ce nœud a toutes les chances de violer les invariants d'arbre rouge-noir (tout comme on violait plus haut les invariants d'arbre 2-3) ;
- ces violations seront déplacées vers le haut de l'arbre, pour disparaître au plus tard à la racine.

On peut constater l'évolution parallèle sur les figures ci-dessous :

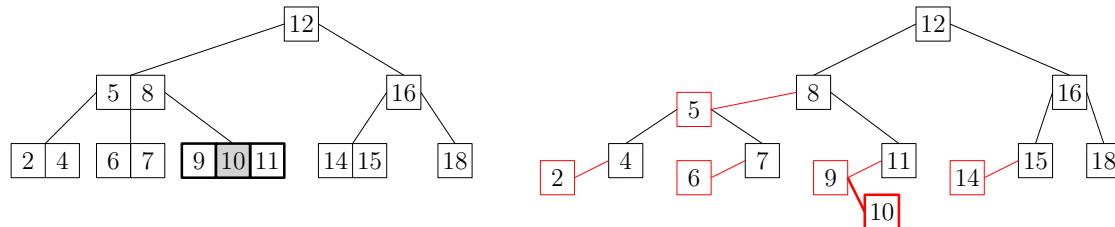


FIGURE 11.33 – Crédit d'une feuille rouge.

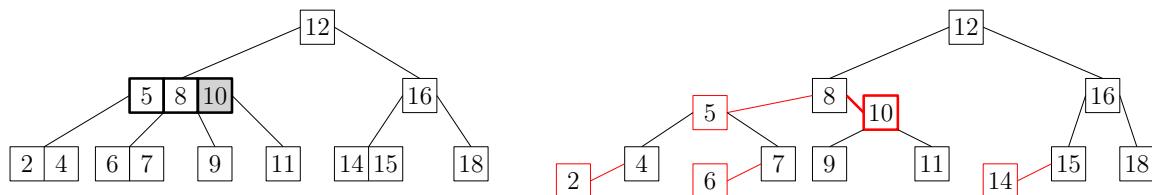


FIGURE 11.34 – Le problème se transmet au niveau supérieur, et persiste.

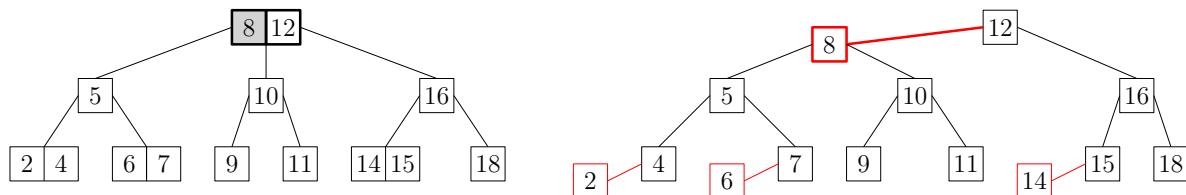


FIGURE 11.35 – Le problème se transmet au niveau supérieur, et disparaît.

4.2.e Rotation d'un ABR

La *rotation autour d'une arête* est l'opération fondamentale permettant de modifier la forme d'un ABR (pour le ré-équilibrer, le plus souvent) tout en respectant la propriété d'ordre.

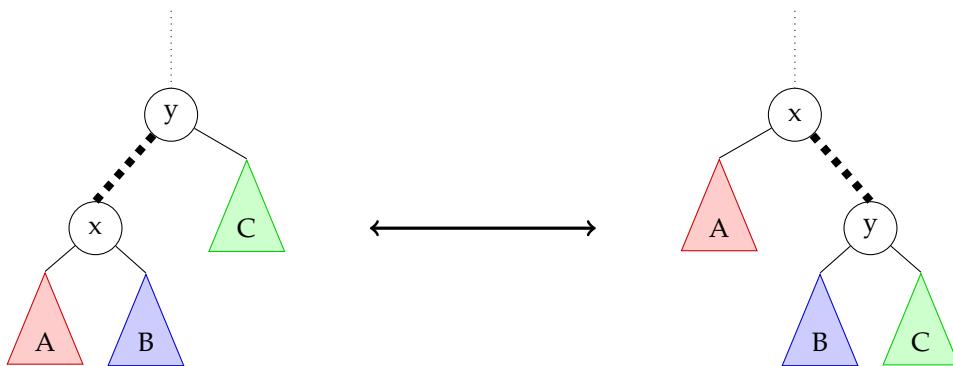


FIGURE 11.36 – Rotation autour de l'arête xy .

Propriété II.12

L'opération de rotation représentée ci-dessus préserve la propriété d'ordre des ABR. Autrement dit, l'arbre de gauche est un ABR si et seulement si l'arbre de droite en est un.

Remarque

C'est évident mais il est bon de le préciser : cette opération laisse invariant l'ensemble des étiquettes !

Rotation dans un arbre rouge-noir Dans un arbre rouge-noir, la rotation doit également gérer les couleurs des noeuds. L'idée est que l'*arête* tourne, et que ce sont les arêtes qui déterminent la couleur des noeuds (un noeud est rouge si et seulement si l'arête qui le relie à son père l'est). Dans le schéma ci-dessus, x et y échangent donc leurs couleurs.

Exercice II.27

p. 216

1. Écrire une fonction `rotate_right` qui effectue une rotation droite (sens gauche vers la droite dans le schéma ci-dessus). La fonction prendra y en argument et renverra x .
2. Écrire de même une fonction `rotate_left`.

```
node *rotate_right(node *y);
node *rotate_left(node *x);
```

4.2.f Cas d'insertion dans un arbre rouge-noir

On maintient le parallèle entre arbres 2-3 et arbres rouge-noir gauches. Les corrections se font en remontant dans l'arbre (c'est-à-dire très naturellement de manière récursive) :

```
node *node_insert(node *n, datatype k){
    if (n == NULL) return new_node(k); // new RED node
    if (k == n->key) return n;
    if (k < n->key) n->left = node_insert(n->left, k);
    if (k > n->key) n->right = node_insert(n->right, k);

    // restore invariants here!

    return n;
}
```

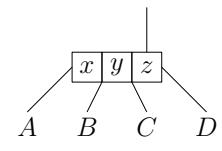
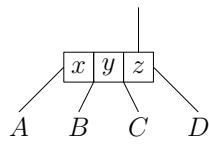
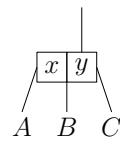
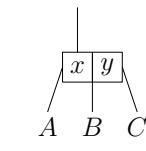


FIGURE 11.37 – Binaire → ternaire, cas problématique.

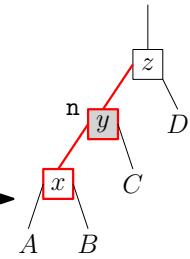
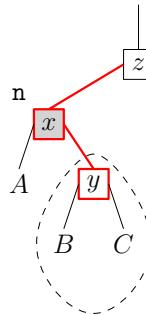
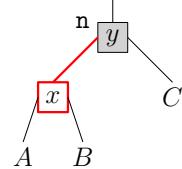
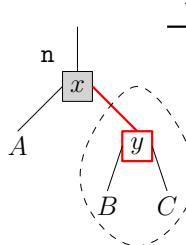


FIGURE 11.38 – Insertion au milieu d'un nœud ternaire.

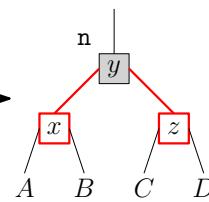
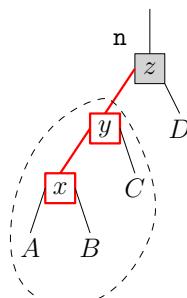
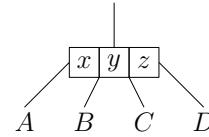
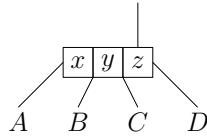


FIGURE 11.39 – Étape intermédiaire pour un nœud ternaire.

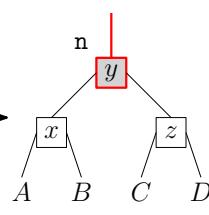
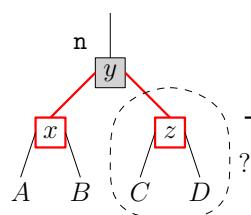
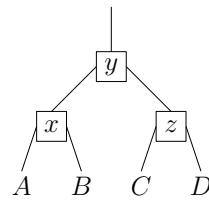
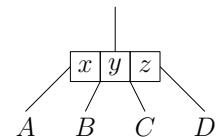


FIGURE 11.40 – Ternaire redevient binaire.

Exercice 11.28

p. 217

1. Écrire la fonction `new_node`, créant un nouveau nœud initialement rouge.
2. Écrire une fonction `is_red` indiquant si un nœud est rouge (cette fonction pourra être appelée sur un nœud vide).
3. Écrire la fonction `flip_colors` effectuant la transformation de la figure 11.40.
4. Compléter la partie manquante de la fonction `node_insert`.

```
node *new_node(datatype k);
bool is_red(node *n);
void flip_colors(node *n);
```

Racine À la fin de notre procédure, il est possible que la racine soit rouge. Pour maintenir la correspondance avec les arbres 2-3, et parce que c'est toujours possible, on la colorie en noir.

Exercice 11.29

p. 217

Écrire la fonction d'insertion pour un arbre rouge-noir.

```
void insert(rb *t, datatype k);
```

4.2.g Correction de la procédure d'insertion

Si l'on n'est pas entièrement convaincu par la correspondance avec les arbres 2-3, on peut faire une preuve par induction structurelle.

- La propriété d'équilibre noir est systématiquement conservée : les nœuds créés sont toujours rouges (initialement), et les transformations décrites n'introduisent aucune violation de la propriété (il faut le vérifier soigneusement, bien évidemment).
- La propriété d'ordre est également conservée : on ne fait que des rotations (dont on a vu qu'elles préservent l'ordre) et des échanges de couleurs (qui n'ont aucun effet sur l'ordre).
- L'insertion dans un sous-arbre dont la racine est noire produit un arbre rouge-noir correct (mais avec une racine potentiellement rouge).
- L'insertion dans un sous-arbre dont la racine est rouge produit un arbre rouge-noir « presque correct » : le seul problème potentiel est que la racine peut être rouge et posséder un fils gauche rouge.

Les deux derniers points ne sont nullement évidents, et doivent être prouvés.

Cas de base Si on insère dans un arbre vide, on obtient un arbre réduit à un nœud rouge avec deux fils vides, c'est bien un ABR correct.

Insertion dans un nœud n rouge

- Si l'étiquette à insérer est celle du nœud, il ne se passe rien et il n'y a rien à prouver.
- Si elle est strictement plus petite, on insère à gauche. Comme le nœud était rouge, son fils gauche était noir. Le résultat de l'appel récursif est donc un arbre rouge-noir correct, mais sa racine peut être rouge. On n'effectue aucune transformation, et l'on renvoie un arbre ayant éventuellement un problème de type « racine rouge, fils gauche rouge ».
- Si elle est strictement plus grande, on insère à droite. Le fils droit était forcément noir (l'arbre de départ était correct), donc on obtient à nouveau un arbre rouge-noir correct, avec éventuellement une racine rouge. Dans ce cas, on arrive dans le cas de la figure 11.38 (puisque n était forcément un fils gauche) et après rotation on renvoie un arbre rouge-noir avec éventuellement le même problème qu'au cas précédent.

Insertion dans un nœud n noir Si l'insertion s'arrête là (parce que la clé du nœud est celle à insérer), il n'y a rien à prouver. Sinon :

- si l'on insère à gauche, l'appel récursif nous renvoie un arbre rouge-noir ayant éventuellement un problème rouge-rouge à la racine (puisque le fils gauche pouvait être rouge). Dans ce cas, on effectue la rotation puis l'échange de couleurs des figures 11.39 et 11.38 et l'on renvoie un arbre rouge-noir correct.
- si l'on insère à droite, l'appel récursif renvoie un arbre rouge-noir correct ayant éventuellement une racine rouge. On arrive dans ce cas à la figure 11.40, on échange les couleurs et l'on renvoie un arbre rouge-noir correct.

L'appel initial est fait sur la racine « globale » de l'arbre, qui est nécessairement noire. Il renvoie donc un arbre rouge-noir correct, avec une racine potentiellement rouge. Dans ce cas, on la colorie en noir, et l'arbre final est donc bien un arbre rouge-noir correct.

4.2.h Tri

Une structure d'arbre binaire de recherche fournit naturellement un algorithme de tri :

- on insère un par un les éléments à trier dans un arbre initialement vide ;
- on effectue un parcours infixé de l'arbre obtenu pour obtenir les éléments dans l'ordre croissant.

En général, on souhaite lorsqu'on l'on trie conserver la multiplicité des éléments (si un élément apparaissait k fois dans la liste initiale, il doit apparaître k fois dans la liste triée). Pour assurer cette propriété, il est nécessaire de modifier légèrement notre définition d'un ABR (ou d'un arbre rouge-noir) en rendant les inégalités sur les clés larges (ou au moins l'une d'entre elles), mais cela ne pose aucun problème et ne modifie en rien l'algorithme d'insertion ni la preuve d'équilibre. Dans le cas d'un arbre rouge-noir, on obtient alors :

Phase d'insertion : on effectue n insertions, dans un arbre de taille 0, puis 1, ..., puis $n - 1$. Une insertion dans un arbre de taille k se fait en temps $O(\log k)$ et donc $O(\log n)$, et le coût total est donc $O(n \log n)$.

Parcours : le parcours de l'arbre et la création de la liste triée peut se faire en temps $O(n)$ (peu importe alors que l'arbre soit équilibré, cela resterait vrai pour un ABR quelconque).

Au total, on obtient donc une complexité en temps en $O(n \log n)$ dans le pire cas, avec une consommation d'espace en $O(n)$ (pour l'arbre).

Exercice 11.30

p. 218

Supposons que l'on utilise un ABR « basique » à la place d'un arbre rouge-noir pour implémenter l'algorithme de tri décrit ci-dessus. Déterminer la complexité obtenue dans le pire cas (on donnera un exemple de liste donnant lieu à ce cas).

Remarque

La complexité *en moyenne* est en $O(n \log n)$ même pour un ABR « basique », mais cela n'a rien d'évident. Nous le démontrerons ultérieurement, et nous mettrons en évidence le parallèle avec l'algorithme de tri rapide.

4.2.i Suppression

On peut implémenter une procédure de suppression dans un arbre rouge-noir, en temps proportionnel à la hauteur de l'arbre (et donc au logarithme de sa taille), et préservant les invariants de la structure. C'est un peu plus compliqué que pour l'insertion : nous ne le ferons qu'en OCaml, pour une variante immuable des arbres rouge-noir.

5 Tables de hachage

Les tables de hachage fournissent une réalisation *fondamentellement impérative* des structures abstraites de dictionnaire et d'ensemble. Pour des raisons partiellement mathématiques et partiellement liées aux spécificités des machines actuelles, elles sont souvent plus rapides que les variantes d'arbres binaires de recherche, mais un peu moins flexibles.

5.1 Un cas simple

Supposons que l'on souhaite stocker un ensemble d'associations (*clé, valeur*), et supposons de plus que l'on sache que :

- il y aura environ 5 000 clés ;
- toutes les clés seront des entiers entre 0 et 9 999.

Dans ce cas, la solution la plus efficace (et de loin !) est d'utiliser un tableau de longueur 10 000, et de stocker l'association (k, v), si elle existe, dans la case d'indice k du tableau. En OCaml, on pourrait par exemple prendre le type suivant :

```
type 'v table = 'v option array
```

Les fonctions `get`, `set` et `remove` sont alors extrêmement simples, et s'exécutent, de manière immédiate, en temps constant :

```
let get table k = table.(k)

let set table k v = table.(k) <- Some v

let rem table k = table.(k) <- None
```

Exercice II.31

p. 218

1. Pourquoi l'information « il y aura environ 5 000 clés » est-elle importante ?
2. Supposons que l'on souhaite en fait réaliser une structure d'ensemble, avec les mêmes informations sur les clés. Quelle structure serait appropriée ?

5.2 Principe

Le principe d'une table de hachage est de conserver une solution proche de celle décrite ci-dessus tout en s'affranchissant des contraintes sur les clés, qui sont désormais librement choisies dans un ensemble K . On va donc fixer une valeur N pour la taille de la table (la longueur du tableau sous-jacent) et ramener les clés dans l'ensemble $[0 \dots N - 1]$. Typiquement, il y aura deux étapes pour cela :

- on se dote d'une fonction $h : K \rightarrow \mathbb{N}$, dite *fonction de hachage* ;
- pour obtenir un indice dans le tableau à partir d'une clé k , on calcule $h_N(k) \stackrel{\text{déf.}}{=} h(k) \pmod{N}$.

Remarques

- Quand on parle de fonction de K dans \mathbb{N} , il faut comprendre que l'on souhaite en réalité obtenir un entier de taille borné : typiquement, un `uint32_t`, un `uint64_t` ou un `int` OCaml positif.
- L'entier $h(k)$ est appelé *empreinte* de la clé k (ou, en bon français informatique, *hash* de k).
- Dans toute la suite, la notation $a \pmod{b}$ désigne le reste de la division euclidienne de a par b .

Exemple II.32

Nous verrons plus loin les propriétés que l'on recherche pour une « bonne » fonction de hachage. Cependant, nous pouvons dès à présent donner quelques exemples de fonctions de hachage *possibles* (sans préjuger de leur qualité) :

- si $K = \mathbb{N}$ (ou $K = \mathbb{Z}$), on peut tout simplement prendre l'identité pour h , et l'on a alors

$$h_N(k) = k \pmod{N};$$

- si K est l'ensemble des chaînes de caractères, on peut voir chaque octet de la chaîne comme un entier (entre 0 et 255) et définir $h(k)$ comme la somme de ces entiers ;
- si K est l'ensemble des nombres flottants (disons sur 64 bits), on peut juste interpréter la représentation binaire de k comme un entier et se ramener au premier cas.

L'idée est alors de créer un tableau un tableau de taille N et de stocker l'association (k, v) dans la case numéro $h_N(k)$ de ce tableau. Si i est l'indice d'une case du tableau, on aura alors trois cas :

- aucune association ne vérifie $h_N(k) = i$: la case du tableau est « vide » ;
- une seule association vérifie $h_N(k) = i$: la case du tableau contient cette association ;
- on a une **collision** : plusieurs associations vérifient $h_N(k) = i$. On a alors un problème : les différents types de tables de hachage correspondent aux différentes manières de le régler.

L'idée fondamentale est que la recherche d'une association devient beaucoup plus rapide : si l'on cherche une clé k , il est inutile de parcourir tout le tableau. On calcule $h_N(k)$ et l'on regarde dans la case correspondante du tableau : si la clé n'y apparaît pas, c'est qu'elle n'apparaît nulle part dans le tableau.

Remarque

Il est important de bien comprendre que l'endroit où l'on stocke une association (k, v) ne dépend **que de la clé k** et pas du tout de la valeur v .

5.3 Résolution par chaînage

Une idée simple et couramment utilisée pour gérer les collisions est celle du *chaînage*. Au lieu d'utiliser un tableau de couples (k, v) , on utilise un **tableau de listes de couples** (k, v) . La case i du tableau contiendra la liste (éventuellement vide) des associations (k, v) vérifiant $h_N(k) = i$: en anglais, on appelle cette liste un *bucket* (*seau*, littéralement, mais on parle plutôt d'*alvéole* en français).

On pourrait donc imaginer le type suivant en OCaml :

```
type ('k, 'v) dict =
  {hash : 'k -> int;
   table : ('k * 'v) list array}
```

- Le champ `hash` contient la fonction de hachage h .
- Le champ `table` contient le tableau à proprement parler.

À l'intérieur d'une alvéole, les opérations de recherche, ajout, remplacement se font comme dans la réalisation naïve de l'exercice 11.2. Globalement, on a donc deux étapes pour chaque opération :

- quand on veut ajouter un couple (k, v) , on calcule $h_N(k)$ et l'on ajoute (k, v) à la liste se trouvant dans la case $h_N(k)$ du tableau (ou l'on remplace la valeur associée à k , s'il y en avait déjà une) ;
- quand on cherche la valeur associée à une clé k , on calcule $h_N(k)$ et on la cherche dans la liste contenue dans la case $h_N(k)$ du tableau (et uniquement dans cette liste).

Exercice 11.33

p. 218

1. Écrire trois fonctions `get_list`, `set_list` et `remove_list` agissant sur une liste d'associations. On pourra supposer que la liste contient au plus une association pour une clé donnée (et on maintiendra cet invariant).

```
get_list : ('k * 'v) list -> 'k -> 'v option
set_list : ('k * 'v) list -> 'k -> 'v -> ('k * 'v) list
remove_list : ('k * 'v) list -> 'k -> ('k * 'v) list
```

2. Écrire les fonctions `get`, `set` et `remove` agissant sur un dictionnaire.

```

get : ('k * 'v) dict -> 'k -> 'v option
set : ('k * 'v) dict -> 'k -> 'v -> unit
remove : ('k * 'v) dict -> 'k -> unit

```

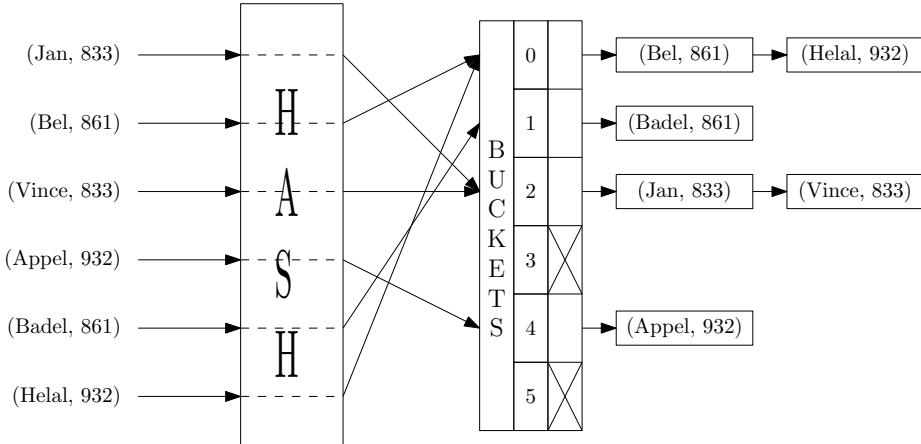


FIGURE 11.41 – Principe d'une table de hachage (chaînée). On a ici $h_5("Bel") = h_5("Helal") = 0$.

En suivant cette méthode, la complexité de la recherche, de l'ajout ou de la suppression d'une clé k ne dépend plus du nombre total n d'associations du dictionnaire, mais seulement du nombre de clés en collision avec k . Plus précisément, on a :

- une évaluation de $h_N(k)$, dont le coût peut dépendre de k (si notre clé est par exemple une chaîne de caractères arbitrairement longue), mais pas de n ;
- un nombre de comparaisons entre clés égal à la longueur de la chaîne présente dans l'alvéole.

Idéalement, on souhaiterait répartir équitablement les valeurs de h_N , pour qu'il y ait (environ) le même nombre d'associations dans chaque alvéole.

5.4 Critères pour une bonne fonction de hachage

Une fonction de hachage « idéale » ne provoquerait jamais de collisions : c'est bien évidemment impossible, puisque l'ensemble K des clés possibles est en général beaucoup plus grand que $[0 \dots N - 1]$ (voire infini). En étant un peu plus raisonnable, on souhaiterait que h_N se comporte comme un tirage aléatoire uniforme dans l'ensemble $[0 \dots N - 1]$. Il n'est pas très difficile de construire des générateurs de nombre pseudo-aléatoires approchant raisonnablement bien cette propriété, mais il y a une difficulté supplémentaire : il est indispensable que deux appels $h_N(k)$ avec le même k renvoient la même valeur ! Si l'on tire au hasard une empreinte à chaque fois, ce ne sera bien sûr pas le cas.

Quand on analysera le coût des opérations sur les tables de hachage, on supposera toujours que la fonction de hachage répartit les clés uniformément et indépendamment dans l'ensemble $[0 \dots n - 1]$.

Remarque

Autrement dit, on supposera :

- pour tout $0 \leq i < N$, pour une clé k choisie aléatoirement, $\mathbb{P}(h_N(k) = i) = \frac{1}{N}$;
- si k et k' sont choisies aléatoirement, alors $\mathbb{P}(h_N(k) = h_N(k')) = \frac{1}{N}$.

Ces propriétés ne sont pas réellement vérifiées par les fonctions de hachage utilisées en pratique (elles n'ont même pas forcément de sens), et de plus les clés n'ont aucune raison d'être choisies au hasard (ce qui invalidera la plus grande partie de notre analyse). Cependant :

- il est possible de rendre cela rigoureux (en tirant la fonction de hachage au hasard dans une famille de fonctions bien choisies) ;
- les performances pratiques sont conformes à l'analyse simplifiée, pour peu que la fonction de hachage soit raisonnablement bien choisie.

Exemple II.34**Remarque**

On peut montrer que si l'on répartit aléatoirement n boules dans n urnes, le nombre de boules dans l'urne la plus « chargée » sera en moyenne $\frac{\ln n}{\ln(\ln n)}$ (environ). Cela signifie que dans une table de hachage chaînée de taille n avec un facteur de charge de 1, le *pire cas* pour la recherche d'une clé sera en $O\left(\frac{\ln n}{\ln(\ln n)}\right)$ (en supposant que la hachage se fasse uniformément).

Une fonction de hachage f est une fonction définie sur un ensemble E de cardinal très grand (voire infini) et à valeurs dans un ensemble F de taille fixée (grande, mais moins...). Un exemple typique : à une chaîne de caractère de taille quelconque, on associe un entier machine sur 32 bits. On appellera $f(x)$ l'*empreinte* de x (ou en bon français informatique le *hash* de x).

La propriété recherchée pour ce type de fonctions est la *pseudo-injectivité* : on souhaite que tout se passe comme si la valeur de $f(x)$ avait été tirée au hasard, uniformément, dans l'ensemble F (sauf bien sûr que l'on veut obtenir le même résultat si l'on calcule plusieurs fois $f(x)$ pour le même x). Ainsi, on pourra faire comme si $P(f(x) = f(y))$ valait $\frac{1}{\text{Card } F}$ dès que $x \neq y$.

Aucune fonction n'a réellement cette propriété (en fait, cette propriété n'a pas vraiment de sens mathématiquement). Cependant, il est possible de formaliser plus précisément ce qui nous intéresse réellement dans la pseudo-injectivité, et de trouver des fonctions convenables. C'est cependant difficile : on utilisera donc `Hashtbl.hash : 'a -> int`, qui à un objet quelconque associe un entier positif ou nul de manière « pseudo-uniforme ».

5.5 Open addressing

Exercices

Type abstrait **MULTISET**

Dans un *multi-ensemble*, l'ordre des éléments n'a pas d'importance mais la multiplicité compte. En anglais, on parle de *bag* (ou de *multiset*) ; en mathématiques, on rencontre parfois l'expression *combinaison avec répétitions*. On peut formaliser un multi-ensemble dont les éléments sont choisis dans un ensemble A par une application $\text{occ}_s : A \rightarrow \mathbb{N}$ qui associe à chaque $x \in A$ son nombre d'occurrences dans s .

Exercice II.35 – Signature pour **MULTISET**

p. 219

Proposer une signature pour le type abstrait **MULTISET** ainsi qu'une spécification formelle des fonctions de la signature (en utilisant la fonction occ_s).

Exercice II.36 – Utilisation d'un dictionnaire

p. 219

On suppose que l'on dispose d'une réalisation (fonctionnelle) du type abstrait **Dict**, satisfaisant la signature de la figure 11.2. Expliquer comment réaliser la structure abstraite de multi-ensemble à l'aide de cette structure de dictionnaire.

Exercice II.37 – Élément le plus fréquent

p. 220

On suppose que l'on dispose d'un type `'a multi` doté des opérations spécifiées à l'exercice 11.35. Écrire une fonction `plus_frequent` : `'a list -> 'a * int` telle que l'appel `plus_frequent u` renvoie l'élément ayant le plus d'occurrences dans `u`, ainsi que ce nombre d'occurrences. Le comportement de la fonction n'est pas spécifié si la liste `u` est vide.

Solutions

Correction de l'exercice II.1 page 180

```
let compte_distincts u =
  (* s est l'ensemble des éléments déjà vus, n son cardinal *)
  let rec aux u s n =
    match u with
    | [] -> n
    | x :: xs -> if member x s then aux xs s n
                   else aux xs (add x s) (n + 1) in
  aux u empty_set 0
```

Correction de l'exercice II.2 page 181

```
let rec get k = function
| [] -> None
| (k', v) :: xs -> if k' = k then Some v else get k xs

let rec set k v = function
| [] -> [(k, v)]
| (k', _) :: xs when k' = k -> (k, v) :: xs
| (k', v') :: xs (* k' <> k *) -> (k', v') :: set k v xs
```

Dans le pire cas, on parcourt toute la liste, ce qui donne donc une complexité en $O(n)$, où n est le nombre d'associations. On supposé implicitement qu'un test d'égalité se fait en temps constant : si les clés sont des objets complexes, ce n'est pas forcément vrai.

Correction de l'exercice II.3 page 181

```
let rec of_list = function
| [] -> empty_map
| (k, v) :: tl -> set k v (of_list tl)

let rec to_list map =
  let u = ref [] in
  iter (fun (k, v) -> u := (k, v) :: !u) map;
  !u
```

Correction de l'exercice II.4 page 181

```
let antecedents t =
  let rec loop i dict =
    if i = Array.length t then dict
    else match get t.(i) dict with
      | None -> loop (i + 1) (set t.(i) i dict)
      | Some u -> loop (i + 1) (set t.(i) (i :: u) dict) in
  loop 0 empty_map
```

Correction de l'exercice II.9 page 185

```
void bst_free(bst* t){
    if (t == NULL) return;
    bst_free(t->left);
    bst_free(t->right);
    free(t);
}
```

Correction de l'exercice II.10 page 185

1.

```
let rec appartient arbre x =
    match arbre with
    | V -> false
    | N (_, y, _) when x = y -> true
    | N (g, y, d) -> if x < y then appartient g x else appartient d x
```

2. On montre par induction structurelle sur l'arbre que $x \in \varphi(\text{arbre})$ si et seulement si appartient arbre x renvoie **true**. Les deux premiers cas ne posent pas de problème (et ne font pas intervenir l'hypothèse d'induction). Dans le troisième cas, on a $x \neq y$ (sinon on serait dans le deuxième cas), donc $x \in \varphi(\text{arbre})$ si et seulement si $x \in \varphi(g)$ ou $x \in \varphi(d)$.

- Si $x < y$, alors $x < \min \varphi(d)$ d'après la propriété d'ABR, donc $x \notin \varphi(d)$. Ainsi, $x \in \varphi(\text{arbre})$ si et seulement si $x \in \varphi(g)$, et donc si et seulement si appartient g x par hypothèse d'induction : c'est correct.
- Le cas $x > y$ est symétrique.

Correction de l'exercice II.11 page 185

```
let rec minimum = function
| V -> failwith "Minimum d'un arbre vide"
| N (V, x, _) -> x
| N (g, _, _) -> minimum g
```

Correction de l'exercice II.12 page 185

Le plus naturel est sans doute d'écrire des versions itératives :

```
bool bst_member(bst *t, item x){
    while (t != NULL) {
        if (t->key == x) {
            return true;
        }
        if (t->key < x) {
            t = t->right;
        } else {
            t = t->left;
        }
    }
    return false;
}
```

```
item bst_minimum(bst *t){
    bst *current = t;
    assert (current != NULL);
    while (current->left != NULL){
        current = current->left;
    }
    return current->key;
}
```

Correction de l'exercice II.13 page 186

```
let rec insere arbre x =
  match arbre with
  | V -> N (V, x, V)
  | N (ga, y, dr) ->
    if x = y then N (ga, y, dr)
    else if x < y then N (insere ga x, y, dr)
    else N (ga, y, insere dr x)
```

Pour permettre d'avoir plusieurs occurrences pour un même élément, il suffit de changer le cas $x = y$ (par exemple en le fusionnant avec le cas $x < y$) :

```
let rec insere arbre x =
  match arbre with
  | V -> N (V, x, V)
  | N (ga, y, dr) ->
    if x <= y then N (insere ga x, y, dr)
    else N (ga, y, insere dr x)
```

Correction de l'exercice II.14 page 186

1. Il n'y a pas de manière très élégante de faire, mais vous pouvez essayer de faire mieux !

```
bst *bst_insert_it(bst *t, item x){
  if (t == NULL) return new_node(x);
  bst *current = t;
  while (true){
    item y = current->key;
    if (x == y) return t;
    if (x < y && current->left == NULL){
      current->left = new_node(x);
      return t;
    } else if (x < y){
      current = current->left;
    } else if (x > y && current->right == NULL){
      current->right = new_node(x);
      return t;
    } else {
      current = current->right;
    }
  }
}
```

2. Quand on appelle la nouvelle fonction sur un arbre non vide, la valeur de retour est inutile (c'est systématiquement le même pointeur que celui donné en argument). En revanche, quand on l'appelle sur un arbre vide, on passe le pointeur nul en argument et on reçoit un pointeur non nul en sortie : la valeur de retour est donc indispensable.

Correction de l'exercice II.15 page 186

1. La version la plus naturelle :

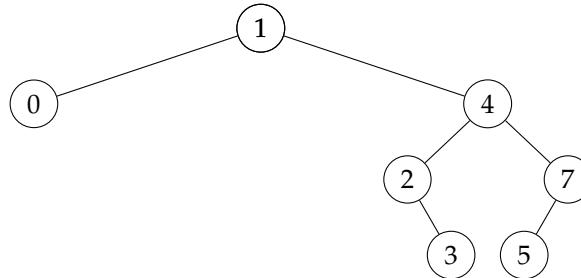
```
let rec construit liste =
  match liste with
  | [] -> V
  | x :: xs -> insere (construit xs) x
```

2. La fonction `construit` n'est pas récursive terminale : après l'appel récursif `construit xs`, il reste à effectuer l'appel à `insere`. Une version récursive terminale :

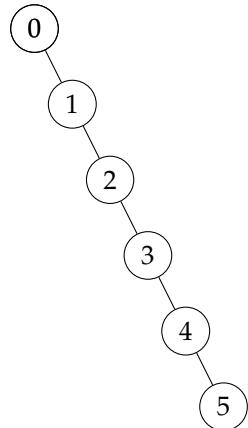
```
let construit_tailrec liste =
  let rec aux restants acc =
    match restants with
    | [] -> acc
    | x :: xs -> aux xs (insere acc x) in
  aux liste V
```

3. Dans la première version (non terminale), les éléments sont insérés dans l'ordre inverse de celui de la liste (le premier élément inséré est le dernier élément de la liste). Dans la version terminale, c'est le contraire : on commence par insérer l'élément de tête.

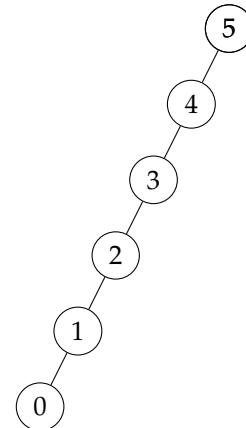
4. Pour [1; 4; 2; 3; 0; 7; 5] :



Pour [0; 1; 2; 3; 4; 5] :



Pour [5; 4; 3; 2; 1; 0] :



Correction de l'exercice II.16 page 187

```
bst *bst_build(item t[], int len){
    bst *tree = NULL;
    for (int i = 0; i < len; i++){
        tree = bst_insert(tree, t[i]);
    }
    return tree;
}
```

Correction de l'exercice II.17 page 188

```
| V -> V
| N (V, x, dr) when x = elt -> dr
| N (ga, x, V) when x = elt -> ga
| N (ga, x, dr) when elt < x -> N (supprime ga elt, x, dr)
| N (ga, x, dr) when elt > x -> N (ga, x, supprime dr elt)
```

Correction de l'exercice II.18 page 189

Il n'y a aucun problème pour l'ensemble des étiquettes (on a bien supprimé x sans toucher au reste). Il faut simplement s'assurer que la propriété d'ordre des ABR reste vérifiée, or :

- c'est le cas à l'intérieur de g et de d (car la propriété était vraie dans l'arbre initial);
- on a $\max g \leq x \leq \min d$ (toujours car l'arbre initial était un ABR), donc en particulier $\max g \leq \min d$, ce qui est exactement ce dont on a besoin.

Si g et d étaient respectivement de hauteur p et q, on passe d'un arbre de hauteur $1 + \max(p, q)$ à un arbre de hauteur $p + q$ (au pire, de $n + 1$ à $2n$). Comme la hauteur de l'arbre est l'élément déterminant pour la rapidité d'exécution des opérations, c'est inacceptable.

Correction de l'exercice II.19 page 189

1. On pourrait renvoyer V dans le cas où l'arbre est vide, mais notre fonction n'est pas censée être appelée sur l'arbre vide : il est sans doute préférable de signaler l'erreur.

```
let rec supprime_min = function
| V -> failwith "minimum d'un arbre vide"
| N (V, x, dr) -> dr
| N (ga, x, dr) -> N (supprime_min ga, x, dr)
```

2.

```
let rec supprime arbre elt =
match arbre with
| V -> V
| N (V, x, dr) when x = elt -> dr
| N (ga, x, V) when x = elt -> ga
| N (ga, x, dr) when elt < x -> N (supprime ga elt, x, dr)
| N (ga, x, dr) when elt > x -> N (ga, x, supprime dr elt)
| N (ga, x, dr) ->
    let m = minimum dr in
    let dr' = supprime_min dr in
    N (ga, m, dr')
```

3. L'élément x sera une feuille dans b , alors que ce n'était pas *a priori* le cas dans a . Par exemple, en partant de $a = N(N(V, 1, V), 2, N(V, 3, V))$, on obtient :

- $\text{supprime } a \ 2 = N(N(V, 1, V), 3, V)$
- $\text{ajoute } (\text{supprime } a \ 2) \ 2 = N(N(V, 1, N(V, 2, V)), 3, V)$

a et b ne sont pas (structurellement) égaux. En reprenant les notations du début du chapitre, on a en revanche bien sûr $\varphi(a) = \varphi(b)$.

Correction de l'exercice II.20 page 189

Ces fonctions sont nettement plus simples à écrire de manière récursive.

1. On choisit de refuser le cas **NULL** plutôt que de renvoyer notre argument inchangé.

```
bst *bst_delete_min(bst *t){
    assert(t != NULL);
    if (t->left == NULL){
        bst *result = t->right;
        free(t);
        return result;
    }
    return bst_delete_min(t->left);
}
```

2. On distingue les 6 mêmes cas qu'en OCaml :

```
bst *bst_delete(bst *t, item x){
    if (t == NULL) {
        return NULL;
    }
    if (t->key < x) {
        t->right = bst_delete(t->right, x);
        return t;
    }
    if (t->key > x) {
        t->left = bst_delete(t->left, x);
        return t;
    }
    // t->key is equal to x
    if (t->left == NULL) {
        bst *result = t->right;
        free(t);
        return result;
    }
    if (t->right == NULL) {
        bst *result = t->left;
        free(t);
        return result;
    }
    item min = bst_min(t->right);
    t->right = bst_delete_min(t->right);
    t->key = min;
    return t;
}
```

Correction de l'exercice II.21 page 191

```

let rec get k dict =
  match dict with
  | Vide -> None
  | Noeud (k', v, g, d) ->
    if k = k' then Some v
    else if k < k' then get k g
    else get k d

let rec set k v dict =
  match dict with
  | Vide -> Noeud (k, v, Vide, Vide)
  | Noeud (k', v', g, d) ->
    if k = k' then Noeud (k, v, g, d)
    else if k < k' then Noeud (k', v', set k v g, d)
    else Noeud (k', v', g, set k v d)

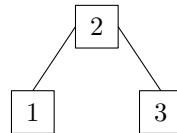
```

Correction de l'exercice II.23 page 192

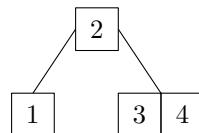
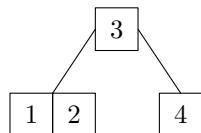
1. Un seul arbre possible :



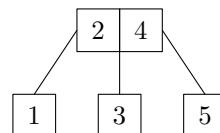
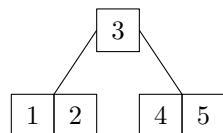
2. À nouveau, un seul arbre possible :



3. Deux arbres possibles :



4. À nouveau, deux arbres possibles :



Correction de l'exercice II.24 page 193

```

let rec appartient x t =
  match t with
  | V -> false
  | B (g, y, d) ->
    (x < y) && appartient x g
    || (x > y) && appartient x d
    || (x = y)
  | T (g, y, m, z, d) ->
    (x < y) && appartient x g
    || (x = y)
    || (x < z) && appartient x m
    || (x = z)
    || appartient x d
  
```

Correction de l'exercice II.25 page 196

- Dans le cas « fils gauche d'un nœud binaire », on a en notant $\text{ord}(t)$ le parcours ordonné (extension du parcours infixe) de t :
 - avant transformation, $\text{ord}(A_1), a, \text{ord}(A_2), b, \text{ord}(A_3), c, \text{ord}(A_4), d, \text{ord}(A_5)$;
 - après transformation, exactement la même chose.

La propriété d'ordre est donc conservée. Pour l'équilibre parfait, on constate que les longueurs des chemins reliant la racine de l'arbre considéré aux nœuds vides sont inchangés, puisque les racines de A_1, A_2, A_3 et A_4 sont à profondeur 2 par rapport à la racine avant et après la transformation, et celle de A_5 de 1, avant et après.

Les cas « fils gauche d'un nœud ternaire » et « fils central d'un nœud ternaire » se traitent de la même manière. Pour le cas « racine », il n'y a pas de problème pour l'ordre mais les profondeurs de tous les nœuds vides sont augmentées de 1. Ce n'est pas un problème précisément parce que la racine du sous-arbre considéré est en fait la racine de l'arbre dans son ensemble : on a augmenté la profondeur de tous les nœuds vides de l'arbre de 1, et on conserve donc l'équilibre parfait.

2.

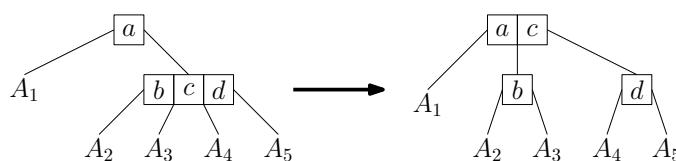


FIGURE 11.42 – Fils droit d'un nœud binaire.

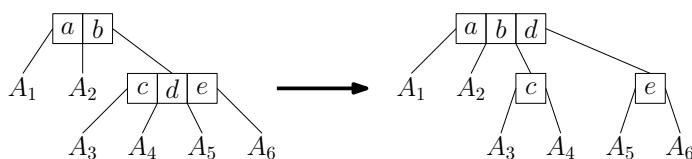


FIGURE 11.43 – Fils droit d'un nœud ternaire.

Correction de l'exercice II.26 page 198

Dans les deux cas, il est plus simple de parcourir deux fois l'arbre : une fois pour vérifier l'ordre, et une fois pour vérifier les contraintes structurelles. Pour vérifier l'ordre, le plus pratique est

en fait de créer un tableau (ou une liste) contenant les étiquettes de l’arbre dans l’ordre infixé, et de vérifier qu’elle est strictement croissante, comme nous l’avons déjà fait. Une autre solution (en C) :

```
bool is_red(node *n){
    return n != NULL && n->red;
}

bool check_node_balance(node *n){
    if (n == NULL) return true;
    if (is_red(n->right)) return false;
    if (is_red(n) && is_red(n->left)) return false;
    return check_node_balance(n->left) && check_node_balance(n->right);
}

bool check_node_order(node *n, datatype *max, bool *first){
    if (n == NULL) return true;
    if (!check_node_order(n->left, max, first)) return false;
    if (*first) {
        *max = n->key;
        *first = false;
    } else if (*max >= n->key) {
        return false;
    }
    *max = n->key;
    return check_node_order(n->right, max, first);
}

bool check_rb(rb *t){
    datatype max = 127;
    bool first = true;
    return !is_red(t->root)
        && check_node_order(t->root, &max, &first)
        && check_node_balance(t->root);
}
```

Correction de l’exercice II.27 page 199

```
node *rotate_right(node *y){
    assert(y != NULL && y->left != NULL);
    node *x = y->left;
    y->left = x->right;
    x->right = y;
    bool y_was_red = y->red;
    y->red = x->red;
    x->red = y_was_red;
    return x;
}
```

```

node *rotate_left(node *x){
    assert(x != NULL && x->right != NULL);
    node *y = x->right;
    x->right = y->left;
    y->left = x;
    bool x_was_red = x->red;
    x->red = y->red;
    y->red = x_was_red;
    return y;
}

```

Correction de l'exercice II.28 page 201

```

node *new_node(datatype k){
    node *n = malloc(sizeof(node));
    n->left = NULL;
    n->right = NULL;
    n->key = k;
    n->red = true;
    return n;
}

bool is_red(node *n){
    return n != NULL && n->red;
}

void flip_colors(node *n){
    assert(n != NULL && !is_red(n) && is_red(n->left) && is_red(n->right));
    n->red = true;
    n->left->red = false;
    n->right->red = false;
}

node *node_insert(node *n, datatype k){
    if (n == NULL) return new_node(k); // new RED node
    if (k == n->key) return n;
    if (k < n->key) n->left = node_insert(n->left, k);
    if (k > n->key) n->right = node_insert(n->right, k);

    if (is_red(n->right) && !is_red(n->left)) n = rotate_left(n);
    if (is_red(n->left) && is_red(n->left->left)) n = rotate_right(n);
    if (is_red(n->left) && is_red(n->right)) flip_colors(n);

    return n;
}

```

Correction de l'exercice II.29 page 201

```

void insert(rb *t, datatype k){
    t->root = node_insert(t->root, k);
    t->root->red = false;
}

```

Correction de l'exercice II.30 page 202

L'insertion dans un ABR se fait en temps $O(h)$, et l'on a bien sûr $h(t) = O(|t|)$. On obtient donc la majoration suivante : $O(\sum_{k=0}^{n-1} k) = O(n^2)$. Si la liste de départ est triée par ordre croissant, l'arbre obtenu après k insertions sera un peigne droit de hauteur $k - 1$, et la $k + 1$ -ème insertion coûtera donc k : c'est bien un exemple de pire cas, le grand- O est dans ce cas un grand- Θ .

Correction de l'exercice II.31 page 203

1. S'il n'y avait que quelques dizaines de clés, l'espace gaspillé nous ferait réfléchir. On pourrait même envisager d'utiliser une liste chaînée de couples (k, v) et de la parcourir pour chaque appel à `get`, `set` et `remove`.
2. Pour un ensemble dont les éléments sont de petits entiers, le mieux est clairement d'utiliser un tableau de booléens. La consommation en espace peut être réduite à 10 000 bits dans le cas qui nous intéresse, et on garde un temps constant pour les trois fonctions.

Remarque

Si l'on utilise un **bool array** (ou un **bool[10000]**), on consommera en fait bien plus de dix mille bits. Cependant, il n'est pas difficile de remédier à ce problème.

Correction de l'exercice II.33 page 204

1. Ces fonctions ont déjà été écrites au début du chapitre :

```
let rec get_list u key =
  match u with
  | [] -> None
  | (k, v) :: xs ->
    if key = k then Some v else get_list xs k

let rec set_list u key value =
  match u with
  | [] -> [(key, value)]
  | (k, v) :: xs ->
    if key = k then (key, value) :: xs
    else (k, v) :: set_list xs key value

let rec remove_list u key =
  match u with
  | [] -> []
  | (k, v) :: xs ->
    if key = k then xs
    else (k, v) :: remove_list xs key
```

2. Pas de difficulté particulière :

```

let get dict key =
  let n = Array.length dict.table in
  let i = (dict.hash key) mod n in
  get_list dict.table.(i) key

let set dict key value =
  let t = dict.table in
  let n = Array.length t in
  let i = dict.hash key mod n in
  t.(i) <- set_list t.(i) key value

let remove dict key =
  let t = dict.table in
  let n = Array.length t in
  let i = dict.hash key mod n in
  t.(i) <- remove_list t.(i) key

```

Correction de l'exercice II.35 page 207

Opération	Type	Commentaire
<i>Caractéristiques</i>		
add	'a -> 'a bag -> 'a bag	Ajout d'une occurrence
remove	'a -> 'a bag -> 'a bag	Suppression d'une occurrence
count	'a -> 'a bag -> int	Nombre d'occurrences
equal	'a bag -> 'a bag	Égalité de multi-ensembles
<i>Complémentaires</i>		
empty_bag	'a bag	Multi-ensemble vide
iter	('a * int -> unit) -> 'a bag -> unit	

On peut spécifier formellement ces fonctions en associant à chaque $s : 'a \text{ bag}$ un multi-ensemble $\varphi(s)$ et en exigeant (pour tous $x, y \in A$) :

- $\text{occ}_{\varphi(\text{empty_bag})}(x) = 0;$
- $\text{occ}_{\varphi(\text{add } x \text{ s})}(x) = \text{occ}_{\varphi(s)}(x) + 1;$
- $\text{occ}_{\varphi(\text{add } x \text{ s})}(y) = \text{occ}_{\varphi(s)}(y)$ si $x \neq y$;
- $\text{occ}_{\varphi(\text{remove } x \text{ s})}(x) = \max(\text{occ}_{\varphi(s)}(x) - 1, 0);$
- $\text{occ}_{\varphi(\text{remove } x \text{ s})}(y) = \text{occ}_{\varphi(s)}(y)$ si $x \neq y$;
- $\text{count } x \text{ s} = \text{occ}_{\varphi(s)}(x);$
- Si $\varphi(s)$ a pour éléments x_1, \dots, x_p avec des nombres d'occurrences n_1, \dots, n_p (tous strictement positifs), alors $\text{iter } f \text{ s}$ a le même effet que $\text{iter } f [(x_1, n_1); \dots; (x_p, n_p)]$ (pour un certain ordre sur les x_i).

Correction de l'exercice II.36 page 207

On utilise une $('a, int) \text{ map}$, et l'on suppose pour éviter les confusions que les fonctions relatives aux dictionnaires ont été placées dans un module **Map** :

```

let add x s =
  match Map.get x s with
  | None -> Map.set x 1 s
  | Some n -> Map.set x (n + 1) s

let remove x s =
  match Map.get x s with
  | None -> s
  | Some 1 -> Map.remove x s
  | Some n -> Map.set x (n - 1) s

let count x s =
  match Map.get x s with
  | None -> 0
  | Some n -> n

let equal s s' =
  Map.to_list s = Map.to_list s'

let iter = Map.iter

```

Correction de l'exercice II.37 page 207

```

let to_list s =
  let u = ref [] in
  iter (fun couple -> u := couple :: !u) s;
  !u

let rec max_list = function
  | [] -> failwith "max d'une liste vide"
  | [(x, n)] -> x
  | (x, n) :: u -> let x', n' = max_list u in
    if n > n' then x else x'

let plus_frequent u =
  let s = List.fold_left (fun acc x -> add x acc) empty_bag u in
  max_list (to_list s)

```

TAS ET FILES DE PRIORITÉ

1 Structure abstraite de file de priorité

Une file de priorité est une structure de données contenant des couples (valeur, priorité) ; les valeurs peuvent être d'un type quelconque, mais les priorités doivent être choisies dans un type totalement ordonné (le plus souvent, ce sont des entiers ou des flottants).

Comme pour une pile ou une file, les deux opérations fondamentales sont l'ajout d'un élément – c'est-à-dire ici d'un couple (valeur, priorité) – et l'extraction du « prochain » élément. Dans une pile, ce « prochain » élément est celui qui a été inséré le plus récemment; dans une file, c'est au contraire le premier à avoir été inséré; dans une *file de priorité*, c'est celui ayant la priorité la plus faible¹.

Un exemple d'application très naturelle des files de priorité est celui du *scheduler* d'un système d'exploitation. Sur un ordinateur actuel, de nombreux *processus* s'exécutent « en parallèle »; certains de ces processus correspondent à des applications lancées par l'utilisateur, d'autres à des services tournant en arrière-plan². En réalité, un processeur (ou un cœur) ne peut exécuter qu'un seul processus à la fois, et le nombre de processus à exécuter sera systématiquement supérieur au nombre de processeurs : il faut donc gérer la pénurie de processeurs. Pour cela, on peut maintenir une file de priorité contenant tous les processus à exécuter. Ensuite, à chaque qu'un processeur se libère :

- on choisit la tâche la plus prioritaire parmi celles à exécuter (c'est-à-dire celle ayant la priorité p la plus basse, si l'on prend la convention usuelle);
- on l'exécute pendant un temps fixé (de l'ordre de la milli-seconde) sur le processeur libre;
- si cette exécution a donné lieu à la création de nouveaux processus, ils sont ajoutés à la file de priorité;
- dans tous les cas, sauf si le processus s'est terminé pendant la fenêtre d'exécution, il est remis dans la file de priorité. Cependant, on l'insère avec une priorité $p' > p$, c'est-à-dire qu'il est à présent moins prioritaire ; la valeur de $p' - p$ dépendra du temps d'exécution dont il vient de bénéficier et du type de processus (on souhaite qu'une application interactive dispose de plus de temps processeur qu'un processus de sauvegarde tournant en arrière-plan).

Opération	Type	Commentaire
<i>Opérations caractéristiques</i>		
insert 'p -> 'v -> ('p, 'v) pq -> unit	Ajout d'un élément (avec sa priorité)	
extract_min ('p, 'v) pq -> 'p * 'v	Extraction de l'élément de priorité minimum (la file est modifiée)	
<i>Opérations complémentaires</i>		
get_min ('p, 'v) pq -> 'p * 'v	Lecture de l'élément de priorité minimum (la file reste inchangée)	
create unit -> ('p, 'v) pq	Création d'une file vide	
card ('p, 'v) pq -> int	Nombre d'éléments dans la file	

FIGURE 12.1 – Signature possible pour un type PRIORITYQUEUE impératif.

1. Ou la plus grande priorité, si l'on choisit l'autre convention.

2. Chaque processus peut posséder plusieurs *fils d'exécution* (ou *threads*), mais cela ne change rien au problème.

Remarques

- Les opérations données correspondent à une *file min* : pour une *file max*, on aura *get_max* et *extract_max*.
 - Les valeurs seront souvent (mais pas toujours) uniques ; en revanche, il sera presque toujours possible d'avoir plusieurs éléments de même priorité.
 - *get_min* peut bien sûr être implémenté à l'aide de *extract_min* et *insert*, mais rarement de manière efficace.
 - Dans certaines applications, il est souhaitable de disposer d'autres opérations :
 - *fusionner* deux files ;
 - *modifier la priorité* d'un élément déjà présent dans la file.
- Certaines réalisations de la structure permettent de réaliser ces opérations de manière efficace, d'autres non.
- La signature donnée est impérative, mais il est tout-à-fait possible de définir des files de priorité fonctionnelles.

Exercice I2.1 – Réalisation par un ABR

p. 230

1. Expliquer comment on pourrait réaliser une file de priorité à l'aide d'un arbre binaire de recherche.
2. Si l'on utilise un arbre rouge-noir, quelle complexité obtient-on pour les opérations *get_min*, *extract_min* et *insert* ?

2 Tas binaire**Définition I2.1 – Arbre binaire complet**

Un *arbre binaire complet gauche* de hauteur h est un arbre binaire qui vérifie les propriétés suivantes :

- tous les niveaux sauf éventuellement le dernier sont complets (autrement dit, il y a exactement 2^i nœuds à profondeur i , pour $0 \leq i < h$) ;
- le dernier niveau est rempli de gauche à droite.

Remarques

- Dans la suite, on écrira parfois simplement « arbre binaire complet » pour « arbre binaire complet gauche ».
- La forme d'un arbre binaire complet est entièrement déterminée par son nombre de nœuds : plus précisément, pour tout $n \in \mathbb{N}$, il y a exactement un arbre binaire complet ayant n nœuds.
- Tous les nœuds internes d'un arbre binaire complet ont un fils gauche ; tous sauf éventuellement un ont un fils droit.
- Les feuilles d'un arbre binaire complet de hauteur h sont toutes à profondeur h ou $h - 1$.

Propriété I2.2

Soit un arbre binaire complet de hauteur h et de taille (nombre de nœuds) $n \geq 1$. On a :

- $h = \lfloor \log_2 n \rfloor$;
- $2^h \leq n < 2^{h+1}$

Démonstration

Notons n_k le nombre de nœuds à profondeur k . Comme l'arbre est complet, on a $n_k = 2^k$ pour $0 \leq k < h$ et $1 \leq n_h \leq 2^h$. Comme $n = \sum_{k=0}^h n_k$, on obtient :

$$\begin{aligned} 1 + \sum_{k=0}^{h-1} 2^k &\leq n \leq \sum_{k=0}^h 2^k \\ 2^h &\leq n \leq 2^{h+1} - 1 \end{aligned}$$

On en déduit alors $h \leq \log_2 n < h + 1$ et donc $h = \lfloor \log_2 n \rfloor$. ■

Remarque

Un arbre binaire complet est donc « parfaitement équilibré » : les opérations nécessitant de parcourir un chemin de la racine à une feuille y sont rapides.

Définition I2.3 – Propriété d'ordre des tas

Soit T un arbre dont les nœuds (internes ou pas) sont étiquetés par des éléments d'un ensemble totalement ordonné (E, \leq) .

- On dit que T a la propriété d'ordre des tas-min si l'étiquette d'un nœud est toujours inférieure ou égale à celles de ses (éventuels) enfants.
- On dit que T est un *tas(-min) binaire* si, de plus, c'est un arbre binaire complet gauche.

Remarques

- On ne sait rien sur l'étiquette du fils gauche par rapport à celle du fils droit (à part qu'elles sont toutes deux plus grandes que celle du père).
- Contrairement aux arbres binaires de recherche, la propriété d'ordre des tas est *purement locale*.
- Si l'étiquette d'un nœud est toujours supérieure ou égale à celle de ses enfants, on parlera de *tas-max*.
- Le terme anglais est (*binary*) *heap* (et donc *min-heap*, *max-heap*).
- Autant la pile d'appel a bien une structure de pile, autant le tas sur lequel on fait des `malloc` n'a absolument rien à voir avec la structure de tas que l'on vient de décrire.

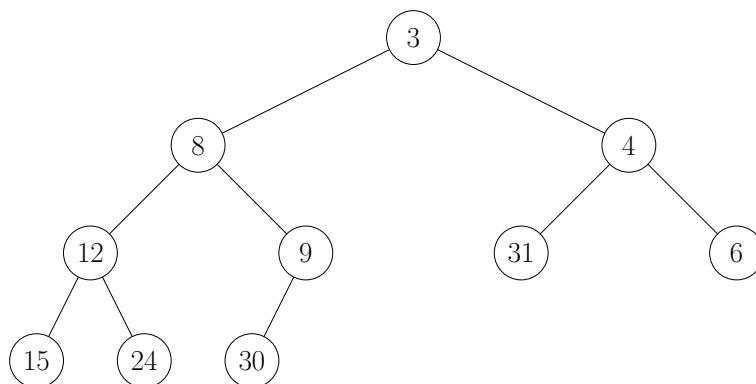
Exemple I2.2

FIGURE 12.2 – Un exemple de tas-min

Propriété I2.4

Quelques propriétés immédiates des tas-min :

- si a est un ancêtre de b , alors l'étiquette de a est inférieure ou égale à celle de b ;
- la racine contient un élément de clé minimale.

Remarque

Pour un ensemble donné d'étiquettes, la forme d'un tas binaire est entièrement déterminée (cette forme ne dépend que du nombre d'étiquettes puisqu'il s'agit d'un arbre complet gauche). En revanche, il y a de nombreuses manières de placer les étiquettes en respectant la propriété d'ordre des tas : les dénombrer est un bon exercice (même s'il ne semble pas y avoir de formule close simple pour le cas général).

3 Tas impératif à l'aide d'un arbre implicite

Le principe général pour réaliser une file de priorité à l'aide d'un tas est de stocker les couples (*priorité, valeur*) dans un arbre en imposant la propriété d'ordre de tas **sur les priorités**. Les valeurs jouent donc un rôle uniquement passif : pour simplifier, on les ignore complètement dans ce qui suit. Bien entendu, quand nous réaliserons cette structure en C ou en OCaml, nous en tiendrons compte.

Théorème 12.5

On peut représenter un arbre binaire **complet gauche** ayant n nœuds dans un tableau de longueur n en énumérant ses étiquettes dans l'ordre du parcours en largeur. Cette représentation est injective : à un tableau donné correspond un seul arbre.

On a les propriétés suivantes :

- la racine est à l'indice 0 ;
- les nœuds de profondeur i sont numérotés consécutivement à partir de l'indice $2^i - 1$;
- les fils gauche et droit du nœud d'indice k sont respectivement les nœuds d'indice $2k + 1$ et $2k + 2$;
- le père d'un nœud d'indice $k > 0$ a pour indice $\lfloor \frac{k-1}{2} \rfloor$.

Démonstration

Notons h la hauteur de l'arbre, n son nombre de nœuds et commençons par montrer par récurrence le deuxième point.

- C'est vrai pour la profondeur 0 car la racine est à l'indice 0 ;
- Supposons que ce soit vrai pour une profondeur $i < h$. Comme $i < h$, le niveau est plein et contient donc 2^i nœuds. D'après l'hypothèse de récurrence, ils sont numérotés de $2^i - 1$ à $2^i - 1 + 2^i - 1 = 2^{i+1} - 2$ et ceux à profondeur $i + 1$ sont donc numérotés à partir de $2^{i+1} - 1$.

On en déduit qu'un nœud a pour indice $2^i - 1 + k$ avec $0 \leq k < 2^i$ si et seulement si il est de profondeur i et possède exactement k nœuds à sa gauche (au niveau i). Le fils gauche d'un tel nœud, en supposant qu'il existe, sera au niveau $i + 1$ et aura exactement $2k$ nœuds à sa gauche (les 2 fils de chacun des k nœuds sus-cités) : il sera donc numéroté $2^{i+1} - 1 + 2k$. Le fils droit sera le nœud suivant, numéroté $2^{i+1} + 2k$. On vérifie alors :

- $2 \cdot (2^i - 1 + k) + 1 = 2^{i+1} - 1 + 2k$, l'indice du fils gauche vérifie la formule ;
- $2 \cdot (2^i - 1 + k) + 2 = 2^{i+1} + 2k$, *idem* pour celui du fils droit.

Pour finir, on vérifie immédiatement que l'on a bien $\lfloor \frac{2k+1-1}{2} \rfloor = \lfloor \frac{2k+2-1}{2} \rfloor = k$, ce qui montre le dernier point. ■

Remarques

- On peut bien sûr stocker le parcours en largeur de n'importe quel arbre binaire dans un tableau, mais si on ne se limite pas aux arbres complets gauches, un même tableau correspondra à plusieurs arbres.
- Un tas binaire étant par définition un arbre binaire complet gauche, il peut être représenté de cette manière.
- Une telle représentation a deux avantages par rapport à la représentation classique d'un arbre :
 - elle est beaucoup plus compact, puisqu'on ne stocke aucun pointeur ;
 - elle permet facilement de passer d'un nœud à son père (ce qui nécessiterait dans la représentation usuelle de rajouter un pointeur parent à la structure, et augmenterait donc encore la consommation mémoire).

Exercice 12.3

p. 230

1. Écrire le tableau obtenu à partir de l'exemple de tas binaire donné plus haut :

- a. en faisant un parcours en largeur ;
- b. en utilisant les propriétés données dans le théorème.

Vérifier que l'on obtient bien la même chose !

2. Dessiner l'arbre correspondant au tableau suivant : [20, 13, 14, 7, 8, 1].

En pratique, les opérations intéressantes sur les tas (insertion d'un élément, extraction du minimum) font varier le nombre d'éléments. La longueur du tableau ne sera donc pas en général égale au nombre d'éléments présents dans le tas, qu'on aura besoin de stocker. En C, on pourrait par exemple utiliser la structure suivante :

```
struct heap {
    int size;
    int capacity;
    datatype *arr;
};

typedef struct heap heap;
```

On remarque que cette structure est exactement celle que nous avons définie pour les tableaux dynamiques. Dans la plupart des applications, il ne sera en fait jamais nécessaire de redimensionner le tableau (parce que l'on disposera d'une borne sur le nombre d'éléments présents, et que l'on pourra directement créer un tableau de taille adéquate); le reste du temps, on utilisera la stratégie habituelle pour le redimensionnement :

- si l'on doit rajouter un élément et que le tableau est « plein » (`size == capacity`), on réalloue un tableau deux fois plus grand;
- si l'on enlève un élément et qu'on obtient `size < capacity / 4`, on réalloue un tableau deux fois plus petit.

En OCaml, une version non redimensionnable utilisera le type suivant :

```
type 'a heap = {mutable size : int; data : 'a array}
```

Pour une version redimensionnable, il suffit de rendre le champ `data` mutable :

```
type 'a heap = {mutable size : int; mutable data : 'a array}
```

Pour simplifier (très légèrement), on utilisera dans le cours une version non redimensionnable, et l'on réservera la version redimensionnable aux TP. On définit quelques fonctions utilitaires :

```
heap *heap_new(int capacity){
    heap *h = malloc(sizeof(heap));
    h->arr = malloc(capacity * sizeof(datatype));
    h->size = 0;
    h->capacity = capacity;
    return h;
}
```

```
void heap_delete(heap *h){
    free(h->arr);
    free(h);
}

void swap(datatype *arr, int i, int j){
    datatype tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
```

```
int up(int i){
    return (i - 1) / 2;
}

int left(int i){
    return 2 * i + 1;
}

int right(int i){
    return 2 * i + 2;
}
```

4 Opérations sur un tas binaire

Le but est ici de réaliser de manière efficace la structure abstraite de file de priorité. **On suppose ici que l'on travaille avec des tas-min.**

4.1 Lecture du minimum

C'est immédiat : le minimum est à la racine.

Propriété I2.6

On peut accéder au minimum d'un tas de taille n en temps $O(1)$.

4.2 Insertion

Pour insérer une nouvelle clé dans un tas, on écrit cette clé dans la première case libre du tableau, on incrémente le cardinal, puis l'on fait une *percolation vers le haut* (ou *sift-up* en anglais) pour rétablir la propriété de tas. Pour faire percoler un élément vers le haut :

- si la clé de l'élément est supérieure ou égale à celle de son père (ou si on est à la racine), on s'arrête ;
- sinon, on échange l'élément avec son père et l'on recommence.

Remarque

Essentiellement, on insère le nouvel élément à la bonne place dans la liste triée correspondant au chemin qui le relie à la racine.

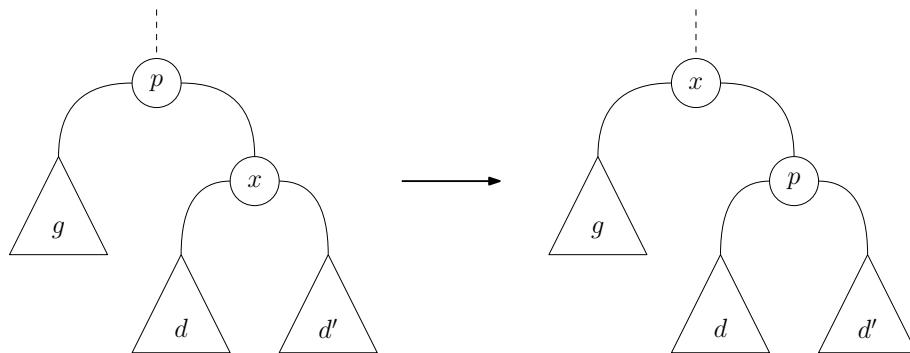
Propriété I2.7

Si T est un tas binaire, alors le résultat T' de l'insertion d'un élément x dans T par le processus décrit ci-dessus est encore un tas binaire (qui contient les éléments de T plus une occurrence supplémentaire de x).

Démonstration

À tout moment, on a au plus une violation de la propriété de tas : entre x et son père p .

- C'est vrai au départ (juste après la création de la feuille x).
- Si x est à la racine ou $x \geq p$, alors il n'y a aucune violation, l'arbre est un tas, on s'arrête.
- Sinon, on a $x < p$ et l'on effectue l'opération suivante :



- il n'y a plus de problème entre x et p ;
- on n'a pas créé de problème entre x et les éléments de g puisque $x < p \leq \min g$;
- on n'a pas créé de problème entre p et d ou d' puisque les éléments de d et d' étaient des descendants de p dans le tas initial.

On a donc supprimé une violation, et on en a créé au plus une : entre x et son éventuel nouveau père. On garde bien l'invariant.

Il reste juste à observer que l'algorithme termine forcément puisque x remonte d'un niveau à chaque étape. ■

Exercice 12.4

p. 230

Simuler les insertions successives de 2, 5 et 1 dans le tas donné en exemple plus haut.

Exercice 12.5

p. 231

1. Programmer en C la fonction `sift_up` qui effectue une percolation vers le haut du nœud dont on fournit l'indice.
2. Écrire la fonction `heap_insert` qui permet d'insérer un élément dans le tas.

```
void sift_up(heap *h, int i);
void heap_insert(heap *h, datatype x);
```

Propriété 12.8 – Complexité de l'insertion

Insérer un élément dans un tas de taille n se fait en temps $O(\log n)$.

Démonstration

Pour le *sift-up*, on a un nombre d'itérations majoré par la hauteur de l'arbre, qui est en $O(\log n)$ car l'arbre est complet, et chaque itération se fait en temps constant. L'ajout initial dans la case vide étant également en temps constant, on a bien du $O(\log n)$ au total. ■

4.3 Extraction du minimum

Pour extraire le minimum d'un tas binaire, on commence par le lire (évidemment!), puis l'on recopie le dernier élément du tableau (la feuille tout en bas et tout à droite de l'arbre) sur la racine. On supprime cette feuille et l'on fait une *percolation vers le bas (sift-down)* de la nouvelle racine. Pour faire percoler un élément vers le bas :

- on le compare à ses deux fils ;
- s'il est inférieur ou égal à ses deux fils, on s'arrête ;
- sinon, on l'échange avec le plus petit de ses deux fils et l'on recommence.

Remarque

Évidemment, on s'arrête aussi si l'élément n'a pas de fils et l'on s'adapte s'il n'en a qu'un seul.

Exercice 12.6

p. 231

Effectuer trois opérations successives d'extraction du minimum sur le tas donné en exemple.

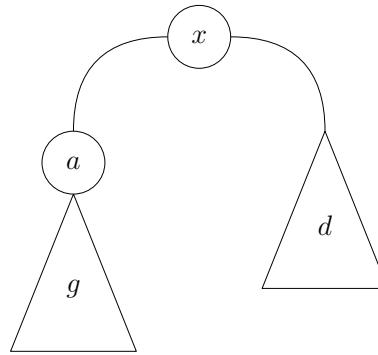
Propriété 12.9

Si T est un tas binaire non vide, alors le résultat T' de l'extraction du minimum de T est encore un tas binaire.

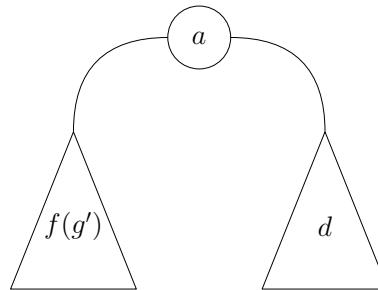
Démonstration

Appelons pseudo-tas un arbre binaire complet dans lequel la propriété de tas est respectée partout sauf éventuellement à la racine. Notons $f(T)$ le résultat de la percolation vers le bas de la racine d'un pseudo-tas T , et montrons par induction structurelle que $f(T)$ est un tas.

- Si T est vide ou réduit à une feuille, c'est évident.
- Sinon, on peut supposer sans perte de généralité que le plus petit fils de la racine se trouve à gauche (d peut éventuellement être vide). Notons que g et d sont des tas.



Si $a \geq x$, alors on s'arrête, $f(\mathcal{T}) = \mathcal{T}$ est bien un tas car $x \leq a \leq \min(d)$ (en convenant que $\min(\emptyset) = +\infty$). Sinon, en notant g' l'arbre g dans lequel on a remplacé la racine par x , on obtient, après l'échange puis le reste de la percolation :



- g' était bien un pseudo-tas (on a juste modifié la racine de g qui était un tas), donc $f(g')$ est un tas par hypothèse d'induction.
- d est toujours un tas.
- a était le plus petit fils de la racine initiale, donc $a \leq \min(d)$.
- Le minimum de $f(g')$ est soit x , soit un élément de g . Mais $a = \min(g)$ et $a < x$, donc $a \leq \min(g')$.

Ainsi, les sous-arbres gauche et droit de $f(\mathcal{T})$ sont des tas et la propriété de tas est vérifiée à la racine : $f(\mathcal{T})$ est bien un tas. ■

Exercice 12.7

p. 232

1. Écrire la fonction `sift_down` qui prend en entrée un pointeur vers un tas et l'indice d'un nœud et effectue une percolation vers le bas de ce nœud.
2. Écrire la fonction `heap_extract_min` qui renvoie le minimum d'un tas et le supprime.

```
void sift_down(heap *h, int i);
datatype heap_extract_min(heap *h);
```

Propriété 12.10 – Complexité de l'extraction du minimum

On peut extraire le minimum d'un tas contenant n éléments en temps $O(\log n)$.

Démonstration

Cette fois, on *descend* le long d'un chemin partant de la racine en faisant toujours un nombre constant d'opération par niveau. Comme la hauteur de l'arbre est en $O(\log n)$, la complexité de *sift-down* (et de l'extraction du minimum) est en $O(\log n)$. ■

4.4 Heapify

Une autre opération possible, très utile en particulier pour le tri par tas, est communément appelée *heapify* (*entassement*, si l'on veut). Elle consiste à prendre un tableau quelconque que l'on interprète comme représentant un arbre binaire complet gauche et à le modifier pour qu'il respecte la propriété de tas.

Exercice 12.8

p. 232

Comment peut-on réaliser cette opération à partir de celles dont nous disposons ? Quelle complexité obtient-on, en fonction de la taille n du tableau ?

On peut faire plus efficace en utilisant l'algorithme suivant, qui prend en entrée un tableau de taille n (éléments numérotés de 0 à $n - 1$) et le modifie en place pour en faire un tas binaire :

Algorithme 2 Algorithme efficace pour *heapify*.

```
fonction HEAPIFY(t)
  pour i =  $\lfloor \frac{n-2}{2} \rfloor$  à 0 faire
    SIFTDOWN(t, i)
```

Exercice 12.9

p. 233

Écrire une fonction *heapify* ayant la spécification suivante :

Entrées : un entier *size* et un tableau *arr* de *size* éléments (de type *datatype*) ;

Sortie : un pointeur *h* vers un tas, ayant *arr* comme tableau de données. Le tableau *arr* aura été modifié de manière à ce que *h* soit effectivement un tas binaire.

On supposera que le tableau *arr* passé en argument est le résultat d'un *malloc*, et que la responsabilité de sa libération est transférée à *h*. Pourquoi est-ce important ?

```
heap *heapify(datatype arr[], int size);
```

Exercice 12.10 – Analyse de *heapify*

p. 233

1. En remplaçant le $i = \lfloor \frac{n-2}{2} \rfloor$ par $i = n - 1$, justifier la correction de l'algorithme.
2. Expliquer pourquoi il est correct de commencer à $i = \lfloor \frac{n-2}{2} \rfloor$.
3. Justifier que la complexité de *HEAPIFY* est en $\Theta(n)$.

5 Tri par tas

À partir d'une réalisation de la structure de file de priorité, il est toujours possible d'obtenir un algorithme de tri :

- on commence par *insérer* les n éléments à trier dans une file initialement vide ;
- on fait ensuite n *extractions du minimum*.

Remarque

Nous verrons plus tard que dans le cadre assez général des *tris par comparaison*, il est impossible d'obtenir un algorithme de tri en temps $O(n \log n)$. On peut donc en déduire qu'aucune réalisation de la structure de file de priorité ne peut proposer *simultanément* des opérations d'insertion et d'extraction du minimum en temps $O(\log n)$. Certaines variantes plus sophistiquées que les tas binaires (tas binomiaux, tas de Fibonacci, ...) peuvent en revanche fournir l'une de ces opérations (l'insertion, typiquement) en temps $O(1)$ et l'autre en $O(\log n)$.

Exercice 12.11 – Tri par tas

p. 233

Écrire une fonction *heapsort* prenant en entrée un tableau *arr* et effectuant un tri en place de ce tableau par ordre décroissant.

```
void HEAPSORT(datatype arr[], int len);
```

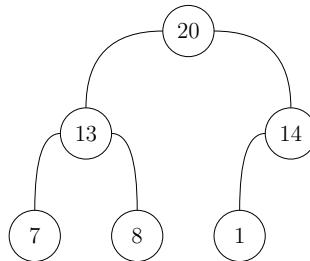
Solutions

Correction de l'exercice 12.1 page 222

1. On utilise les priorités comme clés pour notre arbre comme clés. Il sera nécessaire d'autoriser les clés répétées

Correction de l'exercice 12.3 page 224

1. Dans les deux cas, on obtient $[|3; 8; 4; 12; 9; 31; 6; 15; 24; 30|]$.
2. On obtient l'arbre suivant (qui n'est bien sûr pas un tas) :



Correction de l'exercice 12.4 page 227

On obtient successivement :

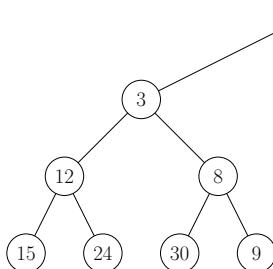


FIGURE 12.9 – Après insertion de 2.

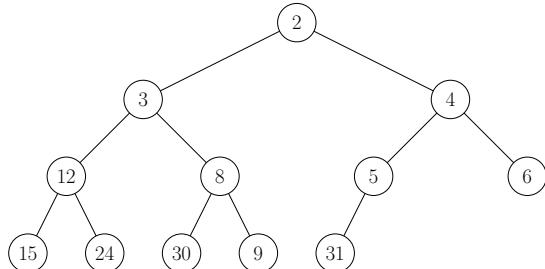


FIGURE 12.10 – Après insertion de 5.

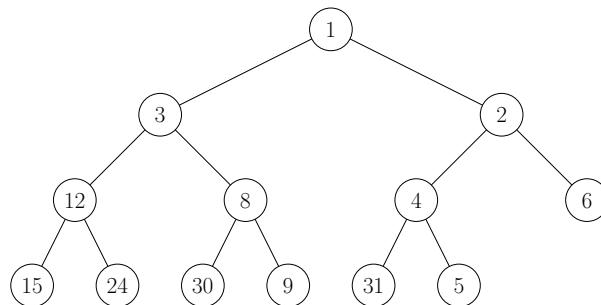


FIGURE 12.11 – Après insertion de 1.

Correction de l'exercice 12.5 page 227

1. La fonction s'écrit très simplement de manière récursive ou itérative :

```
void sift_up(heap *h, int i){
    while (i > 0 && h->arr[i] < h->arr[up(i)]) {
        swap(h->arr, i, up(i));
        i = up(i);
    }
}
```

2. On vérifie que l'insertion est possible, on ajuste la taille, on ajoute une feuille en bas à droite et l'on effectue une percolation vers le haut :

```
void heap_insert(heap *h, datatype x){
    assert(h->size < h->capacity);
    int i = h->size;
    h->arr[i] = x;
    sift_up(h, i);
    h->size += 1;
}
```

Correction de l'exercice 12.6 page 227

On obtient successivement :

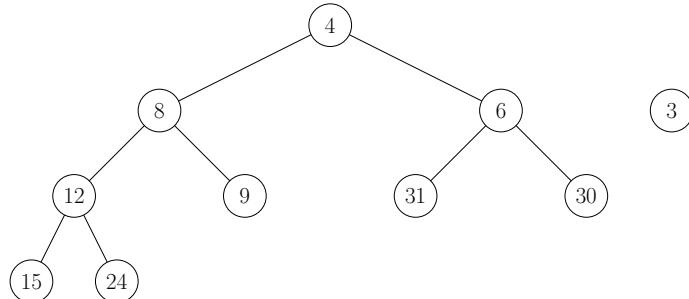


FIGURE 12.12 – On extrait 3.

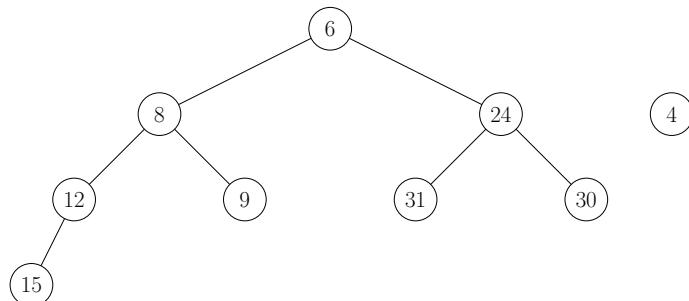


FIGURE 12.13 – On extrait 4.

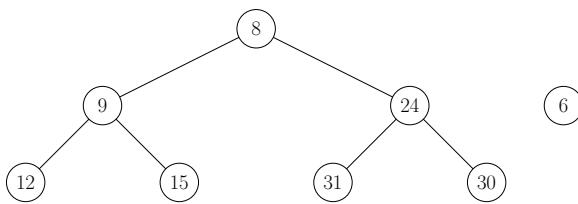


FIGURE 12.14 – On extrait 6.

Correction de l'exercice 12.7 page 228

1. Il est légèrement plus simple d'écrire une version récursive (terminale), et en tout cas il vaut mieux prendre le temps de réfléchir pour arriver à du code simple :

```

void sift_down(heap *h, int i){
    int imin = i;
    if (left(i) < h->size && h->arr[left(i)] < h->arr[imin]) {
        imin = left(i);
    }
    if (right(i) < h->size && h->arr[right(i)] < h->arr[imin]) {
        imin = right(i);
    }
    if (imin != i) {
        swap(h->arr, i, imin);
        sift_down(h, imin);
    }
}
  
```

2. Pas de difficulté, mais il faut penser à décrémenter la taille *avant* d'effectuer la percolation.

```

datatype heap_extract_min(heap *h){
    assert(h->size > 0);
    datatype result = h->arr[0];
    h->size--;
    swap(h->arr, 0, h->size);
    sift_down(h, 0);
    return result;
}
  
```

Correction de l'exercice 12.8 page 229

Le tableau initial (t_0, \dots, t_{n-1}) peut immédiatement être interprété comme un arbre binaire complet gauche. Si l'on fait successivement percoler vers le haut les éléments t_1, \dots, t_{n-1} , on maintient l'invariant suivant : après i étapes, la partie t_0, \dots, t_i du tableau est un tas binaire. Cela revient à partir d'un tas réduit à sa racine et à insérer successivement tous les autres éléments dedans.

Pour la complexité, chaque opération d'insertion prend au pire (s'il faut remonter jusqu'à la racine) un temps proportionnel à $\log i$. Au total, on obtient donc une complexité dans le pire cas proportionnelle à $\sum_{i=1}^n \log i$. Or on obtient successivement (en omettant les parties entières qui ne changent rien) :

$$\begin{aligned} \sum_{i=n/2}^n \log i &\leq \sum_{i=1}^n \log i \leq \sum_{i=1}^n \log n \\ \sum_{i=n/2}^n \log(n/2) &\leq \sum_{i=1}^n \log i \leq n \log n \\ \frac{n}{2}(\log n - 1) &\leq \sum_{i=1}^n \log i \leq n \log n \end{aligned}$$

La complexité est donc en $\Theta(n \log n)$.

Correction de l'exercice 12.9 page 229

Attention, il ne faut pas utiliser la fonction `heap_new`. En effet, elle allouerait un tableau pour `h->arr`, or on veut simplement réutiliser le `arr` fourni.

```
heap *heapify(datatype arr[], int size){
    heap *h = malloc(sizeof(heap));
    h->arr = arr;
    h->size = size;
    h->capacity = size;
    for (int i = (size - 1) / 2; i >= 0; i--){
        sift_down(h, i);
    }
    return h;
}
```

Si l'on libère ultérieurement `h` en appelant `heap_free`, on effectuera un appel `free(h->arr)`, donc :

- il est indispensable que `arr` soit issu d'un `malloc`, sinon l'appel à `free` échouera;
- il ne faut pas libérer ensuite `arr` directement (ce serait un *double free*) ni essayer de s'en servir (ce qui serait un *use after free*).

Une autre possibilité est de libérer `h` sans libérer `h->arr` (en effectuant simplement un appel `free(h)`) : la libération de `arr` n'est alors pas la responsabilité de `h`.

Correction de l'exercice 12.10 page 229

1. On a l'invariant suivant à la fin de chaque itération : les arbres enracinés en t_j avec $j \geq i$ sont des tas. C'est évident au départ, et l'invariant est bien conservé : quand on fait percoler le nœud i vers le bas, ses deux enfants sont des tas (puisque'ils sont situés plus loin dans le tableau), donc lui est un pseudo-tas. Après percolation, il devient donc bien un tas. Une fois que l'on a traité le nœud 0 à la dernière itération, le tableau est un tas binaire.
2. La dernière feuille de l'arbre est en $n - 1$, donc son père est en $\lfloor \frac{n-2}{2} \rfloor$. Comme une feuille est un tas, ce nœud est bien le premier à nécessiter un traitement.
3. Soit h la hauteur de l'arbre (hauteur -1 pour un tas vide). Pour $0 \leq i \leq h$, il y a au plus 2^i nœuds à profondeur i et chacun de ces nœuds nécessite au plus $h - i$ comparaisons lors de sa percolation vers le bas. Au total :

$$\begin{aligned} \sum_{i=0}^h 2^i (h - i) &= \sum_{i=0}^h i 2^{h-i} \\ &= 2^h \sum_{i=0}^h \frac{i}{2^i} \\ &= O(2^h) \quad \text{car la somme converge} \end{aligned}$$

La complexité est donc en $O(2^h) = O(n)$.

Correction de l'exercice 12.11 page 229

- On commence par effectuer un `heapify`.
- Ensuite, on effectue n extractions du minimum, en ignorant les résultats de ces appels. En effet, la fonction `heap_extract_min` a pour effet secondaire de placer le minimum à la dernière position active du tableau (puis de diminuer `size`, et donc de faire sortir ce minimum de la nouvelle portion active).
- On libère le tas `h` sans bien sûr libérer le tableau `arr`!

```
void heap_sort(datatype arr[], int size){  
    heap *h = heapify(arr, size);  
    for (int i = 0; i < size; i++){  
        heap_extract_min(h);  
    }  
    free(h);  
}
```

L'appel à `heapify` se fait en temps $O(n)$, et chacun des n appels à `heap_extract_min` en temps $O(\log n)$. Au total, on obtient du $O(n + n \log n) = O(n \log n)$.

FAMILLES D'ALGORITHMES

I Diviser pour régner

1.1 Principe

Le principe de la méthode *diviser pour régner* (*divide and conquer* en anglais) est fondamentalement récursif :

- on souhaite résoudre une instance de taille n d'un certain problème ;
- si notre instance n'est pas un cas de base, alors :
 - on la sépare en un certain nombre d'instances strictement plus petites du même problème ;
 - on résout récursivement chacune de ces instances ;
 - on reconstruit la solution à notre instance de taille n à partir des solutions aux sous-problèmes.

Si l'on divise le problème en k sous problèmes de taille n/p , et que l'on note $T(n)$ le temps de calcul maximal sur une instance de taille n , on aura :

$$T(n) \leq \underbrace{f(n)}_{\text{ séparation }} + \underbrace{kT(n/p)}_{\text{ appels récursifs }} + \underbrace{g(n)}_{\text{ fusion }}$$

Remarque

Pour être parfaitement rigoureux, il faudrait ici des parties entières supérieures et inférieures. Cela complique énormément l'analyse pour un gain d'information nul (en plus d'alourdir considérablement les notations). Il est possible de s'en débarrasser sans sacrifier la rigueur, et nous les omettrons donc dans la suite.

1.2 Un exemple idiot : recherche du maximum

Considérons le problème suivant : on dispose d'un tableau de n entiers, et l'on souhaite en déterminer le maximum. Il est tout à fait possible d'utiliser une stratégie diviser pour régner :

- les cas de base sont les tableaux de taille 1 (maximum égal à l'unique élément) et de taille 0 (maximum égal à $-\infty$) ;
- si la taille n du tableau est supérieure ou égale à 2, on effectue un appel récursif sur la partie gauche et un sur la partie droite, puis l'on renvoie le maximum des deux résultats.

En notant $f(n)$ le nombre de comparaisons pour un tableau de taille n , on obtient :

$$f(2n) = 1 + 2f(n)$$

Si l'on pose alors $u_k = f(2^k)$, on a

$$\begin{cases} u_0 = 0 \\ u_{k+1} = 1 + 2u_k \end{cases}$$

On en déduit facilement $u_k = 2^k - 1$, c'est-à-dire $f(2^k) = 2^k - 1$. On remarque que ce nombre de comparaisons est exactement le même que celui obtenu par l'algorithme « naturel » (et élémentaire) pour calculer le maximum : on n'a donc rien gagné, et la stratégie diviser pour régner n'a aucun intérêt ici.

Remarques

- La technique consistant à se concentrer sur la complexité pour une instance de taille p^i quand on divise en sous-problèmes de taille n/p sera systématiquement utilisée : elle permet de se débarrasser des parties entières.
- Ici, il n'est pas difficile de montrer que l'algorithme élémentaire est optimal, donc on aurait pu savoir dès le début que notre approche n'avait pas d'intérêt.

1.3 Exemple paradigmatique : le tri fusion

On redonne un code possible pour le tri fusion d'une liste en OCaml :

```
let rec tri_fusion u =
  match u with
  | [] | [_] -> u
  | _ ->
    let a, b = separe u in
    let a_trie, b_trie = tri_fusion a, tri_fusion b in
    fusionne a_trie b_trie
```

Ici, les étapes d'une stratégie diviser pour régner apparaissent clairement, et portent des noms on ne peut plus explicites :

- le premier cas du **match** correspond aux cas de base ;
- dans les autres cas, on commence par diviser le problème en deux sous-problèmes (de taille $n/2$) à l'aide de la fonction **separe** ;
- on effectue les deux appels récursifs pour résoudre les sous-problèmes ;
- on fusionne les résultats à l'aide de la fonction **fusionne**.

Pour étudier la complexité du tri fusion, il est bien sûr nécessaire de connaître celles des fonctions **separe** et **fusionne**. On en rappelle une implémentation possible :

```
let rec separe = function
  | [] -> ([], [])
  | [x] -> ([x], [])
  | x :: y :: xs ->
    let (g, d) = separe xs in
    (x :: g, y :: d)

let rec fusionne u v =
  match u, v with
  | [], _ -> v
  | _, [] -> u
  | x :: xs, y :: ys ->
    if x <= y then x :: fusionne xs v
    else y :: fusionne u ys
```

- La fonction **separe** effectue un parcours de son argument avec un travail constant pour chaque élément, sa complexité est donc en $O(|u|)$, avec u la liste à séparer.
- La fonction **fusionne** parcourt les deux listes qu'on lui passe en argument (elle s'arrête dès qu'elle arrive au bout de l'une des deux), là aussi avec un travail constant pour chaque élément. Sa complexité est donc en $O(|u| + |v|)$.

Pour le tri fusion, en partant d'une liste de taille 2^i , on a donc :

- un appel à **separe** sur une liste de taille 2^i , qui prend un temps majoré par $A2^i$ (où A est une constante) ;
- deux appels récursifs sur des listes de taille 2^{i-1} ;
- un appel à **fusionne** sur deux listes de taille 2^{i-1} , qui prend un temps majoré par $B2^i$.

Au total, on obtient en notant $T(n)$ la complexité dans le pire cas du tri fusion pour une liste de taille n :

$$\begin{aligned} T(2^i) &\leq 2T(2^{i-1}) + A2^i + B2^i \\ &\leq 2T(2^{i-1}) + C2^i \end{aligned} \quad \text{avec } C \text{ une constante}$$

On divise alors par 2^i :

$$\frac{T(2^i)}{2^i} \leq \frac{T(2^{i-1})}{2^{i-1}} + C$$

On fait apparaître une somme télescopique :

$$\frac{T(2^i)}{2^i} - \frac{T(2^{i-1})}{2^{i-1}} \leq C$$

On somme pour $1 \leq i \leq k$:

$$\frac{T(2^k)}{2^k} - \frac{T(2^0)}{2^0} \leq Ck$$

On en déduit :

$$T(2^k) \leq Dk2^k \quad \text{avec } D \text{ une constante}$$

Si $n = 2^k$, on a $k = \log_2 n$ et donc $T(n) \leq Dn \log_2 n$. On voudrait en déduire que $T(n) = O(n \log n)$ dans le cas général, mais on ne peut pas le faire directement puisque le calcul ci-dessus ne concerne que les puissances de deux. En supposant que T est croissante (c'est « essentiellement » vrai, mais nous verrons plus tard comment traiter ce problème rigoureusement), on a :

$$\begin{aligned} T(n) &\leq T(2^{\lceil \log_2 n \rceil}) \\ &\leq D \cdot \lceil \log_2 n \rceil \cdot 2^{\lceil \log_2 n \rceil} \\ &\leq D(1 + \log_2 n)2^{1 + \log_2 n} \\ &= O(n \log n) \end{aligned}$$

Théorème 13.1 – Complexité du tri fusion

En supposant que la comparaison entre deux éléments se fasse en temps constant, le tri fusion a une complexité dans le pire cas en $O(n \log n)$, où n est la longueur de la liste à trier.

Remarque

Nous verrons plus tard que cette complexité est optimale pour un tri par comparaison. Cela ne signifie pas que le tri fusion est « le meilleur tri du monde » (ce n'est pas le cas, généralement), mais simplement qu'un tri par comparaison fait un nombre de comparaison qui est au moins de l'ordre de $n \log n$.

1.4 Tri rapide

Le *tri rapide* (*quicksort* en anglais), introduit par Hoare en 1959, est aujourd'hui l'un des algorithmes les plus utilisés pour trier des tableaux (malgré ses défauts que nous évoquerons ultérieurement).

Table 1

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

FIGURE 13.1 – Comparaison expérimentale entre le tri fusion et le tri rapide, extrait de l'article original.

Son principe est le suivant :

- si le tableau possède 0 ou 1 élément, c'est un cas de base ;
- sinon, on choisit un élément appelé *pivot* ;
- on partage le tableau (ou la liste, mais ce tri n'a de réel intérêt que pour les tableaux) en deux : à gauche les éléments inférieurs au pivot, à droite ceux supérieurs au pivot ;
- on trie récursivement ces deux parties.

Contrairement au tri fusion, l’étape de fusion après le tri des deux parties est triviale : tous les éléments de la « petite » partie sont nécessairement inférieurs à ceux de la « grande » partie.

Il y a trois points à préciser :

- Que fait-on des éléments égaux au pivot ? On peut choisir de les mettre avec les petits ou créer une troisième partie ne contenant que ces éléments (la seule chose qu’on ne peut pas faire, c’est en mettre certains avec les petits et d’autres avec les grands).
- Comment choisit-on le pivot ? On peut prendre systématiquement le premier élément, ou choisir au hasard, ou utiliser une heuristique pour essayer d’obtenir souvent un « bon » pivot (c'est-à-dire un pivot partageant le tableau en deux parties de tailles similaires). Nous y reviendrons, puisque c'est en fait une question importante.
- Le tri se fait-il *en place* ou pas ? Un tri est dit « en place » s'il n'utilise pas de stockage auxiliaire (ou plutôt si la quantité de stockage auxiliaire utilisée est indépendante de la taille du tableau à trier). Il est un peu plus simple d’écrire une version qui n'est pas en place, mais un « vrai » tri rapide se fait toujours en place.

Dans ce qui suit, on utilise la notation $t[i:j]$ (qui n'existe ni en OCaml ni en C) pour désigner la partie du tableau t comprise entre l'indice i inclus et l'indice j exclu.

1.4.a Fonction principale

Pour commencer, on va supposer que l'on dispose d'une fonction `partitionne` ayant la spécification suivante :

- elle prend en entrée un tableau t de taille n et trois entiers deb , fin et i_{piv} vérifiant $0 \leq \text{deb} \leq i_{\text{piv}} < \text{fin} \leq n$;
- elle renvoie un entier k vérifiant $\text{deb} \leq k < \text{fin}$;
- en notant t' l'état du tableau après l'appel (et t l'état avant l'appel), on a :
 - $t'[0:\text{deb}] = t[0:\text{deb}]$ et $t'[\text{fin}:n] = t[\text{fin}:n]$ (la fonction n'a rien modifié en dehors de la zone précisée);
 - $t'[k] = t[i_{\text{piv}}]$ (l'indice renvoyé correspond à la position finale du pivot);
 - tous les éléments de $t'[\text{deb}:k]$ sont inférieurs ou égaux à $t[k]$;
 - tous les éléments de $t'[k+1:\text{fin}]$ sont strictement supérieurs à $t[k]$.

Autrement dit, l'état après l'appel est le suivant :

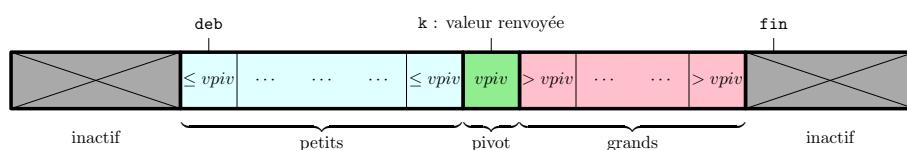


FIGURE 13.2 – État après partitionnement.

Exercice 13.1

p. 261

Écrire en OCaml une fonction `tri_rapide`, utilisant la fonction `partitionne` que l'on supposera déjà écrite, et triant son argument par ordre croissant suivant l'algorithme décrit plus haut. Pour l'instant, on prendra comme pivot le premier élément de la section active.

```
tri_rapide : 'a array -> unit
```

1.4.b Schéma de partitionnement de Lomuto

Il reste à écrire la fonction de partition, dont on souhaite qu'elle s'exécute en temps linéaire (en la taille de la partie à partitionner) et en espace constant. Il y a deux manières classiques de procéder : c'est que nous présentons ici est connue sous le nom de *schéma de partitionnement de Lomuto*. L'autre, connue sous le nom de *schéma de Hoare*, sera vue en travaux pratiques.

- On commence par échanger le pivot avec le dernier élément de la partie active.
- Ensuite, on parcourt cette partie de gauche à droite, en maintenant deux variables i et j et l’invariant suivant :
 - si $x \in t[\text{deb} : i]$, alors $x \leq v_{\text{piv}}$;
 - si $x \in t[i : j]$, alors $x > v_{\text{piv}}$;
 - les éléments de $t[j : \text{fin} - 1]$ n’ont pas encore été étudiés.
- Finalement, on place le pivot au bon endroit et l’on renvoie son nouvel indice.

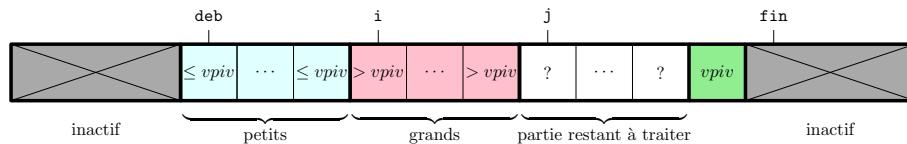


FIGURE 13.3 – Schéma de Lomuto.

Exercice 13.2

p. 261

On dispose de la fonction suivante :

```
let echange t i j =
let x = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- x
```

Écrire une fonction `partitionne`, fonctionnant suivant le schéma de Lomuto et ayant la spécification donnée plus haut, et justifier sa correction.

```
partitionne : 'a array -> int -> int -> int -> int
```

I.4.c Complexité du tri rapide

Si t est un tableau de taille n , le nombre de comparaisons effectuées pour trier n est

$$C(t) = C(t[0 : k]) + C(t[k + 1 : n]) + n - 1$$

avec k la position du pivot après partitionnement.

- Si l’on est capable de garantir à chaque étape que le pivot partage équitablement le tableau, on retrouve essentiellement la relation du tri fusion : $O(n)$ pour la partition/fusion et deux appels sur des tableaux deux fois plus petits. C’est le meilleur cas pour le tri rapide et l’on obtient alors une complexité en $O(n \log n)$.
- Si au contraire le pivot choisi est systématiquement le minimum (ou le maximum) de la partie considérée, la relation devient $C(n) = C(n - 1) + n - 1$. On obtient alors une complexité en $O(n^2)$, ce qui est le pire cas pour le tri rapide.

Théorème 13.2 – Complexité du tri rapide

Le tri rapide a une complexité de $O(n^2)$ dans le pire cas et de $O(n \log n)$ dans le meilleur cas. Si l’on prend comme pivot le premier élément de la partie active, la complexité sera quadratique si l’on appelle la fonction sur un tableau qui est déjà trié.

Remarque

Nous reviendrons sur la complexité du tri rapide ultérieurement. Il y a trois questions importantes :

- peut-on choisir le pivot de manière à garantir un cas favorable ?
- quelle est la complexité *en moyenne* ?
- peut-on obtenir quelque chose qui soit efficace *en pratique* (avec une probabilité très proche de 1) ?

1.5 Algorithme de Karatsuba

On souhaite multiplier deux entiers en base b (fixée), de taille non bornée. On considère que multiplier ou ajouter deux *chiffres* est une opération élémentaire ; en revanche, il est bien évident qu’une opération arithmétique sur deux *nombres* de n chiffres chacun n’en est pas une.

On suppose qu’un entier « long » $x = \sum_{i=0}^{n-1} x_i b^i$ est représenté par un tableau (x_0, \dots, x_{n-1}) d’entiers « machine » et que l’on dispose de trois fonctions :

- $\text{longueur}(x)$ qui renvoie n et s’exécute en temps constant ;
- $\text{chiffre}(x, i)$ renvoie x_i si $0 \leq i < n$, 0 sinon et s’exécute également en temps constant ;
- $\text{normalise}(x)$ qui s’exécute en temps linéaire en n et élimine les « zéros en tête ». Autrement dit, $\text{normalise}((x_0, \dots, x_{n-1}))$ renvoie (x_0, \dots, x_{p-1}) avec p tel que

$$\begin{cases} x_{p-1} \neq 0 \\ x_i = 0 \text{ pour } p \leq i < n \end{cases}$$

Les entiers que nous considérerons seront toujours « normalisés » (chiffre de poids fort non nul), et nous ne renverrons que des entiers normalisés.

1.5.a Addition

On peut commencer par remarquer qu’ajouter deux nombres de n chiffres peut se faire en temps $O(n)$ par l’algorithme élémentaire, et que c’est clairement optimal (on est bien obligé de lire tous les chiffres).

Algorithme 3 Addition d’entiers longs

```

fonction AJOUTE( $x, y$ )
     $n \leftarrow \max(\text{longueur}(x), \text{longueur}(y))$ 
     $z \leftarrow (0, \dots, 0)$  ( $\text{longueur } n + 1$ )
     $\text{retenue} \leftarrow 0$ 
    pour  $i = 0$  à  $n - 1$  faire
         $c \leftarrow \text{chiffre}(x, i) + \text{chiffre}(y, i) + \text{retenue}$ 
         $z_i \leftarrow c \text{ (mod } b)$ 
         $\text{retenue} \leftarrow c/b$  (quotient de la division euclidienne)
     $z_n \leftarrow \text{retenue}$ 
    renvoyer  $\text{normalise}(z)$ 
```

1.5.b Multiplication par une puissance de b

Multiplier un nombre par une puissance de b peut se faire en temps proportionnel à son nombre de chiffres : on a $\text{MulBase}((x_0, \dots, x_{n-1}), k) = (0, \dots, 0, x_0, \dots, x_{n-1})$ (avec k zéros au début).

1.5.c Multiplication d’un nombre par un chiffre

À nouveau, cette opération peut facilement s’exécuter en temps linéaire en le nombre de chiffres. On suppose que l’on dispose d’une fonction $\text{MulChiffre}(x, c)$ qui prend en entrée un nombre x de n chiffres et un chiffre c et renvoie xc , en temps $O(n)$.

1.5.d Multiplication naïve

L’algorithme présenté ci-dessus pour l’addition est l’algorithme « école primaire ». On peut de même formaliser la manière dont on « pose » habituellement une multiplication.

Exercice 13.3

p. 262

1. Écrire en pseudo-code une fonction $\text{MulNaif}(x, y)$ qui calcule le produit xy avec l’algorithme « école primaire ».
2. Déterminer la complexité de cette fonction (on supposera pour simplifier que $\text{longueur}(x) = \text{longueur}(y) = n$).

1.5.e Diviser pour régner : première tentative

On considère deux entiers x et y possédant tous les deux $2n$ chiffres. On peut décomposer ces nombres comme suit :

$$x = x_{lo} + b^n x_{hi} \quad \text{et} \quad y = y_{lo} + b^n y_{hi}$$

On a alors

$$xy = x_{lo}y_{lo} + (x_{lo}y_{hi} + x_{hi}y_{lo})b^n + x_{hi}y_{hi} \cdot b^{2n}.$$

Analysons les opérations à effectuer :

- trois additions dont les opérandes ont au plus $2n$ chiffres, en $O(n)$;
- deux décalages (multiplications par des puissances de la base), encore en $O(n)$;
- quatre multiplications d’entiers possédant chacun n chiffres.

On obtient donc une relation de la forme :

$$T(2n) = 4T(n) + O(n)$$

Malheureusement, ce n’est pas mieux que la version naïve. En effet, on a $T(2n) \geq 4T(n)$, et l’on obtient alors facilement que $T(2^k) \geq A4^k$ (avec $A > 0$ une constante), ce qui est quadratique.

1.5.f L’algorithme de Karatsuba

L’idée de l’algorithme de Karatsuba est d’éliminer l’une des multiplications en la remplaçant par des additions (ou soustractions, ce qui revient au même) supplémentaires. On pose :

$$\begin{cases} u = x_{lo}y_{lo} \\ v = x_{hi}y_{hi} \\ w = (x_{lo} + x_{hi})(y_{lo} + y_{hi}) \end{cases}$$

On a alors :

$$xy = u + (w - u - v)b^n + vb^{2n}.$$

1.5.g Complexité de l’algorithme de Karatsuba

Pour multiplier deux entiers de taille 2^i , on effectue :

- un nombre constant d’opérations en temps linéaire (additions, soustractions, décalages), qui prennent au total un temps $O(2^i)$;
- trois appels récursifs sur des entiers de taille 2^{i-1} .

On a donc :

$$T(2^i) \leq 3T(2^{i-1}) + A2^i$$

En divisant par 3^i :

$$\frac{T(2^i)}{3^i} - \frac{T(2^{i-1})}{3^{i-1}} \leq A(2/3)^i$$

On en déduit en sommant de $i = 1$ à k :

$$\frac{T(2^k)}{3^k} - B \leq A \sum_{i=1}^k (2/3)^i$$

Or la somme de droite est bornée (somme géométrique de raison $2/3$), et l’on obtient donc $T(2^k) \leq C3^k$ en re-multipliant par 3^k .

On observe à présent que $3^k = \exp(k \ln 3) = \exp(k \ln 2 \cdot \frac{\ln 3}{\ln 2}) = (2^k)^{\ln 3 / \ln 2}$.

Ainsi, si $n = 2^k$, on a $T(n) = O(n^{\ln 3 / \ln 2}) = O(n^{1.59})$.

Pour étendre le résultat aux n quelconques, il faut d’abord expliquer comment traiter le cas où le nombre de chiffres est impair (ce qui ne pose pas de problème) puis utiliser la croissance de T :

$$T(n) \leq T(2^{\lceil \log_2 n \rceil}) \leq A \left(2^{\lceil \log_2 n \rceil}\right)^{\log_2 3} \leq A(2n)^{\log_2 3} \leq Bn^{\log_2 3}$$

1.6 Complexité des algorithmes « diviser pour régner »

On rappelle le principe général de la méthode, et l'on fixe les notations pour cette partie :

- si l'instance à traiter est un cas de base, on renvoie ;
- sinon, on la sépare en a instances de taille n/b (en omettant les parties entières), ce qui prend un temps $f_s(n)$;
- on fait les appels récursifs, ce qui prend un temps $aT(n/b)$;
- on « fusionne », c'est-à-dire que l'on calcule le résultat final à partir des résultats des appels récursifs, ce qui prend un temps $f_f(n)$.

Au total, en notant $f = f_s + f_f$, on a la relation suivante :

$$T(n) = aT(n/b) + f(n)$$

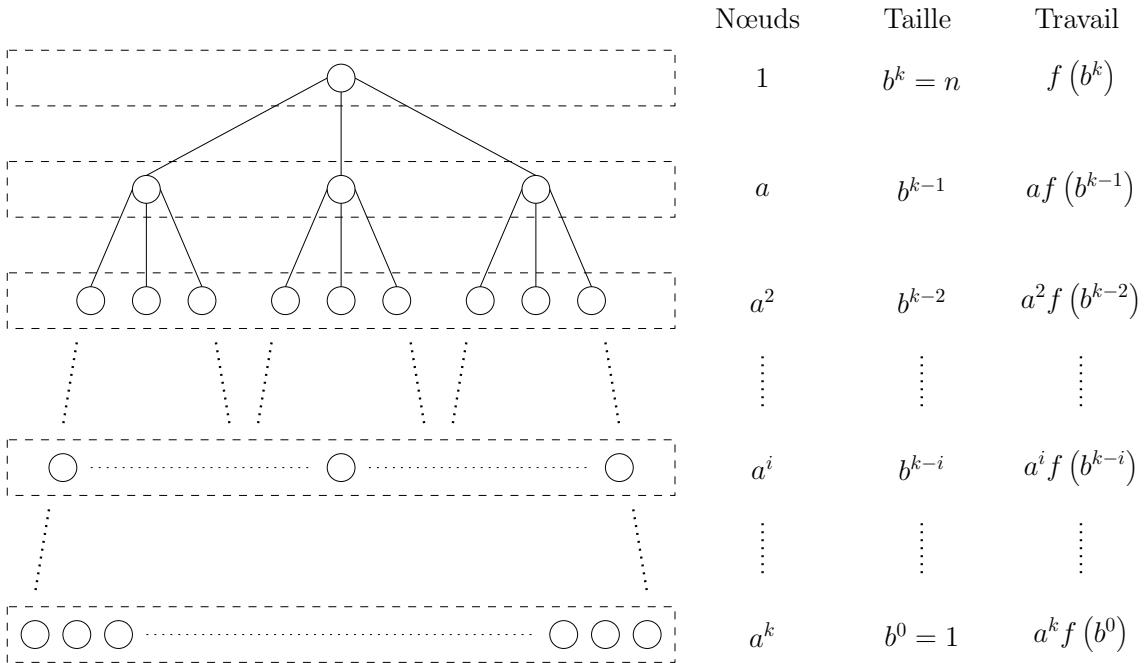
Avant de passer à l'analyse, on peut faire quelques remarques.

- On n'a pas précisé ce que représente $T(n)$, mais il s'agit logiquement de la complexité dans le pire des cas pour une instance de taille n . Comme il n'y a aucune raison que le pire cas pour n donne lieu à des appels récursifs sur des pires cas pour n/b , on aura plutôt $T(n) \leq aT(n/b) + f(n)$. Si l'on souhaite obtenir un O , ce n'est absolument pas un problème.
- Si n n'est pas un multiple b , ce qui précède n'a pas de sens. On fera donc l'analyse pour $n = b^k$ et l'on généralisera ensuite.
- Pour pouvoir faire cette généralisation, on aura besoin que T soit croissante. Ce n'est pas le cas en général (même si ce n'est « pas loin d'être vrai » le plus souvent) : on peut remplacer $T(n)$ par $T'(n) := \max_{k \leq n} T(k)$ et faire disparaître le problème. On aura encore

$$T'(n) \leq aT'(n/b) + f(n)$$

et T' est croissante par construction.

La bonne manière de raisonner ici est d'observer la structure de l'arbre d'appels, et de déterminer la quantité de travail effectuée à chaque niveau de cet arbre :



$$\text{Travail total : } T(b^k) = \sum_{i=0}^k a^i f(b^{k-i})$$

FIGURE 13.4 – Analyse d'une relation $T(n) = aT(n/b) + f(n)$ (le schéma est fait avec $a = 3$).

Si l'on souhaite retrouver ce résultat sans faire de dessin, c'est très simple :

- on pose $u_i = \frac{T(b^i)}{a^i}$;
- on a alors $u_{i+1} = u_i + \frac{f(b^i)}{a^i}$;
- on somme les $u_{i+1} - u_i$ et l'on obtient :

$$u_k - u_0 = \sum_{i=0}^{k-1} \frac{f(b^i)}{a^i}$$

- on multiplie par a^k , on obtient :

$$T(b^k) = a^k T(b^0) + \sum_{i=0}^{k-1} a^{k-i} f(b^i)$$

- en posant $f(1) = T(1)$ (ce qui est parfaitement logique), on obtient exactement :

$$T(b^k) = \sum_{i=0}^k a^{k-i} f(b^i)$$

Ensuite, on remarque que $T(n) \leq T(b^{\lceil \log_b n \rceil})$ pour traiter le cas général (où n n'est pas une puissance de b).

Dans le cas particulier où $f(n) = \Theta(n^p)$ pour une certaine constante p , on dispose d'un théorème classique mais hors-programme et pas vraiment utile à retenir :

Théorème 13.3 – Théorème maître

Si T vérifie $T(n) \leq aT(n/b) + An^p$ (avec A une constante) et si T est croissante, on a :

- si $b^p < a$, alors $T(n) = O(n^{\log_b a})$;
- si $b^p = a$, alors $T(n) = O(n^{\log_b a} \log n)$;
- si $b^p > a$, alors $T(n) = O(n^p)$.

Remarques

- Il faut bien comprendre à quoi correspondent les trois cas :
 - dans le premier cas, le terme en $O(n^p)$ (qui correspond au travail de séparation-fusion) est négligeable devant le coût des appels récursifs. Si l'on considère l'arbre d'appels, la majorité du temps est passé dans les feuilles. Autrement dit, la somme $\sum_{i=0}^k a^i f(b^{k-i})$ est de l'ordre de a^k .
 - dans le troisième cas, c'est le coût des appels récursifs qui est négligeable : la majorité du temps est passé à la racine de l'arbre d'appels, la somme est de l'ordre de $f(b^k)$;
 - dans le deuxième cas, l'étape de séparation-fusion et les appels récursifs ont un coût similaire : autrement dit, le coût total pour traiter chaque niveau de l'arbre d'appels est le même (approximativement) et le coût total s'obtient en multipliant ce coût par niveau par le nombre de niveaux.
- Comme dit plus haut, je vous déconseille de chercher à retenir ce théorème : refaites le raisonnement, en partant de l'arbre d'appels ou directement par le calcul, pour le cas qui vous intéresse.

Exercice 13.4

p. 263

Déterminer par cette méthode la complexité temporelle :

1. de la recherche dichotomique ;
2. du tri fusion ;
3. de l'algorithme de Karatsuba
4. de l'algorithme de Strassen, qui permet de réduire la multiplication de deux matrices $2n \times 2n$ à 7 multiplications de matrices $n \times n$ et un nombre constant d'additions de matrices $n \times n$ ou $2n \times 2n$.

2 Programmation dynamique

2.1 Un exemple simple et classique

Après un changement de l'équipe dirigeante, l'entreprise Mondelez International, qui fabrique les barres Toblerone, décide de rationaliser sa production pour maximiser ses revenus. En effet, leur chaîne de production fabrique des barres de n « carreaux » qui sont ensuite coupées en barres plus petites avant d'être vendues. Mais une récente étude de marché a déterminé le prix auquel on pouvait vendre des barres de longueur k (pour $1 \leq k \leq n$), et ce prix s'avère ne pas avoir de relation simple avec k . Le problème est donc de décider comment découper la barre initiale de n carreaux en des barres plus petites pour maximiser le prix de vente total. Pour simplifier, on néglige le coût de la découpe et de l'emballage.

Dans tout le problème, on considérera que l'on dispose d'un tableau t , indicé de 0 à n , tel que $t.(k)$ soit le prix de vente d'une barre de longueur k , et que le prix d'un morceau de taille 0 est 0 (autrement dit que $t.(0) = 0$). Remarquez que si $t.(n)$ est suffisamment grand, la solution optimale peut très bien être de ne pas découper la barre.



FIGURE 13.5 – Le cœur du problème.

On donne un exemple de tableau t pour $n = 10$.

i	0	1	2	3	4	5	6	7	8	9	10
t_i	0	1	5	8	9	10	17	17	20	24	26

La solution optimale pour cet exemple est de découper la barre en deux blocs de taille 2 et un bloc de taille 6, pour une valeur totale de 27.

Formalisation du problème

On considère donnés un entier $n \geq 0$ et un tableau t d'entiers positifs, de taille $n + 1$, vérifiant $t.(0) = 0$. Dans tout ce qui suit, k désigne un entier de $[0 \dots N]$.

- Une *découpe* de (taille) k est un p -uplet (c_1, \dots, c_p) d'éléments de \mathbb{N}^* vérifiant $\sum_{i=1}^p c_i = k$.
- On note $\mathcal{D}(k)$ l'ensemble des découpes de k .
- La *valeur* d'une découpe $d = (c_1, \dots, c_p)$ est la quantité $v(d) = \sum_{i=1}^p t.(c_i)$.
- La *valeur optimale* d'une découpe de taille k est la quantité $v_{\text{opt}}(k) = \max_{d \in \mathcal{D}(k)} v(d)$.
- Une *découpe optimale* de taille k est un $d \in \mathcal{D}(k)$ vérifiant $v(d) = v_{\text{opt}}(k)$.

Le problème est de calculer $v_{\text{opt}}(n)$ et de fournir un exemple de découpe optimale.

2.1.a Solution en force brute

La solution la plus évidente est d'énumérer toutes les découpes possibles et de déterminer celle ayant la plus grande valeur. On peut noter que deux découpes ne différant que par l'ordre des blocs (3, 3, 2, 2 et 2, 3, 2, 3, par exemple) sont clairement équivalentes : on pourrait donc définir une notion de découpe canonique (morceaux triés par taille croissante, par exemple) et tenter de n'énumérer que ces découpes canoniques. Ce dernier point n'est en fait pas trivial, et l'on va donc se contenter d'énumérer *toutes* les découpes.

Pour cela, on peut établir une bijection entre l’ensemble des tableaux d de $n - 1$ booléens et l’ensemble des découpes de n : on coupe après le $i + 1$ -ème « morceau » si et seulement si d_i est vrai.

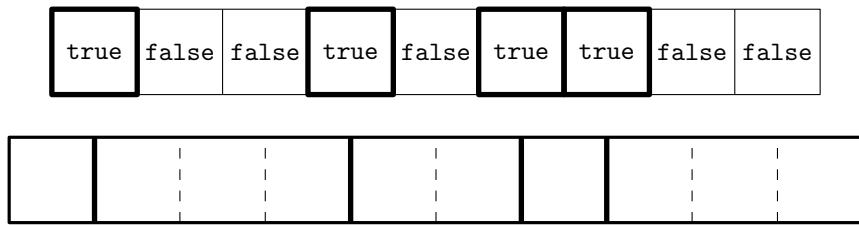


FIGURE 13.6 – La découpe 1, 3, 2, 1, 3 et le tableau de booléens correspondant.

On peut alors :

- générer tous les tableaux (ou toutes les listes) de $n - 1$ booléens ;
- convertir chacun de ces tableaux en une découpe comme décrit ci-dessus ;
- calculer la valeur de chacune de ces découpes ;
- renvoyer la valeur maximale, et un exemple de découpe optimale.

Complexité Il y a 2^{n-1} tableaux à considérer. Les générer prendra un temps proportionnel à 2^n ou $n2^n$ suivant la méthode choisie, et les traiter un temps proportionnel à $n2^n$ (il faut tous les parcourir). Au total, on a donc une complexité en $O(n2^n)$.

2.1.b Sous-structure optimale : solution récursive

Pour arriver à une solution efficace, la première étape est de remarquer qu’une solution du problème (*i.e.* une découpe optimale d’une barre de taille n) « contient » des solutions à des versions plus petites du même problème. Pour simplifier, on peut supposer (sans perte de généralité), qu’à chaque étape on découpe ce qui reste de la barre en un bloc de taille k ($1 \leq k \leq n$) qui fera partie de la découpe finale et une nouvelle barre qui pourra à nouveau être découpée. Il est alors clair que cette nouvelle barre (de longueur $n - k$) devra elle aussi être découpée de manière optimale. On a donc :

$$v(n) = \max_{1 \leq k \leq n} (t_k + v(n - k))$$

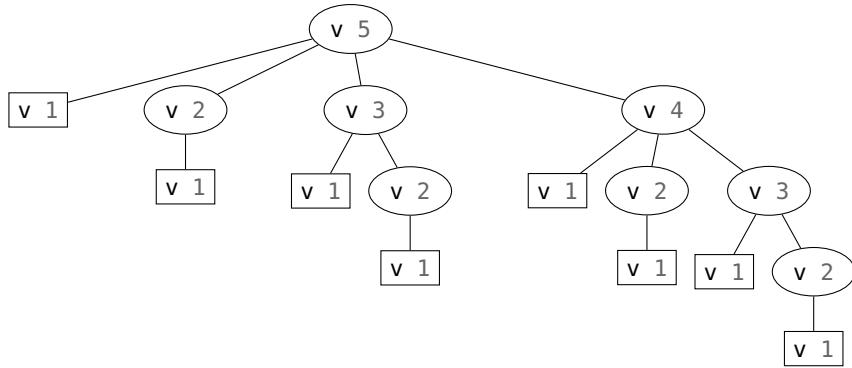
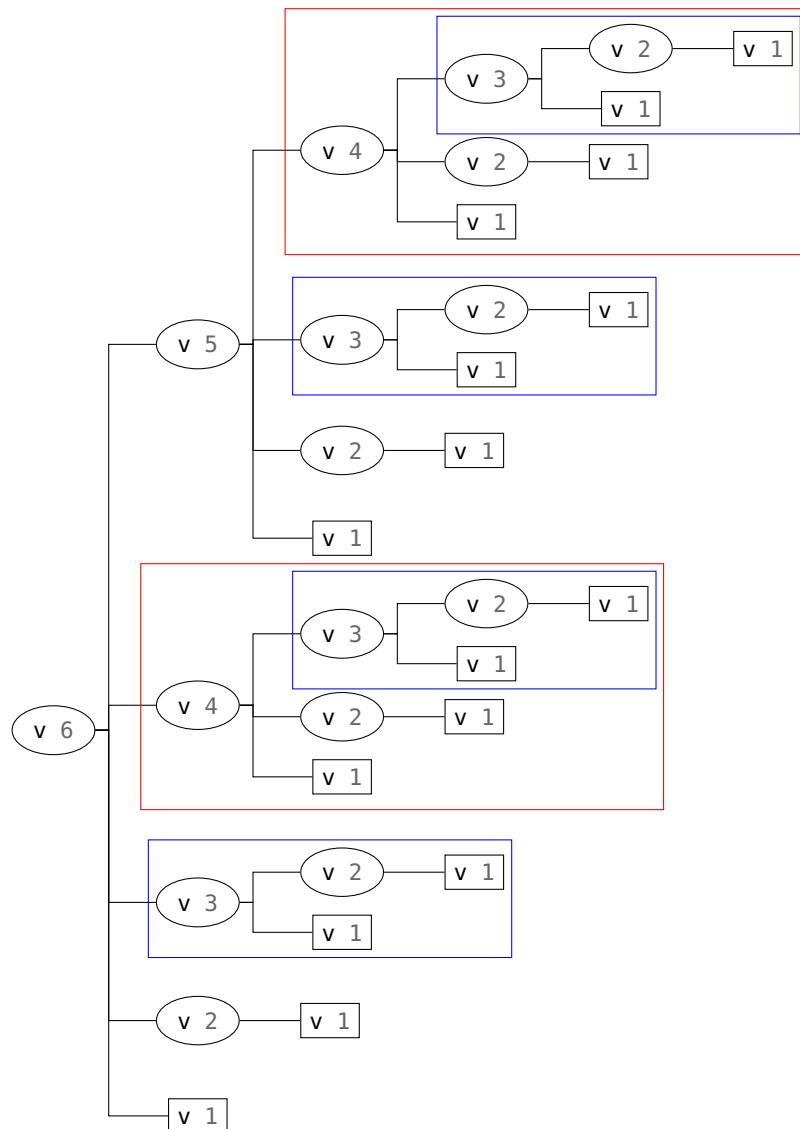
On obtient immédiatement une fonction OCaml :

```
let v_opt_rec t =
(* cas de base de val_aux : n = 0 et n = 1 *)
let rec val_aux n =
  let v_opt = ref t.(n) in
  for k = 1 to n - 1 do
    v_opt := max !v_opt (val_aux k + t.(n - k));
  done;
  !v_opt
val_aux (Array.length t - 1)
```

```
# v_opt_rec tab_exemple;;
- : int = 27
```

2.1.c Analyse de la solution récursive naïve

Pour comprendre l’inefficacité de cette solution, le mieux est de commencer par contempler les arbres d’appels obtenus :

FIGURE 13.7 – Découpage de toblerone pour $n = 5$, récursif naïfFIGURE 13.8 – Découpage optimal pour $n = 6$, récursif naïf. Les sous-arbres encadrés d'une même couleur sont identiques.

Le problème est immédiatement apparent : on passe notre temps à recalculer la même chose ! C'est précisément dans les cas où les sous-problèmes se chevauchent (et donnent donc lieu à une répétition de calculs) que la programmation dynamique est intéressante.

Complexité On peut assez aisément compter le nombre $\varphi(n)$ d’appels effectués par `v_opt_rec n` (c’est-à-dire le nombre total de noeuds de l’arbre correspondant). En effet, on a :

$$\varphi(n) = 1 + \sum_{k=1}^n \varphi(n-k) = 1 + \sum_{k=0}^{n-1} \varphi(k)$$

Une récurrence immédiate montre alors que $\varphi(n) = 2^n$. Comme le nombre d’appels minore la complexité, celle-ci reste au moins exponentielle.

Remarque

On se convaincra sans trop de mal que la complexité est en fait en $O(2^n)$.

2.1.d Solution dynamique : version itérative

Une manière plus « intelligente » d’utiliser la relation de récurrence que l’on a trouvé est de construire un tableau auxiliaire contenant les v_k . On l’initialise à 0, on le remplit au fur et à mesure (en utilisant exactement la même relation de récurrence que plus haut), et quand on a terminé on récupère le dernier élément (qui correspond à $v(n)$, la valeur cherchée).

```
let v_opt_dyn t =
  let n = Array.length t - 1 in
  let v = Array.make (n + 1) 0 in
  for k = 0 to n do
    for i = 1 to k do
      v.(k) <- max v.(k) (t.(i) + v.(k - i))
    done
  done;
  v.(n)
```

```
# v_opt_dyn tab_exemple;;
- : int = 27
```

Complexité La boucle interne s’exécute en temps $O(k)$, et donc la boucle externe en temps $O(\sum_{k=0}^n k) = O(n^2)$. L’initialisation se faisant en temps $O(n)$, on a donc au total une complexité quadratique.

2.1.e Solution dynamique : récursion avec mémoïsation

En fait, il est tout aussi simple de garder une solution récursive : il suffit de s’arranger pour ne pas faire plusieurs fois le même calcul.

Pour cela, on utilise la technique dite de *mémoïsation*¹.

- on crée un tableau `v` destiné à recevoir les valeurs de $v(k)$;
- ce tableau est initialisé avec une valeur « impossible » (ici, on peut prendre `-1`, dans le cas général on utilisera des options et l’on initialisera à `None`) ;
- la fonction auxiliaire (réursive) prend ce tableau en plus de ses arguments « normaux » ;
- quand on l’appelle sur une valeur k :
 - elle regarde dans le tableau `v` si la case k contient une « vraie » valeur ;
 - si c’est le cas, alors on a déjà calculé $v(k)$ et l’on peut directement renvoyer cette valeur ;
 - sinon, on effectue le calcul (comme dans la version récursive naïve), **puis l’on stocke le résultat**, puis l’on renvoie ce résultat ;
- la fonction principale (non récursive) se contente essentiellement de créer le tableau `v` et d’appeler la fonction auxiliaire.

1. Pourquoi pas *mémorisation*? Aucune raison valable, mais c’est comme ça...

```

let rec v_opt_mem_aux n t v =
  if v.(n) = -1 then begin
    let m = ref t.(n) in
    for k = 1 to n - 1 do
      m := max !m (v_opt_mem_aux k t v + t.(n - k))
    done;
    v.(n) <- !m
  end;
  v.(n)

let v_opt_mem t =
  let n = Array.length t - 1 in
  let v = Array.make (n + 1) (-1) in
  v_opt_mem_aux n t v

```

On peut à nouveau regarder ce que donnent les arbres d’appels : c’est beaucoup plus raisonnable !

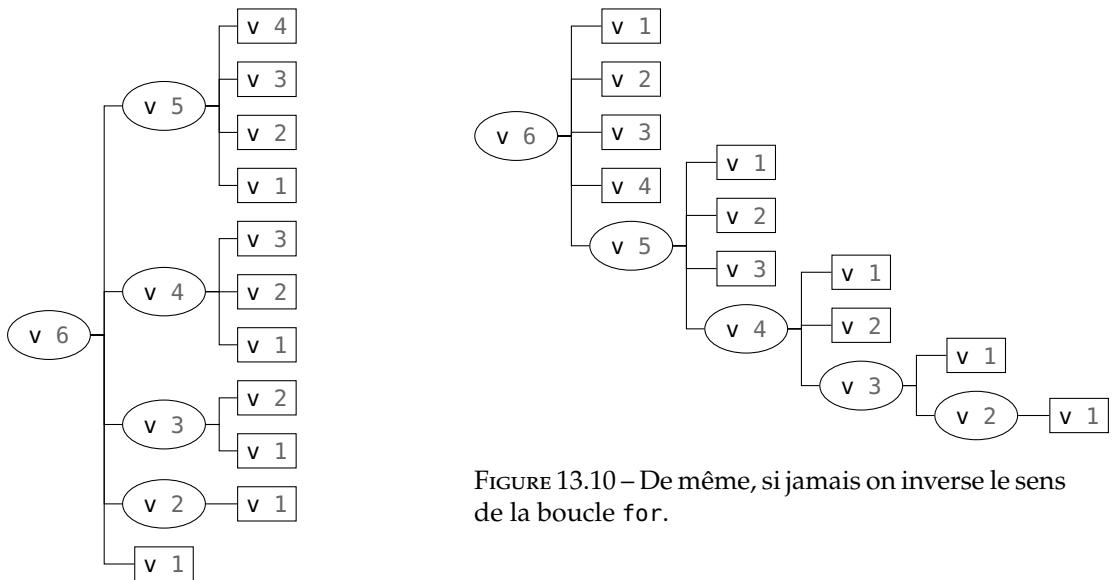


FIGURE 13.10 – De même, si jamais on inverse le sens de la boucle **for**.

FIGURE 13.9 – Découpage de toblerone pour $n = 6$, récursif mémoisé.

Complexité On analyse la version dont le code est au-dessus (boucle **for** ascendante, arbre de gauche). Si l’on a déjà effectué les appels $v_{\text{opt_mem_aux}}(k)$ pour $1 \leq k < n$, l’appel $v_{\text{opt_mem_aux}}(n)$ s’effectue en temps $O(n)$ (puisque chaque appel récursif renvoie immédiatement). Au total, pour un appel à $v_{\text{opt_mem}}(n)$, on a donc :

- une initialisation en $O(n)$ (création du tableau v);
- un appel « nouveau » à $v_{\text{opt_mem_aux}}(n)$, qui se fait en temps $O\left(\sum_{k=1}^{n-1} k\right) = O(n^2)$

Finalement, on obtient une complexité en $O(n^2)$, comme pour la solution dynamique itérative.

2.1.f Reconstruction de la découpe optimale

Pour l'instant, on a déterminé la valeur optimale d'une découpe de la barre, mais on n'est pas en mesure d'exhiber une découpe optimale. C'est à peu près systématique dans ce genre de problème : on commence par se préoccuper de déterminer la valeur optimale, puis on regarde ce qu'il faut modifier dans l'algorithme pour reconstruire la solution. Souvent, le tableau des valeurs calculées suffit ; parfois, il faut stocker un peu plus de choses.

Ici, supposons que l'on dispose du tableau v contenant les valeurs de $v(k)$ pour $k \in [0 \dots n]$. Il est alors possible de reconstruire une découpe optimale en temps $O(n)$:

- on part de la case n du tableau ;
- on se déplace vers la gauche du tableau en cherchant une case d'indice $n - k$ vérifiant $v(n - k) + t_k = v(n)$;
- une telle case existe nécessairement d'après la définition récursive de v ;
- on sait alors que l'on peut prendre un bloc de taille k comme dernier morceau de la découpe, et l'on complète cette découpe en cherchant une découpe optimale pour $n - k$.

Pour commencer, on modifie très légèrement notre fonction v_opt pour qu'elle renvoie le tableau v (et non pas seulement le contenu de sa dernière case) :

```
let v_opt_dyn t =
  let n = Array.length t - 1 in
  let v = Array.make (n + 1) 0 in
  for k = 0 to n do
    for i = 1 to k do
      v.(k) <- max v.(k) (t.(i) + v.(k - i))
    done
  done;
v
```

On peut par exemple écrire le code suivant :

```
let rec reconstruit v t n =
  if n = 0 then []
  else begin
    let i = ref 1 in
    while v.(n - !i) + t.(!i) <= v.(n) do
      i := !i + 1
    done;
    !i :: reconstruit v t (n - !i)
  end
```

Remarques

- Si l'on avait voulu une reconstruction plus simple, et un peu plus efficace², on aurait pu stocker les couples $v(n), k_0$ où $t_{k_0} + v(n - k_0) = \max_{1 \leq k \leq n} (t_k + v(n - k))$ dans le tableau (au lieu de ne stocker que les $v(n)$). Ici, cela n'apporte pas grand chose.
- On reconstruit *une* découpe optimale. Reconstruire *toutes* les découpes optimales ne peut pas se faire rapidement (en général), car leur nombre peut être très grand.

2.2 Principe général

Au départ, la programmation dynamique s'applique à des problèmes d'optimisation ayant les deux propriétés suivantes :

- *sous-structure optimale* : une solution optimale contient des solutions optimales pour des instances plus petites du même problème ;
- *chevauchement des sous-problèmes* : une solution récursive en suivant le principe du « diviser pour régner » conduit à résoudre plusieurs fois les mêmes sous-problèmes.

2. Temps proportionnel au nombre de blocs de la découpe et non à la taille totale.

Notons que les idées de la programmation dynamique peuvent être utilisées dans un cadre un peu plus large : on verra plus loin des exemples qui ne sont pas des problèmes d’optimisation. Le plus souvent, la difficulté est de transformer le problème initial (typiquement en calculant un peu plus que ce qui est demandé) pour que la propriété de sous-structure optimale soit vérifiée.

Exemple 13.5 – Parenthésage optimal

Le problème est le suivant : étant donné un produit de k matrices $A_1 \cdot \dots \cdot A_k$ à effectuer, et en considérant par exemple que la multiplication d’un matrice (n, p) par un matrice (p, q) demande nq multiplications élémentaires, choisir un parenthésage optimal (minimisant le nombre total de multiplications élémentaires).

Si l’on considère par exemple des matrices A_1, \dots, A_4 de taille respective $(5, 2), (2, 10), (10, 4)$ et $(4, 1)$, on a les possibilités suivantes :

- $(A_1 A_2)(A_3 A_4)$: coût $5 \cdot 2 \cdot 10 + 10 \cdot 4 \cdot 1 + 5 \cdot 10 \cdot 1 = 190$;
- $A_1(A_2(A_3 A_4))$: coût $10 \cdot 4 \cdot 1 + 2 \cdot 10 \cdot 1 + 5 \cdot 2 \cdot 1 = 70$;
- $A_1((A_2 A_3) A_4)$: coût $2 \cdot 10 \cdot 4 + 2 \cdot 4 \cdot 1 + 5 \cdot 2 \cdot 1 = 98$;
- $((A_1 A_2) A_3) A_4$: coût $5 \cdot 2 \cdot 10 + 5 \cdot 10 \cdot 4 + 5 \cdot 4 \cdot 1 = 320$;
- $(A_1(A_2 A_3)) A_4$: coût $2 \cdot 10 \cdot 4 + 5 \cdot 2 \cdot 4 + 5 \cdot 4 \cdot 1 = 140$.

Choisir un parenthésage revient à choisir un arbre binaire dont tous les nœuds internes sont étiquetés \times et dont les feuilles, lues de gauche à droite, sont étiquetées A_1, \dots, A_k :

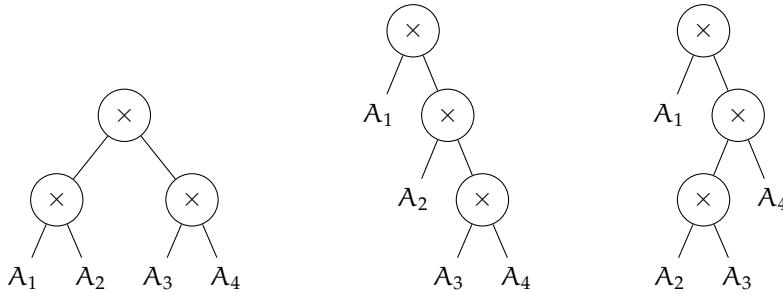


FIGURE 13.11 – Les arbres correspondant à $(A_1 A_2)(A_3 A_4)$, à $A_1(A_2(A_3 A_4))$ et à $A_1((A_2 A_3) A_4)$.

Il y a donc autant de parenthésages possibles que de formes d’arbres binaires à $k - 1$ nœuds internes, c’est-à-dire c_{k-1} . Or $c_k \sim \frac{4^k}{\sqrt{\pi k^3}}$: une solution en force brute n’est pas raisonnable pour k un tant soit peu grand.

Sous-structure optimale Dans une solution optimale, chaque sous-arbre (dont les feuilles sont étiquetées A_i, A_{i+1}, \dots, A_j) fournit une solution optimale au problème pour la multiplication $A_1 \times \dots \times A_j$.

Chevauchement des sous-problèmes Pour choisir qui mettre dans le sous-arbre gauche et dans le sous-arbre droit, il faut calculer les coûts optimaux pour A_1, \dots, A_i et A_{i+1}, \dots, A_n pour $1 \leq i < n$. Mais les sous-problèmes pour A_1, \dots, A_4 et pour A_1, \dots, A_6 ne sont clairement pas disjoints : il faudra résoudre le premier pour résoudre le second...

Plus formellement, en notant l_i, c_i la taille de A_i et $\text{opt}(i, j)$ le coût minimal du calcul de $\prod_{m=i}^j A_m$, on a :

$$\text{opt}(a, b) = \min_{a \leq i < b} (\text{opt}(a, i) + \text{opt}(i + 1, b) + l_a \cdot c_i \cdot c_b)$$

Il faudra finalement calculer les $\text{opt}(i, j)$ pour tous les couples $1 \leq i \leq j \leq n$, et la programmation dynamique permettra de ne calculer chacune de ces valeurs qu’une seule fois.

2.3 Top-down ou bottom-up

Une fois que l'on a identifié une relation de récurrence se prêtant bien à une solution par programmation dynamique, on a deux choix :

- l'approche **ascendante** (*bottom-up*) consiste (en gros) à résoudre d'abord toutes les instances de taille 1, puis celles de taille 2 et ainsi de suite jusqu'à celle qui nous intéresse. Typiquement, on obtient une solution itérative ;
- l'approche **descendante** (*top-down*) consiste à traduire directement la relation de récurrence (en une fonction récursive, donc) et à mémoiser les appels.

En règle générale, l'approche *top-down* est plus simple à écrire (et donc à privilégier) puisqu'il n'y a pas à se préoccuper de l'ordre dans lequel les calculs sont effectués. Au niveau des performances (et surtout de la complexité en espace), tout dépend du problème.

- S'il est simple de déterminer exactement quels calculs vont être nécessaires, l'approche *bottom-up* peut être intéressante. En particulier, on peut dans certains cas ne garder en mémoire que les résultats récents (ceux correspondant à des instances de taille $n - 1$), et donc faire passer la complexité en espace de $O(n^2)$ à $O(n)$ (ou de $O(n)$ à $O(1)$, ou...).
- En revanche, s'il n'est pas évident de déterminer *a priori* quelles instances il va réellement falloir résoudre, l'approche *top-down* peut éviter des calculs inutiles (nous verrons cela dans le problème du sac à dos en TD).

Exercice 13.6 – Suite de Fibonacci

Prenons un exemple sans autre intérêt que sa simplicité : Fibonacci.

Version récursive naïve On utilise directement la relation $u_{n+2} = u_{n+1} + u_n$.

```
let rec fibo_rec n =
  if n <= 1 then n
  else fibo_rec (n - 1) + fibo_rec (n - 2)
```

Chaque appel se fait en temps constant (sans compter les appels récursifs) et le nombre d'appels vérifie $\varphi_{n+2} = \varphi_{n+1} + \varphi_n + 1$. En posant $v_n = \varphi_n + 1$, on a alors $v_{n+2} = v_{n+1} + v_n$ et $v_0 = v_1 = 1$. On en déduit que la complexité temporelle est proportionnelle à u_n et donc à $\left(\frac{1+\sqrt{5}}{2}\right)^n$.

Chaque appel a une consommation mémoire en $O(1)$, et il y a au maximum n appels simultanément actifs (profondeur de l'arbre d'appels). On a donc une complexité mémoire en $O(n)$.

Version mémoisée top-down Il faut créer un tableau de taille $n + 1$ pour stocker les valeurs des u_i .

```
let fibo_mem n =
  let t = Array.make (n + 1) (-1) in
  t.(0) <- 0;
  if n >= 1 then t.(1) <- 1;
  let rec aux k =
    if t.(k) = -1 then t.(k) <- aux (k - 1) + aux (k - 2);
    t.(k) in
  aux n
```

Le tableau t occupe un espace proportionnel à n , et la profondeur d'appel est clairement majorée par n , donc la complexité spatiale est en $O(n)$.

Pour la complexité temporelle, supposons par exemple que l'appel $aux (k - 2)$ soit effectué avant l'appel $aux (k - 1)$ (l'analyse est légèrement différente dans le cas contraire, mais le résultat reste inchangé). L'appel $aux (k - 1)$ s'effectuera alors en temps $O(1)$, puisque t_{k-2} et t_{k-3} auront déjà été remplies. La complexité de aux vérifie donc $T(n) = T(n - 2) + O(1)$, ce qui donne $T(n) = O(n)$: complexité temporelle linéaire.

Version dynamique bottom-up Si l’on ne réfléchit pas particulièrement, on obtient exactement les mêmes complexités que pour la version mémoisée, mais l’on calcule explicitement les valeurs u_0, \dots, u_n dans l’ordre.

```
let fibo_dyn n =
  let t = Array.make (n + 1) 0 in
  if n >= 1 then t.(1) <- 1;
  for i = 2 to n do
    t.(i) <- t.(i - 1) + t.(i - 2)
  done;
  t.(n)
```

On peut alors remarquer que les seules valeurs dont il faut se « souvenir » pour calculer u_k (et les u_i avec $i > k$) sont u_{k-1} et u_{k-2} . Au lieu d’utiliser un tableau, on peut donc se contenter de stocker le couple (u_{k-1}, u_{k-2}) en le mettant à jour au fur et à mesure : on obtient alors une complexité spatiale en $O(1)$ (la complexité temporelle est inchangée).

```
let fibo_dyn_opt n =
  let couple = ref (0, 1) in
  for i = 2 to n + 1 do
    let a, b = !couple in
    couple := (b, a + b)
  done;
  fst !couple
```

On peut bien sûr appliquer cette technique pour les suites récurrentes doubles sans réaliser qu’il s’agit d’un cas particulier de programmation dynamique *bottom-up*.

Exemple 13.7 – Retour sur le parenthésage optimal

Comme on se rapproche ici d’une implémentation réelle, on prend une numérotation plus raisonnable : les matrices vont de A_0 à A_{n-1} , et $\text{opt}(i, j)$ est le coût minimal pour le calcul de $\prod_{k=i}^{j-1} A_k$. On va être amené, pour calculer $\text{opt}(0, n)$, à calculer les $\text{opt}(i, j)$ pour différentes valeurs de i et j .

Approche top-down On crée un tableau bi-dimensionnel t pour stocker les $\text{opt}(i, j)$. On écrit une fonction auxiliaire récursive aux dont le principe est le suivant (pour un appel aux i j :

- si $t_{i,j}$ n’est pas remplie :
 - on le calcule à l’aide de la formule

$$\text{aux } i \ j = \min_{i < k < j} (\text{aux } i \ k + \text{aux } k \ j + l_i l_k c_{j-1})$$

- on écrit le résultat dans $t_{i,j}$;
- dans tous les cas, on renvoie $t_{i,j}$.

Approche bottom-up On crée aussi un tableau t , mais cette fois il faut faire attention à ce que les valeurs soient calculées dans le bon ordre, c'est-à-dire à ce que la case $t_{i,j}$ soit remplie avant qu’on ait besoin de son contenu. Ici, c’est possible en remplaçant le tableau par $j - i$ croissants ; il y a quand même beaucoup plus de chances de faire une erreur d’indice quelque part que dans la version *top-down*.

On se convaincra assez aisément que les complexités tant spatiale que temporelle sont quadratiques dans les deux cas.

2.4 Stockage des résultats

Pour l'instant, on a toujours utilisé des tableaux (éventuellement bi-dimensionnels) pour stocker les résultats partiels. C'est possible le plus souvent, mais cela suppose que :

- la « récursion » (qu'on écrive ou non une fonction récursive) se fait sur un ou des paramètres entiers ;
- ces paramètres sont plus petits pour les résultats partiels que pour le résultat final (ou au moins bornés et prévisibles).

Ces deux conditions ne sont pas toujours réunies, et on peut alors être amené à utiliser une solution plus générale : un dictionnaire.

Exemple 13.8 – Suite de Syracuse

Pour tout entier $a \in \mathbb{N}^*$, on définit la suite de Syracuse associée par

$$u_0(a) = a \text{ et } \forall n \in \mathbb{N}, u_{n+1}(a) = \begin{cases} \frac{u_n(a)}{2} & \text{Si } u_n \text{ pair} \\ 3u_n(a) + 1 & \text{Sinon} \end{cases}$$

On admet que, pour toute valeur raisonnable de a , la suite finit par boucler sur 1, 4, 2. On note $syr(a)$ le plus petit n tel que $u_n = 1$ (appelé *temps de vol* de a).

Si l'on veut calculer *une* valeur de $syr(p)$, on n'a guère d'autre choix que de procéder « bêtement » (on pourrait éventuellement préférer une version itérative ou récursive terminale) :

```
let rec syr a =
  if a = 1 then 0
  else if a mod 2 = 0 then 1 + syr (a / 2)
  else 1 + syr (3 * a + 1)
```

Dès que l'on calcule plusieurs valeurs, en revanche, on a toutes les chances de retomber lors d'un calcul intermédiaire sur une valeur déjà calculée : par exemple, en calculant $syr(3)$, on suit la chaîne $3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$. Si l'on veut un peu plus tard calculer $syr(5)$, il serait judicieux de ne pas tout recommencer.

Supposons par exemple que l'on souhaite déterminer le maximum des $syr(k)$ pour $1 \leq k \leq n$. Vu le comportement complexe de la suite, il est impossible de prévoir à l'avance quelles valeurs on va être amené à calculer : impossible ici de procéder en *bottom-up*. Pour la même raison, le cache sera nécessairement un dictionnaire : on ne peut pas borner les k pour lesquels on va calculer $syr(k)$, et même si on le pouvait cette borne serait beaucoup trop grande. Morale de l'histoire : une fonction récursive mémoisée utilisant un dictionnaire.

```
let max_syr n =
  let cache = Hashtbl.create 1000 in
  let rec syr a =
    if Hashtbl.mem cache a then Hashtbl.find cache a
    else if a = 1 then 0
    else begin
      let res =
        if a mod 2 = 0 then 1 + syr (a / 2)
        else 1 + syr (3 * a + 1) in
      Hashtbl.add cache a res;
      res
    end in
  let rec max_aux k =
    if k = 1 then 0
    else max (syr k) (max_aux (k - 1)) in
  max_aux n
```

3 Algorithmes gloutons

Les algorithmes gloutons (*greedy* en anglais) sont une classe d’algorithmes pour des problèmes d’optimisation (c’est-a-dire des problèmes pouvant se ramener à la minimisation d’une certaine fonction sous contraintes) qui ont les propriétés suivantes :

- ils construisent leur solution de manière graduelle (autrement dit, ils décomposent le choix global à réaliser en une série de choix successifs **sur lesquels ils ne reviennent pas ultérieurement**) ;
- à chaque étape, ils font un choix *localement optimal*.

Ces algorithmes ont de nombreux avantages : ils sont souvent simples à concevoir et à implémenter, et ils s’exécutent rapidement. Cependant, il n’y a pas de raison en général pour qu’une solution construite en ne faisant que des choix locaux soit un optimum global. On a essentiellement trois situations possibles :

- il existe un algorithme glouton pour trouver la solution optimale : cet algorithme est alors le plus souvent le meilleur pour le problème considéré (*i.e.* le plus efficace) ;
- il existe un algorithme glouton donnant une solution sub-optimale mais « pas trop mauvaise » : si le problème est trop difficile à résoudre exactement, cet algorithme peut être très utile en pratique ;
- les algorithmes gloutons donnent des solutions inacceptables : on se débrouille autrement !

3.1 Premier exemple : le rendu de monnaie

On appelle *système monétaire* un ensemble $S = \{a_1, \dots, a_p\}$ d’entiers avec $1 = a_1 < a_2 < \dots < a_p$: les a_i sont les valeurs des pièces ou billets disponibles (par exemple 1, 2, 5, 10, 20, 50, 100, 200 centimes d’euro). On cherche une méthode permettant à un commerçant de rendre la monnaie à un client en utilisant le moins de pièces possibles. Plus formellement, étant donné un entier n , on cherche à déterminer un p -uplet (x_1, \dots, x_p) tel que :

- $\sum_{i=1}^p x_i a_i = n$
- $\sum_{i=1}^p x_i$ soit minimal.

L’algorithme glouton pour résoudre ce problème est le suivant : utiliser à chaque étape la plus grande pièce possible.

Exercice 13.9

p. 263

1. Écrire une fonction C ayant pour prototype :

```
int* greedy_change(int system[], int n, int target)
```

- `system` est un tableau d’entiers codant le système monétaire ;
- `n` est le nombre de pièces présentes dans le système ;
- `target` est la somme à atteindre.

Préconditions

- `system` est de longueur `n` ;
- `system` est trié par ordre croissant ;
- `target` est positif ou nul.

La fonction doit renvoyer un pointeur vers un tableau de taille `n` contenant en case `i` le nombre de pièces de valeur `system[i]` dans la solution gloutonne. S’il est impossible d’obtenir `target` avec le système donné, on le signalera au moyen d’une assertion.

2. Déterminer la complexité de cette fonction.

Pour un certain nombre de systèmes monétaires (dont le système 1, 2, 5, 10, 20, 50…), cet algorithme est optimal : ces systèmes sont dits *canoniques*. En revanche, en prenant le système 1, 7, 8 et `n` = 14, cet algorithme renvoie (8, 1, 1, 1, 1, 1) alors que la solution optimale est (7, 7). Dans ce cas, il faut utiliser une solution par programmation dynamique.

4 Exemple de solutions exactes : problèmes d’ordonnancement

La classe générale des problèmes d’*ordonnancement* (*scheduling* en anglais) est celle où l’on dispose d’un certains nombres de ressources (processeurs, employés, salles de cours) et où l’on a un certain nombre de tâches à accomplir. Typiquement, ces tâches ont une durée spécifiée, éventuellement un horaire de début et de fin, ou peut-être une *deadline*, et surtout elles monopolisent une ressource pendant qu’on les traite. On souhaite trouver une répartition optimale des tâches sur les ressources (pour une certaine définition de « optimale »).

4.1 Interval Scheduling

Entrées : un ensemble non vide d’intervalles non vides $\mathcal{S} = \{\mathopen{]}a_i, b_i\mathclose{[}, 0 \leq i \leq n - 1\}$. On parlera de requête numéro i pour l’intervalle $\mathopen{]}a_i, b_i\mathclose{[}$, et l’on notera $s(i) = a_i$ et $f(i) = b_i$ (*start* et *finish*).

Sortie : le plus grand sous-ensemble de \mathcal{S} (au sens de la cardinalité) constitué d’intervalles deux-à-deux disjoints. On parlera de *requêtes compatibles*.

Algorithme glouton : à chaque étape, on choisit une requête compatible avec celles déjà choisies suivant une certaine règle, et l’on élimine les requêtes devenues incompatibles.

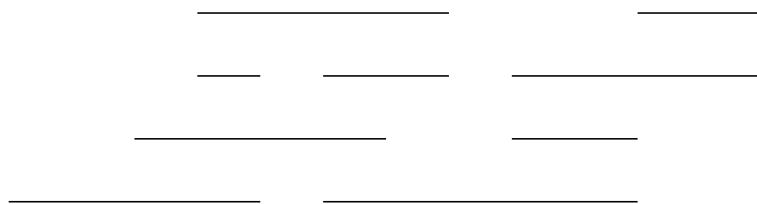


FIGURE 13.12 – Une instance du problème INTERVALSCHEDULING.

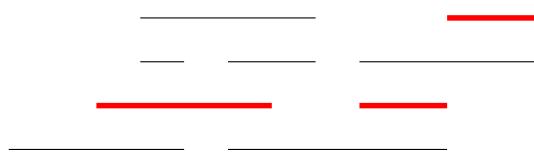


FIGURE 13.13 – Une solution non optimale.

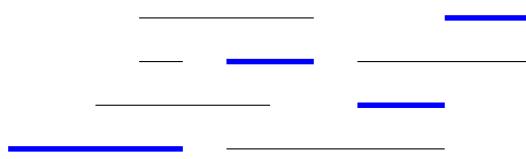


FIGURE 13.14 – Une solution optimale.

Plusieurs choix de règles sont envisageables :

1. prendre la requête commençant le plus tôt;
2. prendre la requête la plus courte (minimisant $b_i - a_i$);
3. prendre la requête ayant le moins de conflits avec des requêtes non encore choisies ;
4. prendre la requête se terminant en premier.

Exercice 13.10

p. 263

Montrer que les trois premières règles ne fournissent pas (en général) une solution optimale.

Exercice 13.11 – Optimalité de l’algorithme glouton

p. 264

On note $G = (i_1, \dots, i_m)$ les requêtes choisies par l’algorithme glouton suivant la règle 4, classées dans l’ordre croissant (de leurs horaires de fin ou de début, cela revient au même). On considère un autre ensemble $H = (j_1, \dots, j_p)$ de requêtes compatibles, également classées, avec $p \geq m$.

1. Montrer par récurrence sur $r \leq p$ la propriété P_r : « $f(i_r) \leq f(j_r)$ ».
2. Conclure.

Exercice 13.12 – Complexité de l’algorithme glouton

Justifier que l’on peut implémenter l’algorithme glouton (avec la règle 4) en temps $O(n \log n)$.

4.2 Interval Partitioning

Entrées : un ensemble non vide d’intervalles non vides $S = \{[a_i, b_i], 0 \leq i \leq n - 1\}$. On parlera de requête numéro i pour l’intervalle $[a_i, b_i]$, et l’on notera $s(i) = a_i$ et $f(i) = b_i$ (*start* et *finish*).

Sortie : une partition de S en k groupes de requêtes, chacun des groupes devant être compatibles et k devant être minimal.

Remarque

Essentiellement, la question posée est : « étant données ces n conférences ayant chacune un horaire de début et de fin fixé, de combien de salles a-t-on besoin ? ».

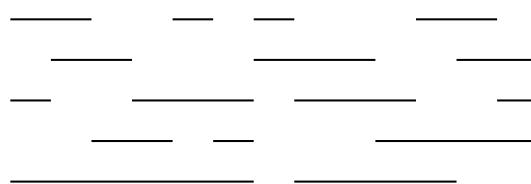


FIGURE 13.15 – Une instance du problème INTERVALPARTITIONING.



FIGURE 13.16 – Une solution non optimale pour cette instance.

FIGURE 13.17 – Une solution optimale pour cette instance.

Définition 13.4

Une *clique d’intervalles* est un ensemble d’intervalles deux à deux non disjoints, c’est-à-dire un ensemble $\{I_1, \dots, I_k\}$ tel que $I_i \cap I_j \neq \emptyset$ pour tous $1 \leq i, j \leq k$. On note $\chi(S)$ le cardinal de la plus grande clique de S .

Propriété 13.5

Tout ordonnancement valide d’un ensemble S de requêtes utilise au moins $\chi(S)$ ressources.

On considère l’algorithme (glouton) suivant :

- Trier les requêtes dans un certain ordre (précisé ultérieurement).
- Programmer la requête r_1 sur la ressource 1.
- $d \leftarrow 1$ (d contiendra toujours le nombre de ressources allouées jusqu’à présent).
- Pour i de 2 à n :
 - Chercher le plus petit $k \leq d$ tel que l’on puisse programmer la requête i sur la ressource k .
 - Si l’en existe un :
 - programmer la requête i sur la ressource k .
 - Sinon :
 - programmer r_i sur $d + 1$;
 - $d \leftarrow d + 1$.

Exercice 13.13

p. 264

1. Montrer que cet algorithme renvoie une solution compatible.
2. Exhiber une instance et un ordre de traitement des requêtes pour lesquels la solution renvoyée n'est pas optimale.
3. On suppose désormais que le tri des requêtes se fait **par ordre de début croissant**. Montrer que la solution renvoyée est optimale.

Exercice 13.14

Montrer que l'on peut implémenter cet algorithme (ou une variante qui lui équivaut) avec une complexité temporelle en $O(n \log n)$.

4.3 Retard minimum

Problème : on considère un ensemble de n requêtes, ayant chacune une durée d_i et un horaire de fin souhaité (une *deadline* pour faire court) h_i . Si la requête i se termine finalement au temps t , son *retard* est $l_i = \max(0, t - h_i)$. On ne dispose que d'une ressource, les requêtes doivent donc être programmées pour s'exécuter pendant des intervalles de temps disjoints, en commençant à l'instant $t = 0$. Le but est de minimiser le retard maximal $l = \max_{1 \leq i \leq n} l_i$.

Algorithme glouton : on programme les requêtes par *deadlines* croissantes (et sans temps mort).

Propriété 13.6

L'algorithme glouton donne une solution optimale.

Exercice 13.15

p. 265

On considère une instance du problème de taille n , et pour une solution x_1, \dots, x_n on note $s(x_i)$ l'horaire de début de x_i , $d(x_i)$ la durée de x_i et $h(x_i)$ l'horaire de fin souhaité pour x_i . On supposera toujours que $s(x_1) \leq s(x_2) \dots \leq d(x_n)$ (autrement dit, on numérote les requêtes dans l'ordre dans lequel on les a planifiées).

1. Une solution $x = (x_1, \dots, x_n)$ est dite *sans temps mort* si $s(x_i) + d(x_i) = s(x_{i+1})$ pour $1 \leq i < n$. Montrer que l'on peut se limiter à considérer des solutions sans temps mort. Une solution optimale est-elle nécessairement sans temps mort ?
2. On considère une solution sans temps mort x , et l'on suppose qu'il existe $i \in [1 \dots n - 2]$ tel que $h(x_i) > h(x_{i+1})$. Montrer que la solution y (sans temps mort) obtenue en inversant les requêtes x_i et x_{i+1} vérifie $l(y) \leq l(x)$.
3. Conclure.

Exercices

Exercice 13.16 – Détection d'un maximum local

p. 265

On considère un tableau $t = (t_0, \dots, t_{n-1})$ d'entiers. On dit que t_i est un *maximum local* de t si t_i est supérieur ou égal à son (éventuel) voisin de gauche et son (éventuel) voisin de droite. En particulier, t_0 est un maximum local si et seulement si $t_0 \geq t_1$.

1. Montrer que tout tableau non vide possède au moins un maximum local.
2. Proposer un algorithme efficace pour déterminer un maximum local d'un tableau, et donner sa complexité.
3. Implémenter cet algorithme en OCaml, sous la forme d'une fonction prenant en argument un tableau supposé non vide et renvoyant l'indice d'un maximum local (ou pic) de ce tableau.

```
indice_pic : 'a array -> int
```

4. **Difficile.** On considère maintenant un tableau bidimensionnel de taille $n \times n$. Une case du tableau possède (au plus) quatre voisins (au-dessus, en dessous, à gauche et à droite) et l'on cherche toujours un maximum local, c'est-à-dire une case dont le contenu est supérieur ou égal à celui de tous ses voisins. Trouver un algorithme permettant de résoudre ce problème en temps $O(n)$.

Exercice 13.17 – Partitions d'un entier

p. 265

Une *partition* d'un entier naturel n est une décomposition de n comme somme d'entiers strictement positifs. Deux partitions ne différant que par l'ordre des termes sont considérées comme égales, et l'on convient que la somme vide est l'unique partition de 0. On note $p(n)$ le nombre de partitions de n . On a par exemple $p(5) = 7$:

- $5 = 5;$
- $5 = 4 + 1;$
- $5 = 3 + 2;$
- $5 = 3 + 1 + 1;$
- $5 = 2 + 2 + 1;$
- $5 = 2 + 1 + 1 + 1;$
- $5 = 1 + 1 + 1 + 1 + 1.$

Combien vaut $p(1000) \pmod{2^{64}}$?

Remarques

- Le lecteur perspicace remarquera que le nombre de partitions de n est exactement le nombre de découpages distinctes d'une barre de Toblerone de taille n .
- On pourra s'intéresser à $s(n, k)$, nombre de partitions de n ne faisant intervenir que des entiers inférieurs ou égaux à k .

Exercice 13.18 – Problème du lâcher d'œufs

p. 265

On considère un immeuble de N étages (numérotés de 1 à N), dans lequel on souhaite mener à bien une expérience scientifique capitale. On dispose de p œufs, rigoureusement identiques, et l'on veut déterminer le plus petit entier k tel que la chute depuis l'étage k d'un œuf se termine par l'explosion de ce dernier. On pourra faire les hypothèses suivantes :

- si un œuf se brise après une chute de l'étage k , alors il se serait également brisé depuis une chute depuis un étage $k' \geq k$;
- inversement, si un œuf survit à une chute de l'étage k , il aurait survécu à une chute d'un

- étage $k' \leq k$;
- de plus, si un œuf survit à une chute, il n’en garde aucune séquelle : il est possible de l’utiliser dans les expériences suivantes sans que cela n’en affecte le résultat.

Le but de l’exercice est de calculer efficacement le nombre minimal de lâchers d’œufs qu’il faut effectuer dans le pire cas pour déterminer k_{\min} (l’étage à partir duquel les œufs se brisent). On notera $\varphi(N, p)$ ce nombre minimal de lancers, et l’on posera $k_{\min} = N + 1$ si l’œuf ne se casse pas quand il est lâché du dernier étage.

1. Déterminer $\varphi(N, 1)$.
2. Supposons que l’on dispose de $p > 1$ œufs, et que l’on décide de lâcher le premier œuf depuis un étage k vérifiant $2 \leq k < N - 1$.
 - a. Si l’œuf se casse, combien de lancers nous faudra-t-il dans le pire des cas pour déterminer k_{\min} ?
 - b. Même question si l’œuf ne casse pas.
3. Comment choisir les valeurs de $\varphi(0, p)$ et $\varphi(n, 0)$ pour que cette relation reste valable dans les cas $p = 1$, $k = 1$ et $k = N$?
4. En déduire une relation de récurrence permettant de calculer $\varphi(N, p)$.
5. Écrire un programme C permettant de calculer $\varphi(N, p)$ par la méthode récursive « naïve ». Ce programme prendra N et p comme arguments en ligne de commande.
 - Il sera plus pratique de coder les valeurs de $\varphi(N, p)$ par des nombres flottants (type **double** par exemple) pour pouvoir utiliser la valeur **INFINITY**.
 - **INFINITY** est définie dans **math.h**, qu’il faudra inclure. Il faudra également ajouter l’option **-lm** à la fin de la ligne de commande de compilation.
6. Écrire une nouvelle version de φ utilisant la programmation dynamique ascendante pour accélérer le calcul.
7. Déterminer la complexité en temps et en espace de cet algorithme.
8. Que peut-on dire du sens de variation de $k \mapsto \varphi(k - 1, p - 1)$ et de $k \mapsto \varphi(N - k, p)$?
9. En notant $f : k \mapsto \max(\varphi(k - 1, p - 1), \varphi(N - k, p))$, en déduire qu’un minimum local de f est forcément un minimum global.
10. Proposer alors une amélioration du calcul de φ permettant d’obtenir une complexité temporelle en $O(pN \log N)$.
11. On définit $\psi(t, p)$ comme la plus grande valeur de N pour laquelle il est possible de déterminer k_{\min} en au plus t essais si l’on dispose de p œufs. Montrer que $f(t, p) = \sum_{i=0}^p \binom{t}{i}$ et en déduire un algorithme résolvant le problème en temps $O(p \log N)$.

Exercice 13.19 – Tri fusion d’un tableau

p. 265

1. Écrire une fonction **merge** ayant la spécification et le prototype suivants :

```
void merge(int arr[], int mid, int len, int buffer[]);
```

Préconditions

- **arr** et **buffer** sont de longueur (au moins) **len**;
- $0 \leq \text{mid} < \text{len}$;
- $\text{arr}[0] \leq \text{arr}[1] \leq \dots \leq \text{arr}[\text{mid} - 1]$ et $\text{arr}[\text{mid}] \leq \dots \leq \text{arr}[\text{len} - 1]$.

Postcondition Après l’appel, $\text{arr}[0] \leq \dots \leq \text{arr}[\text{mid}] \leq \dots \leq \text{arr}[\text{len} - 1]$

2. En déduire une fonction **merge_sort** triant un tableau à l’aide de l’algorithme du tri fusion. On précisera sa complexité en temps et en espace.

```
void merge_sort(int arr[], int len);
```

Exercice 13.20 – Retour sur le rendu de monnaie

p. 265

1.

Exercice 13.21 – Saddle search

Exercice 13.22 – Double search

Exercice 13.23 – Couverture par des segments

Étant donnés n points de la droite réelle x_0, \dots, x_{n-1} .

Solutions

Correction de l'exercice 13.1 page 238

La fonction auxiliaire trie la partie $t[\text{deb} : \text{fin}]$ du tableau; s'il y a 0 ou 1 élément dans cette partie, on est dans un cas de base (la partie est déjà triée).

```
let tri_rapide t =
  let rec aux deb fin =
    if fin - deb > 1 then begin
      let ipiv = deb in
      let k = partitionne t deb fin ipiv in
      aux deb k;
      aux (k + 1) fin
    end in
    aux 0 (Array.length t)
```

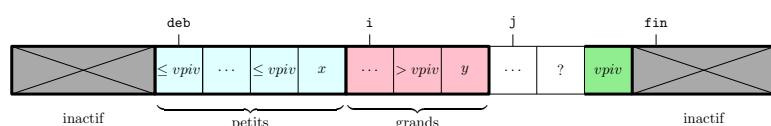
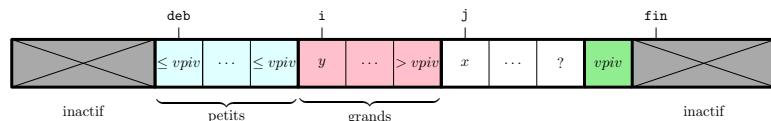
Correction de l'exercice 13.2 page 239

Il suffit en fait d'une boucle **for**:

```
let partitionne t deb fin ipiv =
  let vpiv = t.(ipiv) in
  echange t ipiv (fin - 1);
  let i = ref deb in
  for j = deb to fin - 2 do
    if t.(j) <= vpiv then begin
      echange t j !i;
      i := !i + 1;
    end
  done;
  echange t !i (fin - 1);
  !i
```

L'invariant est bien respecté au départ, puisque $i = j = \text{deb}$ et que les deux parties sur lesquelles on donne des garanties sont donc vides. Ensuite, à chaque étape :

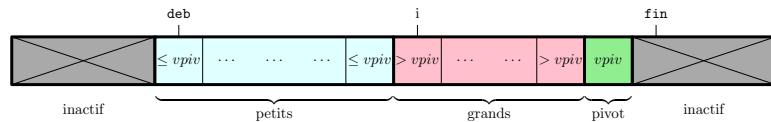
- si l'élément est strictement supérieur au pivot, on ne modifie pas i et j est incrémenté de une unité (automatiquement). Cela revient à intégrer l'élément $t[j]$ à la partie avec les « grands » éléments, ce qui est bien correct.
- sinon, on effectue la transformation suivante :



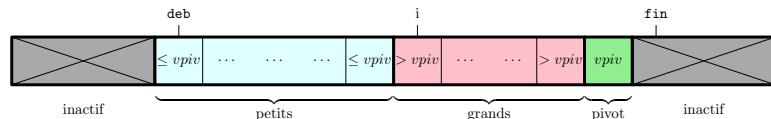
Cette transformation conserve bien l'invariant. Il faut faire attention au cas où la partie

« grands » est vide : cela correspond à $i = j$, et dans ce cas on échange l’élément avec lui-même avant d’incrémenter i et j , ce qui est encore correct.

À la fin de la boucle, on est donc dans la situation suivante :



Après l’échange `echange t[i] (fin - 1)!`, on obtient :



On renvoie i et tous les points de la spécification sont vérifiés. Le fait que la complexité temporelle soit en $O(\text{fin} - \text{deb})$ et la complexité spatiale en $O(1)$ est évident.

Remarque

On peut en fait faire rentrer l’échange du pivot dans la boucle et simplifier très légèrement le code. En réalité, cela ne me semble pas être une bonne idée : le code reste correct mais ce n’est pas immédiatement évident.

```
let partitionne t deb fin ipiv =
  let vpiv = t.(ipiv) in
  echange t ipiv (fin - 1);
  let i = ref deb in
  for j = deb to fin - 1 do
    if t.(j) <= vpiv then begin
      echange t j !i;
      i := !i + 1;
    end
  done;
  !i - 1
```

Correction de l’exercice 13.3 page 240

1.

Algorithme 5 Multiplication naïve

```
fonction MULNAIF(x, y)
  n ← max(longueur(x), longueur(y))
  s ← 0
  pour i = 0 à longueur(x) - 1 faire
    z ← MulChiffre(y, chiffre(x, i))
    s ← Addition(s, z)
  renvoyer s
```

2. On effectue n tours de boucle, et à chaque itération :

- le calcul de $y \cdot x_i$ se fait en temps $O(n)$;
- le calcul de $s + z$ se fait en temps $O(\max(\text{longueur}(s), \text{longueur}(z)))$. Or s possède au plus $2n + 1$ chiffres, donc c’est encore du $O(n)$.

Finalement, on obtient une complexité en $O(n^2)$.

Correction de l'exercice 13.4 page 243

Recherche dichotomique On a $T(n) \leq T(n/2) + An^0$, donc $a = 1$, $b = 2$ et $p = 0$. On a donc $b^p = 1 = a$, on est dans le deuxième cas. On obtient $T(n) = O(n^{\log_2 1} \log n) = O(\log n)$, c'est cohérent.

Tri fusion $a = 2$, $b = 2$ et $p = 1$, donc $b^p = 2 = a$, on est à nouveau dans le deuxième cas. On obtient $T(n) = O(n^{\log_2 2} \log n) = O(n \log n)$, qui est bien le résultat démontré au début du chapitre.

Karatsuba $a = 3$, $b = 2$ et $p = 1$, donc $b^p = 2 < a$. On est dans le premier cas, on retrouve bien $T(n) = O(n^{\log_2 3})$.

Strassen $a = 7$, $b = 2$ et $p = 2$ (puisque l'addition de deux matrices $n \times n$ se fait en temps $O(n^2)$). On a $b^p = 4 < a$, on est dans le premier cas : on obtient $T(n) = O(n^{\log_2 7})$, ce qui est strictement mieux que l'algorithme naïf en $O(n^3 = n^{\log_2 8})$. Pour fixer les idées, on a $\log_2 7 \simeq 2.81$. Attention cependant, cet algorithme n'est quasiment jamais utilisé en pratique : pour des raisons bassement matérielles, les performances sont généralement moins bonnes que ce qu'on obtient avec une version (très) optimisée de l'algorithme « naïf »^a, et il y a de plus des problèmes de stabilité numérique dans le cas des matrices de flottants. La situation est différente pour l'algorithme de Karatsuba : il est effectivement meilleur en pratique que l'algorithme naïf pour des entiers de quelques milliers de bits (mais on peut faire encore mieux pour les entiers vraiment très grands).

a. Pour être clair, les algorithmes utilisés en pratique sont tout sauf naïfs car ils sont soigneusement adaptés à l'architecture des machines actuelles, mais ils sont en $\Theta(n^3)$.

Correction de l'exercice 13.9 page 254

```
int* greedy_change(int* system, int n, int target){
    int* result = malloc(sizeof(int) * n);
    for (i = n - 1; i >= 0; i--) {
        result[i] = target / system[i];
        target = target % system[i];
        i = i - 1;
    };
    assert (target == 0);
    return result;
}
```

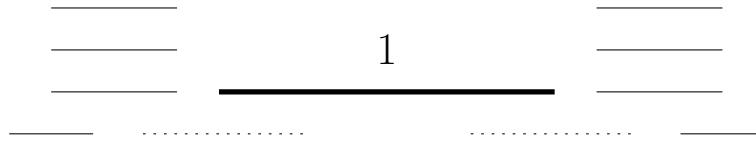
Correction de l'exercice 13.10 page 255

1. Cette règle peut évidemment être arbitrairement mauvaise : dans l'exemple ci-dessous, on ne programmera qu'une seule requête alors que la solution optimale en programme huit.

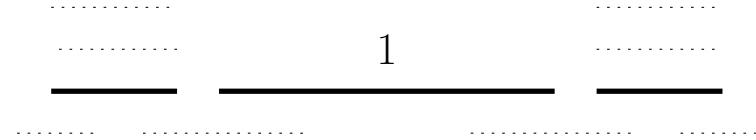
2. À nouveau, il n'est pas difficile de trouver un contre-exemple : ci-dessous, on ne programmera qu'une requête, au lieu de deux dans la solution optimale.

3. Considérons l'instance suivante :

On choisit l’unique requête ayant le moins de conflits (celle notée 1) et l’on élimine les deux requêtes incompatibles :



À présent, on choisit l’une quelconque des requêtes de gauche et l’une quelconque des requêtes de droite, ce qui permet donc au total de satisfaire trois requêtes :



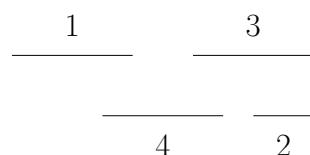
La solution optimale satisfait quatre requêtes :



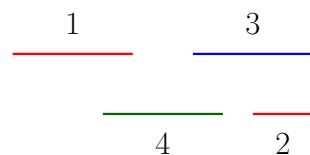
Correction de l’exercice 13.II page 255

Correction de l’exercice 13.III page 257

1. Au départ, aucune requête n'est programmée et il n'y a donc pas de conflit. La programmation d'une nouvelle requête se fait de manière à ne pas créer de conflit, une récurrence finie permet donc de conclure que l'ordonnancement total est sans conflit.
2. On considère l’instance suivante, dans laquelle les requêtes sont traitées dans l’ordre indiqué par leur numérotation :



En appliquant l’algorithme glouton avec cet ordre, on obtient une partition de taille 3 :



La solution optimale est une partition de taille 2.

3. Soit k le nombre de ressources utilisées dans la solution renvoyée par l’algorithme et $[a_i, b_i]$ la première requête programmée sur la ressource k . D’après l’algorithme, cette requête est nécessairement en conflit avec au moins $k - 1$ requêtes traitées précédemment (au moins une pour chacune des ressources $1, \dots, k - 1$). Or, puisque ces requêtes ont été traitées d’abord, leurs horaires de début sont tous inférieurs ou égaux à a_i . Puisqu’elles sont en conflit avec $[a_i, b_i]$, leurs horaires de fin sont tous strictement supérieurs à a_i , ce qui signifie que a_i appartient à k requêtes (la requête i et les $k - 1$ requêtes en conflit). Ces k requêtes forment donc une clique, ce qui montre que $k \leq \chi(\mathcal{S})$. D’après la propriété 13.5, la solution est donc optimale.

Correction de l'exercice 13.15 page 257

Correction de l'exercice 13.16 page 258

Correction de l'exercice 13.17 page 258

Correction de l'exercice 13.18 page 258

1. Si l'on ne dispose que d'un œuf, alors la seule stratégie possible est de tester tous les étages à partir de l'étage 1 : en effet, tester l'étage k sans avoir testé l'étage $k - 1$ ne permet pas de répondre si jamais l'œuf se casse. On a donc $\varphi(N, 1) = N$ (au pire, on teste tous les étages, dans le cas où k_{\min} vaut N ou $N + 1$).
2.
 - a. Si l'œuf se casse, on sait que $1 \leq k_{\min} \leq k$ et il nous reste $p - 1$ œufs. On est donc ramené au même problème avec $N' = k - 1$ et $p' = p - 1$, il faudra $1 + \varphi(k - 1, p - 1)$ lancers dans le pire cas (en comptant le lancer que l'on vient de faire).
 - b. Si l'œuf ne se casse pas, on peut considérer que l'étage $k + 1$ devient l'étage 1 d'un nouveau bâtiment à tester, comprenant $N - k$ étages. Au pire, le nombre de lancers sera $1 + \varphi(N - k, p)$.
3. Si $k = 1$, on obtient $1 + \varphi(0, p - 1)$ si l'œuf se casse. Mais dans ce cas, on sait que $k_{\min} = 1$ donc le nombre maximal d'essais est 1 : il faut donc poser $\varphi(0, x) = 0$, quelle que soit la valeur de x .
Si $p = 1$ et $k > 1$, on obtient $1 + \varphi(k - 1, 0)$ dans le premier cas avec $k - 1 > 0$. Mais dans ce cas il n'est plus possible de déterminer k_{\min} (on ne connaît pas encore la réponse et on n'a plus d'œufs), il faut donc poser $\varphi(x, 0) = \infty$ si $x > 0$.
Si $k = N$, on obtient $1 + \varphi(0, p)$ dans le deuxième cas. On sait alors que $k_{\min} = N + 1$, donc on veut à nouveau $\varphi(0, p) = 0$: c'est cohérent.
4. Pour un certain choix de k , on utilisera donc au pire $1 + \max(\varphi(k - 1, p - 1), \varphi(N - k, p))$ lancers. On choisit librement k et l'on souhaite minimiser le nombre de lancers, donc :

$$\varphi(N, p) = \begin{cases} 0 & \text{si } N = 0 \\ \infty & \text{si } p = 0 \text{ et } N > 0 \\ \min_{1 \leq k \leq N} (1 + \max(\varphi(k - 1, p - 1), \varphi(N - k, p))) & \text{sinon} \end{cases}$$

5.

Correction de l'exercice 13.19 page 259

Correction de l'exercice 13.20 page 260

GRAPHES : ASPECTS THÉORIQUES



FIGURE 14.1 – Une partie du graphe routier de Lyon (données [OpenStreetMap](#)).



FIGURE 14.2 – Les plus courts chemins en distance (à gauche) et en temps (à droite) pour aller du Lycée Jean Perrin au Lycée du Parc.

I Introduction

1.1 Graphe non orienté

Définition 14.1 – Graphe (non orienté)

Formellement, un graphe G est un couple (V, E) où :

- V est un ensemble non vide dont les éléments sont appelés *sommets*^a du graphe
- E est un ensemble de paires $\{x, y\}$ où x et y sont des éléments de V (avec $x \neq y$). Ces paires sont appelées *arêtes*^b du graphe.

- a. *vertex* (pluriel *vertices*) en anglais, d'où la notation V
b. *edge* en anglais, d'où le E

Remarques

- Une paire est un ensemble à exactement deux éléments : il n'y a donc jamais d'arête reliant x à lui-même.
- En fait, il nous arrivera d'autoriser de telles arêtes, appelées *boucles*, mais ce sera précisé explicitement.
- On notera souvent xy pour la paire $\{x, y\}$ (on a alors $xy = yx$).
- Deux sommets reliés par une arête sont dits *adjacents*.
- On appelle *voisins* d'un sommet x les sommets adjacents à x .
- On dit que l'arête $e = \{x, y\}$ est *incidente* aux sommets x et y .
- Normalement, si l'on parle d'un graphe « tout court », cela signifie graphe non orienté. En réalité, cela dépend quand même du contexte.
- Dans la grande majorité des cas, V sera un ensemble fini. Une bonne partie des définitions reste cependant valable pour un graphe infini.

Exemple 14.1 – Graphe de Facebook

Le graphe des utilisateurs de Facebook a un sommet pour chaque utilisateur et une arête entre deux sommets si les utilisateurs concernés sont « amis ». La relation d'amitié étant symétrique^a, il s'agit bien d'un graphe non orienté. Notons que $|V|$ est de l'ordre de 10^9 et $|E|$ de l'ordre de 10^{11} , ce qui pose bien sûr quelques difficultés algorithmiques.

- a. Sur Facebook, en tout cas...

Exemple 14.2 – Quelques familles de graphes usuels

- Le *graphe entièrement déconnecté* possède n sommets et aucune arête.
- Le *graphe complet* K_n possède n sommets et une arête entre chaque paire de sommets.

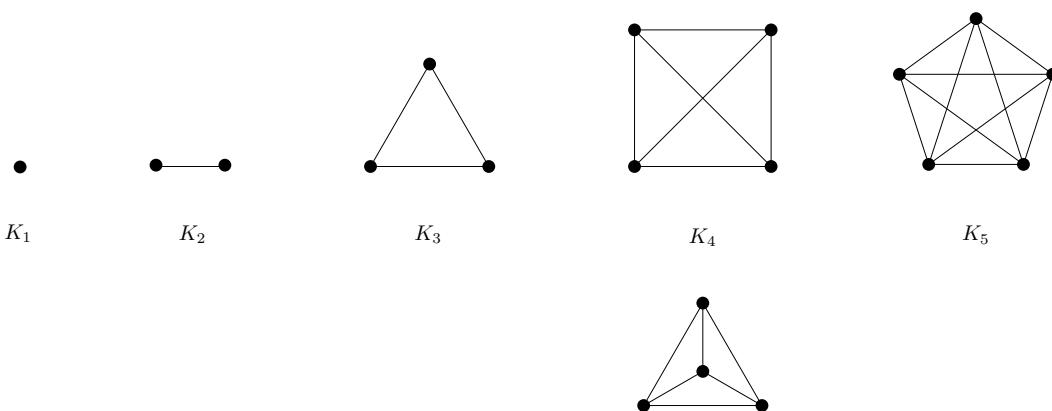


FIGURE 14.3 – Les cinq premiers graphes complets. Pour $n \geq 5$, ces graphes ne sont pas planaires.

- Le graphe chemin P_n possède n sommets numérotés $0, \dots, n - 1$ et une arête entre i et j si et seulement si $i = j + 1$.

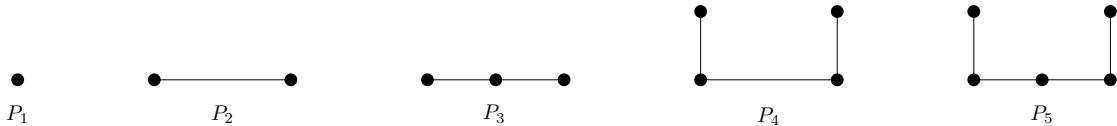
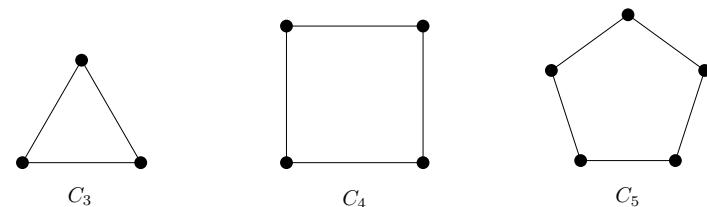
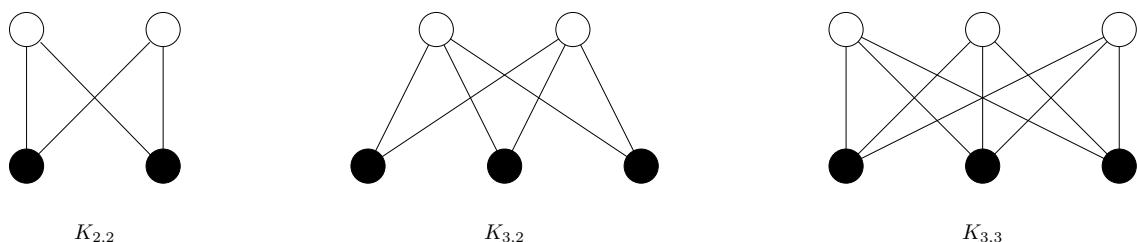


FIGURE 14.4 – Les premiers graphes chemins.

- Le graphe cycle C_n (avec $n \geq 3$) possède n sommets numérotés $0, \dots, n - 1$ et une arête entre i et j si et seulement si $i = j + 1 \pmod{n}$.


 FIGURE 14.5 – Les premiers graphes cycles. C_1 et C_2 ne sont pas définis.

- Le graphe biparti complet $K_{n,p}$ possède $n + p$ sommets, disons n noirs et p blancs, et une arête entre chacun des sommets noirs et chacun des sommets blancs. De manière générale, un graphe $G = (V, E)$ est dit biparti si l'on peut partitionner $V = V_1 \sqcup V_2$ de sorte que toutes les arêtes relient un sommet de V_1 à un sommet de V_2 .


 FIGURE 14.6 – Quelques graphes bipartis complets. Les $K_{n,p}$ avec $n \geq 3$ et $p \geq 3$ ne sont pas planaires.

Exercice 14.3 – Graphe d’intervalles

À un ensemble d’intervalles $(I_k)_{1 \leq k \leq n}$, on peut associer un graphe dont les sommets sont les I_k et qui possède une arête entre I_k et $I_{k'}$ si et seulement si $I_k \cap I_{k'} \neq \emptyset$.

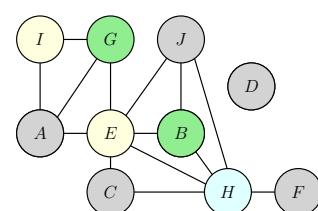
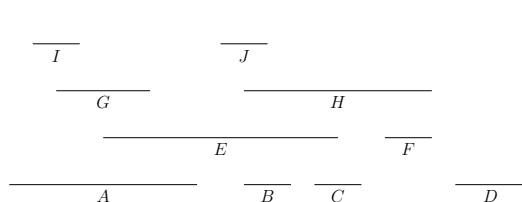


FIGURE 14.7 – Un ensemble d’intervalles et le graphe d’intersection associé.

On a ici colorié les sommets du graphe, c'est-à-dire associé à chaque sommet une couleur de manière à ce que deux sommets voisins aient deux couleurs différentes (et le coloriage choisi est optimal, en ce qu'il utilise le plus petit nombre de couleurs possibles).

1.2 Graphes orientés

Définition 14.2 – Graphe orienté

Formellement, un graphe orienté G est un couple (V, E) où :

- V est l'ensemble des sommets ;
- E est un ensemble de *couples* (x, y) où x et y sont des éléments de V (avec $x \neq y$). Ces couples sont appelées *arcs* (ou éventuellement arêtes) du graphe.

Remarques

- Intuitivement, un arc (x, y) permet de passer du sommet x au sommet y mais pas de y à x . On le représentera avec une flèche de x vers y .
- S'il y a un arc $x \rightarrow y$, on dit que y est un *successeur* de x et que x est un *prédecesseur* de y .
- En « oubliant » l'orientation, on peut toujours obtenir un graphe non orienté à partir d'un graphe orienté (si les arcs $x \rightarrow y$ et $y \rightarrow x$ étaient tous les deux présents, on ne garde qu'une arête xy).
- On pourrait considérer les graphes non orientés comme des cas particuliers de graphes orientés : ceux vérifiant $\forall(x, y) \in E, (y, x) \in E$.
- Nous avons déjà beaucoup travaillé avec un cas particulier de graphe orienté : les arbres (enracinés), où un arc part de chaque nœud pour le relier à chacun de ses enfants.

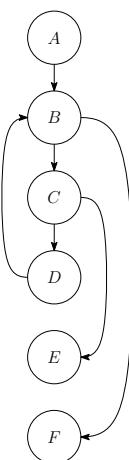
Exemple 14.4 – Graphe du Web

Le graphe du Web contient possède un sommet pour chaque page web et un arc de a vers b si la page a contient un lien vers la page b . C'est ce graphe que les moteurs de recherche parcourent pour construire leur index ; la taille du graphe du Web est inconnue, mais l'index de Google contient un peu plus de 50 milliards de pages.

Exemple 14.5 – Graphe de flot de contrôle

Les graphes jouent un rôle central dans la *compilation*, c'est-à-dire la traduction de programmes écrits dans des langages de haut niveau (comme OCaml, C, Java) vers des langages de bas niveau (code machine x86, *bytecode* pour la *Java Virtual Machine*...). Les deux graphes les plus importants dans ce contexte sont :

- le *graphe d'appels* (*call graph* en anglais) qui a un sommet pour chaque fonction et un arc de f vers g si f contient un appel à g ;
- le *graphe de flot de contrôle* (ou *control flow graph*) qui a un sommet pour chaque *bloc de base* et un arc de A vers B si l'on peut (a priori) exécuter B immédiatement après A . Un bloc de base est une suite d'instructions consécutives ne contenant ni saut ni cible de saut.



```

1:   (A) s := 0
2:   (A) i := 0
3:   (B) if i >= len(t) then GOTO 9
4:   (C) s := s + t[i]
5:   (C) if s > 10 then GOTO 8
6:   (D) i := i + 1
7:   (D) GOTO 3
8:   (E) return 100
9:   (F) s := s * s
10:  (F) return s
  
```

Le code en pseudo-assembleur correspondant au *graphe de flot de contrôle* à gauche. Pouvez-vous écrire une fonction Python donnant ce pseudo-assembleur ?

FIGURE 14.8 – Un exemple de *control flow graph*.

1.3 Extensions

De très nombreuses extensions des notions de graphes orientés ou non orientés sont possibles :

- on peut autoriser les arêtes (ou arcs) reliant un sommet à lui-même (on parle de *boucles* pour ces arcs) ;
- on peut autoriser plusieurs arêtes (ou arcs) entre une même paire (un même couple) de sommets (on parle alors de *multigraphes*) ;
- on peut ajouter des étiquettes sur les arêtes (remplacer les paires $\{u, v\}$ par des triples $\{u, v, d\}$ où d peut être un réel, un ensemble de lettres...).

Exemple 14.6 – Graphe routier

L'extrait du réseau routier lyonnais représenté en figure 14.1 contient 2 144 sommets et 4 282 arcs. Informatiquement, ce réseau est représenté par un multigraphe orienté dans lequel :

- chaque nœud est étiqueté par sa latitude, sa longitude et un identifiant unique entier ;
- chaque arc est un triplet (u, v, d) où u et v sont des numéros de sommets et d contient diverses informations :

```
# Extraction de l'arête la plus longue (segment le plus long entre deux
# intersections) :
In[353]: max(lyon.edges(data = True), key = lambda t: t[2]['length'])

Out[353]:
(60937803,
 26475855,
 {'osmid': [360034408, 440855561, 4347000, 4346838],
  'name': ['Tunnel de Caluire Extérieur', 'Boulevard Périphérique Nord'],
  'maxspeed': '70', 'lanes': ['2', '3'],
  'highway': 'trunk', 'oneway': True,
  'length': 3816.984, 'tunnel': 'yes',
  'geometry': <shapely.geometry.linestring.LineString at 0x7f4fbfb8cf60>})
```

1.4 Matrice d'adjacence

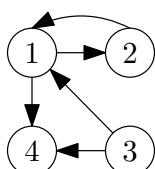
Définition 14.3 – Matrice d'adjacence

Soit $G = (V, E)$ un graphe. Étant donnée une numérotation fixée x_1, \dots, x_n des sommets, la *matrice d'adjacence* $M = (m_{i,j})$ de G est la matrice $n \times n$ définie par :

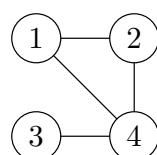
$$m_{i,j} = \begin{cases} 1 & \text{si il y a une arête entre les sommets } x_i \text{ et } x_j \\ 0 & \text{sinon} \end{cases}$$

Remarques

- Si le graphe est orienté, la convention est de mettre un 1 en $m_{i,j}$ s'il y a un arc *de* x_i *vers* x_j .
- Le graphe est non orienté si et seulement si la matrice est symétrique.
- Si l'on s'interdit les arêtes bouclant sur un sommet, il n'y a que des zéros sur la diagonale.



$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

FIGURE 14.9 – Un exemple de matrice d'adjacence pour un graphe orienté.

FIGURE 14.10 – Un exemple de matrice d'adjacence pour un graphe non orienté.

1.5 Degré, densité

Définition 14.4 – Degré d'un nœud

- Dans un graphe non orienté, on appelle *degré* d'un sommet x le nombre d'arêtes incidentes à x , c'est-à-dire le nombre d'arêtes de la forme $\{x, y\}$.
- Dans un graphe orienté, on appelle :
 - *degré entrant* d'un sommet x , noté $d_-(x)$ le nombre d'arcs de la forme (y, x) ;
 - *degré sortant* de x , noté $d_+(x)$ le nombre d'arcs de la forme (x, y) ;
 - *degré total* de x , noté $\deg(x)$ la somme de son degré entrant et de son degré sortant.

Remarques

- Dans un graphe non orienté, le degré d'un nœud est donc son nombre de voisins. La définition ci-dessus a l'avantage de pouvoir être étendue aux multigraphes.
- De même, dans un graphe orienté, le degré entrant d'un sommet est son nombre de prédécesseurs et le degré sortant son nombre de successeurs.
- Le degré sortant d'un sommet s'obtient en sommant la ligne correspondante de la matrice d'adjacence, le degré entrant en sommant la colonne.

Propriété 14.5 – Nombre maximal d'arêtes d'un graphe

- Dans un graphe orienté $G = (V, E)$, on a $|E| \leq |V| \cdot (|V| - 1)$.
- Dans un graphe non orienté, $|E| \leq \frac{|V|(|V|-1)}{2}$.

Remarques

- Pour les graphes non orientés, la borne supérieure est atteinte pour les graphes complets K_n .
- On retiendra surtout que le nombre d'arêtes est au plus de l'ordre de $|V|^2$.

Définition 14.6 – Graphe creux, graphe dense

Un graphe dans lequel le nombre d'arêtes est « faible », c'est-à-dire vérifie $\frac{|E|}{|V|^2} \ll 1$, est dit *creux* (*sparse* en anglais). Dans le cas contraire, on parle de graphe *dense*.

Remarques

- De manière équivalente, un graphe à n sommets est creux si le degré moyen de ses sommets est très petit devant n , dense sinon.
- On parle aussi de *matrice creuse* pour désigner une matrice dont l'immense majorité des coefficients sont nuls. Un graphe est creux si sa matrice d'adjacence est creuse.
- Comme nous le verrons plus tard, le choix des « bons » algorithmes et structures de données dépend souvent de la densité des graphes.
- La « plupart » des graphes sont denses, mais la « plupart » des graphes *rencontrés en pratique* ne le sont pas : voir les exemples 14.1 et 14.6.

1.6 Sous-graphe

Définition 14.7

- Un *sous-graphe* d'un graphe (orienté ou non) $E = (V, G)$ est un graphe $G' = (V', E')$ tel que :
 - $V' \subset V$;
 - $E' \subset E$ (plus précisément, E' doit être inclus dans l'ensemble des arêtes de G reliant deux sommets de V' , sinon ce que l'on obtient n'est pas un graphe).
- Un *sous-graphe induit* de E est un sous-graphe (V', E') de G tel que E' soit exactement l'ensemble des arêtes de G reliant deux sommets de V' .

Remarques

- Choisir un sous-graphe de G , c'est choisir un sous-ensemble des sommets de G et une partie des arêtes reliant ces sommets entre eux.
- Choisir un sous-graphe induit de G , c'est choisir un sous-ensemble des sommets et *toutes* les arêtes reliant ces sommets entre eux.

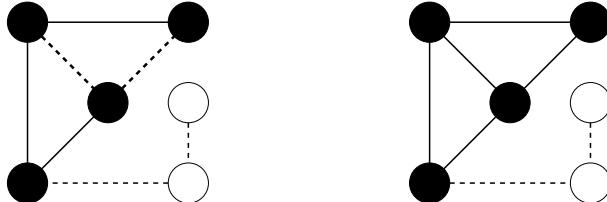


FIGURE 14.11 – À gauche, on définit un sous-graphe en ne gardant que les sommets noirs et les arêtes pleines. Ce n'est pas un sous-graphe induit car il « manque » deux arêtes. Le sous-graphe de droite, lui, est bien un sous-graphe induit.

Exercice 14.7 – Nombre de sous-graphes

p. 288

On considère un graphe G non orienté à n sommets.

1. Combien G admet-il de sous-graphes induits ?
2. Donner un encadrement le plus précis possible du nombre de sous-graphes de G .

1.7 Graphes isomorphes**Définition 14.8 – Isomorphisme de graphes**

Un *isomorphisme* entre deux graphes non orientés $G = (V, E)$ et $G' = (V', E')$ est une bijection $\varphi : V \rightarrow V'$ telle que

$$\forall x, y \in V, \{\varphi(x), \varphi(y)\} \in E' \Leftrightarrow \{x, y\} \in E$$

Deux graphes non orientés sont dits *isomorphes* s'il existe un isomorphisme entre eux.

Remarques

- La définition s'adapte immédiatement au cas des graphes orientés : $(\varphi(x), \varphi(y)) \in E' \Leftrightarrow (x, y) \in E$.
- Essentiellement, deux graphes sont isomorphes s'ils ne diffèrent que par les noms de leurs sommets.
- Comme souvent en mathématiques, on fait généralement comme si deux graphes isomorphes étaient tout simplement *égaux*. Ainsi, la plupart des graphes dessinés depuis le début de ce chapitre l'ont été avec des sommets non étiquetés : cela revient à les considérer à *isomorphisme près*.

Exemple 14.8

L'application qui envoie le sommet 1 du graphe de gauche sur le sommet a du graphe de droite, 2 sur b , ... est un isomorphisme.

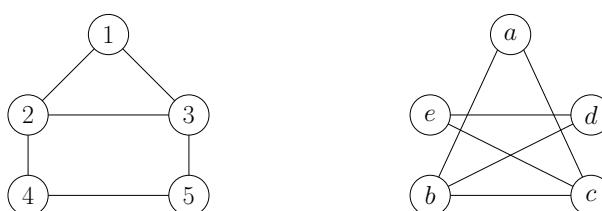
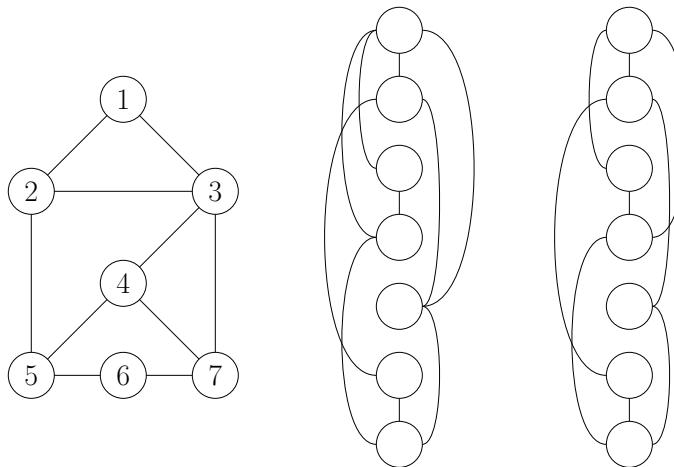


FIGURE 14.12 – Deux graphes isomorphes.

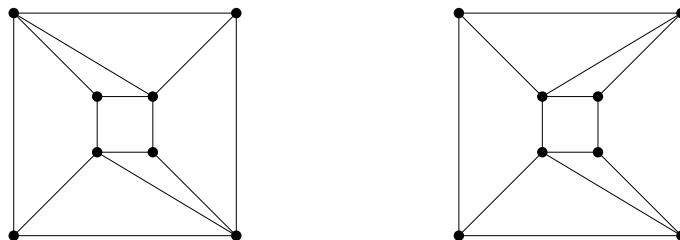
Exercice 14.9

p. 288

1. Montrer que le G_1 est isomorphe à G_2 mais pas à G_3 . On pourra s'intéresser à l'image du sommet 3.

FIGURE 14.13 – Les graphes G_1 , G_2 et G_3 .

2. Montrer que H_1 n'est pas isomorphe à H_2 .

FIGURE 14.14 – Les graphes H_1 et H_2 .

2 Graphes non orientés

2.1 Définitions élémentaires

Définition 14.9 – Chemin

Dans un graphe $G = (V, E)$, un *chemin* de longueur n est une suite z_0, \dots, z_n de $n + 1$ sommets telle que :

$$\forall i \in \llbracket 0, n - 1 \rrbracket, \{z_i, z_{i+1}\} \in E$$

z_0 et z_n sont appelés les *extrémités* du chemin, et l'on dit que le chemin *relie* z_0 à z_n .

Remarques

- On peut aussi voir un chemin de longueur n comme une suite de n arêtes consécutives.
- Un chemin de longueur n est donc constitué de $n + 1$ sommets et n arêtes.
- On accepte les chemins de longueur 0 (reliant un sommet à lui-même sans arête).

Définition 14.10 – Chemin simple, chemin élémentaire

Un chemin sera dit :

- *élémentaire* s'il ne passe pas deux fois par le même sommet, c'est-à-dire si les z_i sont deux à deux distincts ;
- *simple* s'il ne passe pas deux fois par la même arête, c'est-à-dire si les $\{z_i, z_{i+1}\}$ sont deux à deux distincts.

Remarques

- Tout chemin élémentaire est simple, la réciproque est fausse.
- Attention, ces définitions ne sont pas universelles : en théorie des graphes, on restreint souvent l'appellation *chemin* aux chemins élémentaires. La nomenclature définie ici est en revanche la plus répandue pour l'étude des algorithmes sur les graphes.
- Un chemin élémentaire de longueur n d'un graphe G est un sous-graphe de G isomorphe au graphe chemin P_{n+1} .

Définition 14.11 – Cycle

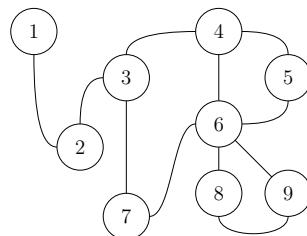
Un *cycle* est un chemin simple de longueur supérieure ou égale à 1 dont les deux extrémités sont identiques.

Remarques

- Il faut se limiter aux chemins simples, sinon on peut construire des « cycles » en parcourant les mêmes arêtes dans un sens puis dans l'autre.
- La longueur d'un cycle est donc supérieure ou égale à 3.
- Un cycle sera dit *élémentaire* si la seule répétition de sommets est celle des extrémités. Un cycle est donc élémentaire si et seulement si il ne contient pas d'autre cycle.

Exemple 14.10

- 3, 7, 6, 5, 4, 6, 7 est un chemin ;
- 7, 6, 8, 9, 6, 5 est un chemin simple ;
- 3, 7, 6, 8 est un chemin élémentaire ;
- 4, 5, 6, 9, 8, 6, 4 est un cycle ;
- ce graphe possède 4 cycles élémentaires.

**Définition 14.12 – Accessibilité**

Un sommet y est dit *accessible* depuis un sommet x s'il existe au moins un chemin entre x et y .

Remarques

- On dira aussi que x et y sont *connectés*.
- On accepte le chemin vide et x est donc toujours accessible depuis x .

Propriété 14.13

La relation d'accessibilité dans un graphe non orienté est une relation d'équivalence.

Démonstration

Réflexivité : immédiat d'après la remarque ci-dessus.

Symétrie : immédiat car le graphe n'est pas orienté.

Transitivité : si l'on dispose de deux chemins $x = x_0 \dots x_n = y$ et $y = y_0 \dots y_p = z$ reliant respectivement x à y et y à z , alors $x_0 \dots x_{n-1}y_0 \dots y_p$ est un chemin reliant x à z . ■

Propriété 14.14

Les propriétés suivantes sont équivalentes :

1. il existe un chemin entre x et y ;
2. il existe un chemin simple entre x et y ;
3. il existe un chemin élémentaire entre x et y .

Démonstration

(3) \Rightarrow (2) et (2) \Rightarrow (1) sont évidentes. Supposons donc l'existence d'un chemin entre x et y , et notons $x = x_0, x_1, \dots, x_n = y$ un tel chemin de **longueur minimale**. Ce chemin est nécessairement élémentaire : en effet, si on avait $x_i = x_j$ avec $i < j$, alors $x_0, \dots, x_i, x_{j+1}, \dots, x_n$ fournirait un chemin strictement plus court. ■

Remarque

Ainsi, quand on s'intéresse à des questions d'accessibilité, il est toujours possible de ne considérer que les chemins simples ou élémentaires. Cela montre également que le choix de la définition (restreindre la notion de chemin aux chemins simples, par exemple) n'est pas crucial.

Exercice 14.11

p. 289

Soit G un graphe non vide de degré minimal $d \geq 2$. Montrer que G possède un chemin élémentaire de longueur supérieure ou égale à d et un cycle élémentaire de longueur supérieure ou égale à $d + 1$.

2.2 Composantes connexes

Définition 14.15 – Connexité

Un graphe est dit *connexe* s'il existe un chemin reliant x et y pour tous sommets $x, y \in G$.

Définition 14.16 – Composantes connexes

Les composantes connexes d'un graphe G sont les sous-graphes induits par les classes d'équivalence de la relation d'accessibilité.

Remarque

Un graphe est connexe si et seulement si il n'a qu'une seule composante connexe.

Propriété 14.17

- La composante connexe C_x d'un sommet x est l'ensemble des sommets accessibles depuis x (y compris x).
- Si $y \in C_x$, alors $C_y = C_x$.
- Par conséquent, tout graphe peut être partitionné en un certain nombre de composantes connexes C_1, \dots, C_n vérifiant (en notant $C_i = (V_i, E_i)$) :
 - (V_1, \dots, V_n) forment une partition de V (chaque sommet est dans exactement une composante connexe);
 - E_i est exactement l'ensemble des arêtes de G dont les deux extrémités sont dans V_i ;
 - (E_1, \dots, E_n) forment une partition de E (il n'y a donc aucune arête dans G reliant un sommet de E_i avec un sommet de E_j si $i \neq j$);
 - chacun des graphes C_i est connexe.

Une telle décomposition est unique (à l'ordre près).

Remarques

- La plupart des points ci-dessus sont des reformulations dans le cas particulier des composantes connexes des propriétés usuelles des classes d'équivalence.
- Une autre caractérisation est possible : les composantes connexes de G sont les sous-graphes induits connexes maximaux de G (ceux auxquels on ne peut rajouter un sommet sans perdre la connexité).

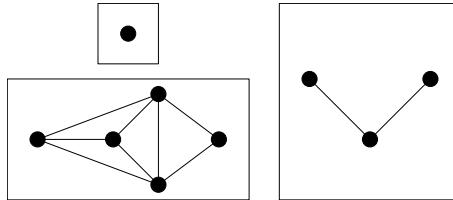


FIGURE 14.16 – Un graphe et ses trois composantes connexes.

Propriété 14.18 – Effet de l'ajout d'une arête

Soient $G = (V, E)$ et u, v deux sommets distincts et non adjacents de G . En posant $G' = (V, E \cup \{uv\})$, il y a deux cas :

- si u et v sont dans la même composante connexe de G ,** alors G' a le même nombre de composantes connexes que G et G' possède un cycle passant par l'arête uv ;
- sinon,** G' a une composante connexe de moins que G et G' ne possède pas de cycle passant par l'arête uv .

Remarque

On note souvent $G + uv$ pour le graphe $(V, E \cup \{uv\})$, et $G - uv$ pour le graphe $(V, E \setminus \{uv\})$.

Démonstration

Notons C_x, C'_x la composante connexe d'un sommet x dans G et dans G' . Notons que les seuls nouveaux chemins simples dans G' sont de la forme $\underbrace{x, \dots, y}_c, \underbrace{u, v, x', \dots, y'}_{c'}$ avec c et c' des chemins de G , et relient donc un sommet de C_u à un sommet de C_v .

Cas où $C_u = C_v$.

- Les nouveaux chemins relient des sommets qui étaient déjà dans la même composante, donc $C'_x = C_x$ pour tout x : les composantes connexes sont inchangées.
- On dispose d'un chemin u, x_1, \dots, x_p, v dans G , et ce chemin peut être choisi simple d'après 14.14. Le chemin u, x_1, \dots, x_p, v, u est alors un cycle de G' passant par l'arête uv (cette arête n'est pas répétée puisque le chemin initial était dans G).

Cas où $C_u \neq C_v$.

- On a $C'_u = C'_v = C_u \cup C_v$, les autres composantes connexes sont inchangées : il y en a donc une de moins que dans G .
- Si l'on avait un cycle x, \dots, u, v, \dots, x dans G' il y aurait alors deux chemins x, \dots, u et v, \dots, x dans G (car les arêtes d'un cycle sont distinctes, uv n'est utilisée qu'une fois). Mais alors $C_u = C_v = C_x$, ce qui contredit l'hypothèse. ■

2.3 Arbres

Définition 14.19 – Graphes acyclique

- Un graphe est dit *acyclique* s'il ne contient pas de cycle, c'est-à-dire de chemins de la forme x_0, \dots, x_n avec $x_0 = x_n$ et $n \geq 3$ ne passant pas deux fois par la même arête.
- Un graphe acyclique est aussi appelé *forêt*.
- Un *arbre* est un graphe acyclique et connexe.

Remarques

- Autrement dit, une forêt est un graphe non orienté dont les composantes connexes sont des arbres.
- Cette notion d'arbre ne correspond pas (exactement) à celle couramment utilisée en informatique :

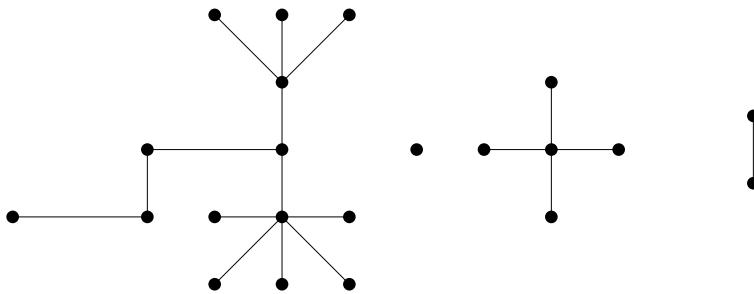


FIGURE 14.17 – Une forêt constituée de quatre arbres.

Théorème 14.20 – Caractérisations des arbres

Pour un graphe $G = (V, E)$ avec $|V| = n \geq 1$ et $|E| = p$, les propositions suivantes sont équivalentes :

1. G est un arbre ;
2. G est sans cycle et $p = n - 1$;
3. G est connexe et $p = n - 1$;
4. G est minimalement connexe (on ne peut enlever d'arête à G en préservant la connexité) ;
5. pour tous sommets x, y de G , il existe un unique chemin élémentaire reliant x et y ;
6. G est maximalement sans cycle (on ne peut rajouter d'arête à G en préservant l'acyclicité).

Démonstration

On numérote (arbitrairement) e_1, \dots, e_p les arêtes de G et l'on considère la suite G_0, \dots, G_p où $G_0 = (V, \emptyset)$ et $G_{i+1} = G_i + e_{i+1}$. On note c_i le nombre de composantes connexes de G_i .

D'après la proposition 14.18, on a $c_i \geq n - i$. De plus, on a égalitéssi tous les ajouts d'arêtes correspondent au deuxième cas de la proposition, et donc par une récurrence immédiatessi G_i est sans cycle.

- (1) \Rightarrow (2) : $G = G_p$ est connexe, donc $c_p = 1$. G est acyclique, donc $c_p = n - p$. On en déduit $p = n - 1$.
- (2) \Rightarrow (3) : $G = G_p$ est acyclique, donc $c_p = n - p$. Or $p = n - 1$, donc $c_p = 1$: G est connexe.
- (3) \Rightarrow (4) : en enlevant une arête à $G = G_{n-1}$, on obtient quitte à renommer les arêtes G_{n-2} . Or $c_{n-2} \geq n - (n - 2) = 2$: le graphe obtenu ne peut être connexe.
- (4) \Rightarrow (5) : G est connexe donc toute paire x, y de sommets est connectée. Dans tout cet item, *chemin* signifie *chemin élémentaire* (sans répétition de sommet, ni donc d'arête).

Montrons par récurrence sur k la propriété suivante : « s'il y a un chemin de longueur k entre x et y , c'est le seul chemin entre x et y ».

Pour $k = 0$, c'est vrai : on a nécessairement $x = y$ et seul le chemin vide convient.

Supposons la propriété prouvée pour $k - 1$ et considérons deux chemins $c : x = x_0, x_1, \dots, x_k = y$ et $c' : x = x'_0, x'_1, \dots, x'_l = y$ dont l'un est de longueur k . Si $x_1 \neq x'_1$, alors c' n'utilise pas l'arête xx_1 (puisque il ne repasse pas par le sommet x). Le chemin $c'x_k \dots x_1$ relie donc x et x_1 dans $G' = G - xx_1$. Donc x et x_1 sont dans la même composante connexe de G' , et G a donc le même nombre de composantes que G' d'après 14.18. Or G est minimalement connexe : c'est absurde. On a donc $x_1 = x'_1$, et l'hypothèse de récurrence nous fournit alors $x_1 \dots x_k = x'_1 \dots x'_l$, et finalement $c = c'$.

- (5) \Rightarrow (6) : si G avait un cycle, il aurait un cycle élémentaire $x_1x_2 \dots x_k = x_1$ (on a forcément $k \geq 3$). Mais alors $x_1 \dots x_{k-1}$ et $x_{k-1}x_k$ seraient deux chemins élémentaires distincts reliant x_1 et x_{k-1} , donc G est sans cycle.

Soient maintenant x, y deux sommets non adjacents de G . Il existe un chemin élémentaire $x \dots y$ dans G , donc un cycle élémentaire $x \dots yx$ dans $G + xy$: G est maximalement acyclique.

- (6) \Rightarrow (1) : il suffit de montrer que G est connexe. D'après 14.18, si deux sommets x et y n'étaient pas dans la même composante connexe, on pourrait ajouter l'arête xy à G sans créer de cycle. Ce n'est pas le cas, donc G est bien connexe.

Exercice 14.12 – Existence d'un arbre couvrant

p. 289

Un sous-graphe T d'un graphe $G = (V, E)$ est un *arbre couvrant* de G si c'est un arbre et si son ensemble de sommets est exactement V . Montrer que tout graphe connexe G admet un arbre couvrant.

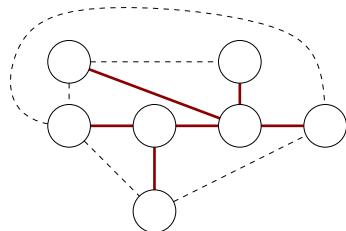


FIGURE 14.18 – Un graphe et l'un de ses arbres couvrants (arêtes en gras).

Définition 14.21 – Arbre enraciné

Un *arbre enraciné* (on parle aussi d'*arborescence*) est un arbre dans lequel on a distingué un sommet que l'on appelle *racine*.

Remarques

- Dans un arbre enraciné, il y a une unique manière d'orienter les arêtes de façon à ce que tout nœud soit accessible depuis la racine.
- Ainsi orienté, tous les nœuds sauf la racine ont un degré entrant égal à 1 (la racine a un degré entrant égal à 0).
- On retrouve donc presque la structure usuelle d'« arbre » utilisée en informatique. Attention cependant, les enfants d'un sommet ne sont pas ordonnés entre eux.

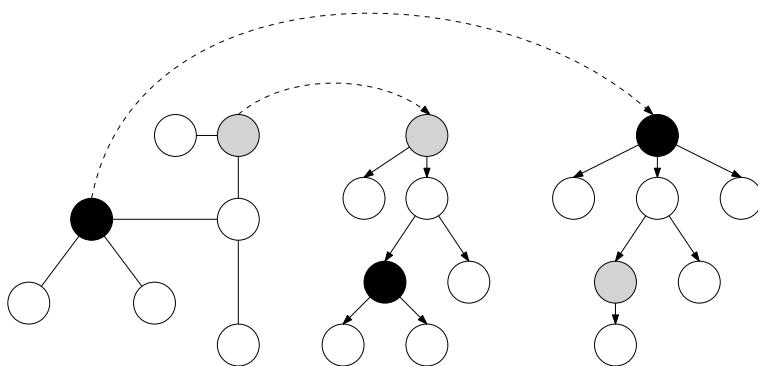


FIGURE 14.19 – Un arbre, et deux manières de l'enraciner

Exercice 14.13 – Feuilles et nœuds internes

p. 289

Un sommet d'un arbre (ayant au moins deux sommets) est appelé *feuille* si son degré vaut 1, *nœud interne* sinon.

1. Montrer que tout arbre (fini) ayant au moins deux sommets possède au moins deux feuilles.
2. Un arbre enraciné orienté de la racine vers les feuilles est dit k -aire si tous ses nœuds internes ont un degré sortant égal à k (c'est-à-dire ont k successeurs). Un arbre est dit k -aire si l'on peut l'enraciner de manière à obtenir un arbre enraciné k -aire.
 - a. Donner une condition nécessaire et suffisante sur les degrés des nœuds pour qu'un arbre soit k -aire.
 - b. Déterminer une relation entre le nombre de feuilles et le nombre de nœuds internes pour un arbre k -aire.

2.4 Coloration de graphes

Définition 14.22 – Coloration d'un graphe

Une k -coloration d'un graphe $G = (V, E)$ est une application $\varphi : V \rightarrow [0 \dots k - 1]$ vérifiant :

$$\forall u, v \in V, \{u, v\} \in E \Rightarrow \varphi(u) \neq \varphi(v).$$

Remarque

L'entier $\varphi(u)$ est appelé *couleur* de u . On exige donc que deux sommets adjacents soient toujours colorés différemment.

Définition 14.23 – Nombre chromatique

- Un graphe est dit *k -colorable* s'il admet une k -coloration.
- Le *nombre chromatique* $\chi(G)$ d'un graphe G est le plus petit entier k tel que G soit k -colorable.

Remarque

$\chi(G)$ est bien défini puisqu'il est toujours possible de colorer un graphe à n sommets en utilisant n couleurs.

Exercice 14.14

Une *clique* d'un graphe $G = (V, E)$ est un ensemble $X \subset V$ tel que $xy \in E$ pour tous x, y distincts de X . On note $\omega(G)$ (*clique number*, ou *nombre de clique*) le cardinal maximum d'une clique de G . On note également $\Delta(G) = \max_{v \in V} \deg v$ le degré maximum de G .

1. Montrer que $\omega(G) \leq \chi(G) \leq \Delta(G) + 1$.
2. Montrer que ces inégalités sont optimales en général.
3. L'une ou l'autre de ces inégalités est-elle une égalité dans le cas général ?
 - a. Qui n'a rien à voir avec le nombre de cliques!

Exemple 14.15

Une application classique de la coloration de graphes est le problème de l'*allocation de registres*. Quand on compile une fonction faisant intervenir n variables, on aimerait idéalement stocker chacune de ces variables dans l'un des p registres du processeur. Si $p \geq n$, ce n'est pas un problème, mais ce n'est généralement pas le cas. Cependant, il est possible de stocker plusieurs variables dans le même registre à condition que leurs durées de vie ne se chevauchent pas. On peut alors construire un *graphe d'interférence* des variables (avec une arête entre x et y si les durées de vie de x et y ne sont pas disjointes) et tenter de colorier ce graphe avec un nombre minimal de couleurs (les couleurs correspondant ici aux registres).

2.5 Graphes bipartis

Définition 14.24 – Graphe biparti

Un graphe $G = (V, E)$ est dit *biparti* si l'on peut partitionner V en $V = A \sqcup B$ (union disjointe), de manière à ce que toutes les arêtes relient un sommet de A à un sommet de B .

Remarques

- A (ou B) peut être vide : que peut-on dire du graphe dans ce cas ?
- On peut trouver la notation $G = (A, B, E)$ ou $G = (A + B, E)$, ou $G = (A \sqcup B, E)$.

Exercice 14.16

Montrer qu'un graphe est biparti si et seulement si il est 2-coloriable.

La notion fondamentale sur les graphes bipartis est celle de *couplage* :

Définition 14.25 – Couplage

Soit $G = (V, E)$ un graphe. Un *couplage* (ou *appariement*, ou *matching* en anglais) de G est une partie X de E ne contenant pas deux arêtes partageant une extrémité. Un couplage X est dit *parfait* si chaque sommet du graphe est incident à une arête de X .

Exercice 14.17

Reformuler en termes de couplage les problèmes suivants :

1. dans une matrice carrée $M \in \mathcal{M}_n(\mathbb{R})$, trouver une permutation σ de $[1 \dots n]$ maximisant $\sum_{i=1}^n m_{i,\sigma(i)}$;
2. étant données n candidats c_1, \dots, c_n et p formations f_1, \dots, f_p , offrant un nombre de places $p(f_1), \dots, p(f_p)$, et sachant que chaque candidat a un ordre de préférence pour les formations, et chaque formation un ordre de préférence pour les candidats, affecter les candidats aux formations de manière « optimale ».

3 Graphes orientés

Une bonne partie de ce que nous avons dit sur les graphes non orientés s'applique tel quel. Il y a cependant quelques différences :

- un arc (x, y) est parfois noté xy , mais cette fois $xy \neq yx$;
- un chemin est une suite finie d'arcs $x_1x_2, x_2x_3, \dots, x_{n-1}x_n$;
- un chemin est simple s'il ne passe pas deux fois par le même arc, élémentaire s'il ne passe pas deux fois par le même sommet ;
- un cycle est un chemin simple vérifiant $x_1 = x_n$;
- la relation d'accessibilité n'est plus une relation d'équivalence : elle reste transitive et réflexive, mais la symétrie est perdue. y est accessible depuis x s'il existe un chemin *de* x *vers* y .

3.1 Composantes fortement connexes

Définition 14.26

Un graphe orienté est dit *fortement connexe* si, pour tout couple de sommets (u, v) de G , il existe un chemin de u vers v .

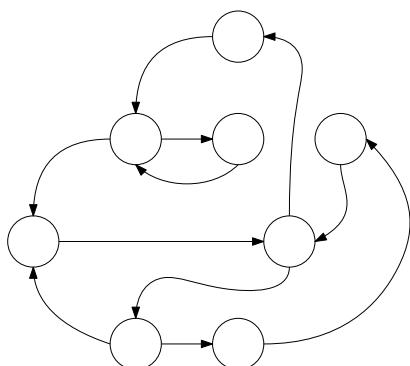


FIGURE 14.20 – Un graphe fortement connexe.

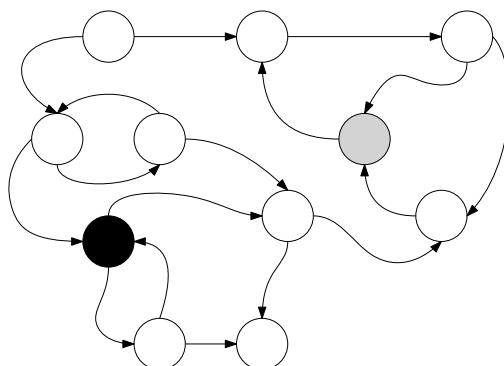


FIGURE 14.21 – Un graphe non fortement connexe : le sommet noir n'est pas accessible depuis le sommet gris.

Remarques

- Si un graphe orienté G est fortement connexe, le graphe non orienté obtenu en oubliant l'orientation de G est connexe.
- La réciproque est fausse, comme le montre la figure 14.21 (c'est d'ailleurs pour cela que l'on a besoin d'une notion différente!).
- Un graphe orienté est dit *faiblement connexe* si l'oubli de l'orientation donne un graphe connexe, *connexe* s'il existe systématiquement un chemin de u vers v ou de v vers u . Ces deux notions sont rarement utiles.
- Un arbre enraciné est faiblement connexe mais n'est pas connexe (sauf exception), et encore moins fortement connexe.

Définition 14.27

La relation \mathcal{R} définie sur les sommets d'un graphe G par :

$$x\mathcal{R}y \Leftrightarrow (x \text{ accessible depuis } y \text{ et } y \text{ accessible depuis } x)$$

est une relation d'équivalence.

Les graphes induits par les classes d'équivalence de \mathcal{R} sont appelés *composantes fortement connexes* de G .

Remarques

- La réflexivité, la symétrie et la transitivité de \mathcal{R} se vérifient aisément.
- Autrement dit, les composantes fortement connexes d'un graphe forment une partition de G en $(V_1, E_1), \dots, (V_n, E_n)$ telle que x et y appartiennent au même V_i si et seulement si x est accessible depuis y et y depuis x .
- De manière similaire au cas non orienté, les composantes fortement connexes de G sont les sous-graphes induits maximalement fortement connexes de G .

Exercice 14.18

p. 290

Déterminer les composantes fortement connexes du graphe de la figure 14.21 (il y en a 6).

3.2 DAG**Définition 14.28**

On appelle *graphe orienté acyclique* (ou plus couramment *DAG* pour *directed acyclic graph*) un graphe orienté ne possédant pas de cycle (orienté).

Remarque

Le graphe obtenu en oubliant l'orientation d'un DAG n'est en règle générale pas acyclique, comme le montre la figure 14.22.

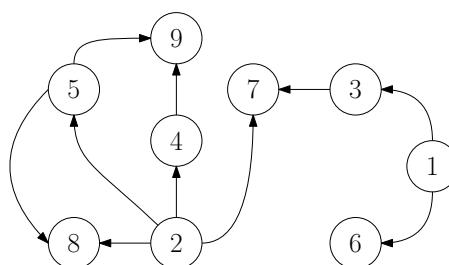


FIGURE 14.22 – Un exemple de DAG.

Exercice 14.19 – Graphe des CFC

p. 290

À un graphe orienté $G = (V, E)$, on peut associer le graphe $G' = (V', E')$ de ses composantes fortement connexes, défini par :

- V' est l'ensemble $\{C_1, \dots, C_k\}$ des CFC de G ;
- G' contient un arc $C_i C_j$ si et seulement si il existe $x \in C_i$ et $y \in C_j$ tels que $(x, y) \in E$.

Montrer que G' est un DAG.

► Exercice 14.20

p. 290

Dans un graphe orienté, on appelle *racine* tout sommet de degré entrant nul et *feuille* tout sommet de degré sortant nul.

Montrer qu'un graphe orienté acyclique (fini) possède au moins une racine et au moins une feuille.

Définition 14.29 – Ordre topologique

Un *ordre topologique* sur un graphe orienté G est un ordre total sur les sommets de G tel que, s'il y a un arc de u vers v , alors $u \prec v$.

Propriété 14.30

Un graphe orienté admet un ordre topologique si et seulement si il est acyclique.

Démonstration

Le sens direct est (presque) immédiat. Le sens indirect peut se traiter par récurrence à l'aide de l'exercice 14.20. ■

Remarques

- Le problème consistant à déterminer un ordre topologique sur un DAG donné est appelé *tri topologique*.
- Un DAG admet en général plusieurs ordres topologiques.
- Une interprétation du problème du tri topologique est la suivante :
 - les nœuds représentent des tâches à accomplir ;
 - les arcs représentent des dépendances : la présence de l'arc (u, v) signifie que la tâche v ne peut être traitée qu'une fois que la tâche u est terminée ;
 - un ordre topologique sur le graphe correspond alors à un ordonnancement admissible des tâches : à chaque fois que l'on traite une tâche, tous les pré-requis ont déjà été traités.

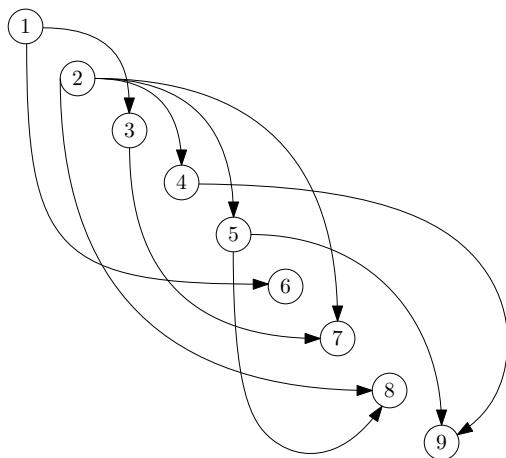


FIGURE 14.23 – L'ordre $1 \prec 2 \prec 3 \cdots \prec 9$ est un ordre topologique sur le DAG de la figure 14.22.

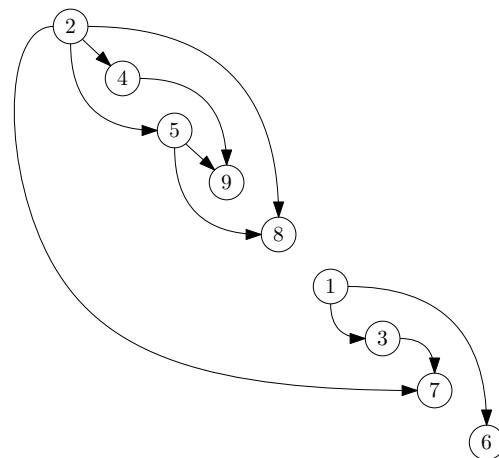


FIGURE 14.24 – L'ordre $2 \prec 4 \prec 5 \prec 9 \prec 8 \prec 1 \prec 3 \prec 7 \prec 6$ est un autre ordre topologique sur le même DAG.

Le cas particulier important d'un DAG ne possédant qu'une seule racine peut être vu comme celui d'un arbre « compressé » par partage des sous-arbres identiques :

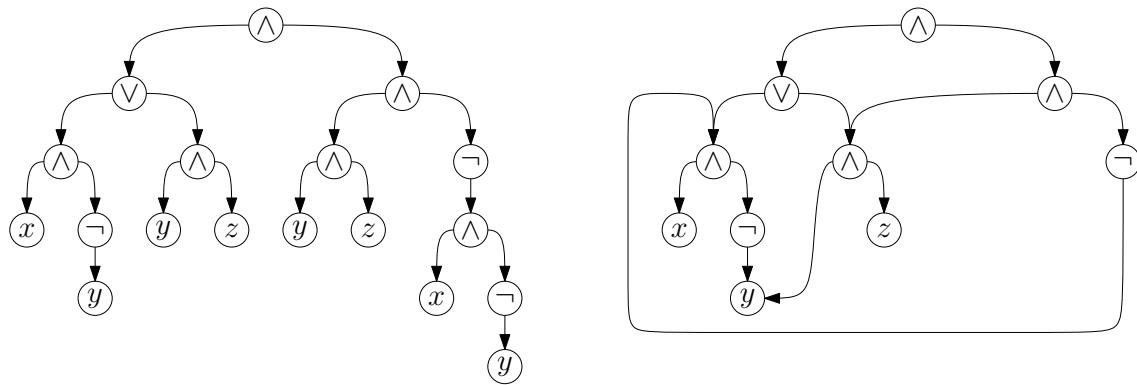


FIGURE 14.25 – Deux représentations de la même formule propositionnelle ; la version de droite peut être stockée de manière bien plus compacte en mémoire.

Exercices

Exercice 14.21

p. 290

Un graphe non étiqueté est, comme son nom l'indique, un graphe dans lequel les sommets ne portent pas d'étiquettes : autrement dit, c'est une classe d'isomorphisme de graphes étiquetés.

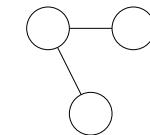
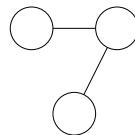
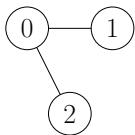
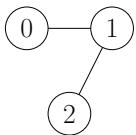


FIGURE 14.26 – Deux graphes étiquetés distincts.

FIGURE 14.27 – Deux graphes non étiquetés identiques.

1. Combien y a-t-il de graphes étiquetés par $\{1, \dots, n\}$?
2. Combien y a-t-il de graphes étiquetés par $\{1, \dots, n\}$ à p arêtes?
3. Déterminer le nombre de graphes non étiquetés à 3 sommets.
4. Énumérer les 11 graphes non étiquetés à 4 sommets.

Il n'y a pas de formule close (ni même de relation de récurrence simple) pour le nombre de graphes non étiquetés à n sommets.

Exercice 14.22

p. 291

Montrer qu'un graphe (non orienté) a forcément un nombre pair de sommets de degré impair.

Exercice 14.23 – Nombre de chemins de longueur n

p. 291

On considère un graphe orienté défini par sa matrice d'adjacence $M \in \mathcal{M}_N(\mathbb{B})$, et l'on note $c_{i,j,n}$ le nombre de chemins de longueur n du sommet i vers le sommet j . On définit également les matrices $A_n = (c_{i,j,n})$.

1. Expliciter les matrices A_0 et A_1 .
2. Trouver une relation de récurrence reliant $c_{i,j,n+1}$ aux $c_{i,k,n}$ pour $1 \leq k \leq N$.
3. En déduire une expression simple de A_n en fonction de M et de n .
4. À quelle condition la matrice d'adjacence d'un graphe orienté est-elle nilpotente?

Exercice 14.24 – Suites graphiques

Une suite finie (d_1, \dots, d_n) d'entiers naturels est dite *graphique* s'il existe un graphe G à n sommets x_1, \dots, x_n tel que $\deg x_i = d_i$ pour $1 \leq i \leq n$. Le *problème de la réalisation d'un graphe* consiste à déterminer si une suite donnée est graphique.

1. Soit $A = (s, t_1, \dots, t_s, d_1, \dots, d_n)$ une suite décroissante (au sens large) d'entiers naturels. On souhaite montrer l'équivalence suivante : A est graphique si et seulement si la suite $A' = (t_1 - 1, \dots, t_s - 1, d_1, \dots, d_n)$ l'est.
 - a. Montrer que A' graphique implique A graphique.
 - b. On suppose que A est graphique, et l'on considère un graphe G dont la suite des degrés est A . On note $(S, T_1, \dots, T_s, D_1, \dots, D_n)$ la suite des sommets de G , dans l'ordre correspondant à A (autrement dit, avec $\deg S = s$, $\deg T_1 = t_1 \dots$). On note $k(G)$ le nombre de voisins de S qui ne sont pas dans l'ensemble $\{T_1, \dots, T_s\}$.

- (i) Montrer que si $k(G) = 0$, alors A' est graphique.
- (ii) On suppose $k(G) \geq 1$. Montrer qu'il existe un graphe H ayant A comme suite de degrés et vérifiant $k(H) < k(G)$.
- (iii) Conclure.
- c. En déduire un algorithme permettant de décider si une suite finie d'entiers est graphique.
- d. Écrire une fonction `est_graphique` implémentant cet algorithme.

```
est_graphique : int list -> bool
```

- e. Quelle est la complexité de `est_graphique` (en fonction de la longueur n de la suite) ? Expliquer comment obtenir une complexité en $O(n^2)$ (on ne demande pas de le faire).

Exercice 14.25 – Théorème de Turán

p. 292

Les graphes de cet exercice ne sont pas orientés.

- Une r -clique d'un graphe $G = (V, E)$ est une clique de G de cardinal r (un ensemble de r sommets de G deux à deux adjacents). Autrement dit, une r -clique de G est un sous-graphe de G isomorphe à K_r .
- G est dit *sans* K_r s'il ne possède pas de r -clique.
- Un ensemble $H \subset V$ est dit *indépendant* s'il ne contient pas deux sommets adjacents dans G .
- G est dit k -parti si l'on peut partitionner V en $V_1 \sqcup \dots \sqcup V_k$ où chacun des V_i est indépendant.
- Un graphe k -parti, avec $V = V_1 \sqcup \dots \sqcup V_k$ est dit *complet* si tous les éléments de V_i sont adjacents à tous les éléments de V_j pour tous $1 \leq i < j \leq k$.
- Un graphe est dit *maximamente sans* K_r s'il est sans K_r et s'il est impossible de lui rajouter une arête sans créer de r -clique.
- Un graphe à n sommets est dit *optimalement sans* K_r s'il est sans K_r et si son nombre d'arêtes est maximal parmi tous les graphes sans K_r à n sommets.

On s'intéresse ici au nombre maximal d'arêtes que peut posséder un graphe sans K_r ayant n sommets.

1. Donner un exemple de graphe maximamente sans K_3 qui ne soit pas optimalement sans K_3 .
2. Montrer qu'un graphe r -parti est nécessairement sans K_{r+1} .
3. On définit le *graphe de Turán* $T_{n,r}$ comme le graphe r -parti complet à n sommets obtenu en partitionnant les n sommets en r ensembles « aussi égaux que possibles ».

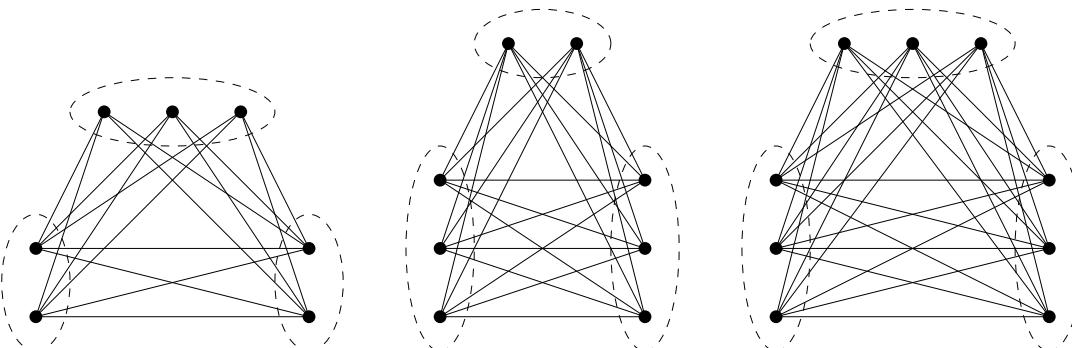


FIGURE 14.30 – Les graphes $T_{7,3}$, $T_{8,3}$ et $T_{9,3}$.

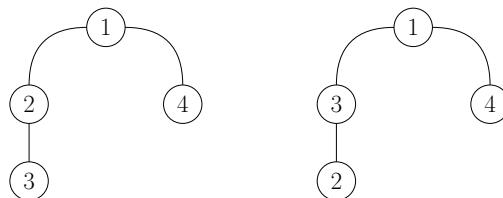
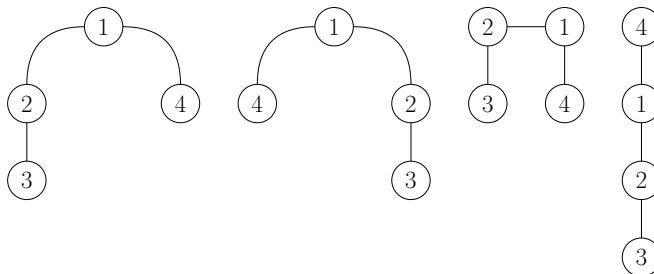
- a. On considère un graphe r -parti $G = (V, E)$, avec $V = V_1 \sqcup \dots \sqcup V_r$ et $|V_i| - |V_j| \geq 2$ (pour un certain couple i, j). Montrer que G n'est pas optimalement sans K_{r+1} .
Indication : on pourra « déplacer » un sommet de V_i .

- b. On note $e_{n,r}$ le nombre d'arêtes du graphe $T_{n,r}$. On pose aussi $n = pr + s$ avec $0 \leq s < r$.
- (i) Justifier que $e_{n,r} = \binom{n}{2} - s\binom{p+1}{2} - (r-s)\binom{p}{2}$.
 - (ii) Montrer alors que :
- $$e_{n,r} = \left(1 - \frac{1}{r}\right) \cdot \frac{n^2}{2}.$$
- Le calcul est assez pénible et peut être sauté.*
4. On considère un graphe G à n sommets sans K_{r+1} , et l'on suppose que u, v et w sont trois sommets de G tels que $uv \notin E$, $vw \notin E$ et $uw \in E$.
- On suppose que $\deg v < \deg u$, et l'on considère le graphe $G' = (V', E')$ obtenu à partir de G en :
 - supprimant v (et toutes les arêtes qui lui étaient incidentes);
 - créant un nouveau sommet u' « copie » du u , qui a exactement les mêmes voisins que u (mais $uu' \notin E'$).
 Montrer que G' est sans K_{r+1} et qu'il possède strictement plus d'arêtes que G .
 - Montrer de même que si $\deg v \geq \max(\deg u, \deg w)$, on peut trouver un graphe G' sans K_{r+1} ayant strictement plus d'arêtes que G .
5. En déduire que si G est optimalement sans K_{r+1} , alors la relation \sim défini sur V par $x \sim y$ si $xy \notin E$ est une relation d'équivalence.
6. En déduire le *théorème de Turán* : les graphes optimalement sans K_{r+1} sont exactement les graphes de Turán $T_{n,r}$.
- a. Vocabulaire non standard.

Exercice 14.26 – Formule de Cayley

p. 292

On souhaite déterminer le nombre T_n d'arbres distincts que l'on peut former à partir de n sommets étiquetés $1, \dots, n$. Deux arbres sont distincts si et seulement si ils n'ont pas les mêmes arêtes :

FIGURE 14.31 – Deux arbres distincts : $E = \{\{1, 2\}, \{1, 4\}, \{2, 3\}\}$ et $E = \{\{1, 3\}, \{1, 4\}, \{2, 3\}\}$.FIGURE 14.32 – Quatre représentations du même arbre ($E = \{\{1, 2\}, \{1, 4\}, \{2, 3\}\}$).

On s'intéresse aux séquences ordonnées e_1, \dots, e_{n-1} d'arcs orientés telles que le graphe orienté (V, E) avec $V = \{1, \dots, n\}$ et $E = \{e_1, \dots, e_{n-1}\}$ soit un arbre enraciné. On note S_n le nombre de

telles séquences.

1. Montrer que $S_n = n! \cdot T_n$.
2. Déterminer S_n .
3. Conclure.

Exercice 14.27 – Caractérisation des graphes bipartis

p. 293

Montrer qu'un graphe est biparti si et seulement si il ne possède pas de cycle de longueur impaire.

Une possibilité est d'utiliser le résultat de l'exercice 14.12.

Exercice 14.28

p. 293

Une application de $X_n = [0 \dots n - 1]$ dans lui-même est dite *nilpotente* s'il existe $r \in \mathbb{N}$ telle que $\forall x \in X_n, f^r(x) = 0$. Dénombrer ces applications.

Solutions

Correction de l'exercice 14.7 page 272

1. Choisir un sous-graphe induit revient à choisir une partie des sommets de G . Il y a donc 2^n sous-graphes induits.
2. Une fois choisi un ensemble de k sommets, choisir un sous-graphe revient à choisir une partie des arêtes reliant ces k sommets entre eux. Il y en a entre 0 et $\binom{k}{2} = \frac{k(k-1)}{2}$; de plus, si le graphe initial est sans arête, il y en aura systématiquement zéro, et systématiquement $\frac{k(k-1)}{2}$ s'il est complet. En notant $sg(G)$ le nombre de sous-graphes de G , on a donc :

$$\sum_{k=0}^n 2^0 \binom{n}{k} \leq sg(G) \leq \sum_{k=0}^n 2^{k(k-1)/2} \binom{n}{k}$$
$$2^n \leq sg(G) \leq \sum_{k=0}^n 2^{k(k-1)/2} \binom{n}{k}$$

Correction de l'exercice 14.9 page 273

1. On peut commencer par dresser la liste (ordonnée) des degrés des nœuds. On obtient :

$G_1 : 2, 2, 3, 3, 3, 3, 4$

$G_2 : 2, 2, 3, 3, 3, 3, 4$

$G_3 : 2, 2, 2, 3, 3, 3, 3$

Cela montre déjà que G_3 n'est pas isomorphe à G_1 . Ensuite, il n'est pas très difficile de trouver un isomorphisme entre G_1 et G_2 , en commençant par l'unique sommet de degré 4, puis ses voisins...

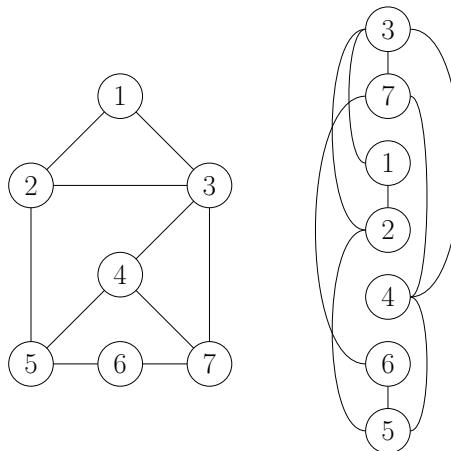


FIGURE 14.37 – Les graphes G_1 et G_2 sont isomorphes.

2. Dans le graphe H_1 , le sommet en haut à gauche est de degré 4 et ses voisins sont de degré 3, 3, 3 et 4. Chacun des sommets de degré 4 dans le graphe H_2 a des voisins de degré 3, 3, 4 et 4. Les deux graphes ne sont donc pas isomorphes.

Correction de l'exercice 14.11 page 275

Soit n le nombre de sommets du graphe. Les sommets d'un chemin élémentaire sont distincts, il y en a donc au plus n , et la longueur d'un tel chemin est donc majorée par $n - 1$. On peut donc considérer un chemin élémentaire x_0, \dots, x_k de longueur k maximale. Si x_k avait un voisin y n'appartenant pas à $A = \{x_0, \dots, x_{k-1}\}$, alors x_0, \dots, x_k, y serait un chemin élémentaire de longueur $k + 1$, ce qui est impossible. Par conséquent, tous les voisins de x_k sont dans A , d'où $\text{Card } A = k \geq \deg x_k \geq d$. Il y a donc bien un chemin élémentaire de longueur au moins d . Considérons maintenant le plus petit i tel que x_i soit un voisin de x_k : $x_i, x_{i+1}, \dots, x_k, x_i$ est un cycle de longueur $k - i + 1$. Or tous les voisins de x_k sont dans l'ensemble $\{x_i, \dots, x_{k-1}\}$ qui est de cardinal $k - i$: on a donc $k - i \geq d$, et le cycle obtenu est de longueur au moins $d + 1$.

Correction de l'exercice 14.12 page 278

Considérons l'algorithme suivant :

- On pose $E' = E$ et $H = (V, E')$.
- Tant que H n'est pas minimalement connexe :
 - soit $e \in E'$ tel que $H - e$ soit connexe;
 - $E' \leftarrow E' - e$
- On renvoie H .

H est connexe initialement car G l'est et reste connexe tout au long de l'exécution par construction. L'algorithme termine en au plus $|E|$ étapes puisqu'on enlève une arête à chaque itération, et renvoie un graphe $H = (V, E')$ minimalement connexe, qui est donc un arbre couvrant de G d'après le théorème 14.20.

Correction de l'exercice 14.13 page 278

1. Considérons un chemin élémentaire (sans répétition de nœud) de longueur maximale, de la forme $x_1 - \dots - x_r$. Notons que $r \geq 2$ puisque le graphe possède au moins deux sommets et qu'il est connexe. x_1 ne peut être voisin d'aucun des x_i avec $3 \leq i \leq r$ (sinon on aurait un cycle); de plus, x_1 ne peut avoir de voisin n'apparaissant pas dans le chemin, puisqu'on pourrait alors le prolonger en un chemin élémentaire strictement plus long. Donc x_2 est le seul voisin de x_1 , et x_1 est donc une feuille. Le même raisonnement s'applique à x_r , qui fournit donc une autre feuille (qui est bien distincte puisque $r \geq 2$).

2. a. Dans l'arbre enraciné, la racine a un degré entrant nul et un degré sortant égal à k , les autres nœuds internes ont un degré entrant égal à 1 et un degré sortant égal à k , les feuilles ont un degré entrant égal à 1 et un degré sortant nul. En oubliant l'orientation, il est donc nécessaire d'avoir :
 - un unique nœud de degré k ;
 - tous les autres nœuds de degré 1 ou $k + 1$.

Inversement, si l'on prend un arbre satisfaisant cette condition et qu'on l'enracine en le nœud de degré k , chaque nœud autre que la racine a un degré entrant égal à 1, ce qui laisse bien un degré sortant égal à k ou 0.

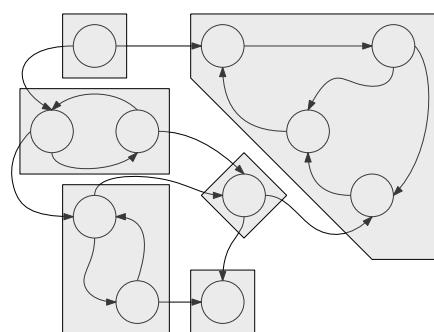
- b. Notons f le nombre de feuilles et n le nombre de nœuds internes. En considérant l'arbre enraciné :
 - chaque nœud autre que la racine est extrémité droite d'exactement un arc, donc $|E| = n + f - 1$ (somme des degrés entrants);
 - chaque nœud interne est extrémité gauche d'exactement k arcs, donc $|E| = kn$ (somme des degrés sortants).

Ainsi, on a $kn = n + f - 1$, d'où l'on déduit immédiatement $f = (k - 1)n + 1$.

Remarque

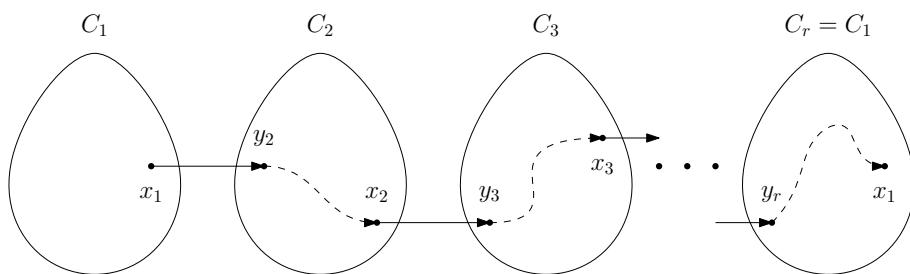
Dans le cas des arbres binaires, on retrouve la formule bien connue $f = n + 1$.

Correction de l'exercice 14.18 page 281



Correction de l'exercice 14.19 page 282

Supposons qu'on ait un cycle $C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_r = C_1$. Pour $1 \leq i < r$, il existe alors $(x_i, y_{i+1}) \in C_i \times C_{i+1}$ tels que $x_i \rightarrow y_{i+1}$. De plus, comme x_{i+1} et y_{i+1} sont dans la même CFC C_{i+1} , il existe un chemin $y_{i+1} \xrightarrow{*} x_{i+1}$ pour chaque $i \in \{1, \dots, n-1\}$. On obtient donc $x_1 \rightarrow y_2 \xrightarrow{*} x_2 \rightarrow y_3 \xrightarrow{*} \dots \rightarrow y_r$. Comme de plus $C_1 = C_r$, on peut prolonger par un chemin de y_r à x_1 : ainsi, on a $x_1 \xrightarrow{*} x_2 \xrightarrow{*} x_1$. Cela montre que x_1 et x_2 sont dans la même CFC, ce qui est absurde. Donc G' est bien acyclique.



Correction de l'exercice 14.20 page 282

Soit n l'ordre du graphe. Si $x_1 \rightarrow \dots \rightarrow x_r$ est un chemin dans G avec $r > n$, alors par le principe des tiroirs il existe $i < j$ tels que $x_i = x_j$. Le graphe possède alors un cycle $x_i \rightarrow \dots \rightarrow x_j = x_i$, ce qui est absurde. La longueur des chemins de G est donc majorée (par $n-1$), ce qui permet de considérer un chemin $x_1 \rightarrow \dots \rightarrow x_r$ avec r maximal.

- x_1 est nécessairement une racine, car s'il existait x_0 tel que $x_0 \rightarrow x_1$ alors $x_0 \rightarrow x_1 \dots \rightarrow x_r$ fournirait un chemin strictement plus long.
- De même, x_r est nécessairement une feuille.

Correction de l'exercice 14.21 page 284

1. Choisir un graphe dont les sommets sont étiquetés par $\{1, \dots, n\}$ revient à choisir sa matrice d'adjacence, parmi toutes les matrices (n, n) symétriques, à éléments dans $\{0, 1\}$ et à diagonale nulle. Il y a $\frac{n(n-1)}{2}$ coefficients à choisir (ceux strictement au-dessus de la diagonale), on a donc $2^{\frac{n(n-1)}{2}}$ graphes distincts.
2. Cette fois, il faut choisir p coefficients égaux à 1 parmi les $\frac{n(n-1)}{2}$ coefficients possibles, donc $\binom{n(n-1)/2}{p}$ graphes possibles.
3. Il y a exactement un graphe pour chaque nombre d'arêtes possible :

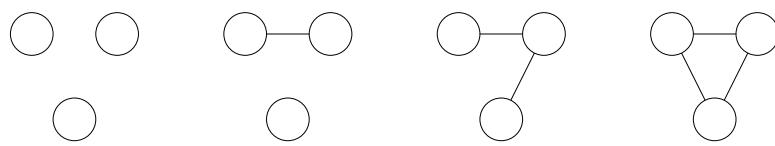


FIGURE 14.38 – Les graphes non étiquetés à 3 sommets.

4. Il suffit d'être méticuleux :

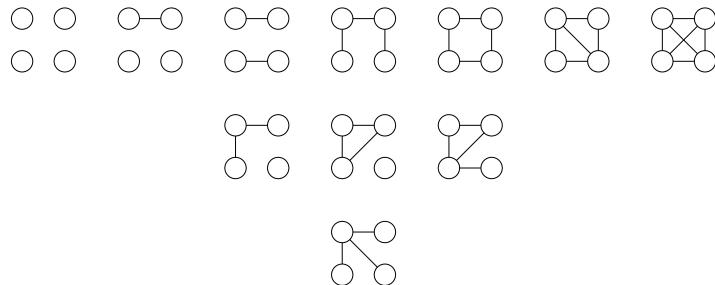


FIGURE 14.39 – Les graphes non étiquetés à 4 sommets.

Correction de l'exercice 14.22 page 284

Notons $d(x)$ le degré du sommet x . Comme chaque arête relie deux sommets, on a $2|E| = \sum_{x \in V} d(x)$. Dans $\mathbb{Z}/2\mathbb{Z}$, on en déduit :

$$\begin{aligned} 0 &\equiv \sum_{x \in V} d(x) \\ &\equiv \overbrace{\sum_{d(x) \text{ pair}}^{=0} d(x)} + \sum_{d(x) \text{ impair}}^{\equiv 1} d(x) \\ &\equiv \text{Card}\{x \in V \mid d(x) \text{ impair}\} \end{aligned}$$

Il y a donc bien un nombre pair de sommets de degré impair.

Correction de l'exercice 14.23 page 284

1. On a immédiatement $A_0 = I_n$ et $A_1 = M$.
2. Un chemin de longueur $n+1$ de i vers j est constitué d'un chemin de longueur n de i à un certain k puis d'un arc (k, j) . Deux choix différents de k , ou deux chemins différents de i à un même k donnent nécessairement deux chemins différents de i à j ; ainsi, on a :

$$\begin{aligned} c_{i,j,n+1} &= \sum_{(k,j) \in E} c_{i,k,n} \\ &= \sum_{k=1}^n c_{i,k,n} c_{k,j,1} \end{aligned}$$

3. Comme $c_{i,k,n} = (A_n)_{i,k}$ et $c_{k,j,1} = (A_1)_{k,j}$, on reconnaît un produit matriciel : $A_{n+1} = A_n \cdot A_1$. Sachant que $A_0 = I_n$ et $A_1 = M$, une récurrence immédiate montre alors que $A_n = M^n$.
4. La question précédente montre que la matrice d'adjacence d'un graphe orienté est nilpotente si et seulement si la longueur des chemins est bornée. De manière presque immédiate, cela est vrai si et seulement si le graphe est acyclique (*cf* exercice 14.20). Ainsi, la matrice d'adjacence d'un graphe orienté est nilpotente si et seulement si il s'agit d'un DAG.

Correction de l'exercice 14.25 page 285

Correction de l'exercice 14.26 page 286

- Notons R_n le nombre d'arbres enracinés de sommets $V = \{1, \dots, n\}$. L'application qui à un arbre t et un sommet i associe l'arbre enraciné en i est clairement une bijection de l'ensemble des arbres de sommets V vers l'ensemble des arbres enracinés de sommets V : on a donc $R_n = nT_n$. D'autre part, il y a $(n-1)!$ manières d'ordonner les arcs d'un arbre enraciné de sommets V , donc $S_n = (n-1)! \cdot R_n$. Finalement, on a bien $S_n = n! \cdot T_n$.
- On part d'un graphe entièrement déconnecté (V, \emptyset) et l'on compte le nombre de manières de choisir successivement des arcs orientés (e_1, \dots, e_{n-1}) de façon à obtenir un arbre (enraciné). On oriente les arcs de la racine vers les feuilles.
 - En oubliant l'orientation, on obtiendra un arbre (non enraciné) si et seulement si on ajoute à chaque étape une arête reliant deux sommets situés dans des composantes connexes distinctes. À chaque étape, on aura alors une forêt (dont les arbres sont les composantes connexes du graphe).
 - Pour obtenir un arbre enraciné, il faut avoir à chaque étape une forêt d'arbres enracinés ; pour cela, il faut que chaque arc ajouté relie un nœud à la racine d'un autre arbre (cf figure 14.40). En effet, on aurait sinon un nœud de degré entrant 2, ce qui est interdit.
 - Lorsqu'on choisit le i -ème arc (x, y) à ajouter, on a donc n choix pour x et $n-i$ choix pour y : il y a à cet instant $n+1-i$ racines, et l'une d'entre elles est interdite.

Il y a $n-1$ arcs à choisir ; au total, on a donc $S_n = \prod_{i=1}^{n-1} n(n-i) = n^{n-1}(n-1)!$.

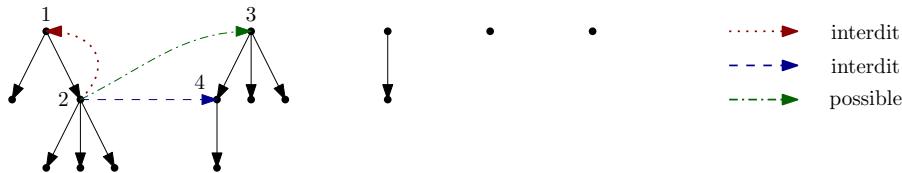


FIGURE 14.40 – On peut ajouter l'arc $2 \rightarrow 3$, mais pas l'arc $2 \rightarrow 1$ (on n'a plus une forêt) ni l'arc $2 \rightarrow 4$ (la forêt obtenue n'est pas correctement enracinée).

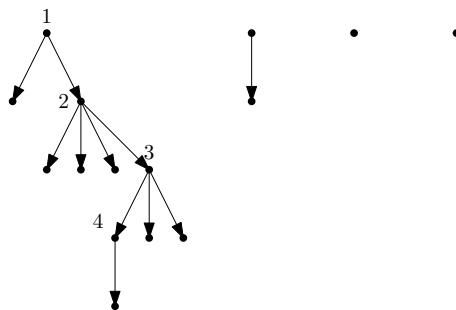


FIGURE 14.41 – La forêt obtenue après l'ajout de l'arc $2 \rightarrow 3$.

- On a donc

$$T_n = \frac{S_n}{n!} = \frac{n^{n-1}(n-1)!}{n!} = \frac{n^{n-1}}{n},$$

et l'on obtient ainsi la *formule de Cayley* :

$$T_n = n^{n-2}.$$

Correction de l'exercice 14.27 page 287

- Supposons $G = (V, E)$ biparti, et soient alors $V = A \sqcup B$ la partition correspondante. Si x_0, \dots, x_n est un chemin de G (de longueur n), alors on a nécessairement $x_{2i} \in A$ pour tout i ou $x_{2i} \in B$ pour tout i . Si ce chemin est un cycle, on a $x_0 = x_n$ et 0 et n doivent donc avoir la même parité, donc n est pair : le graphe ne contient pas de cycle de longueur impaire.
- Inversement, considérons un graphe G sans cycle de longueur impaire, et soit T un arbre couvrant de G (qui existe d'après l'exercice 14.12). On enracine cet arbre en un sommet quelconque r , et l'on définit A comme l'ensemble des sommets de profondeur paire, B comme l'ensemble des sommets de profondeur impaire (où la profondeur de x est la longueur de l'unique chemin élémentaire de r à x). On obtient bien une partition des sommets, il reste à vérifier qu'il n'y a aucune arête entre deux sommets de A (ni entre deux sommets de B). En effet, si $x, y \in A$ et $xy \in E$, alors x, \dots, r, \dots, y, x obtenu en concaténant l'unique chemin élémentaire de x à r , l'unique chemin élémentaire de r à y et l'arête yx serait un cycle de longueur impaire, ce qui est impossible. Donc le graphe est biparti.

Correction de l'exercice 14.28 page 287

Une application $f : X_n \rightarrow X_n$ peut être représentée par un graphe orienté $\varphi(f) = (V, E)$ avec $V = \{0, \dots, n-1\}$ et $(x, y) \in E \Leftrightarrow f(x) = y$; notez qu'on autorise ici les arcs bouclant sur un sommet (ils correspondent aux points fixes de l'application). L'application φ est évidemment injective ; le problème se ramène donc à déterminer $\text{Card } \varphi(A)$, où A désigne l'ensemble des applications nilpotentes.

- Un graphe est l'image d'une application par φ si et seulement si tous ses sommets ont un degré sortant égal à 1. Cela revient à dire que le graphe est constitué d'un certain nombre de cycles disjoints, sur lesquels peuvent venir se greffer des arbres :

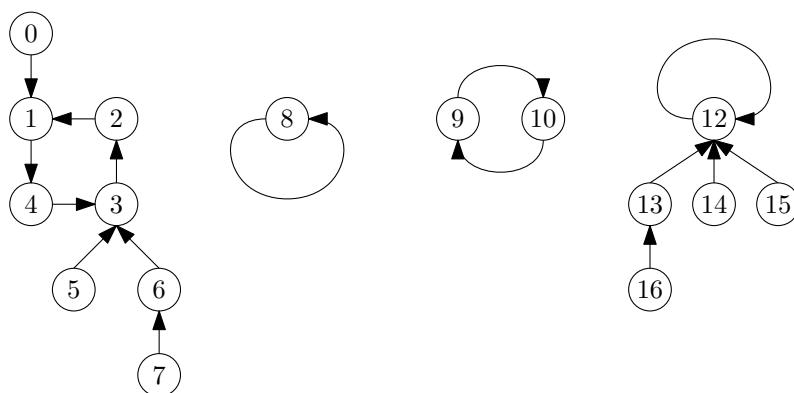


FIGURE 14.42 – Un exemple de graphe d'application (non nilpotente!).

- C'est l'image d'une application nilpotente si et seulement si, de plus, tout chemin suffisamment long se termine en 0. En particulier, cela signifie que le seul cycle est la boucle $0 \rightarrow 0$; si on la supprime, le graphe obtenu est un arbre enraciné en 0. En effet, c'est un DAG dans lequel 0 est accessible depuis tous les sommets. Il suffit donc de montrer qu'il ne possède pas de cycle non orienté : c'est le cas puisqu'un tel cycle x_1, \dots, x_n, x_1 est soit un cycle orienté (impossible puisque c'est un DAG) soit contient une section $x_i \leftarrow x_{i+1} \rightarrow x_{i+2}$ (impossible puisque les degrés sortants valent 1). Inversement, il est immédiat qu'un arbre enraciné en 0 (auquel on ajoute un arc bouclant en 0) est l'image d'une application nilpotente. Ainsi, Le nombre d'applications nilpotentes est égal au nombre d'arbres enracinés en 0, c'est-à-dire au nombre d'arbres étiquetés par $[0 \dots n-1]$: il y en a $T_n = n^{n-2}$.

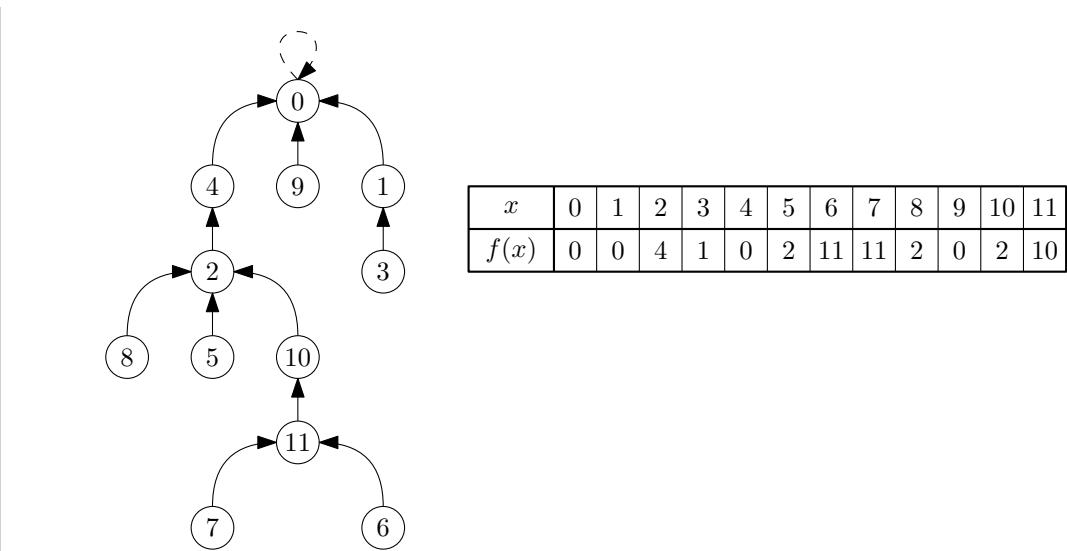


FIGURE 14.43 – Une application nilpotente et l’arbre enraciné correspondant.

ALGORITHMES SUR LES GRAPHES

I Représentation informatique d'un graphe

Définition 15.1 – Taille d'un graphe

On appelle *taille* d'un graphe $G = (V, E)$ la quantité $|V| + |E|$.

Remarques

- Cette quantité n'a aucune signification « mathématique » (on additionne des choux et des carottes...) mais sera est pertinente informatiquement : on dira par exemple qu'un algorithme qui s'exécute en temps $O(|E| + |V|)$ est *linéaire en la taille du graphe*.
- La taille est au plus de l'ordre de $|V|^2$ (cas d'un graphe dense), mais est souvent nettement plus petite (graphe creux).

I.1 Représentation par matrice d'adjacence

La manière la plus simple de représenter un graphe est d'utiliser sa matrice d'adjacence : dans le cas assez fréquent où les étiquettes des nœuds sont les entiers de $[0 \dots n - 1]$, le graphe *est* sa matrice d'adjacence. Dans le cas général où les étiquettes sont de type '`a`' (où '`a`' peut correspondre à une page web, un fichier, une liste d'entiers...), il faudra simplement disposer en plus d'un '`a` **array**' de taille n associant au numéro d'un nœud son étiquette, et éventuellement d'un dictionnaire associant à l'étiquette d'un nœud son numéro.

Avantages

- Le test d'adjacence de deux sommets se fait en temps constant.
- Dans le cas d'un graphe orienté, obtenir les prédécesseurs n'est pas plus compliqué (ni plus coûteux) que d'obtenir les successeurs.
- Rajouter ou supprimer une arête (ou un arc) peut se faire en temps constant.
- Il est possible de n'utiliser qu'un bit par coefficient de la matrice : dans le cas d'un graphe dense, une telle représentation est donc très compacte.

Inconvénients

- Ajouter ou supprimer un sommet nécessite un temps proportionnel à n^2 .
- Pour récupérer les voisins/successeurs d'un nœud, il faut parcourir toute la ligne de la matrice : l'opération se fait donc en $\Theta(n)$, ce qui est regrettable si le degré sortant du nœud est petit devant n .
- On consomme une mémoire proportionnelle à $n^2 = |V|^2$, même quand la taille du graphe est de l'ordre de n (graphe creux).

En résumé, cette représentation est bien adaptée aux graphes denses, mais presque inutilisable pour les graphes creux.

Exemple 15.1

Si l'on considère le graphe d'un gros réseau social, l'ordre de grandeur du nombre de sommets sera sans doute de 10^9 et celui du degré moyen de 10^2 . La matrice d'adjacence aura alors 10^{18} entrées, dont environ 99,999 99% valent zéro : tout stocker n'est pas raisonnable ! Même à raison d'un bit par coefficient, la matrice utiliserait environ 10^{17} octets, c'est-à-dire 100 000 téraoctets.

1.2 Représentation par tableau de listes d'adjacence

Essentiellement, les problèmes liés à la représentation d'un graphe par sa matrice d'adjacence sont simplement ceux liés à la représentation d'une *matrice creuse* (c'est-à-dire majoritairement constituée de zéros). La solution la plus classique à ce problème est de stocker la ligne i de la matrice sous la forme d'une liste $[(j_1, x_1), \dots, (j_k, x_k)]$ où les j_k sont les indices tels que m_{i,j_k} soit non nul, et les x_k les valeurs correspondantes. Ici, la seule valeur non nulle possible est 1 et il est donc inutile de stocker les x_k : on se retrouve simplement avec la liste des j_k , où les j_k sont les numéros des voisins/successeurs du noeud numéro i .

Un graphe G à n sommets sera donc stocké sous la forme d'un **int list array** de longueur n , et G sera non orienté si et seulement si $i \in t(j) \Rightarrow j \in t(i)$ pour tous $i, j \in [0 \dots n]$.

Avantages

- La mémoire utilisée est de l'ordre de $|E| + |V|$, ce qui est nettement mieux que $|V|^2$ si le graphe est creux.
- On a directement accès à la liste des voisins d'un noeud : la parcourir prend un temps proportionnel au degré sortant du noeud (et pas à $|V|$).
- Ajouter un noeud peut normalement se faire en temps $|V|$ (si l'ajout n'impose pas de renommer les noeuds déjà présents).

Inconvénients

- Si l'on a besoin d'un accès en temps raisonnable aux prédecesseurs d'un noeud (dans le cas orienté), il faut stocker séparément le tableau de listes correspondant.
- Le test d'adjacence ne se fait plus en temps constant (mais en temps proportionnel au degré du noeud).
- Si le graphe est dense, on consommera plus de mémoire (d'un facteur constant) qu'avec une matrice d'adjacence.
- Ajouter ou supprimer une arête n'est pas aussi évident que dans une matrice d'adjacence : suivant l'opération précise que l'on souhaite faire, la complexité peut être unitaire ou proportionnelle aux degrés des noeuds impactés.
- Supprimer un noeud n'est pas pratique : le plus simple est de reconstruire entièrement le graphe (en un temps $\Theta(|E| + |V|)$).

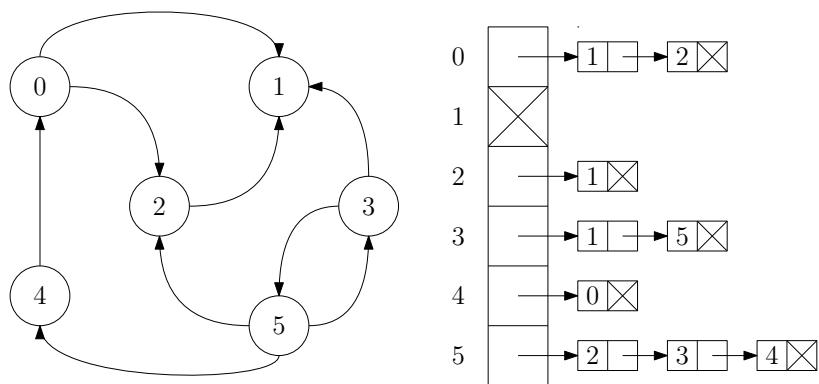


FIGURE 15.1 – Un graphe orienté et le tableau de listes d'adjacence correspondant.

Remarques

- C'est cette représentation que nous utiliserons le plus souvent, et c'est aussi celle avec laquelle sont habituellement formulés les algorithmes agissant sur les graphes.
- On peut tout à fait remplacer les listes chaînées d'adjacence par des tableaux (éventuellement dynamiques si l'on souhaite pouvoir ajouter des arêtes). En C, où les listes chaînées ne sont pas très naturelles, c'est généralement ce que l'on fera.

Exercice 15.2 – Changement de représentation

p. 321

1. Écrire une fonction prenant en entrée un graphe représenté sous forme d'une matrice d'adjacence et renvoyant une représentation du même graphe sous forme de tableau de listes d'adjacence. On fera en sorte que les listes d'adjacence soient triées par ordre croissant.
2. Écrire une fonction réalisant la transformation inverse.

```
listes_of_matrice : bool array array -> int list array
matrice_of_listes : int list array -> bool array array
```

Exercice 15.3 – Utilisation de dictionnaires

p. 321

On suppose que l'on dispose d'une structure de dictionnaire qui fournit les opérations suivantes :

- `Dict.get` : ('k, v) dict -> 'k -> 'v option;
- `Dict.set` : ('k, v) dict -> 'k -> 'v -> unit (ajout ou remplacement);
- `Dict.remove` : ('k, v) dict -> 'k -> unit;
- `Dict.iter` : ('k, v) dict -> ('k -> 'v -> unit) -> unit.

On suppose de plus que `Dict.get` et `Dict.set` sont en temps constant, et que l'appel `Dict.iter` dict f a un coût proportionnel à la somme des appels f k v engendrés.

1. Proposer une structure de données permettant de représenter un graphe orienté de manière à fournir les opérations suivantes :
 - test d'existence d'un arc entre deux sommets en temps constant;
 - ajout et suppression d'un arc en temps constant;
 - ajout d'un sommet en temps constant;
 - itération sur les successeurs d'un sommet en temps proportionnel à son degré sortant;
 - itération sur tous les sommets du graphe en temps proportionnel à leur nombre.

On supposera que les sommets sont connus par un identifiant entier, mais que ces identifiants ne sont pas nécessairement consécutifs (on peut avoir un graphe à trois sommets numérotés 12, 3524 et 123456).

2. Cette structure permet-elle de supprimer un sommet facilement ? Si ce n'est pas le cas, proposer une modification.

1.3 Graphes implicites

Le plus souvent, il n'est pas nécessaire de stocker explicitement la structure du graphe : il suffit d'être capable de générer la liste des voisins d'un nœud donné à la demande. De cette manière, les algorithmes présentés dans ce chapitre peuvent être étendus au cas de graphes infinis (ou trop grand pour être stockés) – avec toutefois quelques subtilités supplémentaires.

Exemple 15.4

Les configurations possibles d'un *Rubik's Cube* forment un graphe à environ $4.3 \cdot 10^{19}$ sommets, qu'il serait pour le moins malaisé de stocker explicitement. Cependant, chaque sommet de ce graphe est de degré 12 (s'il on considère qu'un mouvement élémentaire consiste en la rotation de l'une des 6 faces d'un quart de tour, dans un sens ou dans l'autre), et il n'est pas difficile, à partir de l'étiquette d'une configuration^a, de calculer les étiquettes de ses voisins. Ainsi, il est tout-à-fait possible d'opérer effectivement sur ce graphe (par exemple pour chercher le nombre minimal de coups nécessaire à la « résolution » d'une configuration initiale donnée).

^a. Les configurations ont un étiquetage naturel par les permutations de [1...54].

Dans le pseudo-code de ce chapitre, on supposera seulement que l'on dispose, pour un graphe G :

- d'un moyen d'itérer sur les voisins (ou successeurs) d'un sommet x , en un temps proportionnel au degré (sortant);
- d'un moyen d'itérer sur l'ensemble des sommets de G , en un temps proportionnel au nombre de sommets;
- d'un moyen de représenter un ensemble de n sommets de G en espace $\mathcal{O}(n)$, de manière à disposer d'un test d'appartenance efficace ($\mathcal{O}(1)$, ou à la limite $\mathcal{O}(\log n)$).

2 Parcours d'un graphe

Parcourir un graphe est une tâche assez similaire au parcours d'un arbre, avec quelques différences importantes :

- il n'y a pas de « racine » : en règle générale, on parcourt à partir d'un sommet x et l'on ne parcourra bien sûr que les sommets accessibles à partir de x ;
- le plus important : *a priori*, il y a des cycles, et il ne faut pas tourner en rond ! D'une manière ou d'une autre, il faut donc se souvenir des sommets que l'on a déjà visités¹.

2.1 Parcours en profondeur

Essentiellement, on adapte le parcours en profondeur d'un arbre en rajoutant un test pour détecter les noeuds déjà visités. On maintient donc un ensemble vus contenant les noeuds que l'on a déjà vus.

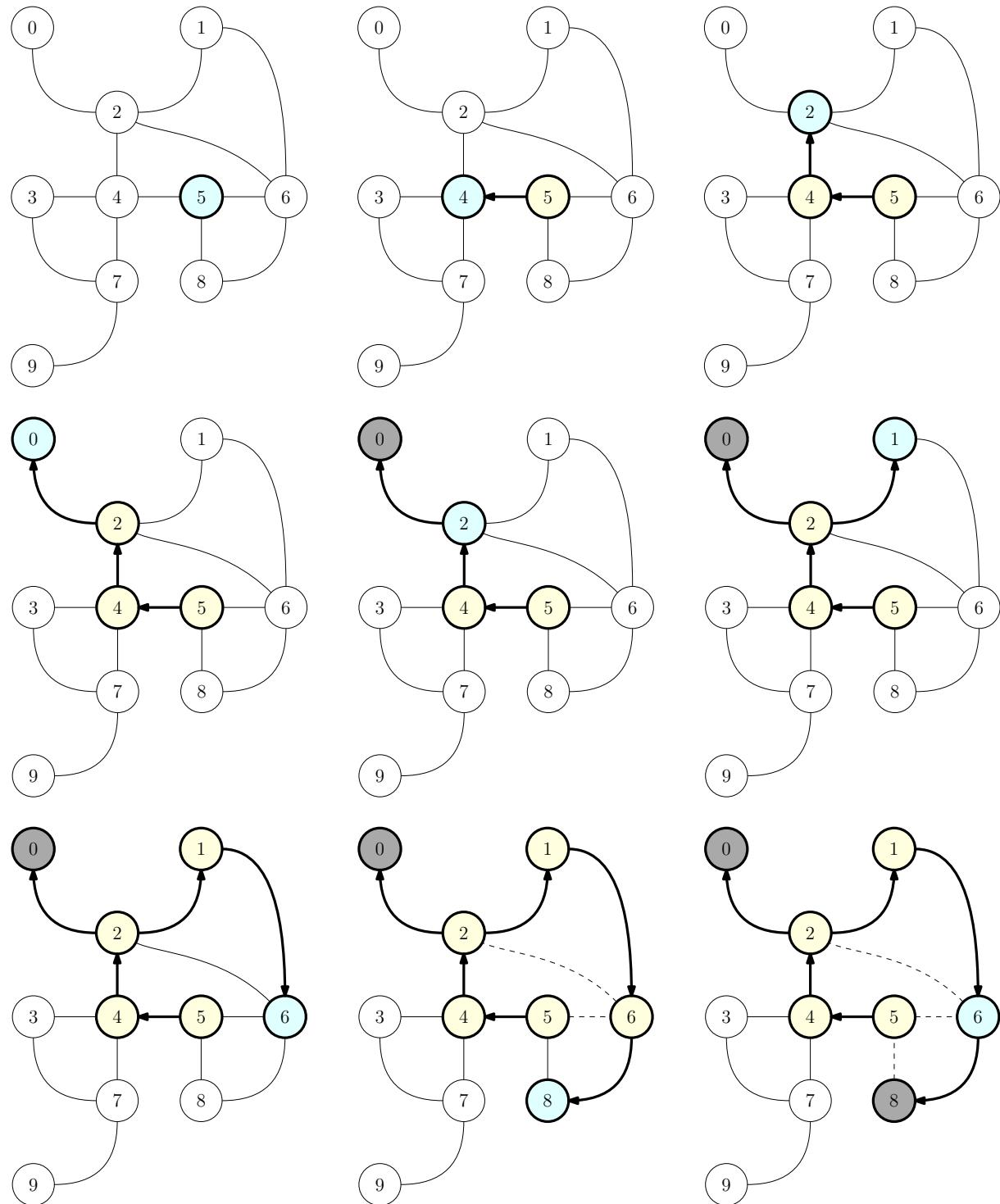
Algorithme 6 Parcours en profondeur (*Depth First Search*).

<pre> 1: fonction DFS(G, v) 2: vus ← ∅ 3: fonction EXPLORER(x) 4: si $x \notin vus$ alors 5: vus ← vus ∪ {x} 6: pré-traitement(x) 7: pour $y \in \text{successeurs}(x)$ faire 8: EXPLORER(y) 9: post-traitement(x) 10: EXPLORER(v) </pre>	<pre> 1: fonction DFS-COMPLET(G) 2: vus ← ∅ 3: fonction EXPLORER(x) 4: si $x \notin vus$ alors 5: vus ← vus ∪ {x} 6: pré-traitement(x) 7: pour $y \in \text{successeurs}(x)$ faire 8: EXPLORER(y) 9: post-traitement(x) 10: pour $v \in V$ faire 11: EXPLORER(v) </pre>
--	--

Remarques

- Dans la version DFS-COMPLET, il est indispensable que tous les appels à EXPLORER partagent le même ensemble vus .
- Il est possible de remplacer le test « $si x \notin vus$ » par un test « $si y \notin vus$ » avant l'appel EXPLORER(y). Attention dans ce cas à modifier en conséquence la boucle principale de la fonction DFS-COMPLET.
- Si le graphe est non orienté, on parlera des voisins de x plutôt que des successeurs de x .
- Assez souvent, on souhaite pouvoir arrêter le parcours avant d'avoir exploré tous les noeuds accessibles (dès qu'on atteint un certain noeud, par exemple). Il faudra alors légèrement modifier l'algorithme.
- L'ensemble vus peut être réalisé de plusieurs manières (arbre binaire de recherche, table de hashage...). Le plus simple est d'utiliser un tableau de n booléens (où $n = |V|$) initialisés à **false**².

1. Si le graphe est trop gros, ce n'est pas forcément possible mais nous ignorerons ce problème pour l'instant.
 2. En plus d'être simple, cette solution est efficace, *sauf si l'on ne compte explorer qu'une petite partie du graphe*.



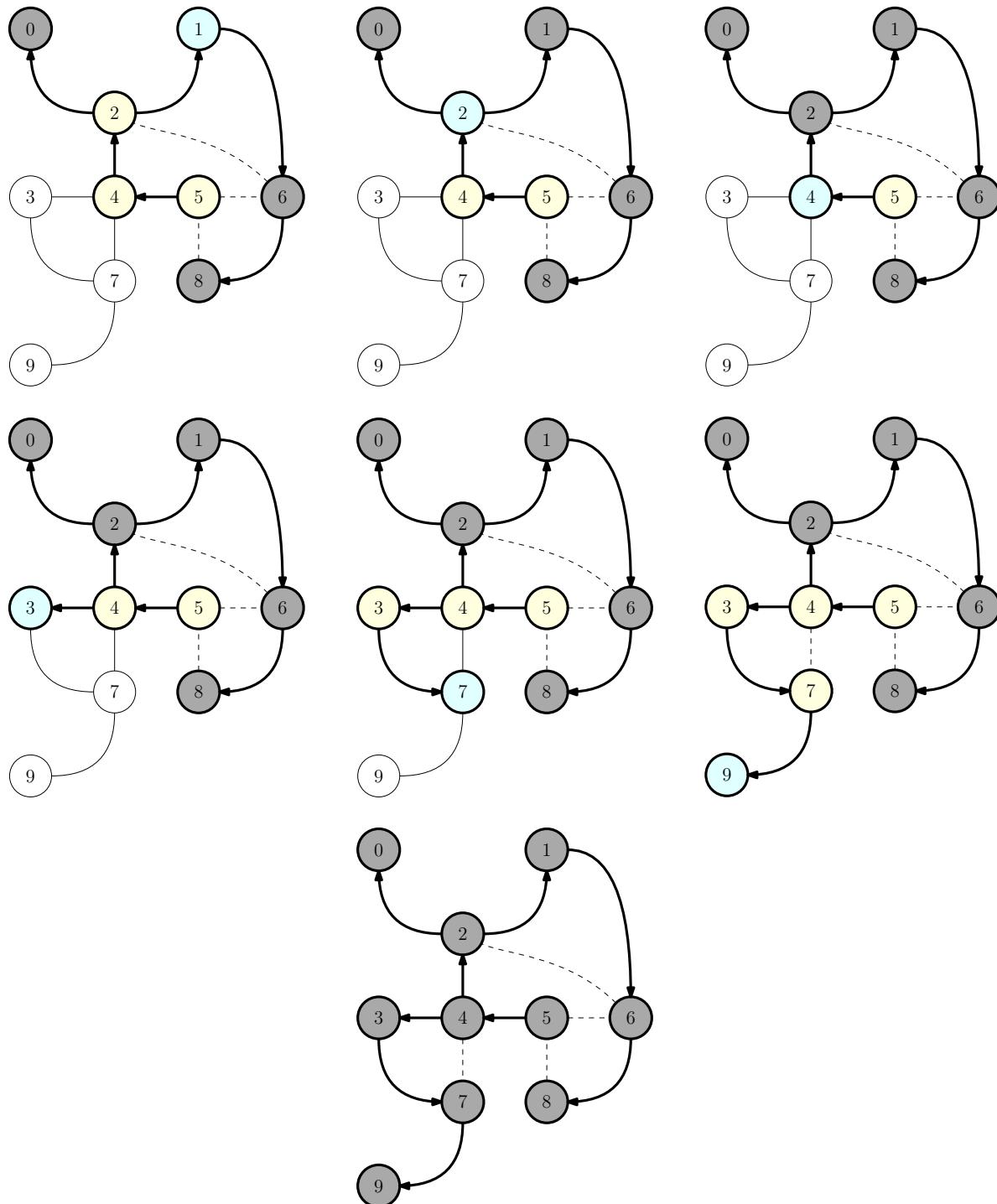


FIGURE 15.2 – Parcours en profondeur d'un graphe connexe non orienté.

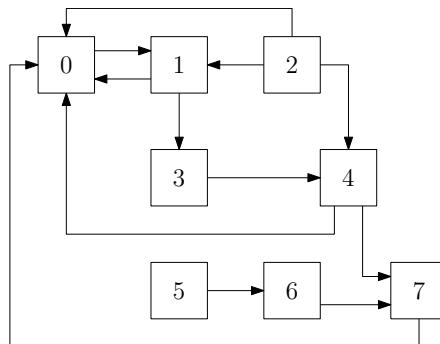
Exercice 15.5

p. 321

- Écrire une fonction `affiche_dfs` qui effectue un parcours en profondeur complet du graphe passé en argument (donné sous forme de tableau de listes d'adjacence), dans l'ordre croissant des numéros de sommets. On affichera `Ouverture i` quand on commence à explorer le sommet `i` et `Fermeture i` quand on termine l'exploration.

```
affiche_dfs : int list array -> unit
```

2. Simuler l'exécution de cette fonction sur le graphe ci-dessous, en supposant que les listes d'adjacence sont triées par ordre croissant.



Theorème 15.2 – Analyse du parcours en profondeur

On suppose dans cette analyse que l'ensemble vus est réalisé par un tableau de booléens et que l'on peut itérer sur les successeurs de x en temps proportionnel au nombre de successeurs

- La fonction DFS appelée sur un graphe fini G et un sommet v termine après avoir visité exactement les nœuds de G accessibles depuis v . Les fonctions *pré-traitement* et *post-traitement* sont appelées exactement une fois sur chacun de ces nœuds.
- La fonction DFS-COMPLET termine après avoir visité tous les nœuds de G . Les fonctions *pré-traitement* et *post-traitement* sont appelées exactement une fois par nœud.
- La complexité temporelle de DFS-COMPLET est $\Theta(|V| + |E|)$.
- La complexité spatiale de DFS-COMPLET est $\Theta(|V|)$ sur le tas et $\mathcal{O}(|V|)$ sur la pile.

Remarques

- Pour le cas non orienté, l'ensemble des nœuds visités est exactement la composante connexe de u .
- L'utilisation d'un espace en $\mathcal{O}(|V|)$ sur la pile peut être problématique si $|V|$ est grand. Une version itérative (ou récursive terminale) efficace est présentée à l'exercice 15.23, et d'autres variantes moins satisfaisantes dans le TP associé à ce chapitre.

Démonstration

- L'ensemble vus et le test associé assure qu'un sommet sera traité au plus une fois.
- Si x est visité, alors la pile d'appels $v \rightarrow \dots \rightarrow explore\ x$ fournit un chemin (élémentaire) de v à x .
- Inversement, supposons qu'il existe un chemin $v = x_1, \dots, x_n = x$ reliant v à x mais que x ne soit jamais visité. Soit alors i le premier indice tel que x_i ne soit pas visité. On a forcément $i > 1$ car v est visité, et donc x_{i-1} a été visité. C'est absurde : lors de cette visite, x_i (successeur de x_{i-1}) n'était pas dans vus et aurait donc dû être exploré.
- On suppose ici que vus est un tableau de $|V|$ booléens. Son initialisation (ligne 2) prend donc un temps $\Theta(|V|)$ et les tests d'appartenance se font en temps $\Theta(1)$. Quand on appelle DFS-COMPLET, la fonction EXPLORE est appelée exactement une fois sur chaque sommet. Or un appel à EXPLORE(x) prend un temps $\Theta(1 + |\text{succ}(x)|)$ (sans tenir compte des appels récursifs). Au total, la complexité temporelle est donc :

$$\Theta(|V|) + \sum_{x \in V} \Theta(1 + |\text{succ}(x)|) = \Theta(|V|) + \Theta(|V|) + \Theta(|E|) = \Theta(|V| + |E|)$$

- Pour la complexité spatiale, il y a deux choses à considérer :
 - le tableau vus prend un espace $\Theta(|V|)$ (sur le tas);
 - l'espace sur la pile est proportionnel à la longueur maximale des chemins explorés. Ces chemins étant élémentaires, on peut la majorer par $|V|$: on obtient donc une consommation mémoire en $\mathcal{O}(|V|)$.

Exercice 15.6

p. 322

Donner trois familles de graphes E_n, T_n, P_n à n sommets pour lesquels le facteur limitant dans l'exécution de DFS-COMPLET sera respectivement :

1. l'espace ;
2. le temps ;
3. l'espace sur la pile.

Théorème 15.3 – Arbre de parcours en profondeur d'un graphe non orienté

Soit $G = (V, E)$ un graphe non orienté et $u \in V$. À un parcours en profondeur de G à partir de u , on peut associer un graphe orienté $T = (V', E')$ défini par :

- V' est l'ensemble des sommets visités par le parcours (i.e V' est la composante connexe de u) ;
- $(x, y) \in E'$ ssi le sommet y a été exploré à partir du sommet x .

On a alors :

- T est un arbre enraciné en u ;
- si xy est une arête de G qui n'apparaît pas dans T , alors x est un ancêtre de y dans T (ou inversement).

T est appelé *arbre de parcours en profondeur à partir de u* .

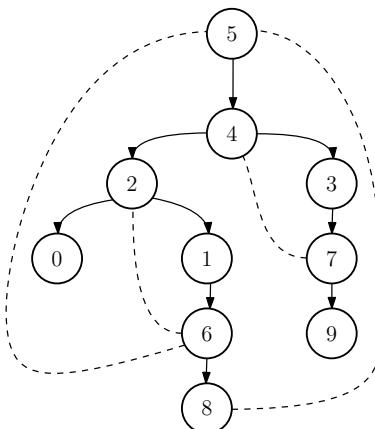


FIGURE 15.3 – Arbre associé au parcours en profondeur de la figure 15.2. Les arêtes en pointillé sont les arêtes du graphe initial qui ne sont pas conservées dans l'arbre.

Remarques

- Si l'on change l'ordre d'exploration des successeurs (qui n'est pas fixé par l'algorithme DFS), on change complètement l'arbre obtenu (hauteur, arité des nœuds internes, nombre de feuilles...). La propriété énoncée dans le théorème n'est bien sûr pas affectée.
- Si l'on fait un parcours en profondeur complet (fonction DFS-COMPLET), on obtiendra une forêt avec exactement un arbre par composante connexe.
- La situation est plus complexe pour un graphe orienté : nous traiterons ce cas en exercice.

Exercice 15.7

Dessiner l'arbre de parcours en profondeur obtenu pour le graphe de la figure 15.2 si l'on explore à partir du sommet 3.

2.2 Parcours en largeur

Définition 15.4 – Distance dans un graphe non pondéré

Soit G un graphe, éventuellement orienté mais non pondéré. La distance $d(x, y)$ d'un sommet x à un sommet y est la longueur minimale, en nombre d'arêtes, d'un chemin reliant x à y . Si un tel chemin n'existe pas, $d(x, y) = \infty$.

Remarques

- Comme vu au chapitre précédent, s'il existe un chemin de x à y , alors il existe un chemin élémentaire de x à y , et un plus court chemin est nécessairement élémentaire. Par conséquent, la définition précise choisie pour la notion de chemin n'influe pas sur la définition de la distance.
- Si G est connexe et non orienté, alors d est bien une distance au sens mathématique usuel.
- Si G est non orienté et non connexe, d est essentiellement une distance, mais prend ses valeurs dans $\mathbb{R}_+ \cup \{\infty\}$.
- Si G est orienté, d n'est plus symétrique : ce n'est donc pas une distance. En revanche, l'inégalité triangulaire est toujours vérifiée (et on a bien sûr $d(x, y) \geq 0$ avec égalité ssi $x = y$).

Propriété 15.5

S'il existe un arc yy' , alors pour tout sommet x on a $d(x, y') \leq d(x, y) + 1$.

Le *parcours en largeur* d'un graphe à partir d'un sommet v permet de visiter les sommets par distance croissante à v :

Algorithme 7 Parcours en largeur (*Breadth-First Search*) à l'aide d'une file.

```

1: fonction BFS( $G, x_0$ )
2:   ouverts ← file_vide()
3:   PUSH( $x_0$ , ouverts)
4:   vus ← { $x_0$ }
5:   tant que ouverts n'est pas vide faire
6:      $x$  ← POP(ouverts)                                ▷ On extrait l'élément de tête
7:     TRAITEMENT( $x$ )
8:     pour  $y \in G.\text{successeurs}(x)$  faire
9:       si  $y \notin vus$  alors
10:        PUSH( $y$ , ouverts)                            ▷ On ajoute  $y$  en queue.
11:        vus ← vus ∪ { $y$ }
```

Remarques

- Le fait que *ouverts* soit une *file* (structure FIFO) est crucial !
- On pourrait bien sûr définir une fonction *BFS-COMPLET* comme plus haut, mais elle serait assez peu utile : le parcours en largeur n'est en règle général intéressant qu'à partir d'un certain noeud distingué.
- L'appel à *TRAITEMENT* pourrait être fait au moment où l'on pousse le noeud sur la file sans impacter la propriété fondamentale (les noeuds sont traités par distance croissante à x_0).

Théorème 15.6 – Propriété fondamentale du parcours en largeur

Un appel à $\text{BFS}(G, x_0)$, où x_0 est un sommet du graphe fini G , termine après avoir visité tous les sommets accessibles depuis x_0 une et une seule fois. Les visites de ces sommets se font par distance croissante à x_0 .

Ainsi, si $d(x_0, x) < d(x_0, y) < \infty$, alors *TRAITEMENT*(x) sera exécuté avant *TRAITEMENT*(y).

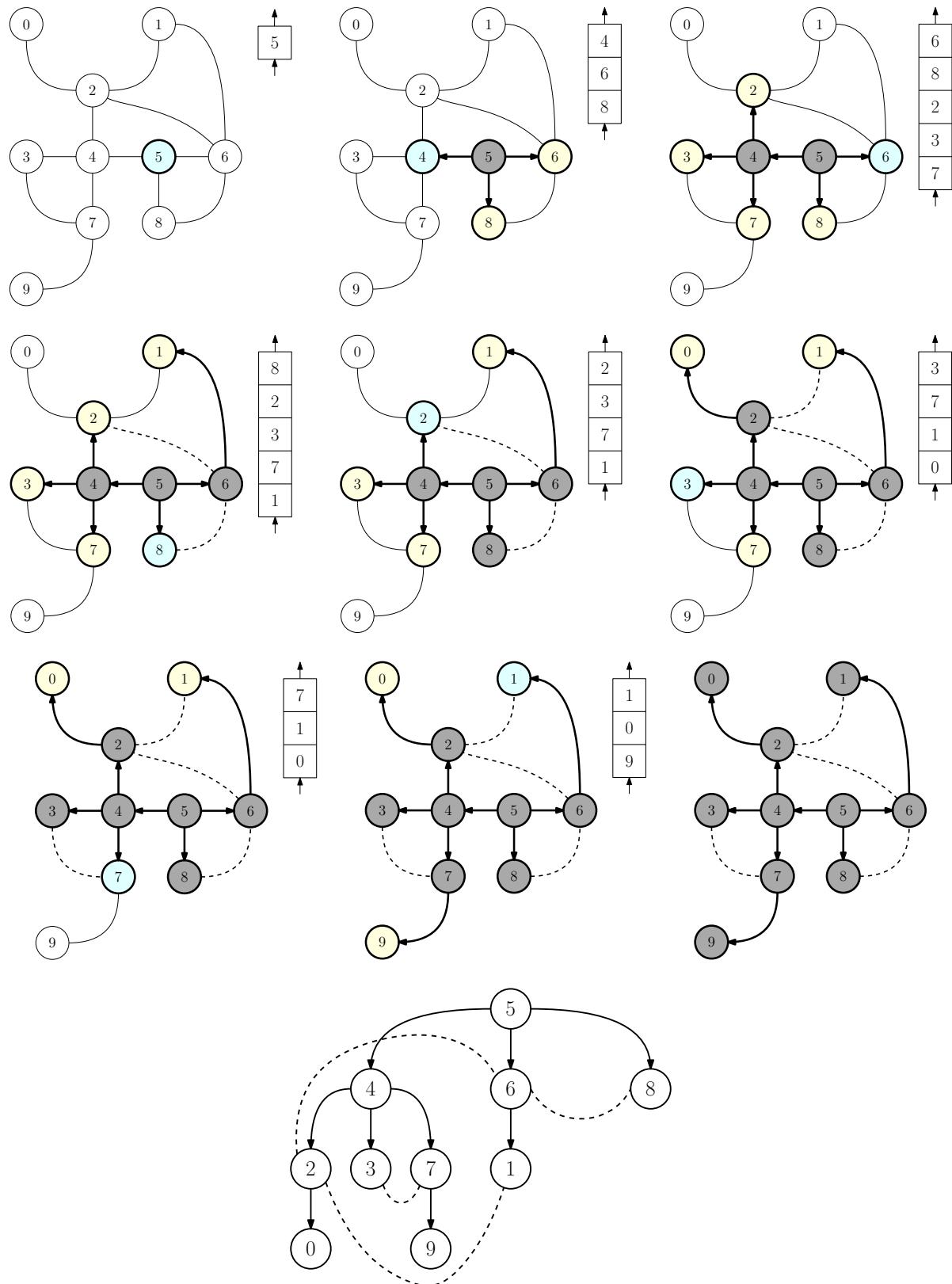


FIGURE 15.4 – Parcours en largeur d'un graphe non orienté.

Démonstration

Quelques observations pour commencer :

- un sommet est ajouté au plus une fois à la file (G étant fini, cela garantit la terminaison), et tout sommet ajouté finit par être traité;
- on adapte facilement la preuve faite pour le parcours en profondeur pour montrer que les sommets visités sont exactement ceux accessibles depuis x_0 ;
- l'ordre de traitement des sommets est le même que l'ordre dans lequel ils sont ajoutés à la file.

À un instant donné, un sommet sera dit :

- *ouvert* s'il appartient à *ouverts*³;
- *fermé* s'il appartient à *vus* mais pas à *ouverts*;
- *vierge* sinon (il n'y a que trois cas car *ouverts* ⊂ *vus*).

On numérote les sommets dans l'ordre de leur ouverture, x_0, \dots, x_n , et l'on note $d_k := d(x_0, x_k)$. L'invariant, valable à la fermeture de x_k , est le suivant :

- (1) la file a la forme x_{k+1}, \dots, x_{k+p} avec $d_k \leq d_{k+1} \leq \dots \leq d_{k+p} \leq d_k + 1$ (un nombre éventuellement nul de sommets à distance d_k suivis d'un nombre éventuellement nul de sommets à distance $d_k + 1$);
- (2) si $d_i < d_k$, alors x_i est fermé (*i.e* $i < k$);
- (3) si $d_i = d_k$, alors x_i n'est pas vierge.

On passe à la preuve :

Initialisation Quand on ferme x_0 , la file est vide ; de plus, aucun i ne vérifie $d_i < d_0$ et seul $i = 0$ vérifie $d_i = d_0$, donc l'invariant est vérifié.

Invariance On suppose l'invariant vérifié à l'étape $k < n$, on ferme x_k et l'on ajoute ses successeurs vierges y_1, \dots, y_l sur la file. Comme les y_i sont vierges, on a d'après l'invariant $d(x_0, y_i) > d_k$. Comme de plus $d(x_0, y_i) \leq d_k + 1$ d'après la proposition 15.5, on a en fait $d(x_0, y_i) = d_k + 1$: la forme de la file est préservée.

On distingue maintenant deux cas :

- si $d_{k+1} = d_k$, il n'y a rien de plus à prouver ;
- si $d_{k+1} = d_k + 1$, alors tous les sommets à distance d_k sont fermés, puisqu'il ne peut en rester sur la file et que l'invariant garantit qu'aucun n'est vierge. Cela montre le point (2) de l'invariant. De plus, comme tous ces sommets sont fermés, aucun de leurs successeurs ne peut être vierge ; comme tout sommet à distance $d_k + 1$ est successeur d'un sommet à distance d_k , cela montre le point (3) de l'invariant.

Conclusion L'invariant est donc vérifié pour tout k , et son point (2) garantit donc que si $d_i < d_k$, alors $i < k$: c'est la conclusion cherchée. ■

Exercice 15.8

p. 322

1. Écrire une fonction `bfs` qui effectue un parcours en largeur d'un graphe à partir d'un sommet x_0 fourni en argument. Cette fonction affichera les sommets du graphe par distance croissante à x_0 . On supposera que le graphe est donné sous forme d'un tableau de listes d'adjacence, et l'on pourra utiliser le module `Queue` pour réaliser la file.

- `Queue.create : unit -> 'a Queue.t` crée une file vide.
- `Queue.is_empty : 'a Queue.t` teste si une file est vide.
- `Queue.push : 'a -> 'a Queue.t -> unit` ajoute un élément à la file.
- `Queue.pop : 'a Queue.t -> 'a` extrait un élément de la file (qui doit être non vide).

```
bfs : int list array -> int -> unit
```

2. Simuler l'exécution de cet algorithme sur le graphe de la figure 15.2 à partir du sommet 0, et dessiner l'arbre de parcours correspondant.

3. Wow !

3. Si l'on ne souhaite pas utiliser le module **Queue**, comment peut-on réaliser de manière efficace une file impérative? fonctionnelle? On ne demande pas d'implémenter ces structures mais simplement de se remémorer les techniques que nous avons vues.

Théorème 15.7 – Complexité du parcours en largeur

Le parcours en larguer a une complexité spatiale en $\Theta(|V|)$. En supposant que les opérations élémentaires sur les files et les ensembles se font en temps constant, sa complexité temporelle est en $\mathcal{O}(|E| + |V|)$.

Démonstration

Si l'on réalise *vus* par un tableau de $|V|$ booléens, l'initialisation se fait en temps $\Theta(|V|)$. Ensuite, le traitement de chaque nœud prend un temps proportionnel à son nombre de successeurs (plus une constante). Ainsi, le temps total est proportionnel à la taille (nombre d'arêtes plus nombre de sommets) du sous-graphe accessible depuis le sommet initial. Cette taille est bien en $\mathcal{O}(|E| + |V|)$, donc la complexité temporelle totale est en $\mathcal{O}(|E| + |V|)$.

Pour l'espace, on consomme $\Theta(|V|)$ pour *vus* et $\mathcal{O}(|V|)$ sur la file (puisque tous les sommets présents sur la file sont distincts). Au total, la complexité spatiale est donc en $\Theta(|V|)$. ■

Exercice 15.9 – Parcours en largeurs avec générations explicites

p. 323

On considère l'algorithme suivant :

Algorithme 8 Parcours en largeur avec générations explicites.

```

fonction BFS(G, x0)
    vus ← {x0}
    actuels ← [x0]                                ▷ Ensemble
                                                    ▷ Liste
    tant que actuels ≠ ∅ faire
        nouveaux ← []
        pour x ∈ actuel faire
            Traiter x.
            pour y ∈ G.successeurs(x) faire
                si y ∉ vus alors
                    Ajouter y à nouveaux.
                    Ajouter y à vus.
            actuels ← nouveaux
```

- Justifier que cet algorithme termine.
- Énoncer un invariant permettant de montrer que cet algorithme effectue bien un parcours en largeur (qu'il traite exactement les sommets accessibles depuis x_0 , par distance croissante à x_0).
- Rédiger la preuve de correction de l'algorithme.
- Écrire en OCaml une fonction **tableau_distances** prenant en entrée un graphe sous forme d'un tableau de listes d'adjacences et un sommet x_0 , et utilisant l'algorithme ci-dessus pour calculer un tableau **dist** tel que **dist**.(i) soit la distance entre x_0 et le sommet i. On mettra une valeur de -1 dans le tableau pour les sommets qui ne sont pas accessibles depuis x_0 .

```
tableau_distances : int list array -> int -> int array
```

3 Test d'acyclicité et tri topologique

3.1 Ordre d'exploration lors d'un parcours en profondeur

Lors d'un parcours en profondeur, on dit qu'un sommet est :

- *vierge* si l'on n'a pas commencé à l'explorer;
- *ouvert* si son exploration a commencé mais n'est pas encore terminée;
- *fermé* si son exploration est terminée.

Pour un sommet x , on définit $\text{pre}(x)$ et $\text{post}(x)$ comme les instants d'ouverture et de fermeture de x comme dans le code suivant :

```

1 type statut = Vierge | Ouvert | Fermé
2
3 let dfs g =
4   let n = Array.length g in
5   let vus = Array.make n Vierge in
6   let pre = Array.make n (-1) in
7   let post = Array.make n (-1) in
8   let compteur = ref 0 in
9   let rec explore x =
10    if vus.(x) = Vierge then begin
11      vus.(x) <- Ouvert;
12      pre.(x) <- !compteur;
13      incr compteur;
14      List.iter explore g.(x);
15      vus.(x) <- Fermé;
16      incr compteur
17    end in
18  for i = 0 to n - 1 do
19    explore i
20  done;
21  vus, pre, post

```

Propriété 15.8

Si y est ouvert pendant l'exploration de x , c'est-à-dire si $\text{pre}(x) < \text{pre}(y) < \text{post}(x)$, alors :

- y est accessible depuis x ;
- $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

De même, si $\text{pre}(x) < \text{post}(y) < \text{post}(x)$, on a y accessible depuis x et $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

Exercice 15.10

p. 324

Les propositions suivantes sont-elles correctes ? Donner une démonstration ou un contre-exemple.

1. Si y est accessible depuis x , alors y sera ouvert pendant l'exploration de x .
2. Si y est accessible depuis x et si $\text{pre}(x) < \text{pre}(y)$, alors $\text{pre}(y) < \text{post}(x)$.
3. S'il y a un arc xy et si $\text{pre}(x) < \text{pre}(y)$, alors $\text{pre}(y) < \text{post}(x)$.

3.2 Condition d'acyclicité

Propriété 15.9

Un graphe orienté est acyclique si et seulement si on ne tombe jamais sur un sommet ouvert lors de son parcours en profondeur.

Remarque

Autrement dit, le graphe est acyclique si et seulement si, lors du test ligne 10, le sommet considéré est toujours vierge ou fermé, ou, ce qui revient au même, qu'un sommet x n'est jamais ouvert lorsqu'on appelle `explore x`.

Démonstration

Supposons que x soit ouvert au moment de l'appel `explore x`. Ce n'est pas possible si l'appel vient de la boucle `for` principale, donc `explore x` a forcément été appelé par `explore y` pour un certain y . On a donc :

- un arc yx puisque `explore y` a appelé `explore x`;
- $\text{pre}(x) < \text{post}(y) < \text{post}(x)$ puisque x est ouvert au moment où l'on considère l'arc yx .

Or le deuxième point implique que y est accessible depuis x d'après la proposition `refprop :imbrication-exploration` : on a donc un chemin de x vers y et un arc yx , et donc un cycle.

Supposons maintenant que le graphe possède un cycle. Tous les sommets de ce cycle seront ouverts puis fermés pendant le parcours, on peut donc considérer le dernier sommet x du cycle à être fermé. Notons $y \rightarrow x \rightarrow \dots \rightarrow y$ le cycle, et intéressons-nous à l'état de x au moment où l'on considère l'arc yx :

- x ne peut être fermé puisque y est encore ouvert et que x est le dernier sommet du cycle à être fermé;
- x ne peut être vierge, puisque sinon on ouvrirait x à ce moment et l'on aurait $\text{pre}(y) < \text{pre}(x) < \text{post}(y)$ et donc $\text{pre}(y) < \text{pre}(x) < \text{post}(x) < \text{post}(y)$;
- x est donc ouvert, ce qui achève la démonstration du sens indirect (par contraposée).

Exercice 15.II – Détection de cycle

p. 324

On travaille avec des graphes donnés sous forme d'un tableau de listes d'adjacence :

```
type sommet = int
type graphe = sommet list array
```

1. Écrire une fonction OCaml `est_dag` qui détermine si un graphe orienté est acyclique.

```
est_dag : graphe -> bool
```

2. Écrire une fonction `est_foret` qui détermine si un graphe non orienté est acyclique.

```
est_foret : graphe -> bool
```

Remarque

Il n'est pas possible d'écrire une fonction traitant correctement les deux cas.

3. **Optionnel.** Écrire une fonction `cycle` prenant en entrée un graphe orienté et renvoyant :

- **None** si le graphe ne possède pas de cycle;
- **Some u**, où u est une liste de sommets du graphe formant un cycle orienté, si le graphe possède un cycle.

```
cycle : graphe -> sommet list option
```

3.3 Tri topologique

Le tri topologique d'un graphe orienté (donné sous forme d'un tableau de listes d'adjacence) peut être réalisé en temps linéaire en la taille du graphe par l'algorithme ci-dessous :

Algorithme 9 Tri topologique d'un graphe orienté acyclique

Entrées : G est un graphe orienté

Sorties : Une liste tri constituant un tri topologique des sommets de G , ou une erreur si G n'est pas un DAG.

```

fonction TRI_TOPOLOGIQUE( $G$ )
    état  $\leftarrow$  (vierge, ..., vierge)                                 $\triangleright$  Tableau de taille  $n$ 
    tri  $\leftarrow \emptyset$                                           $\triangleright$  Liste
    fonction EXPLORE( $v$ )
        si état[ $v$ ] = ouvert alors
            Erreur : le graphe possède un cycle.
        sinon si état[ $v$ ] = vierge alors
            état[ $v$ ]  $\leftarrow$  ouvert
            pour  $v' \in$  successeurs( $v$ ) faire
                EXPLORE( $v'$ )
            état[ $v$ ]  $\leftarrow$  fermé
            tri  $\leftarrow v, tri$ 
        pour  $v \in V$  faire
            EXPLORE( $v$ )
        renvoyer tri
    
```

Démonstration

La détection de cycle fonctionne comme à l'exercice 15.11, on peut donc supposer que le graphe est acyclique et que l'algorithme renvoie une liste tri. Le fait que chaque sommet de G apparaisse exactement une fois dans tri découle directement de la correction de l'algorithme de parcours en profondeur. Il reste donc à prouver que, pour tout sommet x , si l'arc xy existe, alors x est avant y dans tri. C'est bien le cas, puisque :

- si $\text{pre}(x) < \text{pre}(y)$, alors on appellera EXPLORE(y) pendant l'appel EXPLORE(x), ce qui aura pour effet d'ajouter y à tri s'il n'y est pas déjà, puis on ajoutera x en tête de tri (et donc devant y);
- si $\text{pre}(y) < \text{pre}(x)$, alors $\text{post}(y) < \text{pre}(x)$ (sinon on aurait un cycle $y \rightarrow \dots \rightarrow x \rightarrow y$), donc y est déjà dans tri au moment où l'on ajoute x en tête.

Exercice 15.12 – Tri topologique en OCaml

p. 326

Écrire une fonction tri_topologique renvoyant un tri topologique du graphe passé en argument, sous la forme d'une liste. On lèvera l'exception Cycle si un tel tri n'existe pas.

```

exception Cycle
tri_topologique : graphe -> sommet list
    
```

4 Plus court chemin dans un graphe pondéré

4.1 Distance, plus court chemin

Définition 15.10 – Graphe pondéré

Un *graphe pondéré* est un triplet $G = (V, E, \rho)$, où V est l'ensemble des sommets, E l'ensemble des arcs (ou des arêtes) et ρ est une application de E dans \mathbb{R} qui à un arc associe son *poids*.

On étend usuellement ρ en posant $\rho(xy) = +\infty$ si $xy \notin E$.

Remarques

- ρ sera souvent à valeurs dans \mathbb{R}_+ , mais pas systématiquement.
- Pour un graphe non orienté, on aura systématiquement $\rho(xy) = \rho(yx)$, mais dans cette partie nous verrons essentiellement un graphe non orienté comme un cas particulier de graphe orienté.

Définition 15.11 – Poids d'un chemin

Si $c = x_0, \dots, x_k$ est un chemin, on définit le *poids* de c par :

$$\rho(c) = \sum_{i=0}^{k-1} \rho(x_i x_{i+1}).$$

Autrement dit, le poids d'un chemin est la somme des poids des arcs qui le composent.

Remarque

Il vaut mieux éviter de parler de la longueur d'un chemin dans un graphe pondéré : il y a une ambiguïté entre le nombre d'arcs et le poids total.

Définition 15.12 – Distance dans un graphe pondéré

Si G est un graphe pondéré ne possédant pas de cycle de poids strictement négatif, on définit la *distance* de x à y par :

$$d(x, y) = \inf\{\rho(c) \mid c \text{ chemin de } x \text{ à } y\}$$

Un *plus court chemin* de x à y est un chemin c de x à y vérifiant $\rho(c) = d(x, y)$.

Remarques

- Cette définition a bien un sens, puisque la condition sur l'absence de cycle de poids négatif permet de se limiter aux chemins élémentaires (à partir d'un chemin quelconque de x à y de poids p , on peut toujours obtenir un chemin élémentaire de x à y de poids $p' \leq p$ en supprimant les cycles). Les chemins élémentaires étant en nombre fini (ils possèdent au plus $n - 1$ arêtes), on a en fait $d(x, y) = \min\{\rho(c) \mid c \text{ chemin élémentaire de } x \text{ à } y\}$.
- On peut ici avoir une distance négative, puisqu'on n'exige pas que les arcs soient à poids positif.

4.2 Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall permet de calculer $d(x, y)$ pour tous les couples (x, y) d'un graphe pondéré G sans cycle de poids négatif. Il s'agit d'un algorithme de programmation dynamique, qui utilise les deux observations suivantes pour faire apparaître une sous-structure optimale exploitable :

- si $c : x = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k = y$ est un plus court chemin de x à y , alors on peut supposer que les v_i sont distincts ;
- sous les mêmes hypothèses, $v_0 \rightarrow \dots \rightarrow v_i$ et $v_i \rightarrow \dots \rightarrow v_k$ sont des plus courts chemins, respectivement de v_0 à v_i et de v_i à v_k .

Propriété 15.13

On suppose donnée une numérotation x_0, \dots, x_{n-1} des sommets du graphe. Pour $0 \leq k \leq n$, on définit $d_k(x_i, x_j)$ comme le poids minimum d'un chemin de x_i à x_j dont tous les sommets (sauf éventuellement les extrémités) appartiennent à $\{x_0, \dots, x_{k-1}\}$. On pose $d_k(x_i, x_j) = +\infty$ si un tel chemin n'existe pas.

On a alors :

$$\begin{cases} d_0(x_i, x_j) = \rho(x_i x_j) \\ d_{k+1}(x_i, x_j) = \min(d_k(x_i, x_j), d_k(x_i, x_k) + d_k(x_k, x_j)) & \text{si } 0 \leq k < n \end{cases}$$

De plus, $d_n(x_i, x_j) = d(x_i, x_j)$ pour tout couple (x_i, x_j) de sommets.

Démonstration

Notons $E_k = \{x_0, \dots, x_{k-1}\}$ (avec donc $E_0 = \emptyset$) et disons qu'un chemin est *dans* E_k si tous ses sommets, sauf éventuellement ses extrémités, appartiennent à E_k .

- $d_0(x_i, x_j) = \rho(x_i, x_j)$ est immédiat, puisque le seul chemin éventuel dans E_0 de x_i à x_j est celui réduit à l'arc $x_i x_j$.
- Pour l'autre cas, un chemin dans E_{k+1} de x_i à x_j :
 - soit ne passe par x_k , auquel cas c'est un chemin dans E_k , et son poids minimum est $d_k(x_i, x_j)$;
 - soit passe par x_k . Dans ce cas, on peut supposer (à cause de l'absence de cycle de poids négatif, comme dit plus haut) qu'il n'y passe qu'une fois. Il est donc de la forme $x_i \xrightarrow{c} x_k \xrightarrow{c'} x_j$ avec c et c' dans E_k . Son poids minimum est donc $d_k(x_i, x_k) + d_k(x_k, x_j)$.

La dernière remarque découle directement de la définition de $d_n(x_i, x_j)$. ■

La situation est illustrée par la figure ci-dessous (où l'on a représenté x_i et x_j à l'extérieur de E_k , ce qui n'est pas nécessairement le cas).

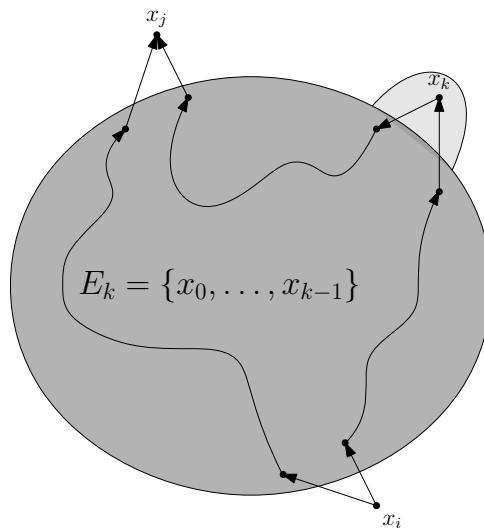


FIGURE 15.5 – Les deux types de chemins à considérer pour le calcul de $d_{k+1}(x_i, x_j)$.

À partir de cette relation de récurrence, on obtient immédiatement un algorithme de programmation dynamique : il suffit de créer $n + 1$ matrices de dimension (n, n) D_0, \dots, D_n , d'initialiser D_0 à partir de la matrice d'adjacence pondérée du graphe, de calculer successivement D_1, \dots, D_n et de renvoyer D_n . On peut tout de suite remarquer que le calcul de D_{k+1} ne fait intervenir que D_k et qu'on peut donc se contenter de stocker deux matrices.

Algorithme 10 Floyd-Warshall, première version

Entrées : La matrice d'adjacence A d'un graphe pondéré à n sommets. $A[i, j] = +\infty$ si l'arc (x_i, x_j) n'existe pas.

Sorties : Une matrice carrée D de taille n tel que $D[i, j] = d(x_i, x_j)$.

```

1: fonction FLOYDWARSHALL( $A$ )
2:   Ancien  $\leftarrow$  copie( $A$ )
3:   pour  $k = 0$  à  $n - 1$  faire  $\triangleright$  Ancien =  $D_k$ 
4:     Nouveau  $\leftarrow$  copie(Ancien)
5:     pour  $i = 0$  à  $n - 1$  faire
6:       pour  $j = 0$  à  $n - 1$  faire
7:         Nouveau[i, j]  $\leftarrow$  min(Ancien[i, j], Ancien[i, k] + Ancien[k, j])
8:     Ancien  $\leftarrow$  Nouveau  $\triangleright$  Ancien =  $D_{k+1}$ 
9:   renvoyer Ancien
  
```

On peut en fait n'utiliser qu'une seule matrice. En effet, juste avant de l'exécution de la ligne 7, on a :

- $\text{Nouveau}[i, j] = \text{Ancien}[i, j] = d_k(x_i, x_j)$ (puisque la case n'a pas encore été mise à jour) ;
- $\text{Nouveau}[i, k]$ qui est égal soit à $d_k(x_i, x_j)$, soit à $d_{k+1}(x_i, x_j)$ (suivant si la case (i, k) a déjà été traitée ou non). Mais de toute façon, un chemin minimal de x_i à x_k dans E_{k+1} est en fait dans E_k (ou en tout cas peut choisi dans E_k) : en effet, il n'y a pas de cycle de poids négatif, donc $\rho(\underbrace{x_i \dots x_k}_{c \in E_k} \dots x_k) \geq \rho(c)$.

On a donc en fait $\text{Nouveau}[i, k] = d_k(x_i, x_k)$ dans tous les cas.

- De même, $\text{Nouveau}[k, j] = d_k(x_k, x_j)$.
- Finalement, la ligne peut être remplacée par $\text{Nouveau}[i, j] \leftarrow \min(\text{Nouveau}[i, j], \text{Nouveau}[i, k] + \text{Nouveau}[k, j])$ sans rien changer.

On en déduit l'algorithme final :

Algorithme 11 Floyd-Warshall, version finale

Entrées : La matrice d'adjacence A d'un graphe pondéré à n sommets. $A[i, j] = +\infty$ si l'arc (x_i, x_j) n'existe pas.

Sorties : Une matrice carrée D de taille n tel que $D[i, j] = d(x_i, x_j)$.

```

1: fonction FLOYDWARSHALL(A)
2:   D ← copie(A)
3:   pour k = 0 à n - 1 faire
4:     pour i = 0 à n - 1 faire
5:       pour j = 0 à n - 1 faire
6:         D[i, j] ← min(D[i, j], D[i, k] + D[k, j])
7:   renvoyer D

```

Propriété 15.14 – Complexité de l'algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall permet d'obtenir la matrice de distance D d'un graphe $G = (V, E)$ en temps $O(|V|^3)$ et en espace $O(|V|^2)$ (espace qui correspond uniquement à la taille du résultat).

Exercice 15.13 – Reconstruction de plus courts chemins

p. 327

Si l'on ne dispose que de la matrice D , il n'y a pas de moyen efficace de reconstruire un plus court chemin entre deux sommets : il faut donc stocker un peu plus d'information.

1. Écrire une fonction `floyd_marshall` prenant en entrée une matrice d'adjacence pondérée (avec des `infinity` pour coder l'absence d'arc) et renvoie :

- une matrice `d` comme plus haut;
- une matrice `prochain`, de même dimension que `d`, telle que :
 - `prochain.(i).(j)` vaut `None` si et seulement si j n'est pas accessible depuis i ;
 - si `prochain.(i).(j)` vaut `Some k`, alors l'un des plus courts chemins de i à j commence par l'arc $i \rightarrow k$ (ou alors $i = j$, auquel cas la valeur de `prochain.(i).(j)` n'a pas d'importance).

```

floyd_marshall : float array array
                  -> (float array array * int option array array)

```

2. Écrire une fonction `reconstruit` prenant en entrée une matrice `prochain` définie comme ci-dessus et deux indices i et j de sommets, et renvoyant un plus court chemin de i vers j sous forme d'une liste d'indices de sommets. On lèvera une exception si j n'est pas accessible depuis i .

```

reconstruit : int option array array -> int -> int -> int list

```

4.3 Algorithme de Dijkstra

L'algorithme de Dijkstra résout un problème similaire à celui de l'algorithme de Floyd-Warshall, avec cependant trois différences :

- on se restreint au cas où les poids sont positifs ;
- il y a un sommet source s distingué, et l'on souhaite calculer $d(s, x)$ pour tous les sommets x ;
- l'algorithme est bien adapté au cas des graphes creux, et prend donc son argument sous forme d'un tableau de listes d'adjacence.

Essentiellement, l'idée est d'adapter le parcours en largeur au cas d'un graphe pondéré. Dans un parcours en largeur, le prochain sommet à explorer est systématiquement celui qui a été découvert en premier : on utilise donc une file. En réalité, ce qui nous intéresse dans ce sommet c'est qu'il s'agit du (ou d'un des) sommet le plus proche du sommet initial parmi ceux qui n'ont pas encore été explorés. Nous allons garder cet ordre d'exploration (par distance croissante au sommet initial) dans l'algorithme de Dijkstra, mais il faudra pour ce faire remplacer la file par une *file de priorité*.

On suppose ici que l'on dispose d'une structure de file de priorité fournissant les opérations suivantes :

- FILEPVIDE() qui renvoie une nouvelle file de priorité ;
- ESTVIDE(f) qui teste si la file f est vide ;
- EXTRAIREMIN(f) qui renvoie un couple (c, p) de f pour lequel la priorité p est minimale, et l'enlève de la file ;
- INSÉRER(f, c, p) qui insère la clé c avec la priorité p dans la file f .
- DIMINUERPRIORITÉ(f, c, p') qui a le comportement suivant :

Préconditions :

- la clé c est présente (une et une seule fois) dans la file f , avec une priorité p ;
- $p' \leq p$.

Effet : après l'appel, la clé c est toujours présente une et une seule fois dans la file, mais sa priorité est désormais p' .

Algorithme 12 Dijkstra

Entrées : un graphe pondéré à poids positifs G à n sommets et un sommet s

Sorties : un tableau $dist$ tel que $dist[k] = d(s, k)$ pour $0 \leq k < n$

```
fonction DIJKSTRA( $G, s$ )
    dist  $\leftarrow (\infty, \dots, \infty)$  ▷ Taille n
    dist[ $s$ ] = 0
    ouverts  $\leftarrow$  FILEPVIDE() ▷ File de priorité min
    INSÉRER(ouverts,  $s, 0$ )
    tant que ouverts  $\neq \emptyset$  faire
         $(j, d_j) \leftarrow$  EXTRAIREMIN(ouverts)
        pour  $k$  successeur de  $j$  faire
             $d \leftarrow d_j + \rho(j \rightarrow k)$ 
            si  $d < dist[k]$  alors
                si  $dist[k] < \infty$  alors
                    DIMINUERPRIORITÉ(ouverts,  $k, d$ )
                sinon
                    INSÉRER(ouverts,  $k, d$ )
                    dist[ $k$ ]  $\leftarrow d$ 
            renvoyer dist
```

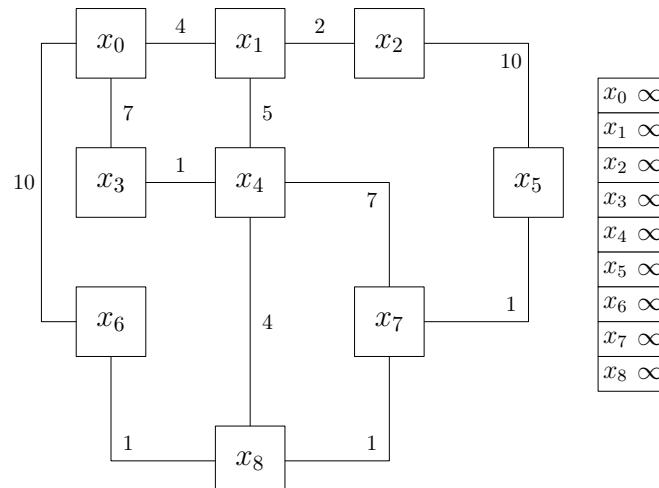
Exercice 15.14 – Terminaison de l'algorithme de Dijkstra	p. 327
---	---------------

Montrer que l'algorithme ci-dessus termine.

Exercice 15.15

p. 327

Simuler à la main l'exécution de l'algorithme de Dijkstra sur le graphe suivant, en prenant $s = x_0$:



Propriété 15.15 – Correction de l'algorithme de Dijkstra

Si $G = (V, E, \rho)$ est un graphe pondéré tel que $\rho(e) \geq 0$ pour tout $e \in E$, alors l'appel `DIJKSTRA(G, s)` renvoie un tableau `dist` tel que $dist[k] = d(s, x_k)$ pour tout $k \in [0 \dots n - 1]$.

Démonstration

On peut déjà remarquer que, si un sommet x apparaît dans `ouverts`, alors $dist[x] < \infty$, et que dans ce cas il y apparaît avec la priorité $dist[x]$ (immédiat à la lecture du code). On dira que :

- un sommet x est *vierge* si $dist[x] = \infty$;
- un sommet est *ouvert* s'il est présent dans la file `ouverts` ;
- un sommet x est *fermé* si $dist[x] < \infty$ et il n'est pas ouvert ;
- un chemin est *fermé* si tous les sommets qui le composent, sauf éventuellement le dernier, sont fermés.

Pour chaque sommet j , on définit :

- $d(j) = d(s, j)$ la longueur d'un plus court chemin de s à j (ou ∞ si un tel chemin n'existe pas) ;
- $d_f(j)$ la longueur d'un plus court chemin fermé de s à j .

On considère à présent l'invariant de boucle suivant (valable à chaque début de boucle, pour tout sommet x) :

- $dist[x] = d_f(x)$;
- si x est fermé, alors $d_f(x) = d(x)$.

Initialisation Au départ, aucun sommet n'est fermé donc $d_f(s) = d(s) = 0$ (chemin vide) et $d_f(j) = \infty$ si $j \neq s$. C'est bien cohérent avec l'état initial de `dist`.

Invariance On suppose l'invariant respecté au début d'une itération, et soit j le sommet choisi (qui est donc ouvert pour l'instant, et que l'on ferme). Pour tout sommet x , on note $d'_f(x)$ et $dist'[x]$ les valeurs en fin d'itération.

- Pour le sommet j , on a $dist'[j] = dist[j] = d_f(j)$ (la première égalité par lecture du code, la seconde d'après l'invariant). Il faut donc prouver $d'_f(j) = d_f(j) = d(j)$. La première égalité est immédiate : le seul sommet supplémentaire autorisé dans $d'_f(j)$ est j lui-même, or on peut se restreindre aux chemins élémentaires puisque les poids sont positifs. De plus, on a $d_f(j) \geq d(j)$ par définition : il suffit donc de prouver $d_f(j) \leq d(j)$. Considérons un chemin c de s à j de poids total l .

— Si les sommets intermédiaires de c sont tous fermés, alors $l \geq d_f(j)$ par définition.

- Sinon, soit x le premier sommet ouvert ou vierge traversé par ce chemin : $\underbrace{\text{init} \rightarrow \cdots \rightarrow x}_{c'} \underbrace{x \rightarrow \cdots \rightarrow j}_{c''}$.

On a $l = p(c') + p(c'') \geq p(c')$ (les poids sont positifs, **hypothèse cruciale**), donc $l \geq d_f(x)$ puisque c' est un chemin fermé de s à x , puis $l \geq d_f(j)$ puisqu'on a extrait le minimum de ouverts.

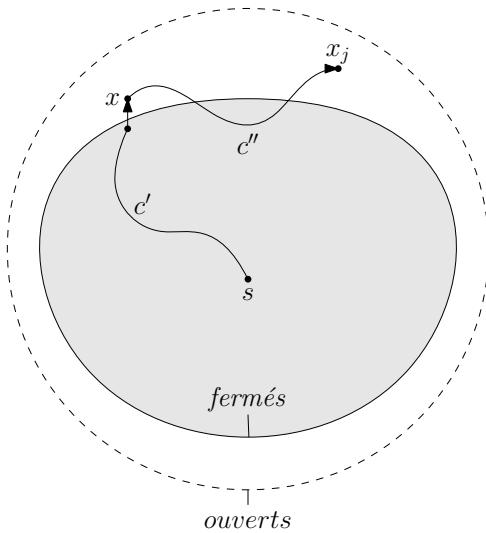


FIGURE 15.7 – Pour le sommet extrait, il existe un plus court chemin qui est fermé.

On a donc bien $d_f(j) = d(j)$.

- Pour les sommets $i \neq j$, les nouveaux chemins à considérer pour $d'_f(i)$ sont les chemins fermés passant par j . Si ce chemin se termine par $x \rightarrow i$ avec $x \neq j$ (et x fermé, nécessairement), alors on peut le remplacer par un chemin au moins aussi léger ne passant pas par j (puisque $d(x) = d_f(x)$). Il suffit donc de considérer les chemins de la forme $s \rightarrow x_1 \rightarrow \cdots \rightarrow x_k \rightarrow j \rightarrow i$ avec les x_i fermés en début d'itération, ce qui est exactement ce que fait le code : un tel chemin est de longueur minimale $d_f(j) + p(j \rightarrow i)$.

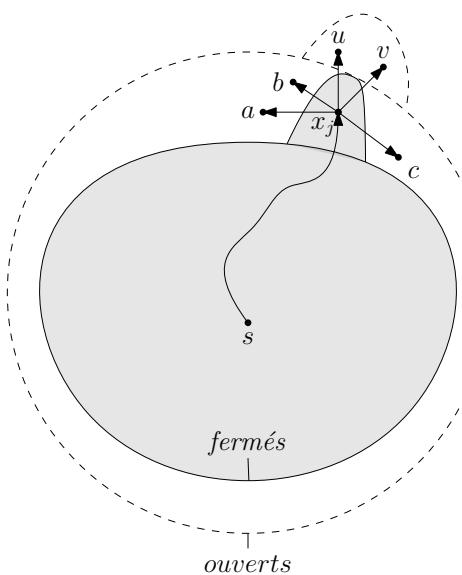


FIGURE 15.8 – Nouveaux chemins à considérer pour d'_f .

Conclusion À la fin de l'exécution, tous les sommets sont fermés donc le tableau dist contient bien les longueurs des plus courts chemins.

Propriété 15.16

Pour un graphe à n sommets et p arcs, la fonction Dijkstra effectue au plus :

- n opérations INSÉRER et n opérations EXTRAIREDIMINUM ;
- p opérations DIMINUERPRIORITÉ.

Toutes ces opérations se font sur une file de taille majorée par n .

La question qui se pose à présent est celle de la réalisation d'une structure de file de priorité permettant une opération DIMINUERPRIORITÉ efficace. On rappelle que la réalisation la plus classique d'une file de priorité est celle qui utilise un tas binaire stocké dans un tableau données. Ici, il suffit de rajouter un deuxième tableau position tel que $\text{position}[i]$ indique l'indice auquel se trouve le couple (i, d_i) associé au sommet i dans le tableau données.

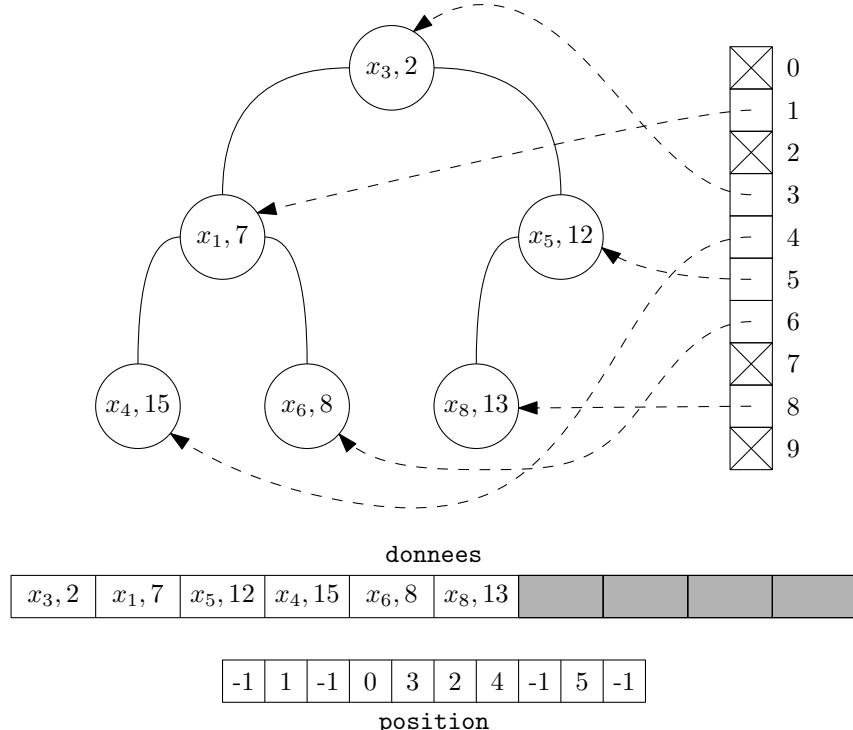


FIGURE 15.9 – Tas binaire avec tableau indiquant la position des clés.

Pour exécuter une opération DIMINUERPRIORITÉ sur le sommet x_i , on fait une percolation vers le haut de la case $\text{position}[i]$ du tableau données. Pendant cette percolation, on effectuera un certain nombre d'échanges dans le tableau données (entre père et fils) : à chaque fois, il faudra faire le même échange dans le tableau position pour maintenir la correspondance.

Exercice 15.16

p. 329

Donner l'état des tableaux données et position de la figure 15.9 après chacune des opérations suivantes (on suppose qu'on les fait successivement) :

- INSÉRER(ouverts, $x_0, 10$) ;
- DIMINUERPRIORITÉ(ouverts, $x_6, 1$) ;
- EXTRAIREDIMINUM(ouverts).

Propriété 15.17

Avec la réalisation décrite ci-dessus pour la structure de file de priorité, la complexité temporelle de l'algorithme de Dijkstra est en $O((n + p) \log n)$.

Exercices

Exercice 15.17

p. 329

1. Dans chacun des cas suivants, la quantité indiquée est-elle la même pour tous les arbres de parcours en largeur à partir d'un sommet x_0 , ou peut-elle dépendre de l'ordre dans lequel les voisins d'un nœud sont examinés ?
 - a. Nombre de nœuds d'arité 1, 2...
 - b. Nombre de feuilles de l'arbre.
 - c. Profondeur du nœud correspondant à un sommet donné.
 - d. Hauteur de l'arbre.
 - e. Nombre de nœuds à chaque niveau de l'arbre.
2. Que peut-on dire des arêtes du graphe qui ne font pas partie de l'arbre ?

Dans tous les exercices qui suivent, on supposera que l'on dispose d'un type `graphe` défini ainsi :

```
type sommet = int
type graphe =
{nb_sommets : int;
 voisins : sommet -> sommet list;
 adjacents : sommet -> sommet -> bool}
```

On supposera de plus que la fonction `voisins` s'exécute en temps constant, ce qui est le cas si le graphe est stocké sous forme d'un tableau de listes d'adjacence :

```
(* of_listes : sommet list array -> graphe *)
let of_listes t =
  let n = Array.length t in
  let v i = t.(i) in
  let adj i j = List.mem j (voisins i) in
  {nb_sommets = n; adjacents = adj; voisins = v}

(* Si g est défini par let g = of_listes t pour un certain t, alors g.voisins
est bien en O(1) (mais pas g.adjacents). *)
```

Exercice 15.18 – Graphes 2-coloriables

p. 329

Une k -coloration d'un graphe non orienté $G = (V, E)$ est une application $\varphi : V \rightarrow [0 \dots k - 1]$ telle que $xy \in E \Rightarrow \varphi(x) \neq \varphi(y)$. Un graphe est dit k -colorable s'il admet une k -coloration. Déterminer si un graphe est k -colorable est difficile (problème NP-complet) dès que $k \geq 3$. En revanche, le problème est très simple pour $k = 2$.

1. Montrer que si $G = (V, E)$ est connexe et $x_0 \in V$, alors il existe au plus une 2-coloration φ de G tel que $\varphi(x_0) = 0$.
2. Écrire une fonction `deux_coloration : graphe -> int array` option qui renvoie `Some t`, où t code une 2-coloration du graphe passé en argument, s'il en existe une, `None` sinon. On exige une complexité linéaire en la taille $|E| + |V|$ du graphe.
Indication : on pourra initialiser t à -1 et utiliser la question précédente.

Exercice 15.19 – Graphe miroir

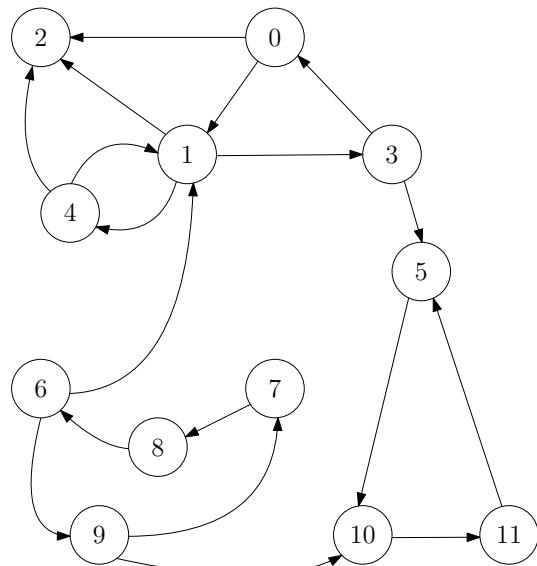
Si $G = (V, E)$ est un graphe orienté, son *graphe miroir* G^\leftarrow est obtenu en gardant le même ensemble de sommets et en inversant le sens de tous les arcs :

$$G^\leftarrow := \left(V, \{(v, u) \mid (u, v) \in E\} \right).$$

Écrire une fonction `miroir` (dont vous devriez pouvoir deviner la spécification). On exige une complexité en $O(n + p)$.

```
miroir : graphe -> graphe
```

```
# miroir g0;;
- : sommet list array =
[| [3]; [6; 4; 0]; [4; 1; 0];
  ↪ [1];
  [1]; [11; 3]; [8]; [9]; [7];
  ↪ [6];
  [9; 5]; [10]|]
```

FIGURE 15.10 – Le graphe g_0

Exercice 15.20 – Test de forte connexité

On considère un graphe orienté $G = (V, E)$, et l'on souhaite déterminer de manière efficace s'il est fortement connexe.

1. Proposer un algorithme naïf pour répondre à la question, et déterminer sa complexité.
2. On note $G^\leftarrow(V, E^\leftarrow)$, le graphe obtenu en changeant le sens de toutes les arêtes de G ; autrement dit, on pose $E^\leftarrow := \{(v, u) \mid (u, v) \in E\}$. Proposer un algorithme répondant au problème posé à l'aide d'un parcours de G et d'un parcours de G^\leftarrow . On justifiera soigneusement sa correction.
3. Déterminer la complexité de l'algorithme.
4. Écrire la fonction `est_fortement_connexe : graphe -> bool`.

Remarque

Le problème de la détermination efficace des composantes fortement connexes est plus délicat. Nous verrons une solution en temps linéaire en la taille du graphe l'année prochaine.

Exercice 15.21 – Théorème de Robbins et détection des ponts

Un graphe non orienté est dit *fortement orientable* s'il existe une orientation de ses arêtes qui le rende fortement connexe. D'autre part, un graphe non orienté connexe est dit 2-arête connexe, ou *sans pont*, s'il n'existe pas d'arête dont la suppression déconnecterait le graphe (une telle arête est appelée *pont*).

1. Montrer qu'une arête est un pont si et seulement si elle ne fait partie d'aucun cycle élémentaire.
2. On considère un graphe connexe G et T un arbre de parcours en profondeur pour G à partir d'un sommet (quelconque) r . On considère l'orientation suivante de G :
 - les arêtes de T sont orientées de la racine vers les feuilles;
 - les autres arêtes, qui relient forcément un nœud et l'un de ses ancêtres dans T (cf. théorème 15.3), sont orientées des feuilles vers la racine.

Montrer que si G est sans pont, cette orientation rend G fortement connexe.

3. En déduire le *théorème de Robbins* (1939) : un graphe est fortement orientable si et seulement si il est sans pont.

Remarque

Autrement dit, il est possible de mettre toutes les rues d'une ville à sens unique (en gardant la possibilité d'aller de n'importe quel point A à n'importe quel point B) si et seulement si on ne peut séparer la ville en deux parties qui ne sont reliées que par une seule rue.

4. On définit :

```
(* Soit un graphe, soit une arête *)
type t = G of graphe | A of int * int
```

Écrire une fonction `orientation_forte` : `graphe -> t` qui prend en entrée un graphe supposé non orienté, et renvoie :

- **G** g' , où g' est une orientation forte de g s'il en existe une;
- **A** (i, j), où ij est un pont dans g , sinon.

On demande une complexité linéaire en la taille du graphe (c'est-à-dire en $\mathcal{O}(|V| + |E|)$).

Exercice 15.22 – Couverture par des tests

On considère la fonction suivante :

```
int count_occurrences(int arr[], int len, int x){
    int count = 0;
    for (int i = 0; i < len; i++) {
        if (arr[i] == x) count++;
    }
    return count;
}
```

1. Dessiner le graphe de contrôle de flot (CFG) associé à cette fonction. On rappelle que ce graphe possède un sommet pour chaque instruction (ou chaque bloc de base, mais ici cela ne changera rien) et un arc du sommet x au sommet y s'il est (syntaxiquement) possible d'exécuter y immédiatement après x .
2. Donner un test ou un jeu de tests permettant d'assurer le critère de *couverture des sommets* : l'exécution du jeu de tests doit passer par tous les sommets du CFG.
3. Proposer une version manifestement fausse de la fonction pour laquelle ce jeu de tests ne détecterait pas d'erreur.
4. Donner un test ou un jeu de tests permettant d'assurer le critère de *couverture des arcs* : l'exécution du jeu de tests doit emprunter tous les arcs du CFG.
5. Proposer une version fausse de la fonction pour laquelle ce nouveau jeu de tests ne détecterait pas d'erreurs.

Exercice 15.23 – Parcours en profondeur itératif efficace

On a vu au TD XXXVIII exercice XXXVIII.8 que la manière la plus simple d'écrire un « vrai » parcours en profondeur (vérifiant le théorème 15.3) itératif avait une complexité spatiale en $\mathcal{O}(|E|)$. Écrire une fonction purement itérative (ou récursive terminale) réalisant un parcours en profondeur, en utilisant l'idée suivante :

- la pile ne contient pas des sommets, mais des *listes de sommets* ;
- un élément de la pile correspond précisément à une *stack frame* du parcours en profondeur récursif : la liste contient les appels récursifs qu'il reste à faire.

En parcourant à partir du sommet 5 dans le graphe de la figure 15.2, les premières étapes

d'évolution de la pile doivent être :

$$(5) \rightarrow (4, 6, 8) \rightarrow \begin{matrix} (2, 3, 5, 7) \\ (6, 8) \end{matrix} \rightarrow \begin{matrix} (0, 1, 4) \\ (3, 7) \\ (6, 8) \end{matrix} \rightarrow \begin{matrix} () \\ (1, 4) \\ (3, 7) \\ (6, 8) \end{matrix} \rightarrow \begin{matrix} (1, 4) \\ (3, 7) \\ (6, 8) \end{matrix} \rightarrow \begin{matrix} (2, 6) \\ (4) \\ (3, 7) \\ (6, 8) \end{matrix} \rightarrow \begin{matrix} (6) \\ (4) \\ (3, 7) \\ (6, 8) \end{matrix} \rightarrow \begin{matrix} (1, 2, 5, 8) \\ () \\ (3, 7) \\ (6, 8) \end{matrix}$$

On justifiera bien que, si le graphe est stocké sous forme de listes d'adjacence, la complexité spatiale est en $\mathcal{O}(|V|)$ (un schéma mémoire peut être utile).

Exercice 15.24 – Dijkstra sans diminution des priorités

p. 332

Proposer une modification de l'algorithme de Dijkstra permettant d'utiliser une file de priorité n'offrant pas l'opération DIMINUERPRIORITÉ. Déterminer la complexité en temps et en espace (dans le pire cas) de cette variante.

Solutions

Correction de l'exercice 15.2 page 297

```
let listes_of_matrice m =
  let n = Array.length m in
  let g = Array.make n [] in
  for i = 0 to n - 1 do
    for j = n - 1 downto 0 do
      if m.(i).(j) then
        g.(i) <- j :: g.(i)
    done;
  done;
g
```

```
let matrice_of_listes g =
  let n = Array.length g in
  let m = Array.make_matrix n n
  in false in
let rec ajoute x voisins =
  match voisins with
  | [] -> ()
  | y :: ys ->
    m.(x).(y) <- true;
    ajoute x ys in
  for i = 0 to n - 1 do
    ajoute i g.(i)
  done;
m
```

Correction de l'exercice 15.3 page 297

1. Une solution est d'utiliser $n + 1$ dictionnaires, où n est le nombres de sommets du graphe :
 - un dictionnaire pour chaque sommet, dont les clés sont les successeurs du sommet et les valeurs les étiquettes des arcs correspondants (ou juste ());
 - un dictionnaire pour le graphe, dont les clés sont les identifiants des sommets et les valeurs les dictionnaires du point précédent.
2. On peut facilement supprimer un sommet et tous les arcs qui en partent en temps constant : il suffit de l'enlever du dictionnaire principal. Le problème est que les arcs pointant vers ce sommet ne sont pas affectés, et que l'on aura une erreur plus tard si on essaie de les suivre. Le plus simple est d'ajouter un dictionnaire prédecesseurs pour chaque sommet.

Correction de l'exercice 15.5 page 300

```
let affiche_dfs g =
  let n = Array.length g in
  let vus = Array.make n false in
  let rec explore x =
    if not vus.(x) then begin
      vus.(x) <- true;
      Printf.printf "Ouverture %d\n" x;
      List.iter explore g.(x);
      Printf.printf "Fermeture %d\n" x
    end in
  for x = 0 to n - 1 do
    explore x
  done
```

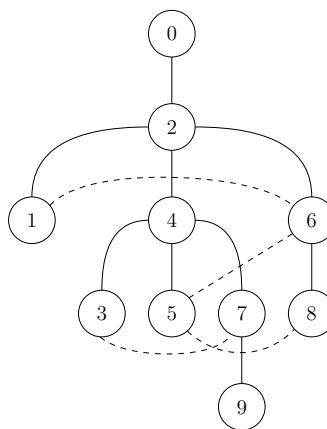
Correction de l'exercice 15.6 page 302

1. Si l'on considère un graphe entièrement déconnecté à n sommets, on aura besoin d'un espace proportionnel à n pour stocker l'ensemble des sommets déjà visités, et le parcours se fera en temps $O(n)$. On remplira rapidement la mémoire disponible, donc le facteur limitant est l'espace.
2. Si l'on prend un graphe complet à n sommets, l'espace nécessaire pour l'ensemble vus est en $O(n)$, tout comme la profondeur de récursion. En revanche, le temps est en $O(n^2)$ puisqu'il faut considérer toutes les arêtes : on sera limité par le temps.
3. Si l'on prend un graphe chemin à n sommets, le temps et l'espace sont en $O(n)$, tout comme l'espace sur la pile. En effet, pour un graphe $x_1 - x_2 - \dots - x_n$ où l'on commence par explorer x_1 , la profondeur de récursion sera de n quand on arrivera à x_n . Ici, le facteur limitant sera l'espace sur la pile d'appel.

Correction de l'exercice 15.8 page 305

1.

```
let bfs g initial =
  let vus = Array.make (Array.length g) false in
  let file = Queue.create () in
  let ajoute x =
    if not vus.(x) then begin
      Printf.printf "%d\n" x;
      vus.(x) <- true;
      Queue.push x file
    end in
  ajoute initial;
  while not (Queue.is_empty file) do
    let x = Queue.pop file in
    List.iter ajoute g.(x);
  done
```

2. On affiche dans l'ordre $0, 2, 1, 4, 6, 3, 7, 5, 8, 9$. L'arbre de parcours correspondant est :

3. Une file impérative peut être réalisée par un tableau circulaire (de taille fixée ou dynamique), par une liste doublement chaînée circulaire, par une liste simplement chaînée mutable pour laquelle on garde un pointeur vers chaque extrémité... Toutes ces variantes offrent des opérations élémentaires en temps constant (amorti dans le cas d'un tableau dynamique). Une file fonctionnelle peut être réalisée de manière efficace par un couple de listes fonctionnelles (temps constant amorti pour l'extraction, temps constant pour l'insertion).

Correction de l'exercice 15.9 page 306

1. Notons $f(p)$ le nombre de sommets du graphe qui ne sont pas dans vus après p passages dans la boucle externe. Ce nombre est décroissant au sens large puisqu'un sommet n'est jamais retiré de vus ; de plus, si $f(p) = f(p + 1)$, alors aucun sommet ne sera ajouté à vus lors de l'itération p , et l'on aura donc $actuels = \emptyset$ au début de l'itération $p + 1$. $f(p)$ est donc une quantité entière, positive, qui décroît strictement tant que l'on ne sort pas de la boucle externe : cela assure la terminaison de cette boucle, et donc de la fonction.
2. L'invariant à considérer est le suivant : « après p passages dans la boucle externe, $actuels$ contient exactement les sommets x tels que $d(x_0, x) = p$, et vus contient exactement les sommets x tels que $d(x_0, x) \leq p$ ».
3. L'invariant est clairement respecté au départ avec $p = 0$: le seul sommet à distance 0 est x_0 , qui est le seul sommet dans $actuels$ et le seul sommet dans vus . Si l'on suppose l'invariant respecté au début d'une itération, et que l'on note $actuels'$, vus' les valeurs en fin d'itération, on a :
 - $actuels' = \{x \in V \setminus vus \mid \exists y \in actuels, y \rightarrow x \in E\}$
 - $vus' = vus \cup actuels'$
 Les sommets de $actuels'$ sont clairement à distance inférieure ou égale à $p + 1$ de x_0 (successeurs de sommets de $actuels$, situés à distance p d'après l'invariant). Ils ne sont pas à distance inférieure ou égale à p puisqu'ils ne sont pas dans vus (invariant). Donc ils sont tous à distance exactement $p + 1$. De plus, un sommet à distance $p + 1$ est nécessairement un successeur d'un sommet à distance p , et il n'est pas dans vus : ainsi, $actuels'$ contient exactement les sommets à distance $p + 1$ du sommet initial. vus' est donc l'union de vus (ensemble des sommets à distance au plus p) et de $actuels'$ (ensemble des sommets à distance $p + 1$) : c'est bien exactement l'ensemble des sommets à distance au plus $p + 1$ du sommet initial. On explore donc les sommets par distance croissante au sommet initial : c'est bien un parcours en largeur.
4. On peut par exemple procéder ainsi :

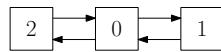
```

let tableau_distances g x0 =
  let n = Array.length g in
  let d = Array.make n (-1) in
  let vus = Array.make n false in
  d.(x0) <- 0;
  vus.(x0) <- true;
  (* ajoute les x non vus de sommets à la liste nouveaux *)
  let rec ajoute_sommets_nouveaux =
    match sommets with
    | [] -> nouveaux
    | x :: xs when not vus.(x) ->
      vus.(x) <- true;
      ajoute_xs (x :: nouveaux)
    | _ :: xs -> ajoute_xs nouveaux in
  (* les sommets de actuels sont à distance p de x0 *)
  let rec loop actuels nouveaux p =
    match actuels, nouveaux with
    | [], [] -> ()
    | [], _ -> loop nouveaux [] (p + 1)
    | x :: xs, _ ->
      d.(x) <- p;
      let nouveaux' = ajoute g.(x) nouveaux in
      loop xs nouveaux' p in
    loop [x0] [] 0;
  d

```

Correction de l'exercice 15.10 page 307

1. C'est faux : si l'on commence l'exploration du graphe $0 \leftrightarrow 1$ en 0, on a 0 accessible depuis 1 et pourtant 0 ne sera pas ouvert pendant l'exploration de 1 (il est déjà ouvert).
2. C'est également faux. On peut considérer le graphe suivant, en commençant l'exploration au sommet 0 :



Le parcours se déroule comme suit :

- ouverture 0;
- ouverture 1;
- fermeture 1;
- ouverture 2;
- fermeture 2;
- fermeture 0.

On a donc $\text{pre}(1) < \text{pre}(2)$ et 2 accessible depuis 1, et pourtant 2 n'est pas ouvert pendant l'exploration de 1.

3. Cette fois c'est vrai. Considérons l'état de y juste avant la fermeture de x :

- y ne peut pas être vierge, puisqu'on a considéré l'arc xy pendant l'appel `explore x` et qu'on aurait alors ouvert y ;
- on a donc $\text{pre}(x) < \text{pre}(y) < \text{post}(x)$, d'où $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$ d'après la proposition 15.8.

Correction de l'exercice 15.11 page 308

1.

```

type statut = Ouvert | Ferme | Vierge

exception Cycle

let est_dag g =
  let n = Array.length g in
  let statuts = Array.make n Vierge in
  let rec explore x =
    match statuts.(x) with
    | Ouvert -> raise Cycle
    | Vierge ->
        statuts.(x) <- Ouvert;
        List.iter explore g.(x);
        statuts.(x) <- Ferme
    | Ferme -> () in
  try
    for i = 0 to n - 1 do
      explore i
    done;
    true
  with
  | Cycle -> false
  
```

2. Pour un graphe non orienté, il ne faut pas considérer l'aller-retour le long d'une arête. On peut écrire :

```

let est_foret g =
  let n = Array.length g in
  let statuts = Array.make n Vierge in
  let rec explore depuis x =
    if x <> depuis then
      match statuts.(x) with
      | Ouvert -> raise Cycle
      | Vierge ->
          statuts.(x) <- Ouvert;
          List.iter (explore x) g.(x);
          statuts.(x) <- Ferme
      | Ferme -> () in
  try
    for i = 0 to n - 1 do
      explore i (-1)
    done;
    true
  with
  | Cycle -> false

```

Une autre solution (plus simple, honnêtement) est de compter le nombre n de sommets, c de composantes connexes et p d'arêtes. Un graphe non orienté est acyclique si et seulement si $c = n - p$.

3. On garde le chemin actuel (qui correspond à la pile d'appel) en mémoire. Si l'on détecte un cycle, ce chemin est de la forme $x \leftarrow y \leftarrow \dots \leftarrow x \leftarrow \dots \leftarrow z$: on extrait la partie $x \leftarrow \dots \leftarrow x$ puis on la remet à l'endroit.

```

exception Cycle of int list

let extraire_cycle chemin =
  let rec aux chemin sommet =
    match chemin with
    | x :: xs when x = sommet -> [x]
    | x :: xs -> x :: aux xs sommet
    | [] -> failwith "mauvais chemin" in
  match chemin with
  | x :: xs -> x :: aux xs x
  | [] -> failwith "mauvais chemin"

```

```

let cycle g =
  let n = Array.length g in
  let statuts = Array.make n Vierge in
  let rec explore chemin x =
    match statuts.(x) with
    | Ouvert -> raise (Cycle (x :: chemin))
    | Vierge ->
        statuts.(x) <- Ouvert;
        List.iter (explore (x :: chemin)) g.(x);
        statuts.(x) <- Ferme
    | Ferme -> () in
  try
    for i = 0 to n - 1 do
      explore [] i
    done;
    None
  with
  | Cycle chemin ->
    let cycle = extraire_cycle chemin in
    Some (List.rev cycle)
  
```

Correction de l'exercice I5.I2 page 309

```

type statut = Ouvert | Ferme | Vierge
exception Cycle

let tri_topologique g =
  let n = Array.length g in
  let marques = Array.make n Vierge in
  let pile = ref [] in
  let rec explore i =
    match marques.(i) with
    | Ouvert -> raise Cycle
    | Ferme -> ()
    | Vierge ->
        marques.(i) <- Ouvert;
        List.iter explore g.(i) ;
        marques.(i) <- Ferme;
        pile := i :: !pile in
  try
    for i = 0 to n - 1 do
      explore i
    done;
    Some !pile
  with
  Cycle -> None
  
```

Correction de l'exercice 15.13 page 312

```

1. let floyd_arbre g =
  let n = Array.length g in
  let dist = Array.make_matrix n n infinity in
  let prochain = Array.make_matrix n n None in
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      dist.(i).(j) <- g.(i).(j);
      if g.(i).(j) <> infinity then prochain.(i).(j) <- Some j
    done
  done;
  for k = 0 to n - 1 do
    for i = 0 to n - 1 do
      for j = 0 to n - 1 do
        if dist.(i).(k) +. dist.(k).(j) < dist.(i).(j) then begin
          dist.(i).(j) <- dist.(i).(k) +. dist.(k).(j);
          prochain.(i).(j) <- prochain.(i).(k)
        end
      done
    done
  done;
  (dist, prochain)

```

2.

```

let rec reconstruit arbre i j =
  if i = j then [i]
  else match arbre.(i).(j) with
    | None -> failwith "inaccessible"
    | Some k -> i :: reconstruit arbre k j

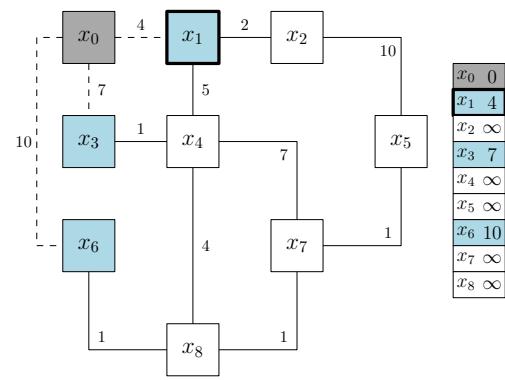
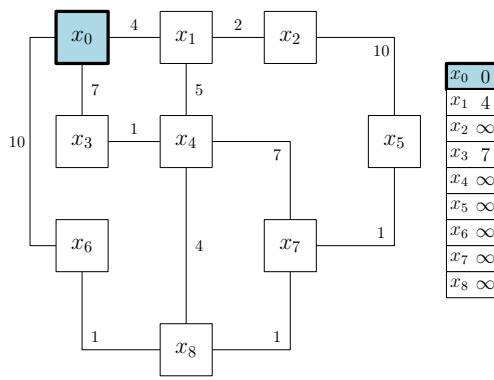
```

Correction de l'exercice 15.14 page 313

Un sommet x est inséré au plus une fois dans la file ouverts (au moment où $\text{dist}[x]$ devient inférieur à ∞), et on retire un sommet de cette file à chaque itération. La boucle s'exécute donc au plus $n = |V|$ fois.

Correction de l'exercice 15.15 page 314

On a représenté en bleu les sommets ouverts, en gris les sommets fermés et en gras le sommet « actif » (le résultat de EXTRAIREDMIN).



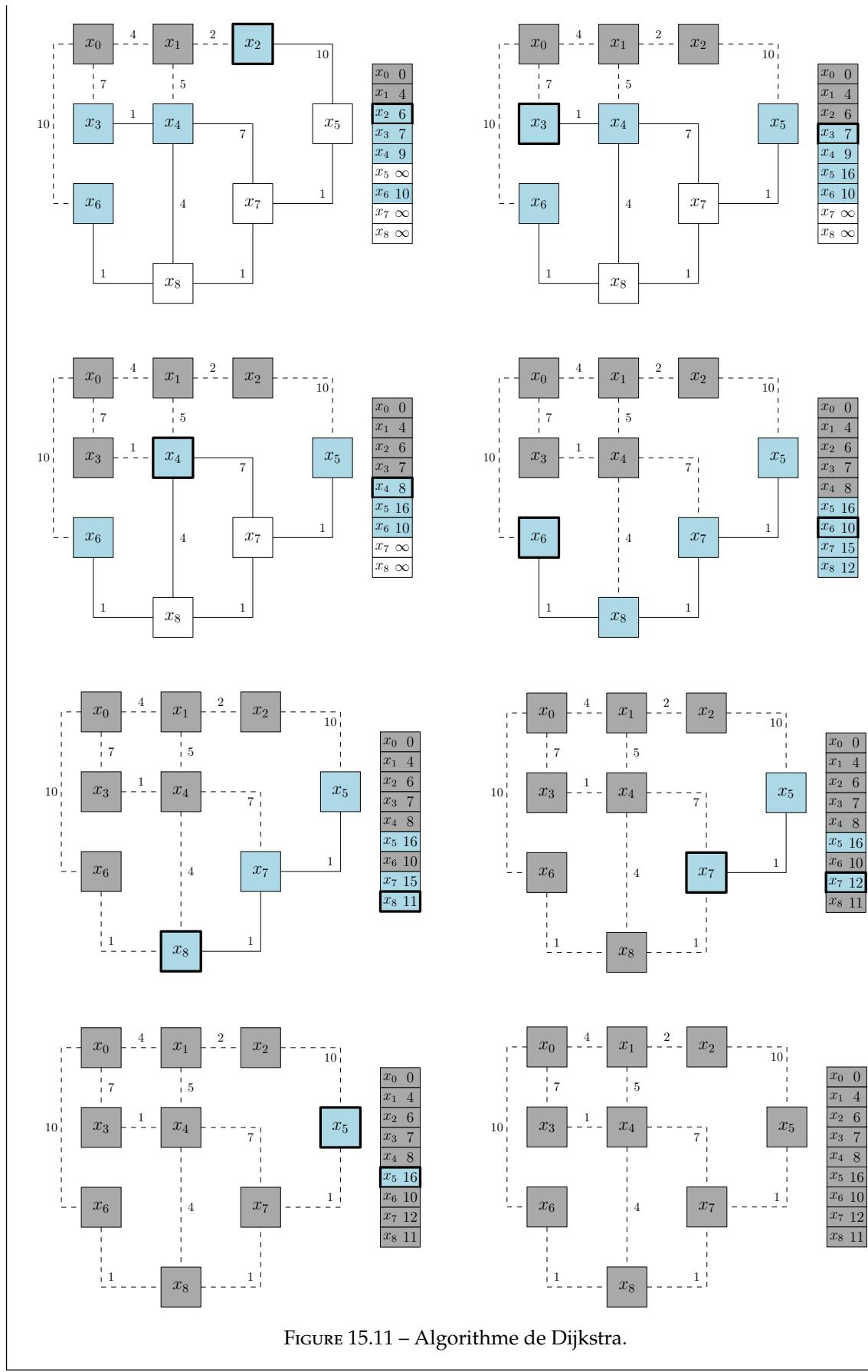


FIGURE 15.11 – Algorithme de Dijkstra.

Correction de l'exercice 15.16 page 316

On obtient successivement :

données							
$x_3, 2$	$x_1, 7$	$x_0, 10$	$x_4, 15$	$x_6, 8$	$x_8, 13$	$x_5, 12$	

2	1	-1	0	3	6	4	-1	5	-1
position									

données							
$x_3, 2$	$x_1, 7$	$x_0, 10$	$x_4, 15$	$x_6, 8$	$x_8, 13$	$x_5, 12$	

2	1	-1	0	3	6	4	-1	5	-1
position									

données							
$x_3, 2$	$x_1, 7$	$x_0, 10$	$x_4, 15$	$x_6, 8$	$x_8, 13$	$x_5, 12$	

2	1	-1	0	3	6	4	-1	5	-1
position									

données							
$x_3, 2$	$x_1, 7$	$x_0, 10$	$x_4, 15$	$x_6, 8$	$x_8, 13$	$x_5, 12$	

2	1	-1	0	3	6	4	-1	5	-1
position									

Correction de l'exercice 15.17 page 317

- La profondeur d'un nœud x est égale à la distance entre x_0 et x pour un arbre de parcours en largeur. Les trois dernières quantités ne dépendent donc pas de l'ordre dans lequel les voisins d'un nœud sont examinés. Les deux premières peuvent varier, en revanche.
- Une arête « manquante » relie forcément deux nœuds de même profondeur ou dont la profondeur diffère de un. Sinon, cette arête donnerait un chemin strictement de la racine à celui des deux nœuds situé le plus bas strictement plus court que celui fourni par l'arbre.

Correction de l'exercice 15.18 page 317

- Supposons que φ soit une coloration de G connexe, et fixons un $x \in G$. Pour tout sommet $y \in G$, il existe un chemin $x = x_1 \dots x_n = y$; mais il est alors immédiat que $\varphi(y) = \varphi(x)$ si et seulement si $n \equiv 1 \pmod{2}$, et $\varphi(y)$ est donc entièrement déterminé par $\varphi(x)$.
- On peut procéder comme suit :
 - pour chaque sommet du graphe, on regarde s'il est déjà colorié ;
 - si c'est le cas, on l'ignore (cela signifie que sa composante connexe a déjà été traitée) ;
 - sinon, on lui attribue arbitrairement la couleur 0 et l'on colorie toute sa composante connexe de la seule manière possible ;
 - si à un moment quelconque on échoue dans la coloration de cette composante connexe, c'est que le graphe n'est pas 2-coloriable ;
 - si l'on arrive à traiter tous les sommets sans échec, on a obtenu une 2-coloration valable.

```

exception Impossible

let deux_coloration g =
  let phi = Array.make g.nb_sommets (-1) in
  let rec colore c v =
    if phi.(v) = -1 then
      begin
        phi.(v) <- c;
        List.iter (colore (1 - c)) (g.voisins v)
      end
    else if phi.(v) <> c then
      raise Impossible in
  try
    for v = 0 to g.nb_sommets - 1 do
      if phi.(v) = -1 then colore 0 v
    done;
    Some phi
  with
  | Impossible -> None

```

Si l'on ne souhaite pas utiliser d'exception, d'autres solutions sont possibles :

```

let deux_coloration_bis g =
  let phi = Array.make g.nb_sommets (-1) in
  let rec colore_sommet c v =
    if phi.(v) = -1 then
      (phi.(v) <- c; colore_voisins (1 - c) (g.voisins v))
    else phi.(v) = c
  and colore_voisins c = function
    | [] -> true
    | v :: vs -> colore_sommet c v && colore_voisins c vs in
  let rec loop i =
    if i = g.nb_sommets then Some phi
    else if phi.(i) <> -1 || colore_sommet 0 i then loop (i + 1)
    else None in
  loop 0

```

```

let deux_coloration_ter g =
  let n = g.nb_sommets in
  let phi = Array.make n (-1) in
  let ok = ref true in
  let rec colore c v =
    if phi.(v) = 1 - c then ok := false
    else if phi.(v) = -1 then
      (phi.(v) <- c; List.iter (colore (1 - c)) (g.voisins v)) in
  let v = ref 0 in
  while !v < n && !ok do
    if phi.(!v) = -1 then colore 0 !v;
    incr v
  done;
  if !ok then Some phi else None

```

Correction de l'exercice 15.19 page 318

```

let miroir g =
  let n = Array.length g in
  let g' = Array.make n [] in
  let rec traite v = function
    | [] -> ()
    | x :: xs -> g'.(x) <- v :: g'.(x); traite v xs in
  for i = 0 to n - 1 do
    traite i g.(i);
  done;
  of_listes g'

```

On peut obtenir un code plus concis avec quelques fonctions d'ordre supérieur, mais cela n'a pas un grand intérêt ici.

Correction de l'exercice 15.20 page 318

1. On peut effectuer $|V|$ parcours en profondeur indépendants, à partir de chaque sommet, et vérifier que l'on atteint bien tous les sommets à chaque fois. Chaque parcours se fait en temps $O(|E| + |V|)$, d'où une complexité en $O(|E| \cdot |V| + |V|^2)$.
2. On choisit un sommet quelconque $x \in G$, puis :
 - on effectue un parcours en profondeur de G depuis x . Si l'on n'atteint pas tous les sommets, G n'est pas fortement connexe, sinon on passe à l'étape suivante;
 - on effectue un parcours en profondeur de G^\leftarrow , toujours depuis x . Si l'on n'atteint pas tous les sommets, G n'est pas fortement connexe. Si l'on atteint tous les sommets, on conclut que G est fortement connexe. En effet, pour tout couple de sommets $(y, z) \in V^2$:
 - il existe un chemin $y \rightarrow \dots \rightarrow x$ d'après la deuxième étape;
 - il existe un chemin $x \rightarrow \dots \rightarrow z$ d'après la première étape;
 - en les concaténant, on obtient un chemin $y \rightarrow \dots \rightarrow z$.

Remarque

on a admis l'équivalence, évidente, entre « x est accessible depuis y dans G » et « y est accessible depuis x dans G^\leftarrow ». Elle se prouve au besoin par récurrence sur la longueur du chemin.

3. Le calcul de G^\leftarrow se fait facilement en temps linéaire en la taille du graphe, tout comme chacun des deux parcours en profondeur. On a donc une complexité temporelle en $O(|V| + |E|)$.
4. On utilise la fonction `miroir` de l'exercice 15.19.

```

let fortement_connexe g =
  let g' = miroir g in
  (* renvoie truessi tous les sommets de g sont accessibles depuis 0 *)
  let tous_accessible g =
    let vus = Array.make g.nb_sommets false in
    let rec explore v =
      if not vus.(v) then begin
        vus.(v) <- true;
        List.iter explore (g.voisins v)
      end in
    explore 0;
    Array.fold_left (&&) true vus in
  tous_accessible g && tous_accessible g'

```

Correction de l'exercice 15.24 page 320

Au lieu de diminuer la priorité d'un sommet, on peut réinsérer le sommet avec une priorité plus faible quand on trouve un chemin plus court. On aura alors potentiellement plusieurs copies d'un même sommet dans la file, avec différentes priorités : ce n'est pas très grave, puisque le résultat d'un `EXTRAIREMINIMUM` sera toujours la copie avec priorité la plus faible. On obtient :

Algorithme 14 Dijkstra sans diminution des priorités

```

fonction DIJKSTRA(G, s)
    dist  $\leftarrow (\infty, \dots, \infty)$  ▷ Taille n
    dist[s] = 0
    ouverts  $\leftarrow$  FILEPRIOVIDE()
    INSÉRER(ouverts, s, 0) ▷ File de priorité min
    tant que ouverts  $\neq \emptyset$  faire
        (j, dj)  $\leftarrow$  EXTRAIREMIN(ouverts)
        si dj < dist[j] alors
            pour k successeur de j faire
                d  $\leftarrow$  dj + ρ(j → k)
                si d < dist[k] alors
                    dist[k]  $\leftarrow$  d
                    INSÉRER(ouverts, k, d)
        renvoyer dist
    
```

Ici, un sommet peut potentiellement être inséré dans la file (puis extrait) une fois par arc entrant : on fait donc jusqu'à p opérations `INSÉRER` et p opérations `EXTRAIREMIN`, et il est possible que la file contienne jusqu'à p éléments. Les opérations étant en temps logarithmique, on obtient du $O(n + p \log p)$ (où $p = |E|$, et le n vient de l'initialisation du tableau `dist`). Mais on sait que $p \leq n^2$, donc $\log p \leq 2 \log n = O(\log n)$, donc en fait la complexité temporelle est en $O(p \log n)$. Il n'y a donc pas vraiment de différence avec la version présentée dans le cours.

Il en va différemment de la complexité spatiale : comme dit plus haut, la file peut contenir jusqu'à p éléments simultanément, et l'on a donc du $O(n + p)$ (le n venant du tableau `dist`). C'est strictement moins bien que la version du cours, qui est en $O(n)$ en espace.

LOGIQUE

I Syntaxe des formules propositionnelles

I.1 Formule propositionnelle

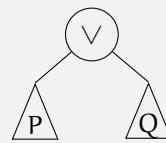
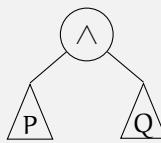
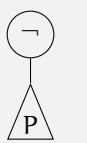
Définition 16.1

Soit \mathcal{V} un ensemble infini dénombrable dont les éléments sont appelés *variables propositionnelles*. Soient les symboles suivants :

- \neg (*non - négation*) : connecteur unary.
- \wedge (*et - conjonction*) : connecteur binaire.
- \vee (*ou - disjonction*) : connecteur binaire.
- \top, \perp (*vrai, faux*) : constantes.

L'ensemble \mathcal{P} des *formules propositionnelles* est défini de manière inductive par :

- $\top, \perp \in \mathcal{P}$.
- $\mathcal{V} \subset \mathcal{P}$.
- si $P, Q \in \mathcal{P}$ alors les arbres suivants sont dans \mathcal{P} :



Remarques

- Les constantes \top et \perp ne feront pas systématiquement partie de la syntaxe.
- Les *variables propositionnelles* (éléments de \mathcal{V}) seront le plus souvent notées $x, y \dots$ ou x_i . Les *formules propositionnelles* seront notées $P, Q \dots$

Le type correspondant en OCaml :

```

type prop =
| Const of bool
| Var of int
| Non of prop
| Et of prop * prop
| Ou of prop * prop
  
```

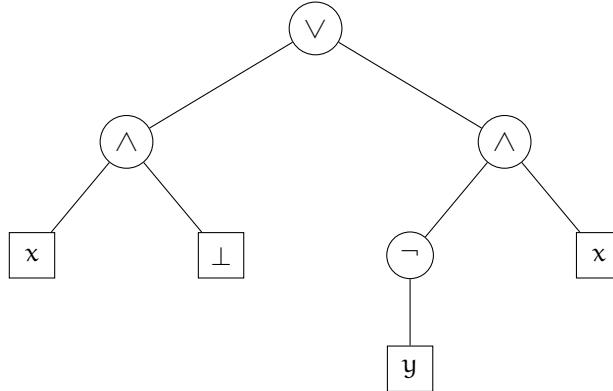
Définition 16.2

Étant donnée une formule propositionnelle P :

- la *hauteur* de P est la hauteur de l'arbre correspondant (zéro si P est réduite à une variable ou une constante);
- la *taille* de P est le nombre de noeuds (internes ou non) de l'arbre;
- une *sous-formule* de P est un sous-arbre, c'est-à-dire l'arbre enraciné en l'un des noeuds de P .

Exemple 16.1

On peut par exemple obtenir l'arbre



qui serait représenté en OCaml par `Ou(Et(Var 0, Const false), Et(Non(Var 1), Var 0))`.

1.2 Notation infixée

Même si une formule est fondamentalement un arbre, il est souvent plus pratique d'utiliser une représentation « à plat » (et nous avons en fait déjà commencé à le faire). Ces notations correspondent aux différents ordres associés au parcours en profondeur de l'arbre (préfixe, infixé, suffixe). Quelques remarques :

- si l'on utilise une notation traditionnelle (infixée), il faut des parenthèses. Pour en limiter le nombre, on se donne les règles de priorité suivantes :
 - \neg est prioritaire sur \wedge et \vee : $\neg A \wedge B$ signifie $(\neg A) \wedge B$;
 - **éventuellement**, \wedge est prioritaire sur \vee (et donc $A \wedge B \vee C$ signifie $(A \wedge B) \vee C$). En pratique, je vous conseille fortement de mettre les parenthèses pour éviter toute ambiguïté;
- comme \wedge et \vee sont associatifs, on peut (par exemple) écrire $\bigwedge_{i=1}^n A_i$.

1.3 Égalité syntaxique

D'après ce qui précède, deux formules sont (syntaxiquement) *égales* si et seulement si les arbres correspondants sont égaux. Cependant, il peut y avoir quelques subtilités :

- les formules $A \wedge \neg A$ et \perp ne sont clairement pas égales : les arbres sont totalement différents ;
- de même par exemple pour $\neg(A \wedge B)$ et $\neg A \vee \neg B$;
- en revanche, on peut vouloir considérer que $A \wedge B$ et $B \wedge A$ dénotent la même formule (c'est-à-dire faire rentrer la commutativité du « et » dans la syntaxe des formules). Il suffit pour cela de ne plus mettre d'ordre sur les deux fils du nœud \wedge (il n'y a plus de fils gauche et de fils droit) ;
- il est également possible de rendre les connecteurs \wedge et \vee « syntaxiquement associatifs ». Dans ce cas, les nœuds étiquetés \wedge et \wedge ne seront plus binaires mais d'arité quelconque.

Remarque

Si l'on choisit comme notion d'égalité syntaxique la version « modulo commutativité et associativité », décider (informatiquement) l'égalité de deux formules n'est plus trivial !

1.4 Notation concise

On note parfois :

- \bar{x} pour $\neg x$;
- $x \cdot y$ ou simplement xy pour $x \wedge y$;
- $x + y$ pour $x \vee y$.

Quand on utilise ces notations, on considère systématiquement que \cdot est prioritaire sur $+$. Ainsi, $x\bar{y}z + y\bar{z}$ correspond à $(x \wedge \neg y \wedge z) \vee (y \wedge \neg z)$. On réserve le plus souvent cette notation au cas où $x, y \dots$ sont des variables.

1.5 Substitution

Définition 16.3

Si P et Q sont des formules propositionnelles et si x est une variable, la *substitution de x par Q* dans P est la formule obtenue en remplaçant dans P toutes les occurrences de x par Q . On la note $P[Q/x]$. Formellement, on a :

- $\perp[Q/x] = \perp$;
- $\top[Q/x] = \top$;
- $x[Q/x] = Q$;
- $y[Q/x] = y$ si $y \neq x$;
- $(P_1 \wedge P_2)[Q/x] = P_1[Q/x] \wedge P_2[Q/x]$;
- $(P_1 \vee P_2)[Q/x] = P_1[Q/x] \vee P_2[Q/x]$;
- $(\neg P')[Q/x] = \neg(P'[Q/x])$.

On définit également la *substitution simultanée* de x_1 par Q_1, \dots, x_n par Q_n (où les x_i sont distinctes) de la même manière, en modifiant uniquement les troisième et quatrième points :

- $x_i[Q_1/x_1, \dots, Q_n/x_n] = Q_i$
- $y[Q_1/x_1, \dots, Q_n/x_n] = y$ si $y \notin \{x_1, \dots, x_n\}$.

Remarques

- Autrement dit, on remplace dans P (vue comme un arbre) toutes les feuilles étiquetée x par une copie de l'arbre de Q .
- Attention, on n'a pas en général $P[x/Q][y/R] = P[x/Q, y/R]$. Dans quel cas est-ce vrai ?

Exercice 16.2

p. 345

Écrire une fonction `substitue` telle que `substitue p x q` doit renvoyer $P[Q/x]$.

```
substitue : prop -> int -> prop -> prop
```

2 Sémantique des formules propositionnelles

2.1 Évaluation d'une proposition logique

Définition 16.4

Notons $\mathcal{B} = \{\text{F}, \text{V}\}$ (ou $\{0, 1\}$). On définit les fonctions Et, Ou, et Non par leur table de vérité :

x	y	$\text{Et}(x, y)$	$\text{Ou}(x, y)$	$\text{Non}(x)$
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

Définition 16.5 – Valuation

Une *valuation* est une application de l'ensemble \mathcal{V} des variables propositionnelles dans \mathcal{B} .

Remarques

- On parle aussi de *distribution de vérité* ou d'*interprétation*.
- Choisir une valuation, c'est donc choisir pour chacune des variables si elle est vraie ou fausse.

Définition 16.6 – Évaluation d'une formule logique

Soit v une valuation. On définit l'*évaluation d'une formule P pour la valuation v* par :

- | | |
|--|---|
| <ul style="list-style-type: none"> ■ $\bar{v}(\top) = \top$ ■ $\bar{v}(\perp) = \perp$ ■ $\bar{v}(x) = v(x)$ si $x \in V$ | <ul style="list-style-type: none"> ■ $\bar{v}(\neg P) = \text{Non}(\bar{v}(P))$ ■ $\bar{v}(P \wedge Q) = \text{Et}(\bar{v}(P), \bar{v}(Q))$ ■ $\bar{v}(P \vee Q) = \text{Ou}(\bar{v}(P), \bar{v}(Q))$ |
|--|---|

Remarques

- On parle aussi d'*évaluation de P dans le contexte v*.
- On notera aussi eval_v ou simplement e_v pour \bar{v} .
- V est un ensemble infini, ce qui pose problème si l'on souhaite programmer cette évaluation. Cependant, pour évaluer une proposition P , il suffit de définir la valuation v sur l'ensemble $V(P)$ des variables présentes dans P . Cet ensemble est bien évidemment fini.
- Une manière souvent utile de voir les choses : si l'on prend $B = \{0, 1\}$ (ou si l'on considère que $F < V$), alors \wedge est un min et \vee un max.

Exercice 16.3

p. 345

Écrire une fonction eval prenant en entrée :

- un tableau de booléens v de taille n codant une valuation v ($v.(i)$ correspond à $v(x_i)$);
 - une formule propositionnelle f dont les variables sont supposées incluses dans $\{x_0, \dots, x_{n-1}\}$
- et renvoyant $\bar{v}(f)$.

```
eval : bool array -> prop -> bool
```

Définition 16.7

On dit qu'une valuation v *satisfait* une formule P si $\bar{v}(P) = V$.

Exercice 16.4

p. 345

1. Dresser la table de vérité de la proposition $P = x \wedge (\neg y \vee z)$.
2. Comment s'interprète la ligne correspondant à $x = 0, y = 1, z = 1$ de cette table de vérité en termes d'évaluation de la formule P dans un contexte ?
3. Combien y a-t-il de tables de vérité différentes avec 2 variables ? et avec n variables ?

2.2 Conséquence logique, équivalence logique**Définition 16.8 – Conséquence logique**

Soient P et Q deux formules propositionnelles.

- On dit que Q est une *conséquence logique* de P , et l'on note $P \models Q$, si toute valuation satisfaisant P satisfait également Q .
- Si Γ est un ensemble de formules propositionnelles, on dit que $\Gamma \models Q$ si toute valuation satisfaisant l'ensemble des formules de Γ satisfait également Q .

Exemple 16.5

On a par exemple :

- | | |
|---|--|
| <ul style="list-style-type: none"> ■ $x \models x \vee y$ ■ $x, y \models x \wedge y$ | <ul style="list-style-type: none"> ■ $x, y \models x$ ■ $x, \neg y \models (x \vee \neg z) \wedge (z \vee \neg y)$ |
|---|--|

Définition 16.9 – Équivalence sémantique

Deux formules propositionnelles P et Q sont dites *sémantiquement équivalentes* (ou *logiquement équivalentes*, ou *tautologiquement équivalentes*) si $P \models Q$ et $Q \models P$.

On notera alors $P \equiv Q$.

Remarques

- $P \equiv Q$ signifie donc que $\bar{v}(P) = \bar{v}(Q)$ pour toute valuation v .
- Attention, deux formules sémantiquement équivalentes ne sont pas en général égales ! On a $\neg x \wedge \neg y \equiv \neg(x \vee y)$, mais les arbres correspondants ne sont pas du tout les mêmes.

Propriété 16.10

Si $P \equiv Q$, alors pour toute variable x et toute formule R , on a $P[R/x] \equiv Q[R/x]$.

Démonstration

Supposons l'hypothèse vérifiée et soit v une valuation. Il faut montrer $\bar{v}(P[R/x]) = \bar{v}(Q[R/x])$. Pour cela, considérons la valuation v' définie par :

$$\begin{cases} v'(x) = \bar{v}(R) \\ v'(y) = v(y) \quad \text{si } y \neq x \end{cases}$$

Montrons que $\bar{v}'(P) = \bar{v}(P[R/x])$, par induction structurelle.

- Si $P = \top$ ou $P = \perp$, c'est évident.
- Si $P = x$, alors $\bar{v}'(P) = v'(x) = \bar{v}(R)$ et $\bar{v}(P[R/x]) = \bar{v}(R)$: c'est bon.
- Si $P = y$, avec $y \neq x$, alors $\bar{v}'(P) = v'(y) = v(y)$ et $\bar{v}(P[R/x]) = \bar{v}(y) = v(y)$.
- Si $P = P_1 \wedge P_2$, alors

$$\begin{aligned} \bar{v}'(P) &= \text{Et}(\bar{v}'(P_1), \bar{v}'(P_2)) && \text{par définition de } \bar{v}' \\ &= \text{Et}(\bar{v}(P_1[R/x]), \bar{v}(P_2[R/x])) && \text{par hypothèse d'induction} \\ &= \bar{v}(P_1[R/x] \wedge P_2[R/x]) && \text{par définition de } \bar{v} \\ &= \bar{v}(P[R/x]) && \text{par définition de la substitution} \end{aligned}$$

- Les cas $P = \neg P'$ et $P = P_1 \vee P_2$ se traitent de manière similaire.

On a donc de même $\bar{v}'(Q) = \bar{v}(Q[R/x])$. Or $\bar{v}'(Q) = \bar{v}'(P)$ puisque $P \equiv Q$, d'où $\bar{v}(P[R/x]) = \bar{v}(Q[R/x])$. Donc $P[R/x] \equiv Q[R/x]$. ■

Remarque

Le résultat s'étend immédiatement à une substitution simultanée : si $P \equiv Q$, alors $P[x_1/R_1, \dots, x_n/R_n] \equiv Q[x_1/R_1, \dots, x_n/R_n]$. La démonstration n'est que très légèrement modifiée.

Propriété 16.11

Si $P \equiv P'$ et $Q \equiv Q'$, alors $P \wedge Q \equiv P' \wedge Q'$, $P \vee Q \equiv P' \vee Q'$ et $\neg P \equiv \neg P'$.

Démonstration

Montrons $P \wedge Q \equiv P' \wedge Q'$. Soit v une valuation, on a :

$$\begin{aligned} \bar{v}(P' \wedge Q') &= \text{Et}(\bar{v}(P'), \bar{v}(Q')) && \text{par définition de } \bar{v} \\ &= \text{Et}(\bar{v}(P), \bar{v}(Q)) && \text{puisque } P \equiv P' \text{ et } Q \equiv Q' \\ &= \bar{v}(P \wedge Q) && \text{par définition de } \bar{v} \end{aligned}$$

2.3 Tautologies, satisfiabilité

Définition 16.12

On appelle *tautologie* une proposition P telle que $\bar{v}(P) = V$ pour toute valuation v . On pourra dans ce cas noter $\models P$.

Remarques

- Cette notation est bien cohérente avec notre notation pour la conséquence logique, avec $\Gamma = \emptyset$.
- P est une tautologie si et seulement si $P \equiv \top$.
- On parle aussi de formule *universellement valide*.

Définition 16.13

Une proposition P est dite *satisfiable* s'il existe une valuation v telle que $\bar{v}(P) = V$. Une telle valuation v est alors appelée *témoin de satisfiabilité* de P .

Remarques

- Une formule est satisfiable si et seulement si l'une au moins des lignes de sa table de vérité donne V , c'est une tautologie si et seulement si toutes les lignes de sa table de vérité donnent V .
- On parle parfois d'*antilogie* pour une proposition non satisfiable (ou *insatisfiable*).
- P est une antilogie si et seulement si $P \equiv \perp$.

Propriété 16.14 – Règles usuelles de raisonnement

■ $P \wedge P \equiv P$	Idempotence de \wedge
■ $P \vee P \equiv P$	Idempotence de \vee
■ $\neg\neg P \equiv P$	Élimination de la double négation
■ $P \vee \neg P \equiv \top$	Tiers exclu
■ $P \wedge \neg P \equiv \perp$	Absurdité
■ $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$	De Morgan
■ $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$	De Morgan
■ $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$	Distributivité de \wedge sur \vee
■ $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$	Distributivité de \vee sur \wedge

Démonstration

Démontrons la première loi de De Morgan (les autres points se traitent de la même manière). On dresse les tables de vérité de $A = \neg(x \wedge y)$ et de $B = \neg x \vee \neg y$:

x	y	$\neg(x \wedge y)$	$\neg x \vee \neg y$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Ces tables de vérité coïncident, donc $A \equiv B$. On en déduit d'après la propriété 16.10 que $A[P/x][Q/y] \equiv B[P/x][Q/y]$, c'est-à-dire $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$. ■

Remarques

- Aucune de ces règles ne correspond à une *égalité* (syntaxique) des formules !
- On a bien d'autres règles valides, comme par exemple $P \wedge \top \equiv P$, ou bien $\perp \wedge P = \perp$: on pourra les utiliser sans justification (tant qu'elles sont valides et très simples).

Propriété 16.15

Si P et Q sont des propositions, on a $P \equiv Q$ ssi $\models ((P \wedge Q) \vee (\neg P \wedge \neg Q))$.

Remarque

Autrement dit, en anticipant un peu, $P \equiv Q$ ssi $\models P \leftrightarrow Q$.

2.4 Autres connecteurs

Un connecteur binaire est fondamentalement une application de \mathcal{B}^2 , de cardinal 4, dans \mathcal{B} qui est de cardinal 2. Il en existe donc $2^4 = 16$. Cependant, un certain nombre de ces connecteurs sont soit constants, soient de la forme $f \circ p_1$ ou $f \circ p_2$ où p_1 et p_2 représentent les projections (autrement dit, ils ne dépendent que d'un de leurs paramètres). Presque tous les autres (hors \wedge et \vee) sont donnés ici :

P	Q	$P \leftarrow Q$	$P \rightarrow Q$	$P \leftrightarrow Q$	$P \text{ NAND } Q$	$P \text{ NOR } Q$	$P \text{ XOR } Q$
1	1	1	1	1	0	0	0
1	0	1	0	0	1	0	1
0	1	0	1	0	1	0	1
0	0	1	1	1	1	1	0

Exercice 16.6

p. 346

1. Écrire tous ces connecteurs à partir de \wedge , \vee et \neg .
2. Combien manque-t-il de connecteurs dans la table ci-dessus (en oubliant ceux qui sont constants ou qui ne dépendent que d'un paramètre) ? Les exprimer à l'aide de \wedge , \vee et \neg .

2.5 Fonctions booléennes**Définition 16.16**

On appelle *fonction booléenne* (d'arité n) une application de \mathcal{B}^n dans \mathcal{B} .

Remarque

n peut éventuellement être nul, la fonction étant alors constante.

Définition 16.17

Soit P une proposition et $\mathcal{V}(P) = \{x_1, \dots, x_n\}$ l'ensemble de ses variables. La *fonction booléenne associée à P* est l'application

$$\begin{aligned}\varphi_P : \quad \mathcal{B}^n &\rightarrow \mathcal{B} \\ (v_1, \dots, v_n) &\mapsto e_d(P) \text{ où } d : x_i \rightarrow v_i\end{aligned}$$

Exercice 16.7

p. 346

Déterminer la fonction booléenne associée à $x_1 \wedge \neg(\neg x_2 \vee (x_1 \wedge x_3))$.

Théorème 16.18

Pour toute fonction booléenne f , il existe une proposition P telle que $f = \varphi_P$.

Démonstration

On procède par récurrence sur le nombre n de variables de la fonction f .

- Si $f : \mathcal{B}^0 \rightarrow \mathcal{B}$, alors $f = \varphi_{\top}$ ou $f = \varphi_{\perp}$.
- Si $f : (v_1, \dots, v_{n+1}) \mapsto f(v_1, \dots, v_{n+1})$, posons les deux fonctions $f_0 : (v_1, \dots, v_n) \mapsto f(v_1, \dots, v_n, 0)$ et $f_1 : (v_1, \dots, v_n) \mapsto f(v_1, \dots, v_n, 1)$. Par hypothèse de récurrence, on a $f_1 = \varphi_{P_1}$ et $f_0 = \varphi_{P_0}$; considérons alors $P = (v_{n+1} \wedge P_1) \vee (\bar{v}_{n+1} \wedge P_0)$. On a alors :

$$\begin{aligned}\varphi_P(v_1, \dots, v_n, 0) &= \bar{v}(P) && \text{où } v(x_1) = v_1, \dots, v(x_n) = v_n, v(x_{n+1}) = 0 \\ &= \bar{v}((x_{n+1} \wedge P_1) \vee (\neg x_{n+1} \wedge P_0)) \\ &= \bar{v}(P_0) && \text{puisque } v(x_{n+1}) = 0 \\ &= \varphi_{P_0}(v_1, \dots, v_n) && \text{par définition} \\ &= f_0(v_1, \dots, v_n) && \text{par hypothèse de récurrence} \\ &= f(v_1, \dots, v_n, 0)\end{aligned}$$

On montre de même que $\varphi_P(v_1, \dots, v_n, 1) = f_1(v_1, \dots, v_n) = f(v_1, \dots, v_n, 1)$. On a donc $\varphi_P = f$, ce qui achève la démonstration. ■

Remarques

- Se référer au théorème 16.23 pour une autre démonstration, plus intuitive et plus parlante.
- Comme $x \wedge \neg x \equiv \perp$ et $x \vee \neg x \equiv \top$, on peut en fait se passer des constantes.

Exercice 16.8

p. 346

Trouver P telle que $\varphi_P = f$.

x	y	z	$f(x, y, z)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Définition 16.19

Un ensemble de connecteurs logiques est dit *système complet de connecteurs* si toute fonction booléenne est associée à une formule n'utilisant que ces connecteurs.

Remarque

Dans le théorème 16.18, les propositions sont (implicitement) construites avec \neg , \wedge et \vee , qui constituent donc un système complet de connecteurs.

Propriété 16.20

- $\{\neg, \wedge\}$ et $\{\neg, \vee\}$ sont des systèmes complets de connecteurs.
- {NAND} et {NOR} sont des systèmes complets de connecteurs.

Remarque

{NAND} et {NOR} sont les seuls systèmes complets constitués d'un seul connecteur. Certains circuits électroniques tirent parti de cette propriété en n'utilisant que des portes NAND (ou que des portes NOR). L'ordinateur de bord du module Apollo (AGC), par exemple, n'utilisait que deux types de circuits intégrés : des *sense amplifiers* pour amplifier le signal de la *core memory* et 2 800 doubles portes NOR à trois entrées pour toute la partie logique et calcul.

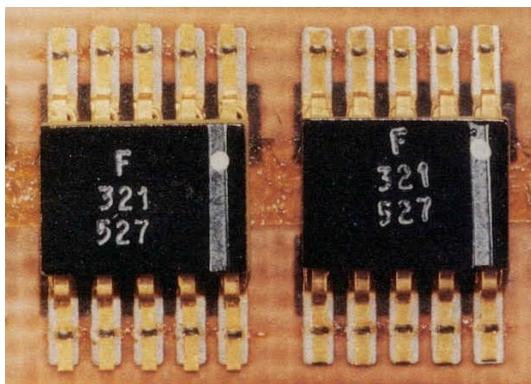


FIGURE 16.1 – Deux *dual three input NOR gates* (dimensions approximatives 70mm × 50mm).

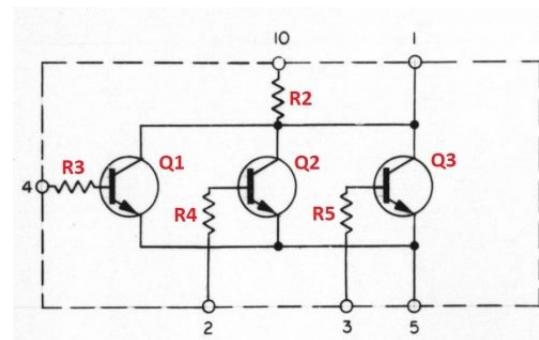


FIGURE 16.2 – Schéma électronique équivalent. La patte 5 est V_0 , 10 est V_+ , 1 est le NOR de 2, 3 et 4. Le schéma est le même pour les pattes 6 à 9.

Exercice 16.9

p. 346

Démontrer la propriété 16.20.

3 Formes normales

Définition 16.21

On appelle :

- *littéral* une formule constituée uniquement d'une variable ou de la négation d'une variable.
- *min-terme* de k variables une conjonction de littéraux dans laquelle chacune des k variables apparaît une et une seule fois (soit telle quelle, soit niée).
- *max-terme* de k variables une disjonction de littéraux dans laquelle chacune des k variables apparaît une et une seule fois.
- *clause* un min-terme ou un max-terme.

Remarque

On peut former 2^k min-termes (ou max-termes) différents sur k variables : pour chaque variable, on choisit si elle apparaît positivement ou négativement.

Exemple 16.10

Pour deux variables, on peut former 4 min-termes et 4 max-termes :

- min-termes : $a \wedge b$, $a \wedge \neg b$, $\neg a \wedge b$, $\neg a \wedge \neg b$.
- max-termes : $a \vee b$, $a \vee \neg b$, $\neg a \vee b$, $\neg a \vee \neg b$.

Définition 16.22

On appelle *forme normale disjonctive* (FND), respectivement *forme normale conjonctive*, FNC, d'une proposition P toute disjonction de min-termes (respectivement conjonction de max-termes) logiquement équivalente à P .

Cette forme normale sera dite *canonique* si chacun des min-termes (respectivement des max-termes) contient les n variables de la formule initiale.

Remarques

- Une formule est sous forme normale conjonctive si et seulement si c'est une conjonction de disjonctions de littéraux.
- Une formule est sous forme normale disjonctive si et seulement si c'est une disjonction de conjonctions de littéraux.

- Par convention, une disjonction vide est égale à \perp , donc \perp est l'unique formule en FND constituée de zéro min-terme.
- De même, une conjonction vide est égale à \top , donc \top est l'unique formule en FNC constituée de zéro max-terme.
- De manière générale, une formule en FND ayant n variables contient entre zéro et 2^n min-termes.

Exercice 16.11

p. 347

Pour chacune des formules suivantes, dire si elle est en FND, en FNC et si elle est ou non canonique (en restreignant les variables à celles qui apparaissent dans la formule). On précisera aussi, le cas échéant, combien de clauses elle contient.

- | | |
|---|---|
| 1. $P_1 = a \wedge b \wedge c$ | 5. $P_5 = \neg(a \wedge b)$ |
| 2. $P_2 = (a \wedge b) \vee (b \wedge c)$ | 6. $P_6 = (a \wedge (b \vee c)) \vee (b \wedge c)$ |
| 3. $P_3 = \perp$ | 7. $P_7 = (a \wedge \neg a) \vee (b \wedge c)$ |
| 4. $P_4 = \top$ | 8. $P_8 = (a \wedge b \wedge c) \vee (\neg b \wedge c)$ |

► Exercice 16.12

p. 347

1. On considère une formule en FND. Peut-on immédiatement déterminer s'il s'agit d'une tautologie ? si elle est satisfiable ?
2. Mêmes questions pour une formule en FNC.

Théorème 16.23

Toute formule propositionnelle admet une unique FND canonique et une unique FNC canonique à l'ordre (et l'associativité) près.

Démonstration

Soit P une formule propositionnelle dont les variables sont x_1, \dots, x_n . Il y a 2^n min-termes sur ces n variables et chacun de ces min-termes s'évalue en \top pour exactement une valuation : par exemple, $x_1 \bar{x}_2 x_3$ est vrai uniquement pour $(1, 0, 1)$. Choisir une FND pour P revient à choisir lesquels des min-termes possibles on met dans la disjonction : il y a exactement un choix possible, qui consiste à choisir ceux correspondant à une valuation d telle que $e_d(P) = \top$. Autrement dit, la FND canonique de P contient une clause pour chaque ligne de la table de vérité de P s'évaluant en \top .

Pour trouver une FNC de P , on commence par prendre la FND de $\neg P$. On a $\neg P \equiv \bigvee_{i=1}^r \bigwedge_{j=1}^n l_j$ où les l_j sont des littéraux. De Morgan donne alors $P \equiv \bigwedge_{i=1}^r \bigvee_{j=1}^n \bar{l}_j$, ce qui fournit une FNC pour P . Pour l'unicité, il suffit de remarquer que deux FNC distinctes pour P donneraient par De Morgan deux FND distinctes pour $\neg P$, ce qui est impossible. Il est aussi possible de procéder directement en exprimant le fait qu'il ne faut pas être dans l'une des lignes s'évaluant en \perp de la table de vérité (ce qui revient essentiellement au même, et qui est illustré à l'exercice 16.13). ■

Remarque

Il y a deux manières d'obtenir une FND pour une formule donnée :

- passer par la table de vérité : on obtient naturellement une FND canonique ;
- utiliser les lois de De Morgan et la distributivité : on obtient une forme qui n'est pas *a priori* canonique.

Exercice 16.13

p. 348

1. Déterminer la FND et la FNC (canoniques) de $(a \wedge b) \vee (a \wedge \neg c)$.
2. De même pour $a \Rightarrow b$.
3. Donner une FND et une FNC (non nécessairement canoniques) pour $x \wedge (y \vee z) \wedge (\neg x \vee y)$.

Une formule en FND ou FNC est très « simple » dans le sens où son arbre¹ est de hauteur au plus 3. Cependant, elle peut être très « grosse » dans le sens où la taille de son arbre peut être exponentielle en

1. Dans la version n -aire gérant l'associativité.

le nombre de variables. De plus, une formule peut avoir une FND concise (n min-termes) mais une FNC très longue (2^n max-termes), ou inversement.

Exercice 16.14

p. 348

Mettre sous forme normale conjonctive la formule $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \dots \vee (x_{2n-1} \wedge x_{2n})$.

4 Problème SAT**Définition 16.24**

Le problème SAT consiste à déterminer, étant donnée une formule propositionnelle en forme normale conjonctive, si elle est satisfiable.

Le problème k -SAT est la restriction de SAT aux formules dont chaque clause contient au plus k littéraux.

Le problème MAX-SAT consiste à déterminer le nombre maximal de clauses que l'on peut simultanément satisfaire (dans une formule en FNC).

Remarques

- Pour SAT, on ne demande pas toujours que la formule soit en FNC : cela ne change rien, le problème restreint aux FNC est aussi difficile que le cas général.
- SAT peut être vu comme une version « plus simple » de MAX-SAT : on ne demande pas combien de clauses on peut satisfaire simultanément, mais seulement si l'on peut toutes les satisfaire simultanément.
- Si l'on restreint SAT aux formules en FND (au lieu de le restreindre aux FNC) le problème devient bien sûr trivial...

Nous reviendrons bien plus en détail sur ces problèmes l'année prochaine, mais on peut dès maintenant retenir quelques points :

- SAT est un problème absolument central en informatique, parce que d'innombrables problèmes se réduisent naturellement à lui ;
- SAT et k -SAT pour $k \geq 3$ sont des problèmes « difficiles » (NP-complets) pour lesquels on ne connaît pas d'algorithme en temps polynomial ;
- 1-SAT et 2-SAT, en revanche, peuvent être résolus en temps linéaire ;
- de même, un certain nombre de versions restreintes de SAT peuvent être résolues en temps polynomial.

Exercice 16.15

p. 348

On considère le problème de la k -coloration d'un graphe : étant donné un graphe G , est-il possible de colorier ses sommets en utilisant au plus k couleurs ? Montrer qu'à partir d'un graphe G à n sommets, on peut calculer en temps polynomial en n une formule propositionnelle $P_k(G)$, de taille polynomiale en n , telle que $P_k(G)$ est satisfiable si et seulement si G est k -colorable.

L'algorithme en « pure force brute » pour SAT consiste, pour une formule faisant intervenir n variables, à tester les 2^n valuations possibles : un tel algorithme n'est clairement pas utilisable pour n plus grand que cent. La conjecture $P \neq NP$ affirme plus ou moins que, dans le pire cas, on ne peut pas faire mieux. Cependant, un énorme effort a été fourni depuis des dizaines d'années pour écrire des *SAT solvers* capables de résoudre en temps raisonnable des instances « typiques » contenant des centaines de milliers de variables, voire davantage. Nous étudierons certains de ces algorithmes en TP.

5 Introduction à la logique du premier ordre

Solutions

Correction de l'exercice 16.2 page 335

```
let rec substitue p x q =
  match p with
  | Var y when y = x -> q
  | Et (f, g) -> Et (substitue f x q, substitue g x q)
  | Ou (f, g) -> Ou (substitue f x q, substitue g x q)
  | Non f -> Non (substitue f x q)
  | _ -> p
```

Correction de l'exercice 16.3 page 336

```
let rec eval v formule =
  match formule with
  | Const b -> b
  | Var i -> v.(i)
  | Non f -> not (eval v f)
  | Et (f, g) -> (eval v f) && (eval v g)
  | Ou (f, g) -> (eval v f) || (eval v g)
```

Remarque

Si jamais on utilise des chaînes de caractères plutôt que des entiers pour nos variables (`Var "x"` plutôt que `Var 0`, essentiellement), il suffit de remplacer le tableau de booléens par un (`string, bool`) dictionnaire.

Correction de l'exercice 16.4 page 336

1.

x	y	z	$\neg y \vee z$	$x \wedge (\neg y \vee z)$
0	0	0	1	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

2. En notant v la valuation $x \rightarrow F, y \rightarrow V, z \rightarrow V$, cette ligne traduit le fait que $\bar{v}(P) = F$.
3. Une table de vérité pour n variables est constituée de 2^n lignes, et on a deux choix pour chacune de ces lignes. Au total, il y a donc 2^{2^n} tables de vérité possibles. En particulier, $2^2 = 2^4 = 16$ tables de vérité pour deux variables.

Correction de l'exercice 16.6 page 339

1. ■ $P \leftarrow Q \equiv P \vee \neg Q$
■ $P \rightarrow Q \equiv \neg P \vee Q$
■ $P \leftrightarrow Q \equiv (P \wedge Q) \vee (\neg P \wedge \neg Q)$
■ $P \text{ NAND } Q \equiv \neg(P \wedge Q)$
■ $P \text{ NOR } Q \equiv \neg(P \vee Q)$
■ $P \text{ XOR } Q \equiv (P \wedge \neg Q) \vee (\neg P \wedge Q)$

2. Il y a 16 connecteurs au total :

- 2 constantes;
- 2 projections et une négation pour chacune d'entre elles (4 connecteurs);
- $\wedge, \vee, \leftarrow, \rightarrow, \leftrightarrow, \text{NAND}, \text{NOR}, \text{XOR}$ (8 connecteurs).

Il manque donc $16 - 2 - 4 - 8 = 2$ connecteurs. On vérifie facilement qu'ils s'expriment $P \wedge \neg Q \equiv \neg(P \rightarrow Q)$ et $\neg P \wedge Q \equiv \neg(Q \rightarrow P)$.

Correction de l'exercice 16.7 page 339

Déterminer la fonction booléenne revient exactement à dresser la table de vérité :

x_1	x_2	x_3	$x_1 \cdot \bar{x}_2 + x_1 \cdot x_3$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

On constate que la formule est logiquement équivalente à $x_1 x_2 \bar{x}_3$, ce qu'on aurait facilement pu montrer directement :

$$\begin{aligned} x_1 \cdot \bar{x}_2 + x_1 \cdot x_3 &\equiv x_1 \cdot x_2 \cdot \bar{x}_1 \cdot \bar{x}_3 \\ &\equiv x_1 \cdot x_2 \cdot (\bar{x}_1 + \bar{x}_3) \\ &\equiv x_1 x_2 \bar{x}_1 + x_1 x_2 \bar{x}_3 \\ &\equiv x_1 x_2 \bar{x}_3 \end{aligned}$$

Correction de l'exercice 16.8 page 340

En suivant la démonstration, on obtient $\bar{x}(\bar{y}(\bar{z} \cdot 0 + z \cdot 1) + y(\bar{z} \cdot 0 + z \cdot 1)) + x(\bar{y}(\bar{z} \cdot 1 + z \cdot 1) + y(\bar{z} \cdot 0 + z \cdot 1))$, ou après élimination des constantes $\bar{x}(\bar{y}z + yz) + x(\bar{y}(z + \bar{z}) + yz)$. On peut bien sûr simplifier, par exemple en $\bar{x}z + x(\bar{y} + yz)$.

Une autre méthode (qui revient à construire la forme normale disjonctive canonique) consiste essentiellement à prendre la disjonction des lignes de la table donnant 1. On obtient alors : $\bar{x} \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot \bar{z} + x \cdot \bar{y} \cdot z + x \cdot y \cdot z$.

Correction de l'exercice 16.9 page 341

1. Montrons par induction structurelle que pour toute formule F , il existe une formule F' n'utilisant que \neg et \wedge telle que $F' \equiv F$.
 - Si F est une constante ou une variable, c'est évident.
 - Si $F = \neg G$, l'hypothèse d'induction fournit $G' \equiv G$ n'utilisant que \neg et \wedge . En posant $F' = \neg G'$, on a immédiatement $F' \equiv F$ et F' n'utilise que \wedge et \neg .
 - Si $F = G \wedge H$, alors on dispose de même de G' et H' et $F' = G' \wedge H'$ convient.

- Si $F = G \vee H$, on pose $F' = \neg(\neg G' \wedge \neg H')$. F' n'utilise que \neg et \wedge , et :

$$\begin{aligned}
 F' &= \neg(\neg G' \wedge \neg H') \\
 &\equiv \neg\neg G' \vee \neg\neg H' && \text{De Morgan} \\
 &\equiv G' \vee H' && \text{involutivité de } \neg \\
 &\equiv G \vee H && \text{par hypothèse d'induction}
 \end{aligned}$$

On a donc bien $F' \equiv F$.

Ainsi, $\{\neg, \wedge\}$ est un système complet de connecteurs. La démonstration pour $\{\neg, \vee\}$ est quasiment identique, en remarquant que $\neg(\neg G' \vee \neg H') \equiv G' \wedge H'$.

2. On montre par induction structurelle que toute formule F n'utilisant que les connecteurs \neg et \wedge est logiquement équivalente à une formule n'utilisant que NAND.
- Si F est une constante ou une variable, c'est évident.
- Si $F = \neg G$, soit $G' \equiv G$ n'utilisant que NAND. On a $\neg G' \equiv \neg(G' \wedge G') \equiv G' \text{ NAND } G'$, donc la formule $F' = G' \text{ NAND } G'$ convient.
- Si $F = G \wedge H$, l'hypothèse d'induction fournit G' et H' et l'on a :

$$\begin{aligned}
 F &= G \wedge H \\
 &\equiv G' \wedge H' \\
 &\equiv \neg(G' \text{ NAND } H') \\
 &\equiv \underbrace{(G' \text{ NAND } H') \text{ NAND } (G' \text{ NAND } H')}_{F'}
 \end{aligned}$$

Pour conclure, toute formule propositionnelle est logiquement équivalente à une formule n'utilisant que \wedge et \neg d'après la première question, et donc à une formule n'utilisant que NAND d'après ce qui précède : $\{\text{NAND}\}$ est un système complet.

Pour $\{\text{NOR}\}$, on procède de manière très similaire en partant cette fois du système complet $\{\neg, \vee\}$ et en remarquant que :

- $\neg A \equiv \neg(A \vee A) \equiv A \text{ NOR } A$;
- $A \vee B \equiv \neg(\neg A \wedge \neg B) \equiv (\neg A \wedge \neg B) \text{ NOR } (\neg A \wedge \neg B)$.

Correction de l'exercice 16.II page 342

- P_1 est une FND canonique constituée d'une seule clause, et une FNC non canonique de 3 clauses.
- P_2 est une FND non canonique de deux clauses, ce n'est pas une FNC.
- P_3 est une FND canonique de zéro clause, ce n'est pas une FNC.
- P_4 est une FNC canonique de zéro clause, ce n'est pas une FND.
- P_5 n'est ni une FND, ni une FNC.
- idem*
- P_7 n'est pas une FND car $a \wedge \neg a$ n'est pas un min-terme, et ce n'est bien sûr pas une FNC.
- P_8 est une FND de deux clauses, ce n'est pas une FNC.

Correction de l'exercice 16.I2 page 342

- Une formule en FND est satisfaite si et seulement si l'un au moins de ses min-termes l'est. Or un min-terme est toujours satisfiable : la seule formule en FND non satisfiable est donc celle réduite à \perp , et déterminer la satisfiabilité est immédiat.
Il n'y a en revanche aucun moyen immédiat de déterminer si une formule en FND est une tautologie.

2. Pour les FNC, c'est l'inverse :
- la seule tautologie est la formule réduite à \top ;
 - il n'y a pas de moyen simple de déterminer la satisfiabilité.

Correction de l'exercice 16.13 page 342

1. On veut des formes canoniques, le plus simple est de passer par la table de vérité :

a	b	c	$P = ab + a\bar{c}$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

On a donc :

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

Pour la FNC, il ne faut pas être dans la ligne $(0, 0, 0)$, ni dans la ligne $(0, 0, 1)$ et ainsi de suite. Comme le fait de ne pas être dans la ligne $(0, 0, 1)$ (par exemple) s'exprime par $\neg(a \wedge \neg b \wedge c) \equiv a \vee b \vee \neg c$, on obtient :

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

2. On a $a \Rightarrow b \equiv \neg a \vee b$ ce qui est la FNC canonique (et une FND non canonique). Les trois lignes autres que $(1, 0)$ s'évaluent en \top , donc la FND canonique est $a \Rightarrow b \equiv (a \wedge b) \vee (\neg a \wedge b) \vee (\neg a \wedge \neg b)$.
3. La formule est déjà en FNC (non canonique). On utilise la distributivité pour obtenir une FND :

$$\begin{aligned} x(y+z)(\bar{x}+y) &\equiv xy\bar{x}+xyy+xz\bar{x}+xzy \\ &\equiv xy+xyz \end{aligned}$$

On peut bien sûr simplifier en xy , mais $xy + xyz$ est déjà une FND.

Correction de l'exercice 16.14 page 343

Il suffit d'utiliser la distributivité de \wedge sur \vee . On obtient

$$\bigwedge_{(\varepsilon_1, \dots, \varepsilon_n) \in \{0,1\}^n} \bigvee_{i=0}^{n-1} x_{2i+\varepsilon_i}$$

Cette FNC contient 2^n clauses, alors que la FND initiale contenait n clauses.

Correction de l'exercice 16.15 page 343

On note $V = s_1, \dots, s_n$ les sommets de G , E les arêtes, $C = \{1, \dots, k\}$ les couleurs possibles et l'on définit nk variables $(x_{i,c})_{1 \leq i \leq n, 1 \leq c \leq k}$. Intuitivement, la variable $x_{i,c}$ prendra la valeur V si le sommet s_i est colorié avec la couleur c . Formellement, il y a une bijection entre les parties de $V \times C$ et les valuations de ces nk variables : à une valuation v , on associe la partie X telle que $(x_{i,c}) \in X$ si et seulement si $v(x_{i,c}) = V$. Il suffit donc de trouver une formule $P_k(G)$ telle que $\bar{v}(P_k(G)) = V$ si et seulement si la partie correspondante de $V \times C$ est un coloriage. Il y a deux

points à considérer :

- chaque sommet doit posséder exactement une couleur ;
- deux sommets adjacents ne peuvent avoir la même couleur.

Le fait que le sommet x_i soit colorié avec la couleur c et aucune autre s'écrit $C(i, c) := x_{i,c} \wedge \bigwedge_{c' \neq c} \overline{x_{i,c'}}$. Le fait qu'il soit colorié avec exactement une couleur s'écrit donc $C(i) = \bigvee_{c=1}^k C(i, c)$, formule dont la taille est en $O(k^2)$. On prend ensuite la conjonction des $C(i) : C := \bigwedge_{i=1}^n C(i)$, formule dont la taille est en $O(nk^2)$ et qui traduit le fait que la partie X définie plus haut est le graphe d'une application de V dans C.

Pour chaque arête $\{x_i, x_j\}$ du graphe et chaque couleur c , $\overline{x_{i,c}} \vee \overline{x_{j,c}}$ traduit le fait que les sommets x_i et x_j ne peuvent être tous les deux coloriés avec c . Le fait que notre application est un coloriage s'écrit donc $D := \bigwedge_{\{x_i, x_j\} \in E} \bigwedge_{c=1}^k (\overline{x_{i,c}} \vee \overline{x_{j,c}})$. Comme $|E| = O(n^2)$, cette formule a une taille en $O(kn^2)$.

On obtient donc finalement la formule $C \wedge D$, de taille $O(n^2k + nk^2)$, qui est satisfiable si et seulement si le graphe est k -colorable. Notons que l'on peut supposer $k < n$ (sinon le graphe est évidemment k -colorable), et que la taille de la formule est donc en $O(n^3)$.

Travaux pratiques

INITIATION À OCAML

I Quelques fonctions élémentaires

Exercice I.1

p. 355

1. Écrire une fonction norme : **float** -> **float** -> **float** telle que norme a b vaille $\sqrt{a^2 + b^2}$. La fonction racine carrée est pré définie, c'est **sqrt** : **float** -> **float**.
2. Écrire une fonction moyenne : **float** -> **float** -> **float** telle que moyenne a b renvoie $\frac{a+b}{2}$.
3. On considère la fonction suivante :

```
let f a b =
  (a + b) / 2
```

- a. Quel est le type de cette fonction ?
- b. Que vaut $f\ 10\ 5$?
- c. À l'aide de la fonction de conversion **float_of_int** : **int** -> **float**, écrire une fonction **moyenne_entiers** : **int** -> **int** -> **float** calculant la moyenne de ses arguments.

Exercice I.2

p. 355

1. Que font les fonctions pré définies **abs** et **abs_float**? Un petit détail : faites bien attention au parenthésage. Si vous écrivez **abs -3**, ce sera compris **abs - 3** et donnera donc une erreur (vous essayez de faire la soustraction entre la fonction **abs** et l'entier **3**). Il faut écrire **abs (-3)**.
2. Réécrire votre propre version de **abs**.

2 Fonctions récursives

Exercice I.3

p. 355

Écrire une fonction suite : **int** -> **float** telle que suite n soit le n-ème terme de la suite u définie par

$$u_0 = 4 \quad \text{et} \quad \forall n \in \mathbb{N}, u_{n+1} = 3u_n + 2$$

Exercice I.4

p. 355

1. Écrire une fonction factorielle : **int** -> **int** telle que l'appel factorielle n renvoie l'entier n!.
On rappelle que $n!$ est défini par $0! = 1$ et $(n+1)! = (n+1) \cdot n!$.
2. Que se passe-t-il si vous appelez cette fonction sur un entier n strictement négatif?

Exercice I.5

p. 356

Écrire une fonction `somme_carres` : `int -> int` qui calcule $\sum_{k=1}^n k^2$.

Exercice I.6

p. 356

1. Écrire une fonction `puissance` : `float -> int -> float` telle que l'appel `puissance x n` renvoie x^n (en supposant $n \geq 0$). On ira au plus simple sans chercher à être efficace.
2. Écrire une nouvelle version de cette fonction, que l'on appellera `puissance_bis`, acceptant les exposants négatifs (en supposant dans ce cas $x \neq 0$).

3 Manipulation de listes**Exercice I.7**

p. 356

1. Écrire une fonction `somme_liste` : `float list -> float` qui calcule la somme des éléments d'une liste de flottants.
2. Écrire une fonction `longueur` : `'a list -> int` qui renvoie la longueur (nombre d'éléments) d'une liste.
3. Écrire une fonction `moyenne_liste` : `float list -> float` qui calcule la moyenne des éléments d'une liste de flottants.

Exercice I.8

p. 357

Écrire une fonction `croissant` : `'a list -> bool` qui détermine si une liste est croissante (au sens large). Notez qu'une liste contenant zéro ou un élément est toujours croissante.

Pour filtrer les listes contenant au moins deux éléments et récupérer les deux premiers éléments ainsi que le reste de la liste, on pourra utiliser un motif de la forme :

```
| x :: y :: xs -> ...
(* x est l'élément de tête, y le deuxième, xs le reste de la liste. *)
```

Exercice I.9

p. 357

L'opérateur de *concaténation* sur les listes se note @ en Caml :

```
# [1; 3; 5] @ [4; 2; 8; 10];
- : int list = [1; 3; 5; 4; 2; 8; 10]
```

Attention, il est très différent du constructeur « :: » : quand on écrit `x :: l`, il faut que `l` soit une `'a list` et que `x` soit un `'a`, alors que dans `u @ v`, `u` et `v` sont tous deux des `'a list`.

Écrire une fonction (réursive) `concat` : `'a list -> 'a list -> 'a list` telle que `concat u v` renvoie `u @ v`. On s'interdira bien sûr d'utiliser l'opérateur @ dans l'écriture de cette fonction.

Exercice I.10

p. 357

En utilisant la fonction `concat` (ou l'opérateur @, ce qui revient au même), écrire une fonction `miroir` : `'a list -> 'a list` qui renverse son argument (*i.e.* telle que `miroir [4; 8; 8; 1; 2]` renvoie `[2; 1; 8; 8; 4]`).

Ce n'est pas la « bonne » manière d'écrire une fonction *miroir* car on effectue beaucoup trop d'opérations. Nous verrons comment faire mieux très prochainement.

Exercice I.11

p. 357

Écrire une fonction *uniques* : '*a list* -> '*a list* qui prend en entrée une liste *u* supposée triée et renvoie la liste des éléments distincts de *u*.

```
utop[14]> uniques [2; 2; 2; 3; 5; 5; 6; 7; 8; 8; 8];
- : int list = [2; 3; 5; 6; 7; 8]
```

4 Pour ceux qui ont fini

Pour les fonctions suivantes, on pourra (ce n'est pas indispensable) utiliser une variante du *pattern matching* que nous verrons au prochain chapitre : les clauses *gardées*.

```
(* Compte le nombre d'occurrences de x dans la liste u *)
let rec nb_occs x u = match u with
| [] -> 0
| y :: ys when x = y -> 1 + nb_occs x ys (* si x = y, on suit la flèche *)
| y :: ys -> nb_occs x ys (* sinon, on arrive ici *)
```

Notez que remplacer « *y :: ys when x = y -> ...* » par « *x :: xs -> ...* » ne marche pas du tout, pour une raison que j'expliquerai au chapitre suivant.

Exercice I.12

p. 358

Écrire une fonction *indice* : '*a* -> '*a list* -> *int* telle que *indice* *x u* renvoie l'indice de la première apparition de *x* dans la liste *u*. Si *x* n'apparaît pas dans *u*, on signalera une erreur à l'aide de l'instruction *failwith "not found"*. On considère (pour cet exercice et pour les suivants) que les indices commencent à zéro.

```
utop[12]> indice 3 [5; 2; 3; 4; 1; 7];
- : int = 2
utop[13]> indice 6 [5; 2; 3; 4; 1; 7];
Exception: (Failure "not found").
```

Exercice I.13

p. 358

Écrire une fonction *sous_liste* : '*a list* -> *int list* -> '*a list* qui prend une liste *u* et une liste (strictement croissante) d'indices (*x₁*, ..., *x_k*) et renvoie la liste (*u_{x₁}*, ..., *u_{x_k}*). Les *x_i* « trop grands » (i.e. supérieurs ou égaux à la longueur de *u*) seront simplement ignorés.

```
utop[6]> sous_liste [12; 17; 15; 4; 3; 8] [0; 2; 3; 15];
- : int list = [12; 15; 4]
```

Exercice I.14

p. 359

La bibliothèque standard fournit une fonction *List.compare_lengths* : '*a list* -> '*b list* -> *int* dont la spécification est la suivante :

- si la première liste (passée en argument) est strictement plus courte que la seconde, la

- fonction renvoie l'entier -1;
- si les deux listes ont le même nombre d'éléments, elle renvoie l'entier 0;
 - si la deuxième liste est strictement plus longue, la fonction renvoie 1.
1. Pourquoi le type de cette fonction est-il '`a list -> 'b list -> int`' et pas '`a list -> 'a list -> int`'?
 2. Écrire une fonction `comparer_longueurs` ayant la même spécification que `List.compare_lengths` et utilisant la fonction `longueur` définie à l'exercice 1.7.
 3. Expliquer pourquoi (et dans quels cas) cette fonction n'est pas très efficace.
 4. Écrire une fonction `comparer_longueurs_efficace` qui arrête le parcours des listes dès que possible.

Exercice I.15

p. 359

Écrire une fonction `indices_min` : '`'a list -> int list`' qui renvoie la liste des « indices » des occurrences du minimum dans une liste. Le comportement de la fonction n'est pas spécifié si on l'appelle sur la liste vide.

```
# indices_min [5.1 ; 5.1 ; 6.3; 4.; 6.2; 4.; 7.5; 8.3; 4.; 9.];;
- : int list = [3; 5; 8]
```

Exercice I.16

p. 360

Écrire une fonctions `paquets` : '`'a list -> 'a list list`' qui regroupe les éléments de son argument par « paquet » d'éléments consécutifs égaux :

```
# paquets [1; 1; 2; 2; 3; 4; 1; 1; 1; 8; 8];
- : int list list = [[1; 1]; [2; 2]; [3]; [4]; [1; 1; 1]; [8; 8]]
```

Solutions

Correction de l'exercice I.1 page 351

1.

```
let norme x y =
  sqrt (x ** 2. +. y ** 2.)
```

2.

```
let moyenne a b = (a +. b) /. 2.
```

3. a. On a $f : \text{int} \rightarrow \text{int} \rightarrow \text{int}$.
b. $f 10 5 = 7$
c. Deux solutions raisonnables, de type $\text{int} \rightarrow \text{int} \rightarrow \text{int}$:

```
let moyenne_entiers a b =
  moyenne (float_of_int a) (float_of_int b)

let moyenne_entiers_bis a b =
  float_of_int (a + b) /. 2.
```

Correction de l'exercice I.2 page 351

1. Ce sont deux versions de la valeur absolue, l'une pour les entiers et l'autre pour les flottants.
2.

```
(* absolute : int -> int *)
let absolute x =
  if x >= 0 then x else -x
```

Correction de l'exercice I.3 page 351

On suit la définition mathématique :

```
(* int -> float *)
let rec suite n =
  if n = 0 then 4.
  else 3. *. (suite (n - 1)) +. 2.
```

Correction de l'exercice I.4 page 351

1.
2. L'exemple est dans le cours :

```
let rec factorielle n =
  if n = 0 then 1
  else n * factorielle (n - 1)
```

3. Si on appelle cette fonction avec un argument n strictement négatif, elle ne termine pas. En pratique, on observe ici un *dépassement de pile* (appelé *stack overflow* en anglais) : nous expliquerons plus tard ce phénomène.

Correction de l'exercice I.5 page 352

```
(* int -> int *)
let rec somme_carres n =
  if n = 0 then 0
  else n*n + somme_carres (n - 1)
```

Correction de l'exercice I.6 page 352

1. Pas de difficulté, mais attention cependant à bien prendre $x^0 = 1$ comme cas de base.

```
(* float -> int -> float *)
let rec puissance x n =
  if n = 0 then 1.
  else x *. puissance x (n - 1)
```

2. Il y a de nombreuses solutions, mais le plus simple est sans doute de réutiliser ce que nous avons déjà fait :

```
(* float -> int -> float *)
let rec puissance_bis x n =
  let res = puissance x (abs n) in
  if n >= 0 then res else 1. /. res
```

Correction de l'exercice I.7 page 352

1.

```
(* float list -> float *)
let rec somme_liste t =
  match t with
  | [] -> 0.
  | x :: xs -> x +. somme_liste xs
```

2.

```
(* 'a list -> int *)
let rec longueur t = match t with
  | [] -> 0
  | x :: xs -> 1 + longueur xs
```

3.

```
(* int list -> float *)
let moyenne_liste t =
  let s = somme_liste t in
  let n = float_of_int (longueur t) in
  s /. n
```

Correction de l'exercice I.8 page 352

Une liste u_1, u_2, \dots, u_n est croissante si et seulement si $u_1 \leq u_2$ et u_2, \dots, u_n est croissante.

```
(* 'a list -> bool *)
let rec croissante liste =
  match liste with
  | [] -> true
  | [x] -> true
  | x :: y :: t -> (x <= y) && croissante (y :: t)
```

On peut facilement regrouper les deux premiers cas en changeant l'ordre dans lequel on examine les motifs :

```
let rec croissante liste =
  match liste with
  | x :: y :: t (* au moins 2 éléments *) -> (x <= y) && croissante (y :: t)
  | _ (* autres cas *) -> true
```

Correction de l'exercice I.9 page 352

On utilise l'égalité $(x :: xs) @ v = x :: (xs @ v)$.

```
(* 'a list -> 'a list -> 'a list *)
let rec concat t u =
  match t with
  | [] -> u
  | x :: xs -> x :: (concat xs u)
```

Correction de l'exercice I.10 page 352

On a $\text{miroir } [u_1; \dots; u_n] = [u_n; \dots; u_2; u_1] = (\text{miroir } [u_2, \dots, u_n]) @ [u_1]$.

```
(* 'a list -> 'a list *)
let rec miroir t =
  match t with
  | [] -> []
  | x :: u -> concat (miroir u) [x]
```

Attention, ce n'est pas la « bonne » manière de procéder, pour des raisons de complexité. Nous reviendrons sur ce problème très bientôt.

Correction de l'exercice I.11 page 353

Une fois qu'on a la bonne tournure d'esprit, écrire cette fonction devient très simple. Acquérir cette tournure d'esprit peut prendre un peu de temps : c'est l'un des objectifs principaux des

premiers TP.

```
(* uniques : 'a list -> 'a list *)
let rec uniques u =
  match u with
  | [] -> []
  | [x] -> [x]
  | x :: y :: xs ->
    if x = y then uniques (y :: xs)
    else x :: uniques (y :: xs)
```

Correction de l'exercice I.I2 page 353

Il y a plusieurs solutions, mais le plus simple, et donc ce qu'il faut privilégier, est :

```
(* indice : 'a -> 'a list -> int *)
let rec indice x u =
  match u with
  | [] -> failwith "not found"
  | y :: ys -> if x = y then 0 else 1 + indice x ys
```

On peut aussi utiliser une fonction auxiliaire, et obtenir ainsi une solution qui ressemble plus à ce qu'on ferait avec une boucle. C'est parfois préférable, mais le principal objectif pour l'instant est de se convaincre que la solution « naturelle » est bien la première.

```
let indice_bis x u =
  (* i est l'indice de l'élément de tête de v dans la liste u initiale. *)
  let rec aux i v =
    match v with
    | [] -> failwith "not found"
    | y :: ys -> if x = y then i else aux (i + 1) ys
  aux 0 u
```

Correction de l'exercice I.I3 page 353

Ici, on va utiliser une fonction auxiliaire aux prenant trois arguments :

- v est le partie de u qui reste à parcourir;
- ind est la partie de indices qui reste à parcourir;
- actuel est l'indice de l'élément de tête de v dans la liste u initiale.

```
let sous_liste u indices =
  let rec aux v ind actuel =
    match v, ind with
    | [], _ -> []
    | _, [] -> []
    | x :: xs, j :: js ->
      if j < actuel then aux v js actuel
      else if j > actuel then aux xs ind (actuel + 1)
      else x :: aux xs js (actuel + 1)
  aux u indices 0
```

Correction de l'exercice I.I4 page 353

1. Si la fonction avait le type '`a list -> 'a list -> int`', il faudrait que les deux listes aient le même type (deux listes d'entiers, ou deux liste de flottants, ou...). Il n'y a aucune raison de se restreindre à ce cas (on compare seulement les longueurs des listes, pas leurs éléments), donc le type sera plus général : '`'a list -> 'b list -> int`'.

2.

```
let comparer_longueurs u v =
  let nu = longueur u in
  let nv = longueur v in
  if nu < nv then -1
  else if nu = nv then 0
  else 1
```

3. On commence ici par calculer `longueur u` et `longueur v`, or en examinant le code de la fonction `longueur` on se rend compte que cela nécessite de parcourir entièrement la liste `u` et la liste `v`. Si par exemple `u` ne contient qu'un seul élément et `v` en contient un million, il est clair qu'on aurait pu s'arrêter dès que l'on voit le deuxième élément de `v`.

4.

```
let rec comparer_longueurs_efficace u v =
  match u, v with
  | [], [] -> 0
  | [], _ -> -1
  | _, [] -> 1
  | _ :: xs, _ :: ys -> comparer_longueurs_efficace xs ys
```

Correction de l'exercice I.I5 page 354

Le plus simple est d'effectuer deux parcours : un pour calculer le minimum, puis un pour déterminer les indices de ses occurrences.

```
(* 'a list -> 'a *)
let rec min_liste t =
  match t with
  | [] -> failwith "pas de minimum"
  | [x] -> x
  | x :: xs -> min x (min_liste xs)

(* 'a list -> int list *)
let indices_min_1 t =
  let rec indices_egaux t valeur ind_actuel =
    match t with
    | [] -> []
    | x :: xs when x = valeur ->
      ind_actuel :: indices_egaux xs valeur (ind_actuel + 1)
    | x :: xs -> indices_egaux xs valeur (ind_actuel + 1) in
  let m = min_liste t in
  indices_egaux t m 0
```

On donne aussi une version n'utilisant qu'un seul parcours, hors de portée pour un premier TD :

```

let indices_min u =
  (* aux i v renvoie le couple (m, ind), où :
   * - m est le minimum de v
   * - ind est la liste de ses occurrences, en considérant
   *   que le premier élément de v a pour indice i. *)
let rec aux i v =
  match v with
  | [] -> failwith "impossible"
  | [x] -> (x, [i])
  | x :: xs ->
    let m, indices = aux (i + 1) xs in
    if x < m then (x, [i])
    else if x = m then (m, i :: indices)
    else (m, indices) in
  let m, indices = aux 0 u in
  indices

```

Correction de l'exercice I.16 page 354

Pour commencer, la version la plus simple (à privilégier) :

```

(* 'a list -> 'a list list *)
let rec paquets t =
  match t with
  | [] -> []
  | x :: xs ->
    match paquets xs with
    | (y :: ys) :: us when x = y -> (x :: y :: ys) :: us
    | v -> [x] :: v

```

Il est également possible d'utiliser un autre style de programmation, plus proche de ce que l'on ferait en Python. Dans la fonction auxiliaire :

- t est le morceau de liste qui reste à traiter;
- l_paquets est la liste des paquets déjà constitués (sauf celui en cours);
- courant est le paquet actuel (auquel on peut encore ajouter des éléments avant de le placer en tête de l_paquets).

En réfléchissant un peu (ou avec de l'habitude), on remarque que la liste des paquets que l'on construit est « à l'envers », il faut donc la retourner avant de la renvoyer. C'est le rôle de l'appel `List.rev` à la fin.

```

(* 'a list -> 'a list list *)
let paquets_2 t =
  let rec aux t l_paquets courant =
    match t, courant with
    | [], [] -> l_paquets
    | [], _ -> courant :: l_paquets
    | x :: u, y :: v when x = y -> aux u l_paquets (x :: courant)
    | x :: u, [] -> aux u l_paquets [x]
    | x :: u, y :: v -> aux u (courant :: l_paquets) [x] in
  List.rev (aux t [] [])

```

MANIPULATION DE LISTES

Exercice II.1 – Opérations élémentaires sur les listes

p. 364

- Écrire une fonction `mem` : `'a -> 'a list -> bool` qui prend une valeur `x` et une liste `u` et renvoie `true` si l'un des éléments de `u` vaut `x`, `false` sinon.
Cette fonction est en fait prédéfinie : c'est `List.mem`.
- Écrire une fonction `nth` : `'a list -> int -> 'a` qui prend une liste `u` et un entier `n` et renvoie l'élément numéro `n` de la liste (en commençant la numérotation à zéro, et en renvoyant une exception si la liste n'a pas assez d'éléments).
Cette fonction est en fait prédéfinie : c'est `List.nth`.
- Écrire une fonction `take` : `int -> 'a list -> 'a list` qui prend un entier `n` et une liste `u` et renvoie la liste constituée des `n` premiers éléments de `u`. Si `u` a moins de `n` éléments, on renverra `u` en entier.
- Écrire une fonction `range` : `int -> int -> int list` telle que `range a b` renvoie la liste `[a; a + 1; ... ; b - 1]` (et donc la liste vide si `a ≥ b`).

Exercice II.2 – Concaténation et miroir

p. 364

- Écrire une fonction `concat` : `'a list -> 'a list -> 'a list` qui renvoie la concaténation de ses deux arguments (on doit donc avoir `concat u v = u @ v`), sans bien sûr utiliser l'opérateur `@` prédéfini.
- Combien cette fonction fait-elle d'appels au constructeur `::` (en fonction des longueurs de ses arguments) ?
- Écrire une fonction `miroir_naif` : `'a list -> 'a list` qui renvoie la liste constituée des mêmes éléments que son argument, mais en ordre inverse.
- Combien cette fonction fait-elle d'appels au constructeur `::` (en fonction de la longueur de son argument) ? *On ne comptera que les appels utilisés pour construire, autrement dit ceux apparaissant à droite de la flèche dans un `match`.*
- Écrire une fonction `rev_append` : `'a list -> 'a list -> 'a list` telle que `rev_append [a1; ... ; an] [b1; ... ; bp]` renvoie `[an; ... ; a1; b1; ... ; bp]`.
Cette fonction est prédéfinie : c'est `List.rev_append`.
- À l'aide de cette fonction `rev_append`, écrire une fonction `miroir` faisant la même chose que `miroir_naif` mais effectuant un nombre d'opérations proportionnel à la longueur de son argument.
Cette fonction est prédéfinie : c'est `List.rev`.

Exercice II.3 – Map

p. 365

- Définir une fonction `applique` : `('a -> 'b) -> 'a list -> 'b list` telle que `applique f [x_1; ...; x_n]` renvoie `[f x_1; ...; f x_n]`.
Regarder maintenant ce que fait la fonction prédéfinie `List.map`.
- Écrire (à l'aide de `applique`) une fonction `liste_carres` : `int list -> int list` (qui renvoie la liste des carrés des éléments de la liste passée en argument). La définition de cette fonction doit tenir sur une ligne (sans forcer!).

Exercice II.4 – Éléments répétés

p. 366

1. Écrire une fonction `sans_doublons_triee` : '`a list -> bool`' qui prend en entrée une liste `u` supposée triée et renvoie `true` si et seulement si les éléments de `u` sont deux à deux distincts.

2. Écrire une fonction `elimine_doublons_triee` : '`a list -> 'a list`' qui prend en entrée une liste `u` supposée triée et renvoie la liste des éléments distincts de `u`.

```
# elimine_doublons [1; 1; 2; 3; 3; 3; 4; 5; 5; 5; 6; 6; 6; 7];
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

3. Écrire une fonction `sans_doublons` : '`a list -> bool`' qui prend en entrée une liste `u` et détermine si ses éléments sont deux à deux distincts.

On pourra utiliser la fonction mem définie plus haut.

4. Écrire une fonction `elimine_doublons` : '`a list -> 'a list`' qui prend en entrée une liste `u` et renvoie la liste des éléments distincts de `u`, dans l'ordre de leur dernière apparition dans `u`.

```
# elimine_doublons [1; 5; 2; 5; 3; 3; 7; 3; 7; 3; 1];
- : int list = [2; 5; 7; 3; 1]
```

5. Combien de tests d'égalité la fonction `sans_doublons_triee` effectuera-t-elle au maximum si on l'appelle sur une liste de longueur `n`? Préciser dans quel cas ce maximum est atteint.

6. Mêmes questions pour la fonction `sans_doublons`.

Exercice II.5 – Run-length encoding

p. 366

Le principe du *run-length encoding* (*codage par plages* en français, paraît-il) est de remplacer, dans une liste, `k` occurrences consécutives d'une même valeur `x` par le couple `(x, k)`. C'est la méthode de compression la plus simple que l'on puisse imaginer, ce qui ne l'empêche pas d'être très utilisée (mais très rarement seule).

1. Écrire une fonction `compresse` : '`'a list -> ('a * int) list`' réalisant la « compression ».

2. Écrire une fonction `decompresse` : '`('a * int list) -> 'a list`' réalisant la « décompression ».

```
# let l = compresse ["a"; "b"; "b"; "b"; "a"; "c"; "c"; "d"];
val l : (string * int) list =
[("a", 1); ("b", 3); ("a", 1); ("c", 2); ("d", 1)]
# decompresse l;;
- : string list = ["a"; "b"; "b"; "b"; "a"; "c"; "c"; "d"]
```

Exercice II.6 – Projet Euler – problème 204

p. 367

On appelle *nombre de Hamming de type n* un entier strictement positif n'ayant aucun facteur premier strictement supérieur à `n`.

Les premiers nombres de Hamming de type 5 sont 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, et il y a 1105 nombres de Hamming de type 5 inférieurs ou égaux à 10^8 .

Combien y a-t-il de nombres de Hamming de type 100 inférieurs ou égaux à 10^9 ?

Exercice II.7 – Projet Euler – problème 539

p. 368

Start from an ordered list of all integers from 1 to n . Going from left to right, remove the first number and every other number^a afterward until the end of the list. Repeat the procedure from right to left, removing the rightmost number and every other number from the numbers left. Continue removing every other number, alternating left to right and right to left, until a single number remains.

Starting with $n = 9$, we have :

- $\overrightarrow{1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9}$
- $\overleftarrow{2 \ 4 \ 6 \ 8}$
- $\overrightarrow{2 \ 6}$
- $\boxed{6}$

Let $P(n)$ be the last number left starting with a list of length n .

$$\text{Let } S(n) = \sum_{k=1}^n P(k).$$

You are given $P(1) = 1$, $P(9) = 6$, $P(1000) = 510$, $S(1000) = 268271$.

Find $S(10^{18}) \pmod{987654321}$.

On pourra par exemple commencer par chercher une expression de $P(4k+r)$ avec $0 \leq r < 4$ en fonction de $P(k)$. Cela devrait permettre de calculer très rapidement $P(n)$ même pour de grandes valeurs de n ; il restera alors un peu de travail (essentiellement mathématique) pour accélérer le calcul de $S(n)$.

a. every other number : un nombre sur deux

Solutions

Correction de l'exercice II.1 page 361

1. La version la plus élégante :

```
let rec mem x u =
  match u with
  | [] -> false
  | y :: ys -> (x = y) || mem x ys
```

il faut remarquer que $(x = y) \parallel \text{mem } x \text{ ys}$ est strictement équivalent à **if** $x = y$ **then true else** $\text{mem } x \text{ ys}$: en particulier, l'appel $\text{mem } x \text{ ys}$ n'est effectué que si x est différent de y . tout **if...then..else** ayant une branche constante (comme **then false** ou **else true** par exemple) peut être éliminé et remplacé par une simple expression booléenne (faisant intervenir les connecteurs \parallel , $\&\&$ et not).

2. Deux possibilités (parfaitement équivalentes) parmi d'autres :

```
let rec nth u n =
  match n, u with
  | 0, x :: xs -> x
  | _, x :: xs -> nth xs (n - 1)
  | _ -> failwith "nth : pas assez d'éléments"

let rec nth u n =
  match u with
  | [] -> failwith "nth : pas assez d'éléments"
  | x :: xs -> if n = 0 then x else nth xs (n - 1)
```

3. Une version relativement concise :

```
let rec take n u =
  match n, u with
  | 0, _ | _, [] -> []
  | _, x :: xs -> x :: take (n - 1) xs
```

4. Un **if...then...else** est plus naturel qu'un **match** ici :

```
let rec range a b =
  if a >= b then []
  else a :: range (a + 1) b
```

Correction de l'exercice II.2 page 361

1. Nous avons déjà programmé cette fonction :

```
let rec concat u v =
  match u with
  | [] -> v
  | x :: xs -> x :: (concat xs v)
```

2. On fait une opération cons (notée ::) par élément de u , donc $|u|$ au total.
3. À nouveau, cette question a déjà été traitée :

```
let rec miroir_naif = function
| [] -> []
| x :: xs -> concat (miroir_naif xs) [x]
```

4. Notons $T(n)$ le nombre d'opérations cons effectuées pour une liste u de longueur n . On a $T(0) = 0$, et $T(n) = n - 1 + T(n - 1)$ sinon. En effet, l'appel à concat se fait avec un premier argument (`miroir_naif xs`) de longueur $n - 1$, donc nécessite $n - 1$ cons, et il faut également compter l'appel récursif à `miroir_naif` sur une liste de longueur $n - 1$ (qui fait $T(n - 1)$ cons par définition). En sommant la relation $T(i) - T(i - 1) = i - 1$ pour i variant de 1 à n , on obtient $T(n) - T(0) = \sum_{i=1}^n (i - 1)$, et donc $T(n) = \frac{n(n-1)}{2}$.
5. Pas de difficulté :

```
let rec rev_append u v = match u with
| [] -> v
| x :: xs -> rev_append xs (x :: v)
```

6. C'est immédiat :

```
let miroir u = rev_append u []
```

`rev_append u v` fait un cons par élément de u , donc au total $|u|$. On en déduit que `miroir u` fait exactement $|u|$ appels au constructeur `cons` : cette fois, la complexité est linéaire.

Correction de l'exercice II.3 page 361

1. Le plus naturel est sans doute d'écrire :

```
let rec applique f l =
match l with
| [] -> []
| x :: xs -> (f x) :: (applique f xs)
```

En réalité, cette version est problématique si f a des *effets de bord* car l'ordre d'évaluation n'est pas défini. On peut ignorer ce problème pour l'instant, mais je donne quand même la « bonne » version :

```
let rec applique f l =
match l with
| [] -> []
| x :: xs -> let y = f x in y :: (applique f xs)
```

2. En une ligne :

```
let liste_carres u = List.map (fun x -> x * x) u
```

Cette définition est de la forme `let f u = g x u`; on peut éventuellement simplifier :

```
let liste_carres = List.map (fun x -> x * x)
```

Correction de l'exercice II.4 page 362

1.

```
let rec sans_doublons_triee = function
| [] | [_] -> true
| x :: y :: tail -> (x <> y) && sans_doublons_triee (y :: tail)
```

2.

```
let rec elimine_doublons_triee u =
match u with
| [] | [_] -> u
| x :: y :: tail ->
  if x = y then elimine_doublons_triee (y :: tail)
  else x :: elimine_doublons_triee (y :: tail)
```

3.

```
let rec sans_doublons = function
| [] -> true
| x :: xs -> not (mem x xs) && sans_doublons xs
```

4.

```
let rec elimine_doublons = function
| [] -> []
| x :: xs ->
  if mem x xs then elimine_doublons xs
  else x :: elimine_doublons xs
```

5. La fonction `sans_doublons_triee` va comparer chaque élément de la liste (sauf le dernier) avec l'élément suivant, jusqu'à trouver deux éléments égaux ou atteindre la fin de la liste. Au pire, c'est-à-dire s'il n'y a pas d'éléments consécutifs répétés, elle effectuera donc $n - 1$ comparaisons.
6. Le pire cas est à nouveau celui où il n'y a pas de répétition et où l'on va donc jusqu'à la fin de la liste. Dans ce cas, le premier élément sera comparé aux $n - 1$ suivants, le deuxième aux $n - 2$ suivants et ainsi de suite. Le nombre total de comparaisons est donc :

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

Correction de l'exercice II.5 page 362

1. La version la plus simple :

```
let rec compresse u =
match u with
| [] -> []
| x :: xs -> begin match compresse xs with
  | [] -> [(x, 1)]
  | (y, n) :: tail when x = y -> (y, n + 1) :: tail
  | u -> (x, 1) :: u
end
```

On peut aussi utiliser une fonction auxiliaire, où :

- `a_traiter` est la partie de la liste qui reste à compresser;
- `courant` est la valeur du dernier élément lu;
- `nb_courant` est le nombre de répétitions de cet élément;
- `res` est la version compressée de la partie déjà traitée de la liste, à l'envers.

```
let compresse_2 u =
  let rec aux a_traiter courant nb_courant res =
    match a_traiter with
    | [] -> (courant, nb_courant) :: res
    | x :: xs when x = courant -> aux xs courant (nb_courant + 1) res
    | x :: xs -> aux xs x 1 ((courant, nb_courant) :: res) in
  match u with
  | [] -> []
  | x :: xs -> List.rev (aux xs x 1 [])
```

2. Pour la décompression, le plus simple est d'écrire :

```
let rec decompresse u =
  (* ajoute x n res = [x; ... ; x] @ res (n copies de x *))
  let rec ajoute x n res =
    if n = 0 then res
    else ajoute x (n - 1) (x :: res) in
  match u with
  | [] -> []
  | (x, n) :: tail -> ajoute x n (decompresse tail)
```

Correction de l'exercice II.6 page 362

Il faut commencer par récupérer la liste des nombres premiers inférieurs ou égaux à 100, dans l'ordre croissant. Le plus efficace est un crible d'Ératosthène, mais c'est très délicat à faire sans tableaux mutables et l'efficacité n'a aucune importance ici (on ne va que jusqu'à 100...).

```
let rec enleve_multiples p = function
| [] -> []
| n :: ns when n mod p = 0 -> enleve_multiples p ns
| n :: ns -> n :: enleve_multiples p ns

let premiers n =
  let rec filtre = function
    | [] -> []
    | p :: xs -> p :: filtre (enleve_multiples p xs) in
  (* candidats = [2; 3; 4; ...; n] *)
  let candidats = range 2 (n + 1) in
  filtre candidats
```

Ensuite, le code est extrêmement simple. Un nombre de Hamming ayant tous ses facteurs premiers dans `p :: ps` et inférieur ou égal à `n` est :

- soit de la forme $p \cdot q$, avec q ayant tous ses facteurs premiers dans $p :: ps$ et inférieur ou égal à n/p ;
- soit de la forme q , avec q ayant tous ses facteurs premiers dans ps et inférieur ou égal à n .

De plus, ces deux ensembles sont disjoints, puisque tous les nombres de la première forme sont multiples de p alors que ce n'est le cas d'aucun nombre de la deuxième forme. En ajoutant

le traitement du cas de base, on obtient :

```
let rec hamming prem borne =
  match prem with
  | [] -> 1
  | x :: xs -> let h = hamming xs borne in
    if x <= borne then h + hamming prem (borne / x)
    else h

let p204 k borne =
  let prem = premiers k in
  hamming prem borne
```

Le résultat demandé est :

```
# p204 100 1_000_000_000;;
- : int = 2944730
```

Correction de l'exercice II.7 page 363

- Si l'on part de $n = 4k$ (avec $k \geq 1$), on obtient après un aller retour $2 \ 6 \ 10 \dots 4k - 2$. Cette séquence est égale à $4 \times (1 \ 2 \ 3 \dots k) - 2$, donc tout va maintenant se passer comme si on était parti de $1 \ 2 \dots k$, sauf que chaque valeur x a été remplacée par $4x - 2$. On obtiendra donc $4P(k) - 2$ à la fin du processus.
- En traitant de la même manière les autres cas, on obtient :

$$\begin{aligned} P(4k) &= 4P(k) - 2 \\ P(4k+1) &= 4P(k) - 2 \\ P(4k+2) &= 4P(k) \\ P(4k+3) &= 4P(k) \end{aligned}$$

- On a donc $\sum_{i=4k}^{4k+3} P(i) = 16P(i) - 4$.
- On obtient alors, pour $k \geq 1$:

$$\begin{aligned} \sum_{i=1}^{4k+r} P(i) &= \underbrace{S(3)}_5 + \sum_{i=4}^{4k-1} P(i) + \underbrace{\sum_{i=4k}^{4k+r} P(i)}_R \\ &= 5 + R + \sum_{j=1}^{k-1} (P(4j) + P(4j+1) + P(4j+2) + P(4j+3)) \\ &= 5 + R + \sum_{j=1}^{k-1} (16P(j) - 4) \\ &= 5 + R + 16S(k-1) - 4(k-1) \end{aligned}$$

- On en déduit le code ci-dessous, où l'on a bien fait attention à réduire modulo m à chaque étape.

```
let m = 987654321

let modm a =
  let x = a mod m in
  if x < 0 then x + m
  else x

let (++) a b = modm (a + b)
let (***) a b = modm (a * b)
let (--) a b = modm (a - b)

let rec p n =
  if n = 0 then 0
  else if n = 1 then 1
  else if n = 2 then 2
  else if n = 3 then 2
  else
    let y = p (n / 4) in
    if n mod 4 < 2 then 4 ** y -- 2
    else 4 ** y

let rec s n =
  let k = n / 4 in
  (* on réutilise la fonction range définie plus haut *)
  let derniers_termes = List.map p (range (4 * k) (n + 1)) in
  let fin_somme = List.fold_left (++) 0 derniers_termes in
  if k = 0 then fin_somme
  else 5 ++ fin_somme ++ 16 ** s (k - 1) -- 4 ** (k - 1)
```

La réponse demandée est obtenue instantanément :

```
# s 1_000_000_000_000_000;;
- : int = 426334056
```

TABLEAUX, ORDRE SUPÉRIEUR

I Manipulations élémentaires de tableaux

Exercice III.1

p. 374

1. Écrire une fonction `extrema` : `int array -> int * int` qui prend en entrée un tableau `t` (supposé non vide) et renvoie le couple (`min t, max t`).

```
# extrema [| 4; 1; 7; 2; 10; 6|];
- : int * int = (1, 10)
```

2. a. Écrire une fonction `nb_occ` : `'a -> 'a array -> int` telle que `nb_occ x t` renvoie le nombre d'éléments de `t` égaux à `x`.

```
# nb_occ 5 [|1; 2; 5; 4; 5; 12; 18|];
- : int = 2
```

- b. En utilisant cette fonction, écrire une fonction `tab_occ` : `int array -> int array` telle que l'appel `tab_occ t` renvoie un tableau `occ` :

- ayant la même longueur `n` que `t`;
- vérifiant `occ.(i) = nb_occ i t` pour $0 \leq i < n$.

```
# tab_occ [|1; 2; 5; 4; 5; 12; 18|];
- : int array = [|0; 1; 1; 0; 1; 2; 0|]
```

- c. Donner, en fonction de `n`, l'ordre de grandeur du nombre d'opérations effectuées par l'appel `tab_occ t` sur un tableau de longueur `n`.

- d. Écrire une fonction `tab_occ_eff` ayant la même spécification que `tab_occ` mais effectuant un nombre d'opérations proportionnel à `n`.

Exercice III.2

p. 375

1. Écrire une fonction `sommes_cumulees` : `int array -> int array` telle que, si `t` est un tableau de taille `n`, l'appel `sommes_cumulees t` renvoie le tableau :

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

```
# sommes_cumulees [|2; 1; 0; 7; 10|];
- : int array = [|2; 3; 3; 10; 20|]
```

2. Combien d'additions cette fonction effectue-t-elle? Si c'est plus que $|t|$, la réécrire.

Exercice III.3

p. 375

On demande ici de réécrire quelques fonctions de la bibliothèque standard. On fera très attention à bien obtenir les types demandés : si ce n'est pas le cas (si le type obtenu est moins général), c'est qu'il y a un problème.

Remarque

Le type d'un tableau est fixé à la création. Même si l'on compte modifier ultérieurement tous les éléments du tableau, il faut donc l'initialiser avec un élément du bon type.

1. `map` : `('a -> 'b) -> 'a array -> 'b array` agissant comme `Array.map`.

```
# Array.map (fun i -> 2 * i - 1) [|2; 4; 1; 5|];
- : int array = [|3; 7; 1; 9|]
```

2. `init` : `int -> (int -> 'a) -> 'a array` agissant comme `Array.init`, c'est-à-dire telle que `init n f` renvoie `[|f 0; f 1; ...; f (n - 1)|]`.

```
# Array.init 5 (fun i -> i * i);
- : int array = [|0; 1; 4; 9; 16|]
```

3. `to_list` : `'a array -> 'a list` agissant comme `Array.to_list`.

```
# Array.to_list [|2; 5; 1|];
- : int list = [2; 5; 1]
```

4. `of_list` : `'a list -> 'a array` agissant comme `Array.of_list`.

```
# Array.of_list [2; 5; 1];
- : int array = [|2; 5; 1|]
```

On complétera le squelette suivant :

```
let of_list u =
  match u with
  | [] -> ...
  | x :: xs ->
    let n = List.length u in
    let t = Array.make ... in
    (* aux v k renvoie t dans lequel les éléments de v ont été
     * recopier à partir de l'indice k (t.(k) reçoit l'élément de
     * tête de v) *)
    let rec aux v k =
      match v with
      | [] -> ...
      | y :: ys -> ... in
    ...
```

2 Fonctions d'ordre supérieur : fold

Exercice III.4 – Fold

p. 376

1. Écrire des fonctions somme et produit calculant respectivement la somme et le produit des éléments d'une liste d'entiers.
2. Écrire une fonction aplatit : '`a list list -> 'a list`' telle que `aplatit [u1; u2; ...; un]` renvoie `u1 @ u2 @ ... @ un` (chacun des u_i est une liste). *Cette fonction est pré définie : c'est `List.flatten`.*
3. Écrire une fonction `max_liste` : '`int list -> int`' calculant le maximum des éléments d'une liste d'entiers. On conviendra que le maximum de la liste vide est égal à la constante pré définie `min_int`.
4. Écrire une fonction `reduction` ayant la spécification suivante :
 - `reduction : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
 - `reduction f [a1, ..., an] b = f a1 (f (a2 ... (f an b)))`
 - `reduction f [] b = b`
 On pourra commencer par exprimer `reduction [a1; ...; an] b` à l'aide de `reduction [a2; ...; an] b`.

Cette fonction est pré définie : c'est `List.fold_right`. Il existe également une fonction `List.fold_left` légèrement différente, nous y reviendrons plus tard.
5. Réécrire les fonctions somme, produit, aplatit et `max_liste` en utilisant la fonction `reduction`.

Exercice III.5 – Encore du fold

p. 377

1. Définir une fonction `longueur` en utilisant `reduction` (une ligne).
2. Écrire une fonction calculant la variance d'une liste de flottants sans utiliser de `rec` (mais avec des `reduction` et des `List.map`, bien sûr).

On rappelle que la variance est la moyenne des carrés des écarts à la moyenne.

```
# variance [ 2.; 3.; 7. ];;
- : float = 4.6666666666666696
```

3. Écrire une fonction `nb_occs` : '`'a -> 'a list -> int`' telle que `nb_occs x u` renvoie le nombre d'éléments de `u` qui sont égaux à `x`. Cette fonction utilisera `reduction`, et tiendra sur une ligne.

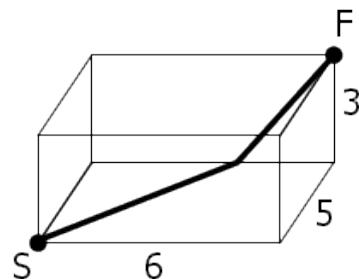
3 Si vous avez fini

Exercice III.6 – Projet Euler – Problème 86

p. 378

A spider, S , sits in one corner of a cuboid room, measuring 6 by 5 by 3, and a fly, F , sits in the opposite corner. By travelling on the surfaces of the room the shortest “straight line” distance from S to F is 10 and the path is shown on the diagram.

However, there are up to three “shortest” path candidates for any given cuboid and the shortest route doesn’t always have integer length.



It can be shown that there are exactly 2060 distinct cuboids, ignoring rotations, with integer dimensions, up to a maximum size of M by M by M , for which the shortest route has integer length when $M = 100$. This is the least value of M for which the number of solutions exceeds two thousand; the number of solutions when $M = 99$ is 1975.

Find the least value of M such that the number of solutions exceeds one million.

Solutions

Correction de l'exercice III.1 page 370

1.

```
let extrema t =
  let n = Array.length t in
  let mini = ref t.(0) in
  let maxi = ref t.(0) in
  for i = 0 to n - 1 do
    mini := min !mini t.(i);
    maxi := max !maxi t.(i)
  done;
  (!mini, !maxi)
```

2. a.

```
let nb_occs x t =
  let n = Array.length t in
  let count = ref 0 in
  for i = 0 to n - 1 do
    if t.(i) = x then incr count
  done;
  !count
```

b.

```
let tab_occs t =
  let n = Array.length t in
  let occs = Array.make n 0 in
  for i = 0 to n - 1 do
    occs.(i) <- nb_occs i t
  done;
  occs
```

- c. Un appel à `nb_occs i t` fait de l'ordre de n opérations, et l'on fait n tels appels, donc de l'ordre de n^2 opérations. Pour le reste, le calcul de `Array.length t` prend un temps constant et la création du tableau `occs` un temps proportionnel à n : on a donc de l'ordre de n^2 opérations au total.
- d. À chaque fois que l'on lit un x (entre 0 et $n - 1$) dans le tableau, on incrémente la case d'indice x de `occs` de 1 :

```
let tab_occs_efficace t =
  let n = Array.length t in
  let occs = Array.make n 0 in
  for i = 0 to n - 1 do
    let x = t.(i) in
    if 0 <= x && x < n then
      occs.(x) <- occs.(x) + 1
  done;
  occs
```

Correction de l'exercice III.2 page 370

Il faut éviter de recalculer entièrement la somme à chaque étape :

```
let sommes_cumulees t =
  let n = Array.length t in
  let u = Array.make n t.(0) in
  for i = 1 to n - 1 do
    u.(i) <- u.(i - 1) + t.(i)
  done;
  u
```

Correction de l'exercice III.3 page 371

- Le plus naturel est d'utiliser une boucle. Il nous faut un élément du bon type pour initialiser le tableau : il nous est fourni par l'appel `f t.(0)`, sauf si `t` est vide.

```
let map f t =
  let n = Array.length t in
  if n = 0 then []
  else begin
    let s = Array.make n (f t.(0)) in
    for k = 1 to n - 1 do
      s.(k) <- f t.(k)
    done;
    s
  end
```

- Fonction très similaire :

```
let init n f =
  if n = 0 then []
  else begin
    let t = Array.make n (f 0) in
    for k = 1 to n - 1 do
      t.(k) <- f k
    done;
    t
  end
```

- Pour `to_list`, on est amené à manipuler simultanément des listes et des tableaux. Pour les listes, on préfère généralement la récursion, pour les tableaux, plutôt l'itération : ici, les deux choix sont donc raisonnables.

```
(* Version récursive *)
let to_list t =
  (* aux t k renvoie [t.(k); t.(k + 1); ...; t.(n - 1)] *)
  let rec aux t k =
    if k = Array.length t then []
    else t.(k) :: aux t (k + 1) in
    aux t 0
```

```
(* Version itérative, il faut faire attention à mettre
   les éléments dans le bon ordre. *)
let to_list_bis t =
  let u = ref [] in
  for i = Array.length t - 1 downto 0 do
    u := t.(i) :: !u
  done;
  !u
```

4. Pour la fonction `of_list`, il est nécessaire de connaître la longueur de la liste au moment de créer le tableau. Le plus simple est de compléter le squelette fourni par l'énoncé :

```
let of_list u =
  match u with
  | [] -> []
  | x :: xs ->
    let n = List.length u in
    let t = Array.make n x in
    let rec aux v k =
      match v with
      | [] -> t
      | y :: ys -> t.(k) <- y; aux ys (k + 1) in
    aux xs 1
```

On effectue deux parcours de `u` : un pour calculer sa longueur, puis un pour recopier les éléments. On peut *plus ou moins* se passer du deuxième parcours avec la version, très astucieuse et absolument pas recommandée, suivante :

```
let of_list_bis u =
  (* aux [u1; ...; up] k renvoie un tableau de taille k + p
     égal à [| up; ... ; up; u1; u2; ...; up |]
     (ou un tableau vide si u est vide). *)
  let rec aux u k =
    match u with
    | [x] -> Array.make (k + 1) x
    | x :: xs -> let t = aux xs (k + 1) in
                  t.(k) <- x;
                  t
    | [] -> [] [] in
  aux u 0
```

Correction de l'exercice III.4 page 372

1. Les deux fonctions sont presque identiques :

```
let rec somme l =
  match l with
  | [] -> 0
  | x :: xs -> x + somme xs

let rec produit l =
  match l with
  | [] -> 1
  | x :: xs -> x * produit xs
```

On peut remarquer que l'on peut ré-écrire la fonction `somme` (et de manière similaire la fonction `produit`) ainsi :

```
let rec somme l =
  match l with
  | [] -> 0
  | x :: xs -> ( + ) x (somme xs)
```

En général, cela n'a pas d'intérêt, mais ici cela permet de mettre en évidence la structure identique des différentes fonctions demandées.

2. C'est à nouveau presque la même chose :

```
let rec applatit = function
| [] -> []
| u :: us -> ( @ ) u (applatit us)
```

3. C'est encore la même chose :

```
let rec max_liste l =
  match l with
  | [] -> min_int
  | x :: xs -> max x (max_liste xs)
```

On tire parti du fait que `max min_int n` est égal à `n` pour tout `n` de type `int`.

4. Essentiellement, on extrait la structure commune aux exemples précédents :

```
let rec reduction f liste init =
  match liste with
  | [] -> init
  | x :: xs -> f x (reduction f xs init)
```

5. Une ligne par fonction :

```
let somme_fold u = reduction ( + ) u 0

let produit_fold u = reduction ( * ) u 1

let applatit_fold u = reduction ( @ ) u []

let max_fold u = reduction max u min_int
```

Correction de l'exercice III.5 page 372

- 1.

```
let longueur_fold u = reduction (fun y acc -> acc + 1) u 0
```

- 2.

```

let variance u =
  let somme_float v = reduction ( +. ) v 0. in
  let moyenne v = (somme_float v) /. (float (longueur_fold v)) in
  let m = moyenne u in
  let ecarts_quadratiques = List.map (fun x -> (x -. m) ** 2.) u in
  moyenne ecarts_quadratiques

```

3.

```

let nb_occs_fold x u =
  reduction (fun y acc -> acc + if x = y then 1 else 0) u 0

```

Correction de l'exercice III.6 page 373

On propose ici une solution simple mais absolument pas optimale, qui permet cependant d'obtenir très rapidement le résultat cherché (qui vaut 1818). Le code est donné tel quel, si vous êtes arrivé jusqu'ici, vous êtes capable de vous débrouiller...

```

let squares = Array.init 10_000 (fun i -> i * i)

let is_square x =
  let rec aux i j =
    let m = (i + j) / 2 in
    i < j
    &&
    (squares.(m) = x
     || if squares.(m) < x then aux (m + 1) j else aux i m) in
  aux 0 (Array.length squares)

let fixed_c c =
  let n = ref 0 in
  let c2 = c * c in
  for a_plus_b = 2 to 2 * c do
    if is_square (c2 + a_plus_b * a_plus_b) then
      let nb = a_plus_b / 2 + 1 - max 1 (a_plus_b - c) in
      n := !n + nb
  done;
  !n

let first_greater bound =
  let n = ref 0 in
  let c = ref 0 in
  while !n <= bound do
    incr c;
    n := !n + fixed_c !c;
  done;
  (!n, !c)

(*
# first_greater 1_000_000;;
- : int * int = (1000457, 1818)
*)

```

TRI INSERTION, TRI FUSION

I Recherche dichotomique

Exercice IV.1 – Recherche dichotomique en OCaml

p. 387

- Sans se référer au cours, compléter l'algorithme ci-dessous pour qu'il vérifie la spécification suivante :

Entrées : un tableau $t = (t_0, \dots, t_{n-1})$ et une valeur x

Précondition : t est trié par ordre croissant

Résultat : un indice $i \in [0 \dots n - 1]$ tel que $t_i = x$ s'il en existe un, n sinon.

Algorithme 15 – Recherche dichotomique

```

fonction RECHERCHE(x, t)
    deb ← ...
    fin ← ...                                ▷ Recherche entre deb inclus et fin exclu.
    tant que ... faire
        milieu ← ...
        si  $t_{\text{milieu}} = x$  alors
            ...
        sinon si  $t_{\text{milieu}} < x$  alors
            ...
        sinon
            ...
    ...

```

- Comme signalé en cours, l'impossibilité de faire un « `return` » au milieu de la boucle en OCaml incite à transformer la boucle `while` en une fonction auxiliaire récursive. De plus, il est plus logique en OCaml de renvoyer `Some` si l'on a trouvé x à l'indice i et `None` si on ne l'a pas trouvé. En tenant compte de ces remarques, et en s'aidant du squelette ci-dessous, écrire une fonction `cherche_dicho` : '`a` `array` -> '`a` -> `int option`.

```

let cherche_dicho t x =
    (* aux deb fin cherche x dans t entre deb inclus
       et fin exclu *)
    let rec aux deb fin =
        if deb >= fin then ...
        else
            let mil = (deb + fin) / 2 in
            ...
            ...
            ... in
            aux 0 (Array.length t)

```

- Tester votre fonction en utilisant ce qui est fourni dans le squelette de TP. Vérifier que vous comprenez bien ce que fait la fonction de test, et regarder ce qui se passe si vous introduisez une erreur dans la fonction `cherche_dicho`.

2 Tri

Dans toute cette partie, on suppose que les éléments des séquences que l'on manipule sont choisis dans un ensemble totalement ordonné.

2.1 Notations et spécifications

Définition IV.1 – Séquence

Une *séquence* est une collection d'éléments dans laquelle l'ordre des éléments et leur nombre d'occurrences compte.

Pour une séquence $s = s_0, \dots, s_{n-1}$, on notera :

- $|s| = n$ la longueur de la séquence (son nombre d'éléments);
- $|s|_x = \text{Card}\{i \in [0 \dots n-1] \mid s_i = x\}$ le nombre d'occurrences de x dans s .

On notera ε la séquence vide (de longueur 0).

Remarques

- Mathématiquement, une séquence correspond à un n -uplet.
- Informatiquement, une séquence correspondra le plus souvent à une liste ou un tableau (unidimensionnel).
- Deux séquences x_0, \dots, x_{n-1} et y_0, \dots, y_{p-1} sont égales si et seulement si $n = p$ et $x_i = y_i$ pour tout $0 \leq i < n$.

Exemple IV.2

Si $s = (12, 4, 0, 4, 4, 12)$, on a :

- | | |
|--|---|
| <ul style="list-style-type: none"> ■ $s = 6$; ■ $s _0 = 1$; | <ul style="list-style-type: none"> ■ $s _1 = s _2 = s _3 = s _\pi = 0$; ■ $s _4 = 3$ |
|--|---|

On a $s \neq (12, 4, 0, 4, 12)$ (4 n'a pas le même nombre d'occurrences) et $s \neq (4, 12, 0, 4, 4, 12)$ (l'ordre des éléments n'est pas le même).

Définition IV.2 – Concaténation

On définit la *concaténation* de deux séquences $u = (u_0, \dots, u_{n-1})$ et $v = (v_0, \dots, v_{p-1})$, que l'on note $u + v$, par :

$$u + v = (u_0, \dots, u_{n-1}, v_0, \dots, v_{p-1})$$

Remarque

Attention, cette notation est courante mais pas totalement standard.

Définition IV.3 – Notation de tranchage

Si $s = (s_0, \dots, s_{n-1})$, on définit :

- $s[d : f] = (s_d, \dots, s_{f-1})$ si $0 \leq d, f \leq n$;
- $s[:] = s[0 : f]$;
- $s[d :] = s[d : n]$.

Remarques

- Cette notation correspond à l'opération de *slicing* en Python (ni C ni OCaml ne proposent de syntaxe dédiée pour cette opération).
- $s[d : f]$ est vide si $d \geq f$.
- Si $d \leq f$, alors $s[d : f]$ est de longueur $f - d$.

Exercice IV.3

p. 387

1. On considère la séquence $s = (10, 20, 30, 40, 40, 30)$. Que valent :
 - a. $s[1 : 4]$?
 - b. $s[: 3]$?
 - c. $s[2 :]$?
 - d. $s[4 : 6]$?
 - e. $s[4 : 4]$?
 - f. $s[4 : 2]$?
2. Pour une séquence s quelconque, comment définir avec cette notation et éventuellement la concaténation :
 - a. la séquence correspondant aux i premiers éléments de s ?
 - b. la séquence correspondant à s privée de ses i premiers éléments ?
 - c. la séquence correspondant aux i derniers éléments de s ?
 - d. la séquence correspondant à s privée de ses i derniers éléments ?
 - e. la séquence correspondant à s privée de son élément d'indice i ?

Définition IV.4 – Multi-ensemble

Un *multi-ensemble* est une collection pour laquelle l'ordre des éléments n'a pas d'importance mais le nombre d'occurrences compte. On notera $\text{ms}(x_0, \dots, x_{n-1})$ le multi-ensemble associé à la collection x_0, \dots, x_{n-1} .

Remarques

- On parle aussi de *multiset* ou de *bag*, ou parfois, dans un contexte mathématique, de *combinaison avec répétitions*.
- Deux multi-ensembles s et t (dont les éléments sont choisis dans un même ensemble A) sont égaux si et seulement si $|s|_x = |t|_x$ pour tout $x \in A$.
- Autrement dit, deux multi-ensembles sont égaux si et seulement si ils ont les mêmes éléments, à l'ordre près mais en tenant compte du nombre d'occurrences.

Exemple IV.4

Si $s = \text{ms}(12, 4, 0, 4, 4, 12)$, on a :

- $|s| = 6$;
- $|s|_0 = 1$;
- $|s|_1 = |s|_2 = |s|_3 = |s|_\pi = 0$;
- $|s|_4 = 3$
- $s = \text{ms}(4, 12, 0, 4, 4, 12) = \text{ms}(0, 4, 4, 4, 12, 12)$;
- $s \neq \text{ms}(4, 0, 4, 4, 12)$ (12 n'a pas le même nombre d'occurrences).

Définition IV.5

Pour une séquence s , on définit :

- $\text{isSorted}(s)$ le prédictat qui vaut vrai si et seulement si

$$s_0 \leq s_1 \leq \dots \leq s_{|s|-1}$$

- $\text{sorted}(s)$ l'unique séquence telle que

$$\begin{cases} \text{isSorted}(\text{sorted}(s)) & \text{sorted}(s) \text{ est triée} \\ \text{ms}(\text{sorted}(s)) = \text{ms}(s) & \text{sorted}(s) \text{ contient les mêmes éléments que } s \end{cases}$$

Remarque

La séquence vide est triée (par convention, si l'on veut). Autrement dit, on a $\text{isSorted}(\varepsilon)$.

Exemple IV.5

- $\text{isSorted}(3, 1, 2)$ est faux.
- $\text{isSorted}(1, 2, 3)$ est vrai, tout comme $\text{isSorted}(1)$ et $\text{isSorted}(1, 1, 1)$.
- $\text{sorted}(3, 1, 2, 3) = \text{sorted}(3, 3, 1, 2) = \text{sorted}(1, 2, 3, 3) = (1, 2, 3, 3)$.

2.2 Tri par insertion**Exercice IV.6 – Tri par insertion d'une liste**

p. 387

Le tri par insertion est un tri très simple à implémenter qui possède plusieurs variantes. On s'intéresse ici à la version la plus naturelle en OCaml, dont le principe est le suivant :

- une liste vide est déjà triée ;
- sinon, la liste à trier est de la forme $x :: xs$. On trie xs (par un appel récursif), puis on insère x à la bonne place dans cette liste triée.

1. Écrire une fonction `insere` prenant en entrée une liste v **supposée triée par ordre croissant** et un élément x , et renvoyant une nouvelle version de v dans laquelle x a été inséré à la bonne place (pour que la liste reste croissante).

```
insere : 'a list -> 'a -> 'a list
```

```
# insere [1; 3; 4; 6; 7] 5;;
- : int list = [1; 3; 4; 5; 6; 7]
```

2. Formaliser la spécification de la fonction `insere` en utilisant les concepts et notations de la partie 2.1
3. Écrire une fonction `tri_insertion`, fonctionnant suivant le principe exposé ci-dessus, et ayant la spécification suivante :

Entrée : une liste u (dont les éléments sont comparables).

Sortie : une liste v telle que $\text{sorted}(u) = v$.

```
tri_insertion : 'a list -> 'a list
```

4. Déterminer précisément le nombre de comparaisons entre éléments qu'effectue la fonction `tri_insertion` quand on l'appelle sur une liste de longueur n dans le pire des cas et dans le meilleur cas.

Exercice IV.7 – Tri insertion d'un tableau

p. 388

Le tri par insertion d'un tableau utilise fondamentalement le même principe que celui d'une liste mais la traduction de ce principe est assez différente.

Pour commencer, ce tri va s'effectuer *en place*, ce qui signifie que l'on va modifier le tableau passé en argument (pour le rendre trié) et qu'on n'utilisera pas de stockage auxiliaire. Autrement dit, on va chercher à écrire une fonction

```
tri_insertion_tableau : 'a array -> unit
```

1. Donner la spécification de la fonction `tri_insertion_tableau`.

Cette fonction va avoir la structure suivante :

```

let tri_insertion_tableau t =
  for i = 1 to Array.length t - 1 do
    (* insérer t.(i) dans t[:i] *)
  done

```

En notant t_{init} l'état initial du tableau et t son état actuel, on va maintenir l'invariant suivant (valable au début de l'itération i) :

- $t[:i] = \text{sorted}(t_{\text{init}}[:i])$
- $t[i:] = t_{\text{init}}[i:]$

2. Si l'on part de $t = [4, 1, 2, 5, 0, 3]$, donner l'état de t après les itérations 1, 2, ..., 5 de la boucle.

Une manière de maintenir cet invariant est illustrée ci-dessous :

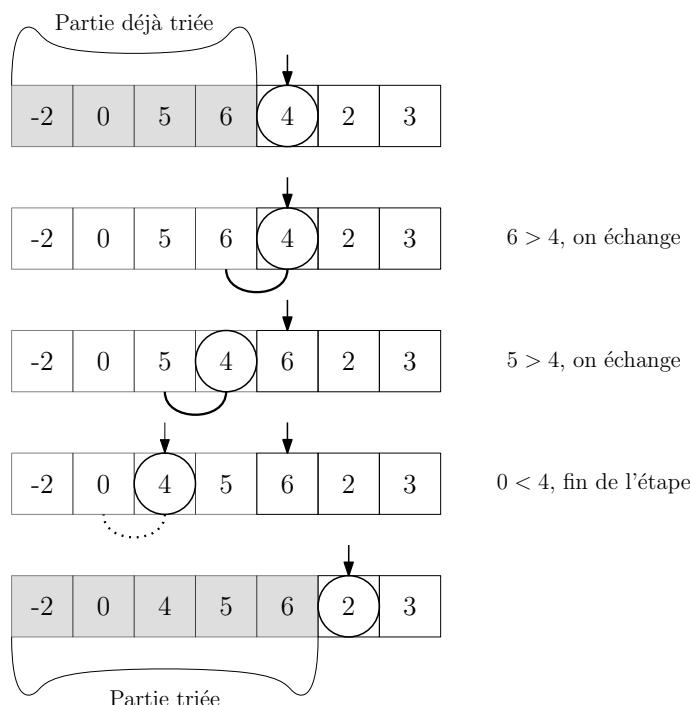


FIGURE IV.1 – Une itération de la boucle externe.

3. Écrire une fonction `échange` : '`a array -> int -> int -> unit`' telle que `échange t i j` échange les éléments d'indice i et j du tableau t .
4. Écrire une fonction `insertion_en_place` : '`a array -> int -> unit`' ayant la spécification suivante :

Entrées : un tableau t et un entier i .

Préconditions :

- $0 < i < |t|$
- $\text{isSorted}(t[:i])$

Effets secondaires : après l'appel, on a en notant t_{init} l'état initial (avant l'appel) de t :

- $\text{ms}(t[:i+1]) = \text{ms}(t_{\text{init}}[:i+1])$
- $\text{isSorted}(t[:i+1])$
- $t[i+1:] = t_{\text{init}}[i+1:]$

5. Écrire la fonction `tri_insertion_tableau`.

6. On note $C(n)$ le nombre maximal de comparaisons entre éléments du tableau effectuées par l'appel `tri_insertion_tableau t` si $|t| = n$, et $E(n)$ le nombre maximal d'écritures dans le tableau. Déterminer $C(n)$ et $E(n)$.

7. Il est possible de remplacer les échanges entre éléments consécutifs par de simples affectations (ce qui accélérera légèrement la fonction) en procédant comme suit :

- on sauvegarde l'élément à insérer dans une variable temporaire ;
- on déplace tous les éléments qui lui sont supérieurs d'une case vers la droite ;
- on le recopie à la bonne place.

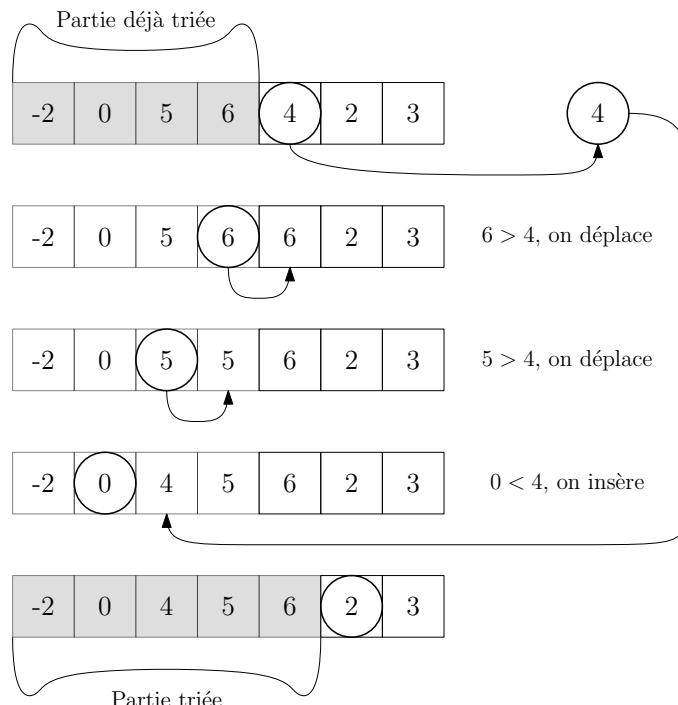


FIGURE IV.2 – Insertion avec deux fois moins d'écritures.

Écrire une nouvelle version de la fonction `insertion_en_place` utilisant ce principe.

3 Tri fusion

Exercice IV.8 – Tri fusion

p. 389

Si l'on dispose de deux listes triées, il est assez aisément de les fusionner en une nouvelle liste également triée, **en temps linéaire en la somme de leurs tailles**. On peut par exemple s'imaginer face à une table sur laquelle sont posées deux piles de copies, triées par ordre alphabétique, et où l'on dispose de la place nécessaire pour créer une troisième pile. À chaque étape, on regarde les deux copies sur le sommet des deux premières piles, on choisit la plus « petite » et on la pose sur la troisième pile.

Ce principe, appliqué récursivement, conduit au *tri par fusion* :

- on sépare la liste u à trier en deux listes g et d de même taille (ou presque, si $|u|$ est impair) ;
- on trie récursivement ces deux listes par fusion ;
- on fusionne les résultats de ces tris.

Les figures IV.3 et IV.4 illustrent le principe.

1. Écrire une fonction `eclate` : `'a list -> ('a list * 'a list)` telle que l'appel `eclate u` renvoie un couple `(ga, dr)` vérifiant :

- $0 \leq |ga| - |dr| \leq 1$
- $\text{ms}(ga + dr) = \text{ms}(u)$

La seule chose que l'on exige, c'est que les éléments de u aient été répartis équitablement entre ga et dr : ils peuvent avoir été mélangés, entrelacés, cela n'a aucune importance.

```
# eclate [7; 2; 4; 1; 9; 10; 2];
- : int list * int list = ([10; 1; 2], [2; 9; 4; 7])
```

2. Écrire la fonction fusionne : ' a list \rightarrow ' a list' \rightarrow ' a list' décrite plus haut. Sa spécification précise est :

Entrées : deux listes u et v .

Précondition : $\text{isSorted}(u)$ et $\text{isSorted}(v)$.

Sortie : une liste w telle que $w = \text{sorted}(u + v)$.

```
# fusionne [1; 1; 3; 5] [2; 3; 7];
- : int list = [1; 1; 2; 3; 3; 5; 7]
```

Il y a au moins deux manières de procéder :

- une qui correspond directement à la situation décrite (trois piles), et qui utilise une fonction auxiliaire de type ' a list \rightarrow ' a list' \rightarrow ' a list' \rightarrow ' a list' (où les deux premiers arguments correspondent aux piles qui restent à fusionner et le troisième à la pile que l'on est en train de construire) ;
 - une, **plus simple et que je vous recommande**, qui s'intéresse simplement à ce que vaut $\text{fusionne}(x :: xs)(y :: ys)$ suivant le résultat de la comparaison de x avec y .
3. Écrire la fonction tri_fusion : ' a list \rightarrow ' a list'. Cette fonction est très simple et correspond directement au schéma de la figure IV.3 (elle utilise eclate et fusionne).

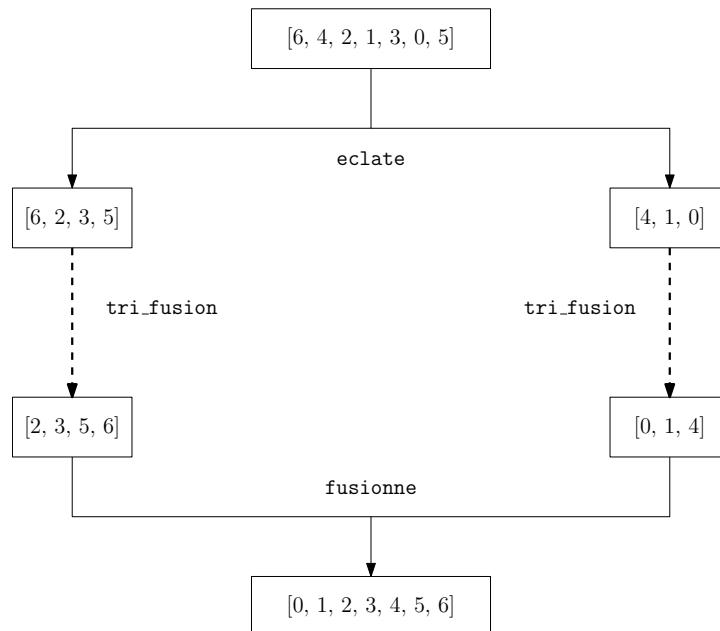


FIGURE IV.3 – Principe du tri fusion

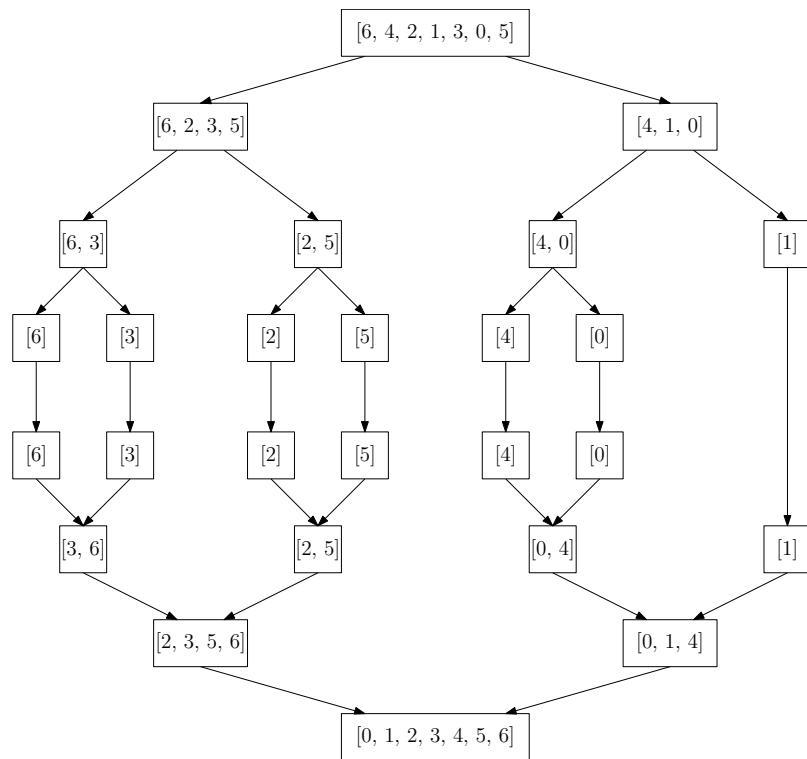


FIGURE IV.4 – Détail du tri fusion

4 Bonus

Exercice IV.9 – Nombre d'occurrences dans un tableau trié

p. 389

Écrire une fonction `nb_occurrences : 'a -> 'a array -> int` telle que `nb_occurrences x t` calcule efficacement $|t|_x$ en supposant que le tableau `t` est trié. Ici, *efficacement* signifie en temps logarithmique en $|t|$, indépendamment du nombre d'occurrences de `x` dans `t`.

Si vous n'avez pas d'idée, n'hésitez pas à me demander une indication.

Exercice IV.10 – Fonction 91 de McCarthy

On considère la fonction suivante, définie (ou pas) pour $n \in \mathbb{Z}$:

$$M(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ M(M(n + 11)) & \text{sinon} \end{cases}$$

Montrer que `M` est totale et déterminer une expression simple de `M(n)` en fonction de $n \in \mathbb{Z}$.
*Si on s'autorise à expérimenter sur ordinateur, trouver l'expression ne présente pas vraiment de difficulté.
 On demande bien sûr une démonstration...*

Solutions

Correction de l'exercice IV.1 page 379

1. Se référer au cours.
- 2.

```
let cherche_dicho t x =
  let rec aux deb fin =
    if deb >= fin then None
    else
      let mil = (deb + fin) / 2 in
      if t.(mil) = x then Some mil
      else if t.(mil) < x then aux (mil + 1) fin
      else aux deb mil in
  aux 0 (Array.length t)
```

Correction de l'exercice IV.3 page 381

1.
 - a. $s[1 : 4] = (20, 30, 40)$
 - b. $s[: 3] = (10, 20, 30)$
 - c. $s[2 :] = (30, 40, 40, 30)$
 - d. $s[4 : 6] = (40, 30)$
 - e. $s[4 : 4] = \varepsilon$
 - f. $s[4 : 2] = \varepsilon$
2.
 - a. $s[: i]$
 - b. $s[i :]$
 - c. $s[|s| - i :]$
 - d. $s[: |s| - i]$
 - e. $s[: i] + s[i + 1 :]$

Correction de l'exercice IV.6 page 382

1. Pas de difficulté :

```
let rec insere m x =
  match m with
  | [] -> [x]
  | y :: ys when x <= y -> x :: m
  | y :: ys -> y :: (insere ys x)
```

2. Entrée : une liste u , un élément x .
Précondition : isSorted(u).
Résultat : une liste v telle que $v = sorted(x :: u)$.
3. La solution la plus simple :

```
let rec tri_insertion m =
  match m with
  | [] -> []
  | x :: xs -> insere (tri_insertion xs) x
```

4. Quand on appelle `insere u x`, on effectuera au pire une comparaison entre `x` et chacun des éléments de la liste `u` (si $x \geq \max u$). La fonction `tri_insertion` fait des appels à `insere` sur des listes de longueur $0, \dots, n-1$ donc le nombre total de comparaison est majoré par $\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$. De plus, si la liste de départ est triée par ordre décroissant, alors on sera dans le pire cas de `insere` à chaque appel : le nombre total de comparaison dans ce cas est donc égal à $\frac{n(n-1)}{2}$.

Correction de l'exercice IV.7 page 382

1. Entrée : un tableau `t`.

Effets secondaires : après l'appel `tri_insertion_tableau t`, le tableau `t` a été modifié de manière à ce que `t = sorted(tinit)` (en notant `tinit` l'état de `t` avant l'appel).

2. On obtient successivement :

- [1, 4, 2, 5, 0, 3]
- [1, 2, 4, 5, 0, 3]
- [1, 2, 4, 5, 0, 3]
- [0, 1, 2, 4, 5, 3]
- [0, 1, 2, 3, 4, 5]

- 3.

```
let echange t i j =
  let x = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- x
```

- 4.

```
let insertion_en_place t i =
  let j = ref i in
  while !j > 0 && t.(!j) < t.(!j - 1) do
    echange t !j (!j - 1);
    decr j
  done
```

5. Tout le travail a déjà été fait :

```
let tri_insertion_tableau t =
  for i = 1 to Array.length t - 1 do
    insertion_en_place t i
  done
```

6. Dans l'appel `insertion_en_place t i`, on passe au maximum `i` fois dans la boucle (si `ti < min t[0:i]`), en faisant à chaque fois une comparaison et un échange, soit deux écritures dans le tableau. Si le tableau est initialement trié par ordre décroissant, on sera à chaque fois dans ce cas, d'où :

$$C(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \quad \text{et} \quad E(n) = 2C(n) = n(n-1)$$

7. Il faut être soigneux, il est très facile d'écrire un code faux ici :

```
let insertion_en_place_bis t i =
  let j = ref i in
  let x = t.(i) in
  while !j > 0 && t.(!j - 1) > x do
    t.(!j) <- t.(!j - 1);
    decr j
  done;
  t.(!j) <- x
```

Correction de l'exercice IV.8 page 384

1. La solution la plus simple :

```
let rec eclate u =
  match u with
  | [] -> ([], [])
  | [x] -> ([x], [])
  | x :: y :: reste -> let a, b = eclate reste in (x :: a, y :: b)
```

Nous verrons plus tard d'autres variantes.

2. À nouveau, on se concentre sur la solution la plus simple :

```
let rec fusionne u v =
  match u, v with
  | [], _ -> v
  | _, [] -> u
  | x :: xs, y :: ys ->
    if x <= y then x :: fusionne xs v
    else y :: fusionne u ys
```

3. Tout le travail a déjà été fait, il suffit de traduire la figure IV.3 :

```
let rec tri_fusion u =
  match u with
  | [] | [_] -> u
  | _ -> let (a, b) = eclate u in
    fusionne (tri_fusion a) (tri_fusion b)
```

Correction de l'exercice IV.9 page 386

On écrit deux fonction auxiliaires, qui prennent toutes les deux un tableau *t* supposé trié et une valeur *x* :

- *premiere_occ* : '*a array* -> '*a* -> *int* qui renvoie l'indice de la *première* occurrence de *x* dans *t*, ou n si $x \notin t$;
- *derniere_occ* : '*a array* -> '*a* -> *int* qui renvoie l'indice de la *dernière* occurrence de *x* dans *t*, ou -1 si $x \notin t$.

Ces deux fonctions effectuent une recherche dichotomique, mais ne s'arrêtent pas dès qu'elles ont trouvé *x* : elles continuent la recherche (dans la partie gauche pour *premiere_occ*, droite pour *derniere_occ*) pour voir s'il n'y a pas une « meilleure » occurrence de *x*.

```
let premiere_occ t x =
  let n = Array.length t in
  let rec aux deb fin =
    if deb >= fin then n
    else
      let mil = (deb + fin) / 2 in
      if t.(mil) = x then min mil (aux deb mil)
      else if t.(mil) < x then aux (mil + 1) fin
      else aux deb mil in
  aux 0 n

let derniere_occ t x =
  let n = Array.length t in
  let rec aux deb fin =
    if deb >= fin then -1
    else
      let mil = (deb + fin) / 2 in
      if t.(mil) = x then max mil (aux (mil + 1) fin)
      else if t.(mil) < x then aux (mil + 1) fin
      else aux deb mil in
  aux 0 n
```

Il ne reste plus qu'à combiner, en n'oubliant pas de traiter correctement le cas où $x \notin t$:

```
let nb_occs_triee t x =
  let premiere = premiere_occ t x in
  let derniere = derniere_occ t x in
  max 0 (derniere - premiere + 1)
```

ANALYSE D'ALGORITHMES

Remarques préliminaires

- Ce sujet comporte de nombreuses questions théoriques : il ne faut bien sûr pas les sauter, et elles ne peuvent être traitées correctement qu'à l'aide d'une feuille et d'un stylo...
- Le squelette fourni contient des tests. Certains sont « vides » (réduits à `let () = assert true`) : cela signifie que c'est à vous de les compléter.

I Suite de Fibonacci

On définit la suite $(F_n)_{n \geq 0}$ par :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_{n+2} = F_n + F_{n+1} \quad \text{pour } n \geq 0 \end{cases}$$

On admet (vous le prouverez en cours de mathématiques) que $F_n = \Theta(\varphi^n)$, où $\varphi = \frac{1+\sqrt{5}}{2} \simeq 1,618$.

Exercice V.1 – Calcul naïf de F_n

p. 395

On considère la fonction suivante, qui calcule F_n pour $n \geq 0$:

```
let rec fib_naif n =
  if n <= 1 then n
  else fib_naif (n - 1) + fib_naif (n - 2)
```

1. On note $\psi(n)$ le nombre d'additions effectuées lors de l'appel `fib_naif n` (en tenant compte de tous les appels récursifs). Montrer que pour $n \geq 0$, on a $\psi(n) = F_{n+1} - 1$.
2. En déduire la complexité de la fonction `fib_naif`. En utilisant cette fonction, peut-on espérer calculer en temps raisonnable F_{10} ? F_{50} ? F_{1000} ?

Exercice V.2

p. 395

1. Écrire une fonction `fib_iter` : `int -> int` calculant F_n en temps $O(n)$. On utilisera une boucle et des références. On annotera le code avec l'invariant de boucle permettant de justifier sa correction.
2. Écrire une fonction `fib_rec` : `int -> int` faisant la même chose (avec la même complexité) mais n'utilisant ni boucles ni référence. On pourra utiliser le schéma suivant :

```
let fib_rec n =
  let rec aux i f1 f2 =
    (* calcule F_n sachant que f1 = F_i, f2 = F_(i + 1) *)
    ...
    ... in
    ...
```

2 Fonctions de comparaison

Lorsqu'on souhaite trier une liste, ou un tableau, il est très courant qu'on veuille utiliser un ordre autre que l'ordre « par défaut ». Par exemple :

- trier une liste d'entiers en ordre décroissant;
- trier une liste de flottants par valeur absolue croissante ;
- trier une liste de couples par deuxième composante croissante... .

Pour spécifier cet ordre, on utilise habituellement une *fonction de comparaison*. Il s'agit d'une fonction `cmp : 'a -> 'a -> int` telle que :

- si x vient « avant » y dans l'ordre considéré, alors $\text{cmp } x \ y$ est strictement négatif;
- si x vient « après » y dans l'ordre considéré, alors $\text{cmp } x \ y$ est strictement positif;
- si x et y sont « indiscernables » pour l'ordre considéré, alors $\text{cmp } x \ y$ vaut 0.

Si x et y sont égaux, on aura bien sûr $\text{cmp } x \ y = 0$, mais ce n'est pas forcément le seul cas : par exemple, si l'on trie des flottants par valeur absolue croissante, alors -3.2 et 3.2 sont « indiscernables » bien qu'ils ne soient pas égaux.

Il existe en OCaml deux fonctions de tri prédéfinies :

- `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list`
- `Array.sort : ('a -> 'a -> int) -> 'a array -> unit`

Dans les deux cas, le premier argument est une fonction de comparaison spécifiant l'ordre, comme ci-dessus. La fonction `List.sort` renvoie une nouvelle liste, la fonction `Array.sort` n'agit que par effet de bord en modifiant le tableau passé en argument.

Exercice V.3

p. 396

En utilisant la fonction `List.sort` et une fonction de comparaison bien choisie, définir :

1. une fonction `tri_decroissant : int list -> int list` triant son argument en ordre décroissant;
2. une fonction `tri_valeur_absolue : float list -> float list` triant son argument par valeur absolue croissante (on utilisera la fonction `abs_float`);
3. une fonction `tri_deuxieme_composante : ('a * int) list -> ('a * int) list` triant son argument par deuxième composante croissante.

Remarque

Il est bien sûr très courant de souhaiter trier une liste ou un tableau en utilisant l'ordre « par défaut ». Pour cela, on peut utiliser la fonction de comparaison prédéfinie `compare : 'a -> 'a -> int`.

```
# List.sort compare [4; 2; 5; 1];;
- : int list = [1; 2; 4; 5]
```

```
# List.sort compare [3.2; 1.7; 2.4];;
- : float list = [1.7; 2.4; 3.2]
```

Exercice V.4

p. 396

Écrire une fonction `records : (cmp : 'a -> 'a -> int) -> (u : 'a list) -> 'a list` qui renvoie la liste des « records » de u , c'est-à-dire des éléments de u qui sont strictement plus grands que tous ceux qui les précèdent, au sens de la fonction de comparaison `cmp` fournie.

```
# records (fun x y -> x - y) [1; 8; 2; 4; 5; 10; 4; 10; 11; 8];;
- : int list = [1; 8; 10; 11]
# records (fun (a, b) (c, d) -> (a + b) - (c + d))
      [(1, 3); (2, 2); (0, 5); (3, 1); (4, 3)];;
- : (int * int) list = [(1, 3); (0, 5); (4, 3)]
```

On pourra exprimer records ($x :: y :: xs$) en fonction de records ($x :: xs$) et/ou de records ($y :: xs$).

3 Étude de l'algorithme d'Euclide

Rappels (ou pas) mathématiques

- Si $a, b \in \mathbb{N}$, on dit que a est un *diviseur* de b s'il existe $d \in \mathbb{N}$ tel que $b = ad$.
- Si $a, b \in \mathbb{N}^*$, l'ensemble des diviseurs communs à a et à b admet un plus grand élément, que l'on appelle *plus grand diviseur commun* de a et b , et que l'on note PGCD(a, b) ou $a \wedge b$.
- Tout entier divise 0, donc la définition précédente peut être étendue au cas où l'un des deux nombres a et b est nul, en posant $0 \wedge n = n \wedge 0 = n$ si $n \neq 0$.
- Par convention (comme toujours, cette convention n'est pas choisie au hasard), on définit $0 \wedge 0 = 0$.
- Si $a \in \mathbb{Z}$ et $b \in \mathbb{N}^*$, alors il existe un unique couple $(q, r) \in \mathbb{Z} \times [0 \dots b - 1]$ tel que $a = bq + r$ (division euclidienne de a par b). En OCaml, si a et b sont tous les deux positifs, alors $a \bmod b$ renvoie le r ainsi défini.

On s'intéresse à l'algorithme d'Euclide pour calculer le PGCD de deux entiers naturels :

```
let rec pgcd a b =
  if b = 0 then a else pgcd b (a mod b)
```

On pourra supposer dans la suite que $a \geq 0$ et $b \geq 0$, et l'on admettra la correction (partielle) de cet algorithme (vous la démontrerez en cours de mathématiques).

On note $f(a, b)$ le nombre de divisions (opérations `mod`) effectuées par cette fonction lorsqu'elle est appelée sur a et b .

Exercice V.5 – Étude expérimentale de l'algorithme d'Euclide

p. 396

1. Démontrer la terminaison de cet algorithme. Quelle borne obtient-on immédiatement sur $f(a, b)$ (en fonction de b) ?
2. Écrire une fonction `etapes` : `int -> int -> int` qui calcule le f défini plus haut.
Cette fonction s'écrit très simplement en modifiant légèrement la fonction `pgcd`; si l'envie vous prend d'utiliser une référence ou une boucle, ne cédez pas à la tentation : vous êtes en train de faire fausse route !
3. On définit la fonction φ pour $n \in \mathbb{N}^*$ par $\varphi(n) = \max_{0 \leq k < n} f(n, k)$.
On pourra ici utiliser avec profit la fonction `List.init` : `int -> (int -> 'a) -> 'a List`. L'appel `List.init n f` renvoie la liste `[f 0; ...; f (n - 1)]`.
 - a. Écrire la fonction `phi` : `int -> int` spécifiée ci-dessus.
 - b. Écrire une fonction `records_euclide` (`borne : int -> (int * int) list`) qui renvoie la liste des couples $(n, \varphi(n))$ avec $1 \leq n < \text{borne}$ et $\varphi(n) \ll \text{record}$ » (c'est-à-dire plus grand que tous les $\varphi(k)$ avec $k < n$).
On pourra commencer par générer la liste des couples $(n, \varphi(n))$ puis appliquer la fonction `records` avec une fonction de comparaison bien choisie.
 - c. Déterminer les records de φ sur l'intervalle $[1 \dots 1\,000]$, ainsi que les valeurs de n pour lesquelles ces records sont battus.
 - d. Conjecturer la forme des « pires cas » pour l'algorithme d'Euclide.

Exercice V.6 – Étude théorique de l'algorithme d'Euclide

p. 397

On définit (F_k) par $F_0 = 1$, $F_1 = 2$ et $F_{k+2} = F_k + F_{k+1}$, et φ et f comme à l'exercice précédent.

1. Déterminer $f(F_{k+1}, F_k)$ pour $k \in \mathbb{N}$.
2. Démontrer par récurrence sur k la propriété suivante : « si $1 \leq a < F_k$, alors $\varphi(a) < k$ ».

3. Montrer que $F_n \geq (\frac{3}{2})^n$.
4. En déduire que $\varphi(a) = O(\ln a)$.

4 Inégalité arithmético-géométrique

Exercice V.7

p. 398

On considère la fonction

```
let rec f n =
  if n = 1 then 0
  else if n mod 2 = 0 then 1 + f (n / 2)
  else 1 + f (n + 1)
```

1. Prouver que f termine sur toute entrée (positive).
2. Déterminer les records de f (pour n entre 1 et 1 000 par exemple) et conjecturer leur forme générale.
3. Démontrer proprement votre conjecture.

Exercice V.8

p. 399

On note $H_n : \forall x_1, \dots, x_n > 0, \prod_{k=1}^n x_k \leq \left(\frac{1}{n} \sum_{k=1}^n x_k \right)^n$ (inégalité arithmético-géométrique au rang n).

Si l'on souhaite démontrer que H_n est vraie pour tout $n \in \mathbb{N}^*$, sans passer par la concavité du logarithme, on peut procéder ainsi :

- on démontre H_2 ;
- on montre que pour $n \geq 2$, on a $H_n \Rightarrow H_{n-1}$;
- on démontre (à l'aide de H_2) que pour $n \in \mathbb{N}^*$, on a $H_n \Rightarrow H_{2n}$;
- on conclut par récurrence.

1. Justifier la dernière étape de ce raisonnement (« on conclut par récurrence »).
2. Quel est le rapport avec l'exercice précédent ?
3. Pourquoi la correction de ce raisonnement par récurrence n'implique-t-elle pas la terminaison de la fonction suivante (pour $n \geq 2$) :

```
let rec g n =
  n = 2 || g (n + 1) || (n mod 2 = 0 && g (n / 2))
```

4. Quelle modification *très simple* peut-on faire sur cette fonction pour qu'elle termine ?
5. Rédiger la démonstration complète de l'inégalité arithmético-géométrique (les trois points avant le « on conclut par récurrence »).

Solutions

Correction de l'exercice V.1 page 391

1. La lecture du code fournit immédiatement $\psi(0) = \psi(1) = 0$ et $\psi(n+2) = 1 + \psi(n) + \psi(n+1)$ si $n \geq 0$. On procède donc par récurrence double, avec l'hypothèse de récurrence H_n : « $\psi(n) = F_{n+1} - 1$ ».

Initialisation : $\psi(0) = 0 = F_1 - 1$ et $\psi(1) = 0 = F_2 - 1$, H_0 et H_1 sont vérifiées.

Héritéité : soit $n \geq 0$, on suppose H_n et H_{n+1} . On a alors :

$$\begin{aligned}\psi(n+2) &= 1 + \psi(n) + \psi(n+1) \\ &= 1 + F_{n+1} - 1 + F_{n+2} - 1 && H_n \text{ et } H_{n+1} \\ &= F_{n+3} - 1 && \text{définition de } F\end{aligned}$$

On en déduit que $\psi(n) = F_{n+1} - 1$ pour tout $n \geq 0$.

2. La complexité de `fib_naif` est clairement proportionnelle au nombre d'additions effectuées, c'est donc un $\Theta(F_{n+1})$. Or $F_{n+1} = \Theta(\varphi^{n+1})$, donc on obtient une complexité exponentielle en $\Theta(\varphi^n)$.
3. φ^{10} est de l'ordre de 100, donc il n'y aura aucun problème. En revanche, φ^{50} est de l'ordre de 10^{10} donc le calcul sera sans doute très long, et $\varphi^{1000} > 10^{200}$ donc ce calcul n'est pas réalisable.

Remarque

Pour `fib_naif 1000`, on aura également un problème de dépassement de capacité puisque le résultat sera beaucoup trop grand pour rentrer dans un `int`, mais ce n'est pas le sujet ici.

Correction de l'exercice V.2 page 391

1. On maintient deux variables contenant les valeurs des deux derniers termes calculés. Attention à traiter correctement les cas $n = 0$ et $n = 1$!

```
let fib_iter n =
  let f1 = ref 0 in
  let f2 = ref 1 in
  (* invariant f1 = F_i, f2 = F_{i + 1} *)
  for i = 0 to n - 1 do
    let tmp = !f1 in
    f1 := !f2;
    f2 := !f2 + tmp
  done;
  (* i = n (conceptuellement) en sortie de boucle,
   * donc f1 = F_n *)
  !f1
```

2. Il s'agit essentiellement de la même fonction, traduite en style fonctionnel :

```
let fib_rec n =
  let rec aux i f1 f2 =
    if i = n then f1 else aux (i + 1) f2 (f1 + f2) in
  aux 0 0 1
```

Correction de l’exercice V.3 page 392

1.

```
let tri_decroissant u = List.sort (fun x y -> y - x) u

(* ou plus simplement *)
let tri_decroissant = List.sort (fun x y -> y - x)
```

2.

```
let compare_abs x y =
  let diff = abs_float x -. abs_float y in
  if diff < 0. then -1
  else if diff > 0. then 1
  else 0

let tri_valeur_absolue = List.sort compare_abs
```

3.

```
let compare_deuxieme (_, y) (_, y') = y - y'

let tri_deuxieme_composante = List.sort compare_deuxieme
```

Correction de l’exercice V.4 page 392

On donne uniquement la version la plus simple :

```
let rec records cmp u =
  match u with
  | [] -> []
  | [x] -> [x]
  | x :: y :: xs ->
    if cmp x y < 0 then x :: records cmp (y :: xs)
    else records cmp (x :: xs)
```

Les deux premiers cas ne posent pas de problème. Dans le troisième cas, x est systématiquement un record, et y en est un si et seulement si il est « strictement plus grand » que x .

Correction de l’exercice V.5 page 393

1. On sait que le reste de la division euclidienne de a par b est un entier compris entre 0 et $b - 1$. Donc le second argument de la fonction pgcd décroît strictement à chaque appel récursif, tout en restant positif. Comme $b = 0$ est un cas de base, la fonction termine.

On en déduit aussi que le nombre d’appel ne peut pas excéder b : $f(a, b) \leq b$.

2. On conserve la structure de la fonction pgcd :

```
let rec etapes a b =
  if b = 0 then 0 else 1 + etapes b (a mod b)
```

3.

```

let phi n =
  let rec max_liste = function
    | [] -> failwith "max d'une liste vide"
    | [x] -> x
    | x :: xs -> max x (max_liste xs) in
  (* liste_etapes = [etapes n 0; etapes n 1; ...; etapes n (n - 1)] *)
  let liste_etapes = List.init n (etapes n) in
  max_liste liste_etapes

```

4.

```

let records_euclide borne =
  (* liste_couples = [(1, phi 1); (2, phi 2); ...; (borne, phi borne)] *)
  let liste_couples = List.init borne (fun n -> (n + 1, phi (n + 1))) in
  let cmp (n, phi_n) (n', phi_n') = phi_n - phi_n' in
  records cmp liste_couples

```

5. Les records de $\varphi(n)$ correspondent à $n = 1, 2, 3, 5, 8, 13, 21 \dots$: autrement dit, les pires cas semblent être pour les nombres de Fibonacci. Plus précisément, en calculant par exemple les valeurs de $f(21, k)$, on observe que la plus grande valeur est $f(21, 13) = 6$. Cela suggère que le pire cas est celui du calcul de $F_n \wedge F_{n-1}$.

Correction de l'exercice V.6 page 393

- Pour $F_1 \wedge F_0 = 2 \wedge 1$, on se ramène en une division à $1 \wedge 0$ qui renvoie. Donc $f(F_1, F_0) = 1$. Ensuite, pour $k \geq 1$, on observe que

$$F_{k+1} = 1 \times F_k + F_{k-1}.$$

Or F est strictement croissante (immédiat par récurrence) et positive, donc $0 \leq F_{k-1} < F_k$. L'égalité ci-dessus correspond donc à la division euclidienne de F_{k+1} par F_k : le quotient vaut 1 et le reste F_{k-1} .

Ainsi, le calcul $F_{k+1} \wedge F_k$ se réduit en une étape à celui de $F_k \wedge F_{k-1}$: on a donc $f(F_{k+1}, F_k) = 1 + f(F_k, F_{k-1})$. Avec la valeur initiale $f(F_1, F_0) = 1$, on obtient donc $f(F_{k+1}, F_k) = k + 1$.

2. Initialisation

- Pour $k = 1$: comme $F_1 = 2$, le seul à qu'il faut considérer est $a = 1$. Or $\varphi(1) = f(1, 0) = 0 < 1$, c'est bon.
- Pour $k = 2$: on a $F_2 = 3$, il faut traiter $a = 1$ (déjà fait) et $a = 2$. Or $\varphi(2) = \max(f(2, 0), f(2, 1)) = \max(0, 1) = 1 < 2$, c'est bon.

Héritéité Soit $k > 1$, on suppose H_k et H_{k-1} . Soit $1 \leq a < F_{k+1}$.

- Si $a < F_k$, alors $\varphi(a) < k$ d'après H_k , donc $\varphi(a) < k + 1$.
- On suppose donc $F_k \leq a < F_{k+1}$. Soit alors $1 \leq b < a$, il faut majorer strictement $f(a, b)$ par $k + 1$.
 - Si $b < F_k$, alors $f(a, b) = 1 + f(b, a \bmod b) \leq 1 + \varphi(b)$ puisque $a \bmod b < b$. D'après H_k , on en déduit $f(a, b) < 1 + k$.
 - Sinon, on a $F_k \leq b < a$. On a alors $2b \geq 2F_k \geq F_{k+1} > a$, donc le quotient de la division euclidienne de a par b vaut 1, et le reste $a - b$. On a alors :

$$1 \leq a - b < F_{k+1} - F_k = F_{k-1}$$

Ainsi :

$$\begin{aligned}
 f(a, b) &= 1 + f(b, a \bmod b) \\
 &= 2 + f(a \bmod b, b \bmod (a \bmod b)) \\
 &\leq 2 + \varphi(a \bmod b)
 \end{aligned}$$

On peut appliquer H_{k-1} à $a \bmod b < F_{k-1}$ et obtenir $f(a, b) < 2 + k - 1 = k + 1$.

Dans tous les cas, $f(a, b) < k + 1$, et donc $\varphi(a) < k + 1$.

Finalement, pour tout $k \geq 1$, $1 \leq a < F_k \Rightarrow \varphi(a) < k$.

3. Pour $n \in \mathbb{N}$, on définit H_n : « $F_n \geq \left(\frac{3}{2}\right)^n$ ».

Initialisation $F_0 = 1 = \left(\frac{3}{2}\right)^0$ et $F_1 = 2 > \left(\frac{3}{2}\right)^1$.

Hérité Soit $n \in \mathbb{N}$, on suppose H_n et H_{n+1} . On a alors :

$$\begin{aligned} F_{n+2} &= F_n + F_{n+1} \\ &\geq \left(\frac{3}{2}\right)^n + \left(\frac{3}{2}\right)^{n+1} && \text{d'après } H_n \text{ et } H_{n+1} \\ &= \left(\frac{3}{2}\right)^n \cdot \frac{5}{2} \end{aligned}$$

Or $\frac{5}{2} = \frac{10}{4} > \frac{9}{4} = \left(\frac{3}{2}\right)^2$, donc $F_{n+2} \geq \left(\frac{3}{2}\right)^{n+2}$.

On a donc bien $F_n \geq \left(\frac{3}{2}\right)^n$ pour tout $n \in \mathbb{N}$.

4. Soit $a \in \mathbb{N}^*$ et $k = \lfloor \log_{3/2} a \rfloor$. On a $a < \left(\frac{3}{2}\right)^{k+1}$, donc $a < F_{k+1}$ et $\varphi(a) < k + 1$ d'après les deux questions précédentes. Or $k = O(\ln a)$, donc $\varphi(a) = O(\ln a)$.

Correction de l'exercice V.7 page 394

1. Soit n un entier naturel non nul.

- Si $n = 1$, la fonction termine.
- Si $n = 2p$ avec ($p \geq 1$), alors $f(n) = 1 + f(p)$ et $1 \leq p < n$.
- Si $n = 2p + 1 \geq 3$, alors

$$f(n) = 1 + f(n + 1) = 2 + f(p + 1) \text{ avec } 1 \leq p + 1 < n.$$

Quitte à étudier deux appels successifs, on voit bien que l'argument décroît strictement en restant supérieur ou égal à 1. On finit donc par atteindre 1 et terminer (si l'appel initial se fait sur $n \in \mathbb{N}^*$, bien sûr).

2. On réutilise la fonction records :

```
let records_f n =
  let liste_couples = List.init n (fun i -> (i + 1, f (i + 1))) in
  let cmp (_, fi) (_, fi') = fi - fi' in
  records cmp liste_couples
```

On obtient alors :

```
# records_f 1000;;
- : (int * int) list =
[(1, 0); (2, 1); (3, 3); (5, 5); (9, 7); (17, 9); (33, 11); (65, 13);
(129, 15); (257, 17); (513, 19)]
```

Les records semblent correspondre aux nombres de la forme $2^k + 1$ (et au nombre 1 que l'on va ignorer).

3. Pour $n \in \mathbb{N}$, notons $u_n = 2^n + 1$. On veut démontrer que $f(u_n) = \max_{1 \leq k \leq u_n} f(k)$.

- De manière évidente,

$$f(u_{n+1}) = f(2^{n+1} + 1) = 1 + f(2^{n+1} + 2) = 2 + f(2^n + 1) = 2 + f(u_n)$$

On en déduit $f(u_n) = 2n + f(u_0) = 2n + 1$.

- On définit pour $n \in \mathbb{N}$ la propriété H_n : « si $1 \leq k \leq u_n$ alors $f(k) \leq 2^n + 1$ ».

Initialisation Si $1 \leq k \leq u_0 = 2$, alors soit $k = 1$ et $f(k) = 0$ soit $k = 2$ et $f(k) = 1$. On a bien $f(k) \leq 2 \times 0 + 1$.

Héritéité Soit $n \in \mathbb{N}$, on suppose H_n . Si $1 \leq k \leq u_{n+1} = 2^{n+1} + 1$, on distingue suivant la parité de k :

- si k est pair, alors $f(k) = 1 + f(k/2)$. Or $k/2 \leq 2^n + 1/2 < u_n$, donc d'après H_n on a $f(k) \leq 1 + 2^n + 1 < 2(n+1) + 1$.

On a bien $k/2 \geq 1$ puisque k est pair et supérieur ou égal à 1.

- si k est impair, alors $f(k) = 1 + f(k+1) = 2 + f((k+1)/2)$. Or $1 \leq (k+1)/2 \leq (2^{n+1} + 2)/2 = 2^n + 1$, donc d'après H_n $f(k) \leq 2 + 2^n + 1 = 2(n+1) + 1$.

Les valeurs record correspondent donc bien exactement aux entiers de la forme $2^n + 1$.

Correction de l'exercice V.8 page 394

1. On peut prouver H_n par récurrence forte sur $n \geq 2$. En supposant H_k pour tout $2 \leq k \leq n$, on doit montrer H_{n+1} :
 - si $n+1$ est pair (et donc supérieur ou égal à 4), alors $2 \leq (n+1)/2 \leq n$. On a donc $H_{\frac{n+1}{2}}$ par hypothèse de récurrence, puis H_{n+1} car $H_i \Rightarrow H_{2i}$;
 - sinon, $n+2$ est pair et $2 \leq (n+2)/2 \leq n$. On a donc $H_{\frac{n+2}{2}}$ par hypothèse de récurrence, puis H_{n+2} comme $H_i \Rightarrow H_{2i}$ et enfin H_{n+1} puisque $H_i \Rightarrow H_{i-1}$.
2. Le raisonnement pour la terminaison de f est exactement le même que celui permettant de démontrer par récurrence H_n à partir de $H_i \Rightarrow H_{2i}$ et $H_i \Rightarrow H_{i-1}$.
3. L'ordre d'évaluation de l'opérateur `||` va imposer la vérification de `g (n+1)` avant la vérification de la parité (c'est cette étape qui permet de réduire l'argument de l'appel récursif et d'aller vers la terminaison). Cette fonction ne va donc jamais terminer si $n \geq 2$.
4. On inverse les opérandes du `||` :

```
let rec g n =
  n = 2 || (n mod 2 = 0 && g (n/2)) || g (n+1)
```

5. Bon exercice de mathématiques pour un début de sup, demandez-moi de l'aide si vous bloquez...

EXERCICES D'ENTRAÎNEMENT

I Coefficients binomiaux

Exercice VI.1 – Calcul récursif des coefficients binomiaux

p. 402

- Écrire une fonction `binom` (`k : int`) (`n : int`) : `int` qui calcule $\binom{n}{k}$ en utilisant la formule du triangle de Pascal. On rappelle que $\binom{n}{k}$ vaut 0 sauf si $0 \leq k \leq n$.
- Utilisez votre calculatrice pour calculer $\binom{30}{15}$, puis faites le calcul à l'aide de la fonction `binom`. Que constate-t-on ?
- On définit $C(k, n)$ comme le nombre total d'appels effectués lors du calcul de `binom k n`, en comptant l'appel initial et les appels récursifs. Démontrer par récurrence sur n la propriété suivante : « si $0 \leq k \leq n$, alors $C(k, n) \geq \binom{n}{k}$ ». En déduire une explication du phénomène constaté à la question précédente.

Exercice VI.2 – Calcul efficace des coefficients binomiaux

p. 402

- Écrire une fonction `triangle` : `int -> int array array` prenant en entrée un entier n supposé positif ou nul et renvoyant un tableau t tel que :
 - t est de longueur $n + 1$;
 - pour $0 \leq i \leq n$, $t.(i)$ est de longueur $i + 1$;
 - pour $0 \leq j \leq i \leq n$, $t.(i).(j)$ vaut $\binom{i}{j}$.
 Quelques indications :
 - on initialisera t à `Array.make (n + 1) [| |]`, puis l'on remplacera chaque « ligne » de t par un tableau de la bonne taille, que l'on remplira ;
 - pour remplir ces lignes, on utilisera la formule du triangle.
- En utilisant `triangle`, écrire une fonction `binom_it` : `int -> int -> int` calculant $\binom{n}{k}$.
- Déterminer les complexité en temps et en espace de `binom_it`. Vérifier que le calcul de `binom_it 15 30` est immédiat, ainsi que celui de `binom 150 300`. Commentez le résultat de ce dernier calcul.
- En utilisant directement la formule $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, écrire une fonction `binom_formule` : `int -> int -> int` effectuant le calcul d'un coefficient binomial en temps linéaire en n .
- Vérifier que votre fonction donne les bons résultats pour $\binom{13}{6}$ et $\binom{24}{12}$. Que se passe-t-il pour $\binom{25}{12}$? Comment expliquer que `binom_it` donne le bon résultat dans ce cas, mais pas la nouvelle fonction?
- Optionnel.** Écrire une fonction calculant $\binom{n}{k}$ en temps linéaire et n'ayant pas le problème de `binom_formule`.

2 Recherche dans une matrice triée

On considère le problème suivant :

Entrées : un entier x et une matrice M à n lignes et p colonnes.

Précondition : chaque ligne de M est croissante (lue de gauche à droite) et chaque colonne de M est croissante (lue de haut en bas).

Résultat : un booléen indiquant si l’entier x apparaît dans la matrice M .

En OCaml, la matrice M sera représentée par un objet m : **int array array** tel que m soit de longueur n et que tous les $m.(i)$ soient de même longueur p .

Exercice VI.3

p. 403

1. Écrire une fonction `cherche_naif : int -> int array array -> bool` résolvant le problème de la manière la plus simple possible. Déterminer la complexité de cette fonction.
2. Écrire une fonction résolvant le problème en temps $O(p \log n)$.
3. Quelle amélioration très simple peut-on apporter à cette fonction si $p \gg n$?
4. Écrire une fonction résolvant le problème en temps $O(n + p)$. On justifiera la correction de l’algorithme (*a minima*, on fournira un invariant de boucle permettant de la justifier).

Exercice VI.4

p. 403

On suppose dans cet exercice que la matrice a n lignes et n colonnes.

1. Montrer qu’il est impossible de résoudre le problème en lisant moins de n éléments de M .
2. Que peut-on en déduire pour la dernière fonction écrite ?

3 Combinatoire

Exercice VI.5 – Couvertures

p. 403

On souhaite pavier une ligne constituée de n cases carrées avec des tuiles de différents types. Chaque type de tuile occupe un certain nombre de cases et est d’une certaine couleur. On considère que deux tuiles de type différent sont toujours de couleur différente, et que l’on dispose d’autant de tuiles de chaque type que l’on souhaite.

Écrire une fonction `couvertures : int list -> int -> int` telle que `couverture u n` renvoie le nombre de manières de couvrir une ligne de n cases avec des tuiles de longueurs données par u . Par exemple, si $u = [1; 3; 1]$ il y a trois types de tuiles, dont deux sont de longueur 1 et un de longueur 3. On demande une solution concise, sans trop se soucier de l’efficacité.

Solutions

Correction de l'exercice VI.1 page 400

1.

```
let rec binom k n =
  if k < 0 || k > n then 0
  else if k = 0 || k = n then 1
  else binom k (n - 1) + binom (k - 1) (n - 1)
```

2. L'appel `binom 15 30` renvoie bien 155 117 520, mais le calcul prend plusieurs secondes alors qu'il « devrait » être instantané.
3. La lecture du code fournit $C(k, n) = 1 + C(k, n - 1) + C(k - 1, n - 1)$ si $0 < k < n$ et $C(0, n) = C(n, n) = 1$. Démontrons par récurrence la propriété H_n de l'énoncé.

Initialisation : le seul cas à considérer est $C(0, 0) = 1$, or on a $\binom{0}{0} = 1$, donc H_0 est vraie.

Héritéité Soit $n \in \mathbb{N}$, supposons H_n .

- $C(0, n + 1) = 1 = \binom{n+1}{0}$.
- $C(n + 1, n + 1) = 1 = \binom{n+1}{n+1}$.
- Si $0 < k < n + 1$, alors :

$$\begin{aligned} C(k, n + 1) &= 1 + C(k - 1, n) + C(k, n) \\ &\geq 1 + \binom{n}{k-1} + \binom{k}{n} && \text{d'après } H_n \\ &\geq 1 + \binom{n+1}{k} && \text{triangle } \geq \binom{n+1}{k} \end{aligned}$$

On a donc bien $C(k, n) \geq \binom{n}{k}$ pour tous $0 \leq k \leq n$. Or la complexité de `binom` est clairement minorée par $C(k, n)$ (elle est en fait de l'ordre de $C(k, n)$), puisque chaque appel se fait en temps constant. Comme $\binom{n}{k}$ croît rapidement, il n'est pas surprenant que le calcul `binom 15 30` soit long.

Correction de l'exercice VI.2 page 400

1. Il faut faire bien attention aux valeurs extrêmes de chaque ligne, et à ne pas sortir des bornes du tableau. En initialisant chaque ligne avec des 1, les valeurs pour $j = 0$ et $j = i$ sont correctes, on utilise la formule uniquement pour les $1 \leq j < i$.

```
let triangle n =
  let t = Array.make (n + 1) [| |] in
  for i = 0 to n do
    t.(i) <- Array.make (i + 1) 1;
    for j = 1 to i - 1 do
      t.(i).(j) <- t.(i - 1).(j) + t.(i - 1).(j - 1)
    done
  done;
  t
```

2.

```
let binom_it k n =
  if k < 0 || k > n then 0
  else (triangle n).(n).(k)
```

3. La fonction triangle :

- initialise t en temps $O(n)$;
- initialise chaque $t.(i)$ en temps $O(i)$;
- remplit les cases de chaque $t.(i)$ en temps $O(i)$.

Au total, on a du $O(n + \sum_{i=0}^n i) = O(n^2)$. On en déduit immédiatement que la fonction `binom_it` a une complexité temporelle en $O(n^2)$. Pour la complexité spatiale, on crée un tableau ayant $\Theta(n^2)$ cases, donc c'est également quadratique.

Le calcul de `binom_it 15 30` est immédiat, comme celui de `binom_it 150 300`. En revanche, si le premier donne bien le résultat escompté, le deuxième renvoie un résultat négatif : ce n'est pas surprenant, la valeur de $\binom{300}{150}$ dépassant largement la capacité d'un entier machine.

4.

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)

let binom_formule k n =
  if k < 0 || k > n then 0
  else (fact n) / (fact k * fact (n - k))
```

5. Le problème est qu'on commence par calculer $n!$, qui est beaucoup plus grand que le résultat final (d'un facteur $k!(n-k)!$). Il est donc tout à fait possible que le résultat final tienne dans un `int` mais que le calcul de $n!$ provoque un dépassement de capacité : c'est ce qui se passe pour `binome_formule 12 25`, par exemple.

6.

```
let rec binom_eff k n =
  if k < 0 || k > n then 0
  else if k = 0 then 1
  else n * binom_eff (k - 1) (n - 1) / k
```

Correction de l'exercice VI.3 page 401

Correction de l'exercice VI.4 page 401

Correction de l'exercice VI.5 page 401

ENSEMBLES ET COMBINATOIRE

I Ensembles représentés par des listes

1.1 Préliminaires

Exercice VII.1 – Quelques fonctions usuelles sur les listes

p. 408

On appelle *prédictat* une fonction de type `'a -> bool`. On dit alors que x vérifie le prédictat `pred` si `pred x = true`.

1. Écrire une fonction `pour_tout : ('a -> bool) -> 'a list -> bool` telle que `pour_tout pred u` soit vrai si et seulement si tous les éléments de u vérifient `pred`.
On évitera les calculs inutiles.
2. Écrire une fonction `existe : ('a -> bool) -> 'a list -> bool` telle que `existe pred u` soit vrai si et seulement si il existe $x \in u$ vérifiant `pred`.
On évitera les calculs inutiles.
3. Écrire une fonction `filtre : ('a -> bool) -> 'a list -> 'a list` telle que `filtre pred u` renvoie la liste des éléments de u vérifiant `pred` (ces éléments seront dans le même ordre que dans la liste u).

Ces trois fonctions sont prédéfinies : il s'agit de `List.for_all`, `List.exists` et `List.filter`.

1.2 Représentation par des listes non triées

Dans cette section, on utilise des listes non triées, et comportant éventuellement des répétitions, pour représenter des parties finies de \mathbb{Z} . Une telle représentation n'est pas unique : la partie $\{3; 7; 10\}$ peut être représentée par $[3; 7; 10]$, par $[7; 3; 10]$ ou même par $[3; 10; 10; 3; 7]$.

Les types donnés dans l'énoncé correspondent au cas des listes d'entiers, mais vos fonctions devraient logiquement avoir un type plus général (`'a list` plutôt que `int list`) : ce n'est bien sûr pas un problème.

Exercice VII.2 – Opérations ensemblistes, versions élémentaires

p. 408

Dans cet exercice, l'idée est de proposer des fonctions aussi concises que possible en utilisant `pour_tout`, `existe` et `filtre`, sans chercher à optimiser la complexité.

1. Écrire une fonction `appartient : int list -> int -> bool` qui détermine si un entier est élément d'une liste.
2. Écrire une fonction `inclus : int list -> int list -> bool` telle que `inclus u v` renvoie `true` si et seulement si la partie représentée par u est incluse dans celle représentée par v .
Quelle est la complexité de cette fonction (dans le pire des cas, en fonction de $|u|$ et $|v|$) ?
3. Écrire une fonction `egal : int list -> int list -> bool` qui détermine si deux listes représentent la même partie de \mathbb{Z} .
Quelle est la complexité de cette fonction ?
4. Écrire une fonction `inter : int list -> int list -> int list` renvoyant l'intersection de deux listes, et donner sa complexité.
5. Écrire une fonction `prive_de : int list -> int list -> int list` renvoyant la différence ensembliste entre son premier et son deuxième argument.
6. Écrire une fonction `union : int list -> int list -> int list`, et donner sa complexité.

1.3 Représentation à l'aide de listes triées

Dans cette section, on représente les parties finies de \mathbb{Z} par des listes **strictement croissantes**, et on essaie de tirer partie de ce fait pour écrire des versions plus efficaces de certaines opérations ensemblistes.

À chaque fois que l'on vous demande d'écrire une fonction renvoyant une liste représentant un ensemble, il est impératif que la liste renvoyée soit bien strictement croissante (et l'on peut supposer sans le vérifier que les listes passées en argument le sont).

Exercice VII.3 – Union, intersection et différence « efficaces »

p. 409

1. En s'inspirant de la fonction `fusionne` que nous avions écrite pour le tri fusion, écrire une version efficace de la fonction `union`.
2. Écrire des versions efficaces des fonctions `inter` et `prive_de`.
3. Déterminer la complexité des fonctions ci-dessus.

Exercice VII.4 – Tests d'inclusion et d'égalité

p. 409

1. Écrire une version efficace de la fonction `inclus`.
2. Écrire une version efficace (et concise) de la fonction `egal`.
3. Quelle est la complexité dans le pire cas des fonctions écrites ci-dessus ?

Exercice VII.5 – Changement de représentation

p. 410

Écrire une fonction `tri_uniques` : `int list -> int list` qui prend en entrée une liste d'entiers non triée et pouvant contenir des répétitions, et renvoie une liste strictement croissante contenant les mêmes éléments (mais bien sûr sans répétitions). On s'inspirera du tri fusion.

2 Combinatoire

Dans cette partie, on s'intéresse à la génération des différents objets combinatoires élémentaires associés à un ensemble : parties, combinaisons, permutations, p-listes et combinaisons avec répétitions.

Je vous conseille de définir la fonction suivante, qui s'avère très utile :

```
let map_prefixe x liste = List.map (fun u -> x :: u) liste
(*
# map_prefixe 3 [[1; 2]; []; [5]];
- : int list list = [[3; 1; 2]; [3]; [3; 5]]
```

Notre définition de `map_prefixe` est de la forme "let `f x = g x`", on aurait pu "simplifier" en écrivant :

```
let map_prefixe x = List.map (fun u -> x :: u)
*)
```

Dans toute la partie, on pourra supposer sans le vérifier que les listes que l'on reçoit en argument sont sans répétition.

Exercice VII.6

p. 410

Définir une fonction `parties` : '`a list -> 'a list list` qui renvoie la liste des parties de `u` (l'ordre dans lequel les parties sont renvoyées n'a pas d'importance).

```
# parties [1; 2; 5];
- : int list list = [[]; [5]; [2]; [2; 5]; [1]; [1; 5]; [1; 2]; [1; 2; 5]]
```

Exercice VII.7

p. 410

1. Définir une fonction `combinaisons` : '`a list -> int -> 'a list list` telle que `combinaisons u n` renvoie la liste des combinaisons de `n` éléments de `u` (l'ordre dans lequel les combinaisons sont renvoyées n'a pas d'importance). Une combinaison de `n` éléments de `u` est juste un autre nom pour une partie à `n` éléments de `u`.

```
# combinaisons [1; 2; 3; 6] 2;;
- : int list list = [[3; 6]; [2; 6]; [2; 3]; [1; 6]; [1; 3]; [1; 2]]
```

2. Donner une autre version de la fonction `parties` utilisant la fonction `combinaisons`.
 3. Définir une fonction `avec_repetitions` : '`a list -> int -> 'a list list` telle que `avec_repetitions u n` renvoie la liste des combinaisons avec répétitions de `n` éléments de `u`:

```
# avec_repetitions [1; 2; 3; 7] 3;;
- : int list list =
[[1; 1; 1]; [1; 1; 2]; [1; 1; 3]; [1; 1; 7]; [1; 2; 2]; [1; 2; 3];
[1; 2; 7]; [1; 3; 3]; [1; 3; 7]; [1; 7; 7]; [2; 2; 2]; [2; 2; 3];
[2; 2; 7]; [2; 3; 3]; [2; 3; 7]; [2; 7; 7]; [3; 3; 3]; [3; 3; 7];
[3; 7; 7]; [7; 7; 7]]
```

Exercice VII.8

p. 411

1. Écrire une fonction `insertions` : '`a -> 'a list -<- 'a list list` telle que `insertions x u` renvoie la liste des listes obtenues en insérant `x` à une certaine position dans `u`:

```
# insertions 7 [2; 5; 3];
- : int list list = [[7;2;5;3]; [2;7;5;3]; [2;5;7;3]; [2;5;3;7]]
```

2. Écrire une fonction `applatit` : '`a list list -> 'a list` qui renvoie la concaténation d'une liste de listes (c'est la fonction `List.flatten` de la bibliothèque standard) :

```
# applatit [[1; 2; 3]; [7; 12]; [4; 1]];
- : int list = [1; 2; 3; 7; 12; 4; 1]
```

3. Écrire une fonction `permutations` : '`a list -> 'a list list` renvoyant la liste des permutations de son argument.

```
# permutations [2; 3; 5];
- : int list list = [[2;3;5]; [3;2;5]; [3;5;2]; [2;5;3]; [5;2;3];
↪ [5;3;2]]
```

Exercice VII.9

p. 412

Écrire une fonction `plistes` : '`a list -> int -> 'a list list`.

```
# plistes [1; 2; 3] 3;;
- : int list list =
[[1; 1; 1]; [1; 1; 2]; [1; 1; 3]; [1; 2; 1]; [1; 2; 2]; [1; 2; 3]; [1; 3;
→ 1];
[1; 3; 2]; [1; 3; 3]; [2; 1; 1]; [2; 1; 2]; [2; 1; 3]; [2; 2; 1]; [2; 2;
→ 2];
[2; 2; 3]; [2; 3; 1]; [2; 3; 2]; [2; 3; 3]; [3; 1; 1]; [3; 1; 2]; [3; 1;
→ 3];
[3; 2; 1]; [3; 2; 2]; [3; 2; 3]; [3; 3; 1]; [3; 3; 2]; [3; 3; 3]]
```

3 Pour chercher

Exercice VII.10 – Plus petit entier manquant

p. 412

Si E est un ensemble fini d'entiers, on définit $\text{ppm}(E) = \min(n \in \mathbb{N} \mid n \notin E)$.

1. On suppose que E est représenté par un tableau d'entiers distincts.
Écrire une fonction `ppm_array : int array -> int array -> int` calculant le plus petit entier manquant de E en temps linéaire en la taille de E .
2. On suppose désormais que E est représenté par une liste d'entiers distincts, et l'on souhaite trouver une solution efficace **purement fonctionnelle** (n'utilisant pas de tableaux, de références...).
 - a. Écrire une fonction `partition : 'a list -> 'a -> ('a list * 'a list * int)` telle que `partition u b` renvoie un triplet (v, w, k) tel que :
 - v est la liste des éléments de u strictement inférieurs à b ;
 - w est la liste des éléments de u supérieurs ou égaux à b ;
 - $k = |v|$.
 L'ordre des éléments dans ces deux listes n'a pas d'importance.
 - b. Écrire une fonction `ppm_list : int list -> int` résolvant efficacement le problème. La complexité attendue est linéaire en la taille de la liste, mais ce ne sera pas forcément évident à la lecture du code.

Solutions

Correction de l'exercice VII.1 page 404

Ces fonctions s'expriment très simplement à l'aide d'un *fold*, mais le calcul se poursuit alors systématiquement jusqu'au bout même quand on aurait pu répondre sans examiner tous les éléments.

1.

```
let rec pour_tout pred u =
  match u with
  | [] -> true
  | x :: xs -> pred x && pour_tout pred xs
```

2.

```
let rec existe pred = function
| [] -> false
| x :: xs -> pred x || existe pred xs
```

3.

```
let rec filtre pred = function
| [] -> []
| x :: xs -> if pred x then x :: filtre pred xs else filtre pred xs
```

Correction de l'exercice VII.2 page 404

1. Complexité : $O(|u|)$.

```
let appartient u x = existe (( = ) x) u
```

2. Complexité : $O(|u| \cdot |v|)$.

```
let inclus u v = pour_tout (appartient v) u
```

3. Complexité : $O(|u| \cdot |v|)$.

```
let egal u v = (inclus u v) && (inclus v u)
```

4. Complexité : $O(|u| \cdot |v|)$.

```
let inter u v = filtre (appartient u) v
```

5. Complexité $O(|u|)$ pour *not_in* u x , et donc $O(|u| \cdot |v|)$ pour *prive_de*.

```
let not_in u x = not (appartient u x)
```

```
let prive_de u v = filtre (not_in v) u
```

6. Complexité $O(|u|)$.

```
let union u v = u @ v
```

Correction de l'exercice VII.3 page 405

1. C'est presque la même chose que fusionne : la seule différence est que, si un élément est présent dans les deux listes, on ne le rajoute qu'une fois.

```
let rec union_eff u v =
  match u, v with
  | [], _ -> v
  | _, [] -> u
  | x :: xs, y :: ys ->
    if x < y then x :: union_eff xs v
    else if x > y then y :: union_eff u ys
    else x :: union_eff xs ys
```

- 2.

```
let rec inter_eff u v =
  match u, v with
  | [], _ -> []
  | _, [] -> []
  | x :: xs, y :: ys when x < y -> inter_eff xs v
  | x :: xs, y :: ys when x > y -> inter_eff u ys
  | x :: xs, y :: ys -> x :: inter_eff xs ys
```

- 3.

```
let rec prive_de_eff u v =
  match u, v with
  | [], _ -> []
  | _, [] -> u
  | x :: xs, y :: ys when x = y -> prive_de_eff xs ys
  | x :: xs, y :: ys when x < y -> x :: prive_de_eff xs v
  | x :: xs, y :: ys -> prive_de_eff u ys
```

4. Pour chacune de ces fonctions, on traite soit un élément de u soit un élément de v à chaque étape (et dans certains cas un élément de chaque liste) : il y a donc au plus $|u| + |v|$ étapes. Comme chaque étape est en temps constant, on obtient une complexité en $O(|u| + |v|)$.

Correction de l'exercice VII.4 page 405

1. C'est toujours la même idée, en faisant attention aux cas de base.

```
let rec inclus_eff u v =
  match u, v with
  | [], _ -> true
  | x :: xs, y :: ys when x = y -> inclus_eff xs ys
  | x :: xs, y :: ys when x > y -> inclus_eff u ys
  | _ -> false
```

2. Il y a unicité de la représentation, donc deux parties sont égales si et seulement si elles sont

représentées par la même liste. On pourrait écrire `let egal_eff u v = (u = v)`, mais on peut faire plus concis :

```
let egal_eff = (=)
```

3. Pour `inclus_eff`, on s'arrête dès que l'une des listes est vide, et à chaque étape on élimine un élément de `v` et éventuellement un élément de `u`. La complexité est donc en $O(|v|)$. Pour `egal_eff`, c'est du $O(\min(|u|, |v|))$.

Correction de l'exercice VII.5 page 405

La seule différence avec le tri fusion est que l'on remplace la fonction `fusionne` avec la fonction `union_eff` écrite plus haut.

```
let rec eclate u =
  match u with
  | [] -> ([], [])
  | [x] -> ([x], [])
  | x :: y :: xs -> let a, b = eclate xs in (x :: a, y :: b)

let rec tri_uniques u =
  match u with
  | [] -> []
  | [x] -> [x]
  | _ ->
    let a, b = eclate u in
    let a', b' = tri_uniques a, tri_uniques b in
    union_eff a' b'
```

Correction de l'exercice VII.6 page 405

On utilise l'observation suivante :

$$\mathcal{P}(x :: xs) = \mathcal{P}(xs) \sqcup \{x \cup A \mid A \in \mathcal{P}(xs)\}$$

où l'union est disjointe.

```
let rec parties u =
  match u with
  | [] -> [[]]
  | x :: xs -> let sans_x = parties xs in
    let avec_x = map_prefixe x sans_x in
    sans_x @ avec_x
```

Correction de l'exercice VII.7 page 406

1. Une combinaison à n éléments de $x :: xs$ est soit de la forme $x :: v$ avec v une combinaison à $n - 1$ éléments de xs , soit une combinaison à n éléments de xs .

Il faut faire **très attention** aux cas de base : pour tout ensemble X , l'ensemble des combinaisons à 0 élément est $\{\emptyset\}$, mais l'ensemble des combinaisons à n éléments avec $n \geq 1$ est \emptyset si X vaut \emptyset .

```
let rec combinaisons u n =
  match u, n with
  | _, 0 -> [[]]
  | [], _ -> []
  | x :: xs, _ ->
    let sans_x = combinaisons xs n in
    let avec_x = map_prefixe x (combinaisons xs (n - 1)) in
    sans_x @ avec_x
```

2. Je donne une version concise mais un peu indigeste. La fonction `List.flatten` (qui appartient à la bibliothèque standard) est redéfinie à l'exercice suivant.

```
let parties_bis u =
  let rec (<|>) a b = if a >= b then [] else a :: (a + 1 <|> b) in
  List.flatten (List.map (combinaisons u) (0 <|> List.length u + 1))
```

3. Très similaire à `combinaisons`, sauf que l'on peut réutiliser `x` même si on l'a choisi.

```
let rec avec_repetitions u n =
  match u, n with
  | _, 0 -> [[]]
  | [], _ -> []
  | x :: xs, _ ->
    let sans_x = avec_repetitions xs n in
    let avec_x = map_prefixe x (avec_repetitions xs (n - 1)) in
    avec_x @ sans_x
```

Correction de l'exercice VII.8 page 406

1. Ce n'est pas difficile mais il faut faire attention au cas de base :

```
let rec insertions x u =
  match u with
  | [] -> [[x]]
  | y :: ys -> (x :: u) :: (map_prefixe y (insertions x ys))
```

2. Une version élémentaire :

```
let rec applatit u =
  match u with
  | [] -> []
  | x :: xs -> x @ applatit xs
```

Cela s'exprime aussi très naturellement à l'aide d'un *fold* :

```
let applatit = List.fold_left ( @ ) []
```

3. Avec tous les outils, il ne reste pas grand-chose à faire :

```
let rec permutations u =
  match u with
  | [] -> [[]]
  | x :: xs -> List.flatten (List.map (insertions x) (permutations xs))
```

Correction de l'exercice VII.9 page 406

On construit toutes les $p - 1$ -listes puis l'on rajoute chaque x de la liste d'entrée devant chacune de ces $p - 1$ -listes :

```
let rec plistes u p =
  if p = 0 then
    []
  else
    let moins_un = plistes u (p - 1) in
    List.flatten (List.map (fun x -> map_prefixe x moins_un) u)
```

Correction de l'exercice VII.10 page 407

- En utilisant un tableau, il n'y a pas de grosse difficulté. On initialise un tableau `present` de n booléens à `false` puis l'on parcourt le tableau `t` en passant la case j de `present` à `true` dès que l'on trouve l'entier j dans `t`. Il ne reste alors plus qu'à chercher le premier `false` dans `present` (s'il n'y en a pas, c'est que le tableau `t` contenait exactement les entiers $0, \dots, n - 1$ et il faut donc renvoyer n).

```
let ppm_array t =
  let n = Array.length t in
  let present = Array.make n false in
  for i = 0 to n - 1 do
    if t.(i) >= 0 && t.(i) < n then present.(t.(i)) <- true
  done;
  let k = ref 0 in
  while !k < n && present.(!k) do
    incr k
  done;
  !k
```

- Si l'on veut procéder de manière purement fonctionnelle, c'est nettement plus délicat.

- La fonction `partition` ne pose pas de problème :

```
let partition u b =
  let rec aux petits grands k = function
  | [] -> (petits, grands, k)
  | x :: xs when x < b -> aux (x :: petits) grands (k + 1) xs
  | x :: xs -> aux petits (x :: grands) k xs in
  aux [] [] 0 u
```

- Maintenant, il faut être astucieux. Si vous êtes arrivé jusqu'ici, vous ferez bien l'effort de lire le code suivant même sans explication...

```
let ppm_list u =
  let rec aux a n u =
    if n = 0 then a
    else
      let b = a + 1 + n / 2 in
      let v, w, m = partition u b in
      if m = b - a then aux b (n - m) w
      else aux a m v in
  aux 0 (List.length u) u
```

ÉCRIVONS DES PROGRAMMES !

I Introduction à la ligne de commande

Pour commencer, lancez un *terminal*. Il y a plusieurs moyens de faire cela, suivant la configuration exacte de l'ordinateur :

- chercher dans le menu d'applications (en haut à gauche en salle 725);
- utiliser le raccourci clavier Ctrl-Alt-T s'il est configuré;
- sur les portables de la salle 822, cliquer sur le raccourci présent à gauche dans la barre de lancement rapide;
- toujours sur ces portables, appuyer sur la touche « Windows » du clavier, puis commencer à taper « terminal » jusqu'à ce que la bonne application soit sélectionnée;
- sur Jupyter, choisir « Terminal » au lieu de « OCaml Notebook » dans le lanceur.

Vous obtenez une *invite de commande*, dans laquelle vous allez pouvoir interagir avec un *shell* (ou *interpréteur de commandes*). Fondamentalement, ce *shell* joue le même rôle qu'un *shell* Python ou qu'un *toplevel OCaml* : il permet d'exécuter des instructions écrites dans un langage de programmation (*bash*, en l'occurrence) et de voir s'afficher le résultat de ces instructions.

► **Question 1** Tapez la ligne suivant (en appuyant sur Entrée uniquement quand vous avez fini la ligne) :

```
for i in 1 3 5 ; do echo $i ; done
```

Il y a cependant une grosse différence avec la boucle interactive d'un langage de programmation : le *shell* va surtout servir à exécuter des programmes pré-existants, écrits dans un langage quelconque au départ (mais mis sous préalablement sous une forme exécutable pour la machine).

Remarque

Nous n'utiliserons jamais *bash* comme un langage de programmation : la boucle *for* ci-dessus sera essentiellement le seul exemple que nous verrons cette année...

Quelques commandes au hasard

► **Question 2** Exécutez les commandes suivantes :

1. `whoami` (qui suis-je?)
2. `date` (ça devrait être clair)
3. `htop` (utilisation des ressources ; appuyez sur F1 pour avoir de l'aide, F10 ou q pour quitter)
4. `ocaml` (pour quitter, exécuter `exit 0`)
5. `python` (pour quitter, exécuter `exit()`)
6. `ping google.com` (pour quitter appuyer sur Ctrl-C, ce qui est une règle générale quand une commande ne termine pas)

► **Question 3**

1. Exécuter la commande `micro hello.py`. Cela lance l'éditeur de texte `micro` et ouvre le fichier `hello.py` (qui est créé à cette occasion, puisqu'il n'existe pas).
2. Dans l'éditeur, taper la ligne `print("Hello, world!")` puis sauvegarder le fichier (Ctrl-S) et quitter (Ctrl-Q).
3. Vous revenez dans le terminal : exécuter à présent la commande `python hello.py`.

Obtenir de l'aide

- **Question 4** Exécuter la commande `ls` (pour *list* : cette commande liste les fichiers du répertoire courant).

En réalité, cette commande admet un très grand de variantes, gouvernées par des *flags*. Si l'on souhaite par exemple un affichage détaillé avec un fichier par ligne, trié par taille décroissante, on pourra exécuter une commande du type `ls -X -Y` où X et Y sont bien choisis. Comme personne ne connaît toutes les options de toutes les commandes, plusieurs mécanismes sont prévus pour obtenir de l'aide.

- **Question 5**

1. Exécuter la commande `ls --help`. Et puis exécuter `date --help`, `ping --help`...
2. Quelle commande faut-il exécuter pour que `ping` s'arrête au bout de 3 réponses ?

Pour `ls`, l'aide obtenue avec `--help` est un peu indigeste. Il y a des moyens d'en extraire l'information que l'on recherche, mais il est souvent plus pratique d'utiliser les *man pages* (les *pages du manuel*). Pour cela on tapera `man nom_de_la_commande`. Les *man pages* sont en général très complètes.

- **Question 6**

1. Exécuter la commande `man ls` et essayer de trouver les options à utiliser pour obtenir l'affichage évoqué plus haut (un fichier par ligne, avec détails, trié par taille décroissante).
2. `gcc` est le compilateur C le plus couramment utilisé sous Linux. Exécuter `man gcc`, et constater que parfois, il peut y avoir un peu trop d'informations...
3. Exécuter la commande `tldr ls` et observer le résultat.

Remarque

L'utilitaire `tldr` (*Too Long, Didn't Read*) est installé sur les machines du lycée mais il ne fait pas partie des outils standard installés par défaut sur tout système Gnu/Linux.

Système de fichiers

Pour se déplacer et se repérer dans l'arborescence, les commandes de base sont :

- `pwd` pour *Print Working Directory*, qui affiche le chemin complet du répertoire courant;

```
> pwd
/home/jb/boulot/mp2i/ocaml-playground
```

La valeur affiché signifie ici que je me trouve dans le sous-répertoire `ocaml-playground` du sous-répertoire `mp2i` du sous-répertoire ... du sous-répertoire `home` de la racine du système de fichier.

- `ls` que nous avons déjà vu pour lister les fichiers dans un répertoire. Sans argument, `ls` liste le contenu du répertoire actuel :

```
> ls
_build  dune  dune-project  hanoi.ml  rpl  rpl.ml  rpl.o  test.ml
```

On peut aussi donner à `ls` un *chemin absolu* commençant à la racine du système de fichiers :

```
> ls /etc/ssh
moduli  ssh_config  sshd_config
```

Ou un *chemin relatif*, qui commence implicitement dans le répertoire courant. Par exemple, le `_build` affiché plus haut par `ls` est un répertoire, et pour lister son contenu on peut faire `ls _build`, qui équivaut à `ls /home/jb/boulot/mp2i/ocaml-playground/_build`.

Tout répertoire contient deux liens spéciaux : « . » qui pointe vers le répertoire lui-même et « .. » qui pointe vers le répertoire parent. `ls .` équivaut donc à `ls` tout court, et `ls ../blah` listerait le contenu de `ls /home/jb/boulot/mp2i/blah` (si ce répertoire existe).

- `cd` pour *Change Directory* pour se déplacer dans un nouveau répertoire. Comme pour `ls`, on peut utiliser un chemin absolu (`cd /etc`) ou relatif (`cd _build`, ou `cd ../blah` par exemple).

2 Compilation et exécution d'un programme OCaml

Jusqu'à présent, nous n'avons pas vraiment écrit de *programme* OCaml : nous n'avons écrit que des *fonctions*. La différence entre un programme et une (ou plusieurs) fonctions, c'est qu'un programme peut être exécuté et est censé faire quelque chose!

2.1 Exécution d'un programme

Pour simplifier, un programme OCaml est simplement une succession de *let bindings* qui sont évalués les uns après les autres. Par exemple :

```
let f x y = 10 * x + y  
let a = 12  
let b = 23  
let resultat = f a b
```

► Question 7

1. En utilisant `micro` ou `VS Code` (que l'on peut lancer depuis le terminal avec la commande `code`), créer un fichier `programme.ml` contenant les lignes ci-dessus.
2. Dans le terminal, naviguer vers le répertoire contenant ce fichier, et exécuter `ocaml programme.ml`. Constater qu'il ne se passe rien de très intéressant.

Le problème est simple : notre programme est « pur », c'est-à-dire qu'il n'a aucun effet secondaire. Si on veut qu'il *fasse* quelque chose (qu'il affiche une valeur, qu'il crée un fichier...), il faut lui demander. On pourrait être tenté d'écrire quelque chose comme :

```
(*****)  
(* CODE FAUX *)  
(*****)  
  
let f x y = 10 * x + y  
  
let a = 12  
  
let b = 4  
  
let resultat = f a b  
  
print_int resultat
```

Cela ne marche pas du tout! Les retours à la ligne n'ont pas de signification particulière en OCaml, donc les trois dernières lignes équivalent en fait à :

```
let mon_resultat = f a b print_int resultat
```

Cela ne veut rien dire, et ce n'est pas correctement typé.

En réalité, on utilisera aussi un *let binding* pour la dernière ligne. Comme `print_int resultat` est de type `unit` et n'est évalué que pour ses effets secondaires, il n'y a aucun intérêt à lui donner un nom. On pourra donc écrire :

```
...  
let resultat = f a b  
  
let _ = print_int resultat
```

En réalité, on préférera écrire :

```
let () = print_int résultat
```

Cette version a l'avantage de provoquer une erreur de type si l'expression à droite du `=` n'est pas de type `unit` (et on veut avoir une erreur dans ce cas : on s'est clairement trompé quelque part).

► Question 8

1. Modifier votre fichier `programme.ml` en utilisant la dernière version présentée ci-dessus.
2. Exécuter à nouveau la commande `ocaml programme.ml`. Constater que l'affichage obtenu n'est pas très satisfaisant.
3. Modifier à nouveau le programme en ajoutant un `print_newline ()`, et vérifier qu'on obtient bien ce que l'on veut.

2.2 Compilation

Pour l'instant, nous avons exécuté notre programme par la commande `ocaml mon_programme.ml`, de la même manière que nous avions utilisé `python hello.py` pour notre "Hello, world!" en Python. C'est une manière normale de procéder en Python, mais en OCaml on ne ferait normalement pas cela. À la place, on *compilerait* `programme.ml`, pour le transformer en un *fichier exécutable* (disons `programme.exe`, le nom importe peu). Ce fichier exécutable n'a plus aucun lien avec OCaml : la personne qui l'exécute n'a pas besoin d'avoir OCaml installé sur son système, ni même de savoir que le programme a été écrit dans ce langage.

Pour compiler `programme.ml` en `programme.exe`, on utilisera la commande suivante :

```
ocamlopt -o programme.exe programme.ml
```

- `ocamlopt` est le nom du compilateur OCaml.
- La partie `-o programme.exe` nous permet de choisir le nom donné au fichier exécutable produit. Elle est optionnelle : si on l'omet, le fichier s'appellera `a.out`

► Question 9

1. Compiler le programme à l'aide de la commande ci-dessus.
2. Lancer l'exécutable produit avec la commande `./programme.exe` (nous verrons plus tard pourquoi le `./` au début est nécessaire).

3 Arguments en ligne de commande

Toutes les commandes vues jusqu'à présent acceptent des arguments en ligne de commande. Par exemple :

- `ls` accepte le nom d'un répertoire et va alors lister le contenu de ce répertoire;
- `ocamlopt` prend le nom du fichier `.ml` à compiler
- ...

Quand on écrit un programme OCaml, on peut accéder aux arguments de ligne de commande à l'aide du tableau `Sys.argv`, de type `string array`.

- `Sys.argv.(0)` contient le nom de la commande qui a été utilisée pour invoquer le programme.
- Les autres cases du tableau contiennent les différents arguments reçus, sous forme de chaînes de caractères. On considère que les arguments sont délimités par les espaces sur la ligne de commande.

► Question 10 Si l'on exécute `./programme.exe 12 aei23 1.3`, quelle sera la taille du tableau `Sys.argv`, et quel sera son contenu ?

► Question 11 Écrire un programme OCaml qui accepte un nombre quelconque d'arguments entiers en ligne de commande et affiche la somme de ces arguments.

La fonction `int_of_string` permet de convertir une chaîne de caractères en entier.

PETITS PROBLÈMES D'ALGORITHMIQUE

Consignes et notations

Notations Si u est une suite et $a \leq b$ sont deux entiers naturels, on note $u[a, b]$ la suite finie u_a, \dots, u_{b-1} (qui contient $b - a$ termes). On parlera de *section* de la suite u . On autorise *a priori* les sections vides (avec $a = b$) mais cela n'a normalement aucun impact sur les réponses attendues.

Consignes

Les deux premières lignes du fichier qui vous a été fourni sont :

```
let u0_exemple = 31122010
let u0 = 42
```

Commencez par remplacer la valeur de $u0$ par votre date de naissance, au format JJMMAAAA (le $u0_exemple$ correspond par exemple à une date de naissance le 31 décembre 2010).

À partir de ces valeurs sont calculés quatre tableaux d'entiers (type **int array**) :

- un tableau u_{ex} , de longueur 2 000 000, qui contient des entiers positifs ou nuls, pseudo-aléatoires, générés à partir de la valeur $u0_exemple$, qui est donc le même pour tout le monde ;
- un tableau u , similaire à u_{ex} mais généré à partir de votre $u0$ personnel ;
- un tableau v_{ex} , de longueur 20 000 000, qui contient **uniquement des 0 et des 1**, et qui est également pseudo-aléatoire, identique pour tout le monde ;
- un tableau v , similaire à v_{ex} mais généré à partir de votre $u0$ personnel.

À part la partie 2 (complètement indépendante), toutes les questions consistent à calculer une certaine valeur à partir du tableau u ou du tableau v . Ces valeurs sont à compléter sur la fiche réponse fournie ; cette fiche contient également les valeurs que vous devez obtenir pour u_{ex} et v_{ex} , et vous permet donc de tester votre code. Dans les parties 3 à 5, un algorithme naïf permettra généralement d'obtenir la première réponse à une question, mais il faudra souvent un algorithme efficace pour les obtenir toutes. À part pour la partie 2, je vous conseille fortement d'utiliser des boucles et des références.

I Calculs élémentaires

► **Question I2** Donner les quatre derniers chiffres de :

1. u_1
2. u_{10}
3. $u_{1\ 000\ 000}$

► **Question I3** Donner les quatre derniers chiffres de :

1. $\sum u[0, 10]$
2. $\sum u[1\ 000, 2\ 000]$
3. $\sum u[1\ 990\ 000, 2\ 000\ 000]$

► **Question I4** Pour chacune des sections suivantes de u , donner l'indice de la première apparition de son minimum :

1. $u[0, 1\ 000]$
2. $u[0, 50\ 000]$
3. $u[1\ 000\ 000, 1\ 050\ 000]$

2 Tri sélection d’une liste

Cette partie est complètement indépendante du reste du sujet, et elle sera évaluée différemment : ces questions doivent être traitées sur feuille et rendues. Bien entendu, vous pouvez commencer par programmer et tester vos fonctions sur machine.

► **Question 15** Écrire une fonction `min_liste` qui prend en entrée une liste supposée non vide et renvoie son minimum.

```
min_liste : 'a list -> 'a
```

► **Question 16** Écrire une fonction `enleve` qui prend en entrée une liste u et une valeur x , et renvoie la liste u dans laquelle la première occurrence de x a été supprimée. Si x n’apparaît pas dans u , on renverra la liste u .

```
utop[2]> enleve 7 [1; 5; 2; 7; 3; 7; 8];
- : int list = [1; 5; 2; 3; 7; 8]
utop[3]> enleve 10 [1; 5; 2; 7; 3; 7; 8];
- : int list = [1; 5; 2; 7; 3; 7; 8]
```

```
enleve : 'a -> 'a list -> 'a list
```

► **Question 17** Écrire une fonction `extrait_min` qui prend en entrée une liste u supposée non vide et renvoie le couple (m, u') où m est le minimum de u et u' est la liste obtenue en supprimant la première occurrence de m dans u .

```
utop[4]> extract_min [4; 7; 1; 6; 2; 4; 1; 8];
- : int * int list = (1, [4; 7; 6; 2; 4; 1; 8])
```

```
extrait_min : 'a list -> 'a * 'a list
```

► **Question 18** Écrire une fonction `tri_selection` qui trie une liste u suivant le principe suivant :

- si la liste est vide, elle est déjà triée ;
- sinon, on extrait son minimum m et l’on obtient une liste u' dans laquelle (la première occurrence de) ce minimum a été supprimée ;
- on obtient une version triée de u en ajoutant m devant une version triée de u' .

```
utop[7]> tri_selection [4; 2; 0; 1; 0; 3; 3; 7; 0];
- : int list = [0; 0; 0; 1; 2; 3; 3; 4; 7]
```

```
tri_selection : 'a list -> 'a list
```

► **Question 19** On souhaite obtenir une fonction qui prend en entrée une liste et renvoie la liste de ses éléments distincts triés par ordre croissant. Par exemple, pour la liste d’entrée $[4; 2; 0; 1; 0; 3; 3; 7; 0]$, le résultat doit être $[0; 1; 2; 3; 4; 7]$. Quelle modification très simple peut-on apporter aux fonctions écrites ci-dessus pour ce faire ?

3 Palindromes

Pour une section $v[a, b] = (v_a, \dots, v_{b-1})$, on définit $\overline{v[a, b]} = (v_{b-1}, v_{b-2}, \dots, v_a)$. Une section est un *palindrome* si $v[a, b] = \overline{v[a, b]}$. Par exemple, en supposant que $v = 1, 1, 1, 0, 0, 1, 0, 0, 1, 1 \dots$:

- $v[1, 2] = (1)$ est un palindrome;
- $v[1, 3] = (1, 1)$ est un palindrome;
- $v[1, 4] = (1, 1, 0)$ n'est pas un palindrome;
- $v[2, 6] = (1, 0, 0, 1)$ est un palindrome;
- $v[4, 7] = (0, 1, 0)$ est un palindrome;
- $v[4, 8] = (0, 1, 0, 0)$ n'est pas un palindrome.

► **Question 20** Combien y a-t-il de couples $(b - l, b)$ tels que $v[b - l, b]$ soit un palindrome avec :

1. $l = 6, l \leq b \leq 1\,000$?
2. $l = 7, l \leq b \leq 1\,000$?
3. $l = 20, l \leq b \leq 1\,000\,000$?

► **Question 21** On note $nb_{pal}(n)$ le nombre total de palindromes non vides contenus dans $v[0, n]$. Autrement dit, $nb_{pal}(n)$ est le nombre de couples (a, b) avec $0 \leq a < b \leq n$ tels que $v[a, b]$ soit un palindrome. Pour chacune des valeurs de n qui suivent, donner les quatre derniers chiffres de $nb_{pal}(n)$.

1. $n = 100$
2. $n = 10\,000$
3. $n = 2\,000\,000$

4 Sections équilibrées

On définit :

- $un(a, b)$ comme le nombre de 1 dans la section $v[a, b]$

$$un(a, b) = \text{Card}\{i \in \{a, \dots, b-1\} \mid v_i = 1\}$$

- $zero(a, b)$ comme le nombre de 0 dans la section $v[a, b]$

$$zero(a, b) = \text{Card}\{i \in \{a, \dots, b-1\} \mid v_i = 0\}$$

- $\delta(a, b)$ comme l'écart entre les deux

$$\delta(a, b) = un(a, b) - zero(a, b)$$

Une section $v[a, b]$ de la suite v est dite *équilibrée* si $\delta(a, b) = 0$ (si elle contient autant de 0 que de 1).

► **Question 22** Combien y a-t-il de sections équilibrées de la forme $v[b - l, b]$ avec :

1. $l = 10, l \leq b \leq 1\,000$?
2. $l = 100, l \leq b \leq 1\,000\,000$?
3. $l = 100\,000, l \leq b \leq 1\,000\,000$?

► **Question 23** Quelle est la longueur maximale d'une section $v[a, b]$ équilibrée avec :

1. $0 \leq a \leq b \leq 100$?
2. $a = 0, 0 \leq b \leq 1\,000\,000$?
3. $0 \leq a \leq b \leq 10\,000$?

► **Question 24** Donner le plus petit et le plus grand $b \in \{0, \dots, 1\,000\,000\}$ pour lesquels $\delta(0, b)$ prend les valeurs suivantes (on répondra -1 si la valeur n'est jamais prise) :

1. $\delta(0, b) = 0$
2. $\delta(0, b) = -12$
3. $\delta(0, b) = 27$

► **Question 25** Quelle est la longueur maximale d'une section $v[a, b]$ équilibrée avec :

1. $0 \leq a \leq b \leq 100\,000$? 2. $0 \leq a \leq b \leq 500\,000$? 3. $0 \leq a \leq b \leq 2\,000\,000$?

5 Question bonus

► **Question 26** On appelle *sous-suite* d'une section $v[a, b]$ une suite extraite de $v[a, b]$, c'est-à-dire une suite finie $(v_{\varphi(0)}, \dots, v_{\varphi(k-1)})$ où $\varphi : \{0, \dots, k-1\} \rightarrow \{a, \dots, b-1\}$ est strictement croissante. La *longueur* de la sous-suite est alors k . Par exemple, en supposant que $w[10, 16] = (12, 7, 3, 5, 3, 12)$:

- la suite vide l'unique sous-suite de $w[10, 16]$ de longueur 0;
- $(12, 7, 3, 5, 3, 12)$ est l'unique sous-suite de $w[10, 16]$ de longueur 6;
- $(12, 3, 3, 12) = (w_{10}, w_{12}, w_{14}, w_{15})$ est une sous-suite de longueur 4;
- $(12, 3, 12, 3)$ n'est pas une sous-suite de $w[10, 16]$.

Parmi toutes les sous-suites *croissantes* de $w[10, 16]$, les plus longues sont $(3, 5, 12)$ et $(3, 3, 12)$; la longueur maximale d'une sous-suite croissante de $w[10, 16]$ est donc 3.

Donner la longueur maximale d'une sous-suite croissante de :

1. $u[0, 20]$ 2. $u[0, 400]$ 3. $u[0, 5000]$

PILES, FILES

I Files fonctionnelles

Dans cette partie, on réalise une file fonctionnelle en utilisant deux listes, comme décrit dans le cours. On choisit le type le plus simple :

```
type 'a file_fonct = 'a list * 'a list
```

Les fonctions élémentaires sont :

```
(* Une constante égale à la file ne contenant aucun élément. *)
file_vide = ([], [])

ajoute : 'a -> 'a file_fonct -> 'a file_fonct
(* ajoute x f renvoie une nouvelle file contenant les éléments de f, plus x *)

enleve : 'a file_fonct -> ('a * 'a file_fonct) option
(* enleve f renvoie Some (x, f'), où x est l'élément le plus ancien
de f et f' est la file obtenue en enlevant x à f.
Si f est vide, on renverra None. *)
```

Exercice X.1 – Programmation des opérations élémentaires

- Écrire une fonction miroir : `'a list -> 'a list` se comportant comme `List.rev`. On exige une complexité linéaire en la taille de la liste.

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

- Écrire la fonction ajoute.
- Écrire la fonction enleve.

Exercice X.2 – Opérations supplémentaires

Dans cet exercice (sauf mention contraire explicite), on n'interagira avec les files que par le biais des opérations élémentaires définies plus haut. Autrement dit, on « oublie » qu'une file est représentée par un couple de listes.

- Écrire une fonction somme : `int file_fonct -> int` renvoyant la somme des éléments d'une file.
- Écrire une fonction `file_fonct_of_list` : `'a list -> 'a file_fonct` qui convertit une liste en file, l'élément de tête de la liste se retrouvant « à droite » de la file.
- Proposer une version plus simple et plus efficace de cette fonction en s'autorisant à interagir directement avec la représentation concrète du type file.
- Écrire une fonction `itere_file` : `('a -> unit) -> 'a file_fonct -> unit` telle que l'appel `itere_file f file`, où `file = (x1, ..., xn)` (avec x_n le plus ancien élément), soit équivalent à appeler la fonction `f` successivement sur x_n , puis x_{n-1}, \dots , puis x_1 .

5. Écrire une fonction `afficher : int file_fonct -> unit` qui affiche tous les éléments d'une file d'entiers, dans leur ordre d'insertion. On rappelle que :

- `print_int : int -> unit` permet d'afficher un entier;
- `print_newline : unit -> unit` permet de revenir à la ligne.

Attention, quand on utilise Jupyter il faut toujours terminer par un `print_newline ()` pour que l'affichage se fasse effectivement.

```
# afficher (file_fonct_of_list [1; 2; 3; 4]);
1
2
3
4
- : unit = ()
```

Exercice X.3 – Complexité amortie

1. On note $|u|$ la longueur d'une liste u , et l'on définit le *potentiel* d'une file $f = (e, s)$ par $\Phi(f) = 2|e|$. On mesure la complexité d'une opération par le nombre d'utilisations du constructeur `::`, en comptant à la fois la version « destructive » (qui permet de récupérer la tête et la queue d'une liste pré-existante) et la version « constructive » (qui permet de construire $x :: xs$ à partir de x et de xs).

On considère une file f et une opération qui est soit une extraction, soit l'ajout d'un élément (quelconque). On note $C_{op}(f)$ le coût de l'opération op sur la file f et f' la file obtenue après l'opération. Donner une majoration (précise) de $C_{op}(f) + \Phi(f') - \Phi(f)$.

2. En déduire que le coût total de n opérations (ajouts ou suppressions) successives sur une file initialement vide est en $O(n)$, et donc que le coût moyen par opération (sur cette série de n opérations) est en $O(1)$. On dira donc que l'insertion et l'extraction ont une *complexité amortie* en $O(1)$.

2 Piles et files impératives

2.1 Réalisation d'une pile par un tableau

On définit :

```
type 'a pile = {donnees : 'a option array; mutable courant : int}

let capacite p = Array.length p.donnees
```

On rappelle le principe de la réalisation (on pourra se référer au schéma du cours) :

- une pile a une taille maximale (on parlera de capacité) fixée à la création : c'est la taille du tableau `donnees`;
- à tout moment, `courant` donne l'indice du sommet de la pile (qui vaudra -1 si la pile est vide et n'a donc pas de sommet);
- les éléments sont empilés de la gauche vers la droite du tableau : les cases du tableau situées à droite du sommet actuel peuvent contenir n'importe quoi (`None` typiquement, mais pas nécessairement) et sont considérées comme vides.

Exercice X.4

Définir les fonctions :

- `nouvelle_pile : int -> 'a pile` (crée une pile vide, l'entier donne la capacité);
- `pop : 'a pile -> 'a option;`

- push : 'a -> 'a pile -> unit.

```
# let s = nouvelle_pile 2;;
val s : '_a pile = {donnees = [|None; None|]; courant = -1}
# push 10 s;;
- : unit = ()
# push 20 s;;
- : unit = ()
# pop s;;
- : int option = Some 20
# push 1 s;;
- : unit = ()
# push 3 s;;
Exception: Failure "Pile pleine".
```

2.2 File dans un tableau circulaire

On définit :

```
type 'a file_i =
{donnees : 'a option array;
 mutable entree : int;
 mutable sortie : int;
 mutable cardinal : int}
```

Pour une file f :

- f.entree pointera vers la case où se fera la prochaine insertion;
- f.sortie pointera vers la case où se fera la prochaine extraction;
- f.cardinal indique la taille actuelle de la file (le nombre de cases actuellement utilisées);
- si la file est vide, on aura f.entree = f.sortie (mais cette valeur peut être quelconque).

En OCaml, l'opération n mod p renvoie un nombre négatif si $n < 0$ et $p > 0$: pour éviter d'avoir à en écrire une variante, on décide que la file se déplacera vers la droite (c'est-à-dire que f.entree ou f.sortie sera incrémenté à chaque opération).

Exercice X.5

1. Représenter deux 'a file_i de capacité 6 correspondant à la file $\leftarrow \boxed{3} \boxed{1} \boxed{2} \boxed{7} \leftarrow$:
 - une avec f.entree < f.sortie;
 - une avec f.entree > f.sortie.
2. Il est pratique mais pas nécessaire de disposer des trois entiers f.entree, f.sortie et f.cardinal. Expliquer comment retrouver f.sortie si l'on dispose de f.entree et f.cardinal.
3. Expliquer pourquoi il n'est en revanche pas possible de retrouver f.cardinal à partir de f.sortie et f.entree (on pourra éventuellement suggérer une « astuce » permettant de régler ce problème sans stocker le cardinal).

Exercice X.6

Écrire les fonctions suivantes :

1. file_vide_i : int -> 'a file_i où l'entier spécifie la capacité de la file;
2. capacite_i : 'a file_i -> int qui renvoie la capacité;
3. ajoute_i : 'a -> 'a file_i -> unit (si la file est pleine, on le signalera en levant une exception par failwith "Insertion dans file pleine").

4. `enleve_i : 'a file_i -> 'a option` (on modifiera la file, et renverra `None` si elle est déjà vide);
5. `de_liste_i : 'a list -> int -> 'a file_i`. On demande que `de_liste_i n u` crée une file de capacité `n` et y ajoute les éléments de `u` en commençant par l'élément de tête. On pourra supposer sans le vérifier que $n \geq |u|$.

2.3 Fonctions supplémentaires sur les piles

Exercice X.7

Techniquement, les trois fonctions définies à la partie 2.1 (plus la fonction `capacite`) suffisent pour réaliser toutes les opérations imaginables sur des piles. Dans cet exercice, on se restreint à n'interagir avec les piles que via ces fonctions : on « oublie » donc l'existence d'un tableau sous-jacent.

1. Écrire une fonction `peek_1 : 'a pile -> 'a option`, qui renvoie `Some x` si on l'appelle sur une pile de sommet `x`, `None` si on l'appelle sur une pile vide. Cette fonction ne doit pas modifier son argument :

```
# let s = nouvelle_pile 3;;
val s : '_a pile = {donnees = [|None; None; None|]; courant = -1}
# push 3 s;;
- : unit = ()
# peek_1 s;;
- : int option = Some 3
# peek_1 s;;
- : int option = Some 3
```

2. Écrire une fonction `est_vide_1 : 'a pile -> bool` qui détermine si son argument est vide.
3. On souhaite écrire une fonction `egal : 'a pile -> 'a pile -> bool` qui détermine si deux piles sont égales (contiennent les mêmes éléments dans le même ordre, sans nécessairement avoir la même capacité). La fonction ci-dessous a deux problèmes : quels sont-ils ?

```
let egal_faux s t =
  while not (est_vide_1 s) && not (est_vide_1 t) && pop s = pop t do
    ()
done;
est_vide_1 s && est_vide_1 t
```

4. Écrire une fonction `iter_destructif : (f : 'a -> unit) -> (s : 'a pile) -> unit` qui applique la fonction `f` successivement à chacun des éléments de `s` en commençant par le sommet. À la fin de l'appel, `s` sera vide.
Par exemple, si `s = (12, 35, 1)` (où le sommet est 1), un appel à `iter_destructif print_int s` doit afficher les entiers 1, 35 et 12 dans cet ordre.
5. Compléter la fonction suivante pour qu'elle renvoie une copie de son argument. Cet argument ne doit pas être modifié au cours de l'appel (plus précisément, il doit être remis dans son état initial avant la fin de l'appel).

```

let copie s =
  let s_copie = nouvelle_pile (capacite s) in
  let miroir = nouvelle_pile (capacite s) in
  iter_destructif (fun x -> ..... ) s;
  iter_destructif (fun x -> ..... ) miroir;
  s_copie

```

6. Écrire alors une version correcte de la fonction testant l'égalité de deux piles. On n'hésitera pas à réutiliser la version « fausse » pour écrire le moins de code possible.

Exercice X.8

L'exercice précédent illustre assez bien la différence entre « possible » et « raisonnable » : certes, nous avons réussi à écrire une fonction d'égalité, mais au prix de contorsions rocambolesques. Comme l'utilisateur de notre structure de données n'aura effectivement pas accès au type concret sous-jacent (*a priori*, il ne saura même pas que les piles sont implémentées à l'aide de tableaux), il est de notre responsabilité de lui fournir **celles des fonctions qui sont beaucoup plus simples à écrire en ayant accès à l'implémentation**. Typiquement :

- `iter_destructif` s'écrit de manière très naturelle avec uniquement des `pop` (et n'est pas d'utilisation très courante) : on peut laisser l'utilisateur l'implémenter;
- une version non destructive de `iter`, qui ferait la même chose mais ne viderait pas la pile, est en revanche pénible à écrire avec l'interface minimale : essentiellement, on va devoir copier la pile et faire un `iter_destructif` sur la copie (ce qui pose aussi un problème d'efficacité). Cette fonction gagnerait donc à être rajoutée à l'interface de manière à pouvoir utiliser la représentation concrète des piles.

Dans cet exercice, on s'autorise donc de nouveau à utiliser la représentation concrète des piles, et on rajoute quelques fonctions à l'interface. On essaiera d'avoir des implémentations simples et efficaces.

1. Fonction `est_vide` : '`a pile` -> `bool` (une ligne).
2. Fonction `flush` : '`a pile` -> `unit` qui vide son argument (sans rien faire avec ses éléments); on attend une implémentation en temps constant.
3. Fonction `egal` : '`a pile` -> '`a pile` -> `bool` (attention, il faut bien tester l'égalité des éléments des piles, ce qui n'est pas la même chose que l'égalité des contenus des tableaux).
4. Fonction `itere` : ('`a` -> `unit`) -> '`a pile` -> `unit`, version non destructive de `iter_destructif`.

INTRODUCTION AU LANGAGE C

I Hello, World !

Le programme minimal en C est le suivant :

```

1 #include <stdio.h>
2
3 int main(void){
4     printf("Hello, World!\n");
5     return 0;
6 }
```

- À la ligne 1, on a une *directive du pré-processeur*, qui se reconnaît au fait que la ligne commence par un caractère **#**. Ici, on demande l'inclusion du fichier `stdio.h` qui permet d'utiliser les fonctions d'entrée-sortie standard (*Standard Input-Output*). Vous n'aurez pas trop à vous préoccuper des lignes de ce type, au moins pour l'instant : je vous les donnerai.
- Des lignes 3 à 6, on définit une fonction, appelée `main`.
 - À la ligne 3, on trouve son *prototype* : `int main(void)`. Cela signifie que c'est une fonction qui renvoie un entier (type `int`) et ne prend rien en argument (type `void`).
 - Immédiatement après le prototype, il y a une accolade ouvrante, et l'accolade fermante qui lui correspond est à la ligne 6. Une partie du programme située entre accolades est appelé un *bloc* en C. Ici, ce bloc est le corps de la fonction.
 - Chacune des lignes 4 et 5 est un *statement* (une instruction, essentiellement). La première est un appel à la fonction `printf` (qui est déclarée dans `stdio.h`), la deuxième contient le mot-clé `return`.
 - Ces deux lignes se terminent par un point-virgule. Le point-virgule n'a pas le même rôle en C qu'en OCaml : en OCaml, il relie deux expressions, alors qu'en C il *termine* un *statement*. Ainsi, le point-virgule est obligatoire y compris à la ligne 5.
 - Le mot-clé `return` fonctionne essentiellement comme en Python : on sort de la fonction, en renvoyant la valeur de l'expression qui suit le `return` (0, ici). Si l'on ne veut rien renvoyer, on pourra mettre `return;`.
- La fonction `main` est particulière : c'est le point d'entrée du programme. Autrement dit, quand on exécutera notre programme compilé, tout se passera comme si on avait commencé par appeler cette fonction.
- Ainsi, tout fichier contenant un programme « complet » aura une (et une seule) fonction `main`.
- On peut être surpris par le fait que `main` renvoie un entier : quand l'appel à `main` se termine, le *programme entier* se termine et il pourrait donc paraître logique de renvoyer `void` (puisque il n'y a plus personne pour utiliser la valeur de retour). En réalité, cette valeur de retour est transmise au système d'exploitation, et par convention une valeur de zéro signifie que l'exécution s'est déroulée normalement.

Exercice XI.I

1. Ouvrir un terminal (en utilisant le raccourci dans le menu ou dans la barre de lancement rapide, ou le raccourci clavier Ctrl-Alt-T). Créer un répertoire pour ce TP, éventuellement comme sous-répertoire d'un répertoire existant. Pour cela vous aurez besoin de deux commandes :

- `mkdir` pour créer un répertoire;
- `cd` pour changer de répertoire.

Par exemple :

```
$ cd tp-info
$ mkdir tp-11
$ cd tp-11
```

À la première ligne, on se déplace dans le répertoire `tp-info` : il faut qu'il existe déjà, et que ce soit un sous-répertoire du répertoire courant. Ensuite, on crée un répertoire `tp-11` puis on se déplace dans ce nouveau répertoire.

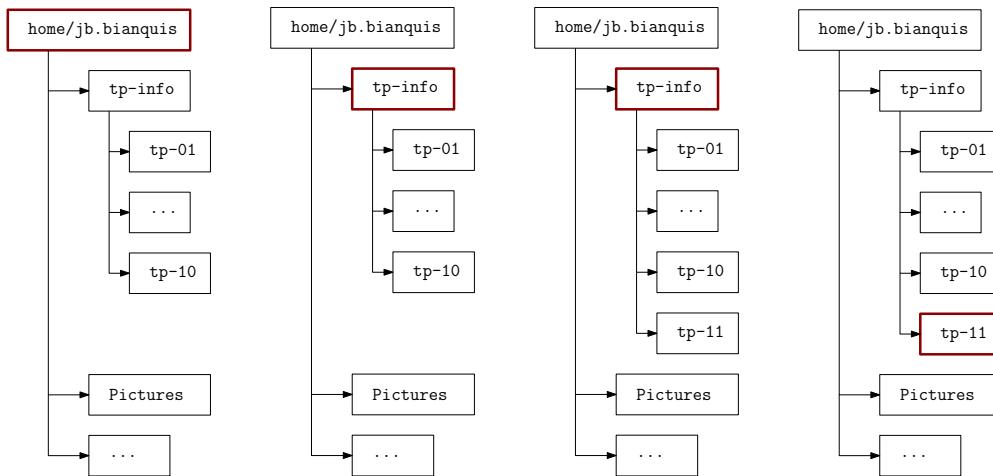


FIGURE XI.1 – Évolution de l’arborescence lors de l’exécution des commandes ci-dessus. Le réertoire en gras est le répertoire courant.

2. Lancer VS Code, ouvrir le répertoire que vous venez de créer, et y créer un fichier que vous appellerez `hello.c`. Recopier le contenu du fichier donné plus haut et sauvegarder.
3. C est un langage purement compilé : autrement dit, il n’y a pas de *toplevel*, de *REPL*, de *boucle interactive*, de *shell*... Pour exécuter notre programme, il faut le *compiler*, c’est-à-dire le traduire en une série d’instructions exécutables par la machine (pour faire très simple...). Pour cela, dans le terminal, exécuter la commande suivante :

```
$ gcc -o hello hello.c
```

- `gcc` est le nom du compilateur C. Vous pourriez le remplacer par `clang` qui est aussi installé sur les machines du lycée, mais de notre point de vue ça ne changera essentiellement rien.
- L’option `-o`, suivie de l’argument `hello`, précise qu’on souhaite que le fichier compilé s’appelle `hello`. Sans cette option (si l’on tape juste `gcc hello.c`), le fichier compilé s’appellera `a.out`.
- `hello.c` est le nom du fichier source C que l’on demande de compiler.

Si vous avez recopié le programme sans erreur, la compilation devrait bien se passer (pas de message d’erreur!).

4. Exécuter ensuite, toujours dans le terminal, la commande suivante :

```
$ ./hello
```

Vous devriez obtenir l’affichage de `Hello, World!`.

Exercice XI.2

1. Observer les messages d'erreur à la compilation :
 - a. si l'on enlève la ligne `#include <stdio.h>` (ce sera sans doute seulement un *warning*);
 - b. si l'on enlève le point-virgule après le `return 0` (cette fois, ce sera une erreur).

Noter aussi que ces problèmes sont signalés en direct sous VS Code.
2. Revenir à une version sans erreur de `hello.c`, la compiler, puis exécuter les deux commandes ci-dessous dans le terminal :

```
$ ./hello
$ echo $?
```

La deuxième commande demande l'affichage (`echo`) du contenu de la variable spéciale `?`. Cette variable contient le code de retour de la dernière commande exécutée, et donc 0 ici puisque notre fonction `main` renvoie 0.

3. Remplacer le `return 0;` par `return 1;`, recommencer les étapes ci-dessus et vérifier que le code de retour a bien changé.

2 Un programme plus complet

```

1 #include <stdio.h>
2
3 int N = 4;
4 double pi;
5
6 double puissance(double x, int n){
7     double p;
8     p = 1.0;
9     while (n > 0){
10         p = p * x;
11         n = n - 1;
12     }
13     return p;
14 }
15
16 void affiche_puis(double x, int n){
17     printf("%f**%d = %f\n", x, n, puissance(x, n));
18 }
19
20 int main(void){
21     pi = 3.1416;
22     int i = 0;
23     while (i < N){
24         affiche_puis(pi, i);
25         i = i + 1;
26     }
27     return 0;
28 }
```

- Aux lignes 3 et 4, on définit deux variables globales.
 - On précise le type de chacune des variables : `int` (l'un des types entiers en C) pour l'une, `double` (l'un des types flottants) pour l'autre.
 - On initialise `N` en même temps qu'on la définit, mais pas `pi`. C'est juste pour illustrer le fait qu'il est possible de définir une variable sans l'initialiser en C.

- On définit ensuite deux fonctions. À chaque fois, on spécifie le type de retour (**double** pour puissance, **void** pour **affiche_puis**) ainsi que le type et le nom des arguments.
- La fonction **affiche_puis** contient une boucle **while**. On remarquera que la condition de boucle est entre parenthèses (obligatoire) et le corps entre accolades (obligatoire sauf s'il ne contient qu'une instruction).
- Toujours dans cette fonction, on a séparé déclaration et initialisation de la variable **p**. À nouveau, aucune raison de le faire, c'est juste pour montrer que c'est possible.
- Dans l'appel à la fonction **printf**, la chaîne de format (le premier argument) contient trois champs : "%f", "%d" et "%f". Cela signifie qu'il faut passer trois arguments supplémentaires, que le premier et le troisième seront affichés comme des flottants et le deuxième comme un entier.
- On initialise la variable globale **pi** à la première ligne de la fonction **main**. Ensuite, on définit et initialise une variable **i**, puis l'on fait une boucle **while**.

Exercice XI.3

p. 435

Écrire un programme **fact.c** produisant, après compilation et exécution, l'affichage suivant :

```
$ ./fact
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
```

On utilisera deux fonctions avec les prototypes suivants :

```
int fact(int n)

void affiche_fact(int n)
```

Si vous débutez complètement en C, vous pouvez sauter l'exercice suivant et y revenir plus tard.

Exercice XI.4

p. 435

On définit $H_n = \sum_{k=1}^n \frac{1}{k}$ et $f(x) = \min(n \in \mathbb{N} \mid H_n \geq x)$ pour $x \in \mathbb{R}$. Écrire un programme donnant l'affichage suivant :

```
$ ./harmonique
f(0.000000) = 0
f(0.500000) = 1
f(1.000000) = 1
f(1.500000) = 2
f(2.000000) = 4
f(2.500000) = 7
f(3.000000) = 11
f(3.500000) = 19
f(4.000000) = 31
```

On essaiera de faire en sorte que le programme soit capable de calculer $f(15)$ en temps raisonnable.

3 Instructions conditionnelles

La syntaxe d'une expression conditionnelle est la suivante :

```
/* Version sans else */
if (condition) statement_true

/* Version avec else */
if (condition) statement_true else statement_false
```

Il s'agit bien d'une *instruction* conditionnelle (comme en Python) et pas d'une *expression* (comme en OCaml) :

- dans le premier cas, `statement_true` est exécuté si la condition est « vraie », rien n'est exécuté (à part l'évaluation de la condition, qui peut avoir des effets secondaires) sinon ;
- dans le deuxième cas, soit `statement_true` soit `statement_false` est exécuté ;
- dans tous les cas, le *if-then-else* n'a pas de « valeur » (contrairement à ce qui se passe en OCaml, répétons-le).

Pour construire la condition, on peut utiliser des opérateurs booléens :

- `&&` pour le « et » logique (comme en OCaml) ;
- `||` pour le « ou » logique (comme en Ocaml) ;
- `!` pour le « non » logique (*pas* comme en OCaml).

Les opérateurs de comparaison, qui fonctionnent sur les types simples (entiers, flottants) sont :

- `==` pour l'égalité ;
- `!=` pour la différence ;
- `<, >, <= et >=`.

Dans presque tous les cas, `statement_true` et `statement_false` seront en fait des *compound statements* (instructions composées). Cela signifie qu'il s'agira à chaque fois d'une série d'instructions, délimitée par des accolades :

```
if (n != 0 && i < 2){
    n = n + 1;
    i = i - 1;
} else {
    n = n + 2;
    i = i + 4;
}
```

Exercice XI.5

p. 436

1. Écrire une fonction `int absolue(int x)` qui renvoie la valeur absolue de son argument. *Cette fonction existe déjà, et s'appelle abs. Il faut inclure l'entête stdlib.h pour y avoir accès.*
2. Écrire une fonction `void affiche_si_pair(int x)` qui affiche l'entier `x` s'il est pair. L'opérateur *modulo* s'écrit `%` en C.
3. Tester ces deux fonctions en écrivant un petit programme, en le compilant et en l'exécutant.

Exercice XI.6

p. 437

Écrire une fonction `bool est_premier(int n)` qui renvoie `true` ou `false` suivant si `n` est premier ou pas (on pourra supposer `n ≥ 0`). **Attention**, pour disposer des constantes `true` et `false`, il faudra ajouter la ligne suivante au début du fichier :

```
#include <stdbool.h>
```

Tester cette fonction en écrivant un programme donnant l'affichage suivant :

```
$ ./premier
0 n'est pas premier
1 n'est pas premier
2 est premier
3 est premier
4 n'est pas premier
5 est premier
6 n'est pas premier
7 est premier
8 n'est pas premier
9 n'est pas premier
```

```
10 n'est pas premier
11 est premier
12 n'est pas premier
13 est premier
14 n'est pas premier
15 n'est pas premier
16 n'est pas premier
17 est premier
18 n'est pas premier
19 est premier
```

4 Boucles

Nous avons déjà rencontré des boucles **while**, qui sont on ne peut plus classiques :

```
while (condition){
    corps_de_la_boucle
}
```

Les boucles **for** ont la structure suivante :

```
for (initialisation; condition; iteration){
    corps_de_la_boucle
}
```

Cela équivaut à :

```
{
    initialisation;
    while (condition){
        corps_de_la_boucle;
        iteration;
    }
}
```

- **initialisation**, **condition** et **iteration** sont optionnels.
- Si **condition** est omis, on considère que la condition est toujours vraie (autrement dit, la boucle est infinie).
- **initialisation** peut être une instruction ou la définition d'une variable. Si l'on utilise une définition de variable, par exemple **for (int i = 0; i < N; i = i + 1)**, alors la variable est locale à la boucle.
- Contrairement à ce qui se passe en Python ou en OCaml, il est tout à fait possible (mais pas du tout recommandé) de modifier manuellement la valeur de la variable de boucle dans le corps de la boucle. Techniquement, il n'y a même pas de notion de variable de boucle, la boucle **for** est juste un raccourci syntaxique pour les formes les plus courantes d'une boucle **while**.
- Comme les boucles **for** sont très flexibles mais restent un cas particulier de boucle **while**, on a très souvent le choix de quelle boucle utiliser. Dans la plupart des cas simples, on préférera une boucle **for**.

Exercice XI.7

p. 437

Ré-écrire les boucles suivantes en utilisant une boucle **while** :

1.

```
for (int i = 0; i < n; i = i + 1){  
    if (i % 2 == 0){  
        printf("%d\n", i);  
    }  
}
```

2.

```
for ( ; i < n; ) {
    if (i % 2 == 1){
        printf("%d\n", i);
        i = i + 1;
    } else {
        i = i * 2;
    }
}
```

3.

```
for (i = 0; n != 0; n = n / 2){  
    i = i + 1;  
}
```

Préciser pourquoi la boucle n'aurait ici aucun intérêt si on remplaçait le `i = 0` par un `int i = 0`.

Exercice XI.8

p. 438

Écrire un programme permettant de réaliser les figures suivantes :

1.

2.

3.

*
**

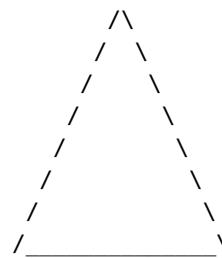
**
*

* *
* *
* *

*
**
* *
* *
* *

*

4.



5 Petits exercices

Exercice XI.9 – Exponentiation rapide

Écrire deux fonctions réalisant une exponentiation rapide, l'une récursive et l'autre itérative.
On utilisera les prototypes suivants :

```
double expo_rapide_it(double x, int n)
```

```
double expo_rapide_rec(double x, int n)
```

Exercice XI.10 – Suite de Syracuse

La suite de Syracuse est définie par la donnée de son premier terme $u_0 \in \mathbb{N}^*$ et par la relation :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Le *temps de vol* $tv(a)$ d'un entier $a \geq 1$ est défini comme le plus petit n tel que $u_n = 1$ lorsqu'on prend $u_0 = a$ comme condition initiale.

1. Déterminer $\max(tv(a) | 1 \leq a \leq 10000)$.
2. Déterminer le nombre d'entiers $a \in [1 \dots 10^5]$ tels que $tv(a) > 20$.

6 Pour chercher

On dispose de deux tableaux d'entiers u et v triés par ordre croissant (le langage n'a pas d'importance, mais ce sont des tableaux : on peut accéder à l'élément i en temps constant). Comment déterminer *efficacement* la valeur qui se trouverait à l'indice i du tableau w que l'on obtiendrait en fusionnant u et v de manière à obtenir un tableau croissant ?

Solutions

Correction de l'exercice XI.3 page 430

On donne une version avec boucle **while**, mais on privilégierait clairement une boucle **for** ici.

```
#include <stdio.h>

int fact(int n){
    int f = 1;
    int i = 1;
    while (i <= n){
        f = f * i;
        i = i + 1;
    }
    return f;
}

void affiche_fact(int n){
    printf("%d! = %d\n", n, fact(n));
}

int main(void){
    int N = 9;
    int i = 0;
    while (i < N){
        affiche_fact(i);
        i = i + 1;
    }
    return 0;
}
```

Correction de l'exercice XI.4 page 430

Pour calculer $f(15)$ en temps raisonnable, il ne faut pas reprendre le calcul de la somme à zéro à chaque fois. On peut procéder ainsi :

```
#include <stdio.h>

int f(double x){
    int n = 0;
    double s = 0. ;
    while (s < x){
        n = n + 1;
        s = s + 1. / n;
    }
    return n;
}
```

```
int main(void){
    double x_max = 15;
    double x = 0. ;
    while (x <= x_max){
        printf("f(%f) = %d\n", x, f(x));
        x = x + 0.5;
    }
    return 0;
}
```

On peut remarquer que cette version n'est pas optimale du tout : pour calculer $f(15)$, il vaudrait mieux repartir de $f(14.5)$! Ce n'est pas très difficile à faire, mais cela ne change pas radicalement le temps de calcul (à cause de la croissance exponentielle de $f(x)$).

Correction de l'exercice XI.5 page 431

```
#include <stdio.h>

int absolue(int n){
    if (n < 0){
        return -n;
    } else {
        return n;
    }
}

// Autre version :
int absolue_2(int n){
    if (n < 0){
        return -n;
    }
    return n;
}

void affiche_si_pair(int n){
    if (n % 2 == 0){
        printf("%d\n", n);
    }
}

int main(void){
    int i = -5;
    while (i < 5){
        affiche_si_pair(i);
        printf("%d\n", absolue(i));
        i = i + 1;
    }
    return 0;
}
```

Correction de l'exercice XI.6 page 431

```
#include <stdio.h>
#include <stdbool.h>

bool est_premier(int n){
    if (n < 2) return false;
    int d = 2;
    while (d * d <= n){
        if (n % d == 0){
            return false;
        }
        d = d + 1;
    }
    return true;
}

void affiche_premier(int n){
    if (est_premier(n)){
        printf("%d est premier\n", n);
    } else {
        printf("%d n'est pas premier\n", n);
    }
}

int main(void){
    int N = 20;
    int i = 0;
    while (i < N){
        affiche_premier(i);
        i = i + 1;
    }
    return 0;
}
```

Correction de l'exercice XI.7 page 433

1.

```
{
    int i = 0;
    while (i < n){
        if (i % 2 == 0){
            printf("%d\n", i);
        }
        i = i + 1;
    }
}
```

2.

```

while ( i < n ){
    if ( i % 2 == 0 ){
        printf("%d\n", i);
        i = i + 1;
    } else {
        i = i * 2;
    }
}

```

3.

```

i = 0;
while ( n != 0 ){
    i = i + 1;
    n = n / 2;
}

```

Cette boucle fait deux choses : elle divise `n` par deux jusqu'à ce qu'il devienne nul et elle compte le nombre d'étapes (c'est le rôle de la variable `i`). Si l'on met `int i = 0` à la place de `i = 0`, la variable `i` devient locale à la boucle, et l'on n'y a donc plus accès après la fin de celle-ci. Dans ce cas, on peut tout aussi bien remplacer toute la boucle par l'instruction `n = 0 !`

Correction de l'exercice XI.8 page 433

```

#include <stdio.h>

void triangle(int n){
    for (int i = 1; i <= n; i = i + 1){
        for (int j = 1; j <= i; j = j + 1){
            printf("*");
        }
        printf("\n");
    }
}

void triangle_inverse(int n){
    for (int i = n; i > 0; i = i - 1){
        for (int j = 1; j <= i; j = j + 1){
            printf("*");
        }
        printf("\n");
    }
}

void ligne(int colonnes){
    for (int i = 0; i < colonnes; i = i + 1){
        printf("*");
    }
    printf("\n");
}

```

```

void rectangle(int lignes, int colonnes){
    for (int i = 0; i < lignes; i = i + 1){
        ligne(colonnes);
    }
}

void rectangle_contour(int lignes, int colonnes){
    ligne(colonnes);
    for (int i = 2; i < lignes; i = i + 1){
        printf("*");
        for (int j = 2; j < colonnes; j = j + 1){
            printf(" ");
        }
        printf("\n");
    }
    ligne(colonnes);
}

void triangle_contour(int n){
    printf("*\n");
    for (int i = 2; i < n; i = i + 1){
        printf("*");
        for (int j = 2; j < i; j = j + 1){
            printf(" ");
        }
        printf("\n");
    }
    ligne(n);
}

void ligne_centree(int largeur, int total){
    int blancs = (total - largeur) / 2;
    for (int i = 0; i < total; i = i + 1){
        if (i < blancs || i >= blancs + largeur){
            printf(" ");
        } else {
            printf("*");
        }
    }
    printf("\n");
}

void triangle_centre(int n){
    int largeur_totale = 2 * n - 1;
    for (int i = 0; 2 * i + 1 <= largeur_totale; i = i + 1){
        ligne_centree(2 * i + 1, largeur_totale);
    }
}

void affiche_blancks(int n){
    for (int i = 0; i < n; i = i + 1){
        printf(" ");
    }
}

```

```
void triangle_centre_contour(int n){
    int blanCs_cote = n;
    int blanCs_milieu = 0;
    while (blanCs_cote >= 1){
        affiche_blanCs(blanCs_cote);
        printf("/");
        affiche_blanCs(blanCs_milieu);
        printf("\\\\");
        affiche_blanCs(blanCs_cote);
        printf("\n");
        blanCs_cote--;
        blanCs_milieu += 2;
    }
    printf("/");
    for (int i = 0; i < blanCs_milieu; i++){
        printf("_");
    }
    printf("\\\\\\n");
}

int main(){
    rectangle(4, 6);
    printf("\n");
    triangle(5);
    printf("\n");
    triangle_inverse(6);
    printf("\n");
    rectangle_contour(5, 4);
    printf("\n");
    triangle_contour(6);
    printf("\n");
    triangle_centre(7);
    printf("\n");
    triangle_centre_contour(7);
    return 0;
}
```

BOUCLES ET TABLEAUX STATIQUES

I Boucles

Si vous n'êtes pas arrivé jusque là lors du TP de mardi, commencez par lire la partie sur les boucles **for** et par faire l'exercice où l'on vous demande d'en ré-écrire quelques unes sous forme de boucle **while**.

Exercice XII.1

Écrire des fonctions calculant les sommes suivantes :

$$1. S_1(n) = \sum_{k=1}^n (2k + 1)$$

$$2. S_2(n) = \sum_{k=1}^n \frac{1}{k^2}$$

Attention, si on écrit n / p avec n et p entiers, on aura une division entière (quotient de la division euclidienne), comme en OCaml. Pour forcer une division flottante, il faut que l'un des opérandes soit un flottant : une manière de forcer cela est d'écrire $1.0 * n / p$.

$$3. S_3(n) = \sum_{i=1}^n \sum_{j=1}^n (i + j)$$

$$4. S_4(n) = \sum_{i=1}^n \sum_{j=i}^n (i + j)$$

$$5. S_5(n) = \sum_{i=1}^n \sum_{j=1}^{i-1} (i + j)$$

On écrira ensuite une fonction `main` permettant d'obtenir l'affichage suivant :

```
$ ./sommes
S1(0) = 0
S2(0) = 0.000000
S3(0) = 0
S4(0) = 0
S5(0) = 0
S1(1) = 3
S2(1) = 1.000000
S3(1) = 2
S4(1) = 2
S5(1) = 0
S1(2) = 8
S2(2) = 1.250000
S3(2) = 12
S4(2) = 9
S5(2) = 3
S1(3) = 15
S2(3) = 1.361111
S3(3) = 36
S4(3) = 24
S5(3) = 12
```

Exercice XII.2

Écrire une fonction prenant en entrée un entier n , et renvoyant le plus petit $k \geq 0$ tel que $2^k \geq n$: on fera une version utilisant une boucle **while** et une utilisant une boucle **for**.

2 Tableaux statiques

On peut déclarer un tableau de la manière suivante :

```
int t[10]; // tableau de 10 entiers, non initialisé
double u[3] = {1.2, 3.4, 2.5} // tableau de 3 doubles, initialisé
int v[] = {1, 2, 7, 8} // la taille est déduite de l'initialisation
```

- Les éléments d'un tableau sont indicés à partir de 0, et on accède à un élément par `t[indice]`. Les éléments du tableau `u` défini ci-dessus sont donc `u[0]`, `u[1]` et `u[2]`.
- Si le tableau n'est pas initialisé explicitement (cas du tableau `t` ci-dessus) :
 - si c'est une variable globale, les éléments sont implicitement initialisés à zéro;
 - si c'est une variable locale, les éléments ne sont pas initialisés, et ont au départ une valeur quelconque.
- La taille d'un tel tableau doit **absolument être connue statiquement** (c'est-à-dire à la compilation). La taille doit être une *constante littérale* :

```
int t[10]; // OK

int n = 10;
int u[n]; // NON

const int p = 5;
int v[p]; / NON PLUS
```

Exercice XII.3

On part du squelette suivant avec un tableau défini comme une variable globale :

```
#include <stdio.h>
#include <stdlib.h>

const int taille = 10;
int tab[10];
```

Noter que si l'on veut modifier le code pour que le tableau soit de taille 10, il faut modifier à la fois la constante `taille` que l'on utilisera pour les boucles dans le programme et le 10 qui apparaît dans la définition de `tab` (ce n'est évidemment pas satisfaisant, et nous verrons ultérieurement comment améliorer cela)..

1. La fonction `int rand(void)`, déclarée dans `stdlib.h`, renvoie un entier positif aléatoire. Écrire une fonction `void remplire(void)` qui remplit le tableau global `tab` avec des entiers tirés aléatoirement.
2. Écrire une fonction `void affiche(void)` qui affiche le contenu du tableau `tab` (en séparant les entiers par une espace et en revenant à la ligne à la fin).
3. Écrire une fonction `int min(void)` qui renvoie le minimum du tableau `tab`.
4. Écrire une fonction `int indice_min(void)` qui renvoie l'indice de la première occurrence du minimum dans `tab`.
5. Écrire une fonction `void tri_insertion(void)` qui trie le tableau `tab` en utilisant l'algorithme du tri par insertion. On pourra se référer au TP 4 pour la présentation de cet algorithme de tri sur les tableaux.
6. Écrire un programme qui effectue les tâches suivantes :
 - initialiser le tableau `tab` avec des valeurs tirées au hasard;
 - l'afficher;

- calculer et afficher le minimum de ce tableau, ainsi que l'indice de sa première occurrence;
- trier ce tableau;
- afficher ce tableau trié.

3 Arguments en ligne de commande

Pour l'instant, notre fonction `main` avait systématiquement le prototype suivant :

```
int main(void)
```

Un autre prototype est possible pour cette fonction :

```
int main(int argc, char* argv[])
```

Le fonctionnement est essentiellement le même qu'en OCaml (disons plutôt qu'OCaml a repris le fonctionnement du C) :

- `argc` est le nombre d'arguments passés au programme, sachant que le premier argument est toujours la commande utilisée pour le lancer;
- `argv` est un tableau de chaînes de caractères, de longueur `argc`. Le premier élément de ce tableau contient le nom de la commande, les autres contiennent les différents arguments.

En C, une chaîne de caractères est simplement un tableau de caractères, avec un dernier caractère nul. Cependant, pour gérer les arguments en ligne de commande, il n'est pas forcément nécessaire de les voir comme telles : on dispose de fonctions pour convertir une chaîne de caractères en entier ou en flottant.

- La fonction `int atoi(char* s)` convertit la chaîne de caractères `s` en entier. Elle saute les éventuels caractères d'espacement trouvés en début de chaîne (et les erreurs ne sont essentiellement pas gérées, donc mieux vaut être sûr que la chaîne correspond bien à un entier!).
- La fonction `double atof(char* s)` fait le même travail, mais lit un nombre flottant, qu'elle renvoie sous la forme d'un `double`.

Exemple XII.4

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
    int n;
    float x;
    if (argc != 3){
        printf("Il faut donner un entier et un flottant\n");
        return 1;
    }
    n = atoi(argv[1]);
    x = atof(argv[2]);
    for (int i = 0; i < n; i = i + 1){
        printf("%f\n", x);
    }
    return 0;
}
```

Le programme ci-dessus attend deux arguments en ligne de commande :

- un entier `n`;

- un flottant x .

Elle affiche ensuite n fois le flottant x . Si le nombre d'arguments n'est pas exactement 2, elle affiche un message d'erreur et termine avec une valeur non nulle pour signaler le problème.

Exercice XII.5

Écrire un programme qui accepte deux arguments en ligne de commande, un flottant x et un entier n et renvoie la valeur de x^n en effectuant ce calcul par une exponentiation rapide. On écrira la version itérative pour se rafraîchir la mémoire.

UN PEU DE DESSIN

Dans ce TP, on compilera systématiquement avec les options `-Wall -Wextra -lm`.

I Image RGB

Une image RBG (*Red Green Blue*) de n lignes et p colonnes peut être vue comme une matrice dont les éléments sont des triplets (r, g, b) d'entiers. Le plus souvent, et c'est ce que nous ferons ici, les valeurs r , g et b varient entre 0 et 255 (ce qui permet de les coder sur un octet chacune). Dans ce cas :

- $(255, 0, 0)$ est un rouge primaire;
- $(0, 0, 255)$ est un bleu primaire;
- $(255, 255, 255)$ est le blanc pur;
- $(0, 0, 0)$ est le noir pur;
- $(50, 50, 50)$ est un gris assez sombre;
- $(255, 0, 255)$ donne du magenta.

La synthèse des couleurs est *additive* (comme des lumières colorées qui se superposent) et non *soustractive* (comme des peintures de couleurs différentes que l'on mélange).

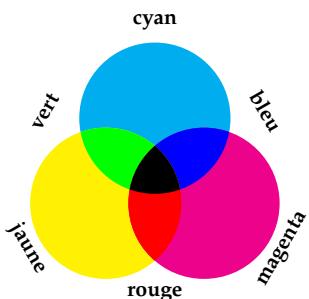


FIGURE XIII.1 – Synthèse soustractive.

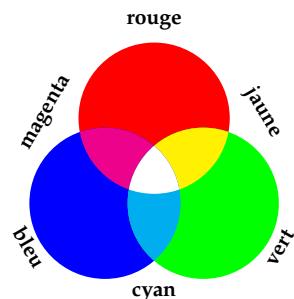


FIGURE XIII.2 – Syntèse additive.

La manière « usuelle » d'indexer les pixels d'une image est d'utiliser des coordonnées correspondant à celles dans une matrice, avec des lignes numérotées de 0 à $n - 1$ et des colonnes numérotées de 0 à $p - 1$.

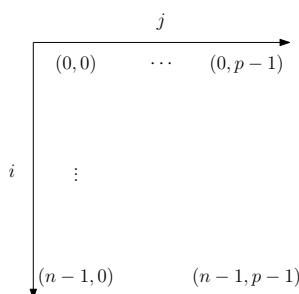


FIGURE XIII.3 – Coordonnées image

Nous utiliserons ces coordonnées pour stocker notre tableau de pixels, mais pour tracer des lignes, cercles et autres il sera plus agréable d'utiliser les coordonnées (x, y) usuelles (avec x qui correspond à un axe des abscisses de gauche à droite et y à un axe des ordonnées de bas en haut).

2 Types utilisés

On va représenter un pixel par un tableau de 3 entiers. Il y a des types spécifiques pour les entiers de différentes tailles en C, mais pour simplifier les valeurs seront juste des **int**. On pourrait donc définir la couleur rouge, par exemple, par :

```
int red[3] = {255, 0, 0}; // red est un tableau de trois int
```

Pour alléger les notations, on va donner un nom au type qui code une couleur `rgb` : en C, il faut pour cela utiliser le mot-clé **typedef**.

```
typedef int rgb[3]; // rgb est le *type* « tableau de trois int »
```

On peut ensuite définir des couleurs de manière plus simple :

```
rgb red = {255, 0, 0};
rgb green = {0, 255, 0};
rgb blue = {0, 0, 255};
rgb black = {0, 0, 0};
rgb white = {255, 255, 255};
```

Pour l'image, on va utiliser un tableau bidimensionnel dont les éléments seront de type `rgb`. Ce tableau bidimensionnel sera une variable globale, et sa taille sera connue statiquement (c'est-à-dire à la compilation). On pourrait procéder ainsi :

```
const int height = 600;
const int width = 800;

rgb canvas[height][width];
```

Cette approche est la seule à être vraiment au programme (pour un tableau statique). Elle permet d'utiliser les constantes `height` et `width` dans le code (pour des bornes de boucles par exemple), mais si l'on veut modifier la largeur de l'image il faut bien penser à le faire à deux endroits : la définition de `height` et celle de `canvas`. Nous allons utiliser la « bonne » manière de procéder, qui est d'utiliser la *directive de pré-processeur* **#define**.

```
#define HEIGHT 600
#define WIDTH 800

rgb canvas[HEIGHT][WIDTH];
```

Essentiellement, l'effet de **#define HEIGHT 600** est de remplacer textuellement toutes les occurrences de `HEIGHT` dans le code par `600` : on parle de *macro*. C'est une pratique très courante en C, qui permet de pallier partiellement certaines faiblesses du langage, mais c'est assez piégeux (et ce n'est pas au programme). Nous nous limiterons strictement à l'utilisation faite ici : définir des constantes pour pouvoir les utiliser comme tailles de tableaux statiques.

3 Format d'image utilisé

Comme il n'y a pas beaucoup de bibliothèques installées (pour l'instant) sur les ordinateurs du lycée, nous n'allons pas afficher « en direct » les images que nous allons créer : nous allons sauvegarder l'image dans un fichier et utiliser ensuite une application dédiée au visionnage d'images.

Nous allons utiliser le format le plus simple du monde : le format PPM. Un fichier PPM est un fichier texte obéissant à des règles très simples.

- Le fichier commence par une ligne réduite à "P3" (c'est ce qu'on appelle un *magic number* qui permet d'indiquer le format du fichier).

- La ligne suivante contient deux entiers séparés par une espace : le premier indique la largeur de l'image (en nombre de pixels), le second la hauteur (attention, ce n'est pas forcément le plus naturel).
- La ligne suivante contient un entier indiquant la valeur maximale des composantes RGB. Si cette valeur est 100 par exemple, alors un pixel blanc sera codé par (100, 100, 100). Nous utiliserons 255 comme valeur maximale, ce qui est le plus standard.
- Ensuite, on donne les valeurs R, G et B de chaque pixel : d'abord la première ligne (celle du haut) de gauche à droite, puis la deuxième ligne... Chaque valeur doit être séparée de la précédente et de la suivante par un caractère d'espacement (espace, retour à la ligne...). En pratique, on mettra les trois valeurs correspondant à un pixel sur une ligne, séparées par des espaces, puis on reviendra à la ligne pour le pixel suivant.

```
P3
3 2 # 3 colonnes, 2 lignes
255
255 0 0 # pixel rouge en haut à gauche
255 0 0
0 0 255
0 255 0 # pixel vert en bas à gauche
255 255 255
150 150 150
```

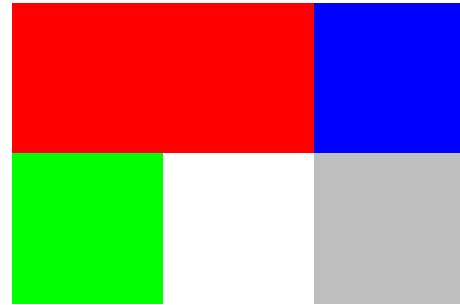


FIGURE XIII.4 – Exemple de fichier au format PPM.

4 Création du fichier

Dans le fichier `squelette.c` fourni, vous trouverez quelques définitions de variables globales et une fonction

```
void put_pixel(int x, int y, rgb c)
```

Cette fonction agit en modifiant le tableau global `canvas`, et elle accepte des coordonnées « mathématiques » usuelles : l'origine est en bas à gauche, l'axe des `x` va de gauche à droit et l'axe des `y` de bas en haut). Pour écrire des valeurs dans `canvas`, on passera systématiquement par cette fonction.

Remarque

Si les coordonnées passées à la fonction `put_pixel` sont « hors cadre », elle ne fait rien. Cela simplifiera les choses par la suite (si par exemple on veut tracer un disque dont seule une partie se trouve dans l'image).

Exercice XIII.1 – Fonction `print_canvas`

p. 454

La fonction `print_canvas` doit afficher sur la sortie standard (à l'aide de la fonction `printf`) le contenu d'un fichier PPM correspondant au tableau `canvas`. Attention, la case `canvas[i][j]` contient le pixel de coordonnées (`i, j`) dans le système décrit à la figure XIII.3.

```
void print_canvas(void)
```

Avec les instructions présentes dans le `main` fourni, on devrait à la compilation obtenir exactement l'affichage (texte) de la figure XIII.4 (sans les commentaires, bien sûr).

Exercice XIII.2 – Redirection de sortie

p. 454

Pour pouvoir *afficher* l'image (graphiquement), nous devons la sauvegarder dans un fichier. On pourrait utiliser une variante de la fonction `printf` permettant d'écrire dans un fichier, mais nous allons choisir une autre solution, qui utilise les fonctionnalités du shell.

Si l'on exécute une commande que l'on fait suivre par `> nom`, alors la *sortie standard* de la

commande est *redirigée* vers le fichier nom. Cela signifie que tout ce qui aurait normalement été affiché lors de l'exécution de la commande est à la place écrit dans le fichier nom. Si ce fichier n'existe pas il est créé ; s'il existe déjà, il est entièrement écrasé.

La commande echo permet d'afficher quelque chose directement depuis le shell. Par exemple, echo hello affichera hello à l'écran :

```
$ echo hello
hello
```

En redirigeant la sortie, on peut écrire dans un fichier :

```
$ echo Hello, World! > hello.txt
```

Cette commande n'affiche rien, mais on a bien créé le fichier. On peut l'éditer avec un éditeur de texte, ou simplement afficher son contenu à l'aide de la commande cat :

```
$ cat hello.txt
Hello, World!
```

1. En utilisant la même technique de redirection, créer un fichier test.ppm en exécutant votre programme compilé.
2. Ouvrir ce fichier avec un programme approprié (je ne sais plus ce qui est installé au lycée). Il faudra sans doute zoomer pour voir quelque chose (l'image fait 3 pixels de large).
3. Modifier le programme pour créer une image 600×400 entièrement rouge, et vérifier que vous arrivez bien à l'afficher.

5 Primitives simples

Dans toute la suite du sujet, on fera bien attention à traiter correctement tous les cas, sans supposer abusivement que tel point est à gauche de tel autre, ou qu'ils ne sont pas l'un au dessus de l'autre, ou que sais-je encore... D'un autre côté, on essaiera d'écrire du code raisonnablement concis.

Exercice XIII.3

p. 455

1. Écrire une fonction permettant de tracer une ligne horizontale. Cette fonction prendra en entrée une ordonnée, deux abscisses et une couleur.

```
void draw_h_line(int y, int x0, int x1, rgb c)
```

2. Écrire de même une fonction pour tracer une ligne verticale.

```
void draw_v_line(int x, int y0, int y1, rgb c)
```

Exercice XIII.4

p. 455

1. Écrire une fonction permettant de tracer un rectangle aligné avec les axes (deux côtés verticaux et deux côtés horizontaux). Le rectangle sera spécifié par la donnée de deux coins opposés (ainsi que par une couleur).

```
void draw_rectangle(int x0, int y0, int x1, int y1, rgb c)
```

2. Écrire une fonction permettant de « remplir » un rectangle du même type qu'à la question

précédente.

```
void fill_rectangle(int x0, int y0, int x1 int y1, rgb c)
```

Exercice XIII.5

p. 456

Écrire une fonction permettant de remplir un disque spécifié par son centre et son rayon.

```
void fill_disk(int xc, int yc, int radius, rgb c)
```

6 Mélange de couleurs

Si on dispose de deux couleurs $c = (r, g, b)$ et $c' = (r', g', b')$, on peut pour $\alpha, \beta \in \mathbb{R}$ définir :

$$\text{mix}(c, c', \alpha, \beta) = \begin{cases} r_m &= \text{clamp}(\alpha r + \beta r') \\ g_m &= \text{clamp}(\alpha g + \beta g') \\ b_m &= \text{clamp}(\alpha b + \beta b') \end{cases}$$

La fonction `clamp` étant définie par :

$$\text{clamp}(x) = \begin{cases} 0 & \text{si } x < 0 \\ 255 & \text{si } x > 255 \\ x & \text{sinon} \end{cases}$$

Exercice XIII.6

p. 456

1. Écrire la fonction `clamp` :

```
int clamp(double x)
```

2. Pour écrire la fonction `mix`, on est confronté à un problème. En effet, on voudrait que cette fonction renvoie un tableau, ce qui n'est pas directement possible. Il y a deux manières de procéder, dont l'une fait appel à l'allocation dynamique de mémoire que nous n'avons pas encore abordée. On va donc utiliser l'autre et écrire une fonction ayant le prototype suivant :

```
void mix(rgb c0, rgb c1, double alpha, double beta, rgb result)
```

On fournira donc à `mix` un tableau `result` qu'elle modifiera pour y mettre son résultat.

Exercice XIII.7

p. 457

1. Écrire une fonction ayant le prototype suivant :

```
void draw_h_gradient(int y, int x0, int x1, rgb c0, rgb c1)
```

Cette fonction tracera une ligne horizontale, avec le point d'abscisse x_0 de couleur c_0 , le point d'abscisse x_1 de couleur c_1 , et les points intermédiaires coloriés grâce à une interpolation linéaire de ces deux couleurs.

2. Écrire de même une fonction :

```
void fill_disk_gradient(int xc, int yc, int radius,
                        rgb c_center, rgb c_edge)
```

Cette fois, l'interpolation de couleur se fera suivant la distance entre le point et le centre : couleur `c_center` si elle vaut 0, couleur `c_edge` si elle vaut `radius`, et une interpolation linéaire entre.

Exercice XIII.8

p. 458

- Écrire une fonction `get_pixel` qui remplit le tableau `result` avec le contenu de la case de `canvas` correspondant aux coordonnées (x, y) (on s'inspirera de `put_pixel`, et on ne fera rien dans le cas où les coordonnées données sont hors-cadre).

```
void get_pixel(int x, int y, rgb result)
```

- Écrire une fonction `mix_pixel` qui remplace le pixel de coordonnées (x, y) par le mélange de sa valeur actuelle (avec coefficient β) et de la couleur `c` fournie (avec coefficient α).

```
void mix_pixel(int x, int y, double alpha, double beta, rgb c)
```

- Écrire une fonction `add_disk` permettant d'obtenir facilement un dessin ressemblant à la figure XIII.2.

```
void add_disk(int xc, int yc, int radius, rgb c)
```

7 Tracé de lignes par l'algorithme de Bresenham

Pour les algorithmes qui suivent, il faut clarifier la relation entre les coordonnées entières x et y d'un pixel et les coordonnées mathématiques dans le plan. **On considère que c'est le centre du pixel qui a pour coordonnées (x, y) dans le plan.**

7.1 Segment oblique

Pour l'instant, on ne sait tracer que des lignes verticales ou horizontales, ce qui est un peu limité. On souhaite à présent tracer la ligne reliant un point (x_0, y_0) au point (x_1, y_1) , sans hypothèses particulières sur ces points.

Exercice XIII.9

p. 459

Pour commencer, on traite le cas où la droite possède une pente comprise entre 0 et 1 et où $x_0 \leq x_1$. Il faut alors sélectionner exactement un pixel dans chaque colonne de x_0 à x_1 :

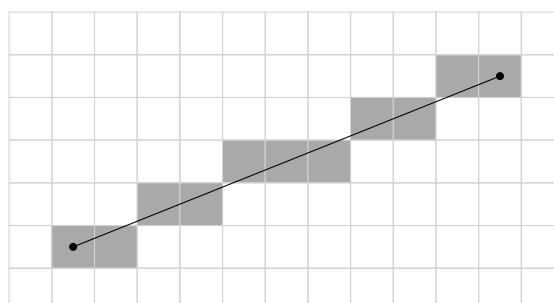


FIGURE XIII.5 – Résultat de l'algorithme de Bresenham

Si l'on suppose que l'on traite les pixels de gauche à droite, il y a donc deux pixels candidats à chaque étape :

- celui situé immédiatement à droite du pixel actuel, de coordonnées $(x + 1, y)$;
- celui situé en diagonale en haut à droite, de coordonnées $(x + 1, y + 1)$.

On veut choisir celui des pixels situé le plus près de la ligne idéale, c'est-à-dire celui dont le centre est situé le plus près de cette ligne. Une première option est de mettre l'équation de la droite sous la forme $y = y_0 + m(x - x_0)$. Comme on connaît systématiquement la valeur de x (qui est incrémenté à chaque étape), on peut calculer y et l'arrondir à l'entier le plus proche pour choisir quel pixel « allumer ». Pour réaliser un tel arrondi en C, on pourra utiliser la fonction **double round(double)**, et transformer son résultat en un entier :

```
double x = 3.76;
double x_rounded = round(x) // x_rounded vaut 4.0
int n = x_rounded           // n vaut 4

// Ou en une seule étape :
int n = round(3.76)         // n vaut 4
```

1. Écrire une fonction utilisant cet algorithme pour tracer une ligne de pente comprise entre 0 et 1.

```
void draw_line(int x0, int y0, int x1, int y1, rgb c)
```

2. Compléter cette fonction pour qu'elle traite correctement tous les autres cas. On essaiera de ne pas se perdre dans le code !

Exercice XIII.10

p. 459

Écrire une fonction ayant le prototype suivant :

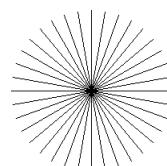
```
void draw_spokes(int xc, int yc, int radius, int nb_spokes, rgb c)
```

Cette fonction tracera `nb_spokes` rayons du cercle de centre (x_c, y_c) et de rayon `radius`. Le premier de ces rayons sera horizontal et les autres seront régulièrement espacés en terme d'angle.

Remarque

L'inclusion de l'entête `math.h` permet d'accéder aux fonctions **double cos(double)** et **double sin(double)**.

```
...
#define HEIGHT 201
#define WIDTH 201
...
int main(void){
    fill_rectangle(0, 0, 200, 200, white);
    draw_spokes(100, 100, 100, 36, black);
    return 0;
}
```



Exercice XIII.II

p. 460

Le « vrai » algorithme de Bresenham ne travaille qu’avec des entiers (à l’époque où il a été inventé, les calculs sur les flottants étaient extrêmement coûteux) et minimise les calculs à effectuer. Il se base sur l’observation suivante (toujours dans le cas d’une pente comprise entre 0 et 1) : pour choisir entre le point $(x + 1, y)$ et le point $(x + 1, y + 1)$, il suffit de déterminer si le point $(x + 1, y + 1/2)$ est situé au-dessus ou en dessous de la droite.

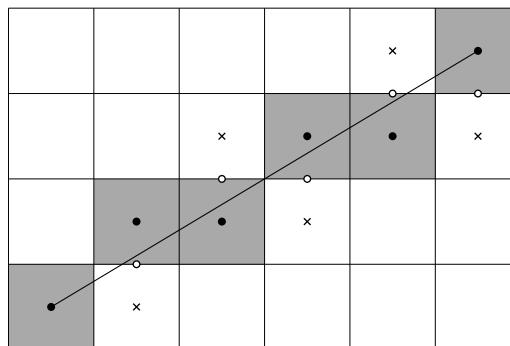


FIGURE XIII.6 – À chaque étape, on détermine si le point marqué par un cercle est au-dessus ou en dessous de la droite. On élimine un candidat (marqué par une croix) et l’on choisit l’autre (marqué par un disque).

- On note $\Delta_x = x_1 - x_0$ et $\Delta_y = y_1 - y_0$. On se place pour l’instant dans le cas où $0 \leq \Delta_y \leq \Delta_x$, et l’on définit :

$$f(x, y) = 2(\Delta_x y - \Delta_y x)$$

$$D(x, y) = f(x + 1, y + 1/2) - f(x_0, y_0)$$

- Montrer que le point $(x + 1, y + 1/2)$ est au-dessus de la droite (au sens large) si et seulement si $D(x, y) \geq 0$.
- Calculer $D(x_0, y_0)$.
- Déterminer $D(x + 1, y)$ et $D(x + 1, y + 1)$ en fonction de $D(x, y)$.
- En déduire une fonction

```
void bresenham_low(int x0, int y0, int x1, int y1, rgb c)
```

Cette fonction aura pour précondition $0 \leq \Delta_y \leq \Delta_x$, travaillera uniquement avec des entiers, et n’effectuera que des additions, soustractions et multiplications par 2.

- Modifier la fonction `bresenham_low` pour que sa précondition devienne $|\Delta_y| \leq \Delta_x$.
- Écrire une fonction `bresenham_high` ayant comme précondition $|\Delta_x| \leq \Delta_y$.
- Écrire finalement une fonction

```
void bresenham(int x0, int y0, int x1, int y1, rgb c)
```

permettant de traiter correctement tous les cas.

7.2 Cercle

Exercice XIII.12

p. 462

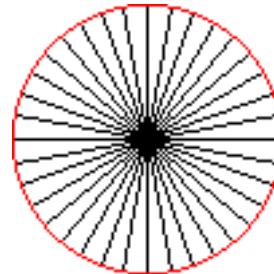
On peut adapter l'algorithme de Bresenham pour tracer un cercle (dont le centre est à coordonnées entières et le rayon entier). L'idée est la suivante :

- on part du point le plus à droite, que l'on allume ;
- on considère le point situé juste au-dessus, et celui situé en diagonale en haut à gauche ;
- pour chacun de ces points, on calcule le carré de la distance au centre ;
- on choisit le point pour lequel cette valeur est la plus proche du rayon au carré, et l'on recommence le même processus à partir de ce point ;
- on s'arrête au bon moment, après avoir tracé une partie du cercle ;
- les autres parties du cercle sont tracées par symétrie.

1. Quand faut-il arrêter le processus ?
2. Écrire une fonction

```
void draw_circle(int xc, int yc, int radius, rgb c)
```

```
...
#define HEIGHT 101
#define WIDTH 101
...
int main(void){
    fill_rectangle(0, 0, 100, 100, white);
    draw_spokes(50, 50, 50, 36, black);
    draw_circle(50, 50, 50, red);
    return 0;
}
```



Remarque

Tout comme pour une droite, le vrai algorithme évite de recalculer entièrement les distances au centre à chaque étape, et utilise le point situé au milieu du segment formé par les candidats pour trancher. Il n'est pas très difficile de trouver les relations rendant cela possible.

Solutions

Correction de l'exercice XIII.1 page 447

```
void print_canvas(void){
    printf("P3\n");
    printf("%d %d\n", WIDTH, HEIGHT);
    printf("255\n");
    for (int i = 0; i < HEIGHT; i++){
        for (int j = 0; j < WIDTH; j++){
            for (int k = 0; k < 3; k++){
                printf("%d ", canvas[i][j][k]);
            }
            printf("\n");
        }
    }
}
```

Correction de l'exercice XIII.2 page 447

- Il suffit de compiler le programme puis de l'exécuter avec une redirection de sortie :

```
$ gcc -Wall -Wextra -lm -o dessin dessin.c
$ ./dessin > test.ppm
```

-
-
- Modifications minimales à apporter :

```
...
#define HEIGHT 400
#define WIDTH 600
...
int main(void){
    for (int x = 0; x < WIDTH; x++){
        for (int y = 0; y < HEIGHT; y++){
            put_pixel(x, y, red);
        }
    }
    return 0;
}
```

Correction de l'exercice XIII.3 page 448

1. Attention, rien ne dit que $x_0 \leq x_1$!

```
void draw_h_line(int y, int x0, int x1, rgb c){
    if (x0 > x1) {
        draw_h_line(y, x1, x0, c);
        return;
    }
    for (int x = x0; x <= x1; x++){
        put_pixel(x, y, c);
    }
}
```

2. Rien de bien différent.

```
void draw_v_line(int x, int y_start, int y_end, rgb c){
    if (y_start > y_end){
        draw_v_line(x, y_end, y_start, c);
        return;
    }
    for (int y = y_start; y <= y_end; y++){
        put_pixel(x, y, c);
    }
}
```

Correction de l'exercice XIII.4 page 448

1. On trace les 4 côtés grâce aux fonctions précédentes. Les pixels des coins sont dessinés deux fois chacun, mais ce n'est pas gênant ici (ça pourrait être le cas dans d'autres contextes).

```
void draw_rectangle(int x0, int y0, int x1, int y1, rgb c){
    draw_h_line(y0, x0, x1, c);
    draw_h_line(y1, x0, x1, c);
    draw_v_line(x0, y0, y1, c);
    draw_v_line(x1, y0, y1, c);
}
```

2. C'est encore plus simple que de tracer le contour.

```
void fill_rectangle(int x0, int y0, int x1, int y1, rgb c){
    if (y0 > y1) {
        fill_rectangle(x1, y1, x0, y0, c);
    } else {
        for (int y = y0; y <= y1; y++){
            draw_h_line(y, x0, x1, c);
        }
    }
}
```

Correction de l'exercice XIII.5 page 449

On considère tous les points de la *boîte englobante* du disque (on est en deux dimensions, donc cette boîte est un rectangle).

```
int distance_squared(int x0, int y0, int x1, int y1){
    int dx = x1 - x0;
    int dy = y1 - y0;
    return dx * dx + dy * dy;
}
```

```
void fill_disk(int xc, int yc, int r, rgb c){
    int xmin = xc - r;
    int xmax = xc + r;
    int ymin = yc - r;
    int ymax = yc + r;
    for (int x = xmin; x <= xmax; x++){
        for (int y = ymin; y <= ymax; y++){
            if (distance_squared(xc, yc, x, y) <= r * r){
                put_pixel(x, y, c);
            }
        }
    }
}
```

Correction de l'exercice XIII.6 page 449

1. Aucun problème.

```
int clamp(double x){
    if (x < 0.) return 0;
    if (x > 255.) return 255;
    return (int)x;
}
```

2. Pas de difficulté non plus, une fois qu'on a compris le principe de « renvoyer un résultat » en modifiant le tableau passé en paramètre.

```
void mix(rgb c0, rgb c1, double alpha, double beta, rgb result){
    for (int k = 0; k < 3; k++){
        result[k] = clamp(alpha * c0[k] + beta * c1[k]);
    }
}
```

Correction de l'exercice XIII.7 page 449

1. Attention encore à traiter correctement le cas $x_0 > x_1$: on inverse les points *et les couleurs*.

```
void draw_h_gradient(int y, int x0, int x1, rgb c0, rgb c1){
    if (x0 > x1){
        draw_h_gradient(y, x1, x0, c1, c0);
        return;
    }
    for (int x = x0; x <= x1; x++){
        double ratio = (double)(x - x0) / (double)(x1 - x0);
        rgb c; // non initialisé pour l'instant
        mix(c0, c1, 1.0 - ratio, ratio, c);
        // c a désormais un contenu bien défini
        put_pixel(x, y, c);
    }
}
```

2. L'énoncé demandant une interpolation linéaire sur la longueur, on est obligé d'utiliser une racine carrée. Il serait aussi possible de faire le ratio des *carrés* des distances, mais le résultat n'est pas le même.

```
void fill_disk_gradient(int xc, int yc, int r, rgb c_center, rgb c_edge){
    int xmin = xc - r;
    int xmax = xc + r;
    int ymin = yc - r;
    int ymax = yc + r;
    for (int x = xmin; x <= xmax; x++){
        for (int y = ymin; y <= ymax; y++){
            double ratio = sqrt(distance_squared(xc, yc, x, y)) / r;
            if (ratio <= 1.0){
                rgb c;
                mix(c_center, c_edge, 1.0 - ratio, ratio, c);
                put_pixel(x, y, c);
            }
        }
    }
}
```

Correction de l'exercice XIII.8 page 450

1.

```
void get_pixel(int x, int y, rgb result){
    int i = HEIGHT - 1 - y;
    int j = x;
    if (i < 0 || i >= HEIGHT || j < 0 || j > WIDTH) return;
    for (int k = 0; k < 3; k++){
        result[k] = canvas[i][j][k];
    }
}
```

2.

```
void mix_pixel(int x, int y, double alpha, double beta, rgb c){
    rgb c0;
    get_pixel(x, y, c0);
    mix(c, c0, alpha, beta, c0);
    put_pixel(x, y, c0);
}
```

3. On utilise deux coefficients égaux à 1 pour obtenir une simple « addition » des couleurs.

```
void add_disk(int xc, int yc, int r, rgb c){
    int xmin = xc - r;
    int xmax = xc + r;
    int ymin = yc - r;
    int ymax = yc + r;
    for (int x = xmin; x <= xmax; x++){
        for (int y = ymin; y <= ymax; y++){
            if (distance_squared(xc, yc, x, y) <= r * r){
                mix_pixel(x, y, 1., 1., c);
            }
        }
    }
}
```

Correction de l'exercice XIII.9 page 450

On donne directement la fonction complète :

```
void draw_line(int x0, int y0, int x1, int y1, rgb c){
    if (x0 == x1){
        // pente non définie
        draw_v_line(x0, y0, y1, c);
        return;
    }
    bool steep = abs(y1 - y0) > abs(x1 - x0);
    double m = (double)(y1 - y0) / (x1 - x0);
    if (!steep){
        // valeur absolue de la pente inférieure à 1
        if (x0 > x1){
            draw_line(x1, y1, x0, y0, c);
            return;
        }
        for (int x = x0; x <= x1; x++){
            int y = round(y0 + m * (x - x0));
            put_pixel(x, y, c);
        }
    } else {
        // valeur absolue de la pente supérieure à 1
        if (y0 > y1){
            draw_line(x1, y1, x0, y0, c);
            return;
        }
        for (int y = y0; y <= y1; y++){
            int x = round(x0 + (y - y0) / m);
            put_pixel(x, y, c);
        }
    }
}
```

Correction de l'exercice XIII.10 page 451

Pas de difficulté particulière.

```
void draw_spokes(int xc, int yc, int r, int nb_spokes, rgb c_center){
    double pi = 3.1416;
    double angle = 2 * pi / nb_spokes;
    for (int i = 0; i < nb_spokes; i++){
        double theta = i * angle;
        int x = round(xc + r * cos(theta));
        int y = round(yc + r * sin(theta));
        draw_line(xc, yc, x, y, c_center);
    }
}
```

Correction de l'exercice XIII.II page 452

1. a. L'équation de la droite est $y = \frac{\Delta_y}{\Delta_x}(x - x_0) + y_0$. En notant $M(x + 1, y + 1/2)$, on a :

$$\begin{aligned} M \text{ au-dessus} &\iff y + 1/2 \geq \frac{\Delta_y}{\Delta_x}(x + 1 - x_0) + y_0 \\ &\iff 2\Delta_x(y + 1/2) \geq 2\Delta_y(x + 1) - 2\Delta_y x_0 + 2\Delta_x y_0 \quad \text{puisque } \Delta_x > 0 \\ &\iff 2(\Delta_x(y + 1/2) - \Delta_y(x + 1)) - 2(\Delta_x y_0 - \Delta_y x_0) \geq 0 \\ &\iff f(x + 1, y + 1/2) - f(x_0, y_0) \geq 0 \end{aligned}$$

b. On a $D(x_0, y_0) = \Delta_x - 2\Delta_y$.

c. On obtient $D(x + 1, y) = D(x, y) - 2\Delta_y$ et $D(x + 1, y + 1) = D(x, y) + 2\Delta_x - 2\Delta_y$.

d. Première version de la fonction bresenham_low :

```
void bresenham_low(int x0, int y0, int x1, int y1, rgb c){
    // Preconditions:
    // 0 <= y1 - y0 <= x1 - x0 et x0 <= x1
    int dx = x1 - x0;
    int dy = y1 - y0;
    int err = dx - 2 * dy;
    int y = y0;
    for (int x = x0; x <= x1; x++){
        put_pixel(x, y, c);
        if (err < 0){
            y++;
            err = err + 2 * dx - 2 * dy;
        } else {
            err = err - 2 * dy;
        }
    }
}
```

2. Il faut traiter le cas où la pente est entre -1 et 0 . Le plus agréable est d'unifier les deux cas, en prenant $dy = |\Delta_y|$ et en utilisant un incrément de -1 pour y si $\Delta_y < 0$.

```

void bresenham_low(int x0, int y0, int x1, int y1, rgb c){
    // Preconditions:
    // abs(y1 - y0) <= abs(x1 - x0) et x0 <= x1
    int dx = x1 - x0;
    int dy = y1 - y0;
    int y_increment = 1;
    if (dy < 0){
        y_increment = -1;
        dy = -dy;
    }
    int err = dx - 2 * dy;
    int y = y0;
    for (int x = x0; x <= x1; x++){
        put_pixel(x, y, c);
        if (err < 0){
            y = y + y_increment;
            err = err + 2 * dx - 2 * dy;
        } else {
            err = err - 2 * dy;
        }
    }
}

```

3. On obtient bresenham_high en échangeant les rôles de x et y dans bresenham_low.

```

void bresenham_high(int x0, int y0, int x1, int y1, rgb c){
    // Preconditions:
    // abs(x1 - x0) <= abs(y1 - y0)
    // y0 <= y1
    int dx = x1 - x0;
    int dy = y1 - y0;
    int x_increment = 1;
    if (dx < 0){
        x_increment = -1;
        dx = -dx;
    }
    int err = dy - 2 * dx;
    int x = x0;
    for (int y = y0; y <= y1; y++){
        put_pixel(x, y, c);
        if (err < 0){
            x = x + x_increment;
            err = err + 2 * dy - 2 * dx;
        } else {
            err = err - 2 * dx;
        }
    }
}

```

4. Il reste deux cas non traités, mais il suffit alors d'échanger les deux extrémités :

```

void bresenham(int x0, int y0, int x1, int y1, rgb c){
    int dx = x1 - x0;
    int dy = y1 - y0;
    if (abs(dx) > abs(dy)){
        if (dx >= 0) bresenham_low(x0, y0, x1, y1, c);
        else bresenham_low(x1, y1, x0, y0, c);
    } else {
        if (dy >= 0) bresenham_high(x0, y0, x1, y1, c);
        else bresenham_high(x1, y1, x0, y0, c);
    }
}

```

Correction de l'exercice XIII.12 page 453

1. Le processus décrit incrémenté y de 1 à chaque étape. Ce n'est correct que sur le premier octant du cercle (pour un angle inférieur ou égal à $\pi/4$), c'est-à-dire tant que la valeur absolue de la pente de la tangente est supérieure à 1.
2. On commence par écrire une fonction auxiliaire qui affiche un pixel et ses symétriques dans les 8 autres octants.

```

void _draw_circle_points(int xc, int yc, int dx, int dy, rgb c){
    put_pixel(xc + dx, yc + dy, c);
    put_pixel(xc - dx, yc + dy, c);
    put_pixel(xc + dx, yc - dy, c);
    put_pixel(xc - dx, yc - dy, c);
    put_pixel(xc + dy, yc + dx, c);
    put_pixel(xc - dy, yc + dx, c);
    put_pixel(xc + dy, yc - dx, c);
    put_pixel(xc - dy, yc - dx, c);
}

```

Ensuite, la fonction principale ne pose pas de problème particulier.

```

void draw_circle(int xc, int yc, int r, rgb c){
    int dx = r;
    int dy = 0;
    _draw_circle_points(xc, yc, dx, dy, c);
    while (dx > dy){
        dy++;
        int d2_above = dx * dx + dy * dy;
        int d2_diag = (dx - 1) * (dx - 1) + dy * dy;
        int err_above = abs(r * r - d2_above);
        int err_diag = abs(r * r - d2_diag);
        if (err_diag < err_above){
            dx--;
        }
        _draw_circle_points(xc, yc, dx, dy, c);
    }
}

```

POINTEURS

I Manipulation de pointeurs

Exercice XIV.1

On considère le code suivant :

```
#include <stdio.h>

int n = 3;
int p;

int f(int n){
    int* p = &n;
    int x = n + *p;
    return x + 1;
}

int g(int x, int y){
    int z = f(x);
    return z + f(y);
}

int main(void){
    p = 4;
    int result = g(n, p);
    printf("result = %d\n", result);
    return 0;
}
```

Aller sur le site <https://pythontutor.com/c.html> (vous pouvez taper *C tutor* sur Google, ça devrait être le premier résultat) et visualiser l'exécution pas à pas du programme. Essayer de bien comprendre ce qui se passe.

Exercice XIV.2

Dans chacun des cas suivants :

- déterminer si le programme est « faux » (lecture d'une variable non initialisée, déréférencement d'un pointeur invalide, erreur de type...);
- écrire sur papier (sans exécuter le programme) ce qu'on obtiendrait sur *C Tutor* (évolution du schéma mémoire et affichage produit);
- copier le code sur *C Tutor* et vérifier.

1.

```
#include <stdio.h>
#include <stdbool.h>

void print_bool(bool b){
    if (b){
        printf("true\n");
    } else {
        printf("false\n");
    }
}

int main(void){
    double pi = 3.14;
    double e;
    double* p = NULL;
    p = &e;
    *p = pi;
    print_bool(e == pi);
    pi = 4.5;
    print_bool(e == pi);
}
```

2.

```
#include <stdio.h>
#include <stdbool.h>

int x = 7;
int y = 12;
int* p;

int f(int x){
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    printf("*p = %d\n", *p);
    int y = 1;
    printf("y = %d\n", y);
    x = x + y;
    return x;
}

int main(void){
    int z;
    p = &x;
    z = f(x + 1);
    printf("z = %d\n", z);
    return 0;
}
```

Exercice XIV.3

Écrire une fonction de prototype :

```
void extrema(int t[], int taille, int* min, int* max)
```

Les préconditions sont :

- la longueur de `t` vaut `taille`, et elle est strictement positive;
- `min` et `max` sont des pointeurs valides.

La fonction affectera le minimum de `t` à l'objet pointé par `min`, et le maximum à celui pointé par `max`.

Exercice XIV.4

On considère la fonction suivante :

```
void mystere(int* x, int* y){
    *x = *x - *y;
    *y = *x + *y;
    *x = *y - *x;
}
```

1. Quel affichage obtiendrait-on avec le code suivant?

```
int x = 3;
int y = 4;
mystere(&x, &y);
printf("x = %d\n", x);
printf("y = %d\n", y);
```

2. De manière générale, quel est l'effet de la fonction `mystere`? On justifiera.
3. Quel affichage obtiendrait-on avec le code suivant?

```
int x2 = 3;
int y2 = 3;
mystere(&x2, &y2);
printf("x2 = %d\n", x2);
printf("y2 = %d\n", y2);
```

4. Quel affichage obtiendrait-on avec le code suivant?

```
int x3 = 3;
mystere(&x3, &x3);
printf("x3 = %d\n", x3);
```

Quel est le problème dans la démonstration faite plus haut?

2 Fonction `scanf`

La fonction (ou *les fonctions* en fait, il en existe toute une série) `printf` permet de *lire* des données sous un format spécifié, tout comme la fonction `printf` permet d'*écrire* des données formatées.

Le principe de base est assez simple : on passe comme premier argument une chaîne de format, contenant un certain nombre de « champs » (`%d`, `%f` et autres), et ensuite une série de pointeurs. Il faut

un pointeur par champ, et il faut que les types des pointeurs correspondent aux types des champs, dans l'ordre d'apparition dans la chaîne de format.

Exercice XIV.5

Le programme suivant permet de lire sur l'entrée standard :

- un entier *n*;
- puis ensuite, *n* fois, un entier *p* et un flottant *x*.

Après chaque lecture de (*p*, *x*), le programme affiche la somme *p* + *x* sur la sortie standard.

```
#include <stdio.h>

int main(void){
    int n = 0;
    float x = 0.;
    int p = 0;
    scanf("%d", &n);
    for (int i = 0; i < n; i++){
        scanf("%d %f", &p, &x);
        printf("%f\n", p + x);
    }
    return 0;
}
```

1. Compiler et exécuter ce programme.
2. Essayer quelques variations, en particulier sur le nombre d'espaces entre les différentes entrées, les retours à la ligne...
3. Essayer également d'exécuter ce programme en redirigeant sa sortie standard vers un fichier (*./programme >sortie.txt*). Cela permet de mieux faire la différence entre les entrées et les sorties.
4. Il est aussi possible de *rediriger l'entrée standard* d'un programme. Cela signifie que les valeurs seront lues dans le fichier spécifié au lieu de devoir être rentrées au clavier en direct. Créer un fichier *entree.txt* contenant les lignes suivantes :

```
3
1 3.14
```

```
7 1 2 1.4
```

Exécuter ensuite la ligne de commande suivante, et observer la sortie obtenue.

```
$ ./programme <entree.txt
```

5. Rien n'empêche de rediriger l'entrée et la sortie (c'est même assez courant). Tester par exemple la commande suivante :

```
$ ./programme <entree.txt >sortie.txt
```

Il y a un certain nombre de subtilités, qu'il n'est pas du tout indispensable de maîtriser :

- les champs numériques (comme `%d` qui lit un **int**, `%f` qui lit un **float**, `%lf` qui lit un **double**...) « mangent » les caractères d'espacement (espaces, retours à la ligne...). Par exemple, si l'entrée est "`\n\n 1234 \n3.14`" et que deux variables **int** *x* et **float** *y* ont été préalablement définies :
 - après un appel `scanf(" %d", &x)`, *x* vaut 1234 et l'entrée restante est "`\n3.14`";
 - si l'on fait ensuite un appel `scanf(" %f", &y)`, *y* vaut 3.14 et l'entrée a été entièrement consommée;

- on aurait pu remplacer ces deux appels successifs par un seul appel `scanf("%d %f", &x, &y)`, qui aurait eu le même effet.
- Un caractère d'espacement dans la chaîne de format est interprété comme un nombre quelconque de caractères d'espacement quelconques.
- `scanf` renvoie un entier pour indiquer ce qui s'est passé.
 - Si tout va bien, cet entier sera égal au nombre de paramètres supplémentaires passés (en plus de la chaîne de format). Un appel `scanf("%d %f", &x, &y)`, par exemple, est censé renvoyer 2.
 - Sinon, cet entier sera égal au nombre de champs pour lesquels on a réussi à extraire une valeur. Si l'entrée valait "1234toto 3.14", l'appel `scanf("%d %f", &x, &y)` affectera 1234 à `x` et renverra 1, et l'entrée restante sera "toto 3.14".
 - L'entier renvoyé sera EOF (ce qui signifie *End Of File*, et est une constante prédéfinie strictement négative) si on est arrivé à la fin de l'entrée et qu'on n'a rien réussi à extraire.

Exercice XIV.6

Écrire un programme acceptant en entrée :

- deux entiers `n` et `p` sur la première ligne;
- puis `n` lignes contenant chacun `p` flottants.

Ce programme devra renvoyer en sortie `n` lignes, contenant chacune la somme des `p` nombres flottants présents sur la ligne correspondante de l'entrée.

Entrée	Sortie
2 3	3.9
1.2 1.3 1.4	6.2
2.3 2.4 1.5	

Remarque

On ne s'intéressera pas au format précis d'écriture des nombres flottants en sortie (nombre de chiffres après la virgule). On peut le spécifier dans la chaîne de format de `printf`, mais ce n'est pas important pour l'instant.

3 Petits problèmes

Ces problèmes sont adaptés du site *France-IOI*. Je vous invite à vous y inscrire et à travailler dessus de temps en temps (si vous n'avez plus de TP à finir...). Les niveaux 1 et 2 n'ont pas un énorme intérêt : commencez par débloquer le niveau 3 (et demandez de l'aide si vous n'y arrivez pas, ce n'est pas évident).

Les deux premiers problèmes ci-dessous font partie de ceux à traiter pour débloquer le niveau 3 : ils ne présentent pas de difficulté algorithmique particulière, mais demandent de comprendre précisément ce qui est demandé et d'être soigneux dans son code. Le troisième problème est un peu plus délicat : une solution excessivement naïve ne s'exécutera pas en un temps satisfaisant.

Exercice XIV.7

La grande bibliothèque de la ville a actuellement des difficultés à gérer son stock car elle est très fréquentée en ce moment et ne dispose pas d'un nombre suffisant d'employés. Vous décidez de l'aider en écrivant un programme informatique permettant de soulager le travail des employés.

La bibliothèque possède `nbLivres` livres indexés de 0 à `nbLivres` – 1. Chaque jour, un certain nombre de clients demandent à emprunter des livres pour une certaine durée. Si le livre est disponible, la requête du client est satisfaite, sinon le client repart sans livre.

Votre programme doit d'abord lire sur une première ligne deux entiers : `nbLivres` ≤ 1000 et `nbJours`. Pour chacun des jours, votre programme lira un entier `nbClients` sur une ligne puis `nbClients` lignes de deux entiers. Le premier entier correspond à l'indice du livre et le second la durée correspondante. Il affichera ensuite, sur des lignes séparées, pour chaque client un 1 si le livre peut être prêté et un 0 dans le cas contraire.

On remarquera que si un client emprunte un livre le jour i Jour pendant une durée duree alors celui-ci ne sera de nouveau disponible qu'au jour i Jour + duree. De plus, si plusieurs personnes demandent le même livre pendant une journée, seule la première a une chance d'être satisfaite.

Entrée	Sortie
2 4	1
2	1
0 3	0
1 3	0
1	1
0 3	0
1	
1 4	
2	
0 2	
0 5	

Exercice XIV.8

On considère une suite finie de nombres réels x_0, \dots, x_{N-1} , et l'on considère une opération de lissage, où l'on remplace chaque valeur (sauf la première et la dernière) par la moyenne de la valeur suivante et de la valeur précédente. Par exemple, la suite 1, 3, 4, 5 serait remplacée par 1, 2.5, 4, 5, puis par 1, 2.5, 3.75, 5 si l'on itère le procédé.

On demande de résoudre le problème suivant :

Entrées

- la première ligne de l'entrée contient un entier N ;
- la deuxième ligne de l'entrée contient un flottant Δ_{\max} ;
- chacune des N lignes suivantes contient un flottant x_i .

Sortie il faut afficher un entier : le nombre d'étapes nécessaires pour que la différence entre deux valeurs successives de la suite ne dépasse pas Δ_{\max} .

Garanties

- $1 \leq N \leq 100$
- $0 \leq \Delta_{\max} \leq 100$
- $-100 \leq x_i \leq 100$ pour $0 \leq i < N$
- On garantit également que la propriété recherchée sera obtenue en au plus 5 000 étapes de lissage.

Exercice XIV.9

On considère une série x_1, \dots, x_p d'entiers de l'intervalle $[1 \dots N]$. Pour $i \in [1 \dots p]$, on définit $\text{ppa}(i)$ (*plus petit absent*) comme le plus petit entier $x \in [1 \dots N]$ tel que $x \notin \{x_1, \dots, x_i\}$, ou -1 si tous les éléments de $[1 \dots N]$ appartiennent à x_0, \dots, x_i .

On demande de résoudre le problème suivant :

Entrées

- La première ligne de l'entrée contient les deux entiers N et P, séparés par une espace.
- Les P lignes suivantes contiennent les entiers x_1, \dots, x_p à raison d'un entier par ligne.

Sortie Il faut afficher P lignes : la i -ème ligne contiendra $\text{ppa}(i)$.

Garanties

- $1 \leq N \leq 10^9$
- $1 \leq P \leq 250\,000$

PILE D'APPEL, ENSEMBLE DE MANDELBROT

Ligne de commande : gcc -Wall -Wextra -Wvla -fsanitize=address,undefined.

I Retour sur scanf

Exercice XVI.1

Si vous n'avez pas eu le temps de la traiter, reprendre la partie sur scanf du TP précédent.

2 Quelques expériences sur la pile et les pointeurs

Exercice XV.2

p. 476

On considère le programme suivant :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double expo(double x, int n){
5     double res = 1.0;
6     for (int i = 0; i < n; i = i + 1){
7         res = res * x;
8     }
9     return res;
10 }
11
12 double f(double x, double y, int n){
13     double xn = expo(x, n);
14     double yn = expo(y, n);
15     return xn + yn;
16 }
17
18 int main(int argc, char* argv[]){
19     if (argc != 4) return -1; // *
20     int n = atoi(argv[1]);
21     double x = atof(argv[2]); // *
22     double y = atof(argv[3]);
23     printf("%f^%d + %f^%d = %f\n", x, n, y, n, f(x, y, n)); // *
24     return 0; // *
25 }
```

- Combien d'arguments ce programme attend-il en ligne de commande ? De quels types sont-ils ?

On suppose dans la suite que les fonctions de la bibliothèque standard appelées (atoi, atof, printf) n'appellent pas elles-mêmes de fonctions.

- Lors de l'exécution de la ligne 19, quelle est la hauteur de la pile (en nombre de blocs

- d’activation) ?
3. Quelle est la hauteur maximale de la pile lors de l’exécution de la ligne 21 ?
 4. Même question pour la ligne 23.
 5. Même question pour la ligne 24.

Exercice XV.3

p. 476

On considère le code suivant :

```

1 #include <stdio.h>
2
3 void f(int n, int* nmax){
4     printf("Début de l'appel de f(%d, _)\n", n); // *
5     printf("n      = %d\n", n);
6     printf("&n      = %p\n", &n);
7     printf("nmax    = %p\n", nmax);
8     printf("*nmax = %d\n", *nmax);
9     printf("&nmax = %p\n", &nmax);
10    if (n < *nmax) f(n + 1, nmax);
11    printf("Fin de l'appel de f(%d, _)\n", n); // *
12 }
13
14 int main(void){
15     int N = 2;
16     f(0, &N);
17     return 0;
18 }
```

1. En oubliant les `printf` situés ailleurs qu’aux lignes 4 et 11, prévoir l’affichage produit par la fonction.
2. Parmi les autres lignes provoquant un affichage, lesquelles :
 - affichent toujours la même chose ?
 - affichent des choses différentes, mais que l’on peut prévoir parfaitement en regardant le code ?
 - affichent des choses différentes et partiellement imprévisibles ?
3. Prévoir le plus complètement possible l’affichage produit par le programme.
4. Exécuter ce programme (dans le terminal et éventuellement avec *C Tutor*) et observer l’affichage produit.
5. Quelle est la taille (en octets) du bloc d’activation de `f` ?

Exercice XV.4

p. 477

1. Écrire une fonction `incremente` qui prend en entrée un pointeur vers un entier et incrémente la valeur de cet entier de 1 (cette fonction ne renverra rien).
2. Dans tous les exemples suivants, on souhaite qu’un appel `f(px, py)` incrémente celle des valeurs pointées par `px` et `py` qui est la plus petite (ou celle pointée par `px` en cas d’égalité). Par exemple :

```
#include <stdio.h>

int main(void){
    int x = 4;
    int y = 3;
    printf("x = %d, y = %d\n", x, y);
    f(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    f(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

doit donner l'affichage suivant :

```
x = 4, y = 3
x = 4, y = 4
x = 5, y = 4
```

Dans chaque cas, dire si :

- il y a un problème de type (et si oui, où) ;
- les fonctions ont bien le comportement attendu (uniquement dans les cas où il n'y a pas de problème de type). Si ce n'est pas le cas, on expliquera d'où vient le problème.

a.

```
int plus_petit(int x, int y){
    if (x <= y) return x;
    return y;
}

void f(int* px, int* py){
    incremente(plus_petit(*px, *py));
}
```

b.

```
int* plus_petit(int x, int y){
    if (x <= y) return &x;
    return &y;
}

void f(int* px, int* py){
    incremente(plus_petit(*px, *py));
}
```

c.

```
int* plus_petit(int* x, int* y){
    if (*x <= *y) return x;
    return y;
}

void f(int* px, int* py){
    incremente(plus_petit(px, py));
}
```

d.

```

int* plus_petit(int* x, int* y){
    int a = *x;
    int b = *y;
    if (a <= b) return &a;
    return &b;
}

void f(int* px, int* py){
    incremente(plus_petit(px, py));
}

```

3 Fractale de Mandelbrot

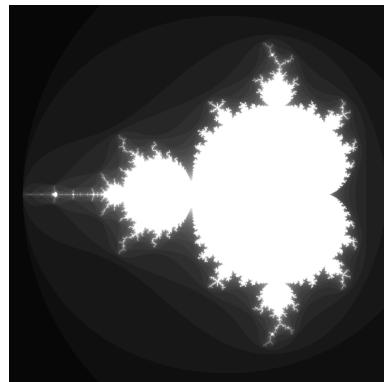


FIGURE XV.1 – L'ensemble de Mandelbrot.

Pour $c \in \mathbb{C}$, on considère la fonction :

$$\begin{aligned} f_c : \mathbb{C} &\rightarrow \mathbb{C} \\ z &\mapsto z^2 + c \end{aligned}$$

À partir de cette fonction, on définit la suite $(z_n(c))$ par :

$$\begin{cases} z_0(c) = 0 \\ z_{n+1}(c) = f_c(z_n(c)) \quad \text{pour } n \geq 0 \end{cases}$$

L'ensemble de Mandelbrot est l'ensemble des complexes c tels que $(u_n(c))$ soit bornée. Dans la suite, on note \mathcal{M} cet ensemble.

Exercice XV.5 – Bornes

Cette question est purement mathématique : il peut être pertinent de la traiter plus tard, chez vous, et d'admettre le résultat de la dernière question pour la suite.

1. Soit $c \in \mathbb{C}$ et $n \in \mathbb{N}$. On note $z_n = z_n(c)$ et l'on suppose $|z_n| > 2$ et $|z_n| > |c|$.
 - a. Montrer que pour $m \geq 0$, on a $|z_{n+m}| \geq |c| + 2^m (|z_n| - |c|)$.
 - b. En déduire que $c \notin \mathcal{M}$.
2. Montrer que si $|c| > 2$, alors $|z_1| > |c|$.
3. Montrer que $c \in \mathcal{M} \iff (\forall n \in \mathbb{N}, |z_n| \leq 2)$.

On définit désormais $\mathcal{M}_N = \{c \in \mathbb{C} \mid \forall n \leq N, |z_n(c)| \leq 2\}$. D'après ce qui précède, on a :

- $\forall N \in \mathbb{N}, \mathcal{M} \subset \mathcal{M}_N$;
- $\mathcal{M} = \bigcap_{N \in \mathbb{N}} \mathcal{M}_N$

Pour N suffisamment grand, \mathcal{M}_N sera une bonne approximation de \mathcal{M} . Pour déterminer si $c \in \mathcal{M}$, on pourra donc se fixer un entier $itermax$ puis calculer les valeurs successives de $z_n(c)$:

- dès que l'une de ces valeurs a un module strictement supérieur à 2, on sait que $c \notin \mathcal{M}$;
- si toutes les valeurs jusqu'à $n = itermax$ ont un module inférieur ou égal à 2, alors $c \in \mathcal{M}_{itermax}$ et l'on considère que $c \in \mathcal{M}$ (ce qui n'est pas forcément vrai).

Exercice XV.6

p. 478

Écrire une fonction **int** divergence(**double** xc, **double** yc, **int** itermax). Cette fonction prend en entrée un complexe c (parties réelle et imaginaire) et un entier $itermax$ et doit renvoyer :

- le plus petit $n \leq itermax$ tel que $c \notin \mathcal{M}_n$, s'il en existe un ;
- $itermax + 1$ sinon.

On définit deux constantes **ROWS** et **COLS** ainsi qu'un tableau bidimensionnel global :

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 800
#define COLS 800

int arr[ROWS][COLS];
```

- $arr[0][0]$ correspondra au pixel en haut à gauche de l'image ;
- $arr[ROWS - 1][0]$ au pixel en bas à gauche ;
- $arr[0][COLS - 1]$ au pixel en haut à droite ;
- $arr[ROWS - 1][COLS - 1]$ au pixel en bas à droite.

Exercice XV.7

p. 478

On souhaite que l'image corresponde dans le plan complexe aux points $z = x + iy$ tels que $x_{\min} \leq x \leq x_{\max}$ et $y_{\min} \leq y \leq y_{\max}$. Écrire deux fonctions

```
double re(int j, double xmin, double xmax);
double im(int i, double ymin, double ymax);
```

tenant en entrée un numéro de ligne ou de colonne dans le tableau **arr** et renvoyant la partie réelle ou imaginaire du complexe correspondant à ce pixel.

On souhaite bien sûr que $x_{\min} + iy_{\min}$ soit en bas à gauche de l'image et $x_{\max} + iy_{\max}$ en haut à droite.

Exercice XV.8

p. 478

Écrire une fonction

```
void fill_tab(double xmin, double xmax,
             double ymin, double ymax,
             int itermax);
```

qui remplit le tableau global **arr** avec les valeurs renvoyées par la fonction **divergence** pour

les différents pixels.

Exercice XV.9

p. 479

On suppose dans cette question que la fonction `fill_tab` a déjà été appelée, et donc que le tableau `arr` contient les valeurs de divergence pour les différents pixels.

1. Écrire une fonction `void print_pixel_bw(int i, int j, int itermax)` qui affiche 255 255 255 \n ou 0 0 0 \n selon que le pixel (i,j) de l’image appartient ou non à $M_{itermax}$.
2. Écrire une fonction `void print_tab(int itermax)` qui affiche le fichier PPM correspondant à l’image calculée. On pourra reprendre (et modifier!) le code écrit au TP 13.
3. Écrire la fonction `main` de manière à obtenir le comportement suivant :
 - si le programme est appelé sans argument en ligne de commande, il utilise les valeurs $x_{\min} = y_{\min} = -2$, $x_{\max} = y_{\max} = 2$ et `itermax = 20`;
 - s’il est appelé avec un argument, cet argument est utilisé pour `itermax`;
 - s’il est appelé avec cinq arguments, le premier est utilisé pour `itermax` et les autres pour x_{\min} , x_{\max} , y_{\min} et y_{\max} (dans l’ordre);
 - s’il est appelé avec un autre nombre d’arguments, il termine sans rien faire en affichant un message d’erreur.

Dans tous les cas (sauf le dernier), le programme doit afficher le contenu du fichier PPM décrit dans les questions précédentes sur la sortie standard.

On utilisera la fonction `atoi` pour convertir une chaîne de caractères en entier et `atof` pour convertir en flottant.

Exercice XV.10

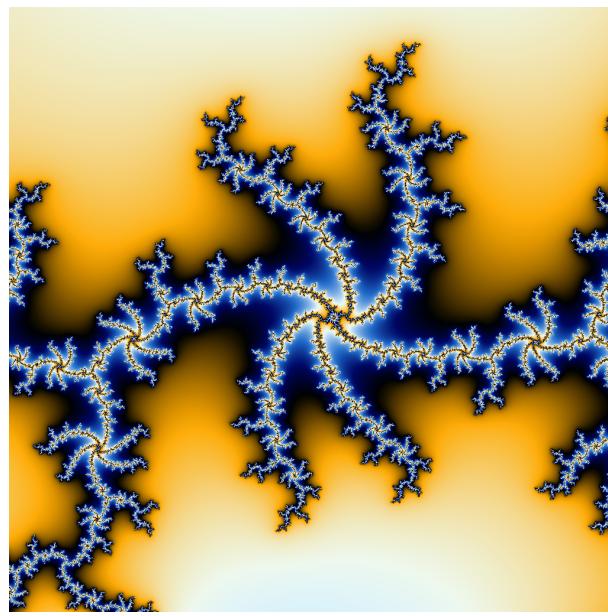
p. 479

Modifier le code pour obtenir un affichage en niveaux de gris. On affichera une couleur d’autant plus sombre que le point c est rapidement sorti du disque de rayon 2 (centré en l’origine).

Remarque

Pour obtenir un dessin vraiment « joli »¹, il faut définir une notion d’itération partielle pour éviter les aplats de couleur, puis utiliser une palette cyclique et interpoler.

1. C’est un terme technique, dont la définition nous emmènerait trop loin du programme...

FIGURE XV.2 – Zoom sur la partie $0.00172 \leq x \leq 0.00184, -0.82258 \leq y \leq -0.82246$.

4 Pour chercher

Exercice XV.II – Maxima par plage

p. 480

On considère un tableau $t = t_0, \dots, t_{n-1}$ et un entier $h \geq 1$. Pour $0 \leq i \leq n-h$, on définit :

$$m_h(i) = \max(t_i, \dots, t_{i+h-1})$$

1. Écrire un algorithme (simple) permettant de calculer tous les $m_h(i)$.
2. Déterminer la complexité de cet algorithme.
3. Trouver un algorithme permettant de trouver tous les $m_h(i)$ en temps $O(n)$. C'est loin d'être évident ! N'hésitez pas à demander des indications.
4. Écrire un programme en C implémentant cet algorithme.

Solutions

Correction de l'exercice XV.2 page 469

1. Ce programme attend 3 arguments : un entier et deux **double**.
2. Le seul bloc d'activation est ici celui de **main** : la pile est de hauteur 1.
3. Pendant l'appel à **atof**, il y aura deux blocs : un pour **main** et un pour **atof** (il pourrait y en avoir davantage si **atof** faisait un appel, mais on suppose que ce n'est pas le cas). Donc la hauteur maximale est 2 à cette ligne.
4. **main** appelle **f** qui appelle **expo** (deux fois successivement). Pendant chacun de ces deux appels à **expo**, la pile est de hauteur 3. L'appel à **f** se termine avant que celui à **printf** ne commence, donc 3 est en fait la hauteur maximale de la pile.
5. Dans cette ligne, le seul bloc d'activation est à nouveau celui de **main**, donc la hauteur vaut 1.

Correction de l'exercice XV.3 page 470

1. On obtiendra :

```
Début de l'appel de f(0, _)
Début de l'appel de f(1, _)
Début de l'appel de f(2, _)
Fin de l'appel de f(2, _)
Fin de l'appel de f(1, _)
Fin de l'appel de f(0, _)
```

2. ■ Les valeurs de **nmax** et ***nmax** ne changent pas au cours des appels récursifs.
 - La valeur de **n** sera 0, puis 1, puis 2.
 - Les valeurs de **&n** et **&nmax** sont des adresses sur la pile : elle seront différentes à chaque appel, et on ne peut pas vraiment connaître leur valeur *a priori* (on peut prévoir comment elles évoluent si on connaît l'architecture).
3. Ce qu'on peut prévoir :

```
Début de l'appel de f(0, _)
n      = 0
&n    = pointeur_1
nmax  = pointeur_2
*nmax = 2
&nmax = pointeur_3
Début de l'appel de f(1, _)
n      = 1
&n    = pointeur_4
nmax  = pointeur_2
*nmax = 2
&nmax = pointeur_5
Début de l'appel de f(2, _)
n      = 2
&n    = pointeur_6
nmax  = pointeur_2
*nmax = 2
&nmax = pointeur_7
Fin de l'appel de f(2, _)
Fin de l'appel de f(1, _)
```

Fin de l'appel de f(0, _)

4.

5. On obtient l'affichage suivant (sur ma machine, en compilant avec une certaine version d'un certain compilateur, pour une certaine exécution, avec certaines options de compilation...):

```
Début de l'appel de f(0, _)
n      = 0
&n    = 0x7ffe594b0ffc
nmax  = 0x7ffe594b1014
*nmax = 2
&nmax = 0x7ffe594b0ff0
Début de l'appel de f(1, _)
n      = 1
&n    = 0x7ffe594b0fdc
nmax  = 0x7ffe594b1014
*nmax = 2
&nmax = 0x7ffe594b0fd0
Début de l'appel de f(2, _)
n      = 2
&n    = 0x7ffe594b0fbc
nmax  = 0x7ffe594b1014
*nmax = 2
&nmax = 0x7ffe594b0fb0
Fin de l'appel de f(2, _)
Fin de l'appel de f(1, _)
Fin de l'appel de f(0, _)
```

On voit que les différents pointeurs obtenus pour &n sont à 0x20 les uns des autres, et qu'il en est de même pour les pointeurs &nmax. Cela indique que le bloc d'activation fait $0x20 = 32$ octets.

Remarque

Ce type de raisonnement est un peu dangereux, parce que rien n'oblige le compilateur à faire quelque chose de simple (si on compile avec toutes les optimisations, on obtient quelque chose de bien plus difficile à interpréter). De toute façon je ne vous demanderai jamais de répondre à une question de ce type en évaluation.

Correction de l'exercice XV.4 page 470

1. Pas de problème :

```
void incremente(int* p){
    *p = *p + 1;
}
```

2. a. `incremente` attend un `int*`, on lui donne un `int` : il y a une erreur de type.
 b. Ici, la fonction `plus_petit` renvoie l'adresse de l'un de ses arguments. Or ces arguments sont des variables locales à la fonction : leur durée de vie est celle de l'appel. On passe donc à `incremente` un pointeur vers un objet qui n'existe plus : le comportement du programme est non défini.
 c. Cette version est correcte.
 d. Cette version n'est pas correcte : on renvoie encore un pointeur vers une variable locale, ce qui est illégal.

Correction de l'exercice XV.6 page 473

```
int divergence(double xc, double yc, int itermax){  
    double x = 0.;  
    double y = 0.;  
    int i = 0;  
    while (x*x + y*y <= 4. && i <= itermax){  
        double tmp = x;  
        x = x*x - y*y + xc;  
        y = 2. * tmp * y + yc;  
        i++;  
    }  
    return i;  
}
```

Correction de l'exercice XV.7 page 473

```
double re(int j, double xmin, double xmax){  
    double ratio = (double)j / (COLS - 1);  
    return xmin + ratio * (xmax - xmin);  
}  
  
double im(int i, double ymin, double ymax){  
    double ratio = (double)i / (ROWS - 1);  
    return ymin + ratio * (ymax - ymin);  
}
```

Correction de l'exercice XV.8 page 473

```
void fill_tab(double xmin, double xmax, double ymin, double ymax, int itermax){  
    for (int i = 0; i < ROWS; i++){  
        for (int j = 0; j < COLS; j++){  
            double x = re(j, xmin, xmax);  
            double y = im(i, ymin, ymax);  
            arr[i][j] = divergence(x, y, itermax);  
        }  
    }  
}
```

Correction de l'exercice XV.9 page 474

```

void print_pixel_bw(int i, int j, int itermax){
    int c = 0;
    if (arr[i][j] > itermax){
        c = 255;
    }
    for (int k = 0; k < 3; k++){
        printf("%d ", c);
    }
    printf("\n");
}

void print_tab(int itermax){
    printf("P3\n");
    printf("%d %d\n", COLS, ROWS);
    printf("255\n");
    for (int i = 0; i < ROWS; i++){
        for (int j = 0; j < COLS; j++){
            print_pixel_gs(i, j, itermax);
        }
    }
}

int main(int argc, char* argv[]){
    double xmin = -2.;
    double xmax = 2.;
    double ymin = -2.;
    double ymax = 2.;
    int itermax = 20;
    if (argc >= 2) {
        itermax = atoi(argv[1]);
    }
    if (argc == 6){
        xmin = atof(argv[2]);
        xmax = atof(argv[3]);
        ymin = atof(argv[4]);
        ymax = atof(argv[5]);
    }
    fill_tab(xmin, xmax, ymin, ymax, itermax);
    print_tab(itermax);
    return 0;
}

```

Correction de l'exercice XV.10 page 474

Pour quelque chose de basique, il n'y a pas grand chose à changer. Il suffit de définir :

```

void print_pixel_gs(int i, int j, int itermax){
    int c = 255 * arr[i][j] / itermax;
    for (int k = 0; k < 3; k++){
        printf("%d ", c);
    }
    printf("\n");
}

```

Ensuite on change une ligne dans `print_tab`. Cela va marcher raisonnablement bien pour la fenêtre d'affichage par défaut, mais si l'on zoomé on risque d'avoir un contraste très mauvais : en effet, rien ne dit que toutes les valeurs de 0 à `itermax` seront prises. Pour éviter ce problème, on peut déterminer les valeurs minimale et maximale de `arr` et les faire correspondre à du noir et à du blanc, respectivement.

Correction de l'exercice XV.II page 475

On donne une solution efficace, qui suppose définies quelques variables globales :

- un entier `N` (la taille du tableau);
- un entier `H` (le `h` de l'énoncé);
- quatre tableaux d'entiers de même taille `N` : `arr` le tableau d'entrée, `gauche` et `droite` des tableaux auxiliaires et `result` le tableau de sortie.

Pour chaque entier `i` vérifiant $kh \leq i < (k+1)h$, on précalcule :

- $d_i = \max(\text{arr}[i], \text{arr}[i+1], \dots, \text{arr}[(k+1)h-1])$ que l'on met dans le tableau `droite`
- $g_i = \max(\text{arr}[kh], \text{arr}[kh+1], \dots, \text{arr}[i])$ que l'on met dans le tableau `gauche`.

Ces précalculs peuvent se faire en temps linéaire. On a ensuite $m_i = \max(d_i, g_{i+h-1})$. Le code calcule correctement les $m_h(i)$ pour les `i` situés entre $n-h$ et n (en prenant le maximum jusqu'à la fin du tableau). L'énoncé ne le demandait pas, ce qui permet de simplifier un peu.

```

void fill_droite(void){
    for (int i = 0; i < N; i = i + H){
        int j = min(i + H - 1, N - 1);
        droite[j] = arr[j];
        j--;
        for ( ; j >= i; j--){
            droite[j] = max(arr[j], droite[j + 1]);
        }
    }
}

void fill_gauche(void){
    for (int i = 0; i < N; i = i + H){
        gauche[i] = arr[i];
        for (int j = i + 1; j < min(i + H, N); j++){
            gauche[j] = max(gauche[j - 1], arr[j]);
        }
    }
}

void f(void){
    fill_gauche();
    fill_droite();
    int i;
    for (i = 0; i + H - 1 < N; i++){
        result[i] = max(droite[i], gauche[i + H - 1]);
    }
    for ( ; i < N; i++){
        result[i] = droite[i];
    }
}

```

TABLEAUX DYNAMIQUES

I Types enregistrement en C : les `struct`

Introduction

On peut définir en C des types enregistrement :

```
struct complexe {
    double re;
    double im;
}
```

On accède ensuite à un champ d'un complexe z par la notation `z.re` et `z.im`. Une fonction peut prendre une `struct` comme argument, par valeur, et aussi renvoyer une `struct` (deux choses qui ne sont pas possibles pour un tableau). Par exemple, la fonction suivante calcule le conjugué d'un nombre complexe, **et elle n'a pas d'effet secondaire** (puisque l'argument est passé par valeur) :

```
struct complexe conjugue(struct complexe z){
    z.im = -z.im;
    return z;
}
```

Initialisation

Pour définir un objet de type `struct`, on peut :

- soit procéder en deux temps :

```
struct complexe I;
I.re = 0.;
I.im = 1.;
```

- soit initialiser directement, avec cette syntaxe :

```
struct complexe I = { .re = 0., .im = 1. };
```

Utilisation d'un `typedef`

Nous avons déjà vu qu'on pouvait utiliser le mot-clé `typedef` pour définir de nouveaux types (ou plutôt des alias de type) :

```
// On définit entier comme un alias de int (intérêt douteux)
typedef entier int;

// On définit int_ptr comme un alias de int*, ce qui
// peut parfois rendre le code plus lisible.
typedef int_ptr int*;

// On définit rgb comme un tableau de trois int
typedef int rgb[3];
```

Il est également possible de définir un alias pour le type **struct complexe** :

```
typedef struct complexe complexe;
```

Remarque

On aurait pu choisir **toto** comme alias :

```
typedef struct complexe toto;
```

Cependant, le plus simple est de reprendre le nom utilisé pour la **struct** (autrement dit, de faire **typedef struct foo foo**).

On peut ensuite définir la fonction **conjugue**, par exemple, de manière plus concise :

```
complexe conjugue(complexe z){  
    z.im = -z.im;  
    return z;  
}
```

Pointeur vers une **struct**

Il est très courant de manipuler des pointeurs vers des **struct** :

- soit parce que la **struct** est un peu grosse et qu'il est inefficace de la passer par valeur à une fonction (ce qui implique une copie) ;
- soit parce qu'on souhaite qu'une fonction puisse modifier la **struct** passée en argument (ce qui demande de passer en fait un pointeur vers cette **struct**).

En soi, cela ne pose aucun problème :

```
void conjugue_en_place(complexe* z){  
    (*z).im = -(z).im;  
}
```

Cependant, les parenthèses autour de ***z** sont **obligatoires** ici : l'expression ***z.im** est lue comme ***(z.im)** (ce qui n'a aucun sens). Comme ces parenthèses deviennent très rapidement pénibles, on dispose d'une syntaxe spéciale :

si **ps** est un pointeur vers une **struct**, alors **ps->champ** signifie **(*ps).champ**.

On peut donc écrire :

```
void conjugue_en_place(complexe* z){  
    z->im = -z->im;  
}
```

Attention à ne pas confondre les deux syntaxes : **s.champ** si **s** est une **struct**, **s->champ** si **s** est un pointeur vers une **struct** :

```
complexe z;  
complexe* pz;  
z.re = 3.;  
pz->im = 1.;
```

2 Un tableau qui connaît sa taille

Comme nous l'avons vu, un tableau ne connaît pas sa taille¹, ce qui est très souvent problématique :

- il faut systématiquement penser à passer la taille comme paramètre supplémentaire aux fonctions, et bien sûr passer la bonne taille;
- le compilateur ne peut pas ajouter de vérifications pour les accès hors-bornes.

Le deuxième point est particulièrement problématique : dans la plupart des langages, effectuer un accès hors-borne à un tableau résulte, de manière certaine, en une erreur bien définie à l'exécution. Si l'on veut aller chercher le dernier pourcent de performance, on peut souvent compiler avec une option demandant de désactiver cette vérification, mais c'est en général une très mauvaise idée. En C en revanche, un accès hors-borne donne un comportement non défini :

- si on a de la chance, cela résultera en une *segmentation fault* immédiate (quand la zone mémoire à laquelle on tente d'accéder n'appartient pas au processus);
- sinon, on va lire ou écrire dans une autre variable, et l'exécution va continuer avec un état du programme incohérent;
- si on n'a vraiment pas de chance, on sera dans le deuxième cas, mais pas par hasard : un accès hors-borne est par nature une faille de sécurité, qui peut être exploitée.

Pour éviter ces problèmes, on peut tout simplement définir une **struct** qui contiendra la taille et un pointeur vers les données :

```
struct int_array {
    int* data;
    int len;
};

typedef struct int_array int_array;
```

Pour créer un `int_array`, on peut utiliser la fonction suivante (que je vous invite à lire **attentivement**) :

```
int_array* array_create(int len, int x){
    // On alloue le stockage pour la struct
    int_array* t = (int_array*)malloc(sizeof(int_array));
    // Et le stockage pour les données
    int* data = (int*)malloc(len * sizeof(int));
    for (int i = 0; i < len; i++){
        data[i] = x;
    }
    t->len = len;
    t->data = data;
    return t;
}
```

Assertions Pour tirer parti tirer du fait que nous connaissons la taille du tableau, il faut arrêter l'exécution du programme en cas d'accès hors-borne. Le plus simple est d'utiliser une assertion :

```
#include <assert.h>

int main(void){
    ...
    // si n > 3, le programme s'arrête et affiche un message d'erreur
    assert(n <= 3);
    ...
}
```

1. Il y a une subtilité pour les tableaux alloués statiquement, mais globalement ça reste vrai.

Exercice XVI.I

p. 489

1. Écrire une fonction `array_get(int_array* t, int i)` qui renvoie l'élément d'indice `i` de `t`. Cette fonction vérifiera que l'accès est licite à l'aide d'une assertion.

```
int array_get(int_array* t, int i);
```

2. Écrire une fonction `array_set(int_array* t, int i, int x)` qui écrit `x` dans la case d'indice `i` de `t`. À nouveau, on utilisera une assertion pour vérifier la licéité de l'accès.

```
void array_set(int_array* t, int i, int x);
```

3. Écrire une fonction `array_delete(int_array* t)` qui libère tout le stockage associé à `t`.

```
void array_delete(int_array* t);
```

On supposera que `t` a été créé par un appel à `array_create`.

3 Tableaux dynamiques (ou vecteurs)

La structure abstraite de *tableau dynamique* est une extension de la structure abstraite de *tableau* : on peut toujours accéder facilement (et rapidement) à un élément quelconque par son indice, mais il est également possible d'ajouter ou de supprimer un élément. En règle générale, cette opération n'est possible qu'à l'extrême droite du tableau².

Opération	Type	Effet
get	DYNARRAY × int → ELT	Accès à un élément
set	DYNARRAY × int × ELT → {}	Modification d'un élément
push	DYNARRAY × ELT → {}	Ajout d'un élément à la fin du tableau
pop	DYNARRAY → ELT	Récupération et suppression de l'élément le plus à droite
create	{ } → DYNARRAY	Création d'un tableau vide (fonction ne prenant pas d'argument)
length	DYNARRAY → int	Nombre d'éléments présents

FIGURE XVI.1 – Signature d'un type DYNARRAY impératif. On utilise {} pour désigner un type comme **void** ou **unit**.

Remarques

- Ce type abstrait est très utilisé en pratique, et de nombreux langages en fournissent une réalisation dans leur bibliothèque standard. C'est le cas de Python (type `list`), de Java (type `ArrayList`), de C++ (type `std::vector`)...
- Étrangement, la bibliothèque standard de OCaml ne propose pas de tableaux dynamiques (ça viendra sans doute un jour). Les deux bibliothèques tierces qui sont souvent utilisées en remplacement de la bibliothèque standard (**Batteries** et **Core**) proposent bien sûr cette structure de données.
- La bibliothèque standard du langage C ne propose pas non plus de tableaux dynamiques, ce qui n'est pas très surprenant puisqu'elle ne propose en fait aucune structure de données. Il existe bien évidemment d'innombrables implémentations tierces.
- On a donné une signature impérative, mais des versions fonctionnelles existent également (nous en rencontrerons cette année). Ces versions fonctionnelles sont cependant d'usage moins courant.

2. On peut concevoir des variantes permettant de le faire aux deux extrémités.

- Attention à ne pas confondre *tableau alloué dynamiquement* (ce qui signifie que les données sont sur le tas, et n'est pertinent, pour simplifier, qu'en C/C++) et *tableau dynamique* (qui est le type abstrait qui nous intéresse).

3.1 Réalisation naïve

On utilise la structure suivante :

```
struct int_dynarray {
    int len;
    int capacity;
    int* data;
};

typedef struct int_dynarray int_dynarray;
```

- L'entier `len` représente le nombre d'éléments *actuellement présents* dans le tableau.
- L'entier `capacity` représente la *capacité* du tableau, c'est-à-dire la taille du bloc vers lequel pointe `data`.
- Les éléments sont stockés à partir du début du bloc : il peut y avoir de la place libre à la fin du bloc (si `capacity > len`). Dans ce cas, les valeurs présentes dans les cases « libres » n'ont aucun sens.

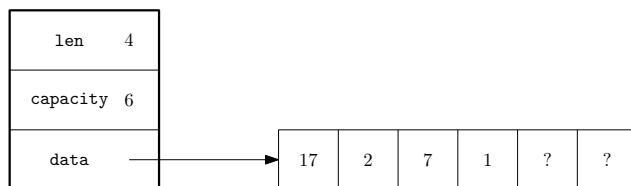


FIGURE XVI.2 – Un `int_dynarray_t` de capacité 6 contenant 4 éléments.

Les fonctions `get` et `set` fonctionnent exactement comme avant. Pour gérer les `pop` et les `push`, on peut imaginer procéder ainsi :

- un `pop` fait diminuer `len` de 1, et ne change pas `capacity` (il y a une erreur si `len` vaut zéro);
- pour un `push`, il y a deux cas :
 - si `len < capacity`, on écrit le nouvel élément à droite et l'on incrémente `len`;
 - si `len = capacity`, on alloue un nouveau bloc `data`, de taille `capacity + 1`, on recopie l'ancien dans le nouveau et on y ajoute l'élément supplémentaire.

Exercice XVI.2

p. 489

1. Écrire une fonction `length(int_dynarray*)`.

```
int length(int_dynarray* t);
```

2. Écrire une fonction `make_empty(void)` renvoyant un pointeur vers un `int_dynarray` vide. On initialisera `data` à `NULL`.

```
int_dynarray* make_empty(void)
```

3. Modifier les fonctions `get` et `set` écrites plus haut pour qu'elles s'appliquent aux `int_dynarray`.

```
int get(int_dynarray* t, int i);
void set(int_dynarray* t, int i, int x);
```

4. Écrire la fonction pop (qui ne modifiera jamais la capacité du tableau) :

```
int pop(int_dynarray* t);
```

5. Écrire une fonction resize(int_dynarray_t* t, int new_capacity) qui alloue un nouveau bloc data, copie le contenu de l'ancien bloc dans le nouveau, met à jour les champs de t et libère l'ancien bloc.

```
void resize(int_dynarray* t, int new_capacity);
```

6. Écrire la fonction push :

```
void push(int_dynarray* t);
```

7. Écrire une fonction delete(int_dynarray* t) qui détruit le tableau dynamique pointé par t en libérant ce qu'il faut libérer.

```
void delete(int_dynarray* t);
```

Exercice XVI.3 – Analyse de la réalisation naïve

p. 490

On considère une série de n opérations push successives sur un tableau dynamique initialement vide. Déterminer le coût total de ces opérations.

On considérera pour simplifier que les opérations malloc et free peuvent se faire en temps constant, mais cela ne change de toute façon pas le résultat ici.

3.2 Réalisation efficace

Le résultat de l'exercice XVI.3 montre que la réalisation naïve n'est pas satisfaisante : on souhaite que les opérations push et pop se fassent rapidement. Il n'est pas vraiment possible d'obtenir une complexité constante dans le pire des cas pour ces fonctions, mais on peut obtenir une complexité *amortie* en O(1) pour push en utilisant la stratégie suivante :

- s'il reste de la place libre, on ajoute l'élément dans le tableau (comme pour la solution naïve) ;
- sinon, on procède aussi comme pour la solution naïve, sauf que le nouveau bloc alloué est de taille $2 * \text{capacity}$. Il faut ajouter un cas particulier si capacity est nul.

Exercice XVI.4

p. 491

Apporter les modifications nécessaires aux fonctions push et resize pour utiliser la nouvelle stratégie.

Exercice XVI.5 – Analyse amortie sans réduction de taille

p. 491

1. Quelles sont les complexités des opérations pop, get et set ?
2. Si t est un tableau dynamique de longueur n, quelle est la complexité d'un push dans le pire cas ? dans le meilleur cas ?
3. On considère une série de n opérations push et pop sur un tableau initialement vide. Il n'y a aucune contrainte sur les opérations effectuées (sauf qu'on ne fait pas de pop sur un tableau vide) : on peut avoir n push, ou n/2 push suivis de n/2 pop, ou une alternance de

push et de pop...

Montrer que le coût total de cette série de n opérations est en $O(n)$.

Comme une série de n opérations (sur un tableau initialement vide) a un coût total en $O(n)$, on dira que la *complexité amortie* d'une opération push (ou pop) est en $O(1)$.

Cette complexité amortie est satisfaisante, mais notre stratégie a un gros défaut : la mémoire utilisée peut-être arbitrairement plus grande que celle nécessaire à stocker le nombre actuel d'éléments. En effet, la taille du bloc alloué ne diminue jamais lors d'une opération pop, et l'on peut donc avoir un tableau vide occupant une place proportionnelle au nombre maximum d'éléments qu'il a contenus par le passé.

Exercice XVI.6

p. 491

On propose la stratégie suivante :

- si `len` devient strictement inférieur à `capacity / 2` après un `pop`, on ré-alloue le bloc de données en lui donnant une taille `capacity / 2`;
- sinon, on procède comme avant.

Le strictement inférieur garantit que l'on ne repasse jamais à une capacité de zéro, ce qui simplifie légèrement les choses.

1. Apporter les modifications nécessaires à la fonction `pop`.
2. Montrer qu'une série de n opérations successives peut avoir un coût de l'ordre de n^2 , et le mettre en évidence expérimentalement.

Exercice XVI.7

p. 492

Pour régler ce problème, on modifie légèrement la stratégie :

- si `len` devient strictement inférieur à `capacity / 4`, on ré-alloue un bloc de taille `capacity / 2`;
- sinon, on supprime l'élément sans ré-allouer.

En s'inspirant de la démonstration faite pour la réalisation d'une file fonctionnelle à l'aide de deux listes, montrer que la complexité amortie des opérations `pop` et `push` est en $O(1)$. On pourra prendre comme potentiel :

$$\Phi(t) = |2\text{len}(t) - \text{capacity}(t)|.$$

3.3 Opérations supplémentaires

Les opérations que nous avons définies jusqu'ici sont les seules opérations élémentaires sur un tableau dynamique.

Exercice XVI.8

p. 493

Dans cet exercice, on considère que les seuls moyens d'interagir avec un tableau dynamique sont les fonctions définies dans la signature donnée en figure XVI.1. Autrement dit, l'implémentation est « cachée » : on ne sait même pas que `int_dynarray` est défini comme une `struct`, et on ne connaît certainement pas les champs de cette `struct`.

1. Écrire une fonction permettant d'insérer un nouvel élément à un emplacement arbitraire i du tableau. Les éléments présents aux indices $j \geq i$ seront décalés d'une case vers la droite.

```
void insert_at(int_dynarray* t, int i, int x)
```

Remarque

Les valeurs acceptables pour i vont de 0 à la longueur du tableau incluse (dans ce cas, l'insertion revient à un `push`).

2. Déterminer la complexité de `insert_at` (en fonction de `i` et `len(t)`).
3. Écrire une fonction permettant de supprimer un élément à un emplacement arbitraire, en récupérant sa valeur. Les éléments situés à droite seront décalés vers la gauche.

```
int extract_at(int_dynarray* t, int i)
```

Remarque

Les valeurs acceptables pour `i` vont de 0 à $n - 1$ (où n est la longueur du tableau). Si $i = n - 1$, l'opération équivaut à un `pop`.

4. Déterminer la complexité de cette fonction.

Exercice XVI.9 – Une variante du tri insertion

p. 493

On se propose d'écrire une variante du tri insertion sur les `int_dynarray`. Ce tri ne sera pas en place : on renverra un nouveau tableau (trié) sans modifier celui passé en paramètre. L'idée est la suivante, en notant `in` le tableau à trier :

- on crée un tableau vide `out` – tout au long de l'exécution de l'algorithme, ce tableau sera trié;
- pour chaque élément de `in` :
 - on détermine à quelle position de `out` il faut l'insérer pour que `out` reste trié;
 - on effectue l'insertion (à la position déterminée)
- on renvoie le tableau `out`.

1. Écrire une fonction `position(int_dynarray* t, int x)` qui renvoie le plus grand entier `i` tel que l'insertion de `x` en position `i` laisse le tableau `t` trié (en supposant qu'il était trié avant l'appel).

```
int positon(int_dynarray* t, int x);
```

2. Écrire une fonction `insertion_sort(int_dynarray* t)` qui trie un tableau suivant l'algorithme décrit ci-dessus.

```
int_dynarray* insertion_sort(int_dynarray* t);
```

3. Déterminer la complexité de cette fonction (dans le pire cas). On distinguera le nombre de comparaisons entre éléments du tableau effectuées du nombre d'opérations des autres opérations élémentaires.
4. Modifier la fonction `position` pour qu'elle effectue un nombre de comparaisons en $O(\log n)$ (où n est la longueur du tableau dans lequel on insère).
5. Peut-on imaginer une situation où cette amélioration est significative ?

Solutions

Correction de l'exercice XVI.1 page 484

Pour `array_delete`, il faut bien penser à libérer `t` (qui a été alloué sur le tas par `array_create`). Bien sûr, il faut libérer `t->data` avant de libérer `t` (sinon c'est un *use after free*).

```
int array_get(int_array* t, int i){
    assert(0 <= i && i < t->len);
    return t->data[i];
}

void array_set(int_array* t, int i, int x){
    int n = t->len;
    assert(0 <= i && i < t->len);
    t->data[i] = x;
}

void array_delete(int_array* t){
    free(t->data);
    free(t);
}
```

Correction de l'exercice XVI.2 page 485

Pas de grosse difficulté pour les premières fonctions ; la fonction `make_empty` est nettement plus simple que `array_create`.

```
int length(int_dynarray* t){
    return t->len;
}

int_dynarray* make_empty(void){
    int_dynarray* t = malloc(sizeof(int_dynarray));
    t->len = 0;
    t->capacity = 0;
    t->data = NULL;
    return t;
}

int get(int_dynarray* t, int i){
    assert(0 <= i && i < t->len);
    return t->data[i];
}

void set(int_dynarray* t, int i, int x){
    assert(0 <= i && i < t->len);
    t->data[i] = x;
}
```

Pour la fonction `pop`, il faut penser à mettre à jour la longueur :

```
int pop_naif(int_dynarray* t){
    assert(t->len > 0);
    int x = t->data[t->len - 1];
    t->len--;
    return x;
}
```

C'est la fonction `resize` qui demande d'être attentif. Elle a plusieurs responsabilités :

- vérifier que la nouvelle capacité est suffisante (c'est optionnel, puisque cette fonction n'a pas vocation à être publique);
- allouer le nouveau stockage;
- copier le tableau dans le nouveau stockage;
- mettre à jour la capacité;
- libérer l'ancien stockage.

```
void resize(int_dynarray* t, int new_capacity){
    assert(t->len <= new_capacity);
    int* new_data = (int*)malloc(new_capacity * sizeof(int));
    for (int i = 0; i < t->len; i++){
        new_data[i] = t->data[i];
    }
    free(t->data);
    t->data = new_data;
    t->capacity = new_capacity;
}
```

La fonction `push` commence par redimensionner si c'est nécessaire, puis ajoute l'élément.

```
void push_naif(int_dynarray* t, int x){
    if (t->len == t->capacity) {
        resize(t, 1 + t->capacity);
    }
    t->data[t->len] = x;
    t->len++;
}
```

La fonction `delete` est identique à celle écrite plus haut :

```
void delete(int_dynarray* t){
    free(t->data);
    free(t);
}
```

Correction de l'exercice XVI.3 page 486

Une opération de redimensionnement a une complexité temporelle en $\Theta(k)$, où k est la taille du tableau actuel (puisque l'on néglige le coût du `malloc` et qu'il faut recopier les n éléments). Si l'on ne fait que des `push`, il faut redimensionner à chaque fois (avec notre stratégie actuelle). Le coût total est donc en :

$$\Theta\left(\sum_{k=1}^n k\right) = \Theta(n^2)$$

Correction de l'exercice XVI.4 page 486

Pas de difficulté, le cas particulier à ne pas oublier est signalé par l'énoncé.

```
void push(int_dynarray* t, int x){
    if (t->len == t->capacity) {
        if (t->capacity == 0) {
            resize(t, 1);
        } else {
            resize(t, 2 * t->capacity);
        }
    }
    t->data[t->len] = x;
    t->len++;
}
```

Correction de l'exercice XVI.5 page 486

1. Ces trois opérations sont en temps constant.
2. Sans redimensionnement, un push s'effectue en temps constant (c'est le meilleur cas). S'il faut redimensionner, la complexité est en $\Theta(n)$ (n éléments à recopier).
3. On peut majorer brutalement pour commencer :
 - il y a au plus n pop, chacun en temps constant, donc un coût total en $O(n)$ pour ces opérations;
 - il y a au plus n push ne causant pas de redimensionnement, de nouveau en $O(n)$ au total;
 - les redimensionnements se font toujours vers une taille strictement plus grande, et le nombre d'éléments ne peut dépasser n . De plus, les redimensionnements n'ont lieu que quand on passe d'une taille 2^k à une taille 2^{k+1} . On peut donc majorer le coût total par :

$$\sum_{2^k \leq n} 2^k = \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k = 2^{1+\lfloor \log_2 n \rfloor} - 1 = O(n)$$

On obtient une complexité totale (pour l'ensemble des n opérations) en $O(n)$, et donc une complexité amortie de $O(1)$ par opération.

Correction de l'exercice XVI.6 page 487

1. Pas de difficulté :

```
int pop(int_dynarray* t){
    assert(t->len > 0);
    int x = t->data[t->len - 1];
    t->len--;
    if (2 * t->len < t->capacity) {
        resize(t, t->capacity / 2);
    }
    return x;
}
```

2. Le problème est que l'on peut avoir une série de redimensionnements quand on fait osciller la longueur du tableau autour d'une puissance de 2.

Plus précisément, soit $n = 2^k + 1$. On fait :

- 2^{k-1} push successifs, ce qui donne $\text{len} = \text{capacity} = 2^{k-1}$;
- un push supplémentaire, ce qui donne $\text{len} = 2^{k-1} + 1$ et $\text{capacity} = 2^k$;
- ensuite, une série de 2 pop suivis de 2 push (cette série sera de longueur 2^{k-3}).

À chaque fois, le deuxième pop nous fait passer à $\text{len} = 2^{k-1} - 1$ et $\text{capacity} = 2^k$, donc il déclenche un redimensionnement. Ce redimensionnement se fait en $\Theta(2^k)$ et nous fait passer à $\text{len} = 2^{k-1} - 1$ et $\text{capacity} = 2^{k-1}$. Après un push, on a $\text{len} = \text{capacity}$, donc le deuxième déclenche un redimensionnement (temps $\Theta(2^k)$) et l'on se retrouve à $\text{len} = 2^{k-1} + 1$ et $\text{capacity} = 2^k$, c'est-à-dire au point de départ.

Au total, il y aura donc 2^{k-2} redimensionnements, chacun en temps $\Theta(2^k)$: on obtient donc du $\Theta(2^{2k}) = \Theta(n^2)$.

Correction de l'exercice XVI.7 page 487

On considère t_0, \dots, t_{n-1} les états successifs du tableau (t_0 est donc vide), et l'on note C_i le coût de l'opération qui fait passer de t_i à t_{i+1} . On prend toutes les constantes multiplicatives égales à 1 pour les complexités, ce qui est possible sans perte de généralité (on pourrait multiplier le potentiel par une constante), et l'on distingue les cas suivant la nature de la i -ème opération :

- Si c'est un push sans redimensionnement, alors $\Phi(t_{i+1}) = |2 + 2\text{len}(t_i) - \text{capacity}(t_i)| \leq 2 + \Phi(t_i)$ et $C_i = 1$. On a donc :

$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leq 3$$

- Si c'est un push avec redimensionnement, on a nécessairement $\Phi(t_i) = |2\text{len}(t_i) - \text{len}(t_i)| = \text{len}(t_i)$ et $\Phi(t_{i+1}) = 0$. De plus, $C_i = \text{len}(t_i) + 1$ (copie des éléments actuels et ajout du nouvel élément). Donc :

$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leq 1$$

- Si c'est un pop sans redimensionnement, alors $\Phi(t_{i+1}) = |2\text{len}(t_i) - 2 - \text{capacity}(t_i)| \leq 2 + \Phi(t_{i+1})$ et $C_i = 1$, donc :

$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leq 3$$

- Si c'est un pop avec redimensionnement, alors nécessairement $\text{capacity}(t_i) = 4\text{len}(t_i)$ d'où $\Phi(t_i) = 2\text{len}(t_i)$. D'autre part, $\text{len}(t_{i+1}) = \text{len}(t_i) - 1$ et $\text{capacity}(t_{i+1}) = \text{capacity}(t_i)/2 = 2\text{len}(t_i)$. On obtient $\Phi(t_{i+1}) = |2\text{len}(t_i) - 2 - 2\text{len}(t_i)| = 2$. Comme $C_i = \text{len}(t_i)$, le résultat final est :

$$C_i + \Phi(t_{i+1}) - \Phi(t_i) = \text{len}(t_i) + 2 - 2\text{len}(t_i) = 2 - \text{len}(t_i) \leq 2$$

On en déduit

$$\sum_{i=0}^{n-1} (C_i + \Phi(t_{i+1}) - \Phi(t_i)) \leq 3n.$$

Or, d'autre part, on a :

$$\sum_{i=0}^{n-1} (C_i + \Phi(t_{i+1}) - \Phi(t_i)) = \sum_{i=0}^{n-1} C_i + \underbrace{\Phi(t_n)}_{\geq 0} - \underbrace{\Phi(t_0)}_{= 0} \geq \sum_{i=0}^{n-1} C_i$$

On en déduit donc :

$$\sum_{i=0}^{n-1} C_i \leq 3n = O(n)$$

On a donc bien une complexité amortie en $O(1)$ pour pop et push.

Correction de l'exercice XVI.8 page 487

- On respecte bien la consigne, en ne regardant pas sous le capot de `t`. Il faut donc commencer par faire un `push` (avec une valeur quelconque, qui sera de toute façon écrasée), puis décaler les éléments pour faire une place et enfin insérer l'élément.

```
void insert_at(int_dynarray* t, int i, int x){
    int n = length(t);
    assert(0 <= i && i <= n);
    push(t, x);
    for (int j = n; j > i; j--){
        set(t, j, get(t, j - 1));
    }
    set(t, i, x);
}
```

L'assertion au début de la fonction n'est en fait pas strictement nécessaire :

- si $i < 0$, le dernier passage dans la boucle se fait pour $j = i+1 \leq 0$, donc le `get(t, j - 1)` va forcément lever une assertion (ce qui termine l'exécution du programme);
 - si $i > n$, le dernier `set(t, i, x)` va lever une assertion, puisque `length(t)` vaut n à cet instant.
- En notant n la longueur de `t`, on fait un nombre d'opérations proportionnel à $n - i$, et chacune de ces opérations est en temps constant. La complexité est donc en $O(n - i)$.
 - Le principe est similaire : on récupère la valeur à renvoyer, on décale tout vers la gauche, en on finit par un `pop` (dont on ignore la valeur de retour) pour mettre à jour la taille du tableau.

```
int pop_at(int_dynarray* t, int i){
    int n = length(t);
    int x = get(t, i);
    for (int j = i; j < n - 1; j++){
        set(t, j, get(t, j + 1));
    }
    pop(t);
    return x;
}
```

- Même idée : la complexité est en $O(n - i)$.

Correction de l'exercice XVI.9 page 488

- La fonction est marginalement plus simple à écrire en partant de la gauche du tableau, mais partir de la droite permet d'obtenir un tri s'exécutant en temps linéaire si le tableau est déjà trié (ou presque).

```
int position_linear(int_dynarray* t, int x){
    int i = length(t) - 1;
    while (i >= 0 && get(t, i) > x){
        i--;
    }
    return i + 1;
}
```

- Pas de difficulté.

```
int_dynarray* insertion_sort(int_dynarray* t){
    int_dynarray* out = make_empty();
    int n = length(t);
    for (int i = 0; i < n; i++){
        int x = get(t, i);
        int pos = position_linear(out, x);
        insert_at(out, pos, x);
    }
    return out;
}
```

3. Les deux fonctions s'exécutent d'autant plus lentement que l'élément doit être inséré vers la gauche du tableau. Si le tableau initial est trié par ordre (strictement) décroissant, toutes les insertions se feront au début, ce qui est donc le pire cas. Il y aura alors $i - 1$ comparaisons pour trouver l'emplacement du i -ème élément, puis $\Theta(i)$ opérations pour l'insérer. Au total, on a dans le pire cas $\Theta(n^2)$ comparaisons et $\Theta(n^2)$ opérations de lecture/écriture dans le tableau.
4. On peut chercher la position d'insertion de manière dichotomique : la fonction ci-dessous a exactement la même spécification que `position_linear` mais ne fait que $O(\log(\text{length}(t)))$ comparaisons (et opérations).

```
int position(int_dynarray* t, int x){
    int start = 0;
    int end = length(t);
    // Invariant :
    //   - les éléments d'indice  $\geq end$  sont  $> x$ 
    //   - les éléments d'indice  $< start$  sont  $\leq x$ 
    while (end > start){
        int mid = start + (end - start) / 2;
        int xmid = get(t, mid);
        if (x  $\geq$  xmid){
            start = mid + 1;
        } else {
            end = mid;
        }
    }
    return start;
}
```

5. Avec cette fonction, le tri insertion effectue $O(n \log n)$ comparaisons et $O(n^2)$ opérations élémentaires sur les tableaux. Dans certains cas, le coût d'une comparaison peut être nettement supérieur à celui d'une opération élémentaire sur les tableaux : par exemple, si l'on trie un tableau de pointeurs vers des objets compliqués, pour lesquels la comparaison est très coûteuse. La nouvelle version peut alors avoir un comportement qui ressemble à du $O(n \log n)$ même pour des valeurs de n assez grandes (10 000 par exemple).

EXCEPTIONS EN OCAML

I Principe des exceptions

Comme beaucoup de langages, OCaml dispose d'un mécanisme pour gérer les erreurs (dans un sens très large) appelé *exceptions*. Le principe de base des exceptions est le suivant :

- une exception est générée lors de l'évaluation d'une certaine expression ;
- cette exception peut être rattrapée (c'est-à-dire gérée) par la fonction dans laquelle elle s'est produite (bloc `try ... with`) ;
- si ce n'est pas le cas, l'exécution de la fonction se termine et l'exception est transmise à la fonction appelante, qui peut à son tour la rattraper (toujours par un bloc `try ... with`) ;
- on continue ainsi à remonter la pile d'appels ; si l'on arrive au bout de la pile sans que personne n'ait géré l'exception, l'exécution du programme se termine et l'exception est signalée à l'utilisateur.

Remarques

- « Beaucoup de langages », mais pas le C. Il existe un mécanisme en C qui permet de construire quelque chose d'essentiellement équivalent (sans syntaxe particulière en revanche) : `setjmp/longjmp`. C'est complètement (mais alors vraiment complètement) hors-programme.
- Quand on dit que « l'exécution de la fonction se termine », cela signifie qu'on sort complètement du flot de contrôle normal : tous les blocs d'activation qui sont traversés sans que l'exception ne soit gérée sont dépilés.

Supposons par exemple que l'on souhaite écrire une fonction `somme_inverses` : `int array -> int` renvoyant la somme des inverses des éléments d'un tableau. Logiquement, on procéderait ainsi :

```
let somme_inverses t =
  let s = ref 0 in
  for i = 0 to Array.length t - 1 do
    s := !s + 1 / t.(i)
  done;
  !s
```

Évidemment, si l'un des éléments est nul, une exception sera levée :

```
utop[54]> somme_inverses [| 12; 0; 1; 3 |];
Exception: Division_by_zero.
Raised by primitive operation at unknown location
Called from file "toplevel/toploop.ml", line 180, characters 17-56
```

Si l'on décide arbitrairement de renvoyer `max_int` dans le cas où l'un au moins des éléments est nul, on peut utiliser un bloc `try ... with` :

```
1 let somme_inverses t =
2   let s = ref 0 in
3   try
4     for i = 0 to Array.length t - 1 do
5       s := !s + 1 / t.(i)
6     done;
7     !s
8   with
9   | Division_by_zero -> max_int
```

Dans ce cas, dès que l'un des éléments vaut 0, l'exception **Division_by_zero** est levée ligne 5 et l'on saute à la ligne 9 : comme l'exception rattrapée est bien celle qui a été levée, on renvoie le `max_int` situé à droite de la flèche.

```
utop[57]> somme_inverses [| 12; 0; 1; 3 |];
- : int = 4611686018427387903
```

La partie **with** du bloc **try ... with** fait un filtrage par motif : dans l'exemple ci-dessus, le seul motif accepté est **Division_by_zero**. Si l'exception que l'on essaie de rattraper ne correspond pas au motif (c'est-à-dire, ici, si ce n'est pas *exactement* **Division_by_zero**), elle n'est pas rattrapée et elle est donc transmise à la fonction appelante (comme s'il n'y avait pas eu de **try...with**). Par exemple :

```
let somme_inverses t =
  let s = ref 0 in
  try
    for i = 0 to Array.length t do (* Ligne modifiée *)
      s := !s + 1 / t.(i)
    done;
    !s
  with
  | Division_by_zero -> max_int
```

```
utop[45]> somme_inverses [| 2; 0; 1 |];
Exception: Invalid_argument "index out of bounds".
```

Exercice XVII.I

p. 503

1. Que va-t-il se passer si l'on appelle la fonction `somme_inverses` (modifiée) sur le tableau `[| 2; 0; 1 |]` ?
2. Vérifier en exécutant le code.

Il est tout à fait possible d'avoir plusieurs motifs dans le **with**. Par exemple, dans le code ci-dessous, la fonction renvoie :

- la valeur finale de `!s` si aucune exception n'est levée (ce qui n'arrivera jamais vu l'erreur dans les bornes de la boucle);
- `max_int` si la première exception levée dans le bloc **try** est **Division_by_zero**;
- deux fois la valeur de `!s!` (au moment où l'exception est levée) si la première exception levée est **Invalid_argument** `x` pour un certain `x`.

Si jamais la première exception levée n'est pas l'une des deux qui sont rattrapées dans le **with**, elle est transmise à la fonction appelante.

```
let somme_inverses t =
  let s = ref 0 in
  try
    for i = 0 to Array.length t do (* Ligne modifiée *)
      s := !s + 1 / t.(i)
    done;
    !s
  with
  | Division_by_zero -> max_int
  | Invalid_argument message -> 2 * !s
```

Pour déclencher (on dit *lever*) volontairement une exception, on utilise la fonction **raise**, appliquée à l'exception souhaitée :

```
let mini t =
  let n = Array.length t in
  if n = 0 then raise (Invalid_argument "tableau vide");
  let m = ref t.(0) in
  for i = 1 to n - 1 do
    m := min !m t.(i)
  done;
  !m
```

```
utop[52]> mini [];;
Exception: Invalid_argument "tableau vide".
```

Remarque

Nous avons déjà levé volontairement des exceptions, par deux moyens :

- `failwith message`, qui équivaut à `raise (Failure message)`;
- `assert(condition)`, qui lève l'exception `Assert_failure (f, l, c)` si condition s'évalue à `false` ((`f, l, c`) est un triplet qui indique le fichier, la ligne et le caractère où l'assertion a échoué).

2 Exemple d'utilisation : lecture d'un fichier

La manière la plus simple¹ de lire un fichier texte en OCaml est d'utiliser les fonctions suivantes :

- `open_in : string -> in_channel` qui prend en argument un nom de fichier (c'est-à-dire un chemin vers un fichier) et renvoie une valeur de type `in_channel`;
- `input_line : in_channel -> string` qui lit des caractères dans le `in_channel` jusqu'à tomber sur un retour à la ligne, et renvoie la chaîne de caractères lue (sans le "`\n`");
- `close_in : in_channel -> unit` pour fermer le fichier une fois qu'on a fini de l'utiliser.

Il y a deux remarques importantes :

- une valeur de type `in_channel` se comporte comme un « flux », c'est-à-dire que les lignes sont « consommées » au fur et à mesure qu'elles sont lues (si l'on appelle deux fois de suite `input_line` sur le même `in_channel`, on obtiendra une ligne puis la suivante);
- si on appelle `input_line` sur un `in_channel` « épousé » (dans lequel il n'y a plus rien à lire), l'exception `End_of_file` est levée. C'est en fait le seul moyen de savoir (en utilisant ces fonctions) qu'on a terminé la lecture du fichier.

Exemple XVII.2

Le programme suivant accepte un nombre quelconque d'arguments en ligne de commande, qui doivent tous être des noms de fichier (qui doivent exister). Il les lit ligne par ligne, les uns après les autres, et affiche leur contenu sur la sortie standard.

```
let print_file filename =
  let f = open_in filename in
  let rec loop () =
    try
      let s = input_line f in
      print_endline s;
      loop ()
    with
    | End_of_file -> () in
  loop ();
  close_in f
```

```
let () =
  let n = Array.length Sys.argv in
  (* On commence à 1 puisque
   * Sys.argv.(0) n'est pas un
   * « vrai » argument. *)
  for i = 1 to n - 1 do
    print_file Sys.argv.(i)
  done
```

1. Ce qui ne veut certainement pas dire la meilleure.

Remarque

On pourrait penser qu'un tel programme ne sert à rien, mais en réalité c'est que fait la commande `cat`^a, qui est d'usage très courant. Typiquement, on redirigerait la sortie : `cat part1 part2 part3 > file`.

- a. Elle est un peu plus versatile, mais c'est la manière la plus courante de l'utiliser

Exercice XVII.3

p. 503

1. Écrire une fonction qui accepte un nom de fichier en argument et renvoie le nombre de lignes de ce fichier.
2. Écrire un programme qui accepte des noms de fichier (en nombre quelconque) comme arguments et affiche :
 - une ligne pour chaque fichier, avec le nombre de lignes du fichier puis le nom du fichier ;
 - une dernière ligne pour le total.

```
$ ./lc.out cat.ml lc.ml lecture1.ml
18 cat.ml
22 lc.ml
29 lecture1.ml
69 total
```

Remarques

- C'est ce que fait la commande `wc` (*word count*) appelée avec l'option `--lines`.
- Si l'on utilise une *wildcard* comme `*`, le *shell* fait l'expansion avant de passer les arguments au programme. Ça a l'air compliqué, mais ça veut dire que si l'on appelle `./lc.out *.ml`, c'est comme si on avait passé *tous les fichiers du répertoire dont le nom se termine par .ml comme argument*.

```
$ wc --lines .../*.tex
178 ../exceptions.tex
18 ../td-exceptions.tex
196 total
$ ./lc.out .../*.tex
178 ../exceptions.tex
18 ../td-exceptions.tex
196 total
```

- Les plus attentifs auront remarqué que la sortie de `wc --lines` est plus joliment formatée dans l'exemple ci-dessus. S'ils devaient dans les jours prochains se trouver confrontés à une violente crise d'ennui, ils pourraient éventuellement chercher à régler ce problème, en commençant sans doute par chercher comment fixer la largeur d'un champ dans `Printf.printf`.

3 Utilisation dans le *flot de contrôle*

Comme nous l'avons vu à la partie précédente, les exceptions ne servent pas *que* à gérer les « erreurs » : quand on appelle `input_line`, on s'attend bien à déclencher l'exception `End_of_file` à un moment ou un autre. On peut en fait aller plus loin et utiliser les exceptions pour manipuler le *flot de contrôle* (le chemin suivi par l'exécution).

Remarque

OCaml a été conçu pour que les exceptions soient rapides (à lever et à récupérer), ce qui rend raisonnables (et même relativement idiomatiques) les techniques présentées dans cette partie. Dans la plupart des langages, ce serait catastrophique (mais ces langages proposent généralement des mots-clés tels que `break` et `return`, et les techniques présentées y sont donc bien moins utiles).

Exemple XVII.4

Supposons qu'on souhaite tester la présence d'un élément `x` dans un tableau `t` (non trié). Pour l'instant, nous avons vu trois manières de procéder :

- faire une boucle `for` qui parcourt systématiquement le tableau jusqu'au bout :

```
let appartient_for x t =
  let trouve = ref false in
  for i = 0 to Array.length t - 1 do
    if t.(i) = x then trouve := true
  done;
!trouve
```

- faire une boucle `while` pour s'arrêter dès qu'on a trouvé l'élément (deux variantes) :

```
let appartient_while_1 x t =
  let n = Array.length t in
  let trouve = ref false in
  let i = ref 0 in
  while !i < n && not !trouve do
    if t.(!i) = x then trouve := true;
    i := !i + 1;
  done;
!trouve
```

```
let appartient_while_2 x t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n && t.(!i) <> x do
    i := !i + 1
  done;
!i < n
```

- utiliser une fonction auxiliaire récursive pour s'arrêter dès qu'on a trouvé l'élément :

```
let appartient_rec x t =
  let rec loop i =
    if i = Array.length t then false
    else t.(i) = x || loop (i + 1) in
  loop 0
```

En utilisant une exception, il y a une possibilité supplémentaire : emballer une boucle **for** dans un bloc **try...with** et lever une exception dès qu'on trouve l'élément. Dans ce cas, il est **fortement recommandé** de définir une nouvelle exception :

```
exception Trouve

let appartient_exn x t =
  try
    for i = 0 to Array.length t - 1 do
      if t.(i) = x then raise Trouve
    done;
    false
  with
  | Trouve -> true
```

Remarque

Ici, je ne recommande pas du tout la version qui utilise une exception, inutilement lourde. On préférera :

- la version avec boucle **for** si parcourir la liste jusqu'au bout ne pose pas de problème ;
- une version avec **while** ou la version récursive sinon.

Une exception peut être paramétrée par un objet d'un certain type : c'est par exemple le cas de **Failure** et de **Invalid_argument**, qui sont toutes deux paramétrées par une chaîne de caractères. Pour définir une nouvelle exception paramétrée par un entier, par exemple, on écrirait :

```
exception Mon_exception of int
```

Exercice XVII.5

p. 503

On souhaite écrire une fonction `premiere_occ` : `'a -> 'a array -> int option` telle que l'appel `premiere_occ x t` renvoie :

- **Some** `i` si `i` est la première occurrence de `x` dans `t` ;
- **None** si `x` n'apparaît pas dans `t`.

Écrire trois versions de cette fonction :

1. l'une utilisant une boucle **while** ;
2. l'une utilisant une fonction auxiliaire récursive ;
3. l'une utilisant une boucle **for** et un bloc **try...with**.

Remarque

À nouveau, on préférera l'une des deux premières solutions ici.

Exercice XVII.6 – Un exemple où ça a un intérêt, pour changer...

p. 504

On considère une matrice `M` (représentée sous la forme d'un `m : int array array`) et un entier `x`. On souhaite trouver la première occurrence de `x` dans `M`, où « première » est à comprendre dans l'ordre lexicographique sur les couples d'indices :

$$(i, j) \preceq (i', j') \iff \begin{cases} i < i' \\ \text{ou} \\ i = i' \text{ et } j \leq j' \end{cases}$$

Autrement dit, il s'agit de trouver l'occurrence le plus en « haut » de la matrice, et en cas d'égalité la plus à « gauche ». Pour pouvoir traiter le cas où `x` n'apparaît pas dans la matrice, on veut écrire une fonction ayant le type suivant :

```
cherche_matrice : int -> int array array -> (int * int) option
```

Sachant que l'on souhaite arrêter la recherche dès que possible, écrire deux versions de cette fonction :

1. l'une à base de boucles **while**;
2. l'une utilisant des boucles **for** et une exception.

4 Rien à voir

Exercice XVII.7 – D'après France-IOI

p. 505

Des affiches rectangulaires sont collées les unes après les autres sur un mur. Toutes les affiches ont la même largeur. On vous donne la hauteur H_i de chacune de ces affiches, dans l'ordre dans lequel elles sont collées. Le coin supérieur gauche de chaque affiche est toujours placé exactement sur le coin supérieur gauche du mur, et ce dernier est toujours plus grand que les affiches. Régulièrement, entre deux collages d'affiches, on vous demande d'indiquer combien d'affiches, parmi celles déjà collées, sont au moins en partie visibles, c'est à dire qu'une surface non nulle n'a été recouverte par aucune affiche collée depuis.

Limites de temps et de mémoire

Temps 2s sur une machine à 1 GHz.

Mémoire 32 Mo.

Garanties

- $1 \leq nbRequetes \leq 100\,000$;
- $1 \leq H_i \leq 1\,000\,000$ où les H_i sont les hauteurs des affiches.

Entrée

- La première ligne de l'entrée est constituée d'un unique entier : $nbRequetes$.
- Chacune des $nbRequetes$ lignes suivantes commence par un caractère pouvant être '**Q**' ou '**C**'. Un caractère '**Q**' correspond à la question : « Combien d'affiches sont actuellement visibles ? ». Un caractère '**C**' est suivi d'un entier hauteur sur la même ligne, séparé par un espace, et correspond au collage d'une affiche de hauteur hauteur.

Sortie La sortie de votre programme doit correspondre aux réponses aux questions posées en entrée, à raison d'une réponse par ligne, dans l'ordre d'apparition des questions.

Entrée	Sortie	Entrée	Sortie
12	1	10	0
C 2	2	Q	1
Q	1	C 8	3
C 4	2	C 7	
C 2	2	C 11	
Q		Q	
C 9		C 2	
Q		C 4	
C 9		C 3	
C 2		Q	
Q		C 3	
C 8			
Q			

1. En considérant les limites de ressources et les garanties sur les entrées données, en notant $n = nbRequetes$:

- a. une complexité en $\Theta(n^2)$ est-elle satisfaisante ?
 - b. même question pour $\Theta(n \log n)$ et $\Theta(n)$;
 - c. peut-on stocker les hauteurs de toutes les affiches ?
2. Essayer de réfléchir à un algorithme efficace pour résoudre ce problème. Ce n'est pas évident^a, donc il ne faut pas hésiter à me demander une indication.
3. Implémenter votre solution en C, ou en OCaml, ou, mieux, dans les deux langages. Vous aurez sans doute besoin d'aide pour les entrées-sorties en OCaml, n'hésitez pas à me demander.
- a. En plus d'être un problème de niveau 3 de France-IOI, c'est le début d'un oral d'informatique de Ulm...

Solutions

Correction de l'exercice XVII.1 page 496

On obtient `max_int` : la première exception levée est une `Division_by_zero`, qui est rattrapée.

Correction de l'exercice XVII.3 page 498

On donne directement le programme complet, qu'on a choisi d'écrire avec une boucle `while true` plutôt qu'une fonction récursive, pour changer :

```
let count_lines filename =
  let f = open_in filename in
  let n = ref 0 in
  try
    while true do
      let _ = input_line f in
      incr n
    done;
    -1 (* on n'arrivera jamais ici *)
  with
  | End_of_file -> close_in f; !n

let () =
  let total = ref 0 in
  let nb_args = Array.length Sys.argv - 1 in
  for i = 1 to nb_args do
    let filename = Sys.argv.(i) in
    let n = count_lines filename in
    Printf.printf "%d %s\n" n filename;
    total := !total + n
  done;
  if nb_args > 1 then Printf.printf "%d total\n" !total
```

Correction de l'exercice XVII.5 page 500

1.

```
let premiere_occ x t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n && t.(!i) <> x do
    incr i
  done;
  if !i < n then Some !i else None
```

2.

```

let premiere_occ x t =
  let rec aux i =
    if i = Array.length t then None
    else if t.(i) = x then Some i
    else aux (i + 1)
  aux 0

```

3.

```

exception Trouve of int

let premiere_occ x t =
  try
    for i = 0 to Array.length t - 1 do
      if t.(i) = x then raise (Trouve i)
    done;
    None
  with
    | Trouve i -> Some i

```

Correction de l'exercice XVII.6 page 500

1. Pas très agréable à écrire, mais si on s'y prend bien ça reste acceptable :

```

let cherche_matrice x m =
  let n = Array.length m in
  let res = ref None in
  let i = ref 0 in
  while !i < n && !res = None do
    let j = ref 0 in
    while !j < Array.length m.(!i) && !res = None do
      if m.(!i).(!j) = x then res := Some (!i, !j);
      incr j
    done;
    incr i
  done;
  !res

```

2. Moins de chance de se tromper, à condition de savoir manipuler des exceptions :

```

exception Trouve of int * int

let cherche_matrice x m =
  try
    for i = 0 to Array.length m - 1 do
      for j = 0 to Array.length m.(i) - 1 do
        if m.(i).(j) = x then raise (Trouve (i, j))
      done
    done;
    None
  with
    | Trouve (i, j) -> Some (i, j)

```

Correction de l'exercice XVII.7 page 501

1. Avec $n = 10^5$, on a $n^2 = 10^{10} = 10$ milliards. On est censé répondre en moins de deux secondes sur une machine à 1 GHz (un milliard d'opérations par seconde, pour simplifier) : aucune chance.
 $n \log n$ sera de l'ordre de 1 million pour $n = 10^5$ donc une solution avec cette complexité devrait sans doute pouvoir être acceptée, mais ce n'est pas sûr.
Une complexité en $\Theta(n)$ offre pas mal de marge.
2. En OCaml, en utilisant explicitement une pile contenant les hauteurs des affiches actuellement visibles :

```

let rec ajoute x pile visibles =
  match pile with
  | [] -> [x], 1
  | y :: ys ->
    if x < y then (x :: pile, visibles + 1)
    else ajoute x ys (visibles - 1)

let rec loop pile visibles =
  let s = read_line () in
  if s = "Q" then begin
    Printf.printf "%d\n" visibles;
    loop pile visibles
  end else begin
    (* on ignore le caractère et on extrait l'entier *)
    let h = Scanf.sscanf s "%c %d" (fun _ x -> x) in
    let nv_pile, nv_visibles = ajoute h pile visibles in
    loop nv_pile nv_visibles
  end

let () =
  let _ = read_int () in
  try
    loop []
  with
  | End_of_file -> ()

```

En C, en faisant essentiellement la même chose mais sans pile explicite :

```
#include <stdio.h>

#define TAILLE_MAX 100 * 1000

int prec[TAILLE_MAX];

int tailles[TAILLE_MAX];

int maj(int taille, int i, int visibles){
    int x = i - 1;
    tailles[i] = taille;
    visibles++;
    while (x >= 0 && tailles[x] <= taille){
        x = prec[x];
        visibles--;
    }
    prec[i] = x;
    return visibles;
}

int main(void){
    int nbRequetes;
    int iRequete = 0;
    int iAffiche = 0;
    int visibles = 0;
    int taille;
    char c;
    scanf("%d\n", &nbRequetes);
    while (iRequete + iAffiche < nbRequetes){
        scanf("%c", &c);
        if (c == 'C'){
            scanf("%d\n", &taille);
            visibles = maj(taille, iAffiche, visibles);
            iAffiche++;
        } else {
            scanf("\n");
            printf("%d\n", visibles);
            iRequete++;
        }
    }
    return 0;
}
```

PROGRAMMATION D'UN ALLOCATEUR MÉMOIRE

Le but de ce TP est d'écrire un allocateur dynamique de mémoire utilisant un tableau alloué en début de programme. Afin d'éviter de manipuler différents types de pointeurs, notre allocateur ne renverra que des pointeurs de type `uint64_t*` (pointeurs vers des blocs d'entiers non signés de 64 bits). La signature de notre allocateur sera donc :

```
uint64_t *malloc_ui(uint64_t size);
void free_ui(uint64_t *p);
```

La fonction `malloc_ui` prend en paramètre une taille `size` et renvoie un pointeur `p` de type `uint64_t*` vers une portion mémoire pouvant accueillir `size` entiers `uint64_t`. Si le tas ne dispose plus de place, elle renvoie le pointeur `NULL`. La fonction `free_ui` est utilisée avec un pointeur non nul renvoyé par `malloc_ui`. Son rôle est de libérer la mémoire.

Afin de gérer notre tas, nous utiliserons une constante `HEAP_SIZE` et un tableau statique `heap` :

```
#define HEAP_SIZE 32
uint64_t heap[HEAP_SIZE];
```

Enfin, afin d'initialiser une zone mémoire obtenue *via* un appel à `malloc_ui`, le « client » pourra utiliser la fonction suivante :

```
void set_memory(uint64_t *p, uint64_t n, uint64_t value) {
    for (uint64_t i = 0; i < n; i++) {
        p[i] = value;
    }
}
```

Remarques générales

- Comme `heap` est une variable globale, elle est initialisée à zéro (toutes les cases sont nulles). Cependant, nous ferons comme si son contenu initial était imprévisible.
- Dans tout le sujet, quand on parle du « bloc d'indice `i` » ou de la « zone mémoire d'indice `i` » ou toute expression équivalente, on veut dire que la première case *accessible par l'utilisateur* porte l'indice `i` (ou précédemment accessible, si la zone a depuis été libérée). Dans la plus grande partie du sujet, il y aura des informations relatives à cette zone stockées dans la case `i - 1`.
- On suppose que l'utilisateur ne fait que des appels licites à nos fonctions. Plus précisément :
 - il n'appelle `malloc_ui` qu'avec des tailles strictement positives;
 - il n'appelle `free_ui` que sur des pointeurs issus directement de `malloc_ui` (et une seule fois sur un pointeur donné);
 - après avoir récupéré un pointeur `p` par `malloc_ui(size)`, il n'effectue que des accès licites (entre `p[0]` et `p[size - 1]`);
 - après un appel à `free`, il n'accède plus à la mémoire libérée.

Remarque

Vous trouverez un certain nombre de remarques dans l'énoncé. Elles expliquent pourquoi certaines stratégies étudiées peuvent être intéressantes, même si elles ne sont pas satisfaisantes pour implémenter le couple `malloc / free`. Typiquement, elles peuvent être utilisées soit *en plus* d'un allocateur général (allocateur linéaire), soit pour gérer rapidement certains types d'allocation dans le cadre d'un allocateur général (allocation de blocs de taille fixe). Ces remarques peuvent être sautées lors d'une première lecture¹.

1. Remarque : la remarque à laquelle cette remarque fait référence ne s'applique pas à elle-même.

I Allocateur linéaire

Dans cette partie, nous organisons notre mémoire heap comme une suite contiguë de portions mémoire entre les indices 1 et `heap_size` - 1. La case `heap[0]` joue un rôle spécifique : elle contient un entier `end` tel que les portions réservées se trouvent parmi les cases d’indices 1, …, `end` - 1. Lors de la prochaine allocation, la nouvelle portion sera placée à partir de l’indice `end`. Dans cette stratégie naïve, la libération de mémoire n’a pas d’effet sur le tas. Comme on ne libère jamais, la fonction `free_ui` est triviale :

```
void free_ui(uint64_t *p) {}
```

Afin d’initialiser notre structure, c’est-à-dire la première case du tableau `heap`, on écrira une fonction de signature

```
void init_heap(void);
```

à la question 27.

Notre code pourra donc être utilisé de la manière suivante. On crée ici 2 tableaux d’entiers de taille respective 6 et 5 que l’on initialise respectivement avec les valeurs 42 et 52. On libère enfin ces tableaux.

```
init_heap();
uint64_t *p1 = malloc_ui(6);
uint64_t *p2 = malloc_ui(5);
```

```
set_memory(p1, 6, 42);
set_memory(p2, 5, 52);
free_ui(p2);
free_ui(p1);
```

Après initialisation de le mémoire, le tableau `heap` est dans l’état suivant. Dans toute la suite de ce TP, une case vide signalera soit une case mémoire qui n’est pas initialisée, soit une case mémoire dont la valeur ne nous intéresse plus.

12	42	42	42	42	42	42	42	52	52	52	52	52				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	

► **Question 27** Écrire la fonction `init_heap` qui initialise le tas pour que `heap[0]` contienne une valeur appropriée pour cette stratégie.

► **Question 28** Écrire la fonction `malloc_ui`, et déterminer sa complexité. Cette fonction renverra le pointeur nul (`NULL`) si la requête ne peut être satisfaite.

Remarque

On parle d’*allocateur linéaire* pour un allocateur qui utilise cette méthode. C’est bien sûr complètement inadapté dans le cas général (la mémoire n’étant jamais libérée...), mais cela a l’avantage d’être extrêmement rapide. En complément d’un `malloc` « général », cela peut être très intéressant. Un exemple typique est celui d’un ensemble d’objets temporaires qui vont tous cesser d’être utiles en même temps : dans ce cas, le bloc complet pourra être libéré en une seule fois par un seul appel au « vrai » `free`.

2 Réservations de blocs de taille fixe

Nous cherchons désormais à permettre la réutilisation de la mémoire libérée. Nous proposons pour cela une nouvelle stratégie d’implémentation. Nous fixons une variable globale `block_size` et nous allouerons systématiquement des blocs de taille `block_size` : si on nous demande un bloc plus petit, il y aura de la mémoire non utilisée, et si l’on nous demande un bloc plus grand, l’allocation échouera.

```
uint64_t block_size = 8;
```

Une portion mémoire i réservée avec la taille n (où $n + 1 \leq \text{block_size}$) occupe $n + 1$ cases d’un tel bloc. L’en-tête $\text{heap}[i - 1]$ vaut 1 si la portion est encore réservée, ou 0 si elle a été libérée. Les cases suivantes sont utilisées pour stocker les données. Comme précédemment, la case $\text{heap}[0]$ est réservée pour donner l’indice i de la prochaine portion libre pour créer un bloc lorsqu’aucun recyclage de bloc libéré n’est possible.

Une fois le tas initialisé et les opérations suivantes effectuées

```
uint64_t *p1 = malloc_ui(6);
uint64_t *p2 = malloc_ui(3);
```

```
set_memory(p1, 6, 42);
set_memory(p2, 3, 52);
```

l’état du tas est donné par le tableau ci-dessous. Le pointeur $p1$ pointe vers la portion mémoire d’indice $i = 2$ tandis que le pointeur $p2$ pointe vers la portion mémoire d’indice $i = 10$.

18	1	42	42	42	42	42	42		1	52	52	52			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

► **Question 29** Écrire la fonction `init_heap` pour cette stratégie.

► **Question 30** Écrire une fonction `is_free` renvoyant `true` si la portion mémoire i est libre, et `false` sinon. Créez de même les fonctions `set_free` et `set_used` permettant de changer l’en-tête de la portion mémoire d’indice i .

```
bool is_free(uint64_t i);
void set_free(uint64_t i);
void set_used(uint64_t i);
```

Pour réserver une nouvelle portion, on cherche en priorité à réutiliser un bloc laissé libre par une précédente libération. La portion libre pointée par $\text{heap}[0]$ n'est utilisée que si un tel bloc n'existe pas.

► **Question 31** Écrire la fonction `malloc_ui` pour cette stratégie d’implémentation, et déterminer sa complexité. On renverra `NULL` lorsque la taille n est trop grande vis-à-vis de `block_size`.

► **Question 32** Écrire la fonction `free_ui` pour cette stratégie d’implémentation, et déterminer sa complexité. Si possible (c'est-à-dire si la zone libérée est là « plus à droite »), on mettra à jour $\text{heap}[0]$, sinon on marquera simplement la zone comme étant libre.

Remarques

- Pour un « vrai » couple `malloc/free`, la taille du tas varie au cours de l’exécution du programme : quand il n’y a pas de zone libre suffisamment grande pour satisfaire la requête, `malloc` réclame de la mémoire au système d’exploitation, ce qui se traduit par un déplacement de la limite droite du tas. Quand de la mémoire est libérée par `free`, elle est à nouveau disponible pour l’application (évidemment), mais elle n'est en général « rendue » au système d’exploitation, sauf s'il s'agit de la zone la plus à droite. Essentiellement, les mises à jour de $\text{heap}[0]$ correspondent ici à ces interactions avec le système d’exploitation :
 - incrémenter cette valeur revient à réclamer davantage de mémoire à l’OS;
 - décrémenter cette valeur revient à rendre de la mémoire à l’OS.
- Notre allocateur est très limité puisqu’il ne gère qu’une seule taille d’allocation. Une telle stratégie ne serait bien sûr jamais utilisée seule, mais elle peut être utile pour gérer efficacement les petites allocations (on utiliserait quand même une technique permettant d’accélérer la recherche d’une zone libre).

3 Zones mémoire avec en-tête et pied de page

Nous abordons maintenant une autre stratégie d'implémentation qui permettra de ne pas limiter autant la taille de chaque portion mémoire. Nous munissons pour cela chaque portion d'une en-tête mais aussi d'un *pied de page*. Pour chaque portion, ces deux cases additionnelles contiennent la même valeur : un entier encodant deux informations sur la portion courante. La première information indique si la portion est réservée ou pas. La deuxième indique la taille réservée à cette portion. Nous imposons que cette taille soit toujours un entier pair. Si un programmeur demande à réserver une zone de taille n impaire, nous attribuerons une taille $n + 1$ à cette zone, mais le programmeur n'est pas autorisé à consulter cet espace supplémentaire. Si la taille de la portion est $2k$ et la portion est utilisée, l'en-tête et le pied de page contiendront la valeur $2k + 1$. Si la portion est libre, ils contiendront la valeur $2k$. Pour une zone d'indice i et de taille $2k$, l'en-tête sera donc située en $i - 1$ et le pied de page en $i + 2k$.

Nous utiliserons deux portions spéciales, une portion *prologue* et une portion *épilogue*. Ces deux portions spéciales sont de taille nulle et toujours marquées réservées. Nous les plaçons en début et en fin de la zone de réservation. La portion prologue se situe à une position fixée donnée par la variable globale suivante.

```
uint64_t prologue = 2;
```

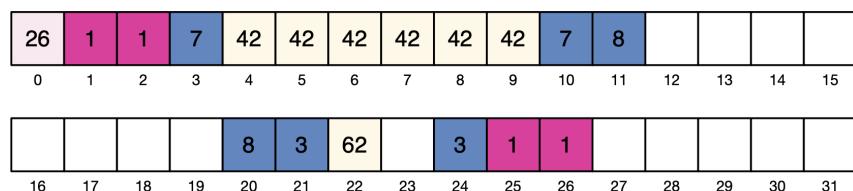
La case `heap[0]` indique l'indice de l'épilogue. L'épilogue est contigu au prologue au démarrage, puis est déplacé vers des indices plus élevés quand la zone des portions réservées doit être agrandie, ou vers des indices plus faibles lorsque cette zone rétrécit.

Après initialisation du tas et appels suivants par le programmeur

```
uint64_t *p1 = malloc_ui(6);
uint64_t *p2 = malloc_ui(7);
uint64_t *p3 = malloc_ui(1);
```

```
set_memory(p1, 6, 42);
set_memory(p2, 7, 52);
set_memory(p3, 1, 62);
free ui(p2);
```

la figure suivante présente le contenu de heap.



► **Question 33** Écrire les fonctions `is_free` et `read_size` permettant de savoir si une portion mémoire `i` est libre ou non et de connaître sa taille.

► **Question 34** Écrire les fonctions `set_free` et `set_used` permettant de définir l'en-tête et le pied de page d'une portion de mémoire `i` et de fixer sa taille à `size`.

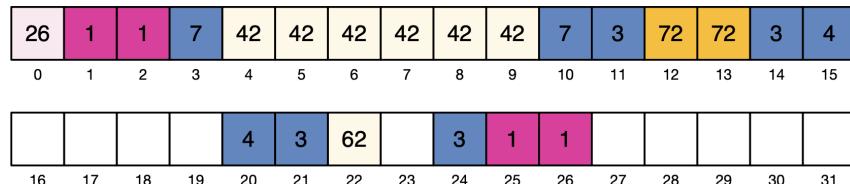
► **Question 35** Écrire les fonctions `next` et `previous` permettant respectivement d'obtenir l'indice de la portion mémoire suivante et de la portion mémoire précédente. Ces fonctions renverront respectivement les indices du prologue et de l'épilogue lorsque la portion mémoire `i` est la première ou la dernière zone de notre tas.

```
bool is_free(uint64_t i);
uint64_t read_size(uint64_t i);
void set_free(uint64_t i, uint64_t size);
void set_used(uint64_t i, uint64_t size);
uint64_t next(uint64_t i);
uint64_t previous(uint64_t i);
```

► Question 36 Écrire la fonction init_heap pour cette stratégie.

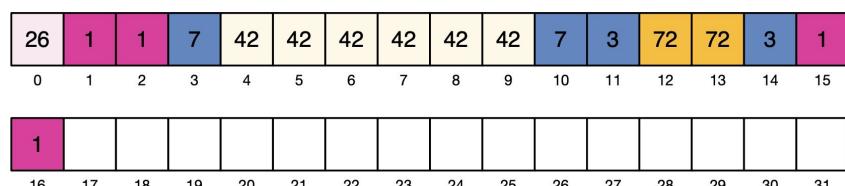
À part les deux portions spéciales épilogue et prologue, toutes les portions ont une taille strictement positive. Lors de la réservation d'une nouvelle portion, on réserve en priorité dans la zone mémoire comprise entre le prologue et l'épilogue, et en dernier recours, on déplace l'épilogue. Si une portion libre est suffisamment grande, une réservation dans cette zone la sépare en une portion réservée et une portion libre. La figure suivante illustre ce mécanisme en présentant le contenu de la mémoire de la figure précédente après l'appel suivant :

```
uint64_t *p4 = malloc_ui(2);
set_memory(p4, 2, 72);
```



► **Question 37** Écrire la fonction `malloc_uit` pour cette stratégie, et déterminer sa complexité.

Lors de la libération d'une portion, on étudie les portions adjacentes libres et on réalise si possible une fusion afin qu'il n'y ait jamais deux portions adjacentes libres après un appel à la fonction `free_ui`. De plus, si possible, on déplace l'épilogue vers la gauche. La figure ci-dessous illustre ce mécanisme en présentant le contenu de `heap` après l'appel de `free_ui(p3)`.



► **Question 38** Écrire la fonction `free_ui` pour cette stratégie et déterminer sa complexité. Expliquer l'intérêt des portions prologues et épilogues.

► **Question 39** Pourquoi est-il important de fusionner les zones libres adjacentes ? Cela suffit-il à régler entièrement le problème ?

► **Question 40** La stratégie d'allocation utilisée ici est dite *first fit* : on utilise la première zone libre suffisamment grande pour satisfaire notre allocation. Une autre stratégie courante est dite *best fit* : on utilise la *plus petite* zone libre suffisamment grande pour satisfaire notre allocation. Quels sont les avantages et inconvénients d'une telle stratégie?

4 Chaînage explicite des portions libres

Nous souhaitons maintenant améliorer l'implémentation de la partie précédente pour accélérer la recherche de portions libérées. Nous allons pour cela maintenir une *chaîne des portions libres*. Il s'agit d'une séquence de portions libres organisée de la manière suivante.

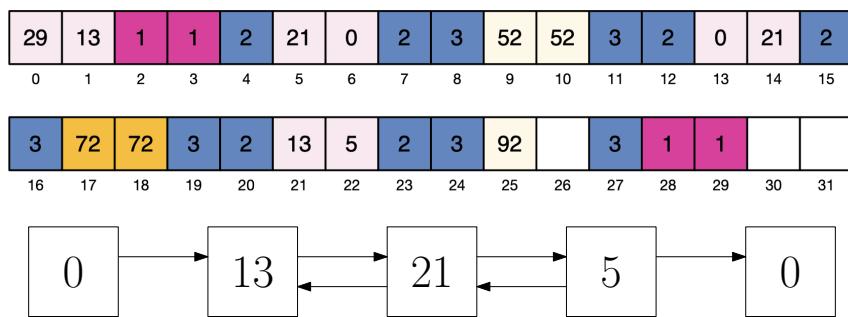
- Dans chaque portion libre i dans la chaîne, on stocke une information dans les cases $\text{heap}[i]$ et $\text{heap}[i + 1]$.
 - La case $\text{heap}[i]$ contient l'indice de la portion libre prédecesseur dans la chaîne.
 - La case $\text{heap}[i + 1]$ contient l'indice de la portion libre successeur dans la chaîne.
 - L'indice de l'entrée de la chaîne est stockée dans la case $\text{heap}[1]$. Par convention, elle vaut 0 si et seulement si la chaîne est vide. Si la chaîne n'est pas vide, son premier élément est une portion dont le prédecesseur vaut 0. De même, elle contient un dernier élément, éventuellement égal au premier, dont le successeur vaut 0. Toutes les autres portions de la chaîne ont des prédecesseurs et successeurs non nuls.

Après initialisation du tas et appels suivants par le programmeur

```
uint64_t *p1 = malloc_ui(2);
uint64_t *p2 = malloc_ui(2);
uint64_t *p3 = malloc_ui(2);
uint64_t *p4 = malloc_ui(2);
uint64_t *p5 = malloc_ui(1);
uint64_t *p6 = malloc_ui(1);
```

```
set_memory(p1, 2, 42);
set_memory(p2, 2, 52);
set_memory(p3, 2, 62);
set_memory(p4, 2, 72);
set_memory(p5, 1, 82);
set_memory(p6, 1, 92);
free_ui(p1);
free_ui(p5);
free_ui(p3);
```

la figure suivante présente le contenu de heap, ainsi que la chaîne des zones libres :



► **Question 41** Écrire la fonction `add_begin_chain` qui ajoute la portion libre `i` en tête de la chaîne et en fait son premier élément, la portion `i` n’appartenant pas à la chaîne au moment de l’appel.

```
void add_begin_chain(uint64_t i);
```

► **Question 42** Écrire la fonction `remove_from_chain` qui supprime la portion libre `i` de la chaîne.

```
void remove_from_chain(uint64_t i);
```

► **Question 43** Écrire la fonction `init_heap` pour cette stratégie.

► **Question 44** Écrire la fonction `malloc_ui` pour cette stratégie et déterminer sa complexité.

Pour libérer une portion, on réalise comme dans la partie précédente une fusion avec les éventuelles portions libres adjacentes en mémoire, mais en les supprimant au préalable de la chaîne, puis en ajoutant en entrée de la chaîne la portion libre créée par la fusion. À nouveau, on déplace l’épilogue vers la gauche lorsque c’est possible.

► **Question 45** Écrire la fonction `free_ui` pour cette stratégie et déterminer sa complexité.

► **Question 46** Commenter les avantages de cette stratégie par rapport à la stratégie précédente.

Cet énoncé est une adaptation du sujet d’informatique de tronc commun 2021 du concours XENS (il était donc prévu pour être traité en Python).

TRI RADIX

Le *tri radix* est généralement la manière la plus efficace de trier des entiers ou des flottants (mais il ne permet pas de trier des données plus compliquées, comme par exemple des chaînes de caractères).

I Tri stable

1.1 Tri suivant une clé

Pour l'instant, nous avons essentiellement trié des tableaux de nombres, suivant l'ordre « naturel », c'est-à-dire la relation d'ordre usuelle \leq sur \mathbb{N} , sur \mathbb{Z} , sur \mathbb{R} ...

Il est cependant très courant de vouloir trier des données suivant un critère plus ou moins arbitraire :

- une liste de triplets (Nom, Prénom, Date de naissance) par date de naissance croissante ;
- une liste de listes par longueurs décroissante (les listes les plus longues en premier, les listes les plus courtes en dernier) ;
- une liste de complexes par module croissant ;
- une liste de triplets (Nom, Prénom, Date de naissance) par date de naissance croissante, puis en cas d'égalité par nom croissant, puis en cas d'égalité par prénom croissant.

Remarque

Dans les trois premiers exemples, l'ordre des éléments dans la liste triée n'est pas entièrement défini : les listes $(1, i, 2)$ et $(i, 1, 2)$, par exemple, sont toutes les deux triées par module croissant. Nous y reviendrons dans la partie suivante.

Pour spécifier l'ordre suivant lequel on souhaite trier, le plus courant est de fournir une *fonction de comparaison* : c'est la manière standard de procéder en OCaml, en C, en C++, en JavaScript¹... Une telle fonction prend deux éléments à trier et renvoie un entier qui vérifie :

- $\text{compare}(x, y) < 0$ si x doit être placé avant y (si x est « plus petit » que y pour l'ordre considéré) ;
- $\text{compare}(x, y) > 0$ si x doit être placé après y (si x est « plus grand » que y pour l'ordre considéré) ;
- $\text{compare}(x, y) = 0$ si x et y sont « équivalents » pour l'ordre considéré, ce qui signifie que l'on peut placer x avant ou après y , indifféremment.

Remarque

Pour qu'une telle fonction de comparaison ait un sens, il faut qu'elle vérifie certaines propriétés. Nous y reviendrons quand nous étudierons plus en détail les ensembles ordonnés, mais pour l'instant nous pouvons l'ignorer : si la fonction traduit effectivement le fait pour x de devoir être placé avant ou après y , ces propriétés seront toujours vérifiées.

Exercice XIX.I – `List.sort` et `Array.sort`

p. 519

En OCaml, les fonctions de tri pré-définies ont le type suivant :

```
List.sort : ('a -> 'a -> int) -> 'a list -> 'a list
Array.sort : ('a -> 'a -> int) -> 'a array -> unit
```

Il y a une différence dans le type d'arrivée (ce qui est normal puisque `List.sort` renvoie une nouvelle liste alors que `Array.sort` modifie le tableau fourni en argument), mais les deux fonctions ont un premier argument de type `'a -> 'a -> int`. Cet argument est la fonction de comparaison qui définit l'ordre suivant lequel on souhaite trier.

1. Mais pas en Python.

1. Que va renvoyer `List.sort (fun x y -> x - y) [3; 2; 4; 1; 7]`?
2. Même question pour `List.sort (fun x y -> y - x) [3; 2; 4; 1; 7]`.
3. Comment trier une liste d'entiers par valeur absolue décroissante? La fonction `abs : int -> int` permet de calculer la valeur absolue d'un entier.
4. Comment trier une liste de type (`int * int`) `list` suivant le critère suivant :
 - par valeur absolue croissante de la première composante;
 - en cas d'égalité, par seconde composante décroissante.

Comme la fonction de comparaison est ici un peu plus compliquée, il est fortement conseillé de la définir séparément.

Remarque

Si l'on veut trier une liste (ou un tableau) suivant l'ordre « usuel » (celui correspondant à l'opérateur `<=`), on peut utiliser la fonction prédéfinie `compare` : `'a -> 'a -> int`.

1.2 Tri stable

Comme nous l'avons vu plus haut, une fonction de comparaison ne définit pas en général un ordre unique sur les éléments. Considérons par exemple la liste $u = [(3, 4), (5, 4), (2, 7), (6, 1), (3, 3), (7, 2), (2, 1)]$. Si l'on souhaite trier les éléments (x, y) de u par somme croissante, toutes les réponses suivantes sont acceptables :

- $[(2, 1), (3, 3), (3, 4), (6, 1), (5, 4), (2, 7), (7, 2)]$;
- $[(2, 1), (3, 3), (6, 1), (3, 4), (5, 4), (2, 7), (7, 2)]$;
- $[(2, 1), (3, 3), (6, 1), (3, 4), (7, 2), (2, 7), (5, 4)]$;
- $[(2, 1), (3, 3), (6, 1), (3, 4), (7, 2), (5, 4), (2, 7)]$;
- ...

Définition XIX.1 – Tri stable

Un tri (suivant une clé) est dit *stable* si, pour tous x, y de la liste initiale tels que `compare(x, y) = 0`, x est avant y dans la liste triée si et seulement si il était avant y dans la liste initiale.

Remarques

- Dans l'exemple ci-dessus, un tri *stable* par somme croissante donne nécessairement le premier résultat proposé. Il y a trois couples dont la somme vaut 9, ils doivent nécessairement être dans le même ordre à l'arrivée qu'au départ, c'est-à-dire $(5, 4), (2, 7), (7, 2)$.
- Quelle que soit la clé choisie pour le tri, le résultat d'un tri stable est uniquement défini.
- Un tri est stable s'il n'échange la position relative de deux éléments que lorsque c'est nécessaire pour respecter l'ordre demandé.

Le caractère stable ou non est une propriété de l'algorithme de tri utilisé. En OCaml, il existe deux fonctions `List.stable_sort` et `Array.stable_sort`, dont la stabilité est garantie; ce n'est pas le cas pour `List.sort` et `Array.sort`². Un des principaux intérêts d'un tri stable est que l'on peut effectuer plusieurs tris successifs suivant des critères simples et obtenir le même résultat qu'après un unique tri suivant un critère plus compliqué.

Exercice XIX.2 – Tris stables successifs

p. 519

On considère une liste de couples d'entiers, et l'on souhaite trier cette liste :

- par somme croissante;
- en cas d'égalité, par première composante croissante.

Par exemple, sur la liste $u = [(3, 4), (5, 4), (2, 7), (6, 1), (3, 3), (7, 2), (2, 1)]$, on doit obtenir :

$[(2, 1), (3, 3), (3, 4), (6, 1), (2, 7), (5, 4), (7, 2)]$

2. À l'heure actuelle, `List.sort` utilise le même algorithme que `List.stable_sort` (et est donc stable), mais `Array.sort` utilise un algorithme différent (qui n'est pas stable).

On considère les deux fonctions suivantes :

```
let cmp_somme (x, y) (x', y') = x + y - (x' + y')
```

```
let cmp_premiere (x, _) (x', _) = x - x'
```

Si \mathbf{u} est donnée sous la forme d'un (`int * int`) `array`, comment la trier en utilisant deux appels à `Array.stable_sort` ?

2 Tri radix

Principe

Supposons que l'on dispose d'un tableau de n entiers, tous compris entre 0 et 999. L'idée du tri radix (en base 10 dans cet exemple) est d'effectuer trois passes successives de tri :

- une première en ne tenant compte que du chiffre des unités ;
- une seconde en ne tenant compte que du chiffre des dizaines ;
- une troisième en ne tenant compte que du chiffre des centaines.

Si chaque étape de tri est effectuée de manière stable, le tableau sera trié par ordre croissant à la fin.

Exercice XIX.3

p. 519

Dérouler les grandes étapes de l'algorithme sur la liste :

(123, 211, 312, 321, 133, 121, 213, 30, 103, 200)

Pour effectuer un tri radix *efficace*, il faut :

- que chaque passe de tri s'effectue rapidement ;
- que le nombre de passes soit aussi petit que possible.

Pour le premier point, une première idée est de travailler en base 2 plutôt qu'en base 10 (ou autre) : en effet, extraire le k -ème chiffre en base 2 peut se faire de manière très efficace à l'aide d'opérations *bitwise*, comme nous l'avons vu en cours. Il reste ensuite à trouver une manière efficace d'effectuer l'étape de tri, mais nous en parlerons plus tard.

Pour le deuxième point en revanche, la base 2 n'est pas idéale : en effet, pour un nombre n donné, plus la base est petite, plus il y a de chiffres. Pour diminuer le nombre d'étapes tout en gardant des opérations arithmétiques efficaces, l'idée est alors de *regrouper les bits* par paquet de taille fixée. Cela revient en fait à remplacer la base 2 par une base 2^p pour un certain p .

Remarque

Pour fixer l'intuition, on peut remarquer que 754215 possède 6 chiffres en base 10, 3 chiffres en base $10^2 = 100$ ($75 \cdot 100^2 + 42 \cdot 100 + 15$) et deux chiffres en base $10^3 = 1000$ ($754 \cdot 1000 + 215$).

Exercice XIX.4 – Nombre de passes

p. 519

On considère que l'on souhaite trier des entiers non signés codés sur w bits, et que l'on va utiliser la base 2^p .

1. Combien d'étapes faudra-t-il faire dans le tri radix (au maximum) ?
2. Comme nous le verrons plus tard, le coût d'une étape croît avec p . Si $w = 32$, quelles sont les valeurs de p qui ont une chance d'être optimales ?

Algorithme pour une passe

Chaque passe du tri radix va se faire suivant le principe suivant :

- on a un tableau `in` d'entiers non signés, que l'on souhaite trier, de manière stable, par valeur croissante du chiffre de poids radix^k ;
- le tableau `in` ne sera pas modifié : on renverra un nouveau tableau `out` ;
- on commence par calculer un tableau `hist` de longueur radix indiquant, pour chaque valeur $i \in [0 \dots \text{radix} - 1]$, combien d'éléments du tableau `in` ont leur chiffre de poids radix^k égal à i ;
- on calcule ensuite un tableau `sums`, également de longueur radix , indiquant pour chaque i combien d'éléments de `in` ont leur chiffre de poids radix^k strictement inférieur à i ;
- on remarque que les x de `in` dont le chiffre considéré vaut i occuperont des cases consécutives du tableau `out` à partir de la case `sums[i]` ;
- on parcourt le tableau `in` en répartissant les éléments dans le tableau `out` suivant la valeur de leur chiffre (il faut mettre à jour `sums` au fur et à mesure).

Les figures XIX.1 et XIX.2 illustrent le principe de l'algorithme, avec $\text{radix} = 4$ et $k = 0$ (on s'intéresse donc au chiffre des unités en base 4).

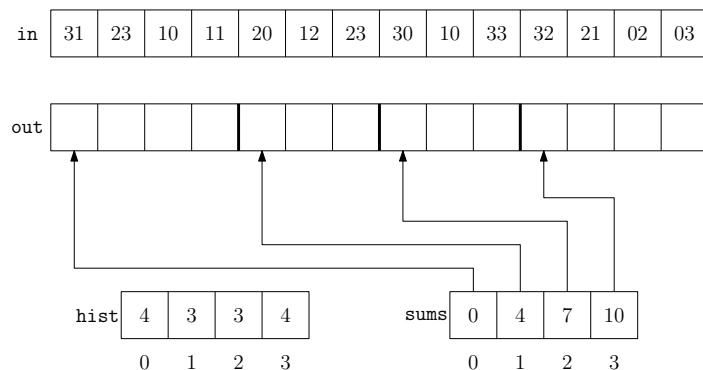


FIGURE XIX.1 – État initial après le calcul de `hist` et `sums`.

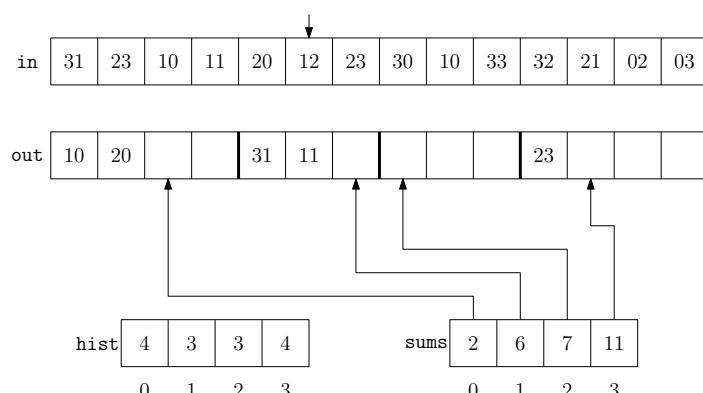


FIGURE XIX.2 – État après avoir traité 5 éléments (l'élément 12 pointé sera le prochain à être traité).

Programmation du tri radix

Dans toute la suite, on suppose que :

- on souhaite trier un tableau d'entiers de type `uint32_t` ou `uint64_t`, et que l'on a fait l'un des **typedef** suivants :

```
typedef uint32_t ui;
// ou bien
typedef uint64_t ui;
```

- le nombre d’éléments dans le tableau à trier est inférieur à $2^{31} - 1$ (soit environ deux milliards), de sorte que l’on peut utiliser des `int` pour tous nos compteurs;
- on a défini une constante globale `BLOCK_SIZE`, et que l’on effectuera le tri en base $2^{\text{BLOCK_SIZE}}$ (c’est-à-dire en regroupant les bits par paquets de `BLOCK_SIZE`);
- on notera `radix` notre base (autrement dit, `radix = 2BLOCK_SIZE`).

Exercice XIX.5 – Définition des constantes

p. 520

Définir les constantes globales `RADIX = 2BLOCK_SIZE` et `MASK = 2BLOCK_SIZE - 1`.

Exercice XIX.6 – Fonctions utilitaires

p. 520

Écrire deux fonctions :

```
void copy(ui *out, ui *in, int len);
void zero_out(int *arr, int len);
```

L’appel `copy(out, in, len)` doit recopier le contenu du tableau `in` sur le tableau `out` (on suppose que les deux tableaux ont la même taille).

L’appel `zero_out(arr, len)` doit remplacer toutes les valeurs présentes dans le tableau `arr` par des zéros.

Remarque

Il existe bien sûr des fonctions pour faire cela dans la bibliothèque standard, mais ici les ré-écrire pour notre usage particulier ne nous coûte pas grand-chose.

Exercice XIX.7 – Extraction d’un chiffre en base radix

p. 520

Écrire une fonction `extract_digit` qui renvoie la valeur du chiffre de poids radix^k dans l’écriture de `n` en base `radix`.

```
ui extract_digit(ui n, int k);
```

On procédera à l’aide d’opérations bitwize : la technique a déjà été vue exactement en exercice. On pourra supposer que la valeur de `k` correspond bien à un chiffre qui existe

Exercice XIX.8 – Calcul de l’histogramme

p. 520

Écrire une fonction `histogram` prenant en entrée un tableau d’entiers `arr` et un entier `k` et renvoyant un tableau `hist` de longueur `radix` tel que `hist[i]` soit égal au nombre d’éléments de `arr` dont le `k`-ème chiffre en base `radix` vaut `i` (pour $0 \leq i < \text{radix}$).

Ce calcul doit absolument se faire en un seul parcours du tableau `arr`.

```
int* histogram(ui *arr, int len, int k);
```

Exercice XIX.9 – Sommes préfixes

p. 520

Écrire une fonction `prefix_sum` prenant en entrée un tableau `hist` et renvoyant un tableau `sums` de même longueur tel que `sums[i]` soit égal à $\sum_{j=0}^{i-1} \text{hist}[j]$. On aura donc en particulier `sums[0] = 0`.

Ce calcul doit absolument se faire en un seul parcours du tableau `hist`.

```
int* prefix_sums(int* hist, int len);
```

Exercice XIX.I0

p. 521

Écrire une fonction `radix_pass` ayant le prototype suivant :

```
void radix_pass(ui *out, ui *in, int len, int k);
```

Cette fonction implémentera l'algorithme décrit ci-dessus en considérant le chiffre de poids radix^k .

Exercice XIX.II

p. 521

Écrire la fonction `radix_sort`. Cette fonction ne renverra rien mais modifiera le tableau passé en argument. Pour l'instant, on essaiera surtout d'écrire une fonction correcte (quitte à faire des copies de tableaux superflues).

```
void radix_sort(ui *arr, int len)
```

Exercice XIX.I2 – Complexité du tri radix

p. 521

Déterminer la complexité totale du tri radix en fonction de la largeur w des entiers, de la valeur ℓ de `BLOCK_SIZE` et de la longueur n du tableau.

Optimisations

Dans cette partie, on s'intéresse aux performances pratiques de notre implémentation. Mesurer ces performances n'a pas trop de sens avec nos options de compilation usuelles ; on utilisera plutôt :

```
$ gcc -O3 -march=native -DNDEBUG -Wall -Wextra -o radix.out radix.c
```

Pour mesurer le temps écoulé lors de l'exécution d'une partie du programme, on pourra ajouter l'entête `#include <time.h>` et utiliser la fonction `clock`. La différence entre deux appels à `clock()` peut être divisée par la constante `CLOCKS_PER_SEC` pour obtenir le temps (en secondes) entre les deux appels.

Exercice XIX.I3

p. 521

Tester différentes valeurs de `BLOCK_SIZE` et essayer de déterminer la valeur idéale.

Exercice XIX.I4

p. 521

En pratique, le facteur limitant pour les performances (dans le cas du tri radix) est le sous-système mémoire. Il est donc très important de minimiser le nombre de lectures et d'écriture : il est possible de ne faire que 5 passes de lecture et 4 passes d'écriture pour trier un tableau de `uint32_t` avec un `BLOCK_SIZE` de 8. Déterminer si c'est bien le cas pour la version que vous avez écrite, et la modifier le cas échéant. *Attention, il est très facile d'introduire des bugs ici. On prendra bien soin de tester les modifications apportées, et ce pour plusieurs valeurs de `BLOCK_SIZE`.*

Cas des entiers signés

Exercice XIX.I5

p. 523

Écrire une version de `radix_sort` permettant de trier un tableau d'entiers signés.

Solutions

Correction de l'exercice XIX.1 page 513

1. On trie par ordre croissant : [1; 2; 3; 4; 7].
2. On trie par ordre décroissant : [7; 4; 3; 2; 1].
3. `List.sort (fun x y -> abs y - abs x) liste.`
- 4.

```
let cmp (x, y) (x', y') =
  if abs x < abs x' then -1
  else if abs x > abs x' then 1
  else y' - y

let trier liste = List.sort cmp liste
```

Correction de l'exercice XIX.2 page 514

Il faut faire :

```
Array.stable_sort cmp_premiere u;
Array.stable_sort cmp_somme u
```

Correction de l'exercice XIX.3 page 515

On obtient successivement :

- (30, 200, 211, 321, 121, 312, 123, 133, 213, 103)
- (200, 103, 211, 312, 213, 321, 121, 123, 30, 133)
- (30, 103, 121, 123, 133, 200, 211, 213, 312, 321)

Correction de l'exercice XIX.4 page 515

1. En k étapes on traite kp bits, il faut donc que $kp \geq w$, c'est-à-dire $k \geq \frac{w}{p}$. Le nombre d'étapes nécessaire vaut donc $\lceil w/p \rceil$.
2. On peut éliminer les valeurs de p telles que $\lceil w/p \rceil = \lceil w/(p-1) \rceil$, puisque $p-1$ sera toujours préférable dans ce cas (même nombre d'étape, et chaque étape est moins coûteuse). Les valeurs potentiellement intéressantes sont donc (pour $w = 32$) :
 - $p = 1$, 32 étapes;
 - $p = 2$, 16 étapes;
 - $p = 3$, 11 étapes;
 - $p = 4$, 8 étapes;
 - $p = 5$, 7 étapes;
 - $p = 6$, 6 étapes;
 - $p = 7$, 5 étapes;
 - $p = 8$, 4 étapes;
 - $p = 11$, 3 étapes;
 - $p = 16$, 2 étapes;
 - $p = 32$, 1 étape (cette valeur est complètement irréalistique comme on le verra une fois l'algorithme expliqué).

Correction de l'exercice XIX.5 page 517

```
const int BLOCK_SIZE = 5; // par exemple
const int RADIX = 1 << BLOCK_SIZE;
const int MASK = RADIX - 1;
```

Correction de l'exercice XIX.6 page 517

```
void copy(ui *out, ui *in, int len){
    for (int i = 0; i < len; i++){
        out[i] = in[i];
    }
}

void zero_out(int *arr, int len){
    for (int i = 0; i < len; i++){
        arr[i] = 0;
    }
}
```

Correction de l'exercice XIX.7 page 517

```
ui extract_digit(ui n, int k){
    return (n >> (k * BLOCK_SIZE)) & MASK;
}
```

Correction de l'exercice XIX.8 page 517

```
int* histogram(ui *arr, int len, int k){
    int* hist = malloc(RADIX * sizeof(int));
    zero_out(hist, RADIX);
    for (int i = 0; i < len; i++){
        int digit = extract_digit(arr[i], k);
        hist[digit]++;
    }
    return hist;
}
```

Correction de l'exercice XIX.9 page 517

```
int* prefix_sum(int *hist, int len){
    int* sums = malloc(len * sizeof(int));
    int s = 0;
    for (int i = 0; i < len; i++){
        sums[i] = s;
        s += hist[i];
    }
    return sums;
}
```

Correction de l'exercice XIX.10 page 518

On suit précisément les étapes décrites dans l'algorithme et illustrées sur les schémas. Attention à ne pas oublier de libérer les deux tableaux auxiliaires.

```
void radix_pass(ui *out, ui *in, int len, int k){
    int* hist = histogram(in, len, k);
    int* sums = prefix_sum(hist, RADIX);
    for (int i = 0; i < len; i++){
        ui digit = extract_digit(in[i], k);
        out[sums[digit]] = in[i];
        sums[digit]++;
    }
    free(hist);
    free(sums);
}
```

Correction de l'exercice XIX.11 page 518

On peut faire nettement plus efficace, mais la version ci-dessous est la plus simple, et elle a la bonne complexité. Pour le calcul de nb_digits, on a remarqué que $\lceil w/b \rceil = 1 + \lfloor (w-1)/b \rfloor$, sinon il faut utiliser un **if** ($w \% b == 0$) {...} **else** {...}.

```
void radix_sort(ui *in, int len){
    int nb_digits = 1 + (sizeof(ui) * 8 - 1) / BLOCK_SIZE;
    for (int k = 0; k < nb_digits; k++){
        ui *tmp = malloc(len * sizeof(ui));
        radix_pass(tmp, in, len, k);
        copy(in, tmp, len);
        free(tmp);
    }
}
```

Correction de l'exercice XIX.12 page 518

Pour chaque passe :

- le calcul de l'histogramme est en $O(n + 2^\ell)$ (puisque'on crée un tableau de taille 2^ℓ);
- le calcul des sommes préfixes est en $O(2^\ell)$;
- la répartition des éléments dans out est en $O(n)$.

Une passe est donc en $O(n + 2^\ell)$ et comme il y a $O(w/\ell)$ passes, on a au total du $O\left(\frac{w(n + 2^\ell)}{\ell}\right)$.

Correction de l'exercice XIX.13 page 518

Tester les valeurs de BLOCK_SIZE avant d'avoir optimisé le programme n'est pas forcément idéal, mais cela permet quand même d'avoir une idée. Le plus souvent, le meilleur choix est 8, mais les valeurs 6, 7 et 11 peuvent aussi être intéressantes (ça dépend de détails architecturaux, et il faut donc tester pour savoir ce qui est le mieux *sur une machine donnée*).

Correction de l'exercice XIX.14 page 518

On propose ci-dessous un code raisonnablement optimisé :

```

void inplace_prefix_sum(int* t, int n){
    int s = 0;
    for (int i = 0; i < n; i++){
        int tmp = t[i];
        t[i] = s;
        s += tmp;
    }
}

void radix_sort_2(ui* t, int len){
    int nb_digits = 1 + (sizeof(ui) * 8 - 1) / BLOCK_SIZE;
    ui* t_aux = malloc(len * sizeof(ui));
    int* hists = malloc(RADIX * nb_digits * sizeof(int));

    // initialize hists
    for (int i = 0; i < RADIX * nb_digits; i++){
        hists[i] = 0;
    }

    // Compute all histograms
    for (int i = 0; i < len; i++){
        for (int k = 0; k < nb_digits; k++){
            int digit = extract_digit(t[i], k);
            hists[k * RADIX + digit]++;
        }
    }

    for (int i_digit = 0; i_digit < nb_digits; i_digit++){
        // select the current histogram
        int* hist = &hists[i_digit * RADIX];

        // replace it with its prefix sum
        inplace_prefix_sum(hist, RADIX);

        // scatter
        for (int i = 0; i < len; i++){
            int digit = extract_digit(t[i], i_digit);
            t_aux[hist[digit]] = t[i];
            hist[digit]++;
        }

        // switch t and t_aux
        ui* tmp = t;
        t = t_aux;
        t_aux = tmp;
    }

    if (nb_digits % 2 != 0){
        copy(t_aux, t, len);
        free(t);
    } else {
        free(t_aux);
    }

    free(hists);
}

```

Sur ma machine, on gagne un facteur 2 environ par rapport à la première version, pour arriver à approximativement 8 nano-secondes par élément du tableau⁷

a. Pour un tableau d'un million d'éléments. La complexité est linéaire en n pour l et w fixés, mais en réalité il y a de petites variations.

Correction de l'exercice XIX.I5 page 518

Le plus simple est de trier comme nous l'avons fait jusqu'à présent et de rajouter une ultime passe dans laquelle on place tous les éléments négatifs avant les positifs (en utilisant le même principe que pour une passe du tri radix, mais avec deux catégories au lieu de 2^{radix}).

EXPRESSIONS ARITHMÉTIQUES

1 Arbre d'une expression arithmétique

Une expression arithmétique est fondamentalement un arbre :

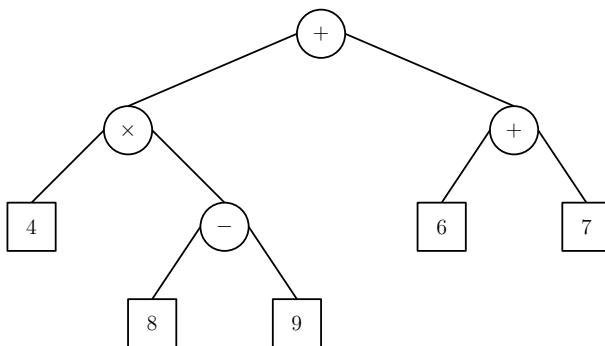


FIGURE XX.1 – Un exemple d'expression arithmétique

On choisit de représenter une expression en utilisant le type suivant :

```

type op =
| Plus
| Fois
| Moins

type expr =
| C of int
| N of op * expr * expr
  
```

Exercice XX.1

p. 528

1. Donner la définition en OCaml de l'arbre représenté ci-dessus.
2. Représenter graphiquement, et définir en OCaml, les arbres correspondant aux expressions $2 + 3 * 4$ et $(2 + 3) * 4$ (en appliquant les règles de priorité usuelles).

Exercice XX.2 – Évaluation d'une expression

p. 528

1. Écrire une fonction `applique` : `op -> int -> int -> int` qui prend en entrée un opérateur binaire et deux opérandes et renvoie le résultat.

```

# applique Moins 2 5;;
- : int = -3
# applique Fois 4 8;;
- : int = 32
  
```

2. Écrire une fonction `eval` : `expr -> int` qui prend l'arbre d'une expression et l'évalue.

2 Différentes notations pour les expressions arithmétiques

En appliquant les différents parcours en profondeur que nous avons vus à l'arbre d'une expression, on obtient différentes représentations « à plat ».

- En notation *infixe*, c'est-à-dire en plaçant les opérateurs entre les opérandes. C'est la notation traditionnelle, mais elle a l'inconvénient de nécessiter des parenthèses.

Exemples : $2 + (3 * 5) = 17$ et $(2 + 3) * 5 = 25$.

On fixe habituellement des règles de priorité qui permettent d'éliminer une partie des parenthèses (mais seulement une partie...).

- En notation *préfixe*, c'est-à-dire en plaçant les opérateurs avant les opérandes. Aucune parenthèse n'est nécessaire.

Exemples : $+ 2 * 3 5 = 17$ et $* + 2 3 5 = 25$.

- En notation *postfixe*, avec les opérateurs après les opérandes. Là non plus, aucune parenthèse n'est nécessaire.

Exemples : $2 3 5 * + = 17$ et $2 3 + 5 * = 25$.

Remarque

Le fait qu'on ait besoin de parenthèses uniquement dans le cas infixé vient du fait qu'il est possible de reconstruire un arbre à partir de son parcours préfixe ou suffixe (à condition qu'on distingue bien les feuilles des nœuds internes), mais pas à partir de son parcours infixé. Nous l'avons vu, mais pas encore prouvé, en cours.

Exercice XX.3

p. 528

Donner les trois notations possibles pour l'arbre dessiné au début du sujet.

Exercice XX.4

p. 528

1. Écrire les expressions suivantes en notation préfixe et en notation postfixe :

- $(3 - 2) * 4$
- $(2 + 3) * (1 + 8)$
- $(2 + (3 + 4)) * (5 - 6)$

2. Les expressions suivantes sont en notation préfixe. Les traduire en infixé.

- $- * 2 3 + 1 4$
- $+ * - 4 5 6 7$

3. Les expressions suivantes sont en notation postfixe. Les traduire en infixé.

- $2 3 + 4 5 * -$
- $1 2 3 4 + * 5 - *$

2.1 À partir de l'arbre

On définit le type suivant :

```
type lexeme = P0 | PF | Op of op | Val of int
```

Les constructeurs **P0** et **PF** correspondent respectivement aux parenthèses ouvrantes et fermantes.

Exercice XX.5

p. 529

Écrire les fonctions suivantes (on ira au plus simple sans trop se préoccuper de la complexité) :

- prefixe** : expr → lexeme list qui construit la représentation préfixe d'un arbre.
- postfixe** et **infixe** similaires. Pour **infixe**, on ne cherchera pas à éliminer les parenthèses inutiles.

2.2 À partir de la notation postfixe

Exercice XX.6 – Évaluation d'une expression postfixe

p. 529

Il est très aisés d'évaluer une expression postfixe à l'aide d'une pile. Par exemple, à partir de l'expression $2 \ 4 \ - \ 5 \ * \ 6 \ +$, on obtient (en écrivant la pile avec le sommet à gauche) :

Étape	Pile
2	[2]
4	[4; 2]
-	[-2]
5	[5; -2]
×	[-10]
6	[6; -10]
+	[-4]

1. Traiter de la même manière l'expression $1 \ 2 \ 3 \ 4 \ 5 \ + \ * \ - \ 6 \ * \ +$.
2. Que se passe-t-il pour $1 \ 2 \ + \ - \ 3$? et pour $1 \ 2 \ 3 \ + \ ?$
3. Si s est une expression postfixe, on note $\text{ent}_s(i)$ le nombre d'entiers apparaissant dans les i premiers éléments de s et $\text{op}_s(i)$ le nombre d'opérateurs dans les i premiers éléments. Donner à l'aide de ces fonctions une condition nécessaire et suffisante pour que s soit bien formée (on ne demande pas de démonstration).
4. Écrire une fonction `eval_post` : `lexeme list -> int` qui prend une liste de lexèmes et l'évalue en tant qu'expression postfixe. On pourra supposer que la liste ne contient pas de parenthèses et on lèvera une exception si l'expression n'est pas bien formée.

Exercice XX.7

p. 530

Écrire une fonction `arbre_of_post` : `lexeme list -> arbre` qui prend une expression postfixe en entrée et renvoie l'arbre correspondant.

Il suffit d'apporter quelques modifications à la fonction d'évaluation écrite à l'exercice précédent.

3 Expressions avec variables

On considère un ensemble infini dénombrable de variables entières $\mathcal{V} = \{x_0, x_1, \dots\}$, et l'on étend le type des expressions :

```
type expr2 =
| N of op * expr2 * expr2
| C of int
| V of int
```

Ici, une feuille `V i` ne représente pas l'entier i mais la variable x_i . Une feuille `C x`, en revanche, représente directement l'entier x (comme depuis le début du sujet).

Une *valuation* est une application de \mathcal{V} dans \mathbb{Z} (qui associe une valeur à chaque variable). En pratique, on n'aura besoin de spécifier les valeurs que d'un nombre fini de variables (celles apparaissant dans l'expression que l'on évalue). On définit donc le type suivant :

```
type valuation = int array
```

Si $t : \text{valuation}$ est de longueur n , alors la valeur de x_i pour $0 \leq i < n$ est donnée par $t.(i)$ (et les x_i avec $i \geq n$ ont des valeurs non spécifiées).

Exercice XX.8

p. 530

1. Écrire une fonction `max_var : expr2 -> int` qui renvoie le plus grand indice de variable apparaissant dans l'expression reçue en argument. Si l'expression ne contient aucune variable, la fonction pourra avoir un comportement quelconque.
2. Écrire une fonction `eval_contexte : expr2 -> valuation -> int` qui évalue une expression `e` étant donnée une valuation `v` de ses variables. On supposera (sans le vérifier) que `max_var e` est strictement inférieur à la longueur de `v`.

Remarque

La fonction `max_var` n'est donc pas utile.

Exercice XX.9

p. 530

On souhaite maintenant pouvoir évaluer *partiellement* une expression étant donnée une valuation qui ne recouvre pas nécessairement toutes les variables présentes dans l'expression. Le résultat de cette évaluation sera donc encore une expression, dans laquelle les calculs possibles ont été effectués.

1. Dans le cas où la valuation spécifie la valeur de toutes les variables de l'expression, quelle forme doit avoir l'expression renvoyée ?
2. Écrire la fonction `eval_partielle : expr2 -> valuation -> expr2`.

4 Forme normale pour l'associativité

Dans cette partie, on se limite pour simplifier à des expressions ne contenant que les opérateurs d'addition et de multiplication :

```
type op = Plus | Fois

type expr3 =
| C of int
| V of int
| N of op * expr3 * expr3
```

Comme les opérateurs \times et $+$ sont associatifs, les expressions $1 + (2 + 3)$ et $(1 + 2) + 3$ (par exemple) sont équivalentes : on préférerait les représenter toutes les deux par un même arbre.

Pour se faire, on introduit un nouveau type d'arbre pour les expressions :

```
(* Un arbre est soit une feuille, soit un nœud interne avec une
   liste d'enfants *)
type expr_naire =
| Cn of int
| Vn of int
| Nn of op * expr_naire list
```

Une expression `t : expr_naire` sera dite *en forme normale* (pour l'associativité) si :

- chaque nœud interne a au moins deux fils (elle ne contient donc pas de nœud de la forme `N (op, [])` ou `N (op, [e])`);
- un nœud `Plus` (respectivement `Fois`) n'a jamais de fils `Plus` (respectivement `Fois`).

Exercice XX.10

p. 531

Écrire une fonction `normalise : expr3 -> expr_naire` qui prend en entrée une expression (sous forme d'arbre binaire) et renvoie une expression équivalente en forme normale.

Solutions

Correction de l'exercice XX.1 page 524

1.

```
let arbre_exemple =
  N (Plus,
    N (Fois,
      C 4,
      N (Moins,
        C 8,
        C 9)),
    N (Plus,
      C 6,
      C 7))
```

2. ■ Pour $2 + 3 * 4$, l'arbre est **N (Plus, C 2, N (Fois, C 3, C 4))**.
■ Pour $(2 + 3) * 4$, l'arbre est **N (Fois, N (Plus, C 2, C 3), C 4)**.

Correction de l'exercice XX.2 page 524

1.

```
let applique op x y =
  match op with
  | Plus -> x + y
  | Moins -> x - y
  | Fois -> x * y
```

2.

```
let rec eval expr =
  match expr with
  | C x -> x
  | N (op, g, d) -> applique op (eval g) (eval d)
```

Correction de l'exercice XX.3 page 525

Préfixe : $+ * 4 - 8 9 + 6 7$
Postfixe : $4 8 9 - * 6 7 + +$

Infixe : $(4 * (8 - 9)) + (6 + 7)$

Correction de l'exercice XX.4 page 525

1. a. Préfixe : $* - 3 2 4$
Postfixe : $3 2 - 4 *$
b. Préfixe : $* + 2 3 + 1 8$
Postfixe : $2 3 + 1 8 + *$
c. Préfixe : $* + 2 + 3 4 - 5 6$
Postfixe : $2 3 4 + + 5 6 - *$

2. a. $(2 * 3) - (1 + 4)$
b. $((4 - 5) * 6) + 7$
3. a. $(2 + 3) - (4 * 5)$
b. $1 * (2 * (3 + 4) - 5)$

Correction de l'exercice XX.5 page 525

Les implémentations proposées sont **très inefficaces** (complexité quadratique dans le pire cas). Nous verrons de meilleures manières de procéder, mais ce n'est pas l'objectif aujourd'hui.

```
let rec prefixe expr =
  match expr with
  | C x -> [Val x]
  | N (op, g, d) -> Op op :: prefixe g @ prefixe d

let rec postfixe expr =
  match expr with
  | C x -> [Val x]
  | N (op, g, d) -> postfixe g @ postfixe d @ [Op op]

let rec infixe expr =
  match expr with
  | C x -> [Val x]
  | N (op, g, d) -> P0 :: infixe g @ [Op op] @ infixe d @ [PF]
```

Correction de l'exercice XX.6 page 526

1. Laissé au lecteur... On doit obtenir -149 .
2. Pour $1 \ 2 \ + \ - \ 3$ il n'y a que l'entier 3 sur la pile quand on veut appliquer le $-$, donc il y a une erreur. Pour $1 \ 2 \ 3 \ +$, il reste à la fin 1 et 5 sur la pile, alors qu'il ne devrait y avoir qu'un seul entier (le résultat final), donc c'est également une erreur.
3. Notons n le nombre de symboles (au total). Il faut, et il suffit, que :
 - $\text{ent}_s(i) \geq \text{op}_s(i) + 1$ pour $1 \leq i < n$;
 - $\text{ent}_s(n) = 1 + \text{op}_s(n)$.

Nous ferons la démonstration en cours, dans un cadre plus général.

4. Tout le travail est fait par la fonction auxiliaire, qui prend en argument :
 - expr la liste des symboles qu'il faut encore lire;
 - pile l'état actuel de la pile d'évaluation.

Attention, expr est de type lexeme **list** alors que pile est de type **int list** (et l'on voit expr comme une liste alors que l'on voit pile comme une pile, mais c'est une différence plus philosophique qu'autre chose).

```
let eval_post expr =
  let rec eval_aux expr pile =
    match expr, pile with
    | [], [x] -> x
    | Val x :: xs, _ -> eval_aux xs (x :: pile)
    | Op op :: xs, dr :: ga :: reste_pile ->
        eval_aux xs ((applique op ga dr) :: reste_pile)
    | _ -> failwith "expression incorrecte" in
  eval_aux expr []
```

Remarque

Le dernier cas regroupe en fait trois erreurs possibles :

- on lit un lexème **P0** ou **PF** (qui n'a rien à faire dans une expression postfixe);
- on lit un opérateur alors qu'il y a zéro ou un entier sur la pile;
- on arrive à la fin de l'expression et le nombre d'entiers sur la pile n'est pas exactement un.

Correction de l'exercice XX.7 page 526

On fait exactement la même chose, sauf que l'on remplace la pile d'entiers par une pile d'*arbres*.

```
let arbre_of_post expr =
  let rec aux expr pile =
    match expr, pile with
    | [], [x] -> x
    | Val x :: xs, _ -> aux xs (C x :: pile)
    | Op op :: xs, dr :: ga :: reste_pile ->
        aux xs (N (op, ga, dr) :: reste_pile)
    | _ -> failwith "expression incorrecte" in
  aux expr []
```

Correction de l'exercice XX.8 page 527

1. Renvoyer `min_int` quand il n'y a pas de variable permet de simplifier l'écriture de la fonction :

```
let rec max_var = function
  | C _ -> min_int
  | V i -> i
  | N (_, g, d) -> max (max_var g) (max_var d)
```

On peut bien sûr décider de lever une exception dans ce cas, mais il faut alors (absolument!) distinguer davantage de cas ensuite :

```
let rec max_var = function
  | C _ -> failwith "max de l'ensemble vide"
  | V i -> i
  | N (_, C _, d) -> max_var d
  | N (_, g, C _) -> max_var g
  | N (_, g, d) -> max (max_var g) (max_var d)
```

2. On adapte simplement la fonction d'évaluation écrite au début du sujet :

```
let rec eval_contexte e v =
  match e with
  | C x -> x
  | V i -> v.(i)
  | N (op, g, d) -> applique op (eval_contexte g v) (eval_contexte d v)
```

Correction de l'exercice XX.9 page 527

1. Dans le cas où toutes les valeurs des variables sont connues, il faut renvoyer une feuille `C x`.
- 2.

```
let rec eval_partielle e v =
  match e with
  | C x -> C x
  | V i when i < Array.length v -> C (v.(i))
  | V i -> V i
  | N (op, g, d) ->
    match eval_partielle g v, eval_partielle d v with
    | C xg, C xd -> C (applique op xg xd)
    | g', d' -> N (op, g', d')
```

On pourrait très facilement ajouter des règles de simplification : par exemple, dans le `match` interne, le cas `C 0`, `d` donnerait 0 si `op = Fois` et `d` si `op = Plus...` Cependant, ce n'était pas demandé ici.

Correction de l'exercice XX.10 page 527

Le code est très simple, ce qui ne veut pas forcément dire qu'il est simple à écrire.

```
let rec remonte op = function
  | Nn (op', enfants) when op = op' -> enfants
  | e -> [e]

let rec normalise = function
  | C x -> Cn x
  | V x -> Vn x
  | N (op, g, d) ->
    Nn (op, remonte op (normalise g) @ remonte op (normalise d))
```

Si vous êtes arrivé jusqu'ici, vous avez bien mérité vos vacances. Si vous n'êtes pas arrivé jusqu'ici, vous les avez méritées quand même, mais vous ne le saurez jamais... .

PROMENADE SYLVESTRE

Dans tout le début du TP, on utiliser le type suivant :

```
type ('a, 'b) arbre =
| Interne of 'a * ('a, 'b) arbre * ('a, 'b) arbre
| Feuille of 'b
```

I Fonctions élémentaires sur les arbres

Exercice XXI.1

- Représenter graphiquement l'arbre suivant :

```
let exemple1 =
  Interne (12,
    Interne (4,
      Interne (7, Feuille 20, Feuille 30),
      Interne (14, Feuille 1, Feuille 2)),
    Feuille 20)
```

- Donner le type et la définition en OCaml de l'arbre ci-contre :

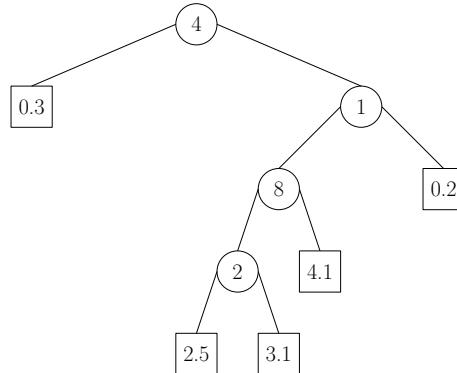


FIGURE XXI.1 – L'arbre exemple2.

Exercice XXI.2

- Écrire une fonction hauteur : ('a, 'b) arbre -> int qui calcule la hauteur d'un arbre (définie comme la longueur maximale d'un chemin reliant la racine à une feuille).
On doit avoir hauteur exemple1 = 3 et hauteur exemple2 = 4.
- Écrire une fonction taille : ('a, 'b) arbre -> int qui renvoie le nombre total de nœuds (internes ou non) dans un arbre.
On doit avoir taille exemple1 = 9 et taille exemple2 = 9.
- Écrire une fonction dernier : ('a, 'b) arbre -> 'b qui renvoie l'étiquette de la feuille située le plus à droite de l'arbre.
On doit avoir dernier exemple1 = 20 et dernier exemple2 = 0.2.

2 Parcours d'arbres

Exercice XXI.3 – À la main

Donner la liste des étiquettes de l'arbre exemple2 :

1. dans l'ordre du parcours en largeur;
2. dans l'ordre préfixe;
3. dans l'ordre infixé;
4. dans l'ordre postfixe.

Exercice XXI.4 – Parcours en profondeur

1. Écrire une fonction `affiche_prefixe : (int, int) arbre -> unit` qui affiche toutes les étiquettes des nœuds (internes ou non) de l'arbre passé en argument dans l'ordre préfixe. On utilisera :

- `print_int : int -> unit` pour afficher un entier;
- `print_newline : unit -> unit` pour revenir à la ligne.

2. Écrire de même des fonctions `affiche_infixe` et `affiche_postfixe`.

```
# affiche_prefixe exemple1;;
12
4
7
20
30
14
1
2
20
- : unit = ()
```

Exercice XXI.5 – Liste des étiquettes

On souhaite écrire des fonctions permettant d'obtenir la liste des étiquettes d'un arbre dans un ordre spécifié (préfixe, infixé ou postfixe). Comme les étiquettes des nœuds internes et celles des feuilles peuvent être de types différents, il nous faut pour cela créer un type à deux variantes :

```
type ('a, 'b) token = N of 'a | F of 'b
```

Les fonctions à écrire seront donc de type `('a, 'b) arbre -> ('a, 'b) token list`.

1. Écrire une fonction `postfixe_naif` la plus simple possible.
2. Expliquer pourquoi cette fonction risque d'être inefficace.
3. Écrire une fonction `postfixe` plus efficace, et préciser sa complexité. On utilisera une fonction aux : `('a, 'b) arbre -> ('a, 'b) token list -> ('a, 'b) token list` telle que aux a liste renvoie u @ liste, où u est la liste des étiquettes de a dans l'ordre postfixe.
Cette fonction ne fera aucune concaténation : le u @ liste est juste là pour donner la spécification.
4. En utilisant la même technique, écrire des fonctions `prefixe` et `infixe` efficaces.

Exercice XXI.6 – Version récursive terminale

On souhaite écrire une version récursive terminale de la fonction `postfixe`. La difficulté vient du fait qu'un appel à `postfixe` sur un arbre de la forme **Interne** (x, g, d) donne *a priori* deux appels récursifs : un sur l'arbre g et un sur d . Évidemment, au plus l'un de ces appels peut être en position terminale. Pour surmonter cette difficulté, on va maintenir une liste d'arbres à traiter (on parle de *forêt*) ; à chaque étape, on récupérera le premier élément de cette forêt, traitera sa racine, et, le cas échéant, ajoutera ses sous-arbres gauche et droit à la forêt. Le principe est illustré ci-dessous :

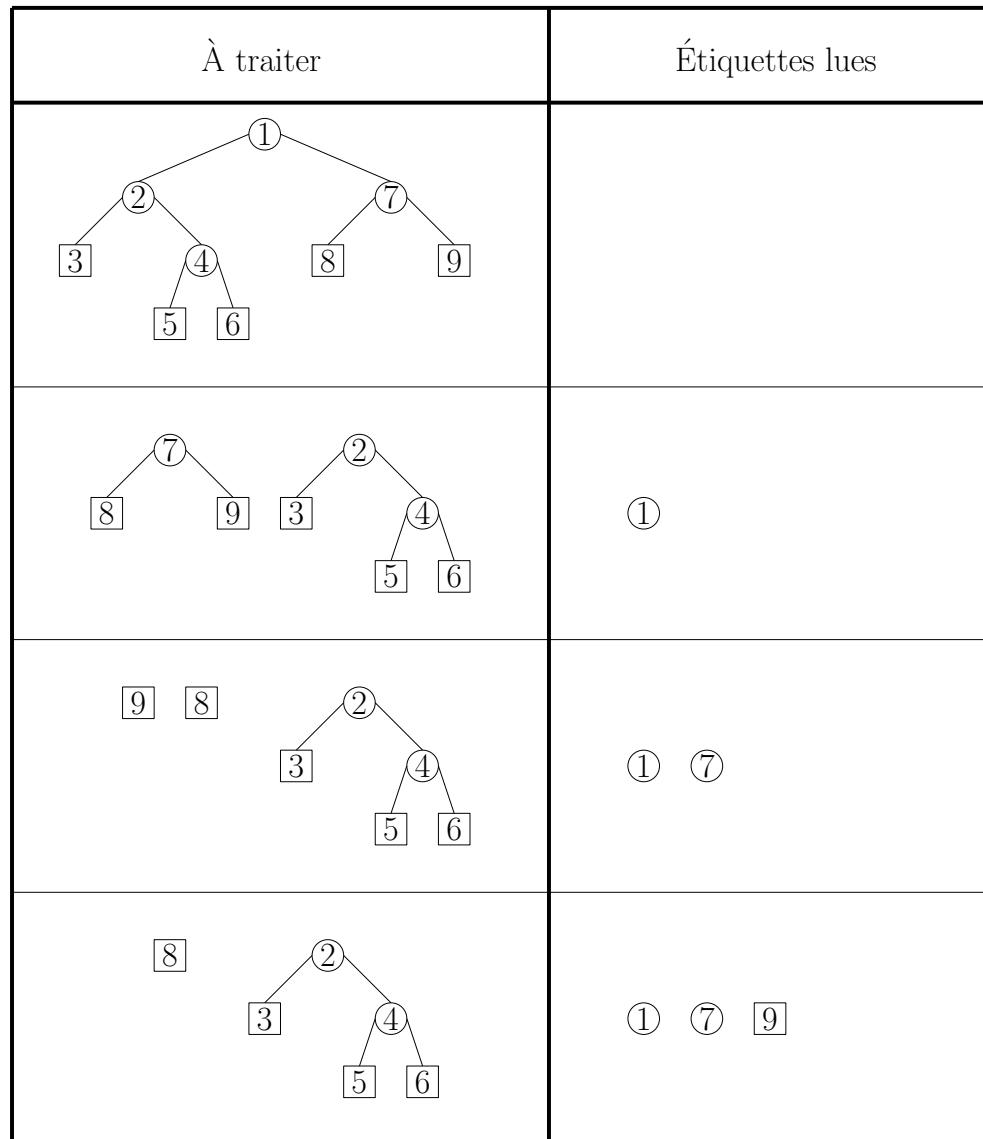


FIGURE XXI.2 – Principe du parcours en profondeur récursif terminal

1. Poursuivre l'exécution simulée de l'algorithme jusqu'à sa conclusion.
2. Écrire une fonction `postfixe_term` ayant les mêmes spécifications que `postfixe` mais étant récursive terminale. On utilisera une fonction auxiliaire
`aux : ('a, 'b) arbre list -> ('a, 'b) token list -> ('a, 'b) token list`
 prenant en entrée la forêt qui reste à traiter et les étiquettes déjà lues, et renvoyant la liste complète des étiquettes en suivant l'algorithme illustré ci-dessus.

Exercice XXI.7 – Parcours en largeur

On souhaite écrire une fonction `largeur` : (`'a, 'b`) arbre -> (`'a, 'b`) token **list** similaire à celle écrite à l'exercice précédent. Pour ce faire, on va utiliser l'algorithme illustré ci-dessous :

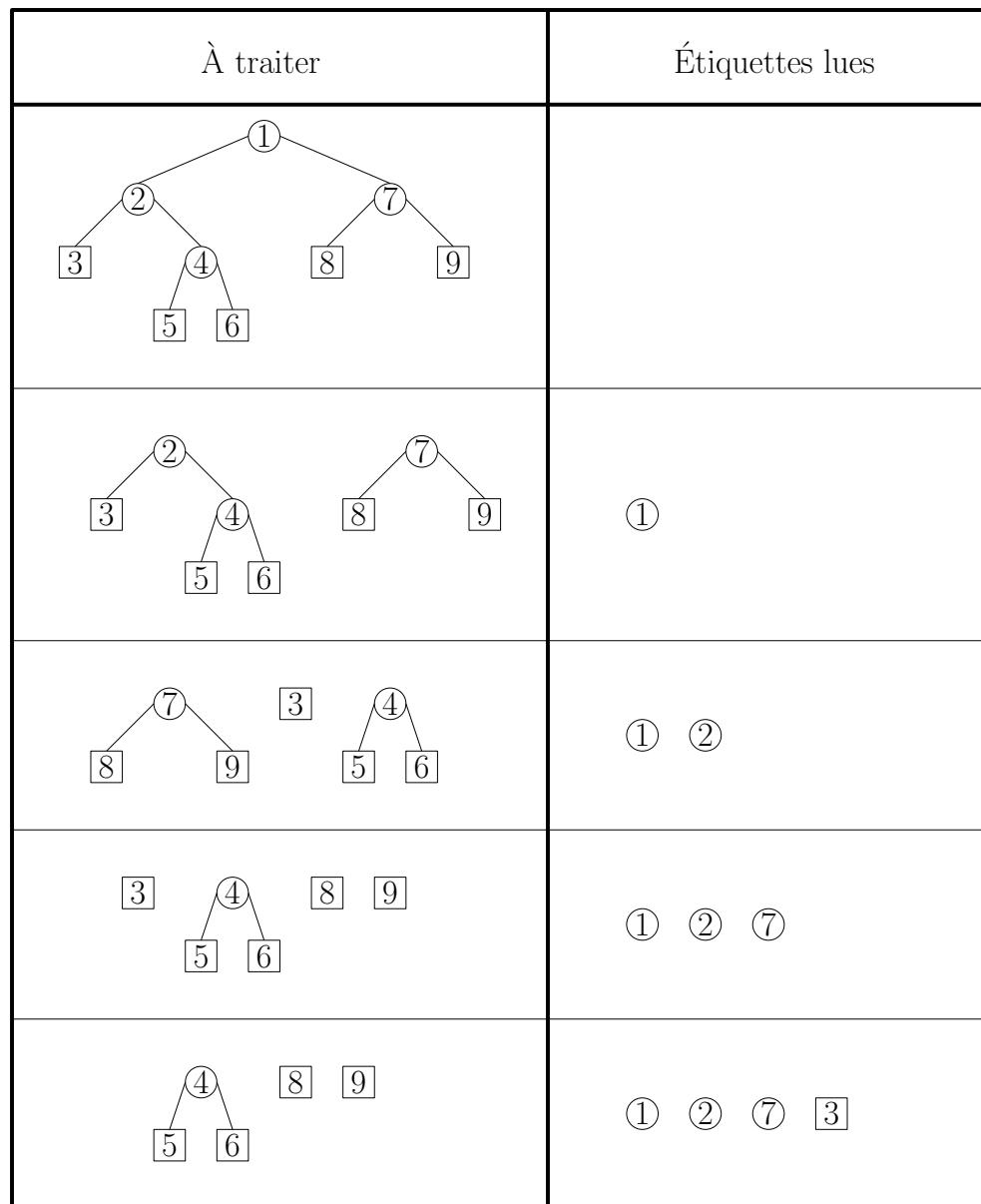


FIGURE XXI.3 – Algorithme de parcours en largeur

1. Poursuivre l'exécution simulée de cet algorithme jusqu'à sa conclusion.
2. Expliquer pourquoi il est raisonnable d'utiliser une liste pour stocker les étiquettes lues, et pourquoi faire de même pour la forêt à traiter pose un problème de performances.
3. Quelle structure de données étudiée cette année peut-on utiliser pour stocker de manière efficace les arbres à traiter ?
4. Écrire la fonction `étiquettes_largeur`; on pourra se référer aux indications présentes dans le squelette fourni, et écrire au choix une fonction récursive terminale ou itérative.

3 Adressage

Dans un arbre binaire, un nœud situé à profondeur p (où la profondeur de la racine vaut 0) peut être repéré par une suite de p éléments (x_0, \dots, x_{p-1}) de $\{\leftarrow, \rightarrow\}$: on part de la racine et pour $i = 0 \dots p - 1$ on descend vers le fils gauche ou droit du nœud actuel suivant que x_i vaut \leftarrow ou \rightarrow . Informatiquement, les valeurs \leftarrow et \rightarrow seront codées par les booléens `false` et `true` ou par les entiers 0 et 1.

Pour l'arbre `exemple1`, on obtient :

Adresse	Étiquette
\emptyset	12
0	4
00	7
000	20
001	30
01	14
010	1
011	2
1	20

Exercice XXI.8

1. Dresser le tableau des adresses et étiquettes correspondantes pour l'arbre `exemple2`.
2. Écrire une fonction `lire_etiquette : bool list -> ('a, 'b) arbre -> ('a, 'b) token` prenant en arguments une adresse et un arbre et renvoyant l'étiquette correspondante. Si l'adresse ne correspond à aucun nœud, on lèvera une exception à l'aide de `failwith`.

```
utop[50]> lire_etiquette [false; true; false] exemple1;;
- : (int, int) token = F 1
utop[51]> lire_etiquette [false] exemple1;;
- : (int, int) token = N 4
```

3. Écrire une fonction `incremente : (int, int) arbre -> adresse -> (int, int) arbre` prenant en entrée un arbre T et une adresse s et renvoyant l'arbre T' obtenu à partir de T en incrémentant de 1 l'étiquette du nœud d'adresse s . À nouveau, on lèvera une exception si l'adresse est invalide.
4. Écrire une fonction `tableau_adresses : (int, int) arbre -> unit` affichant la liste des adresses de l'arbre avec les étiquettes correspondantes. On s'aidera de la fonction `affiche_avec_adresse` fournie.

```
utop[58]> affiche_avec_adresse (12, [true; false; false]);;
100 : 12
utop[59]> tableau_adresses exemple1;;
: 12
0 : 4
00 : 7
01 : 14
010 : 1
011 : 2
1 : 20
```

4 Reconstruction d'un arbre

On s'intéresse ici à la possibilité d'écrire des fonctions réciproques de `prefixe`, `postfixe`, `largeur`, `infixe`. Notre entrée est donc une ('a, 'b) token **list** (dont on sait à quel ordre elle correspond), et notre sortie doit être l'unique ('a, 'b) arbre donnant cette liste.

Exercice XXI.9

1. Donner deux arbres distincts dont le parcours infixé produit la liste [F 0; N 1; F 2; N 3; F 4]. Que peut-on en conclure ?
2. Pour reconstruire l'arbre à partir de son parcours postfixé, on propose l'algorithme illustré ci-dessous :

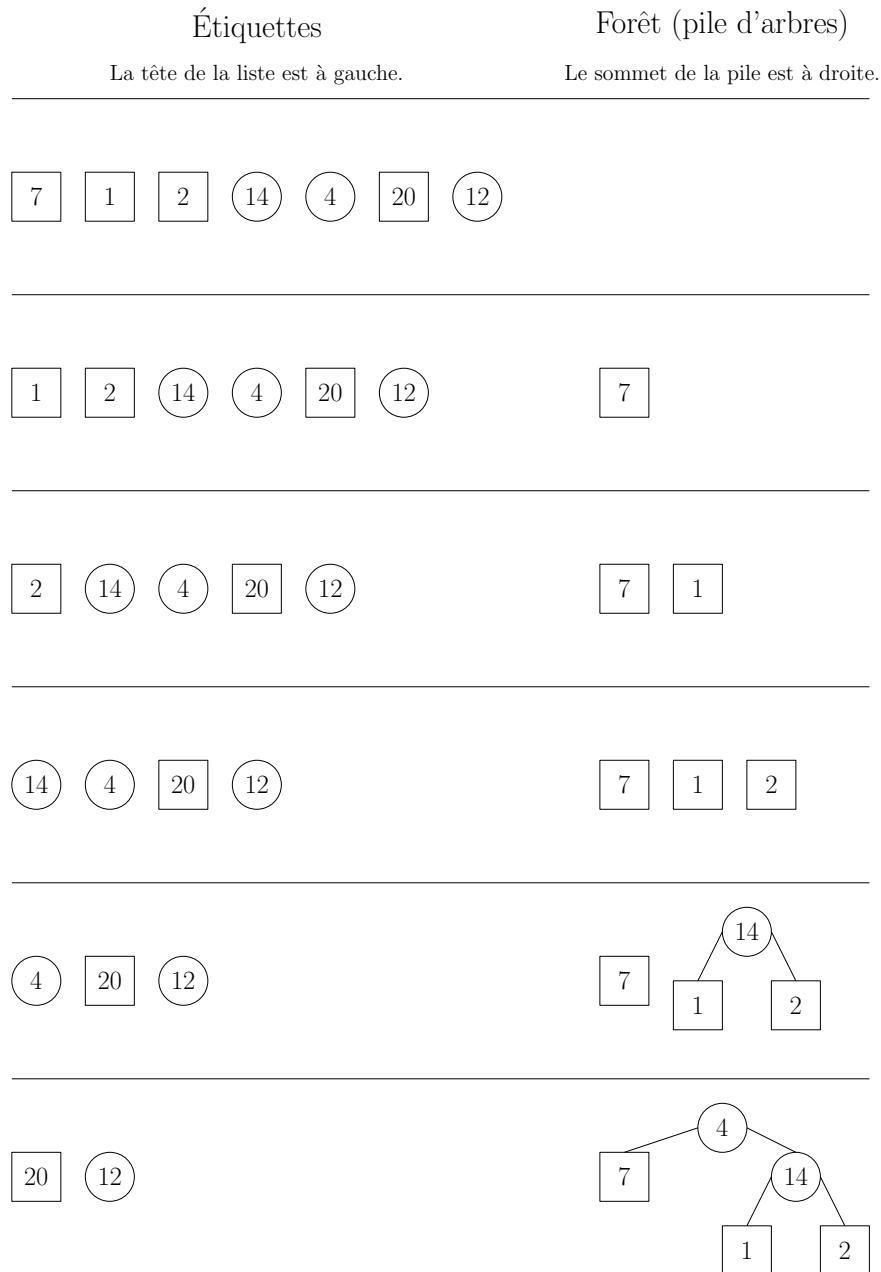


FIGURE XXI.4 – Principe de reconstruction à partir du parcours postfixé.

- a. Terminer la simulation de l'exécution de l'algorithme sur l'exemple donné.
- b. À l'aide d'une fonction auxiliaire

aux : ('a, 'b) arbre list -> ('a, 'b) token list -> ('a, 'b) arbre, écrire une fonction lire_postfixe reconstruisant un arbre à partir du résultat de son parcours postfixe.

3. Écrire une fonction lire_prefixe (on peut utiliser une idée similaire).
4. Écrire une fonction lire_largeur (il faudra utiliser une file).

5 Au cas où

Exercice XXI.10 – Retour sur les nombres de Catalan

On considère les types suivants :

```
type cote = G | D

(* arbre binaire entier non étiqueté *)
type arbre = Feuille | Noeud of arbre * arbre

(* on annote chaque noeud interne avec le cardinal du sous-arbre
correspondant *)
type arbre_annotate =
| N of int * arbre_annotate * arbre_annotate
| F
```

ainsi que la fonction suivante qui renvoie le cardinal (nombre total de nœuds) d'un arbre annoté :

```
(* arbre_annotate -> int *)
let card = function
| F -> 1
| N(taille, _, _) -> taille
```

1. Écrire une fonction annotate : arbre -> arbre_annotate (dont la spécification devrait aller de soi).
2. Écrire une fonction insere : arbre_annotate * int * cote -> arbre_annotate * int correspondant à la fonction insere du cours. L'entier pris en argument sera donc un numéro valable de nœud pour l'arbre de départ (dans l'ordre préfixe) et l'entier renvoyé un numéro valable de feuille dans l'arbre d'arrivée.
3. Écrire une fonction efface : arbre_annotate * int -> arbre_annotate * int * cote, réciproque de la précédente.
4. Écrire une fonction test prenant en entrée un arbre et vérifiant sur cet arbre que les deux fonctions précédentes sont bien réciproques l'une de l'autre (on fera toutes les insertions et toutes les suppressions légales).

STRUCTURES DE DONNÉES CHAÎNÉES EN C

On rappelle les options de compilations (particulièrement importantes aujourd’hui) :

```
$ gcc -o fichier.out -fsanitize=address,undefined -Wall -Wextra -Wvla fichier.c
```

On ignorerai les *warnings* relatifs aux fuites de mémoire jusqu’à avoir écrit la fonction `free_list` (mais on y sera très attentif ensuite).

I Listes simplement chaînées

Plusieurs variations sont possibles pour définir des listes simplement chaînées en C. Ici, on va utiliser l’une des plus simples :

```
typedef int datatype;

struct Node {
    datatype data;
    struct Node *next;
};

typedef struct Node node;
```

- En général, on n’aura pas besoin de supposer que `datatype` est égal à `int`, mais on supposera en revanche que c’est un type numérique (pour pouvoir faire des additions, des comparaisons...).
- Une liste sera simplement un `node`* :
 - la liste vide est représentée par le pointeur `NULL`;
 - une liste non vide est représentée par un pointeur vers son premier noeud.

Exercice XXII.I – Création de listes

p. 545

1. Écrire une fonction créant un nouveau node (et renvoyant un pointeur vers ce node), avec le champ `data` initialisé avec l’argument fourni et le champ `next` initialisé à `NULL`.

```
node *new_node(datatype data);
```

2. Écrire une fonction `cons` prenant en entrée une liste `list` et un argument `data`, et renvoyant une liste constitué d’un noeud contenant `data`, suivi de `list`.

```
node *cons(node *list, datatype data);
```

3. Écrire une fonction `from_array` ayant le prototype suivant :

```
node *from_array(datatype array[], int len);
```

- `array` est un tableau d’objets de type `datatype` de longueur `len`.
- La fonction renvoie une liste contenant les mêmes éléments que `array`, dans l’ordre (autrement dit, l’élément de tête de la liste est `array[0]`).

À partir de ce point, vous trouverez dans le squelette un certain nombre de fonctions vous permettant de tester votre code. Il est strictement interdit de passer à la question $n + 1$ si le test associé à la question n échoue !

Exercice XXII.2 – Parcours de listes

p. 545

- Écrire une fonction `free_list` qui libère toute la mémoire utilisée par une liste.

```
void free_list(node *u);
```

- Écrire une fonction `length` calculant la longueur d'une liste.

```
int length(node *u);
```

On écrira une fonction purement itérative.

- Écrire une fonction `print_list` affichant les éléments d'une liste. On produira l'affichage suivant, terminé ou non par un retour à la ligne suivant la valeur du paramètre `newline` :

```
[7 1 3 4 12]
```

On pourra supposer ici que `datatype` est égal à `int`.

```
void print_list(node *u, bool newline);
```

- Écrire une fonction `to_array` convertissant une liste de longueur n en un tableau de taille n (l'élément en tête de la liste se retrouvera à l'indice 0).

```
datatype *to_array(node *u);
```

- Écrire une fonction `is_equal` qui teste l'égalité de deux listes. On parle ici d'égalité *structurelle* (les deux listes ont les mêmes éléments, dans le même ordre) et non d'égalité *physique* (les deux pointeurs désignent la même liste).

```
bool is_equal(node *u, node *v);
```

Exercice XXII.3 – Ordre et tri

p. 547

- Écrire une fonction (non récursive) `is_sorted` qui prend en entrée une liste et renvoie un booléen indiquant si elle est triée par ordre croissant.

```
bool is_sorted(node *u);
```

- Écrire une fonction récursive `insert_rec` ayant le prototype suivant :

```
node *insert_rec(node *u, datatype x);
```

- u est une liste (éventuellement vide) supposée triée par ordre croissant.
- La fonction renvoie une liste triée par ordre croissant contenant les mêmes éléments que u , plus une occurrence de x .
- Cette fonction créera un seul nouveau nœud !

- Faire un schéma mémoire illustrant ce qui se passe, en OCaml d'une part et en C d'autre part, lorsque :

- on part de u vide et l'on fait $v = \text{insert}(u, 10)$ (ou **let** $v = \text{insert } u \ 10$);
 - on part de $u = (1, 2, 5, 6)$ et l'on fait $v = \text{insert}(u, 0)$ (ou l'équivalent OCaml);
 - on part de $u = (1, 2, 5, 6)$ et l'on fait $v = \text{insert}(u, 4)$ (ou l'équivalent OCaml).
4. Pourquoi vaut-il mieux considérer que la variable u est « invalide » après l'instruction $v = \text{insert}(u, x)$ (autrement dit, pourquoi vaut-il mieux se limiter à des instructions du type $u = \text{insert}(u, x)$)?
 5. Pourquoi serait-il problématique d'avoir en C une fonction d'insertion se comportant comme en OCaml?
 6. Écrire une fonction récursive `insertion_sort_rec` prenant en entrée une liste (éventuellement vide) et renvoyant une liste triée contenant les mêmes éléments. La liste renvoyée ne partagera pas sa représentation mémoire avec la liste initiale (qui n'aura pas été modifiée).

```
node *insertion_sort_rec(node *u);
```

Exercice XXII.4

p. 548

1. Écrire une fonction `reverse_copy` qui prend en argument une liste et renvoie une copie de cette liste, à l'envers. La liste et sa copie seront totalement indépendantes en mémoire.

```
node *reverse_copy(node *u);
```

2. Écrire une fonction `copy_rec`, récursive, qui effectue une copie d'une liste (à l'endroit).

```
node *copy_rec(node *u);
```

3. Écrire une fonction `copy` ayant la même spécification mais n'étant pas récursive.

4. Écrire une fonction `reverse` renversant une liste « en place » (c'est-à-dire sans créer aucun nouveau nœud). On renverra un pointeur vers le nœud de tête de la nouvelle liste (qui était donc le nœud de queue de l'ancienne liste).

```
node *reverse(node *u);
```

Exercice XXII.5

p. 549

Nous avons déjà utilisé la fonction `scanf` : l'appel `scanf("%d", &n)`, par exemple, lit un entier sur l'entrée standard (en commençant par sauter les caractères d'espacement) et stocke sa valeur dans la variable n (qui doit avoir été préalablement définie). Cependant, nous avons omis un point important : `scanf` renvoie un entier. Cet entier est égal :

- à la constante (strictement négative) prédéfinie `E0F` si l'on est arrivé à la fin de l'entrée sans rien pouvoir lire ;
- au nombre d'éléments que l'on a réussi à lire, sinon. Par exemple, un appel `scanf("%d %f", &n, &x)` renverra 2 si l'entrée commence par "12 3.14", 1 si elle commence par "12 P3.14" et 0 si elle commence par "bonjour 17".

1. Écrire une fonction `read_list` qui lit des entiers sur l'entrée standard tant que c'est possible, et renvoie la liste lue (dans l'ordre). On s'arrêtera si l'on obtient `E0F` et aussi si l'on obtient 0 (comme résultat de `scanf`).
2. Écrire un programme complet qui lit une série d'entiers sur l'entrée standard et écrit ces entiers par ordre croissant sur la sortie standard, à raison d'un par ligne.
3. Utiliser ce programme pour lire un fichier d'entiers et écrire un fichier contenant les mêmes entiers triés par ordre croissant.

2 Piles et files

En général, il est nettement plus pertinent d'utiliser des tableaux plutôt que des listes chaînées pour implémenter les piles et les files en C (les tableaux peuvent être dynamiques si l'on ne peut pas se contenter de structures ayant une capacité fixée à la création). Il faut donc voir ce qui suit comme des exercices, intéressants dans le cadre scolaire mais ne correspondant pas vraiment à ce qu'on ferait en réalité.

2.1 Pile à l'aide d'une liste chaînée

On pourrait fournir l'interface minimale d'une pile en utilisant simplement un `node*` (une liste, autrement dit). Ici, on choisit une variante légèrement différente :

```
struct Stack {
    int len;
    node* top;
};

typedef struct Stack stack;
```

Exercice XXII.6

p. 550

Écrire les fonctions suivantes :

1. `empty_stack` qui renvoie un pointeur vers une nouvelle pile vide ;
2. `peek` qui renvoie l'élément situé au sommet d'une pile (on mettra un `assert` pour vérifier que la pile n'est pas vide) ;
3. `push` qui rajoute un élément au sommet d'une pile ;
4. `pop` qui supprime l'élément situé au sommet d'une pile, et le renvoie (à nouveau, on mettra une assertion pour générer une erreur prévisible si la pile est vide) ;
5. `free_stack` qui libère toute la mémoire associée à une pile.

```
stack *empty_stack(void);
datatype peek(stack *s);
void push(stack *s, datatype x);
datatype pop(stack *s);
void free_stack(stack *s);
```

On fera bien attention à maintenir l'invariant sur la longueur (`s.len` doit toujours être égal à la longueur de la liste `s.top`).

2.2 File à l'aide d'une liste simplement chaînée

On peut implémenter une file de manière efficace¹ en utilisant une liste simplement chaînée, à condition que cette liste soit mutable. En C cela ne posera aucun problème, mais en OCaml cela demanderait de définir un nouveau type liste. L'idée a été expliquée dans le chapitre *Piles et files*, on remet les illustrations :

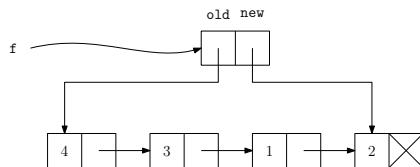


FIGURE XXII.7 – La file $\leftarrow (4, 3, 1, 2) \leftarrow$ (4 est l'élément le plus ancien).

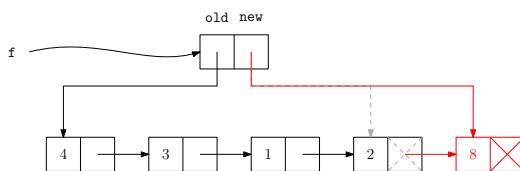


FIGURE XXII.8 – Ajout de l'élément 8.

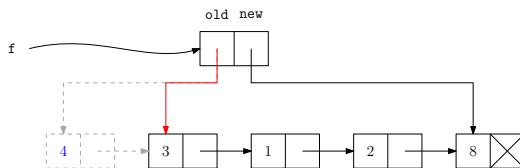


FIGURE XXII.9 – Extraction d'un élément.

À nouveau, on choisit de stocker également la longueur dans la structure de file. On obtient donc :

```
struct Queue {
    int len;
    node *left;
    node *right;
};

typedef struct Queue queue;
```

La convention est la même que sur les illustrations :

- les pointeurs de la liste vont de la gauche vers la droite ;
- les insertions se font à droite ;
- les extractions se font à gauche.

Exercice XXII.7

p. 551

1. Pourrait-on (efficacement) faire des insertions à gauche ? des extractions à droite ?
2. Écrire les fonctions suivantes, dont la spécification devrait être claire :

1. C'est-à-dire avec insertion et extraction en $O(1)$, le facteur constant étant en revanche assez mauvais par rapport à ce qu'on obtiendrait avec un tableau.

```
queue *empty_queue(void);

void free_queue(queue *q);

datatype peek_left(queue *q);

void push_right(queue *q, datatype data);

datatype pop_left(queue *q);
```

Exercice XXII.8 – Nombres de Hamming

p. 552

En reprenant l'algorithme du DS 2, écrire une fonction hamming prenant en entrée un entier n et renvoyant un tableau de taille n contenant les n plus petits nombres de Hamming, dans l'ordre.

```
int *hamming(int n);
```

3 Retour sur les listes chaînées**Exercice XXII.9**

p. 553

Écrire une version purement itérative du tri insertion.

Exercice XXII.10

p. 553

1. Écrire une fonction split qui prend en argument une liste u et un entier n et a le comportement suivant :

- elle renvoie une liste composée des $|u| - n$ derniers éléments de u ;
- elle modifie u de manière à ce qu'elle ne contienne plus que ses n premiers éléments.

Cette fonction sera purement itérative, ne créera aucun nouveau nœud, et traitera les cas où $n > |u|$ avec des assertions.

```
node *split(node *u, int n);
```

2. Écrire une fonction merge qui prend en entrée deux listes supposées croissantes et renvoie leur fusion. Cette fonction sera purement itérative et ne créera aucun nouveau nœud.

```
node *merge(node *u, node *v);
```

3. Écrire une fonction merge_sort qui trie une liste en utilisant l'algorithme du tri fusion. Ce tri se fera « en place », dans le sens où l'on ne créera aucun nouveau nœud (on ne fera que réarranger des pointeurs).

```
node *merge_sort(node *u);
```

Solutions

Correction de l'exercice XXII.1 page 539

1. On alloue un nouveau nœud (sur le tas) et on l'initialise :

```
node *new_node(datatype data){  
    node *new = malloc(sizeof(node));  
    new->data = data;  
    new->next = NULL;  
    return new;  
}
```

2. On renvoie un pointeur vers un nouveau nœud, dont on a fait pointer le champ next vers la liste fournie :

```
node *cons(node *list, datatype data){  
    node *new = new_node(data);  
    new->next = list;  
    return new;  
}
```

3. Il faut parcourir le tableau à l'envers pour obtenir la liste dans le bon ordre :

```
node *from_array(datatype array[], int len){  
    node *current = NULL;  
    for (int i = len - 1; i >= 0; i--) {  
        current = cons(current, array[i]);  
    }  
    return current;  
}
```

Correction de l'exercice XXII.2 page 540

1. On libère les nœuds dans l'ordre :

```
void free_list(node* n) {  
    while (n != NULL) {  
        node *next = n->next;  
        free(n);  
        n = next;  
    }  
}
```

Il est possible de procéder récursivement :

```
void free_list_rec(node *n){  
    if (n == NULL) { return; }  
    free_list_rec(n->next);  
    free(n);  
}
```

Cependant, cette fonction (qui commence en fait par libérer le *dernier* noeud), n'est pas récursive terminale et peut donc poser des problèmes de dépassement de pile (la profondeur de récursion est égale à la longueur de la liste).

2. Pas de difficulté :

```
int length(node* list){
    int i = 0;
    while (list != NULL) {
        i++;
        list = list->next;
    }
    return i;
}
```

3. Pour respecter exactement l'affichage demandé, il faut être méticuleux : tous les éléments sauf *le dernier* sont suivis d'un espace.

```
void print_list(node *n, bool newline){
    printf("[");
    while (n != NULL && n->next != NULL) {
        printf("%d ", n->data);
        n = n->next;
    }
    if (n != NULL) {
        printf("%d", n->data);
    }
    printf("]");
    if (newline) { printf("\n"); }
}
```

4. On détermine la longueur, on alloue, puis on parcourt la liste pour remplir le tableau :

```
datatype *to_array(node *u){
    int len = length(u);
    datatype *t = malloc(len * sizeof(t));
    for (int i = 0; i < len; i++){
        t[i] = u->data;
        u = u->next;
    }
    return t;
}
```

5. Attention, il est facile d'écrire une fonction qui considère comme égales les listes [1 2 3] et [1 2] !

```
bool is_equal(node *u, node *v){
    while (u != NULL && v != NULL){
        if (u->data != v->data) { return false; }
        u = u->next;
        v = v->next;
    }
    return (u == NULL && v == NULL);
}
```

Correction de l'exercice XXII.3 page 540

1. À nouveau, on privilégie une version purement itérative :

```
bool is_sorted(node *u){
    if (u == NULL) { return true; }
    while (u->next != NULL){
        if (u->data > u->next->data) { return false; }
        u = u->next;
    }
    return true;
}
```

2. En récursif ça ne pose aucun problème :

```
node *insert_rec(node *u, datatype x){
    if (u == NULL || u->data >= x) { return cons(u, x); }
    u->next = insert_rec(u->next, x);
    return u;
}
```

3. On obtient les schémas suivants :

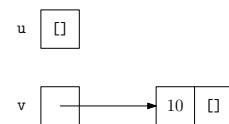
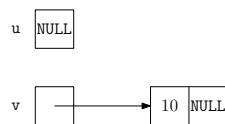


FIGURE XXII.10 – C, insertion dans une liste vide.

FIGURE XXII.11 – OCaml, insertion dans une liste vide.

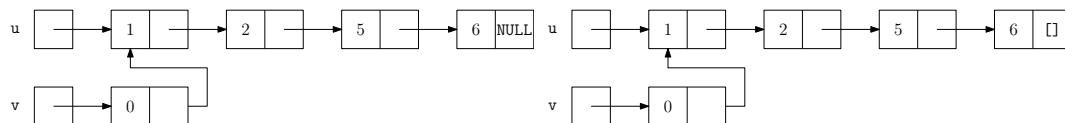


FIGURE XXII.12 – C, insertion en tête.

FIGURE XXII.13 – OCaml, insertion en tête.

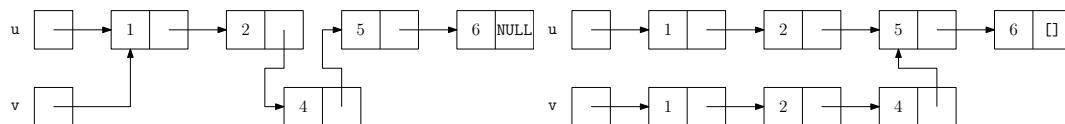


FIGURE XXII.14 – C, insertion en milieu de liste.

FIGURE XXII.15 – OCaml, insertion en milieu de liste.

4. Après un appel $v = \text{insert}(u, x)$, on a plusieurs situations possibles :

- Soit u était vide, et dans ce cas u vaut toujours **NULL** et v n'a fondamentalement « rien à voir » avec u .
- Soit u était non vide et on a inséré en tête. Dans ce cas, u pointe vers le deuxième élément de v , ce qui est une situation dangereuse. Une modification de l'un peut ou non affecter l'autre suivant les cas, et la libération est très dangereuse : si l'on appelle `free_list(v)`, alors u pointe vers un bloc qui n'existe plus ; si l'on appelle `free_list(u)`, alors l'élément de tête de v pointe vers un bloc qui n'existe plus.
- Soit u était non vide et on a inséré ailleurs qu'en tête. Dans ce cas, u et v désignent tous deux la nouvelle liste (celle avec un élément en plus). Le point vraiment problématique, c'est que la situation est complètement différente du cas précédent !

5. Si la fonction C se comportait comme la fonction OCaml (u n'est pas modifiée, le début est recopié avec un élément de plus et la fin est partagée entre u et v), on serait essentiellement toujours dans un cas équivalent au deuxième cas de la question précédente. C'est (très) problématique pour les raisons précédemment énoncées (en particulier, il est essentiellement impossible de libérer correctement la mémoire).

Il serait bien sûr possible d'écrire une fonction qui agit comme la version OCaml, sauf qu'elle libère les nœuds de u au fur et à mesure qu'elle les copie. Cependant, ce serait assez absurde : autant ne pas faire de copie.

En OCaml, il n'y a pas de problème pour deux raisons :

- la libération de la mémoire est automatique ;
- les listes ne sont pas mutables, donc il n'est pas gênant d'avoir une situation dans laquelle u et v sont partiellement aliasées (même sans savoir quelle partie de leur représentation mémoire elles partagent).

6. Pas de difficulté majeure :

```
node *insertion_sort_rec(node *u){
    if (u == NULL) { return NULL; }
    return insert_rec(insertion_sort_rec(u->next), u->data);
}
```

Correction de l'exercice XXII.4 page 541

1. Essentiellement, c'est une copie inversée de pile, ce qui ne pose pas de problème :

```
node* reverse_copy(node* u){
    node *v = NULL;
    while (u != NULL){
        v = cons(v, u->data);
        u = u->next;
    }
    return v;
}
```

2. La version récursive est très simple :

```
node *copy_rec(node *u){
    if (u == NULL) { return NULL; }
    return cons(copy_rec(u->next), u->data);
}
```

3. La version itérative est nettement plus délicate, quelques expériences avec C Tutor peuvent sans doute être utiles :

- u désigne le nœud courant de la liste passée en argument (qui varie au cours de l'exécution) ;
- current désigne le nœud courant de la copie ;
- start sauvegarde le premier nœud de la copie (puisque c'est ce que l'on doit renvoyer à la fin).

```

node* copy(node* u){
    if (u == NULL) { return NULL; }
    node* current = new_node(u->data);
    node* start = current;
    while (u->next != NULL) {
        u = u->next;
        node* new = new_node(u->data);
        current->next = new;
        current = current->next;
    }
    return start;
}

```

4. C'est aussi assez délicat. Noter que cette fois, c'est le dernier noeud (devenu le premier par inversion des pointeurs) que l'on doit renvoyer :

```

node *reverse(node *u){
    if (u == NULL) { return u; }
    node *current = u;
    node *next = u->next;
    current->next = NULL;
    while (next != NULL){
        node *tmp = current;
        current = next;
        next = next->next;
        current->next = tmp;
    }
    return current;
}

```

Correction de l'exercice XXII.5 page 541

1. Cette version n'est pas très idiomatique, mais elle est conforme au programme et facile à comprendre :

```

node *read_list(void){
    node *u = NULL;
    while (true){
        int x;
        int nb = scanf("%d", &x);
        if (nb == 1){
            u = cons(u, x);
        } else {
            return u;
        }
    }
}

```

2. On commence par écrire une petite fonction pour gérer l'écriture :

```
void print_list_2(node *u){
    while (u != NULL){
        printf("%d\n", u->data);
        u = u->next;
    }
}
```

Ensuite, le main est très simple :

```
int main(void) {

    node *u = read_list();
    node *v = insertion_sort_rec(u);
    print_list_2(v);

    free_list(u);
    free_list(v);

    return 0;
}
```

3. Il suffit d'utiliser la ligne de commande suivante (où l'on redirige l'entrée standard et la sortie standard) :

```
$ ./linked.out <input.txt >output.txt
```

Correction de l'exercice XXII.6 page 542

1. Tant qu'on pense bien à allouer sur le tas, il n'y a pas de problème :

```
stack *empty_stack(void){
    stack *s = malloc(sizeof(stack));
    s->len = 0;
    s->top = NULL;
    return s;
}
```

2. Aucun problème :

```
datatype peek(stack *s){
    assert(s->len > 0);
    return s->top->data;
}
```

3. Il faut juste penser à mettre à jour la longueur :

```
void push(stack *s, datatype x){
    s->top = cons(s->top, x);
    s->len = s->len + 1;
}
```

4. Rien de compliqué, mais il faut faire attention à faire les choses dans l'ordre.

```
datatype pop(stack *s){
    assert(s->len > 0);
    node *top = s->top;
    datatype res = top->data;
    s->top = top->next;
    s->len = s->len - 1;
    free(top);
    return res;
}
```

5. On libère la liste, puis la **struct** elle-même.

```
void free_stack(stack *s){
    free_list(s->top);
    free(s);
}
```

Correction de l'exercice XXII.7 page 543

- Il n'y a aucun problème pour insérer à gauche. En revanche, il faudrait parcourir toute la liste pour supprimer à droite puisqu'on n'a pas de pointeur vers l'avant-dernier élément.
- Pour push_right, il faut penser à traiter séparément le cas où la file est vide :

```
queue *empty_queue(void){
    queue *q = malloc(sizeof(queue));
    q->len = 0;
    q->left = NULL;
    q->right = NULL;
    return q;
}

void free_queue(queue *q){
    free_list(q->left);
    free(q);
}

datatype peek_left(queue *q){
    assert(q->len > 0);
    return q->left->data;
}

void push_right(queue *q, datatype data){
    node *n = new_node(data);
    if (q->right == NULL) {
        q->right = n;
        q->left = n;
    } else {
        q->right->next = n;
        q->right = n;
    }
    q->len++;
}
```

Pour pop_left, le cas particulier est pour une file ne contenant qu'un seul élément :

```

datatype pop_left(queue *q){
    assert(q->len > 0);
    datatype res = q->left->data;
    if (q->len == 1){
        free(q->left);
        q->left = NULL;
        q->right = NULL;
    } else {
        node *tmp = q->left->next;
        free(q->left);
        q->left = tmp;
    }
    q->len--;
    return res;
}

```

Correction de l'exercice XXII.8 page 544

J'ai omis la définition (triviale) de la fonction min3 :

```

int *hamming(int n){
    int *t = malloc(n * sizeof(int));
    queue *q2 = empty_queue();
    queue *q3 = empty_queue();
    queue *q5 = empty_queue();
    push_right(q2, 1);
    push_right(q3, 1);
    push_right(q5, 1);
    for (int i = 0; i < n; i++){
        int x2 = peek_left(q2);
        int x3 = peek_left(q3);
        int x5 = peek_left(q5);
        int x = min3(x2, x3, x5);
        t[i] = x;

        if (x == x2) { pop_left(q2); }
        if (x == x3) { pop_left(q3); }
        if (x == x5) { pop_left(q5); }

        push_right(q5, 5 * x);
        if (x % 5 != 0){
            push_right(q3, 3 * x);
            if (x % 3 != 0){
                push_right(q2, 2 * x);
            }
        }
    }

    free_queue(q2);
    free_queue(q3);
    free_queue(q5);

    return t;
}

```

Correction de l'exercice XXII.9 page 544

Il faut faire très attention pour l'insertion :

```
node *insert_it(node *u, datatype x){
    if (u == NULL || u->data >= x) { return cons(u, x); }
    node *current = u;
    while (current != NULL){
        if (current->next != NULL && current->next->data < x){
            current = current->next;
        } else {
            current->next = cons(current->next, x);
            break;
        }
    }
    return u;
}
```

Ensuite le tri lui-même ne pose pas vraiment de problème :

```
node *insertion_sort_it(node *u){
    node *v = NULL;
    while (u != NULL){
        v = insert_it(v, u->data);
        u = u->next;
    }
    return v;
}
```

Correction de l'exercice XXII.10 page 544

1. C'est assez naturel mais il est quand même facile de ne pas respecter parfaitement la spécification :

```
node *split(node *u, int n){
    node *v = u;
    while (n - 1 > 0){
        assert(v != NULL);
        v = v->next;
        n--;
    }

    assert(v != NULL);
    node *tmp = v->next;
    v->next = NULL;
    return tmp;
}
```

2. Il y a peut-être plus simple, mais en tout cas c'est pénible :

```
node *merge(node *u, node *v){  
    if (u == NULL) { return v; }  
    if (v == NULL) { return u; }  
    node *res = NULL;  
    if (u->data <= v->data){  
        res = u;  
        u = u->next;  
    } else {  
        res = v;  
        v = v->next;  
    }  
    node *current = res;  
    while (u != NULL || v != NULL){  
        if (v == NULL || (u != NULL && u->data <= v->data)){  
            current->next = u;  
            u = u->next;  
            current = current->next;  
        } else {  
            current->next = v;  
            v = v->next;  
            current = current->next;  
        }  
    }  
    return res;  
}
```

3. Cette fonction est assez simple. Attention, l'appel `split(u, n/2)` a un effet secondaire (que l'on exploite) sur `u`.

```
node *merge_sort(node *u){  
    int n = length(u);  
    if (n <= 1) { return u; }  
    node *second_half = merge_sort(split(u, n/2));  
    node *first_half = merge_sort(u);  
    return merge_it(first_half, second_half);  
}
```

CE DUC Y PARLE

I Structure de trie

Considérons l'ensemble de mots suivant :

```
let mots = ["diane"; "dire"; "diva"; "divan"; "divin"; "do"; "dodo";
            "dodu"; "don"; "donc"; "dont"; "ame"; "ames"; "amen"]
```

On peut représenter cet ensemble sous forme d'un arbre d'arité variable, où un nœud grisé signifie que le mot correspondant (c'est-à-dire le mot qu'on lit en allant de la racine au nœud) appartient au dictionnaire : on parle de *trie* pour cette structure de données.

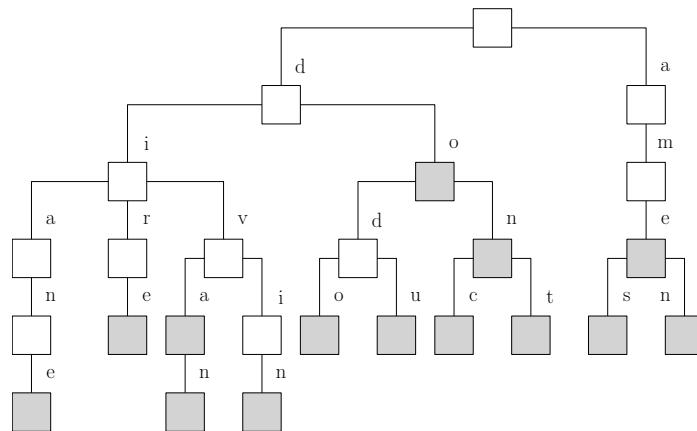


FIGURE XXIII.1 – Représentation arborescente de mots.

Pour simplifier la programmation, nous allons utiliser une technique assez courante :

- on choisit un caractère qui n'apparaît dans aucun mot de notre dictionnaire (pour nous, ce sera \$);
- ce caractère devient un « marqueur de fin de mot » : il est ajouté à la fin de tous les mots.

Il n'est alors plus nécessaire de distinguer les nœuds correspondant à un mot du dictionnaire et les autres : les mots du dictionnaire sont exactement ceux qui correspondent aux feuilles de notre arbre.

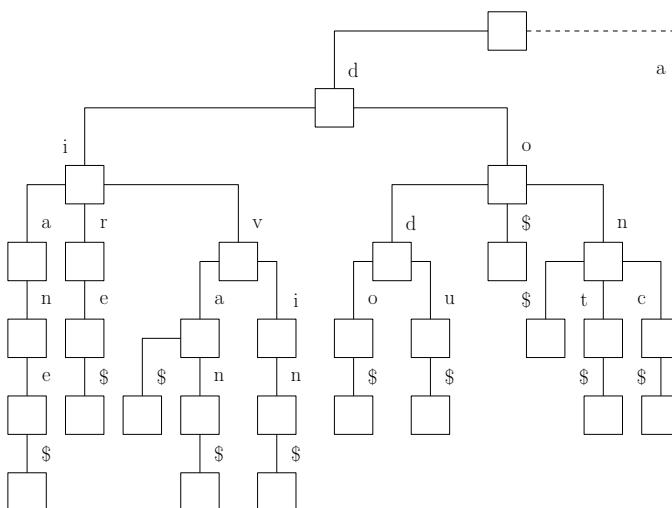


FIGURE XXIII.2 – Partie gauche de l'arbre en figure XXIII.1, avec marqueurs de fin de mot.

Ensuite, nous avons déjà parlé de plusieurs représentations mémoire possibles pour un arbre d'arité quelconque :

- chaque nœud peut contenir un tableau d'enfants (ici, l'enfant correspondant au caractère `c` pourrait être placé dans la case `int_of_char c` du tableau);
- chaque nœud peut aussi contenir une liste d'enfants, ou plutôt une liste de couples (caractère, enfant) (puisque les arêtes portent des étiquettes);
- enfin, il est possible de « binariser » l'arbre.

C'est cette solution que nous allons choisir, et l'on définit donc le type suivant :

```
type dict =
| V
| N of char * dict * dict
```

On obtient alors (en omettant les nœuds `V`) :

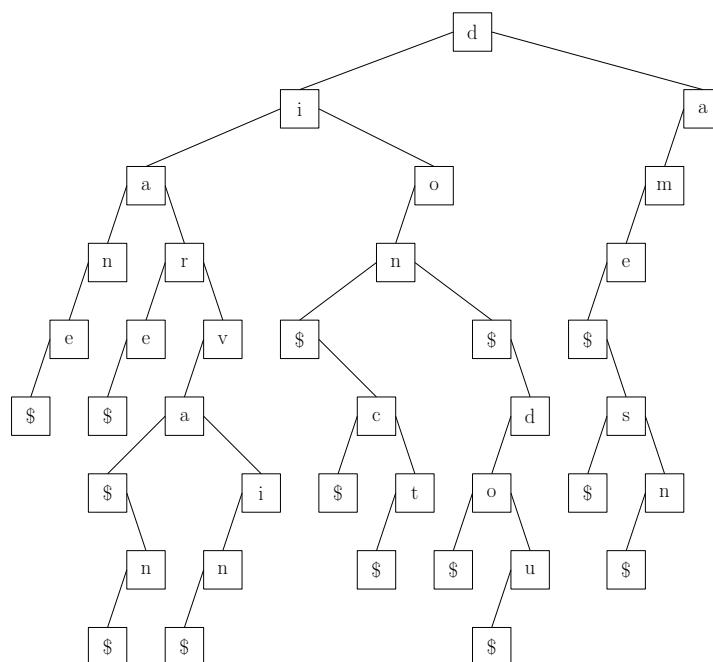


FIGURE XXIII.3 – Version binaire de l'arbre de la figure XXIII.2.

On impose deux contraintes sur nos arbres :

- un nœud étiqueté par `$` a forcément `V` comme fils gauche;
- un nœud étiqueté par un caractère autre que `$` n'a jamais `V` comme fils gauche.

Exercice XXIII.I

p. 561

Définir une fonction `est_bien_forme` qui vérifie si un `dict` est bien formé.

```
est_bien_forme : dict -> bool
```

Remarque

À partir de maintenant, les tries passés en argument seront systématiquement supposés bien formés, et toute fonction renvoyant un trie devra obligatoirement renvoyer un trie bien formé.

Un mot (une suite finie de caractères) sera dit *bien formé* s'il contient exactement un caractère `$`, placé en dernière position.

Remarque

Un mot bien formé est donc nécessairement non vide.

Exercice XXIII.2

p. 561

Donner une définition (inductive) de la fonction φ qui à un arbre binaire (bien formé) du type ci-dessus associe un ensemble de mots bien formés. Dans l'exemple ci-dessus, on a :

$$\varphi(t) = \{"diane\$", "dire$", "diva$", "divan$", "divin$", "don" "donc$", "dont$", "dodo$", "dodu$", "ame$", "ames$", "amen$\"}$$

2 Fonctions utilitaires

On définit l'alias de type suivant :

```
type mot = char list
```

Exercice XXIII.3

p. 561

- Écrire une fonction `mot_of_string` prenant en entrée une chaîne de caractères et renvoyant la liste de ses caractères, avec un caractère \$ rajouté à la fin.

```
mot_of_string : string -> mot
```

```
# mot_of_string "bonjour";;
- : char list = ['b'; 'o'; 'n'; 'j'; 'o'; 'u'; 'r'; '$']
```

- Écrire une fonction `afficher` qui prend en entrée une liste de caractères et affiche le mot correspondant, suivi d'un retour à la ligne. Les éventuels caractères \$ seront ignorés.

```
afficher : mot -> unit
```

```
# afficher_liste ['b'; 'o'; 'n'; 'j'; 'o'; 'u'; 'r'; '$'];
bonjour
- : unit = ()
```

Un objet de type mot sera dit bien formé s'il contient exactement un caractère \$, placé en dernière position.

3 Opérations élémentaires sur les tries**Exercice XXIII.4**

p. 561

- Écrire une fonction `cardinal` renvoyant le nombre de mots bien formés contenus dans un dictionnaire.

```
cardinal : dict -> int
```

- Écrire une fonction `appartient` qui détermine si un certain mot bien formé appartient à un dictionnaire.

```
appartient : dict -> mot -> bool
```

Exercice XXIII.5

p. 562

1. Écrire une fonction `ajouter` qui ajoute un mot, supposé bien formé, à un dictionnaire.
2. Écrire une fonction `dict_of_list` prenant en entrée une liste de chaînes de caractères et renvoyant un dictionnaire contenant exactement les mots de cette liste (auxquels on a ajouté un \$).

```
ajouter : dict -> mot -> dict
dict_of_list : string list -> dict
```

Exercice XXIII.6

p. 562

1. Écrire une fonction `afficher_mots` qui affiche tous les mots bien formés appartenant à un dictionnaire (sans les \$ finals), à raison d'un mot par ligne.
2. Écrire une fonction `longueur_maximale` qui renvoie la longueur maximale d'un mot bien formé du dictionnaire (on ne comptera pas le \$ final, et l'on renverra -1 s'il n'y a pas de mot bien formé).
3. Écrire une fonction `afficher_mots_longs` qui affiche tous les mots de longueur supérieure ou égale à l'entier passé en argument (toujours sans compter le \$).

```
afficher_mots : dict -> unit
longueur_maximale : dict -> int
afficher_mots_longs : dict -> int -> unit
```

4 Lecture de fichier

Vous trouverez sur le serveur quelques fichiers contenant des mots, dont le format est très simple : chaque mot est sur une ligne. Ces mots ne contiennent que des lettres minuscules de a à z, sans signe diacritique (autrement dit, les accents, cédilles et autres ont été retirés des fichiers en français).

- Le fichier `ab.txt` contient 9 940 mots anglais, commençant tous par a ou par b.
- Le fichier `10000.txt` contient les 10 000 mots anglais les plus courants.
- Le fichier `nettoye.txt` contient 336 531 mots français, débarassés de leurs signes diacritiques.

Dans tous les cas, il y a quelques doublons (dûs à la suppression des signes diacritiques).

Exercice XXIII.7

p. 563

Écrire une fonction `lire_fichier` qui prend en entrée un nom de fichier (ou plutôt un chemin relatif vers un fichier) et renvoie le dictionnaire correspondant. On supposera que le format est celui décrit ci-dessus (un mot par ligne).

```
lire_fichier : string -> dict
```

```
# cardinal (lire_fichier "ab.txt");
- : int = 9938
# cardinal (lire_fichier "10000.txt");
- : int = 9989
# cardinal (lire_fichier "nettoye.txt");
- : int = 323422
```

5 Filtrage

Exercice XXIII.8

p. 563

- Écrire une fonction `calculer_occurrences` qui prend en entrée une chaîne de caractères `s` et renvoie un `int array` de longueur 256 tel que `t[i]` soit égal au nombre d'occurrences de `int_of_char i` dans `s`.
- Écrire une fonction `afficher_mots_contenus` qui prend en entrée un mot sous forme de chaîne de caractères (sans \$ final) et un dictionnaire, et affiche tous les mots du dictionnaire que l'on peut former en utilisant tout ou partie des lettres du mot fourni (en tenant compte des répétitions).
- Écrire une fonction `afficher_anagrammes` qui prend en entrée un mot sous forme de chaîne de caractères et affiche toutes ses anagrammes présentes dans le dictionnaire. Une *anagramme* d'un mot est un mot constitué exactement des mêmes lettres (avec le même nombre d'occurrences) mais dans un ordre différent (on considérera qu'un mot est anagramme de lui-même).
- Quel est sont les mots français les plus courts contenant toutes les voyelles ?

```
calculer_occurrences : string -> int array
afficher_mots_contenus : dict -> string -> unit
afficher_anagrammes : dict -> string -> unit
```

Exercice XXIII.9

p. 565

- Écrire une fonction `filtrer_mots_contenus` qui prend les mêmes arguments que `afficher_mots_contenus` mais renvoie un dictionnaire contenant les mots que l'on peut former. On produira directement le dictionnaire, sans commencer par produire la liste des mots.
- Écrire une fonction `filtrer_mots_contenant` qui fait la même chose que la précédente, sauf qu'on s'intéresse cette fois aux mots *à partir desquels* on peut former le mot fourni (c'est-à-dire ceux contenant toutes les lettres du mot fourni, en tenant compte des répétitions).
- Écrire une fonction similaire `filtrer_anagrammes`.

6 Décomposition en anagrammes

On appelle *décomposition en anagrammes* d'un mot `m` dans un dictionnaire `d` une suite de mots de `d` qui, mis bout à bout, forment un anagramme de `m`. Par exemple, "sans ame", "sa mes na", "a mes ans" sont des décompositions possibles de "massena" avec un dictionnaire français standard.

Exercice XXIII.10

p. 567

Dans cet exercice, on s'autorise à générer plusieurs fois une décomposition : par exemple, "sans ame" et "ame sans" (autrement dit, une décomposition en `n` mots sera générée $n!$ fois).

- Écrire une fonction `afficher_decompositions` qui affiche toutes les décompositions en anagrammes d'un mot dans un dictionnaire.
- Écrire une fonction `decompositions` qui renvoie un dictionnaire contenant toutes ces décompositions. La décomposition "sans ame", par exemple, sera stockée comme un mot de huit caractères (sans compter le \$ final), avec un caractère « espace » entre le « s » et le « a ».

```
afficher_decompositions : dict -> string -> unit
decompositions : dict -> string -> dict
```

Exercice XXIII.II

p. 568

Écrire une fonction `decompositions_uniques` qui génère le dictionnaire des décompositions dans lequel deux décompositions ne différant que par l'ordre des mots sont considérées comme identiques (et ne contenant donc que l'une de ces décompositions). On pourra par exemple ne générer que les décompositions pour lesquelles la suite des mots est croissante (dans l'ordre lexicographique).

```
decompositions_uniques : dict -> string -> dict
```

Solutions

Correction de l'exercice XXIII.1 page 556

```
let rec est_bien_forme = function
| V -> true
| N ('$', V, d) -> est_bien_forme d
| N (_, V, _) | N ('$', _, _) -> false
| N (c, g, d) -> est_bien_forme g && est_bien_forme d
```

Correction de l'exercice XXIII.2 page 557

En notant cs le mot formé du caractère c suivi du mot s, on a :

- $\varphi(V) = \emptyset$
- $\varphi(N($, V, d)) = \{ \$ \} \cup \varphi(d)$
- $\varphi(N(c, g, d)) = \{ cs \mid s \in \varphi(g) \} \cup \varphi(d)$ si $c \neq \$$.

Correction de l'exercice XXIII.3 page 557

```
let mot_of_string s =
  let n = String.length s in
  let rec aux i =
    if i = n then ['$']
    else s.[i] :: aux (i + 1) in
  aux 0

let rec afficher = function
| [] -> print_newline ()
| '$' :: xs -> afficher xs
| x :: xs -> print_char x; afficher xs
```

Correction de l'exercice XXIII.4 page 557

Le cardinal est égal au nombre de noeuds de la forme $N ('$', V, d)$.

```
let rec cardinal = function
| V -> 0
| N ('$', V, d) -> 1 + cardinal d
| N (c, g, d) -> cardinal g + cardinal d
```

Pour appartient, il n'y a pas de raison de traiter les noeuds étiquetés '\$' séparément :

```
let rec appartient dict mot =
  match dict, mot with
  | V, [] -> true
  | N (c, g, d), x :: xs ->
    if c = x then appartient g xs else appartient d mot
  | _ -> false
```

Correction de l'exercice XXIII.5 page 558

- La structure est similaire à celle de appartient.

```
let rec ajouter dict mot =
  match dict, mot with
  | V, [] -> V
  | V, x :: xs -> N (x, ajouter V xs, V)
  | N (c, g, d), x :: xs ->
    if c = x then N (c, ajouter g xs, d) else N (c, g, ajouter d mot)
  | _ -> failwith "mal formé"
```

- La solution la plus simple à écrire :

```
let rec dict_of_list u =
  match u with
  | [] -> V
  | s :: tl -> ajouter (dict_of_list tl) (mot_of_string s)
```

Notons que cette fonction n'est pas récursive terminale, ce qui peut être gênant si jamais on a une très longue liste de mots à ajouter (mais ce ne sera pas le cas ici). On peut donc préférer écrire :

```
let dict_of_list u =
  let rec aux restant d =
    match restant with
    | [] -> d
    | s :: tl -> aux tl (ajouter d (mot_of_string s)) in
      aux u V
```

Ces deux fonctions ne font pas les ajouts dans le même ordre, et ne renvoient pas, en général, le même dictionnaire. Cependant, elles sont toutes les deux correctes : dans les deux cas, le dictionnaire est bien formé et contient exactement les mots de la liste.

Correction de l'exercice XXIII.6 page 558

- On parcourt l'arbre en maintenant à jour une liste prefixe contenant le mot lu (actuellement) depuis la racine. Quand on tombe sur un noeud **N** ('\$', V, d), on affiche le mot (en retournant la liste, puisqu'elle se construit naturellement « à l'envers »). Pour un noeud **N** (c, g, d) avec c un caractère autre que \$, on descend à gauche en ajoutant c au préfixe, et à droite sans modifier le préfixe.

```
let afficher_mots dict =
  let rec aux dict prefixe =
    match dict with
    | V -> ()
    | N ('$', V, d) -> afficher (List.rev prefixe); aux d prefixe
    | N (c, g, d) -> aux g (c :: prefixe); aux d prefixe in
      aux dict []
```

- Il n'est pas nécessaire de traiter séparément les noeuds étiquetés \$:

```
let rec longueur_maximale = function
  | V -> -1
  | N (c, g, d) -> max (1 + longueur_maximale g) (longueur_maximale d)
```

3. On procède comme pour `afficher_mots`, en ajoutant un paramètre donnant la longueur actuelle du préfixe. On pourrait se contenter de la calculer avant de décider si on affiche le mot sans que cela n'impacte réellement la complexité.

```
let afficher_mots_longs dict n =
  let rec aux dict prefixe i =
    match dict with
    | V -> ()
    | N ('$', V, d) ->
      if i >= n then afficher (List.rev prefixe);
      aux d prefixe i
    | N (c, g, d) ->
      aux g (c :: prefixe) (i + 1);
      aux d prefixe i in
  aux dict [] 0
```

Correction de l'exercice XXIII.7 page 558

Le plus simple est sans doute d'utiliser une boucle `while true` dont on sortira *via* une exception :

```
let lire_fichier f =
  let ic = open_in f in
  let dict = ref V in
  try
    while true do
      let s = input_line ic in
      dict := ajouter !dict (mot_of_string s)
    done;
    V
  with
  | End_of_file -> !dict
```

Correction de l'exercice XXIII.8 page 559

1. Aucune difficulté, on utilise la fonction `String.iter` (similaire à `Array.iter` et `List.iter`) mais l'on pourrait bien sûr faire une boucle `for` :

```
let calculer_occurrences s =
  let occs = Array.make 256 0 in
  for i = 0 to String.length s - 1 do
    let x = int_of_char c in
    occs.(x) <- occs.(x) + 1
  done;
  occs
```

2. La version la plus simple à comprendre est sans doute celle-ci :

```

let afficher_mots_contenus dict s =
  let rec aux dict prefixe occs =
    match dict with
    | V -> ()
    | N ('$', V, d) ->
      afficher (List.rev prefixe);
      aux d prefixe occs
    | N (c, g, d) ->
      aux d prefixe occs;
      let i = int_of_char c in
      if occs.(i) > 0 then begin
        let nv_occs = Array.copy occs in
        nv_occs.(i) <- nv_occs.(i) - 1;
        aux g (c :: prefixe) nv_occs
      end in
      aux dict [] (calculer_occurrences s)

```

Il faut bien faire une copie de `occs` et modifier cette copie (et non l'original) : sinon, lorsqu'on remonte dans l'arbre après avoir terminé l'exploration d'une branche, les lettres « consommées » ne sont pas rendues.

On peut cependant faire plus efficace en utilisant un seul tableau `occs` pour tous les appels récursifs. Il faut alors penser à le « remettre en état » quand l'appel sur la branche de gauche se termine :

```

let afficher_mots_contenus_bis dict s =
  let occs = calculer_occurrences s in
  let rec aux dict prefixe =
    match dict with
    | V -> ()
    | N ('$', V, d) ->
      afficher (List.rev prefixe);
      aux d prefixe
    | N (c, g, d) ->
      aux d prefixe;
      let i = int_of_char c in
      if occs.(i) > 0 then begin
        occs.(i) <- occs.(i) - 1;
        aux g (c :: prefixe);
        occs.(i) <- occs.(i) + 1
      end in
      aux dict []

```

Quasiment toutes les fonctions qu'il nous reste à écrire sont des variations sur cette idée : dans le corrigé, on utilisera la deuxième variante, un peu plus concise et plus efficace, mais la première variante conviendrait aussi.

3. La fonction est extrêmement similaire, seul le test pour afficher diffère :

```

let est_nul t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n && t.(!i) = 0 do
    incr i
  done;
  !i = n

```

```

let afficher_anagrammes dict s =
  let occs = calculer_occurrences s in
  let rec aux dict prefixe =
    match dict with
    | V -> ()
    | N ('$', V, d) ->
      if est_nul occs then afficher (List.rev prefixe);
      aux d prefixe
    | N (c, g, d) ->
      aux d prefixe;
      let i = int_of_char c in
      if occs.(i) > 0 then begin
        occs.(i) <- occs.(i) - 1;
        aux g (c :: prefixe);
        occs.(i) <- occs.(i) + 1
      end in
      aux dict []

```

Correction de l'exercice XXIII.9 page 559

1. Ce n'est pas très différent de l'affichage. Attention cependant à renvoyer un arbre bien formé : c'est le rôle du `if g' = V then d' else N (c, g', d')`.

```

let rec filtrer_contenus_occs dict occs =
  match dict with
  | V -> V
  | N ('$', V, d) ->
    N ('$', V, filtrer_contenus_occs d occs)
  | N (c, g, d) when occs.(int_of_char c) = 0 ->
    filtrer_contenus_occs d occs
  | N (c, g, d) ->
    let i = int_of_char c in
    let d' = filtrer_contenus_occs d occs in
    occs.(i) <- occs.(i) - 1;
    let g' = filtrer_contenus_occs g occs in
    occs.(i) <- occs.(i) + 1;
    if g' = V then d'
    else N (c, g', d')

let filtrer_mots_contenus dict s =
  let occs = calculer_occurrences s in
  filtrer_contenus_occs dict occs

```

2. Très similaire :

```

let est_negatif t =
  let n = Array.length t in
  let rec loop i =
    i = n || t.(i) <= 0 && loop (i + 1) in
  loop 0

```

```

let filtrer_contenant_occs dict s =
  let occs = calculer_occurrences s in
  let rec aux = function
    | V -> V
    | N (c, g, d) when est_natif occs -> N (c, g, d)
    | N (c, g, d) ->
        let i = int_of_char c in
        let d' = aux d in
        occs.(i) <- occs.(i) - 1;
        let g' = aux g in
        occs.(i) <- occs.(i) + 1;
        if g' = V then d'
        else N (c, g', d') in
  aux dict

```

3. Toujours le même principe :

```

let filtrer_anagrammes_occs dict occs =
  let rec aux = function
    | V -> V
    | N ('$', V, d) ->
        if est_nul occs then N ('$', V, V)
        else aux d
    | N (c, g, d) when occs.(int_of_char c) = 0 -> aux d
    | N (c, g, d) ->
        let i = int_of_char c in
        let d' = aux d in
        occs.(i) <- occs.(i) - 1;
        let g' = aux g in
        occs.(i) <- occs.(i) + 1;
        if g' = V then d' else N (c, g', d') in
  aux dict

let filtrer_anagrammes dict s =
  let occs = calculer_occurrences s in
  filtrer_anagrammes_occs dict occs

```

4. On écrit une fonction pour afficher les mots les plus courts d'un dictionnaire :

```

let afficher_mots_les_plus_courts dict =
  let rec aux prefixe d =
    match d with
    | V -> [], max_int
    | N ('$', V, d) -> [List.rev prefixe], 0
    | N (c, g, d) ->
        let mots_g, l_g = aux (c :: prefixe) g in
        let mots_d, l_d = aux prefixe d in
        if l_g >= l_d then mots_d, l_d
        else if l_g = l_d - 1 then mots_g @ mots_d, l_d
        else mots_g, l_g + 1 in
  let mots, _ = aux [] dict in
  List.iter afficher mots

```

On peut ensuite l'appliquer au dictionnaire constitué des mots contenant toutes les voyelles, et obtenir : *boyaudier, guerroyai, noyautiez, paumoyiez, rougeoyai*.

Correction de l'exercice XXIII.10 page 559

1. La différence fondamentale, c'est qu'il y a deux cas quand on arrive sur un nœud $N('$', V, d)$:

- soit on a utilisé toutes les lettres, et dans ce cas on a fini de décomposer;
- soit il nous reste des lettres, et dans ce cas il faut ajouter au moins un mot à la décomposition, en repartant au sommet de l'arbre sans changer le tableau occs.

```

let afficher_decompositions dict mot =
  let occs = calculer_occurrences mot in
  let rec aux prefixe = function
    | V -> ()
    | N('$', V, d) ->
      if est_nul occs then afficher (List.rev prefixe)
      else begin
        aux (' ' :: prefixe) dict;
        aux prefixe d
      end
    | N(c, g, d) ->
      let i = int_of_char c in
      if occs.(i) > 0 then begin
        occs.(i) <- occs.(i) - 1;
        aux (c :: prefixe) g;
        occs.(i) <- occs.(i) + 1
      end;
      aux prefixe d in
  aux [] dict

```

2. Pas de problème majeur si l'on a compris la question précédente. On choisit de restreindre le dictionnaire aux mots que l'on peut former à partir de `mot`, ce qui rend la fonction légèrement plus efficace. Notons quand même que, pour que l'arbre renvoyé soit bien formé, la distinction de cas ligne 10 est nécessaire.

```

1 let decompose_anagrammes dict mot =
2   let occs = calculer_occurrences mot in
3   let initial = filtrer_mots_contenus dict mot in
4   let rec aux = function
5     | V -> V
6     | N('$', V, d) ->
9       if est_nul occs then N('$', V, V)
8       else
9         let g' = aux initial in
10        if g' <> V then N(' ', g', aux d) else aux d
11      | N(c, _, d) when occs.(int_of_char c) = 0 -> aux d
12      | N(c, g, d) ->
13        let i = int_of_char c in
14        let d' = aux d in
15        occs.(i) <- occs.(i) - 1;
16        let g' = aux g in
17        occs.(i) <- occs.(i) + 1;
18        if g' = V then d' else N(c, g', d') in
19  aux initial

```

Correction de l'exercice XXIII.II page 560

Une solution possible est de se souvenir du mot précédent de la décomposition, et d'imposer que le mot actuel vienne après ce mot précédent dans l'ordre lexicographique, ce qui n'est pas complètement évident à faire sans rendre le code excessivement lourd. Ici :

- si ce qui reste du mot précédent est vide, c'est bon ;
- si la prochaine lettre du mot précédent est strictement inférieure à la lettre que l'on choisit (en suivant une branche gauche), alors on peut choisir n'importe quoi ensuite (ce qu'on traduit en remplaçant le mot précédent par la liste vide) ;
- si elle est égale à la lettre choisie, on passe à la lettre suivante du mot précédent ;
- si elle est strictement supérieure, on ne peut pas prendre cette branche.

```

let supprimer_tete = function
| [] -> []
| _ :: xs -> xs

let comparer_tete (x : char) v =
  match v with
  | y :: _ ->
    if x < y then -1
    else if x = y then 0
    else 1
  | [] -> 1

let decompose_anagrammes_unique dict mot =
  let occs = calculer_occurrences mot in
  let initial = filtrer_contenus_occs dict occs in
  let rec aux precedent actuel dict =
    match dict with
    | V -> V
    | N ('$', V, d) when precedent = [] ->
      if est_nul occs then N ('$', V, V)
      else
        let g' = aux (List.rev actuel) [] initial in
        if g' <> V then N (' ', g', aux precedent actuel d)
        else aux precedent actuel d
    | N (c, _, d) when occs.(int_of_char c) = 0 -> aux precedent actuel d
    | N (c, g, d) ->
      let i = int_of_char c in
      let d' = aux precedent actuel d in
      let comparaison = comparer_tete c precedent in
      if comparaison = -1 then d'
      else begin
        occs.(i) <- occs.(i) - 1;
        let precedent' =
          if comparaison = 0 then supprimer_tete precedent
          else [] in
        let g' = aux precedent' (c :: actuel) g in
        occs.(i) <- occs.(i) + 1;
        if g' = V then d' else N (c, g', d')
      end in
    aux [] [] initial
  
```

LISTES DOUBLEMENT CHAÎNÉES

Une *liste doublement chaînée* est une liste dans laquelle chaque maillon contient un lien vers le maillon suivant et un lien vers le maillon précédent. C'est une structure fondamentalement impérative, et assez pratique, en particulier parce qu'elle permet de supprimer un élément arbitraire en temps constant.

I Première version

On peut utiliser la structure suivante :

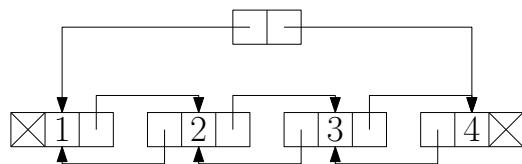


FIGURE XXIV.1 – Liste doublement chaînée (1, 2, 3, 4), première version.

Cette structure correspond aux définitions ci-dessous :

```

struct Node {
    int value;
    struct Node *prev;
    struct Node *next;
};

typedef struct Node node;

struct Dll_1 {
    node *start;
    node *end;
};

typedef struct Dll_1 dll_1;
  
```

On a comme invariants (si u est de type `dll_1*`)

- la liste est vide si et seulement si `u->start == NULL` si et seulement si `u->end == NULL`;
- si la liste n'est pas vide, alors `u->start->prev == NULL`, `u->end->next == NULL`, et ce sont les deux seuls pointeurs nuls présents dans la chaîne.

On donne les fonctions permettant de créer un nouveau noeud (avec une valeur donnée et des pointeurs nuls des deux côtés) et une nouvelle liste (vide) :

```

node *new_node(int x){
    node *n = malloc(sizeof(node));
    n->value = x;
    n->next = NULL;
    n->prev = NULL;
    return n;
}
  
```

```

dll_1 *new_dll_1(void){
    dll_1 *d = malloc(sizeof(dll_1));
    d->start = NULL;
    d->end = NULL;
    return d;
}
  
```

Exercice XXIV.1

p. 572

1. Écrire une fonction `delete_node` qui supprime un nœud d'une liste. Cette fonction prend un pointeur vers le nœud et un pointeur vers la liste en arguments, et elle gère à la fois la suppression du nœud et la libération de la mémoire correspondante.
2. Écrire une fonction `insert_before` qui insère un nœud (avec la valeur fournie) juste avant le nœud passé en argument.
3. Écrire la fonction symétrique `insert_after`.
4. Ces fonctions sont-elles suffisantes pour écrire une fonction `from_array` (par exemple) ? Si non, pourquoi ?

```
void delete_node(dll_1 *d, node *n);
void insert_before(dll_1 *d, node *n, int x);
void insert_after(dll_1 *d, node *n, int x);
```

2 Version avec sentinelle

La structure proposée n'est pas satisfaisante (même si on pourrait la faire marcher) : on peut en fait simplifier le code et enlever presque tous les cas particuliers en la modifiant légèrement. On ne change rien au type `node`, mais on convient de rajouter un nœud « fictif », appelé *sentinelle*, à l'extrême de la liste. La valeur présente dans le champ `value` de ce nœud ne sera pas significative, et la liste aura la structure suivante :

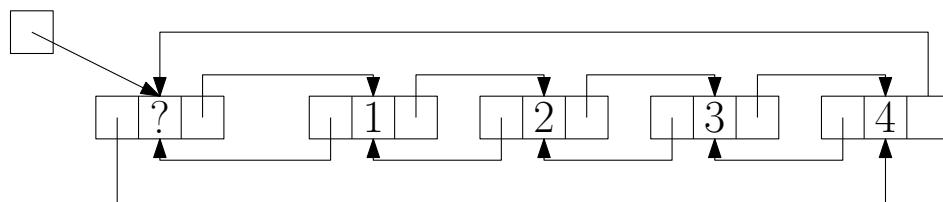


FIGURE XXIV.2 – Liste doublement chaînée (1, 2, 3, 4), version avec sentinelle.

```
struct DLL {
    struct Node *sentinel;
};

typedef struct DLL dll;
```

Exercice XXIV.2

p. 573

1. Écrire la fonction `new_dll` qui crée une nouvelle liste doublement chaînée. Cette liste sera vide, ce qui signifie qu'elle ne contiendra que le nœud sentinelle, correctement initialisé.
2. Ré-écrire les fonctions `delete_node`, `insert_before` et `insert_after` pour la nouvelle structure. La fonction `delete_node` pourra supposer sans le vérifier que le nœud passé en argument n'est pas le nœud sentinelle (et aucune de ces fonctions n'aura besoin de prendre la liste elle-même en argument).

```
dll *new_dll(void);
void delete_node(node *n);
node *insert_before(node *n, int x);
node *insert_after(node *n, int x);
```

3. Écrire une fonction `free_dll` qui libère la totalité de la mémoire utilisée par une liste doublement chaînée.

```
void free_dll(dll *d);
```

4. Une liste doublement chaînée permet de réaliser facilement la structure abstraite de *deque* (*double ended queue*, ou *file bilatère*).

Écrire les quatre fonctions suivantes, dont la spécification devrait être facile à deviner. Pour `pop_left` et `pop_right`, on vérifiera la licéité de l'appel à l'aide d'un `assert`.

```
void push_left(dll *d, int x);
void push_right(dll *d, int x);

int pop_left(dll *d);
int pop_right(dll *d);
```

5. Écrire une fonction `from_array` qui convertit un tableau en liste doublement chaînée.

```
dll *from_array(int t[], int len);
```

3 Nombres chanceux

On considère le processus suivant :

- on part de la liste des entiers impairs (jusqu'à une certaine borne n);
- l'entier 1 est *chanceux*;
- on considère l'entier qui suit 1 dans la liste (c'est 3);
- on élimine un nombre sur 3 de la liste, en commençant au début;
- l'entier 3 est *chanceux*;
- on considère l'entier qui suit 3 dans la liste (c'est 7);
- on élimine un nombre sur 7 de la liste, en commençant au début;
- l'entier 7 est *chanceux*...

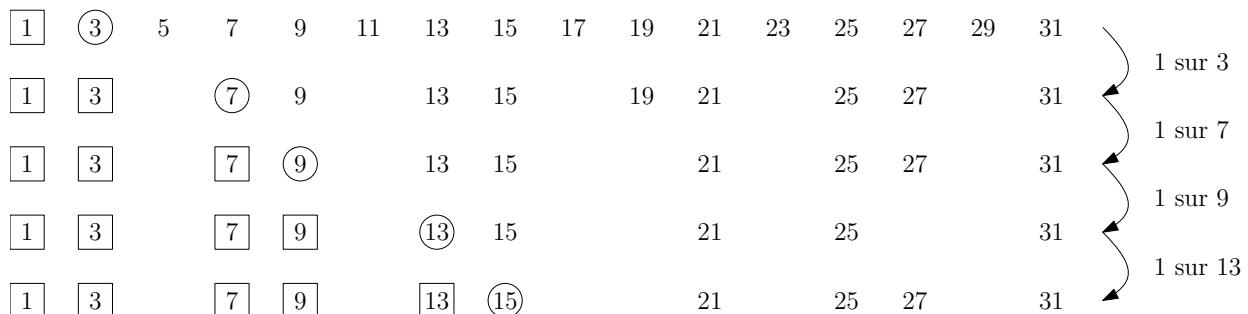


FIGURE XXIV.3 – Génération des nombres chanceux.

Exercice XXIV.3

Combien vaut la somme des nombres chanceux inférieurs ou égaux à 10^5 .

Remarque

L'idée est d'utiliser une liste doublement chaînée : je ne dis pas que c'est le plus simple ou le plus efficace, c'est juste un exercice...

Solutions

Correction de l'exercice XXIV.I page 570

```
void delete_node_1(dll_1 *d, node *n){
    if (n->prev != NULL){
        n->prev->next = n->next;
    } else {
        d->start = n->next;
    }
    if (n->next != NULL){
        n->next->prev = n->prev;
    } else {
        d->end = n->prev;
    }
    free(n);
}

void insert_before_1(dll_1 *d, node *n, int x){
    node *new = new_node(x);
    new->prev = n->prev;
    new->next = n;
    n->prev = new;
    if (new->prev != NULL){
        new->prev->next = new;
    } else {
        d->start = new;
    }
}

void insert_after_1(dll_1 *d, node *n, int x){
    node *new = new_node(x);
    new->next = n->next;
    new->prev = n;
    n->next = new;
    if (new->next != NULL){
        new->next->prev = new;
    } else {
        d->end = new;
    }
}
```

Telles que nous les avons écrites, ces fonctions ne permettent pas de créer une liste non vide à partir d'une liste vide. En effet, `insert_before_1` et `insert_after_1` n'acceptent pas que le noeud `n` passé en argument soit `NULL`.

Correction de l'exercice XXIV.2 page 570

1. Le nœud sentinelle pointe vers lui-même :

```
dll *new_dll(void){  
    dll *d = malloc(sizeof(dll));  
    node *sentinel = new_node(0);  
    sentinel->prev = sentinel;  
    sentinel->next = sentinel;  
    d->sentinel = sentinel;  
    return d;  
}
```

2. C'est nettement plus simple, il n'y a plus aucun cas particulier :

```
void delete_node(node *c){  
    assert(c != NULL);  
    c->prev->next = c->next;  
    c->next->prev = c->prev;  
    free(c);  
}  
  
node *insert_before(node *c, int x){  
    node *n = new_node(x);  
    n->next = c;  
    n->prev = c->prev;  
    c->prev = n;  
    n->prev->next = n;  
    return n;  
}  
  
node *insert_after(node *c, int x){  
    node *n = new_node(x);  
    n->prev = c;  
    n->next = c->next;  
    c->next = n;  
    n->next->prev = n;  
    return n;  
}
```

3. Il sera pratique pour la suite de disposer de deux fonctions renvoyant respectivement le premier et le dernier nœud de la liste.

```
node *start(dll *d){
    return d->sentinel->next;
}

node *end(dll *d){
    return d->sentinel->prev;
}

void free_dll(dll *d){
    node *n = start(d);
    while (n != d->sentinel){
        node *tmp = n->next;
        free(n);
        n = tmp;
    }
    free(d->sentinel);
    free(d);
}
```

4. C'est immédiat avec ce que l'on a déjà écrit :

```
void push_left(dll *d, int x){
    insert_after(d->sentinel, x);
}

void push_right(dll *d, int x){
    insert_before(d->sentinel, x);
}

int pop_left(dll *d){
    node *first = start(d);
    assert(first != d->sentinel);
    int value = first->value;
    delete_node(first);
    return value;
}

int pop_right(dll *d){
    node *last = end(d);
    assert(last != d->sentinel);
    int value = last->value;
    delete_node(last);
    return value;
}
```

5. À nouveau, on dispose de tout le nécessaire :

```
dll *from_array(int t[], int len){
    dll *d = new_dll();
    for (int i = 0; i < len; i++){
        push_right(d, t[i]);
    }
    return d;
}
```

LISTES À ACCÈS DIRECT

La première partie sert d'inspiration pour la deuxième, même si elles sont techniquement indépendantes. La troisième a un rôle similaire par rapport à la quatrième.

En informatique, on oppose le stockage à accès direct (appelé *random access* en anglais) au stockage séquentiel. Le premier permet d'accéder directement à n'importe quelle donnée (c'est le cas de la mémoire vive par exemple, appelée aussi RAM pour *random access memory*) alors que le deuxième impose de lire les données dans un ordre fixé au départ (c'est le cas par exemple des bandes magnétiques, très utilisées encore aujourd'hui pour l'archivage de données).

Dans le domaine des structures de données, une distinction similaire peut être faite entre les structures de type tableau qui permettent d'accéder à un élément quelconque en temps constant et les structures de type liste qui peuvent demander un temps linéaire en leur taille pour accéder à un élément.

On s'intéresse dans ce problème à des structures hybrides, dites *listes à accès direct* (ou en anglais *random access lists*) permettant un accès rapide à un élément quelconque tout en gardant les avantages des listes (en particulier, ajout rapide d'un élément en tête). Ces structures seront *purement fonctionnelles* (aucune modification en place). Pour éviter les confusions, on utilisera « liste » pour désigner les structures que nous définirons et **list** quand on voudra parler d'une liste OCaml usuelle.

Plus précisément, on veut définir un type de données 'a t doté de la signature suivante, dans laquelle toutes les fonctions doivent être au pire en temps logarithmique en le nombre d'éléments :

```
cons : ('a t) -> ('a t) -> 'a t
(* renvoie la liste obtenue en rajoutant x en tête de u *)

head : 'a t -> 'a
(* renvoie l'élément de tête d'une liste (ou une exception si la liste est vide) *)

tail : 'a t -> 'a t
(* renvoie la liste obtenue en éliminant l'élément de tête de l'argument *)

get : ('a t) -> (int) -> 'a
(* renvoie l'élément d'indice i de u (les indices commencent à 0) *)

set : ('a t) -> (int) -> ('a t) -> 'a t
(* renvoie la liste obtenue en remplaçant l'élément d'indice i de u par x *)
```

► **Question 1** Quelle complexité obtiendrait-on pour ces différentes fonctions si l'on utilisait pour t le type **list** de OCaml ? On ne demande pas d'écrire les fonctions, ni de justifier la réponse.

I Nombres en binaire

On rappelle que la représentation binaire d'un entier $n \geq 1$ est l'unique liste a_0, \dots, a_p d'éléments de $\{0, 1\}$ telle que $a_p \neq 0$ et $n = \sum_{i=0}^p a_i 2^i$. Le nombre 0 a lui une représentation vide.

On dira que a_i est le chiffre de *rang* i de n et que son poids est 2^i .

On notera $n = \overline{a_0 \dots a_p}^2$ (chiffre le moins significatif en premier, attention).

Un nombre en binaire se représente de manière naturelle en OCaml par la liste de ses chiffres (moins significatif en premier) :

```
type bit = Z | U
type nombre = bit list
```

On veillera à toujours garder des représentations canoniques : 0 est représenté par la liste vide, toute liste non vide de type `nombre` doit se terminer par un `U`. On pourra supposer que les arguments passés aux fonctions vérifient ces contraintes, il faudra faire en sorte que les résultats en fassent de même.

Les fonctions demandées dans cette partie travaillent uniquement sur les représentations en binaire : à aucun moment on n'essaiera de convertir les `nombre` en des `int`.

► **Question 2** Écrire une fonction `succ` : `nombre` -> `nombre` prenant la représentation d'un entier `n` et renvoyant la représentation de `n + 1` (le *successeur* de `n`).

Déterminer sa complexité en fonction de la longueur de son argument ; on caractérisera le pire des cas.

```
# succ [U; Z; U; U];;
- : nombre = [Z; U; U; U]
```

► **Question 3** Écrire une fonction `pred` : `nombre` -> `nombre` prenant la représentation d'un entier `n` et renvoyant la représentation de `n - 1` (le *prédecesseur* de `n`). On renverra une exception *via failwith* si l'argument représente 0.

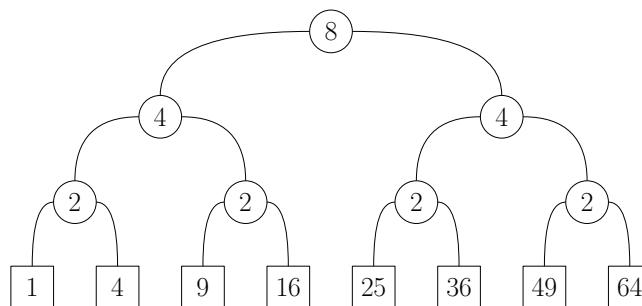
Déterminer sa complexité en fonction de la longueur de son argument ; on caractérisera le pire des cas.

2 Listes binaires à accès direct

On commence par considérer des arbres binaires dont les feuilles sont étiquetées par une valeur de type `'a` et dont les noeuds internes contiennent un champ entier dans lequel on stockera le nombre de feuilles du sous-arbre correspondant (que l'on appellera simplement *taille*).

```
type 'a arbre = F of 'a | N of int * 'a arbre * 'a arbre
```

Les arbres que nous considérerons auront toutes leurs feuilles à la même profondeur : autrement dit, il s'agira d'arbres binaires parfaits, ayant 2^i feuilles si leur hauteur vaut `i`. À l'intérieur d'un arbre, on considère que les feuilles sont ordonnées de gauche à droite. Ainsi, la liste `(1, 4, 9, 16, 25, 36, 49, 64)` peut être représentée par l'arbre suivant (de hauteur 3 et de taille $2^3 = 8$) :



Remarque

Pour les fonctions `get` et `set` demandées ci-dessous, on demande des implémentations efficaces, c'est-à-dire tirant partie des informations portées par les noeuds internes pour éviter de parcourir tout l'arbre.

► **Question 4** Écrire une fonction `get_arbre` : `(u : 'a arbre) -> (i : int) -> 'a` qui renvoie l'élément d'indice `i` de l'arbre `u`.

Par exemple, si `u` est l'arbre dessiné ci-dessus, `get_arbre u 2` doit renvoyer `9` et `get_arbre u 7` renvoyer `64`.

On renverra une exception `failure "dépassement"` (par `failwith "dépassement"`) si i n'est pas un indice valide pour u .

► **Question 5** Déterminer la complexité de `get_arbre` en fonction de la taille n de u . *On rappelle que u est supposé parfait.*

► **Question 6** Écrire une fonction `set_arbre : (u : 'a arbre) -> (i : int) -> (x : 'a) -> 'a arbre` qui renvoie l'arbre obtenu en remplaçant l'élément d'indice i de u par x . Donner sa complexité en temps.

► **Question 7** Si u est de taille n et si v est le résultat du calcul de `set_arbre u k x` pour un certain k et un certain x , quelle quantité de mémoire supplémentaire faut-il pour stocker v tout en conservant u ? On donnera la réponse en considérant qu'un nœud interne ou une feuille occupe une quantité constante C de mémoire, et l'on fera un schéma.

Une *liste binaire à accès direct* (on écrira simplement *liste binaire*) de longueur n sera constituée d'un arbre du type que nous venons de définir pour chaque chiffre 1 dans la représentation binaire de n . Si le chiffre 1 considéré a un poids de 2^i (*i.e.* s'il s'agit du chiffre de rang i), l'arbre correspondant aura une taille de 2^i . On a donc les types suivants :

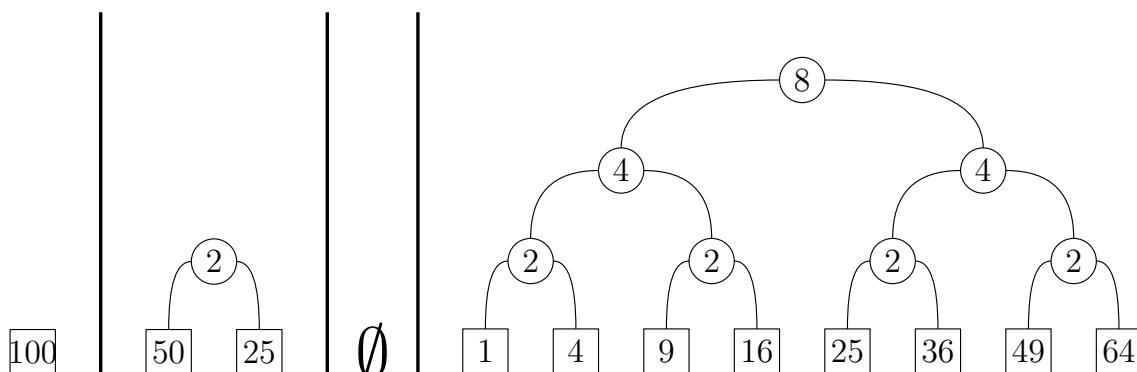
```
type 'a chiffre = Ze | Un of 'a arbre
type 'a liste_binaire = 'a chiffre list
```

À l'intérieur d'une `'a chiffre list`, les arbres seront classés par taille croissante, et l'ordre induit sur les feuilles sera de gauche à droite dans la liste, puis de gauche à droite à l'intérieur de chaque arbre.

Si u est une liste binaire :

- on appellera *taille* de u , et l'on notera $|u|$, la somme des tailles des arbres qui la composent, c'est-à-dire le nombre total de feuilles qu'elle contient;
- on appellera *longueur* de u , et l'on notera $\text{longueur}(u)$, le nombre de chiffres qu'elle contient, c'est-à-dire $\text{longueur}(u) = \text{List}.length u$.

Par exemple, la liste $(100, 50, 25, 1, 4, 9, 16, 25, 36, 49, 64)$, de longueur $11 = 2^0 + 2^1 + 2^3$, sera représentée par la liste binaire suivante (de longueur 4 et de taille 11) :



Cette liste pourrait être définie en OCaml par :

```
(* Les trois arbres : *)
let a = F 100
let b = N(2, F 50, F 25)
let c = N(8,
          N(4, N(2, F 1, F 4), N(2, F 9, F 16)),
          N(4, N(2, F 25, F 36), N(2, F 49, F 64)))
(* et la liste : *)
let li = [Un a; Un b; Ze; Un c]
```

Par commodité, on définit une fonction donnant la taille d'un arbre :

```
let size = function
| F _ -> 1
| N (n, _, _) -> n
```

► **Question 8** Soit u une liste binaire non vide, $n > 0$ sa taille et k sa longueur. Exprimer k en fonction de n (on justifiera).

► **Question 9** Écrire une fonction $\text{get} : (\text{li} : \text{'a liste_binaire}) \rightarrow (i : \text{int}) \rightarrow \text{'a}$ renvoyant l'élément d'indice i de li . On utilisera la fonction get_arbre écrite plus haut.
Avec le li défini plus haut, on doit avoir $\text{get li } 3 = 1$.

► **Question 10** Donner en fonction de $|u|$ la complexité dans le pire des cas de get u k . Peut-on garantir l'exécution en temps constant si $k = 0$?

► **Question 11**

Écrire une fonction $\text{set} : (\text{li} : \text{'a liste_binaire}) \rightarrow (i : \text{int}) \rightarrow (x : \text{'a}) \rightarrow \text{'a liste_binaire}$ renvoyant la liste binaire obtenue en remplaçant l'élément d'indice i de li par x . Donner la complexité de cette fonction.

► **Question 12** Écrire une fonction $\text{cons} : (\text{li} : \text{'a liste_binaire}) \rightarrow (x : \text{'a}) \rightarrow \text{'a liste_binaire}$ renvoyant la liste binaire obtenue en rajoutant x en tête de li .

On pourra s'aider d'une fonction $\text{cons_arbre} : \text{'a liste_binaire} \rightarrow \text{'a arbre} \rightarrow \text{'a liste_binaire}$ inspirée du succ de la partie précédente.

Avec le li précédent, on doit avoir

```
# cons li 12;;
- : int chiffre list =
[Ze;
 Ze;
 Un (N (4, N (2, F 12, F 100), N (2, F 50, F 25)));
 Un
 (N (8, N (4, N (2, F 1, F 4), N (2, F 9, F 16)),
 N (4, N (2, F 25, F 36), N (2, F 49, F 64))))]
```

► **Question 13** Déterminer la complexité de la fonction cons dans le pire des cas et dans le meilleur des cas.

► **Question 14** Écrire une fonction $\text{uncons} : (\text{li} : \text{'a liste_binaire}) \rightarrow \text{'a * 'a liste_binaire}$ qui renvoie l'élément en tête de li ainsi que la liste binaire obtenue en retirant cet élément de li .
Donner la complexité de cette fonction dans le pire et dans le meilleur des cas.

► **Question 15** En déduire les fonctions head et tail spécifiées au début du problème, ainsi que leur complexité.

3 Nombres en binaire décentré

Dans le système en binaire décentré (*skewed binary* en anglais), les chiffres peuvent prendre les valeurs 0, 1 et 2, et le chiffre de rang i a pour poids $r_i = 2^{i+1} - 1$. Autrement dit, une représentation décentrée est une liste (a_0, \dots, a_p) d'éléments de $\{0, 1, 2\}$ et le nombre correspondant est $n = \sum_{i=0}^p a_i r_i$. On pourra alors noter $n = \overline{a_0 \dots a_p}^d$.

Ainsi, $21 = \overline{0201}^d = \overline{122}^d$. Comme on peut le voir, la représentation décentrée n'est pas unique. Cependant, on peut définir une représentation canonique : une représentation décentrée est dite canonique si elle ne contient pas de 2 ou si son seul chiffre égal à 2 est le chiffre non nul de rang le plus petit. La

représentation canonique de 21 est alors $\overline{0}2\overline{0}1$ (bien sûr, on impose toujours qu'il n'y ait pas de zéros à la fin de la représentation).

On notera l'identité suivante, utile dans plusieurs questions : $r_{i+1} = 2r_i + 1$.

► **Question 16** Cette question, un peu délicate, n'a pas vraiment de rapport avec le reste du sujet. Je vous encourage plutôt à la chercher chez vous.

Montrer que tout entier naturel a une unique représentation décentrée canonique.

Dans la suite du problème, on supposera toujours que les représentations utilisées sont canoniques (et il faudra faire en sorte que les représentations créées par les fonctions que l'on demande d'écrire le soient aussi).

L'intérêt principal de la représentation décentrée est que les opérations succ et prec peuvent se faire sans propagation de retenue.

► **Question 17** Donner la représentation canonique des entiers de 6 à 15.

► **Question 18** Expliquer comment calculer le successeur et le prédécesseur d'un nombre binaire décentré en ne changeant la valeur que de deux chiffres au maximum.

► **Question 19** On suppose à cette question (uniquement) que l'on représente les nombres en binaire décentré à l'aide du type suivant, en stockant le chiffre le moins significatif en tête de liste :

```
type chiffre_decentre = ZZ | UU | DD
type nombre_decentre = chiffre_decentre list
```

Expliquer pourquoi la méthode vue à la question précédente ne permet pas d'écrire des fonctions succ et pred en temps constant.

Pour éviter le problème soulevé à la question précédente, nous allons utiliser une représentation creuse pour les nombres en binaire décentré : au lieu de stocker tous les chiffres, nous allons stocker la liste des poids des chiffres non nuls. Cette liste sera en ordre strictement croissant, sauf que le premier élément pourra être présent deux fois, ce qui sera notre manière de marquer la présence d'un 2.

On définit donc le type decentre des représentations décentrées par :

```
type decentre = int list
```

Ainsi, 21 sera représenté par la liste [3; 3; 15] alors que 41 sera représenté par [3; 7; 31].

► **Question 20** Écrire une fonction succ_d : decentre -> decentre calculant le successeur d'un entier décentré en temps constant.

► **Question 21** Écrire une fonction pred_d : decentre -> decentre calculant le prédécesseur d'un entier décentré en temps constant. On renverra une exception si l'argument représente 0.

Listes en binaire décentré

Nous avons vu plus haut que la représentation des séquences par des listes binaires ne permet pas de conserver des opérations cons, head et tail en temps constant comme pour les listes usuelles.

Pour obtenir ces performances, on modifie la structure de liste binaire en utilisant la représentation décentrée creuse. De plus, on stocke à présent les éléments de la séquence dans les nœuds internes aussi bien que dans les feuilles : par conséquent, la taille d'un arbre est désormais le nombre total de nœuds et non plus seulement le nombre de feuilles. On ne stocke plus cette taille à chaque nœud, mais seulement la taille de chacun des arbres constituant la liste en binaire décentré. On obtient donc le type suivant :

```
type 'a arbre_decentre = Fd of 'a | Nd of 'a * 'a arbre_decentre * 'a arbre_decentre
type 'a liste_decentre = (int * 'a arbre_decentre) list
```

Une liste en binaire décentré contenant 21 éléments de type 'a sera donc de la forme

```
[ (3, u); (3, v); (15, w) ]
```

où u , v et w sont des `'a arbre_decentre` de tailles respectives 3, 3 et 15.

► **Question 22** Pour avoir une chance d'implémenter `head` en temps constant, dans quel ordre faut-il stocker les étiquettes à l'intérieur d'un arbre? *On admettra que les seuls candidats raisonnables à considérer sont les ordres préfixe, infixe et suffixe.*

Donner explicitement (en dessinant les arbres) la liste en binaire décentré correspondant à la séquence $(1, 2, \dots, 13)$.

► **Question 23** Écrire la fonction `get_d : 'a liste_decentre -> int` -> `'a` et préciser (sans justification) sa complexité.

► **Question 24** Écrire la fonction `cons_d : 'a liste_decentre -> 'a -> 'a liste_decentre`. On exige que cette fonction s'exécute en temps constant.

► **Question 25** Écrire la fonction `head_d : 'a liste_decentre -> 'a`. On exige que cette fonction s'exécute en temps constant.

► **Question 26** Écrire la fonction `tail_d : 'a liste_decentre -> 'a liste_decentre`. On exige que cette fonction s'exécute en temps constant.

Ce problème est directement inspiré de Purely Functional Data Structures, Chris Okasaki, CAMBRIDGE UNIVERSITY PRESS. La thèse de Chris Okasaki (dont l'ouvrage cité est une version légèrement enrichie) est librement disponible en ligne ; elle porte le même titre et fournira une excellente lecture de vacances aux plus motivé·e·s d'entre vous.

Solutions

► **Question 1** On aurait les complexités temporelles suivantes :

- `cons`, `head` et `tail` en $\mathcal{O}(1)$;
- `get` et `set` en $\mathcal{O}(i)$ (et donc $\mathcal{O}(|u|)$).

Nombres en binaire

► **Question 2**

```
let rec succ : nombre -> nombre = function
| [] -> [U]
| Z :: xs -> U :: xs
| U :: xs -> Z :: succ xs
```

On parcourt toute la liste u jusqu'à trouver un Z (ou arriver à la fin), la complexité est donc en $\mathcal{O}(|u|)$ dans le pire des cas, qui correspond à une liste constituée uniquement de U .

► **Question 3**

```
let rec pred : nombre -> nombre = function
| [] -> failwith "zero moins un"
| [U] -> []
| Z :: xs -> U :: pred xs
| U :: xs -> Z :: xs
```

La situation est symétrique de celle pour `succ` : la complexité est en $\mathcal{O}(|u|)$ et le pire cas correspond à une liste constituée uniquement de Z (sauf un U en dernière position pour assurer le caractère canonique).

Listes binaires à accès direct

► **Question 4**

```
let rec get_arbre u n =
  match u with
  | F x -> if n = 0 then x else failwith "depassement"
  | N (p, ga, dr) ->
    if n < p / 2 then
      get_arbre ga n
    else
      get_arbre dr (n - p / 2)
```

► **Question 5** On descend dans l'arbre le long d'un chemin reliant la racine à une feuille. Les opérations effectuées à chaque noeud étant en temps constant, le temps de calcul est proportionnel à la longueur de ce chemin (*i.e.* à la profondeur de la feuille).

L'arbre étant parfait, ses 2^i feuilles sont situées à profondeur i : autrement dit, si n est la taille de l'arbre, alors la longueur du chemin est exactement $\log_2 n$. La complexité de `get_arbre` est donc en $\mathcal{O}(\log n)$.

► Question 6

```
let rec set_arbre u n x =
  match u with
    | F y -> if n = 0 then F x else failwith "dépassement"
    | N (p, ga, dr) ->
        if n < p / 2 then
          N (p, set_arbre ga n x, dr)
        else
          N (p, ga, set_arbre dr (n - p / 2) x)
```

On obtient de même une complexité en $\mathcal{O}(\log n)$.

► Question 7 On copie tous les nœuds situés sur le chemin reliant la racine à la feuille « modifiée », le reste de l’arbre est partagé entre les deux versions. Comme dit plus haut, il y a $\log_2 n$ nœuds sur ce chemin, et la quantité d’espace supplémentaire consommée par la coexistence des deux versions est donc de $C \log_2 n$.

► Question 8 La longueur k de la liste est par définition le nombre de chiffres de l’écriture binaire de n . Or n s’écrit avec k chiffres si et seulement si $2^{k-1} \leq n < 2^k$, c’est-à-dire $k-1 \leq \log_2 n < k$ et donc

$$k = 1 + \lfloor \log_2 n \rfloor.$$

► Question 9

```
let rec lookup liste n =
  match liste with
    | [] -> failwith "dépassement"
    | Ze :: xs -> lookup xs n
    | Un t :: xs -> if n < size t then (t, n)
      else lookup xs (n - size t)

let get liste n =
  let t, i = lookup liste n in
  get_arbre t i
```

► Question 10 Notons n la taille de u et k sa longueur. On commence par faire un appel à `lookup` qui parcourt u (en entier si la valeur cherchée est dans le dernier arbre). Cet appel prend donc au pire un temps proportionnel à k , c’est-à-dire à $\log n$ d’après la question 8.

On fait ensuite un appel à `get_arbre t`, où t est l’un des arbres de la liste. Cet appel prend un temps proportionnel à $\log |t|$ d’après 5, or comme $n = \sum_{t \in u} |t|$, on a $\log |t| \leq \log n$.

Finalement, la complexité dans le pire des cas de `get u k` est en $\mathcal{O}(\log |u|)$.

On ne peut pas garantir une complexité en $\mathcal{O}(1)$ si $k = 0$ car il faut parcourir la liste jusqu’à trouver un arbre : c’est immédiat si n est impair mais peut prendre un temps proportionnel à $\log n$ si n est une puissance de 2.

► Question 11

```
let rec set liste n x =
  match liste with
    | [] -> failwith "dépassement"
    | Ze :: ts -> Ze :: set ts n x
    | Un t :: ts when size t > n -> Un (set_arbre t n x) :: ts
    | Un t :: ts -> Un t :: set ts (n - size t) x
```

On a à nouveau une complexité en $\mathcal{O}(\log n)$.

► Question 12

- La fonction `link` fusionne deux arbres supposées de même taille 2^i pour en former un de taille 2^{i+1} (les feuilles héritées du premier arbre se retrouvent à gauche de celles héritées du deuxième).
- La fonction `cons_tree` prend une liste **qui commence au chiffre de poids 2^i** et un arbre de taille 2^i et le « rajoute à la liste » : si le chiffre de poids 2^i vaut zéro, on met l’arbre à cette place, sinon on le fusionne avec l’arbre présent, le chiffre passe à zéro et l’on « propage la retenue ».
- Pour ajouter un élément x à une liste, il suffit alors de créer une feuille $F\ x$ et d’appeler `cons_tree`.

```

let link u t =
  N (size u + size t, u, t)

let rec cons_tree u t =
  match u with
  | [] -> [Un t]
  | Ze :: xs -> Un t :: xs
  | Un t' :: xs -> Ze :: cons_tree xs (link t t')

let cons u x = cons_tree u (F x)

```

► Question 13 La complexité dépend uniquement du nombre d’appels récursifs à `cons_tree` puisque toutes les opérations (y compris `link`) se font en temps constant. Ce nombre d’appels est égal au nombre de `Un t` présents en début de liste (essentiellement, la question est de savoir combien de temps on doit propager la retenue) :

- dans le meilleur des cas (n est pair, le premier chiffre est nul), l’exécution sera **en temps constant** ;
- dans le pire des cas (n de la forme $2^k - 1$, tous les chiffres valent un), on aura un temps proportionnel à k , donc en **$\mathcal{O}(\log n)$** .

► Question 14

- La fonction `uncons_tree`, appelée sur une liste binaire dont le premier chiffre est de poids 2^i , renvoie l’arbre de taille 2^i contenant les 2^i premières feuilles de la liste binaire, ainsi que la liste binaire contenant les $n - 2^i$ dernières étiquettes. L’idée est similaire à celle de la fonction `pred`.
- La fonction `uncons` traite le cas particulier de la liste de taille 1 (nécessaire car une liste binaire de taille 0 s’écrit [] et non [Ze]), et appelle sinon `uncons_tree` pour récupérer la feuille contenant le premier élément et la « queue » de la liste.
- La complexité dépend uniquement du nombre d’appels récursifs à `uncons_tree`, qui est égal au nombre de zéros en tête de la liste.

Dans le meilleur des cas (n impair, le premier chiffre vaut 1), on aura donc une complexité **en $\mathcal{O}(1)$** , et dans le pire (n est une puissance de deux, tous les chiffres sauf le dernier valent zéro) une complexité **en $\mathcal{O}(\log n)$** .

```

let rec uncons_tree u =
  match u with
  | [] -> failwith "depassement"
  | Un t :: ts -> (t, Ze :: ts)
  | Ze :: ts ->
    let y, ys = uncons_tree ts in
    match y with
    | N (_, ga, dr) -> (ga, Un dr :: ys)
    | _ -> failwith "impossible"

let uncons u =
  match u with
  | [Un (F x)] -> x, []
  | _ ->
    match uncons_tree u with
    | F x, ts -> x, ts
    | _ -> failwith "impossible"

```

► Question 15

```

let head u = fst (uncons u)
let tail u = snd (uncons u)

```

Nombres en binaire décentré

► Question 16

Existence On montre par récurrence la propriété suivante : « si $0 \leq x < r_i = 2^{i+1} - 1$, alors x a une représentation canonique n'utilisant que des chiffres de rang strictement inférieur à i ».

- Pour $i = 0$, on a $x = 0$ (représentation vide).
- Soit $x < r_{i+1}$, on cherche une représentation canonique sans chiffre de rang supérieur ou égal à $i+1$. Si $x < r_i$, alors il a une représentation canonique (dont les chiffres sont de rang $< i$) par hypothèse de récurrence.
- Si $x = r_{i+1} - 1$, alors $x = 2r_i$ d'après la remarque de l'énoncé, ce qui fournit une représentation canonique convenable.

Sinon, on a $r_i \leq x < r_{i+1} - 1$. En posant $y = x - r_i$, on a $0 \leq y < r_{i+1} - 1 - r_i$, c'est-à-dire $0 \leq y < r_i$. On dispose donc d'une représentation canonique de y sans chiffre de rang $\geq i$, ce qui fournit une représentation canonique convenable de x en ajoutant un 1 comme chiffre de rang i .

Unicité Comptons les représentations canoniques n'utilisant que des chiffres de rang $0 \dots i-1$.

- Il y a la représentation vide (aucun chiffre non nul).
- Pour chaque k de $[0 \dots i-1]$, il y a $2 \times 2^{i-1-k}$ représentations dont le premier chiffre non nul est celui de rang k . En effet, $a_k \in \{1, 2\}$ (deux choix) et chacun des a_j avec $k < j < i$ est dans $\{0, 1\}$ (deux choix à chaque fois, donc 2^{i-k-1} au total).

On a donc $1 + \sum_{k=0}^{i-1} 2^{i-k} = 1 + \sum_{j=1}^i 2^j = 1 + 2^{i+1} - 2 = r_i$ représentations canoniques n'utilisant que des chiffres de rang strictement inférieurs à i . Or, d'après l'existence, l'ensemble des valeurs correspondantes contient les r_i entiers de $[0, r_i - 1]$. On en déduit l'unicité.

► Question 17 On a :

- | | | |
|---|--|---|
| <ul style="list-style-type: none"> ■ $6 = 3 + 3 = \overline{02}^d$ ■ $7 = 7 = \overline{001}^d$ ■ $8 = 1 + 7 = \overline{101}^d$ ■ $9 = 1 + 1 + 7 = \overline{201}^d$ | <ul style="list-style-type: none"> ■ $10 = 3 + 7 = \overline{011}^d$ ■ $11 = 1 + 3 + 7 = \overline{111}^d$ ■ $12 = 1 + 1 + 3 + 7 = \overline{211}^d$ ■ $13 = 3 + 3 + 7 = \overline{021}^d$ | <ul style="list-style-type: none"> ■ $14 = 7 + 7 = \overline{002}^d$ ■ $15 = 1 + 7 + 7 = \overline{0001}^d$ |
|---|--|---|

► Question 18

Zéro n'a pas de prédécesseur et son successeur est 1, il n'y a pas de problème. Soit $x \in \mathbb{N}^*$, et $\overline{a_0 \dots a_k}^d$ sa représentation canonique. Soit également a_j son premier chiffre non nul.

Successeur

- Si $a_j \neq 2$, alors il n'y a aucun 2 dans la représentation (car elle est canonique), et $x + 1 = x + r_0 = \overline{b_0 \dots b_k}$ où $b_0 = a_0 + 1$ et $b_i = a_i$ si $i \geq 1$. La représentation donnée pour $x + 1$ est bien canonique, b_0 étant le seul chiffre potentiellement égal à 2.
- Si $a_j = 2$, alors c'est le seul 2 dans la représentation. Notons y le nombre obtenu à partir de x en remplaçant a_j par $b_j = 0$ et a_{j+1} par $b_{j+1} = 1 + a_{j+1}$.
 - La représentation obtenue pour y est canonique : on avait $a_{j+1} \in \{0, 1\}$, donc $b_{j+1} \in \{1, 2\}$; les autres chiffres, inchangés, sont dans $\{0, 1\}$.
 - $y = x - 2r_i + r_{i+1} = x + 1$ d'après la remarque de l'énoncé.

On a donc obtenu le successeur de x en changeant 1 ou 2 chiffres de la représentation.

Prédécesseur À nouveau, on considère le premier chiffre non nul de (la représentation canonique de) x . C'est le seul potentiellement égal à 2.

- S'il s'agit de a_0 , on le diminue de 1 : on obtient bien une représentation canonique de $y = x - 1$.
- S'il s'agit d'un a_i avec $i > 0$, on remplace a_i par $b_i = a_i - 1$ et a_{i-1} par $b_{i-1} = 2$. On obtient une représentation canonique (b_{i-1} vaut 2 mais b_i ne peut plus valoir 2) et $y = x + 2r_{i-1} - r_i = x - 1$.

Le prédécesseur s'obtient en changeant 1 ou 2 chiffres.

► Question 19 Les deux méthodes données plus haut nécessiteraient de parcourir la liste pour trouver le premier chiffre non nul, qui peut être arbitrairement loin.

► Question 20

```
let succ_d : = function
| x :: y :: xs when x = y -> (x + y + 1) :: xs
| xs -> 1 :: xs
```

► Question 21

```
let pred_d = function
| [] -> failwith "zero moins un"
| 1 :: xs -> xs
| x :: xs -> let y = x / 2 in y :: y :: xs
```

Listes en binaire décentré

► Question 22 On veut pouvoir renvoyer l'élément de rang 0 de la liste en temps constant. Il doit donc se trouver à la racine du premier arbre, ce qui ne sera le cas que si l'on stocke les étiquettes en ordre préfixe.
Pour la liste $(1, \dots, 13)$, on obtiendra

```
[(3, Nd (1, Fd 2, Fd 3));
 (3, Nd (4, Fd 5, Fd 6));
 (7, Nd (7,
          Nd (8, Fd 9, Fd 10),
          Nd (11, Fd 12, Fd 13)))]
```

► Question 23

```
(* Il faut connaître la taille (nombre total de noeuds) de l'arbre dans
lequel on fait la recherche, et déterminer si le noeud numéro i dans
l'ordre préfixe se trouve à la racine, à gauche ou à droite. *)

let rec get_tree_d t i taille =
  match t with
  | Fd x when i = 0 -> x
  | Nd (x, _, _) when i = 0 -> x
  | Nd (_, ga, _) when i <= taille / 2 -> get_tree_d ga (i - 1) (taille / 2)
  | Nd (_, _, dr) -> get_tree_d dr (i - 1 - taille / 2) (taille / 2)
  | _ -> failwith "get_tree_d : dépassement"

let rec get_d u i =
  match u with
  | [] -> failwith "get_d : dépassement"
  | (taille, t) :: xs ->
    if i < taille then get_tree_d t i taille
    else get_d xs (i - taille)
```

On a une complexité logarithmique en la taille de la liste.

► Question 24

```
let cons_d liste x =
  match liste with
  | (p1, t1) :: (p2, t2) :: ts when p1 = p2 ->
    (1 + p1 + p2, Nd (x, t1, t2)) :: ts
  | _ -> (1, Fd x) :: liste
```

► Question 25

```
let head = function
  | (_, Fd x) :: ts -> x
  | (_, Nd (x, _, _)) :: ts -> x
  | [] -> failwith "head : empty list"
```

► Question 26

```
let tail = function
  | (1, Fd x) :: ts -> ts
  | (p, Nd (x, t1, t2)) :: ts -> (p / 2, t1) :: (p / 2, t2) :: ts
  | _ -> failwith "tail : empty list"
```

ARBRES BINAIRES DE RECHERCHE EN C

Le but de cette séance est de programmer les opérations élémentaires sur les arbres binaires de recherche en C. Pour la grande majorité, nous les avons déjà vues en cours, mais l'idée est de :

- commencer par essayer d'écrire la fonction sans consulter le cours ;
- au besoin, consulter la description de l'algorithme dans le cours ;
- en désespoir de cause, consulter le code.

Nous allons utiliser le même type qu'en cours, mais avec un emballage supplémentaire :

```
typedef int item;

struct Node {
    item key;
    struct Node *left;
    struct Node *right;
};

typedef struct Node node;

struct BST {
    node *root;
};

typedef struct BST bst;
```

- Le type **node** correspond à un nœud (et donc à un sous-arbre), et il s'agit d'un type « interne » : les utilisateurs de la structure de donnée n'y auraient pas accès. Les fonctions qui manipulent des nœuds seront préfixées par **node_**.
- Le type **bst** correspond à un arbre binaire de recherche entier, sous forme d'un pointeur vers sa racine. Toutes les fonctions destinées à être fournies aux utilisateurs manipulent uniquement des **bst** et sont préfixées par **bst_**.
- Un nœud vide est représenté par un pointeur nul de type **node*** ; un *arbre* vide est un **bst*** dans lequel le champ **root** contient un pointeur nul.

Exercice XXVI.I – Fonctions utilitaires

p. 590

1. Écrire la fonction **new_node** créant un nouveau nœud, avec la clé fournie.
2. Écrire la fonction **bst_make_empty** renvoyant un nouvel arbre, vide.
3. Écrire la fonction **node_free** libérant la mémoire utilisée par un nœud ainsi que par tous ses descendants.
4. Écrire la fonction **bst_free** libérant la mémoire utilisée par un **bst**.

```
node *new_node(item x);
bst *bst_make_empty(void);
void node_free(node *n);
void bst_free(bst *t);
```

Exercice XXVI.2 – Insertion et construction

1. Écrire la fonction `node_insert` qui ajoute un élément à un sous-arbre. Cette fonction renverra la racine du sous-arbre modifié, et n'aura aucun effet si l'élément à ajouter est déjà présent dans le sous-arbre.
2. Écrire la fonction `bst_insert` qui ajoute un élément à un arbre binaire de recherche.
3. Écrire la fonction `bst_from_array` qui construit un arbre binaire de recherche en insérant un par un tous les éléments d'un tableau, dans l'ordre.

```
node *node_insert(node *t, item x);
void bst_insert(bst *t, item x);
bst *bst_from_array(item arr[], int len);
```

Exercice XXVI.3 – Parcours

1. Écrire deux fonctions `node_min` et `bst_min` permettant de déterminer le minimum d'un arbre binaire de recherche. On utiliser une assertion pour arrêter l'exécution de manière déterministe si l'arbre est vide.
2. Écrire deux fonctions `node_member` et `bst_member` testant si une certaine clé est présente dans un arbre.
3. Écrire deux fonctions `node_size` et `bst_size` renvoyant le nombre d'étiquettes d'un arbre.
4. Écrire deux fonctions `node_height` et `bst_height` renvoyant la hauteur. La hauteur d'un arbre vide vaut -1 .
5. Écrire une fonction `node_write_to_array` qui écrit les étiquettes présentes dans un sous-arbre dans le tableau fourni.
 - Les étiquettes seront écrites en ordre croissant.
 - La valeur initiale de `*offset_ptr` indique l'indice du tableau dans lequel il faudra écrire la première (la plus petite) étiquette.
 - La fonction devra avoir l'effet secondaire suivant : après l'appel, `*offset_ptr` indique l'indice immédiatement après celui de la dernière case dans laquelle on a écrit au cours de l'appel.
6. Écrire la fonction `bst_to_array` qui renvoie un tableau contenant les étiquettes d'un arbre binaire de recherche en ordre croissant. Cette fonction modifiera la valeur pointée par `nb_elts` pour qu'elle soit égale au nombre d'étiquettes (et donc à la taille du tableau renvoyé).

Remarque

`nb_elts` est purement un « argument de sortie » : sa valeur au début de l'appel n'a aucune importance.

```
item node_min(node *n);
item bst_min(bst *t);
bool node_member(node *n, item x);
bool bst_member(bst *t, item x);
int node_size(node *n);
int bst_size(bst *t);
int node_height(node *n);
int bst_height(bst *t);
void node_write_to_array(node *n, item arr[], int *offset_ptr);
item *bst_to_array(bst *t, int *nb_elts);
```

Exercice XXVI.4 – Suppression

p. 593

1. Écrire la fonction `node_extract_min`. Cette fonction supprime le minimum du sous-arbre passé en argument, et renvoie la nouvelle racine. De plus, elle a un effet secondaire : `*min_ptr` doit être égal à l'étiquette du minimum après l'appel.
2. Écrire les fonctions `node_delete` et `bst_delete`.

```
node *node_extract_min(node *n, item *min_ptr);
node *node_delete(node *n, item x);
void bst_delete(bst *t, item x);
```

Exercice XXVI.5 – Détermination expérimentale de la hauteur moyenne

p. 593

À l'aide des deux fonctions fournies dans le squelette, écrire un programme ayant le comportement suivant :

- il attend deux arguments entiers en ligne de commande, `max_power` et `rep_count` ;
- pour chaque entier k entre 4 et `max_power`, il génère `rep_count` arbres binaires de recherche de taille 2^k en insérant les éléments de $[0 \dots 2^k - 1]$ dans un ordre aléatoire ;
- pour chacun de ces arbres, il calcule la hauteur ;
- pour chacun des k , il affiche une ligne contenant la valeur de 2^k et la valeur moyenne de la hauteur, séparées par une espace.

On pourra ensuite générer un graphique semi-log à l'aide du script Python fourni, et conjecturer un équivalent de l'espérance de la hauteur en fonction de la taille n de l'arbre.

Solutions

Correction de l'exercice XXVI.I page 587

```
node* new_node(item x){
    node* n = malloc(sizeof(node));
    n->key = x;
    n->left = NULL;
    n->right = NULL;
    return n;
}

bst *bst_make_empty(void){
    bst *t = malloc(sizeof(bst));
    t->root = NULL;
    return t;
}

void node_free(node *n){
    if (n == NULL) return;
    node_free(n->left);
    node_free(n->right);
    free(n);
}

void bst_free(bst *t){
    node_free(t->root);
    free(t);
}
```

Correction de l'exercice XXVI.2 page 588

```
node *node_insert(node *t, item x){
    if (t == NULL) {
        return new_node(x);
    }
    if (t->key < x) {
        t->right = node_insert(t->right, x);
    }
    else if (t->key > x) {
        t->left = node_insert(t->left, x);
    }
    return t;
}

void bst_insert(bst *t, item x){
    t->root = node_insert(t->root, x);
}

bst *bst_from_array(item arr[], int len){
    bst *t = bst_make_empty();
    for (int i = 0; i < len; i++){
        bst_insert(t, arr[i]);
    }
    return t;
}
```

Correction de l'exercice XXVI.3 page 588

1.

```
item node_min(node *n){
    assert (n != NULL);
    while (n->left != NULL){
        n = n->left;
    }
    return n->key;
}

item bst_min(bst *t){
    return node_min(t->root);
}
```

2.

```
bool node_member(node *n, item x){
    if (n == NULL) return false;
    item key = n->key;
    if (x == key) return true;
    if (x < key) return node_member(n->left, x);
    return node_member(n->right, x);
}
```

```
bool bst_member(bst *t, item x){
    return node_member(t->root, x);
}
```

3.

```
int node_size(node *n){
    if (n == NULL) return 0;
    return node_size(n->left) + node_size(n->right) + 1;
}

int bst_size(bst *t){
    return node_size(t->root);
}
```

4.

```
int max(int x, int y){
    if (x <= y) return x;
    return y;
}

int node_height(node *n){
    if (n == NULL) return -1;
    return 1 + max(node_height(n->left), node_height(n->right));
}

int bst_height(bst *t){
    return node_height(t->root);
}
```

5.

```
void node_write_to_array(node *n, item arr[], int *offset_ptr){
    if (n == NULL) return;
    node_write_to_array(n->left, arr, offset_ptr);
    arr[*offset_ptr] = n->key;
    *offset_ptr = *offset_ptr + 1;
    node_write_to_array(n->right, arr, offset_ptr);
}
```

6.

```
item *bst_to_array(bst *t, int *nb_elts){
    int len = node_size(t->root);
    *nb_elts = len;
    item *arr = malloc(len * sizeof(item));
    int offset = 0;
    node_write_to_array(t->root, arr, &offset);
    return arr;
}
```

Correction de l'exercice XXVI.4 page 589

1.

```
node *node_extract_min(node *n, item *min_ptr){
    assert(n != NULL);
    if (n->left == NULL){
        node *result = n->right;
        *min_ptr = n->key;
        free(n);
        return result;
    }
    n->left = node_extract_min(n->left, min_ptr);
    return n;
}
```

2.

```
node *node_delete(node *n, item x){
    if (n == NULL) return n;
    if (x < n->key) {
        n->left = node_delete(n->left, x);
        return n;
    }
    if (x > n->key) {
        n->right = node_delete(n->right, x);
        return n;
    }
    if (n->left == NULL) {
        node *result = n->right;
        free(n);
        return result;
    }
    if (n->right == NULL) {
        node *result = n->left;
        free(n);
        return result;
    }
    item successor;
    n->right = node_extract_min(n->right, &successor);
    n->key = successor;
    return n;
}

void bst_delete(bst *t, item x){
    t->root = node_delete(t->root, x);
}
```

Correction de l'exercice XXVI.5 page 589

On ne met que la partie à ajouter au squelette :

```
bst *random_bst(item items[], int len){
    shuffle(items, len);
    return bst_from_array(items, len);
}

int average_height(int size, int rep_count){
    item *arr = malloc(size * sizeof(item));
    for (int i = 0; i < size; i++){
        arr[i] = i;
    }

    int sum = 0;
    for (int i = 0; i < rep_count; i++){
        bst *t = random_bst(arr, size);
        sum += bst_height(t);
        bst_free(t);
    }
    return (float)sum / rep_count;
}

int main(int argc, char* argv[]){
    assert(argc == 3);
    int pow_max = atoi(argv[1]);
    int rep_count = atoi(argv[2]);

    for (int k = 4; k <= pow_max; k++){
        int len = 1 << k;
        float avg = average_height(len, rep_count);
        printf("%d %f\n", len, avg);
    }

    return 0;
}
```

ARBRES BINAIRES DE RECHERCHE EN OCAML

I Fonctions élémentaires

On considère le type suivant :

```
type 'a abr =
| V
| N of 'a abr * 'a * 'a abr
```

Exercice XXVII.1

p. 599

Écrire les fonctions suivantes (les spécifications devraient être évidentes) :

```
insere : 'a abr -> 'a -> 'a abr
appartient : 'a abr -> 'a -> bool
cardinal : 'a abr -> int
```

Exercice XXVII.2

p. 599

1. Écrire la fonction `construit` qui prend en entrée une liste d'objets de type `'a` et renvoie l'arbre binaire de recherche obtenu en insérant successivement tous les éléments de la liste, dans l'ordre, dans un arbre initialement vide.
2. Écrire la fonction `elements` qui renvoie la liste des éléments d'un arbre binaire de recherche, dans l'ordre croissant. On exige une complexité en $O(|t|)$.

```
construit : 'a list -> 'a abr
elements : 'a abr -> 'a list
```

Exercice XXVII.3

p. 600

1. Écrire une fonction `extrait_min` qui prend en entrée un ABR `t`, supposé non vide, et renvoie le couple (m, t') , où :
 - m est le minimum de `t`;
 - t' est l'arbre binaire de recherche obtenu en supprimant m de `t`.
2. Écrire la fonction `supprime` qui supprime un élément d'un arbre binaire de recherche. Si l'élément fourni n'appartient pas à l'arbre, ce dernier sera renvoyé inchangé.

```
extrait_min : 'a abr -> 'a * 'a abr
supprime : 'a abr -> 'a -> 'a abr
```

2 Fonctions supplémentaires

Exercice XXVII.4 – Séparation d'un ABR

p. 600

Écrire une fonction `separe` telle que l'appel `separe t x` renvoie un couple (inf, sup) d'ABR vérifiant :

- tous les éléments de `inf` sont inférieurs ou égaux à `x`;
- tous les éléments de `sup` sont strictement supérieurs à `x`;
- la réunion des éléments de `inf` et de ceux de `sup` est égal à l'ensemble des éléments de `t`.

On demande une complexité en $O(h(t))$.

Exercice XXVII.5

p. 600

1. Écrire une fonction `verifie_abr` qui détermine si l'arbre passé en argument vérifie la condition d'ordre des ABR. On n'hésitera pas à utiliser les fonctions préalablement définies, et l'on précisera les complexités en temps et en espace de `verifie_abr`.
 2. Écrire une fonction `tab_elements` qui convertit un ABR en un tableau trié.
 3. Ré-écrire les fonctions `verifie_abr` et `tab_elements` pour que leur complexité en espace (sans compter la taille du résultat pour `tab_elements`) soit en $O(h(t))$ ⁴. Pour la fonction `verifie_abr`, on pourra se limiter au cas des arbres à étiquettes entières (et supposer qu'aucun nœud ne porte l'étiquette `min_int`).
- a. On réfléchira aussi à la question suivante : pourquoi l'énoncé demande-t-il $O(h(t))$ et non $O(1)$?

3 Structure de multi-ensemble ordonné

On considère un type totalement ordonné '`a`', et l'on souhaite représenter des *multi-ensembles* d'éléments de '`a`' de manière à pouvoir réaliser un certain nombre d'opérations de manière efficace. On rappelle que dans un multi-ensemble, chaque élément possède une *multiplicité* (ou nombre d'occurrences).

La liste des opérations qui nous intéressent :

- `get_occurrences` : '`a multiset` -> '`a` -> `int` qui renvoie le nombre d'occurrences (éventuellement nul) d'un objet de type '`a` dans un '`a multiset`';
- `add_occurrence` : '`a multiset` -> '`a` -> '`a multiset` qui ajoute une occurrence;
- `rem_occurrence` : '`a multiset` -> '`a` -> '`a multiset` qui enlève une occurrence;
- `size` : '`a multiset` -> `int` qui renvoie le nombre total d'éléments dans un multi-ensemble, en tenant compte de la multiplicité;
- `select` : '`a multiset` -> `int` -> '`a` qui renvoie x_i , où $x_0 < x_1 < \dots < x_{\text{size}(t)}$ sont les éléments de `t`, avec multiplicité (on lèvera une exception si i n'est pas un indice valide).

3.1 Utilisation d'un dictionnaire

Une première idée serait de remplacer la structure (g, x, r) d'un ABR par (g, x, mul, r) , où `mul` est un entier (strictement positif) indiquant la multiplicité de `x`. Cette idée fonctionne, et correspond en fait à un cas particulier de dictionnaire à clé de type '`a`' et valeur de type `int`.

Exercice XXVII.6

p. 602

On définit le type suivant :

```
type ('k, 'v) dict =
| Empty
| Node of ('k, 'v) dict * 'k * 'v * ('k, 'v) dict
```

Écrire les fonctions suivantes (vues en cours) :

1. `get` qui renvoie `Some v` si la clé fournie est associée à la valeur `v`, `None` sinon;
2. `set` qui crée une association, ou remplace la valeur associée à une clé s'il y en avait déjà une;
3. `remove` qui supprime l'association correspondant à la clé fournie s'il y en avait une, et ne fait rien sinon.

```
get : ('k, 'v) dict -> 'k -> 'v option  
  
set : ('k, 'v) dict -> 'k -> 'v -> ('k, 'v) dict  
  
remove : ('k, 'v) dict -> 'k -> ('k, 'v) dict
```

Exercice XXVII.7

p. 603

1. Écrire les fonctions `get_occurrences`, `add_occurrence` et `rem_occurrence` à l'aide des fonctions `get`, `set` et `remove`.
2. Donner la complexité de ces trois fonctions.

```
get_occurrences : ('a, int) dict -> 'a -> int  
  
add_occurrence : ('a, int) dict -> 'a -> ('a, int) dict  
  
rem_occurrence : ('a, int) dict -> 'a -> ('a, int) dict
```

Exercice XXVII.8

p. 603

1. Écrire la fonction `size`, et déterminer sa complexité.
2. Proposer un algorithme pour la fonction `select` (on ne demande pas de l'implémenter en OCaml).
3. Quelle est la complexité de cet algorithme ?

```
size : ('a, int) dict -> int  
  
select : ('a, int) dict -> int -> 'a
```

3.2 Enrichissement de la structure

Pour obtenir des fonctions `select` et `size` plus efficaces, on décide d'enrichir la structure en ajoutant dans chaque nœud (non vide) un entier indiquant la taille (nombre d'éléments avec multiplicité) du sous-arbre correspondant.

```
type 'a multiset =  
| Empty  
| Node of int * 'a multiset * 'a multiset * 'a multiset
```

On maintiendra les invariants suivants :

- en considérant uniquement les étiquettes de type `'a`, on a un ABR;
- dans un nœud `t = Node (n, left, x, i, right)`, on a $i > 0$ et n égal à la taille de `t` (avec multiplicité).

Exercice XXVII.9

p. 604

1. Écrire les fonctions `get_occurrences`, `add_occurrence` et `rem_occurrence`.
2. Écrire les fonctions `size` et `select`, et déterminer leur complexité.

4 Un problème pour finir

Ce problème est adapté du *Projet Euler* (projecteuler.net), site qui contient des centaines de problèmes intéressants sur lesquels vous pouvez travailler (problèmes beaucoup plus mathématiques en moyenne que ceux de France-IOI).

On définit deux suites $(u_k)_{k \geq 1}$ et $(v_k)_{k \geq 1}$ par :

- $u_k = (p_k)^k \bmod 10007$, où p_k est le k -ème nombre premier (avec donc $p_1 = 2$);
- $v_k = u_k + u_{\lfloor k/10000 \rfloor + 1}$

On définit ensuite $M(i, j)$ pour $i \leq j$ comme la médiane des éléments v_i, \dots, v_j , en convenant que la médiane d'une série de longueur paire est la moyenne des deux éléments centraux. On a alors $M(1, 10) = 2\ 021,5$ et $M(10^2, 10^3) = 4\ 715$.

Finalement, on pose $F(n, k) = \sum_{i=1}^{n-k+1} M(i, i+k-1)$. On a alors $F(100, 10) = 433\ 628,5$.

Exercice XXVII.10

1. En utilisant ce que l'on a fait depuis le début du sujet, déterminer $F(10^5, 10^4)$.
2. En essayant de garder une consommation mémoire raisonnable (quelques centaines de mégaoctets, mais pas quelques giga-octets), déterminer $F(10^7, 10^5)$.
3. Proposer une solution plus simple en utilisant le fait que le modulo utilisé est petit.

Solutions

Correction de l'exercice XXVII.1 page 595

```
let rec insere t x =
  match t with
  | V -> N (V, x, V)
  | N (l, y, r) ->
    if x = y then t
    else if x < y then N (insere l x, y, r)
    else N (l, y, insere r x)

let rec appartient t x =
  match t with
  | V -> false
  | N (l, y, r) ->
    (x = y) || (x < y && appartient l x) || (x > y && appartient r x)

let rec cardinal t =
  match t with
  | V -> 0
  | N (gauche, _, droite) -> 1 + cardinal gauche + cardinal droite
```

Correction de l'exercice XXVII.2 page 595

```
let construit u =
  let rec aux v acc =
    match v with
    | [] -> acc
    | x :: xs -> aux xs (insere acc x) in
      aux u V

(* Version efficace (en  $O(|t|)$ ). *)
let elements t =
  let rec aux arbre acc =
    match arbre with
    | V -> acc
    | N (g, x, d) ->
      let avec_d = aux d acc in
      aux g (x :: avec_d) in
        aux t []
```

Correction de l'exercice XXVII.3 page 595

```

let rec extrait_min t =
  match t with
  | V -> failwith "vide"
  | N (V, x, d) -> (x, d)
  | N (g, x, d) ->
    let m, g' = extrait_min g in
    (m, N (g', x, d))

let rec supprime t x =
  match t with
  | V -> V
  | N (g, y, d) when x < y -> N (supprime g x, y, d)
  | N (g, y, d) when x > y -> N (g, y, supprime d x)
  | N (V, y, d) (* y = x *) -> d
  | N (g, y, V) (* y = x *) -> g
  | N (g, y, d) (* y = x *) ->
    let successeur, d' = extrait_min d in
    N (g, successeur, d')

```

Correction de l'exercice XXVII.4 page 596

On descend le long d'une branche de l'arbre jusqu'à trouver x ou arriver à un nœud vide, en faisant des opérations en temps constant à chaque niveau : la complexité est bien en $O(h(t))$.

```

let rec separe t x =
  match t with
  | V -> V, V
  | N (l, y, r) ->
    if x = y then N (l, y, V), r
    else if x < y then
      let lo, hi = separe l x in
      lo, N (hi, y, r)
    else
      let lo, hi = separe r x in
      N (l, y, lo), hi

```

Correction de l'exercice XXVII.5 page 596

1. Un arbre est un ABR si et seulement si ses étiquettes lues dans l'ordre infixé forment une suite (strictement) croissante :

```

let verifie_abr t =
  let rec croissant u =
    match u with
    | x :: y :: xs -> (x < y) && croissant (y :: xs)
    | _ -> true in
  croissant (elements t)

```

La fonction `elements` a une complexité en temps en $O(|t|)$, et la fonction `croissant` en $O(|u|) = O(|t|)$. On obtient donc une complexité en temps en $O(|u|)$.

Pour la complexité en espace, il y a deux choses à considérer :

- le stockage auxiliaire est constitué de la liste `elements t`, de longueur $|t|$;

- il faut aussi prendre en compte l'espace consommé sur la pile d'appels : la fonction auxiliaire de `elements` a une profondeur d'appel en $O(h(t))$ et la fonction `croissant` en $O(|t|)$ dans le pire cas (qui sera atteint dès que t est effectivement un ABR).

On obtient donc une complexité en espace en $O(|t|)$.

2. Avec tout ce que l'on a déjà écrit, le plus simple est clairement :

```
let tab_elements t =
  Array.of_list (elements t)
```

3. Le parcours de l'arbre (qui est clairement nécessaire) nécessite un espace $O(h(t))$, que ce soit pour la pile d'appel si on l'effectue récursivement ou pour la pile « tout court » si on choisit une version itérative (ou récursive terminale). Pour ne pas dépenser plus que cela, il faut se débarrasser de la liste `elements t`.

Pour `verifie_abr`, plusieurs solutions sont envisageables : on en présente deux ici.

```
let verifie_abr_bis t =
  let courant = ref min_int in
  let rec aux t =
    match t with
    | V -> true
    | N (g, x, d) ->
      aux g && x > !courant && (courant := x; aux d) in
    aux t

exception Faux

let verifie_abr_exception t =
  let courant = ref min_int in
  let rec aux t =
    match t with
    | V -> ()
    | N (g, x, d) ->
      aux g;
      if x <= !courant then raise Faux;
      courant := x;
      aux d in
    try
      aux t;
      true
    with
    | Faux -> false
```

Pour `tab_elements`, on peut aussi procéder de plusieurs manières mais le plus simple est sans doute :

```
let tab_elements_bis t =
  match t with
  | V -> []
  | N (_, x, _) ->
    let n = cardinal t in
    let arr = Array.make n x in
    let indice = ref 0 in
    let rec aux arbre =
      match arbre with
      | V -> ()
      | N (g, x, d) ->
        aux g;
        arr.(!indice) <- x;
        incr indice;
        aux d in
    aux t;
    arr
```

On prend garde à gérer correctement le cas où l'arbre est vide, et à ne pas se limiter aux arbres à étiquettes entières (ce qui nécessite de récupérer une étiquette pour initialiser le tableau).

Correction de l'exercice XXVII.6 page 596

```
let rec get dict key =
  match dict with
  | Empty -> None
  | Node (left, k, v, right) ->
    if key = k then Some v
    else if key < k then get left key
    else get right key

let rec set dict key value =
  match dict with
  | Empty -> Node (Empty, key, value, Empty)
  | Node (left, k, v, right) ->
    if key = k then Node (left, k, value, right)
    else if key < k then Node (set left key value, k, v, right)
    else Node (left, k, v, set right key value)

let rec extract_min = function
  | Empty -> failwith "empty"
  | Node (Empty, k, v, right) ->
    (k, v, right)
  | Node (left, k, v, right) ->
    let km, vm, left' = extract_min left in
    (km, vm, Node (left', k, v, right))
```

```

let rec remove dict key =
  match dict with
  | Empty -> Empty
  | Node (left, k, v, right) when key < k ->
    Node (remove left key, k, v, right)
  | Node (left, k, v, right) when key > k ->
    Node (left, k, v, remove right key)
  | Node (Empty, k, v, child) | Node (child, k, v, Empty) ->
    child
  | Node (left, k, v, right) ->
    let km, vm, right' = extract_min right in
    Node (left, km, vm, right')
  
```

Correction de l'exercice XXVII.7 page 597

- C'est très simple avec ce que l'on a écrit :

```

let get_occurrences ms x =
  match get ms x with
  | None -> 0
  | Some i -> i

let add_occurrence ms x =
  let i = get_occurrences ms x in
  set ms x (i + 1)

let rem_occurrence ms x =
  let i = get_occurrences ms x in
  if i = 1 then remove ms x
  else set ms x (i - 1)
  
```

- get_occurrences a la même complexité que get, c'est-à-dire $O(h)$. add_occurrence fait un appel à get_occurrence et un appel à set, tous deux en $O(h)$, donc est à nouveau en $O(h)$. Finalement, rem_occurrence fait un appel à get_occurrence puis soit un appel à remove soit un appel à set, donc toujours du $O(h)$.

Correction de l'exercice XXVII.8 page 597

- Il suffit de parcourir l'arbre en sommant les nombres d'occurrences, pour une complexité en $O(n)$:

```

let rec size = function
  | Empty -> 0
  | Node (left, _, i, right) ->
    i + size left + size right
  
```

- Il n'y a rien de très satisfaisant pour select avec cette structure de données. Une possibilité est de commencer par convertir l'arbre en une liste de couple $(x, \text{occ}(x))$ classée par x croissants (grâce à un parcours infixé), en un temps $O(n)$. Ensuite, on parcourt cette liste en sommant les occurrences jusqu'à dépasser l'indice souhaité, ce qui se fait à nouveau en $O(n)$ (dans le pire des cas).

Correction de l'exercice XXVII.9 page 598

1. Pour get_occurrences et add_occurrence, on adapte get et set en mettant à jour les tailles :

```
let rec get_occurrences ms x =
  match ms with
  | E -> 0
  | N (_, left, y, i, right) ->
    if x = y then i
    else if x < y then get_occurrences left x
    else get_occurrences right x

let rec add_occurrence ms x =
  match ms with
  | E -> N (1, E, x, 1, E)
  | N (n, left, y, i, right) ->
    if x = y then N (n + 1, left, x, i + 1, right)
    else if x < y then N (n + 1, add_occurrence left x, y, i, right)
    else N (n + 1, left, y, i, add_occurrence right x)
```

Pour rem_occurrence, il faut faire attention : on ne peut pas savoir si les tailles doivent être modifiées avant de savoir s'il y a une occurrence à supprimer. Le plus simple est de faire deux parcours : un pour savoir si l'élément est présent, et, au besoin, un pour supprimer une occurrence (en sachant que les tailles de tous les sous-arbres rencontrés doivent être diminuées de une unité).

```
let rec extract_min ms =
  match ms with
  | E -> failwith "minimum of empty multiset"
  | N (_, E, x, i, right) ->
    (x, i, right)
  | N (n, left, x, i, right) ->
    let m, i_m, left' = extract_min left in
    (m, i_m, N (n - i_m, left', x, i, right))

let rec rem_occurrence_aux ms x =
  match ms with
  | E -> E
  | N (n, left, y, i, right) when x < y ->
    N (n - 1, rem_occurrence_aux left x, y, i, right)
  | N (n, left, y, i, right) when x > y ->
    N (n - 1, left, y, i, rem_occurrence_aux right x)
  | N (_, E, y, 1, ms') | N (_, ms', y, 1, E) -> ms'
  | N (n, left, y, 1, right) ->
    let m, i_m, right' = extract_min right in
    N (n - 1, left, m, i_m, right')
  | N (n, left, y, i, right) -> N (n - 1, left, y, i - 1, right)

let rem_occurrence ms x =
  if get_occurrences ms x = 0 then ms
  else rem_occurrence_aux ms x
```

2. Les fonctions size et select s'écrivent très facilement :

```
let size = function
| E -> 0
| N (n, _, _, _, _) -> n

let rec select ms index =
  match ms with
  | E -> failwith "invalid index"
  | N (n, left, x, i, right) ->
    if index < size left then select left index
    else if index < size left + i then x
    else select right (index - size left - i)
```

size est bien évidemment en O(1), et pour select on ne parcourt qu'une branche de l'arbre, avec des opérations en temps constant (y compris les appels à size) à chaque nœud traversé : la complexité est en O(h).

ARBRES ROUGE-NOIR EN OCAML

L'objectif de ce sujet est d'obtenir une implémentation complète des arbres rouge-noir fonctionnels en OCaml : test d'appartenance, insertion et suppression. Nous allons utiliser une version des arbres rouge-noir légèrement différente de celle vue en cours (et nous ne réfléchirons pas en termes d'arbre 2-3).

Le type :

```
type 'a rn =
| V
| N of 'a rn * 'a * 'a rn
| R of 'a rn * 'a * 'a rn
```

Les contraintes :

- les étiquettes lues dans l'ordre infixé sont strictement croissantes ;
- un nœud rouge ne peut pas avoir de fils rouge ;
- tous les chemins de la racine à un nœud vide contiennent le même nombre de nœuds noirs ;
- la racine est noire.

Remarque

Dans le cours, nous avons utilisé des arbres rouge-noir *gauches*, dans lesquels on interdisait les fils droits rouges. Ce n'est pas le cas ici.

On appellera :

- *arbre rouge-noir correct* un arbre vérifiant les quatre conditions ci-dessus ;
- *sous-arbre rouge-noir correct* un arbre vérifiant les trois premières conditions (et peut-être la dernière) ;
- *sous-arbre rouge-noir presque correct* un arbre vérifiant les trois premières conditions, sauf que sa racine peut être rouge et posséder un (ou deux) fils rouges.

I Insertion

Le principe est essentiellement le même que dans le cours : on crée une nouvelle feuille rouge avec la clé à insérer, puis on corrige les problèmes en remontant jusqu'à la racine. Cependant, les cas à considérer sont un peu différents, et l'on ne fera pas *explicitement* de rotation : on remplacera cela par un usage massif du filtrage par motif.

Comme dans la méthode vue en cours, on ne violera jamais la condition d'équilibre noir. Le seul problème potentiel sera un nœud rouge n avec un fils rouge, et la résolution de ce problème sera la responsabilité du père (nécessairement noir) de n .

► **Question 1** Dessiner les quatre cas problématiques possibles pour le père de n , et montrer que tous ces cas peuvent être résolus exactement de la même manière, de façon à obtenir un sous-arbre rouge noir correct dans lequel les hauteurs noires n'ont pas été modifiées.

► **Question 2** Écrire une fonction `corrige_rouge` qui prend en entrée un arbre et :

- effectue la transformation de la question précédente si c'est nécessaire (racine noire, un fils rouge qui a un fils rouge) ;
- renvoie l'arbre tel que sinon.

Cette fonction renverra un sous-arbre rouge-noir presque correct.

```
corrige_rouge : 'a rn -> 'a rn
```

► **Question 3** Écrire une fonction `insere_aux` qui prend en entrée un sous-arbre rouge-noir correct et renvoie un sous-arbre rouge-noir presque correct dans lequel la clé fournie a été insérée.

```
insere_aux : 'a rn -> 'a -> 'a rn
```

► **Question 4** Écrire la fonction `insere`, qui prend en entrée un arbre rouge-noir correct et une clé, et renvoie un arbre rouge-noir correct dans lequel la clé a été insérée.

```
insere : 'a rn -> 'a -> 'a rn
```

2 Suppression

Le principe général de la suppression est le suivant :

- on commence par rechercher l'élément à supprimer (s'il n'est pas présent, il n'y a rien à faire);
- s'il a au plus un fils non vide, on le supprime;
- sinon, on le remplace par son successeur (le minimum de son fils droit) et on supprime ce successeur;
- dans les deux cas, on risque d'avoir introduit une violation de la condition d'équilibre noire;
- on déplace ce problème vers le haut de l'arbre, ou on le règle suivant les cas;
- en s'occupant de ce problème, on risque de violer la condition rouge-rouge, mais il sera toujours possible de rétablir immédiatement cette propriété.

► **Question 5 Cas de base pour la suppression.**

Il y a quatre cas de base où l'on peut directement supprimer un élément, suivant que le nœud à supprimer est rouge ou noir et que son fils gauche ou droit est vide. On a représenté ci-dessous les deux cas correspondant à un fils gauche vide, avec les conventions suivantes (valables pour toute la suite) :

- les nœuds rouges sont en rouge **et en gras** (donc visibles après impression...);
- les arêtes rouges (celles menant à un nœud rouge) également;
- les nœuds noirs sont grisés;
- les arêtes noires sont en trait plein d'épaisseur normale;
- les arêtes en pointillés sont de couleur inconnue.



FIGURE XXVIII.1 – Cas de base pour la suppression.

Indiquer dans les deux cas le résultat de la suppression, en précisant si la hauteur noire a été modifiée ou non (et si oui, comment).

2.1 Suppression du minimum

On souhaite écrire une fonction `supprime_min` ayant la spécification suivante :

Entrées : un sous-arbre rouge-noir correct t , non vide;

Sorties : un sous-arbre rouge-noir correct t' et un booléen b .

Post-conditions :

- $\text{étiquettes}(t') = \text{étiquettes}(t) \setminus \{\min t\}$;
- en notant h la hauteur noire de t et h' celle de t' , on a soit $h' = h$, soit $h' = h - 1$;
- b est vrai si $h' = h - 1$, faux sinon.

Cette fonction va avoir la structure suivante :

```

1 let rec supprime_min arbre =
2   match arbre with
3   | V -> failwith "vide"
4   | R (V, x, d) -> ...
5   | N (V, x, d) -> ...
6   | R (g, x, d) | N (g, x, d) ->
7     let g', a_diminue = supprime_min g in
8     ...

```

► **Question 6** Compléter les lignes 4 et 5 de la fonction.

Quand on récupère le couple g' , $a_diminue$ (qu'il faut lire « a diminué »!), on ne peut pas *a priori* renvoyer $\text{cons arbre } g' \times d$ puisque la hauteur noire de g' peut être inférieure (de une unité) à celle de d . Il faut donc écrire une fonction permettant de rétablir l'équilibre noir dans ce cas. Les différents cas sont présentés ci-dessous :

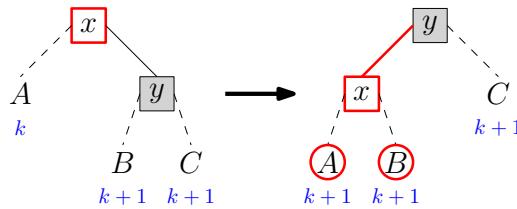


FIGURE XXVIII.2 – `repare_noir_gauche`, racine rouge.

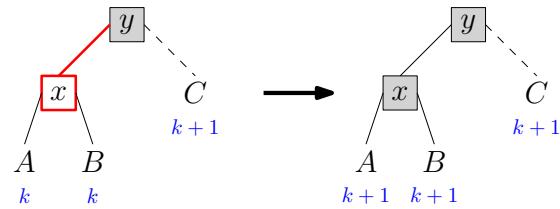


FIGURE XXVIII.3 – `repare_noir_gauche`, racine noire et fils gauche rouge.

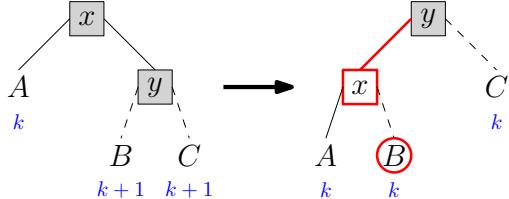


FIGURE XXVIII.4 – `repare_noir_gauche`, racine noire, deux fils noirs.

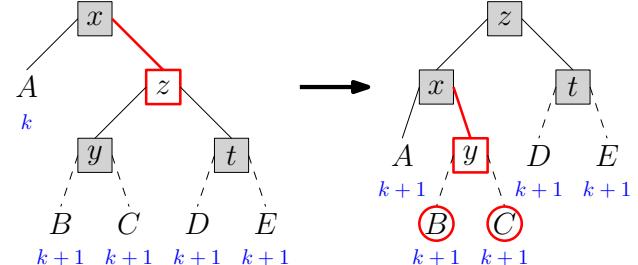


FIGURE XXVIII.5 – `repare_noir_gauche`, racine noire et fils droit rouge.

► **Question 7** Quelle est la signification des cercles rouges présents autour de certains nœuds dans les situations finales ?

► **Question 8** Justifier que tous les cas sont présents, et bien traités.

► **Question 9** Écrire la fonction `repare_noir_gauche`, dont la spécification est la suivante :

Entrées : un arbre t et un booléen b .

Sorties : un arbre t' et un booléen b' .

Pré-conditions :

- si b est faux, alors t est un sous-arbre rouge-noir correct;
- si b est vrai, alors t est de la forme (g, x, d) (la racine de t pouvant être rouge ou noire), g et d sont deux sous-arbres rouge-noir corrects, et la hauteur noire de d vaut exactement un de plus que celle de g .

Post-conditions :

- t' est un sous-arbre rouge-noir presque correct;
- les étiquettes de t' sont exactement celles de t ;
- la hauteur de t' est
 - soit égale à celle de t , et dans ce cas b' vaut **false**;
 - soit égale à celle de t moins un, et dans ce cas b' vaut **true**.

► **Question 10** Compléter la fonction `supprime_min`.

```
supprime_min : 'a rn -> ('a rn * bool)
```

2.2 Suppression d'un élément quelconque

Quand on supprime un élément quelconque, on est amené à traiter le cas d'un arbre dont le fils *droit* possède une hauteur noire inférieure (de une unité) à celle du fils gauche : c'est par exemple le cas si la suppression du successeur a fait diminuer la hauteur du fils droit.

► **Question 11** Représenter les cas symétriques de ceux des figures XXVIII.2 à XXVIII.5.

► **Question 12** En déduire la fonction `repare_noir_droite`, « symétrique » de `repare_noir_gauche`.

```
repare_noir_droite : 'a rn -> bool -> ('a rn * bool)
```

► **Question 13** Écrire une fonction `supprime_aux`, qui prend en entrée un sous-arbre rouge-noir correct t et une clé x et renvoie un couple (t', b) tel que :

- t' est un sous-arbre rouge-noir correct;
- étiquettes(t') = étiquettes(t) \ { x };
- en notant h la hauteur noire de t et h' celle de t' , on a
 - soit $h' = h$, et dans ce cas $b = \text{false}$;
 - soit $h' = h - 1$, et dans ce cas $b = \text{true}$.

```
supprime_aux : 'a rn -> 'a -> ('a rn * bool)
```

► **Question 14** Écrire finalement la fonction `supprime`, dont la spécification devrait aller de soi (elle doit renvoyer un arbre rouge-noir correct).

```
supprime : 'a rn -> 'a -> 'a rn
```

Solutions

I Insertion

► Question 1

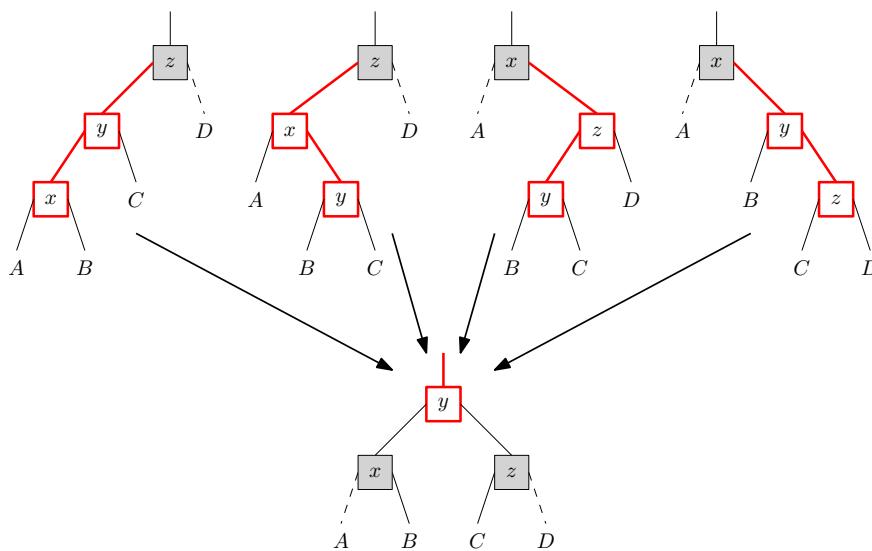


FIGURE XXVIII.6 – Traitement des quatre cas rouge-rouge.

► Question 2

Sans le dessin, cette fonction peut faire un peu peur... Avec le dessin, en revanche, ça devrait être limpide.

```
let corrige_rouge = function
| N (R (R (a, x, b), y, c), z, d)
| N (R (a, x, R (b, y, c)), z, d)
| N (a, x, R (R (b, y, c), z, d))
| N (a, x, R (b, y, R (c, z, d)))
-> R (N (a, x, b), y, N (c, z, d))
| t -> t
```

► Question 3

On utilise la fonction `cons` fournie pour éviter d'écrire deux fois la même chose. L'appel à `corrige_rouge` sera systématiquement inutile quand la racine de `t` est rouge, mais ce n'est pas grave (`corrige_rouge` ne fera simplement rien dans ce cas).

```
let rec insere_aux t x =
match t with
| V -> R (V, x, V)
| R (l, y, r) | N (l, y, r) ->
  if x = y then t
  else if x < y then corrige_rouge (cons t (insere_aux l x) y r)
  else corrige_rouge (cons t l y (insere_aux r x))
```

- **Question 4** Il ne reste plus qu'à gérer la racine :

```
let noircit = function
| R (g, x, d) -> N (g, x, d)
| t -> t

let insere t x =
  noircit (insere_aux t x)
```

2 Suppression

- **Question 5** On renvoie A dans les deux cas. Dans le premier cas, la hauteur noire du nouvel arbre est un de moins que celle de l'arbre initial, dans le deuxième cas c'est la même.

2.1 Suppression du minimum

- **Question 6** Ces lignes correspondent aux cas de base de la question précédente :

```
let rec supprime_min arbre =
  match arbre with
  | V -> failwith "vide"
  | R (V, x, d) -> d, false
  | N (V, x, d) -> d, true
  | R (g, x, d) | N (g, x, d) -> ...
```

Remarque

On avait parlé de cas symétriques à la question précédente : ils ne sont pas présents ici, puisque l'on supprime le *minimum* et qu'on a donc toujours un fils *gauche* vide.

- **Question 7** Les cercles rouges indiquent les nœuds susceptibles d'être rouges alors qu'ils ont un père rouge.

- **Question 8** On suppose qu'on partait d'un sous-arbre rouge-noir correct, que le sous-arbre gauche (résultat de la suppression) est un sous-arbre rouge-noir correct et qu'il a une hauteur noire strictement plus petite que celle du sous-arbre droit. En particulier, cela signifie que la hauteur noire du fils droit est non nulle.

- Si la racine est rouge, le sous-arbre droit a une racine noire et n'est pas vide : c'est notre premier cas.
- Si la racine est noire :
 - soit le fils gauche est rouge (et donc non vide), ce qui correspond au deuxième cas ;
 - soit les deux fils sont noirs, et le fils droit non vide d'après la remarque ci-dessus : c'est le troisième cas ;
 - soit le fils gauche est noir et le fils gauche rouge. Les enfants du fils droit sont donc noirs, et comme la hauteur noire du fils droit est non nulle, ils ne peuvent pas être vides : c'est le quatrième cas.

Le fait que les cas soient correctement traités se vérifie facilement, surtout si l'on considère les annotations k / k + 1 présentes sur les schémas. Les arbres obtenus en sortie vérifient la condition d'équilibre noir (dans le troisième cas, la hauteur noire a décrue).

- **Question 9** À nouveau, c'est surtout une traduction du dessin. Il ne faut pas oublier d'appliquer *corrige_rouge* quand c'est nécessaire (et de l'appliquer au bon endroit, c'est-à-dire sur le père noir du nœud rouge ayant un fils rouge). Remarquons que dans le premier et le dernier cas, on peut appeler *corrige_rouge* sur un noeud noir ayant un fils rouge dont *les deux fils* sont rouges. Nous n'avions pas considéré ce cas dans *corrige_rouge*, mais on vérifie facilement qu'il est correctement traité.

```

let corrige_noir_gauche arbre a_faire =
  if not a_faire then (arbre, false)
  else match arbre with
    | R (a, x, N (b, y, c)) ->
      corrige_rouge (N (R (a, x, b), y, c)),
      false
    | N (R (a, x, b), y, c) ->
      N (N (a, x, b), y, c),
      false
    | N (a, x, N (b, y, c)) ->
      corrige_rouge (N (R (a, x, b), y, c)),
      true
    | N (a, x, R (N (b, y, c), z, N (d, t, e))) ->
      N (corrige_rouge (N (a, x, R (b, y, c))),
          z,
          N (d, t, e)),
      false
    | _ -> failwith "impossible"
  
```

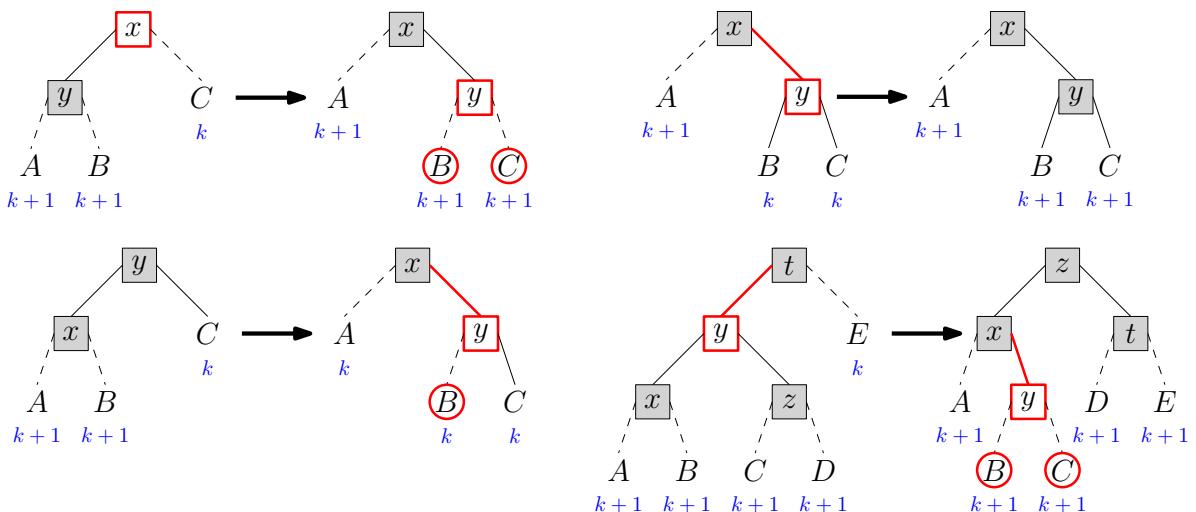
► **Question 10** Tout le travail a été fait : on pense juste à utiliser la fonction `cons` pour regrouper les deux cas identiques.

```

let rec supprime_min arbre =
  match arbre with
  | V -> failwith "vide"
  | R (V, x, d) -> d, false
  | N (V, x, d) -> d, true
  | R (g, x, d) | N (g, x, d) ->
    let g', a_diminue = supprime_min g in
    corrige_noir_gauche (cons arbre g' x d) a_diminue
  
```

2.2 Suppression d'un élément quelconque

► **Question 11**



► Question 12 On traduit les schémas :

```
let corrige_noir_droite arbre a_faire =
  if not a_faire then (arbre, false)
  else match arbre with
    | R (N (a, x, b), y, c) ->
      corrige_rouge (N (a, x, R (b, y, c))), 
      false
    | N (a, x, R (b, y, c)) ->
      N (a, x, N (b, y, c)),
      false
    | N (N (a, x, b), y, c) ->
      corrige_rouge (N (a, x, R (b, y, c))), 
      true
    | N (R (N (a, x, b), y, N (c, z, d)), t, e) ->
      N (corrige_rouge (N (a, x, R (b, y, c))), 
          z,
          N (d, t, e)),
      false
    | _ -> failwith "impossible"
```

► Question 13 Le plus gros du travail a déjà été fait, mais il faut quand même être soigneux.

```
let rec supprime_aux t x =
  match t with
  | V -> V, false
  | N (g, y, d) | R (g, y, d) when x < y ->
    let g', a_diminue = supprime_aux g x in
    corrige_noir_gauche (cons t g' y d) a_diminue
  | N (g, y, d) | R (g, y, d) when x > y ->
    let d', a_diminue = supprime_aux d x in
    corrige_noir_droite (cons t g y d') a_diminue
  | N (V, _, t') | N (t', _, V) -> t', true
  | R (V, _, t') | R (t', _, V) -> t', false
  | N (g, _, d) | R (g, _, d) ->
    let m = minimum d in
    let d', a_diminue = supprime_min d in
    corrige_noir_droite (cons t g m d') a_diminue
```

► Question 14 Tout le travail a déjà été fait!

```
let supprime t x =
  let t', _ = supprime_aux t x in
  noircit t'
```

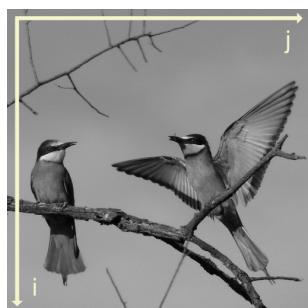
SEAM CARVING

Le but de ce TP est de réduire automatiquement la largeur d'une image sans toutefois changer la taille des zones les plus intéressantes de cette dernière. L'algorithme que nous allons implémenter, appelé *seam carving* est implémenté dans PHOTOSHOP sous le nom de *content aware scaling*.



I Travailler avec des images

Dans ce TP, nous travaillerons avec des images en niveaux de gris, où la valeur d'un pixel peut varier de 0 (pixel noir) à 255 (pixel blanc). Pour stocker cette valeur, nous utilisons donc le type entier non signé `uint8_t`.



Une image est donc une matrice de pixels que nous stockerons dans la structure suivante :

```
struct image {
    uint8_t **at;
    int h;
    int w;
};
typedef struct image image;
```

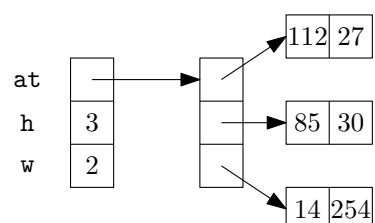


FIGURE XXIX.1 – Schéma mémoire pour une image.

Si `im` est un `image*`, on pourra donc utiliser `im->at[i]` pour accéder à la ligne `i` de l'image (qui est de type `uint8_t*`) et `im->at[i][j]` pour accéder à la valeur du pixel `(i, j)` (qui est de type `uint8_t`).

► **Question 1** Écrire une fonction `image_new` renvoyant un pointeur `im` vers une image allouée ayant la hauteur et la largeur spécifiées en paramètre. On fera bien attention aux problèmes d'*aliasing* : si le nombre d'appels à `malloc` que vous effectuez ne dépend pas de la hauteur de l'image, c'est que vous avez fait une erreur.

```
image *image_new(int h, int w);
```

► **Question 2** Écrire une fonction `image_delete` qui libère toute la mémoire associée à une image.

```
void image_delete(image *im);
```

Dans la suite, on utilisera les fonctions `image_load` et `image_save` disponibles dans le squelette. Ces fonctions permettent de charger et de sauvegarder une image dans un fichier au format png :

```
image *image_load(char *filename);
void image_save(image *im, char *filename);
```

```
...
image *im = image_load("chemin/fichier.png");
// Do some processing on im...
image_write(im, "chemin/nouveau_fichier.png");
```

► **Question 3** Écrire une fonction `invert` qui inverse les niveaux de gris d'une image, le noir devenant blanc et le blanc devenant noir. Cette fonction travaillera en place en modifiant l'image.

```
void invert(image *im);
```

► **Question 4** Écrire une fonction `binarize` qui transforme tout pixel sombre (de valeur strictement inférieure à 128) en pixel noir et tout pixel clair en pixel blanc.

```
void binarize(image *im);
```

► **Question 5** Écrire une fonction `flip_horizontal` qui effectue une symétrie de l'image par rapport à un axe vertical.

```
void flip_horizontal(image *im);
```



2 Détection de bords

Afin de détecter les contours des objets présents dans l'image, pour chaque pixel de coordonnées (i, j) n'étant pas sur le bord de l'image, on définit son énergie par

$$e_{i,j} = \frac{|p_{i,j+1} - p_{i,j-1}|}{2} + \frac{|p_{i+1,j} - p_{i-1,j}|}{2}$$

où $p_{i,j}$ est la valeur du pixel de coordonnées (i, j) . Afin de prendre en compte les cas où l'on se trouve sur les bords de l'image, on définit plus généralement, pour tous i, j vérifiant $0 \leq i < h$ et $0 \leq j < w$:

$$e_{i,j} = \frac{|p_{i,j_r} - p_{i,j_l}|}{j_r - j_l} + \frac{|p_{i_{b,j}} - p_{i_{t,j}}|}{i_b - i_t} \quad \text{avec} \quad j_r = \begin{cases} j+1 & \text{si } j < w-1 \\ j & \text{sinon} \end{cases} \quad j_l = \begin{cases} j-1 & \text{si } j > 0 \\ j & \text{sinon} \end{cases}$$

$$i_b = \begin{cases} i+1 & \text{si } i < h-1 \\ i & \text{sinon} \end{cases} \quad i_t = \begin{cases} i-1 & \text{si } i > 0 \\ i & \text{sinon} \end{cases}$$

Afin de stocker les énergies des pixels de l'image, on définit enfin la structure

```
struct energy {
    double **at;
    int h;
    int w;
};
typedef struct energy energy;
```

Pour alléger le code, on pourra (ce n'est pas indispensable) utiliser par la suite l'*opérateur ternaire* :

```
(cond) ? val1 : val2
```

Il s'agit d'une **expression** (et pas d'une instruction) qui vaut `val1` si `cond` est vraie, `val2` sinon. Essentiellement, c'est la même chose qu'un `if...then...else` en OCaml, mais plus limité (`cond`, `val1` et `val2` doivent être des expressions, et ne peuvent donc pas, par exemple, contenir une boucle `for...`).

```
int min(int x, int y){
    return (x <= y) ? x : y;
}
```

► **Question 6** Écrire une fonction `energy_new` renvoyant un pointeur `e` vers un tableau d'énergie de hauteur `h` et de largeur `w`, alloué sur le tas. Écrire également la fonction `energy_delete` permettant de libérer la mémoire correspondante.

```
energy *energy_new(int h, int w);
void energy_delete(energy *en);
```

► **Question 7** Écrire une fonction `compute_energy` prenant en entrée une image `im` et un tableau d'énergie `e` de même taille et remplissant ce tableau avec les données d'énergie de l'image.

```
void compute_energy(image *im, energy *en);
```

► **Question 8** Écrire une fonction `energy_to_image` prenant en entrée un tableau d'énergie et générant une image de mêmes dimensions, où un pixel d'énergie minimale sera représenté par un pixel noir et un pixel d'énergie maximale par un pixel blanc.

```
image *energy_to_image(energy *en);
```



3 Deux approches naïves

Nous pouvons maintenant nous attaquer à notre problème qui consiste à réduire la largeur de l'image tout en conservant la taille des objets intéressants. Pour cela, nous allons retirer sur chaque ligne un pixel d'énergie minimale.

► **Question 9** Écrire une fonction `remove_pixel` prenant une ligne de pixels `line` de longueur `w` et une ligne d'énergies `e` correspondante et qui élimine un pixel d'énergie minimale, tout en décalant vers la gauche les pixels se trouvant après lui.

```
void remove_pixel(uint8_t *line, double *e, int w);
```

Remarque

Après l'appel, la case d'indice `w - 1` de `line` pourra contenir une valeur quelconque.

► **Question 10** Écrire une fonction `reduce_one_pixel` ayant la spécification suivante :

Entrées : un pointeur `im` vers une structure `image`, un pointeur `en` vers une structure `energy`.

Préconditions : les dimensions du tableau `image` et du tableau `énergie` sont identiques. Les valeurs contenues dans le tableau `énergie` n'ont aucune importance.

Post-conditions :

- un pixel (d'énergie minimale) a été éliminé de chaque ligne de l'image ;
- les valeurs de `en->w` et `im->w` ont été décrémentées.

```
void reduce_one_pixel(image *im, energy *en);
```

► **Question 11** Écrire une fonction `reduce_pixels` qui retire à chaque ligne de l'image le nombre de pixels spécifié en entrée en itérant la fonction précédente. Tester cet algorithme sur les différentes images qui vous ont été fournies. Qu'en pensez-vous ?

```
void reduce_pixels(image *im, int n);
```

Pour remédier à ce problème, nous allons enlever uniquement des pixels situés sur la même colonne.

► **Question 12** Écrire la fonction `best_column` qui prend un tableau d'énergie et qui renvoie l'indice de la colonne dont la somme des énergies est minimale.

```
int best_column(energy *en);
```

► **Question 13** Écrire la fonction `reduce_one_column` qui calcule l'énergie de chaque pixel puis réduit l'image en lui enlevant la colonne d'énergie minimale. Le tableau `e` devra faire la même taille que l'image `im` et les valeurs qu'il contient initialement devront être ignorées. On veillera à diminuer `im->w` ainsi que `e->w` de 1.

```
void reduce_one_column(image *im, energy *en);
```

► **Question 14** Écrire la fonction `reduce_columns` qui itère la fonction précédente pour retirer `n` colonnes à l'image `im`. Testez cet algorithme sur les différentes images qui vous ont été fournies. Qu'en pensez-vous ?

```
void reduce_columns(image *im, int n);
```

4 Seam carving

L'idée de l'algorithme de *seam carving* est d'assouplir un peu la contrainte de réduction colonne par colonne. Pour cela, on définit un *chemin de pixels* comme une suite de pixels connectés soit verticalement soit en diagonale, contenant exactement un pixel de chaque ligne de l'image et commençant sur la ligne du haut. L'énergie d'un chemin est défini comme la somme des énergies des pixels le constituant. Par exemple, voici un chemin d'énergie 6 pour une image de 4 pixels par 4 pixels.

1	1	0	3
4	1	2	4
1	2	2	1
4	1	1	0

Afin de réduire l'image d'un pixel, on souhaite trouver puis enlever un chemin d'énergie minimale. Pour se faire, on définit un *chemin partiel* comme un chemin, sans la contrainte qu'il atteigne le bas de l'image. Afin de trouver un chemin d'énergie minimale, on va calculer pour chaque pixel, l'énergie minimale d'un chemin partiel terminant sur ce pixel. Par exemple, pour notre image de 4 pixels par 4 pixels dont le tableau des énergies a été donné plus haut, on obtient le tableau suivant.

1	1	0	3
5	1	2	4
2	3	3	3
6	3	4	3

► **Question 15** Calculer à la main, le tableau des énergies minimales des chemins partiels pour le tableau d'énergie suivant.

2	1	1	0
3	3	2	2
2	0	1	2

► **Question 16** Écrire une fonction `energy_min_path` qui prend en entrée un tableau d'énergie et le transforme en un tableau des énergies minimales des chemins partiels.

```
void energy_min_path(energy *en);
```

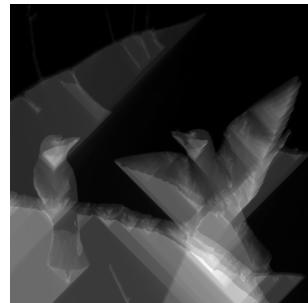


FIGURE XXIX.2 – Représentation graphique de l'énergie des chemins partiels.

On définit la structure suivante pour stocker un chemin de l'image :

```
struct path {  
    int *at;  
    int size;  
};  
typedef struct path path;
```

Une telle structure aura une taille de la hauteur de l'image et `p->at[i]` désignera la colonne par laquelle le chemin passe à la ligne `i`.

► **Question 17** Écrire une fonction `path_new` renvoyant un pointeur vers une nouvelle structure `path`, contenant un tableau de la taille spécifiée, ainsi qu'une fonction `path_delete` libérant la mémoire associée à un `path`.

```
path *path_new(int n);  
void path_delete(path *p);
```

► **Question 18** Écrire la fonction `compute_min_path` prenant en entrée un tableau des énergies minimales des chemins partiels et un chemin `p` dont la taille correspond à la hauteur du tableau des énergies, et remplissant `p` avec le chemin d'énergie minimale.

```
void compute_min_path(energy *en, path *p);
```

► **Question 19** Écrire enfin la fonction `reduce_seam_carving` enlevant successivement `n` chemins d'énergie minimale dans l'image `im`. Testez votre fonction sur les différentes images fournies.

```
void reduce_seam_carving(image *im, int n);
```

L'algorithme présenté ici a été inventé en 2007 par Shai Avidan et Ariel Shamir. Ce sujet est basé sur le travail de Mickaël Péchaud (Lycée Joffre), adapté en C par François Fayard (Les Lazaristes).

Solutions

- Question 1 Attention à bien allouer h lignes!

```
image *image_new(int h, int w) {
    image *im = malloc(sizeof(image));
    im->at = malloc(h * sizeof(int8_t *));
    for (int i = 0; i < h; i++) {
        im->at[i] = malloc(w * sizeof(int8_t));
    }
    im->h = h;
    im->w = w;
    return im;
}
```

- Question 2 Le nombre d'appels à free doit correspondre au nombre d'appels à malloc.

```
void image_delete(image *im) {
    for (int i = 0; i < im->h; i++) {
        free(im->at[i]);
    }
    free(im->at);
    free(im);
}
```

- Question 3

```
void invert(image *im) {
    for (int i = 0; i < im->h; i++) {
        for (int j = 0; j < im->w; j++) {
            im->at[i][j] = 255 - im->at[i][j];
        }
    }
}
```

- Question 4

```
void binarize(image *im) {
    for (int i = 0; i < im->h; i++) {
        for (int j = 0; j < im->w; j++) {
            im->at[i][j] = im->at[i][j] < 128 ? 0 : 255;
        }
    }
}
```

- Question 5 L'un des rares cas où je vous conseille une boucle while même si un for est possible :

```

void flip_horizontal(image *im) {
    for (int i = 0; i < im->h; i++) {
        int jl = 0;
        int jr = im->w - 1;
        while (jl < jr){
            uint8_t value = im->at[i][jl];
            im->at[i][jl] = im->at[i][jr];
            im->at[i][jr] = value;
            jl++;
            jr--;
        }
    }
}

```

► Question 6

```

energy *energy_new(int h, int w) {
    energy *e = malloc(sizeof(energy));
    e->at = malloc(h * sizeof(double *));
    for (int i = 0; i < h; i++) {
        e->at[i] = malloc(w * sizeof(double));
    }
    e->h = h;
    e->w = w;
    return e;
}

void energy_delete(energy *e) {
    for (int i = 0; i < e->h; i++) {
        free(e->at[i]);
    }
    free(e->at);
    free(e);
}

```

► Question 7

```

void compute_energy(image *im, energy *e) {
    for (int i = 0; i < im->h; i++) {
        int it = (i > 0) ? i - 1 : i;
        int ib = (i < im->h - 1) ? i + 1 : i;
        for (int j = 0; j < im->w; j++) {
            int jl = (j > 0) ? j - 1 : j;
            int jr = (j < im->w - 1) ? j + 1 : j;
            double delta_i = (double)(im->at[it][j]) - (double)(im->at[ib][j]);
            double delta_j = (double)(im->at[i][jr]) - (double)(im->at[i][jl]);
            e->at[i][j] = fabs(delta_i) / (ib - it) + fabs(delta_j) / (jr - jl);
        }
    }
}

```

► Question 8 Il faut commencer par déterminer les valeurs extrêmes de l'énergie, puis effectuer une transformation affine qui envoie le minimum sur zéro et le maximum sur 255.

```

image *energy_to_image(energy *e) {
    int h = e->h;
    int w = e->w;
    double v_min = e->at[0][0];
    double v_max = e->at[0][0];
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            if (e->at[i][j] < v_min) {
                v_min = e->at[i][j];
            }
            if (e->at[i][j] > v_max) {
                v_max = e->at[i][j];
            }
        }
    }
    image *im = image_new(h, w);
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            double v = e->at[i][j];
            im->at[i][j] = (uint8_t) (255 * (v - v_min) / (v_max - v_min));
        }
    }
    return im;
}

```

► **Question 9** On fait exactement ce que nous demande l'énoncé (en particulier, on ne modifie donc pas le tableau d'énergie).

```

void remove_pixel(uint8_t *line, double *e, int w) {
    double e_min = e[0];
    double j_min = 0;
    for (int j = 1; j < w; j++) {
        if (e[j] < e_min) {
            e_min = e[j];
            j_min = j;
        }
    }
    for (int j = j_min; j < w - 1; j++) {
        line[j] = line[j + 1];
    }
}

```

► **Question 10**

```

void reduce_one_pixel(image *im, energy *e) {
    compute_energy(im, e);
    for (int i = 0; i < im->h; i++) {
        remove_pixel(im->at[i], e->at[i], im->w);
    }
    im->w--;
    e->w--;
}

```

- Question II On alloue un seul tableau d'énergie, que l'on réutilise pour toutes les étapes :

```
void reduce_pixels(image *im, int n) {
    energy *e = energy_new(im->h, im->w);
    for (int k = 0; k < n; k++) {
        reduce_one_pixel(im, e);
    }
    energy_delete(e);
}
```

Le résultat est catastrophique. Voilà ce qu'on obtient en réduisant de 100 pixels :



- Question I2 Pas de difficulté particulière :

```
int best_column(energy *e) {
    double e_min;
    double j_min = -1;
    for (int j = 0; j < e->w; j++) {
        double e_col = 0.0;
        for (int i = 0; i < e->h; i++) {
            e_col += e->at[i][j];
        }
        if (j_min == -1 || e_col < e_min) {
            e_min = e_col;
            j_min = j;
        }
    }
    return j_min;
}
```

- Question I3

```
void reduce_one_column(image *im, energy *e) {
    compute_energy(im, e);
    int j_min = best_column(e);
    for (int i = 0; i < im->h; i++) {
        for (int j = j_min; j < im->w - 1; ++j) {
            im->at[i][j] = im->at[i][j + 1];
        }
    }
    im->w--;
    e->w--;
}
```

► Question 14

```
void reduce_columns(image *im, int n) {
    energy *e = energy_new(im->h, im->w);
    for (int k = 0; k < n; k++) {
        reduce_one_column(im, e);
    }
    energy_delete(e);
}
```

Le résultat est nettement meilleur, mais l'on introduit quand même des discontinuités visibles (dans les branches entre les deux oiseaux) :



► Question 15 On obtient :

2	1	1	0
4	4	2	2
6	2	3	4

► Question 16

```
void energy_min_path(energy *e) {
    for (int i = 1; i < e->h; i++) {
        for (int j = 0; j < e->w; j++) {
            int jl = j > 0 ? j - 1 : j;
            int jr = j < e->w - 1 ? j + 1 : j;
            double e_min = e->at[i - 1][jl];
            for (int jj = jl + 1; jj <= jr; jj++) {
                if (e->at[i - 1][jj] < e_min) {
                    e_min = e->at[i - 1][jj];
                }
            }
            e->at[i][j] += e_min;
        }
    }
}
```

► Question 17

```

path *path_new(int n) {
    path *p = malloc(sizeof(path));
    p->at = malloc(n * sizeof(int));
    p->size = n;
    return p;
}

void path_delete(path *p) {
    free(p->at);
    free(p);
}

```

- **Question 18** On commence par déterminer l'extrémité inférieure du chemin, puis l'on remonte, en sachant que l'une des trois (ou deux) cases situées juste au-dessus du point actuel permet d'étendre le chemin.

```

void compute_min_path(energy *e, path *p) {
    int h = e->h;
    double e_min = e->at[h - 1][0];
    int j_min = 0;
    for (int j = 1; j < e->w; j++) {
        if (e->at[h - 1][j] < e_min) {
            e_min = e->at[h - 1][j];
            j_min = j;
        }
    }
    p->at[h - 1] = j_min;
    int j = j_min;
    for (int i = h - 2; i >= 0; i--) {
        int jl = j > 0 ? j - 1 : j;
        int jr = j < e->w - 1 ? j + 1 : j;
        e_min = e->at[i][jl];
        j_min = jl;
        for (int jj = jl + 1; jj <= jr; jj++) {
            if (e->at[i][jj] < e_min) {
                e_min = e->at[i][jj];
                j_min = jj;
            }
        }
        p->at[i] = j_min;
        j = j_min;
    }
}

```

- **Question 19** Le plus gros du travail a été fait :

```

void reduce_seam_carving(image *im, int n) {
    energy *e = energy_new(im->h, im->w);
    path *p = path_new(im->h);
    for (int k = 0; k < n; k++) {
        compute_energy(im, e);
        energy_min_path(e);
        compute_min_path(e, p);
        for (int i = 0; i < im->h; i++) {
            for (int j = p->at[i]; j < im->w - 1; ++j) {
                im->at[i][j] = im->at[i][j + 1];
            }
        }
        im->w--;
        e->w--;
    }
    energy_delete(e);
    path_delete(p);
}

```

Le résultat est de très bonne qualité :



ARBRES DE BRAUN

I Définition et propriétés élémentaires

Définition XXX.I – Arbres de Braun

On définit la taille $|t|$ d'un arbre binaire t comme son nombre de nœuds non vides. Un *arbre de Braun* est alors :

- soit l'arbre vide \perp ;
- soit de la forme $N(x, g, d)$, où g et d sont des arbres de Braun tels que $|d| \leq |g| \leq |d| + 1$.

► **Question 1** Montrer que, en ignorant les étiquettes, il existe un unique arbre de Braun de taille n pour chaque entier $n \geq 0$.

► **Question 2** Dessiner la forme des arbres de Braun de taille 1 à 6.

► **Question 3** On définit la hauteur d'un arbre binaire de la manière usuelle (avec $h(\perp) = -1$). Montrer que si t est un arbre de Braun de taille $n \geq 1$, alors $h(t) = \lfloor \log_2 n \rfloor$.

Pour la programmation, on choisit de se limiter aux arbres de Braun à étiquettes entières (cela simplifiera légèrement l'écriture de certaines fonctions). On utilise donc le type suivant :

```
type braun = E | N of int * braun * braun
```

► **Question 4** Écrire une fonction `height` calculant la hauteur d'un arbre de Braun en temps logarithmique en la taille de l'arbre.

```
height : braun -> int
```

2 Calcul de la taille

L'objectif de cette partie est d'écrire une fonction permettant de calculer la taille n d'un arbre de Braun en temps $O(n)$.

► **Question 5** Si l'on sait que t est un arbre de Braun vérifiant $2n \leq |t| \leq 2n + 1$ (avec $n \geq 1$), quelles sont les tailles possibles pour son sous-arbre gauche et pour son sous-arbre droit ?

► **Question 6** Même question si t vérifie $2n + 1 \leq |t| \leq 2n + 2$.

► **Question 7** En déduire une fonction `diff` ayant la spécification suivante :

Entrées : un arbre de Braun t et un entier n .

Précondition : $n \leq |t| \leq n + 1$.

S sortie : $|t| - n$ (0 ou 1 suivant les cas, donc).

Cette fonction devra avoir une complexité en $O(h)$.

```
diff : braun -> int -> int
```

► **Question 8** Écrire à présent une fonction `size` calculant de manière efficace la taille d'un arbre de Braun.

► **Question 9** Déterminer la complexité de la fonction `size`.

3 Réalisation d'un tas fonctionnel par un arbre de Braun

On appelle *tas de Braun* un arbre de Braun vérifiant la condition d'ordre des tas (l'étiquette d'un nœud est toujours inférieure ou égale à celles de ses fils éventuels). On souhaite programmer, de manière efficace, les trois opérations élémentaires sur les tas :

```
get_min : braun -> int
insert : braun -> int -> braun
extract_min : braun -> int * braun
```

L'appel `extract_min t` renverra un couple (m, t') où m est le minimum de t et t' est un tas de Braun contenant les mêmes étiquettes que t moins (une occurrence de) m .

► **Question 10** Écrire la fonction `get_min`. On renverra `max_int` si jamais l'arbre est vide (ce sera pratique par la suite).

► **Question 11** Écrire la fonction `insert`. On exige une complexité logarithmique en la taille de l'arbre. *Attention, il faut bien sûr que l'arbre renvoyé soit un tas de Braun (et donc en particulier un arbre de Braun).*

► **Question 12** Supposons que l'on ait écrit une fonction `merge` : `braun -> braun -> braun` ayant le comportement suivant :

- elle prend en entrée deux tas de Braun t et t' vérifiant $|t'| \leq |t| \leq |t'| + 1$;
- elle renvoie un tas de Braun contenant les éléments de t plus ceux de t' .

Écrire alors une fonction `extract_min` ayant la spécification donnée plus haut.

Il nous reste à écrire cette fonction `merge`. On peut commencer par traiter les cas simples :

► **Question 13** Que doit valoir `merge l r` si r est vide ? si $\min l \leq \min r$?

Le cas délicat est donc celui où la racine de r est strictement plus petite que celle de r : on voudrait prendre la racine de r comme racine « globale », mais on ne peut pas diminuer le nombre d'éléments dans r , puisqu'on violerait alors la condition d'équilibre des arbres de Braun.

► **Question 14** Écrire une fonction `extract_element` qui prend en entrée un tas de Braun non vide t et renvoie un couple (x, t') tel que :

- x est un élément (quelconque) de t ;
- t' est un tas de Braun contenant les éléments de t moins (une occurrence de) x .

```
extract_element : braun -> int * braun
```

► **Question 15** Écrire une fonction `replace_min` tel que l'appel `replace_min t x` renvoie un tas de Braun t' contenant les mêmes éléments que t , moins une occurrence du minimum de t , plus une occurrence de x .

```
replace_min : braun -> int -> braun
```

► **Question 16** Écrire à présent la fonction `merge`.

► **Question 17** Déterminer la complexité de `extract_min`.

Solutions

► **Question 1** Par récurrence forte sur n :

- pour $n = 0$, seul l'arbre vide convient;
- si $n = 2k + 1$, un arbre de Braun est forcément de la forme (g, d) avec $|g| = |d| = k$. Par hypothèse de récurrence, il existe exactement un arbre de Braun non étiqueté B_k de taille k , et (B_k, B_k) est donc l'unique arbre de Braun de taille $2k + 1$;
- si $n = 2k + 2$, on a nécessairement $|g| = k + 1$ et $|d| = k$. On a alors de même (B_{k+1}, B_k) comme unique arbre de Braun de taille $2k + 2$.

► **Question 2**

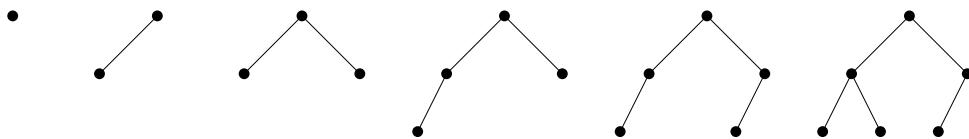


FIGURE XXX.1 – Arbres de Braun à 1, 2, ..., 6 nœuds.

► **Question 3** On procède par récurrence forte sur n .

- Pour $n = 1$, on a bien $h(B_1) = 0 = \lfloor \log_2 1 \rfloor$.
- Si $n = 2k + 1$ avec $k \geq 1$ alors $B_n = (B_k, B_k)$, donc $h(B_n) = 1 + h(B_k) = 1 + \lfloor \log_2 k \rfloor$ par hypothèse de récurrence. Or $1 + \lfloor \log_2 k \rfloor = \lfloor 1 + \log_2 k \rfloor = \lfloor \log_2(2k) \rfloor$. Il reste à remarquer que l'on a nécessairement $\lfloor \log_2(2k) \rfloor = \lfloor \log_2(2k + 1) \rfloor$, puisque $2k + 1$ ne peut pas être une puissance de 2. Finalement, on a donc bien $h(B_{2k+2}) = \lfloor \log_2(2k + 1) \rfloor$.
- Si $n = 2k + 2$ avec $k \geq 1$, alors $B_{2k+2} = (B_{k+1}, B_k)$, donc $h(B_{2k+2}) = 1 + \max(h(B_{k+1}), h(B_k)) = 1 + \lfloor \log_2(k + 1) \rfloor = \lfloor \log_2(2k + 2) \rfloor$, ce qui achève la preuve.

► **Question 4** La hauteur d'un arbre de Braun est une fonction croissante de sa taille (d'après la question précédente), et le sous-arbre gauche a toujours une taille supérieure ou égale à celle du sous-arbre droit. On en déduit la fonction suivante :

```
let rec height = function
| E -> -1
| N (_, l, _) -> 1 + height l
```

Il est immédiat que la complexité est en $O(h(t))$, et donc en $O(\log |t|)$ d'après la question précédente.

► **Question 5**

Si l'on sait que t est un arbre de Braun vérifiant $2n \leq |t| \leq 2n + 1$ (avec $n \geq 1$), quelles sont les tailles possibles pour son sous-arbre gauche et pour son sous-arbre droit? Notons $t = (g, d)$ (t ne peut être vide puisque $n \geq 1$).

- Si $|t| = 2n$, alors $|g| = n$ et $|d| = n - 1$.
- Si $|t| = 2n + 1$, alors $|g| = n$ et $|d| = n$.

On a donc $|g| = n$ dans tous les cas, et $|d| \in \{n - 1, n\}$.

► **Question 6** Cette fois, on a $|g| \in \{n, n + 1\}$ et $|d| = n$.

► **Question 7** D'après les deux questions précédentes, la taille de l'un des sous-arbres est directement connue (le gauche ou le droit suivant la parité de n), et celle de l'autre est connue à un près et peut donc être calculée par un appel récursif. On obtient le code suivant :

```
let rec diff t n =
  match t, n with
  | E, 0 -> 0
  | N (_, E, E), 0 -> 1
  | N (_, l, r), _ ->
    if n mod 2 = 0 then diff r (n / 2 - 1)
    else diff l (n / 2)
  | _ -> failwith "impossible"
```

On effectue un travail constant à chaque niveau de l'arbre, la complexité est bien en $O(h) = O(\log n)$.

► **Question 8** On commence par calculer la taille n du sous-arbre droit par un appel récursif. On sait que celle du sous-arbre gauche vaut n ou $n + 1$, on peut donc utiliser `diff` pour terminer le calcul.

```
let rec size t =
  match t with
  | E -> 0
  | N (_, l, r) ->
    let n = size r in
    2 * n + 1 + diff l n
```

► **Question 9** Notons $\varphi(h)$ le nombre d'opérations élémentaires (à une constante multiplicative près) effectuées par `size` sur un arbre de hauteur h . On a $\varphi(h+1) = 1 + \varphi(h) + \psi(h)$ où $\psi(h)$ est le nombre d'opérations faites par `diff` sur un arbre de hauteur h . Or $\psi(h) = h$, donc $\varphi(h+1) = h + 1 + \varphi(h)$. En sommant la relation $\varphi(h+1) - \varphi(h) = h + 1$, on obtient $\varphi(h) - \varphi(0) = \frac{h(h+1)}{2}$, et la complexité est donc en $O(h^2) = O((\log n)^2)$.

► **Question 10** C'est immédiat :

```
let get_min = function
  | E -> max_int
  | N (x, _, _) -> x
```

► **Question 11** Dans le cas intéressant, on a $t = (y, g, d)$. Pour respecter la contrainte d'ordre des tas, il faudra prendre comme racine $m = \min(x, y)$ et insérer $M = \max(x, y)$ dans l'un des deux sous-arbres. Pour respecter la contrainte de structure des arbres de Braun, on choisit d'insérer dans d et d'inverser d et g . On a alors $t' = (m, insere(d, M), g)$, et $|insere(d, M)| = 1 + |d| \in \{|g|, |g| + 1\}$, ce qui est exactement ce que l'on veut.

```
let rec insert t x =
  match t with
  | E -> N (x, E, E)
  | N (y, l, r) ->
    if x < y then N (x, insert r y, l)
    else N (y, insert r x, l)
```

► **Question 12** Il suffit de faire :

```
let extract_min t =
  match t with
  | E -> failwith "empty queue"
  | N (x, l, r) -> x, merge l r
```

► **Question 13** Si r est vide, on renvoie bien sûr l . Si $\min l \leq \min r$ et que l n'est pas vide ($l = (x, ll, lr)$), alors la racine de $\text{merge}(l, r)$ sera x . On peut fusionner récursivement ll et lr et obtenir ainsi l' tel que $|l'| = |l| - 1$. De manière symétrique à insert , on peut alors conserver la structure d'arbre de Braun en faisant de l' le fils *droit* du nouvel arbre. Autrement dit, on a alors $\text{merge}(l, r) = (x, r, \text{merge}(ll, lr))$.

► **Question 14** On extrait un élément à gauche et l'on inverse fils gauche et fils droit.

```
let rec extract_element t =
  match t with
  | E -> failwith "extract_element from empty queue"
  | N (x, E, E) -> x, E
  | N (x, l, r) ->
    let y, l' = extract_element l in
    y, N (x, r, l')
```

► **Question 15** Pas de difficulté majeure ici, le nombre d'éléments de l'arbre ne change pas donc les contraintes de structure sont automatiquement respectées.

```
let rec replace_min t x =
  match t with
  | E -> failwith "nope"
  | N (y, l, r) ->
    if x <= get_min l && x <= get_min r then N (x, l, r)
    else if get_min l <= get_min r then
      N (get_min l, replace_min l x, r)
    else
      N (get_min r, l, replace_min r x)
```

► **Question 16** Tout le travail a été fait :

```
let rec merge l r =
  match l, r with
  | _, E -> l
  | N (lx, ll, lr), N (rx, rl, rr) ->
    if lx <= rx then
      N (lx, r, merge ll lr)
    else
      let x, l' = extract_element l in
      N (rx, replace_min r x, l')
  | _ -> failwith "merge"
```

► **Question 17** La complexité de extract_min est évidemment la même que celle de merge , et les complexités de extract_element et replace_min sont clairement en $O(h)$. Un appel à merge descend le long d'une branche de l'arbre en faisant un temps constant à chaque noeud (pour un coût $O(h)$ au total pour cette phase), jusqu'à arriver dans le cas $lx > lr$. À ce moment, on fait un appel à extract_min et un à replace_min , pour un coût $O(h)$ (et l'on arrête les appels récursifs à merge). Au total, on a donc du $O(h) = O(\log n)$.

ADRESSAGE OUVERT

Le but de ce sujet est de construire une réalisation efficace de la structure de donnée impérative SET. Elle permet de manipuler des ensembles de valeurs de type T. La signature souhaitée est donnée ci-dessous :

```
set *set_new(void);
bool set_is_member(set *s, T x);
void set_add(set *s, T x);
void set_remove(set *s, T x);
void set_delete(set *s);
```

Ces fonctions nous permettent de créer un ensemble vide, de savoir si un élément x fait partie de l'ensemble s, d'ajouter ou d'enlever un élément à notre ensemble et enfin de libérer la mémoire utilisée par s. On peut imaginer l'utilisation d'une telle structure pour gérer l'ensemble des adresses IP bannies d'un réseau pour des raisons de sécurité. Les adresses IP étant codées sur 32 bits, dans la suite de ce TP, nous utiliserons T = **uint32_t**, le symbole T étant simplement utilisé dans l'énoncé comme un raccourci.

Nous allons réaliser cette signature en utilisant une *table de hachage en adressage ouvert*. Cette structure fonctionne à l'aide d'un tableau de taille 2^p (où $p \in [1 \dots 63]$ est amené à évoluer au cours de la durée de vie de la structure) ainsi qu'une fonction de prototype

```
uint64_t hash(T x, int p);
```

appelée *fonction de hachage*. À tout élément x de type T et tout entier $p \in \mathbb{N}$, elle associe un indice de tableau $i \in [0 \dots 2^p]$ dans lequel nous souhaitons placer l'élément x. La fonction de hachage le plus simple à notre disposition est définie par

$$\forall x \in \mathbb{Z}, \text{hash}_p(x) = x \bmod 2^p.$$

C'est celle que nous utiliserons dans la première partie de ce TP. Si $p = 2$, en partant d'une table de hachage vide, l'ajout des valeurs $x = 1492$ et $x = 1515$ dont les hachages respectifs sont $\text{hash}_2(1492) = 0$ et $\text{hash}_2(1515) = 3$, aboutira au tableau suivant :

1492			1515
0	1	2	3

En suivant, cette stratégie, il est alors facile de voir que 1515 est présent dans notre tableau : il suffit de calculer son hachage $\text{hash}_2(1515) = 3$ et de constater que l'élément 1515 est bien dans la case d'indice 3.

Le rôle d'une bonne fonction de hachage est de répartir le plus uniformément possible les éléments de type T dans les différentes cases du tableau. Malheureusement, il est possible que des valeurs x et y soient différentes tout en ayant $\text{hash}_p(x) = \text{hash}_p(y)$; on parle alors de *collision*. Imaginons un instant que l'élément x ait déjà été placé dans le tableau à la case $\text{hash}_p(x)$. Il nous est alors impossible de placer y dans la même case. La stratégie d'une table de hachage en « adressage ouvert » consiste à le placer dans une case adjacente en suivant la stratégie décrite dans le paragraphe suivant.

Tout d'abord, afin de savoir si une case du tableau est vide ou occupée, nous allons les marquer d'une couleur. Sur nos schémas, les cases seront par défaut de couleur jaune pour signifier qu'elles sont « libres ». L'ajout et la recherche d'un élément se passent alors de la manière suivante.

Ajout d'un élément Lorsqu'on souhaite ajouter l'élément x dans notre tableau, on commence par calculer son hachage $i = \text{hash}_p(x)$.

- Si la case d'indice i est libre, on y stocke l'élément x et on la colorie en bleu pour signifier qu'elle est « occupée ».
- Si cette case n'est pas libre, nous allons tester successivement les cases d'indices $i + 1 \bmod 2^p$, $i + 2 \bmod 2^p$, $i + 3 \bmod 2^p$, ... jusqu'à trouver une case qui est libre ; on parle de *sondage linéaire*. Dès qu'une telle case est trouvée, on y place l'élément x et on la colorie en bleu pour signifier qu'elle est « occupée ».

Bien entendu, un adressage ouvert suppose que le nombre d'éléments présents dans le tableau est toujours inférieur ou égal à 2^p . Pour simplifier la recherche, on imposera de plus que le tableau contienne toujours au moins une case « libre ». Lorsque l'occupation du tableau deviendra trop grande, il sera nécessaire de le redimensionner ; sa taille sera alors doublée.

Recherche d'un élément Lorsqu'on cherche la présence d'un élément x , il suffit de calculer son hachage $i = \text{hash}_p(x)$ et de chercher, par sondage linéaire, la présence de x à partir de l'indice i .

- Si au cours de la recherche, on trouve une case « libre », c'est que l'élément n'est pas présent.
- Si la recherche passe par une case « occupée » contenant l'élément x , c'est qu'il est présent dans la table.

Afin de mieux comprendre notre stratégie, en partant d'un tableau de taille 4 initialement vide, après les opérations :

- ajout de l'élément 1492 dont le hachage est $\text{hash}_2(1492) = 0$,
- ajout de l'élément 1515 dont le hachage est $\text{hash}_2(1515) = 3$,
- ajout de l'élément 1939 dont le hachage est $\text{hash}_2(1939) = 3$,

voici l'état de notre table de hachage.

1492	1939		1515
0	1	2	3

Les éléments 1492, 1515 sont tout d'abord placés dans les cases vides données par leur hachage. L'élément 1939 a pour hachage 3. Puisque cette case est déjà « occupée », on va sonder les cases suivantes de manière circulaire jusqu'à en trouver une de « libre ». C'est la case d'indice $j = 1$ qui est trouvée.

Pour prendre en compte la couleur de chacune de nos cases, nous allons utiliser un « statut » qui peut prendre deux valeurs :

- **empty** = 0, pour signifier que la case est « libre ».
- **occupied** = 1, pour signifier que la case est « occupée ».

Notre table sera donc formée d'un tableau d'alvéoles (on parle aussi de seaux, ou *bucket* en anglais) chacune composée d'un statut et d'un élément. Nous utiliserons donc les structures suivantes :

```
const uint8_t empty = 0;
const uint8_t occupied = 1;

struct bucket {
    uint8_t status;
    T element;
};

typedef struct bucket bucket;
```

```
struct set {
    int p;
    bucket *a;
    uint64_t nb_empty;
};

typedef struct set set;
```

où $p \in [1 \dots 63]$ et le tableau a est de taille 2^p . L'entier nb_empty contiendra le nombre de cases « libres » du tableau.

I Constructeur, destructeur et recherche d'éléments

- **Question 1** Quelle est l'écriture binaire de l'entier renvoyé par cette fonction ?

```
uint64_t ones(int p){
    return (1ull << p) - 1ull;
}
```

On utilise `1ull` à la place de `1` pour que les calculs intermédiaires s'effectuent sur des entiers 64 bits non signés. Par défaut ces calculs utiliseraient des `int`, ce qui poserait problème.

- **Question 2** Écrire la fonction `hash` implémentant le hachage $\text{hash}_p(x) = x \bmod 2^p$.

```
uint64_t hash(T x, int p);
```

Afin de proposer une implémentation efficace, on utilisera les opérations bit à bit disponibles en C.

- **Question 3** Écrire la fonction `set *set_new(void)` créant une table de hachage pour laquelle $p = 1$, dont toutes les cases sont « libres ». Afin de pouvoir tester plus facilement nos prochaines fonctions, on écrira aussi une fonction `set *set_example(void)` générant artificiellement une table de hachage pour laquelle $p = 2$ et contenant les dates 1492, 1515 et 1939 comme décrit dans l'exemple plus haut.

- **Question 4** Écrire la fonction `void set_delete(set *s)` libérant la mémoire utilisée par `s`.

- **Question 5** Écrire la fonction `bool set_is_member(set *s, T x)` permettant de déterminer si `x` est un élément de `s`. Afin de proposer une implémentation efficace, on utilisera les opérations bit à bit pour les calculs modulo 2^p . On donnera de plus un argument justifiant la terminaison de cette fonction.

2 Parcours de la table

Pour permettre de parcourir les éléments de l'ensemble sans avoir à se préoccuper des détails d'implémentation, nous allons fournir un itérateur qui va prendre successivement les indices des cases « occupées » de notre tableau. Il commencera à l'index i_{begin} qui est égal, soit au plus petit index i tel que $a[i]$ est occupé (si une telle valeur existe), soit à 2^p . Il terminera par la valeur $i_{\text{end}} = 2^p$. Si i est l'indice d'une case occupée, la fonction `set_next(s, i)` renverra, soit l'indice de la prochaine case occupée, si une telle case existe, soit 2^p .

```
uint64_t set_begin(set *s);
uint64_t set_end(set *s);
uint64_t set_next(set *s, uint64_t i);
```

Nous utiliserons aussi la fonction `set_get(s, i)` qui renvoie l'élément contenu dans la case d'index i (i désignant bien évidemment un index de case occupée).

```
T set_get(set *s, uint64_t i);
```

Ces fonctions permettent de parcourir facilement l'ensemble des valeurs de notre table de hachage. Par exemple, la fonction suivante permet de savoir si tous les éléments de la table sont pairs.

```
bool all_even(set *s) {
    for (uint64_t i = set_begin(s); i != set_end(s); i = set_next(s, i)) {
        if (set_get(s, i) % 2 == 1) {
            return false;
        }
    }
    return true;
}
```

- ▶ **Question 6** Écrire la fonction `set_get`.
- ▶ **Question 7** Écrire les fonctions `set_begin`, `set_end` et `set_next`.

3 Ajout d'éléments

Afin de préparer la possibilité d'ajouter des éléments à notre table, nous allons factoriser notre code et écrire une fonction

```
uint64_t set_search(set *s, T x, bool *found);
```

qui renvoie un entier i et qui écrit un booléen dans `*found`. Ces valeurs doivent posséder les caractéristiques suivantes :

- si x est un élément de s , l'entier i renvoyé est l'index de la case contenant x et `*found` est égal à `true`;
- sinon, on calcule l'index i de la case dans laquelle on placerait x si on avait à l'ajouter à s . On renvoie alors i et `*found` est égal à `false`.

- ▶ **Question 8** Écrire `set_search` et réimplémenter `set_is_member` à l'aide de cette nouvelle fonction.

▶ **Question 9** Écrire la fonction `void set_resize(set *s, int p)` prenant en entrée une table de hachage possédant n éléments et « redimensionnant » son tableau en un tableau de taille 2^p . On supposera que $n < 2^p$ et on placera tous les éléments dans ce nouveau tableau en utilisant la fonction de hachage `hash_p` associée à cette nouvelle taille de tableau. On pourra commencer par créer une nouvelle table de hachage, avant de modifier s .

▶ **Question 10** Écrire la fonction `void set_add(set *s, T x)` ajoutant l'élément x à la table s (la fonction n'aura aucun effet si l'élément était déjà présent). Afin de toujours conserver une case « libre » dans notre tableau et de ne pas trop le charger, on décidera de doubler la taille du tableau dès que le nombre de cases « libres » est inférieur au tiers de la taille du tableau.

4 Suppression d'éléments

On souhaite désormais pouvoir supprimer des éléments de notre table de hachage.

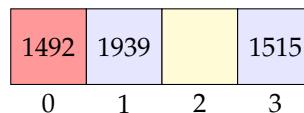
- ▶ **Question 11** Expliquer pourquoi il n'est pas possible de supprimer un élément de la table en changeant simplement le statut de sa case en case « libre ».

Afin de remédier à ce problème, nous allons créer un nouveau statut appelé « pierre tombale » (*tombstone* en anglais). Lorsqu'un élément de la table sera supprimé, le statut de sa case passera de « occupé » à « pierre tombale ». Lors du sondage intervenant dans la recherche d'un élément, il faudra considérer les pierres tombales comme des cases ne contenant aucun élément mais signalant que la recherche doit continuer. Si nous cherchons une case pour y placer un nouvel élément, il faudra déterminer la première case qui est soit libre, soit une pierre tombale. On ajoute donc le statut suivant (que nous représenterons en rouge sur nos schémas) :

```
const uint8_t tombstone = 2;
```

Par exemple, le schéma ci-dessous donne l'état de la structure si l'on part d'un tableau de taille 4 ayant toutes ses cases « libres », et qu'on effectue les opérations suivantes :

- ajout de l'élément 1492 dont le hachage est $\text{hash}_2(1492) = 0$,
- ajout de l'élément 1515 dont le hachage est $\text{hash}_2(1515) = 3$,
- ajout de l'élément 1939 dont le hachage est $\text{hash}_2(1939) = 3$,
- suppression de l'élément 1492.



► **Question 12** Proposer des nouvelles implémentations des fonctions `set_search`, `set_begin`, `set_next` et `set_add` fonctionnant avec les pierres tombales.

► **Question 13** Implémenter la fonction

```
void set_remove(set *s, T x);
```

permettant d'enlever l'élément `x` de la table `s`. Cette fonction n'aura pas d'effet si l'élément n'était pas présent.

5 La liste des adresses IP

Dans un réseau, chaque ordinateur possède une unique adresse nommée *adresse IP*. Le standard IPv4 définit une telle adresse comme un entier non signé 32 bits, généralement représenté sous forme de sa décomposition en base 256 où les « chiffres » sont séparés par des points : $d_3.d_2.d_1.d_0$ où $d_k \in [0 \dots 255]$. Le fichier `ip.txt` contient une liste de 172 754 adresses IP que nous souhaitons charger dans notre table de hachage.

Vous trouverez dans le fichier fourni une fonction

```
T *read_data(char *filename, int *n);
```

Cette fonction a le comportement suivant :

- la fonction renvoie un pointeur vers un bloc alloué sur le tas, contenant les adresses lues dans le fichier ;
- si une erreur s'est produite (si le fichier n'existe pas, par exemple), le pointeur renvoyé sera nul ;
- l'argument `n` est un argument de sortie. Après l'appel, la valeur pointée par `n` sera égale à la taille du tableau renvoyé.

► **Question 14** Écrire une fonction `read_set` qui prend en entrée le nom d'un fichier et renvoie un `set*` dont les éléments sont les adresses présentes dans le fichier.

```
set *read_set(char *filename);
```

► **Question 15** Écrire une fonction

```
void set_skip_stats(set *s, double *average, uint64_t *max);
```

calculant le nombre moyen et le nombre maximum de sondages nécessaires pour trouver une adresse `x` appartenant à `s`. En observant le fichier des adresses IP, expliquer pourquoi ces nombres sont si élevés.

6 Une meilleure fonction de hachage

Afin de résoudre le problème soulevé dans la partie précédente, nous allons implémenter une fonction de hachage plus efficace. Pour cela, on choisit un réel $\varphi \in [0, 1]$ et on définit hash_p par

$$\forall x \in \mathbb{Z}, \text{hash}_p(x) = \lfloor 2^p \{x\varphi\} \rfloor$$

où $\{a\} = a - \lfloor a \rfloor$ désigne la partie fractionnaire de $a \in \mathbb{R}$. Même si cette méthode fonctionne quelle que soit la valeur de φ , on peut montrer que lorsque φ est proche de

$$\frac{\sqrt{5} - 1}{2} \approx 0.618034$$

la fonction hash_p va favoriser la répartition uniforme des valeurs `x` dans notre tableau (voir les théorèmes d'ergodicité sur l'équirépartition modulo 1). Nous utiliserons donc une approximation de $(\sqrt{5} - 1)/2$ de la forme $\varphi = s/2^{64}$ où $s \in \mathbb{N}$.

- **Question 16** Montrer que la fonction

```
uint64_t f(uint64_t x, uint64_t s) {  
    return x * s;  
}
```

calcule l'entier $x_s \in [0 \dots 2^{64}]$ tel que

$$\{x\varphi\} = \frac{x_s}{2^{64}}.$$

- **Question 17** Montrer que la décomposition en base 2 de $\text{hash}_p(x)$ est formée des p bits de poids forts de la décomposition en base 2 de x_s .

- **Question 18** En déduire une implémentation

```
uint64_t hash(uint32_t x, int p)
```

permettant de calculer efficacement $\text{hash}_p(x)$. On utilisera la valeur

$$s = 11\ 400\ 714\ 819\ 323\ 198\ 549$$

qui a été choisie pour être un nombre premier et pour que $s/2^{64}$ soit une bonne approximation de $(\sqrt{5} - 1)/2$.

- **Question 19** Quel est le nombre moyen et le nombre maximum de sondages nécessaires pour trouver une adresse IP présente dans notre base avec cette nouvelle fonction de hachage ?

7 Sondage quadratique

Afin de faire encore baisser le nombre de sondages nécessaires pour trouver un élément dans notre table, nous allons changer la technique de sondage. Au lieu de sonder les cases d'indices $i + 1 \bmod 2^p$, $i + 2 \bmod 2^p$, $i + 3 \bmod 2^p$, ... nous allons sonder les cases d'indices $i + 1 \bmod 2^p$, $i + (1 + 2) \bmod 2^p$, $i + (1 + 2 + 3) \bmod 2^p$, ... afin d'éviter la formation de clusters qui ont tendance à apparaître avec la technique de sondage linéaire. Cette méthode est appelée méthode de sondage quadratique.

- **Question 20** Implémenter cette nouvelle méthode et observer son influence sur les nombres de sondage à effectuer pour notre ensemble d'adresses IP.

- **Question 21** Prouver enfin que cette méthode de sondage est correcte, c'est-à-dire que si il existe une case libre dans notre tableau, le sondage finira par la trouver.

Solutions

► **Question 1** Cette fonction renvoie le nombre $\overline{1\dots 1}^2$ (avec p chiffres 1).

► **Question 2** Le reste de la division modulo 2^p est obtenu en ne conservant que les p bits de poids faible, ce qui peut se faire ainsi :

```
uint64_t hash(uint32_t k, int p) {
    return k & ones(p);
}
```

► **Question 3**

```
set *set_new(void) {
    set *s = malloc(sizeof(set));
    s->p = 1;
    s->a = malloc(2 * sizeof(bucket));
    s->a[0].status = empty;
    s->a[1].status = empty;
    s->nb_empty = 2;
    return s;
}
```

```
set *set_example(void) {
    set *s = malloc(sizeof(set));
    s->p = 2;
    s->a = malloc(4 * sizeof(bucket));
    s->a[0].status = occupied;
    s->a[0].element = 1492;
    s->a[1].status = occupied;
    s->a[1].element = 1939;
    s->a[2].status = empty;
    s->a[3].status = occupied;
    s->a[3].element = 1515;
    s->nb_empty = 1;
    return s;
}
```

► **Question 4**

```
void set_delete(set *s) {
    free(s->a);
    free(s);
}
```

► **Question 5**

```
bool set_is_member(set *s, uint32_t x) {
    uint64_t i = hash(x, s->p);
    while (true) {
        if (s->a[i].status == empty) {
            return false;
        } else if (s->a[i].element == x) {
            return true;
        }
        i += 1;
        i &= ones(p);
    }
}
```

► Question 6

```
uint32_t set_get(set *s, uint64_t i) {
    return s->a[i].element;
}
```

► Question 7 set_end est immédiat :

```
uint64_t set_end(set *s) {
    return lull << s->p;
}
```

Pour set_begin, on parcourt le tableau jusqu'à trouver une case occupée, ou en sortir :

```
uint64_t set_begin(set *s) {
    uint64_t i = 0;
    while (i < set_end(s) && s->a[i].status == empty)
        i++;
    return i;
}
```

set_next est similaire, sauf que l'on commence le parcours à la case $i + 1$.

```
uint64_t set_next(set *s, uint64_t i) {
    i++;
    while (i < set_end(s) && s->a[i].status == empty)
        i++;
    return i;
}
```

► Question 8 On parcourt la table, à partir de l'indice $\text{hash}_p(x)$ jusqu'à tomber soit sur une case vide, soit sur une case contenant x . La fonction set_is_member ne pose ensuite aucune difficulté.

```
uint64_t set_search(set *s, uint32_t x, bool *found) {
    uint64_t i = hash(x, s->p);
    while (true) {
        if (s->a[i].status == empty) {
            *found = false;
            return i;
        } else if (s->a[i].element == x) {
            *found = true;
            return i;
        }
        i += 1;
        i &= ones(s->p);
    }
}

bool set_is_member(set *s, uint32_t x) {
    bool found;
    set_search(s, x, &found);
    return found;
}
```

- **Question 9** Il n'y a rien de compliqué, mais il y a plusieurs étapes et il faut faire attention aux potentielles fuites de mémoire :

```
void set_resize(set *s, int p) {
    // Prepare an empty table of size 2**p
    uint64_t m = 1ull << p;
    set *s_new = malloc(sizeof(set));
    s_new->p = p;
    s_new->a = malloc(m * sizeof(bucket));
    for (uint64_t i = 0; i < m; ++i) {
        s_new->a[i].status = empty;
    }
    s_new->nb_empty = m;
    // Add the elements of s to that table
    for (uint64_t i = set_begin(s); i != set_end(s); i = set_next(s, i)) {
        uint32_t x = set_get(s, i);
        bool found;
        uint64_t j = set_search(s_new, x, &found);
        s_new->a[j].status = occupied;
        s_new->a[j].element = x;
        s_new->nb_empty--;
    }
    free(s->a);
    // The next three lines could be replaced with *s = *s_new;
    s->a = s_new->a;
    s->p = p;
    s->nb_empty = s_new->nb_empty;

    free(s_new);
}
```

- **Question 10** On commence par vérifier si l'élément est déjà présent (rien à faire dans ce cas). Cela permet en même temps de savoir où il faudra insérer l'élément. Ensuite :

- si nécessaire, on redimensionne (sans oublier de recalculer ensuite le point d'insertion, qui aura changé);
- on effectue l'insertion.

```
void set_add(set *s, uint32_t x) {
    bool found;
    uint64_t j = set_search(s, x, &found);
    if (found) return;

    uint64_t m = 1ull << s->p;
    if (s->nb_empty <= 1 || 3 * s->nb_empty <= m) {
        set_resize(s, s->p + 1);
        j = set_search(s, x, &found);
    }

    s->a[j].status = occupied;
    s->a[j].element = x;
    s->nb_empty--;
}
```

► **Question 11** Si l'on supprime un élément en passant le statut de la case à « libre », on va casser les séquences de recherche. Par exemple, supposons qu'on insère successivement x et y dans une table vide, et que ces deux éléments soient en collision ($\text{hash}_p(x) = \text{hash}_p(y) = 0$, par exemple). On va alors placer x dans la case 0 et y dans la case 1. Si l'on supprime x puis que l'on cherche y , on va tomber sur une case vide, et en déduire que y n'est pas présent !

► **Question 12** Remarquons d'abord qu'il n'y a aucune modification à apporter à `set_end`. Pour `set_search`, on commence par calculer $i = \text{hash}_p(x)$ et l'on parcourt le tableau (circulairement) à partir de ce i . Simultanément, on maintient une variable `i_tombstone` qui vaut initialement `set_end(s)` (ce qui n'est jamais un indice valable du tableau). Si l'on rencontre au moins une pierre tombale lors de la recherche, `i_tombstone` deviendra égal à l'indice de la première pierre tombale rencontrée. La recherche se termine dans l'un des deux cas suivants :

- on trouve l'élément, auquel cas il faut l'indiquer dans `found` et renvoyer sa position ;
- on tombe sur une case vide, ce qui signifie que l'élément n'est pas présent. Dans ce cas, on renvoie l'indice de la première pierre tombale rencontrée s'il y en a une, l'indice de la case libre sur laquelle on a terminé sinon.

```
uint64_t set_search(set *s, uint32_t x, bool *found) {
    uint64_t i = hash(x, s->p);
    uint64_t i_tombstone = set_end(s);
    while (true) {
        if (s->a[i].status == empty) {
            *found = false;
            if (i_tombstone == set_end(s)) return i;
            else return i_tombstone;
        } else if (s->a[i].status == tombstone && i_tombstone == set_end(s)) {
            i_tombstone = i;
        } else if (s->a[i].element == x) {
            *found = true;
            return i;
        }
        i += 1;
        i &= ones(s->p);
    }
}
```

Pour `set_next`, c'est un peu plus simple : il suffit de modifier la fonction écrite plus haut pour sauter les cases vides et les pierres tombales.

```
uint64_t set_next(set *s, uint64_t i) {
    i++;
    while (i < set_end(s) && (s->a[i].status == empty || s->a[i].status == tombstone))
        i++;
    return i;
}
```

Pour `set_begin`, le plus simple est d'apporter la même modification. Cependant, il est en fait possible de voir cette fonction comme un cas particulier de `set_next` : en effet, on a `set_begin(s) == set_next(s, -1)`. Noter que le deuxième argument est non signé, donc le -1 est en fait implicitement converti en $2^{64} - 1$: ça ne pose pas de problème, les calculs se font modulo 2^{64} et on a bien $(\text{uint64_t}) - 1 + (\text{uint64_t}) 1 == 0$.

```
uint64_t set_begin(set *s){
    return set_next(s, -1);
}
```

- Question 13 Aucune difficulté si l'on pense à utiliser `set_search` :

```
void set_remove(set *s, uint32_t x) {
    bool found;
    uint64_t i = set_search(s, x, &found);
    if (!found) return;
    s->a[i].status = tombstone;
}
```

- Question 14 La fonction `read_data` fournie, qui utilise certaines choses sur la gestion des fichiers que nous n'avons pas encore vues (et des détails sordides sur les conversions implicites).

```
uint32_t *read_data(char *filename, int *n) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) return NULL;

    int nb_lines = 0;
    char line[16];
    while (!feof(file)) {
        fscanf(file, "%15s\n", line);
        nb_lines++;
    }
    rewind(file);

    uint32_t *t = malloc(nb_lines * sizeof(uint32_t));
    int a, b, c, d;
    for (int i = 0; i < nb_lines; ++i) {
        int nb_read = fscanf(file, "%d.%d.%d.%d", &a, &b, &c, &d);
        if (nb_read != 4){
            fclose(file);
            free(t);
            return NULL;
        }
        t[i] = (((a * 256u) + b) * 256u + c) * 256u + d;
    }

    fclose(file);
    *n = nb_lines;
    return t;
}
```

La fonction que l'on demandait d'écrire, elle, ne pose pas trop de problème. Attention cependant à ne pas oublier de libérer le tableau!

```
set *read_set(char *filename){
    set *s = set_new();
    int n = 0;
    uint32_t *arr = read_data(filename, &n);
    assert(arr != NULL);
    printf("read_set : n = %d\n", n);
    for (int i = 0; i < n; i++){
        set_add(s, arr[i]);
    }
    free(arr);
    return s;
}
```

► Question 15

```

void set_skip_stats(set *s, double *average, uint64_t *max) {
    uint64_t nb_skip_total = 0;
    uint64_t nb_skip_max = 0;
    uint64_t nb_elements = 0;
    for (uint64_t i = set_begin(s); i != set_end(s); i = set_next(s, i)) {
        nb_elements++;
        uint32_t x = set_get(s, i);
        uint64_t j = hash(x, s->p);
        uint64_t nb_skip = (i - j) & ones(s->p);
        nb_skip_total += nb_skip;
        if (nb_skip > nb_skip_max) {
            nb_skip_max = nb_skip;
        }
    }
    *average = ((double)nb_skip_total) / nb_elements;
    *max = nb_skip_max;
}

```

On obtient un nombre moyen d'éléments sautés d'environ 912 et un peu plus de 19 000 comme nombre maximum. C'est catastrophique, mais ce n'est pas surprenant puisque notre fonction de hachage est profondément stupide. En effet, on se contente de prendre x modulo 2^p , ce qui revient très précisément à conserver les p bits de poids faible de x . Ici, une grande partie des adresses (plus de la moitié, à vue d'œil) sont de la forme $a.b.c.0$, et se terminent donc par 8 bits nuls (il s'agit de masques de sous-réseaux). On essaie donc de les envoyer sur 1/256^{ème} des cases de la table, ce qui crée évidemment d'innombrables collisions.

Remarque

Hacher un entier en prenant simplement sa valeur modulo la taille de la table est une technique certes rustique, mais pas complètement déraisonnable. En revanche, il ne faut surtout pas prendre une table de taille 2^p dans ce cas (typiquement, on prend un nombre premier).

► Question 16 Soit $x, s \in \mathbb{N}$. Puisque x et s sont des entiers non signés, $x * s$ est le reste de la division euclidienne de xs par 2^{64} . On effectue donc une telle division euclidienne : il existe $q, r \in \mathbb{N}$ tels que $xs = q2^{64} + r$ et $0 \leq r < 2^{64}$. On a donc :

$$\begin{aligned}
 \{x\varphi\} &= \left\{ \frac{xs}{2^{64}} \right\} \\
 &= \left\{ \frac{q2^{64} + r}{2^{64}} \right\} \\
 &= \left\{ q + \frac{r}{2^{64}} \right\} \\
 &= \frac{r}{2^{64}}
 \end{aligned}
 \quad \text{car } q \in \mathbb{N} \text{ et } 0 \leq r < 2^{64}$$

► **Question 17** On décompose x_s en base 2 : $x_s = \sum_{k=0}^{63} d_k 2^k$. On a alors :

$$\begin{aligned}
 \text{hash}_p(x) &= \lfloor 2^p \{x\varphi\} \rfloor \\
 &= \left\lfloor 2^p \frac{x_s}{2^{64}} \right\rfloor \\
 &= \left\lfloor 2^{p-64} \sum_{k=0}^{63} d_k 2^k \right\rfloor \\
 &= \left\lfloor \sum_{k=0}^{63} d_k 2^{k-(64-p)} \right\rfloor \\
 &= \left\lfloor \sum_{k=0}^{64-(p+1)} d_k 2^{k-(64-p)} + \sum_{k=64-p}^{63} d_k 2^{k-(64-p)} \right\rfloor \\
 &= \sum_{k=64-p}^{63} d_k 2^{k-(64-p)} = \sum_{k=0}^{p-1} d_{64-p+k} 2^k
 \end{aligned}$$

On en déduit que la décomposition en base 2 de $\text{hash}_p(x)$ est formé des p bits de poids fort de la décomposition en base 2 de x_s .

► **Question 18** On obtient :

```

uint64_t hash(uint32_t x, int p) {
    uint64_t s = 11400714819323198549u;
    return (x * s) >> (64 - p);
}

```

En réalité, on pouvait écrire cela directement. En effet, on a vu en cours que si n était un entier positif, alors $n >> k$ renvoyait le quotient de la division euclidienne de n par 2^k , c'est-à-dire $\lfloor n/2^k \rfloor$. Or ici, on a précisément $\text{hash}_p(x) = \lfloor x_s/2^{64-p} \rfloor$.

► **Question 19** Avec cette nouvelle fonction de hachage, le nombre moyen d'étapes de sondage passe à 0.98. Le maximum de sondages passe quant à lui à 56. On a donc une chute drastique de ces valeurs par rapport à la fonction de hachage naïve utilisée dans la première partie.

► **Question 20** Il faut modifier `set_search` et `set_skip_stats`. On ne donne que la nouvelle version de `set_search` :

```

uint64_t set_search(set *s, uint32_t x, bool *found) {
    uint64_t i = hash(x, s->p);
    uint64_t i_tombstone = set_end(s);
    uint64_t i_step = 1;
    while (true) {
        if (s->a[i].status == empty) {
            *found = false;
            return i_tombstone == set_end(s) ? i : i_tombstone;
        } else if (s->a[i].status == tombstone && i_tombstone == set_end(s)) {
            i_tombstone = i;
        } else if (s->a[i].element == x) {
            *found = true;
            return i;
        }
        i += i_step;
        i_step++;
        i = i & ones(s->p);
    }
}

```

Avec cette nouvelle stratégie, le nombre moyen d'étapes de sondage passe à 0.78. Le maximum de sondages passe quant à lui à 22. On a donc une baisse de ces valeurs par rapport à la technique de sondage linéaire.

Remarque

En pratique, les deux méthodes sont utilisées : le sondage quadratique a tendance à donner des séquences de recherche plus courtes (comme c'est le cas ici), mais pas forcément plus rapides (pour des raisons techniques).

► **Question 21** Il reste à montrer que cette stratégie de sondage est correcte, c'est-à-dire que si le tableau possède au moins une case vide, le sondage va la trouver. À l'étape k , le sondage va visiter la case d'indice $i + 1 + \dots + k \bmod 2^p = i + k(k+1)/2 \bmod 2^p$. Nous allons montrer que pour $0 \leq k < 2^p$, ces sondages se font dans des cases deux à deux distinctes. Pour prouver cela, on se donne $k_1, k_2 \in [0 \dots 2^p]$ tels que

$$\frac{k_1(k_1 + 1)}{2} \equiv \frac{k_2(k_2 + 1)}{2} [2^p]$$

et on souhaite montrer que $k_1 \equiv k_2 [2^p]$. On a donc

$$\begin{aligned} k_1(k_1 + 1) &\equiv k_2(k_2 + 1) [2^{p+1}] \\ \text{donc } (k_1 - k_2)(k_1 + k_2 + 1) &\equiv 0 [2^{p+1}] \end{aligned}$$

donc $2^{p+1}|(k_1 - k_2)(k_1 + k_2 + 1)$.

- Supposons que $2|(k_1 - k_2)$. Alors k_1 et k_2 ont même parité, donc $k_1 + k_2 + 1$ est impair, donc $k_1 + k_2 + 1$ est premier avec 2^{p+1} . D'après le lemme de Gauss, on en déduit que $2^{p+1}|k_1 - k_2$, donc $2^p|k_1 - k_2$ donc $k_1 = k_2$.
- Sinon, 2 est premier avec $k_1 - k_2$, donc 2^{p+1} est premier avec $k_1 - k_2$. D'après le lemme de Gauss, on en déduit que 2^{p+1} divise $k_1 + k_2 + 1$. Or $0 \leq k_1 \leq 2^p - 1$ et $0 \leq k_2 \leq 2^p - 1$, donc $1 \leq k_1 + k_2 + 1 \leq 2^{p+1} - 1$. C'est absurde.

Donc $k_1 = k_2$, ce qui montre que les 2^p premiers sondages de la progression quadratique visitent des cases deux à deux distinctes. Comme il y en a 2^p , elles sont toutes visitées.

AUTOUR DU TRI RAPIDE

I Tri rapide en C

Pour passer un sous-tableau à une fonction en C, on peut passer un pointeur vers la première case de la partie qui nous intéresse et la longueur de la partie. Pour obtenir un pointeur vers la partie qui commence à l'indice i , il suffit de prendre `&t[i]` (ce n'est pas idiomatique du tout, mais c'est la manière de procéder en restant dans les bornes du programme).

Exercice XXXII.1

p. 648

- Écrire une fonction `partition` ayant le prototype suivant :

```
int partition(int *arr, int len);
```

Cette fonction partitionnera le tableau passé en argument en utilisant le premier élément comme pivot.

- Écrire une fonction `quicksort` ayant le prototype suivant :

```
void quicksort(int *arr, int len);
```

Remarque

Contrairement à ce que nous avons fait en OCaml, il n'y aura pas de fonction auxiliaire ici.

Exercice XXXII.2

p. 648

- À l'aide des fonctions présentes dans le squelette, écrire une fonction `test_quicksort` vérifiant la correction de `quicksort` sur divers tableaux aléatoires de petite taille (entre 1 et 100 éléments, disons).
- Observer le comportement de `quicksort` sur des tableaux plus grands (dizaines ou centaines de milliers d'éléments), dans le cas d'un tableau aléatoire d'entiers entre 0 et 1 000 000 et dans le cas d'un tableau trié,

2 Quickselect

Le problème de la *sélection* est le suivant :

- on nous donne un tableau t de n éléments (d'un type totalement ordonné, des entiers par exemple), et un entier k vérifiant $0 \leq k < n$;
- on doit renvoyer l'élément de t qui serait à l'indice k si t était trié.

Exercice XXXII.3

p. 649

- Que doit renvoyer `select(t, 0)`? `select(t, n - 1)`?
- Comment peut-on calculer une médiane d'un tableau si l'on dispose d'une fonction `select`?

Remarque

Une médiane d'un tableau de taille n est un élément x de ce tableau tel que au moins $n/2$ éléments de t soient inférieurs ou égaux à x et au moins $n/2$ soient supérieurs ou égaux.

Exercice XXXII.4

p. 650

- Proposer une méthode permettant de réaliser la sélection en temps $O(n \log n)$.

Pour faire mieux, on peut adapter l'algorithme du tri rapide : c'est l'algorithme *quickselect*. L'idée est de réutiliser la fonction *partition* écrite précédemment (sans la modifier), mais de ne pas trier entièrement le tableau : un seul appel récursif est nécessaire à chaque étape.

- Écrire une fonction *quickselect_aux* de prototype

```
int quickselect_aux(int *arr, int k, int len);
```

Cette fonction renverra le $k - 1$ -ème plus petit élément du tableau (c'est-à-dire *select(arr, k)*) et pourra avoir un effet secondaire quelconque sur le contenu de ce tableau.

- Écrire une fonction *quickselect* ayant le même prototype que *quickselect_aux* et la même valeur de retour, mais ne modifiant pas le tableau qu'on lui passe en paramètre.
- Analyser la complexité temporelle de *quickselect* :
 - dans le pire cas ;
 - dans le cas où le pivot choisi partage équitablement le tableau à chaque étape.

3 Introsort**Exercice XXXII.5**

p. 651

Écrire une fonction *heapsort* ayant le prototype suivant :

```
void heapsort(int *arr, int len);
```

Cette fonction triera le tableau *arr* par ordre croissant en utilisant l'algorithme du tri par tas. On réfléchira aux fonctions auxiliaires utiles (on n'a pas besoin de toutes les fonctions sur les tas).

Exercice XXXII.6

p. 651

- Écrire une fonction *ilog* calculant la partie entière du logarithme en base 2 de son argument (que l'on pourra supposer strictement positif).

```
int ilog(int n);
```

- Dans le cas où le pivot partage équitablement le tableau à chaque étape, quelle est la profondeur de récursion maximale de *quicksort* ?
- Écrire une fonction *introsort* de prototype

```
void introsort(int *arr, int len);
```

Cette fonction triera le tableau passé en argument, en place, en commençant par un tri rapide mais en basculant vers un tri fusion dès que la profondeur de récursion est deux fois supérieure à ce que l'on pourrait attendre dans un « bon » cas.

Remarque

Une fonction auxiliaire sera nécessaire.

Solutions

Correction de l'exercice XXXII.1 page 646

- On procède comme dans le cours (schéma de Lomuto), sauf qu'il est inutile ici de déplacer le pivot (puisque'on utilise le premier élément). La toute fin diffère également un peu : le pivot doit être échangé avec le dernier éléments de la zone « petits » et pas avec le premier de la zone « grands ».

```
int partition(int *t, int len){  
    int vpiv = t[0];  
    int i = 1;  
    for (int j = 1; j < len; j++){  
        if (t[j] <= vpiv){  
            swap(t, i, j);  
            i++;  
        }  
    }  
    swap(t, 0, i - 1);  
    return i - 1;  
}
```

- C'est une traduction directe du cours, en utilisant la remarque sur la possibilité de passer un pointeur vers autre chose que le début du tableau :

```
void quicksort(int *t, int n){  
    if (n <= 1) return;  
    int k = partition(t, n);  
    quicksort(&t[k + 1], n - k - 1);  
    quicksort(t, k);  
}
```

Correction de l'exercice XXXII.2 page 646

- On peut par exemple écrire cela :

```
void test_quicksort(int maxlen, int iterations, int bound){  
    for (int len = 1; len <= maxlen; len++){  
        for (int i = 0; i < iterations; i++) {  
            int *arr = random_array(len, bound);  
            int *arr_copy = copy(arr, len);  
            insertion_sort(arr, len);  
            quicksort(arr_copy, len);  
            assert(is_equal(arr, arr_copy, len));  
            free(arr);  
            free(arr_copy);  
        }  
    }  
}
```

Choisir une valeur de bound pas trop grande permet de s'assurer que l'on sera bien confronté à des tableaux avec valeurs répétées.

2. On écrit un petit programme qui prend une taille en ligne de commande et mesure le temps pour trier un tableau d'entiers aléatoires et un tableau déjà trié de cette taille :

```
int main(int argc, char* argv[]){
    assert(argc == 2);
    int len = atoi(argv[1]);
    int bound = 1000 * 1000;
    int *random = random_array(len, bound);
    int *sorted = malloc(len * sizeof(int));
    for (int i = 0; i < len; i++){
        sorted[i] = i;
    }
    clock_t t0 = clock();
    quicksort(random, len);
    clock_t t1 = clock();
    quicksort(sorted, len);
    clock_t t2 = clock();
    double time_random = 1.0 * (t1 - t0) / CLOCKS_PER_SEC;
    double time_sorted = 1.0 * (t2 - t1) / CLOCKS_PER_SEC;
    printf("random array of size %d : %.3f s\n", len, time_random);
    printf("sorted array of size %d : %.3f s\n", len, time_sorted);
    free(random);
    free(sorted);
    return 0;
}
```

On obtient alors (en compilant en -O2 et sans fsanitize pour obtenir des résultats un minimum réalistes) :

```
$ gcc -o qsort -Wall -Wextra -O2 qsort.c
$ ./qsort 10000
random array of size 10000 : 0.003 s
sorted array of size 10000 : 0.047 s
$ ./qsort 20000
random array of size 20000 : 0.001 s
sorted array of size 20000 : 0.183 s
$ ./qsort 30000
random array of size 30000 : 0.002 s
sorted array of size 30000 : 0.408 s
$ ./qsort 40000
random array of size 40000 : 0.003 s
sorted array of size 40000 : 0.712 s
$ ./qsort 50000
random array of size 50000 : 0.004 s
sorted array of size 50000 : 1.127 s
$ ./qsort 100000
random array of size 100000 : 0.008 s
sorted array of size 100000 : 4.354 s
```

Correction de l'exercice XXXII.3 page 646

1. `select(t, 0)` doit renvoyer le minimum du tableau, `select(t, n - 1)` le maximum.
2. `select(t, [n/2])` renvoie une médiane.

Correction de l'exercice XXXII.4 page 647

1. Pour faire la sélection en temps $O(n \log n)$, il suffit de trier le tableau à l'aide d'un algorithme ayant cette complexité dans le pire cas (tri fusion ou tri par tas, par exemple), puis de renvoyer la case k du tableau trié.
2. Pour calculer $\text{select}(t, k)$, on partitionne t et l'on regarde où se retrouve le pivot :
 - s'il est en position k , alors c'est la valeur recherché;
 - s'il est en positon $i_{\text{pivot}} < k$, alors l'élément recherché est à droite du pivot, et l'on fait donc un appel récursif $\text{select}(t[i_{\text{pivot}} + 1 : n], k - i_{\text{pivot}} - 1)$ (il faut changer la valeur de k puisqu'il y a k éléments dans la zone de gauche, plus le pivot);
 - s'il est en position $i_{\text{pivot}} > k$, alors l'élément cherché est dans la partie de gauche, et l'on effectue simplement l'appel $\text{select}(t[0 : i_{\text{pivot}}], k)$.

Cette fonction a pour effet secondaire de « trier partiellement » le tableau.

```
int quickselect_aux(int *t, int k, int len){
    assert (0 <= k && k < len);
    int ipiv = partition(t, len);
    if (ipiv == k) {
        return t[ipiv];
    }
    if (ipiv < k) {
        return quickselect_aux(&t[ipiv + 1], k - ipiv - 1, len - ipiv - 1);
    }
    return quickselect_aux(t, k, ipiv);
}
```

3. On effectue une copie du tableau puis l'on appelle `quickselect_aux` sur cette copie. On n'oublie pas de libérer la copie avant de renvoyer notre résultat.

```
int quickselect(int *t, int k, int len){
    int *t_copy = malloc(len * sizeof(int));
    memcpy(t_copy, t, len * sizeof(int));
    int res = quickselect_aux(t_copy, k, len);
    free(t_copy);
    return res;
}
```

4. À chaque étape, on effectue une partition (temps $O(n)$), puis :

- soit le pivot tombe à la bonne place et l'on a terminé;
- soit il faut faire un appel récursif sur l'une des parties, de taille au plus $n - 1$.

Si le tableau initial est trié par ordre croissant et que l'on calcule $\text{select}(t, n - 1)$ (en prenant le premier élément de la zone comme pivot à chaque fois), on partagera systématiquement en une zone vide et une de taille $n - 1$ et il faudra continuer les appels jusqu'à arriver à une zone de taille 1. La complexité est alors en $O(\sum_{k=1}^n k) = O(n^2)$.
 Si le pivot tombe à chaque fois au milieu, on a $T(n) \leq T(n/2) + An$, avec A une constante.

$$\begin{aligned} T(2^i) &\leq T(2^{i-1}) + A2^i \\ T(2^i) - T(2^{i-1}) &\leq A2^i \\ T(2^n) - T(1) &\leq A \sum_{i=2}^n 2^i \leq B2^n \end{aligned}$$

On en déduit $T(2^n) = O(2^n)$. Par croissance de T , on obtient $T(n) \leq T(2^{\lceil \log_2 n \rceil}) \leq B2^{\lceil \log_2 n \rceil} \leq B2^{\log_2 n + 1} = O(n)$. La complexité est donc linéaire si le pivot tombe au milieu du tableau à chaque fois. On peut prouver que la complexité en moyenne (sur un tableau mélangé de manière uniforme) est elle aussi linéaire.

Correction de l'exercice XXXII.5 page 647

On utilise un tas max (pour obtenir un tableau trié en ordre croissant à la fin). On n'utilise pas la fonction `siftdown` (donc on ne l'écrit pas), et la fonction d'extraction du maximum n'a pas à renvoyer de résultat (on l'utilise uniquement pour son effet de bord).

```

void siftdown(int *heap, int i, int len){
    int imin = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < len && heap[left] > heap[i]) imin = left;
    if (right < len && heap[right] > heap[imin]) imin = right;
    if (imin != i){
        swap(heap, i, imin);
        siftdown(heap, imin, len);
    }
}

void extract_max(int *heap, int len){
    swap(heap, 0, len - 1);
    siftdown(heap, 0, len - 1);
}

void heapify(int *heap, int len){
    for (int i = (len - 1) / 2; i >= 0; i--){
        siftdown(heap, i, len);
    }
}

void heapsort(int *heap, int len){
    heapify(heap, len);
    for (int i = len; i > 0; i--){
        extract_max(heap, i);
    }
}

```

Correction de l'exercice XXXII.6 page 647

1. On va au plus simple :

```

int ilog(int n){
    int i = 0;
    while (n > 1){
        i++;
        n = n >> 1;
    }
    return i;
}

```

2. Si le pivot tombe au milieu à chaque fois, la taille du tableau est divisée par 2 à chaque étape (à une unité près) et il faut donc $\log_2 n$ étapes (à une unité près) pour arriver à un tableau de taille inférieure ou égale à 1. La profondeur de récursion est donc de $\log_2 n$ dans ce cas.
3. On se fixe une profondeur maximale de $2 \log_2 n$, et si l'on dépasse cette profondeur on finit par un tri par tas. On peut donc avoir certaines zones du tableau qui finiront par être traitées par `heapsort` alors que d'autres seront traitées par `quicksort` jusqu'au bout.

```
void introsort_aux(int *t, int len, int max_depth){
    if (len <= 1) return;
    if (max_depth < 0) {
        heapsort(t, len);
    } else {
        int k = partition(t, len);
        introsort_aux(&t[k + 1], len - k - 1, max_depth - 1);
        introsort_aux(t, k, max_depth - 1);
    }
}

void introsort(int *t, int len){
    int max_depth = 2 * ilog(len);
    introsort_aux(t, len, max_depth);
}
```

DEUX EXEMPLES DE DIVISER POUR RÉGNER

I Paire la plus proche

On considère un ensemble de n points du plan et l'on souhaite déterminer la distance minimale entre deux de ces points. On représente un point par le type suivant :

```
type point = {x : float; y : float}
```

Un ensemble (ou *nuage*) de points sera représenté par une liste ou un tableau de points suivant les cas. On pourra éventuellement utiliser les fonctions `Array.of_list` et `Array.to_list` pour passer d'une représentation à l'autre quand l'une est plus appropriée.

Remarque

On pourra utiliser la valeur spéciale `infinity` (de type `float`) pour simplifier certaines fonctions.

Exercice XXXIII.1

p. 656

- Écrire une fonction `distance` calculant la distance euclidienne entre deux points.

```
distance : point -> point -> float
```

- Écrire une fonction `dmin_naif` résolvant le problème de la manière la plus simple possible.

```
dmin_naif : point array -> float
```

On se propose de chercher une solution « diviser pour régner » à ce problème. Pour ce faire, on va tenter d'exploiter l'idée suivante :

- separer le nuage de point en deux, suivant la valeur de x (la moitié de points la plus à gauche d'une part, la moitié la plus à droite d'autre part);
- calculer d_g (respectivement d_d), la distance minimale entre deux points situés tous à gauche (respectivement deux points situés à droite);
- en déduire d , la distance minimale entre deux points du nuage.

Exercice XXXIII.2

p. 656

- Que faut-il renvoyer dans les cas où le nuage contient zéro, un ou deux points ? Attention, ce sont bien sûr des cas différents !
- Écrire une fonction `separe_moitie` qui prend en argument une liste u de longueur n et renvoie un couple v, w de listes telles que $u = v @ w$, $|v| = \lfloor n/2 \rfloor$ et $|w| = n - |v|$.

```
separe_moitie : 'a list -> ('a list * 'a list)
```

- Écrire une fonction `compare_x` telle que l'appel `compare_x a b` renvoie :

- 1 si $a.x < b.x$;
- 0 si $a.x = b.x$;
- 1 sinon.

```
compare_x : point -> point -> int
```

4. Écrire une fonction `tri_par_x` qui trie une liste de points par coordonnée x croissante. On pourra utiliser la fonction `List.sort` de la bibliothèque standard, en cherchant sa documentation (ou en se référant au TP du début d'année sur le tri fusion).

```
tri_par_x : point list -> point list
```

5. Écrire une fonction `dmin_gauche_droite` qui prend en entrée deux listes de points et renvoie la distance minimale entre un point de la première liste et un point de la deuxième liste.

```
dmin_gauche_droite : point list -> point list -> float
```

6. En déduire une fonction `dmin_dc_naif` qui calcule la distance minimale entre deux points d'un nuage à l'aide d'une stratégie « diviser pour régner ».

```
dmin_dc_naif : point list -> float
```

7. Donner la relation de récurrence vérifiée par la complexité de `dmin_dc_naif`. Que peut-on en conclure ?

Pour obtenir une complexité satisfaisante, nous allons améliorer l'étape de fusion.

Exercice XXXIII.3

p. 657

On suppose ici que l'on a séparé notre ensemble de points en deux suivant l'axe des x et l'on note x_{med} une abscisse médiane (par exemple l'abscisse du point le plus à gauche de la partie de droite). On note également d_g (respectivement d_d) les distances minimales entre deux points de la partie de gauche (respectivement droite), que l'on suppose calculées. On note $d = \min(d_g, d_d)$, et l'on souhaite calculer d_{\min} (distance minimale entre deux points du nuage).

1. Justifier que l'on peut se limiter aux points dont l'abscisse vérifie $x_{\text{med}} - d \leq x \leq x_{\text{med}} + d$.
2. On suppose désormais que l'on dispose des points de cette bande, triés par *ordonnée* croissante. Justifier que l'on peut se contenter de calculer la distance minimale entre chaque point de cette liste et les 7 points suivants.

Exercice XXXIII.4

p. 658

1. Écrire une fonction `dmin_dc` implémentant la stratégie exposée ci-dessus.

```
dmin_dc : point list -> float
```

2. Montrer que la complexité temporelle de cette fonction vérifie $T(n) \leq 2T(n/2) + An \log n$.
3. En déduire qu'on a $T(n) = O(n(\log n)^2)$.
4. Comment pourrait-on réduire la complexité à $O(n \log n)$?
5. Si vous avez fini le reste du sujet. Écrire une nouvelle version de `dmin_dc` ayant cette complexité.

2 Nombre d'inversions

Exercice XXXIII.5 – Nombre d'inversions

p. 660

On définit le nombre d'inversions $\sigma(x)$ d'une séquence $x = x_0, \dots, x_{n-1}$ d'entiers comme le nombre de couples (i, j) tels que $1 \leq i < j \leq n$ et $x_i > x_j$.

On représentera ici les séquences par des listes.

1. Au maximum, combien vaut $\sigma(x)$?
2. Quelle est la complexité de l'algorithme naïf pour calculer $\sigma(x)$?
3. Écrire une fonction `nb_inv_naif`.

```
nb_inv_naif : 'a list -> int
```

4. En utilisant une stratégie « diviser pour régner », trouver un algorithme de complexité $O(n \log n)$ permettant de résoudre ce problème.

Remarque

On pourra s'inspirer du tri fusion.

5. Implémenter cet algorithme en OCaml.

```
nb_inv : 'a list -> int
```

Solutions

Correction de l'exercice XXXIII.1 page 653

1.

```
let distance a b = sqrt ((a.x -. b.x) ** 2. +. (a.y -. b.y) ** 2.)
```

2. On initialise à infinity, c'est le plus simple (et cela reste valable si $n < 2$).

```
let dmin_naif points =
  let n = Array.length points in
  let dmin = ref infinity in
  for i = 0 to n - 1 do
    for j = i + 1 to n - 1 do
      dmin := min !dmin (distance points.(i) points.(j))
    done
  done;
  !dmin
```

Correction de l'exercice XXXIII.2 page 653

- Pour $n = 0$ ou $n = 1$, il faut renvoyer infinity (ou lever une exception, mais cela rendrait l'écriture de la fonction nettement plus compliquée). Pour $n = 2$, on renvoie la distance entre les deux points (c'est le « vrai » cas de base).
- On a écrit ici une version récursive terminale (qui nécessite un appel à `List.rev` à la fin) pour ne pas être limité par la pile d'appels sur de très grandes listes, mais ce n'est pas indispensable. On pourra regarder le dernier exercice de ce sujet pour une autre version.

```
let separe u =
  let rec separe_aux v k pris =
    match v with
    | [] -> List.rev pris, []
    | x :: xs ->
      if k = 0 then List.rev pris, v
      else separe_aux xs (k - 1) (x :: pris) in
  separe_aux u (List.length u / 2) []
```

Remarque

On peut en fait remplacer `List.rev pris` par `[]` dans le premier cas du `match`.

3.

```
let compare_x a b =
  if a.x -. b.x < 0. then -1
  else if a.x = b.x then 0
  else 1
```

4.

```
let trie_par_x = List.sort compare_x
```

5. On pourrait convertir les deux listes en tableaux, mais ce n'est pas nécessaire :

```
let rec dmin_gauche_droite u v =
  match u with
  | [] -> infinity
  | hd :: tl ->
    (* on calcule la distance minimale entre p et un point de v *)
    let rec aux = function
      | [] -> infinity
      | hd' :: tl' -> min (distance hd hd') (aux tl') in
    min (aux v) (dmin_gauche_droite tl v)
```

6. On traduit exactement l'algorithme donné, en évitant cependant de re-trier à chaque étape : puisque `separe` respecte l'ordre, l'argument `points` de la fonction aux restera toujours trié.

```
let dmin_dc_naif points =
  let rec aux points =
    match points with
    | [] | [_] -> infinity
    | [a; b] -> distance a b
    | _ ->
      let gauche, droite = separe points in
      let dg = aux gauche in
      let dd = aux droite in
      let d_gd = dmin_gauche_droite gauche droite in
      min d_gd (min dg dd) in
  let par_x = trie_par_x points in
  aux par_x
```

7. Il y a $n/2$ points (à un près) dans gauche et dans droite, donc l'appel `dmin_gauche_droite` se fera en temps proportionnel à $(n/2)^2$ et donc à n^2 . La séparation peut être négligée puisqu'elle est en temps linéaire, et l'on obtient donc $T(n) = 2T(n/2) + An^2$. Il est alors immédiat que la complexité est au moins quadratique : on n'a donc rien gagné par rapport à la solution naïve.

On obtient en fait une complexité en $\Theta(n^2)$ (en appliquant la technique usuelle), et l'on n'a donc « rien perdu » non plus, sauf que l'algorithme est plus compliqué...

Correction de l'exercice XXXIII.3 page 654

1. Les seules distances que l'on souhaite considérer sont celles entre un point A situé à gauche et un point B situé à droite. Si par exemple A n'est pas dans la bande suggérée, on a $x_A < x_{\text{med}} - d$ et comme $x_B \geq x_{\text{med}}$ on en déduit $x_B - x_A > d$ et donc $\text{distance}(A, B) > d$.

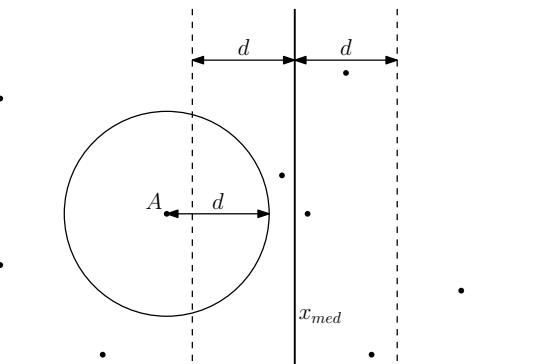


FIGURE XXXIII.3 – Un point de la moitié droite ne peut être à une distance inférieure à d de A.

2. Un bon dessin vaut mieux qu'un long discours :

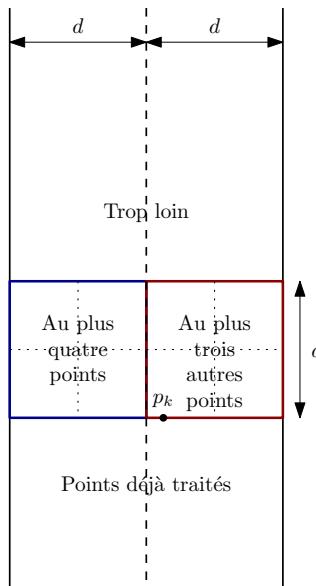


FIGURE XXXIII.4 – ei

Si l'on place 5 points dans le carré de gauche, on aura nécessairement deux points situés à une distance inférieure à d l'un de l'autre, ce qui est impossible. En effet, en divisant ce carré en quatre carrés de côté $d/2$, le principe des tiroirs qu'il y aurait deux points dans le même carré. Or la distance maximale entre deux points d'un tel carré est $\frac{d\sqrt{2}}{2} < d$ (longueur de la diagonale). Il y a donc au plus 4 points dans le carré de gauche, et 4 (dont le point actif) dans celui de droite : il suffit donc de s'intéresser aux 7 prochains points.

Correction de l'exercice XXXIII.4 page 654

1. Pour le calcul de la distance minimale avec l'un des sept prochains points, il est plus agréable d'utiliser un tableau :

```
let dmin_7 par_y =
  let dmin = ref infinity in
  let n = Array.length par_y in
  for i = 0 to n - 1 do
    for j = i + 1 to min (n - 1) (i + 7) do
      dmin := min !dmin (distance par_y.(i) par_y.(j))
    done
  done;
!dmin
```

Ensuite, on traduit l'énoncé, avec deux remarques :

- l'appel à `List.hd` est légitime, puisqu'on est dans un cas où `liste_points` est de longueur au moins 3 (et donc droite de longueur au moins 1);
- `liste_points` reste trié par `x` croissants tout au long des appels (on la trie par `y` croissants quand on en a besoin).

```

let dmin_dc points =
  let rec aux liste_points =
    match liste_points with
    | [] | [_] -> infinity
    | [a; b] -> distance a b
    | _ ->
      let gauche, droite = separe liste_points in
      let x_median = (List.hd droite).x in
      let d_gauche = aux gauche in
      let d_droite = aux droite in
      let d = min d_gauche d_droite in
      let dans_bande a =
        x_median -. d <= a.x && a.x <= x_median +. d in
      let bande = List.filter dans_bande liste_points in
      let tab_trie_y = Array.of_list (trie_par_y bande) in
      min d (dmin_7 tab_trie_y) in
    aux (trie_par_x points)
  
```

2. La séparation se fait en temps $O(n)$. Pour la fusion :

- le calcul de bande est en $O(n)$;
- le calcul de tab_trie_y est en $O(p \log p)$ où $p = |\text{bande}|$;
- le calcul de $d_{\min_7} \text{ tab_trie_y}$ est en $O(p)$.

On a $p \leq n$, et c'est la meilleure majoration que l'on puisse obtenir; on en déduit $T(n) \leq 2T(n/2) + An \log n$ avec A une constante.

3. On pose maintenant $T(n)$ le nombre maximal d'opérations élémentaires pour une instance de taille inférieure ou égale à n (ce qui assure la croissance). On a alors successivement :

$$\begin{aligned}
 T(2^i) &\leq 2T(2^{i-1}) + Ai2^i \\
 \frac{T(2^i)}{2^i} - \frac{T(2^{i-1})}{2^{i-1}} &\leq Ai \\
 \frac{T(2^k)}{2^k} - B &\leq A \sum_{i=1}^k i && B \text{ constante} \\
 T(2^k) &\leq Ck^22^k && C \text{ constante}
 \end{aligned}$$

Si $n = 2^k$, on a donc bien $T(n) = O(n \log^2 n)$. Dans le cas général, on a $n \leq 2^{\lceil \log_2 n \rceil}$, d'où $T(n) \leq C \cdot (\lceil \log_2 n \rceil)^2 2^{\lceil \log_2 n \rceil} \leq C(1 + \log_2 n) \cdot 2n$, donc également $T(n) = O(n \log^2 n)$.

4. Pour passer à du $O(n \log n)$, il faut éliminer l'étape de tri, ou plutôt ne trier qu'une seule fois au début. C'est possible : il commence par calculer la liste des points triés suivant x et la liste triée suivant y , puis filtrer ces listes à chaque appel récursif pour ne garder que ceux qui nous intéressent, sans modifier l'ordre (c'est ce que l'on fait déjà pour la liste triée suivant x).

Remarque

Si l'on savait trier une liste de flottants en temps linéaire (ce qui est possible en adaptant le tri radix), ce serait également une option valable.

5. Laissé en exercice... Cela ne pose aucun problème si l'on suppose que deux points distincts du nuage n'ont jamais la même abscisse ni la même ordonnée, il faut faire un peu plus attention si l'on ne fait plus cette hypothèse.

Correction de l'exercice XXXIII.5 page 655

1. $\sigma(x)$ est clairement majoré par le nombre de couples (i, j) vérifiant $0 \leq i < j < n$, c'est-à-dire $\frac{n(n-1)}{2}$. Inversement, tous les couples considérés sont en inversion si la séquence est strictement décroissante : ce majorant est donc un maximum.
2. Naïvement, on parcourt tous les couples (i, j) tels que $0 \leq i < j < n$ en testant à chaque fois si les éléments sont inversés. Comme dit plus haut, il y a de l'ordre n^2 tels couples, et la complexité serait donc en $\Theta(n^2)$.
3. C'est un peu plus facile à écrire avec un tableau qu'avec une liste, mais on va éviter de convertir : c'est un bon entraînement.

```
let rec nb_inv_naif u =
  match u with
  | [] -> 0
  | x :: xs ->
    (* on compte le nombre d'éléments de xs qui sont < x *)
    let rec aux = function
    | [] -> 0
    | y :: ys -> if x > y then 1 + aux ys else aux ys in
      aux xs + nb_inv_naif xs
```

4. On va simultanément trier la séquence par ordre croissant (à l'aide d'un tri fusion) et compter le nombre d'inversions. Le point crucial est que, si s et t sont deux séquences triées et que l'on considère $u = s @ t$, alors il est possible de déterminer le nombre d'inversions dans u en temps linéaire. En effet :
 - il n'y a pas d'inversion entre éléments de s ou entre éléments de t , et il suffit donc de compter le nombre d'inversions entre un élément de s et un élément de t (que l'on notera $\text{inv}(s, t)$) ;
 - si s ou t est vide, c'est trivial ;
 - sinon, on a $s = x :: xs$ et $t = y :: ys$ et l'on distingue deux cas :
 - si $x \leq y$, alors x n'est en inversion avec *aucun* élément de t (puisque elle est croissante), donc $\text{inv}(s, t) = \text{inv}(xs, t)$;
 - sinon, y est en inversion avec *tous* les éléments de s (puisque elle est croissante), et donc $\text{inv}(s, t) = |s| + \text{inv}(s, ys)$.

Ensuite, on remarque que si $u = s @ t$ (sans hypothèse sur s et t), on a en notant s' , t' les versions triées de s et t :

$$\sigma(u) = \sigma(s) + \sigma(t) + \text{inv}(s', t')$$

On va donc écrire une fonction (disons aux) qui prend en entrée une liste u et renvoie le couple p, u' où $p = \sigma(u)$ et u' est la version triée de u . On procède comme suit (si l'on n'est pas dans un cas de base) :

- on coupe la liste en deux (contrairement au tri fusion classique, il est indispensable de couper en première moitié, deuxième moitié) ;
- on appelle récursivement aux sur chacune des moitiés, on récupère $\sigma(s), s'$ et $\sigma(t), t'$;
- on fusionne s' et t' suivant la variante décrite plus haut pour obtenir $\text{inv}(s', s')$ et u' ;
- on renvoie le couple $(\sigma(s) + \sigma(t) + \text{inv}(s', u'), u')$.

En prenant le soin de ne pas recalculer inutilement la longueur à chaque étape de fusion, on obtiendra la complexité souhaitée.

5.

```

let separe u =
  let rec debut_fin u n =
    match n, u with
    | 0, _ -> [], u
    | _, x :: xs -> let a, b = debut_fin xs (n - 1) in (x :: a, b)
    | _ -> failwith "erreur" in
  debut_fin u ((List.length u) / 2)

let fusionne u v =
  let rec fus_aux u v n =
    match u, v with
    | [], _ -> (0, v)
    | _, [] -> (0, u)
    | x :: xs, y :: ys when x <= y ->
      let p, w = fus_aux xs v (n - 1) in
      (p, x :: w)
    | x :: xs, y :: ys ->
      let p, w = fus_aux u ys n in
      (p + n, y :: w)
    in
  let n = List.length u in
  fus_aux u v n

let nb_inv u =
  let rec aux u =
    match u with
    | [] | [_] -> (0, u)
    | _ -> let v, w = separe u in
      let n, a = aux v in
      let p, b = aux w in
      let q, c = fusionne a b in
      ((n + p + q), c) in
  let sigma, _ = aux u in sigma

```

`separe` est en $O(|u|)$ et `fusionne` en $O(|u| + |v|)$ (on ajoute un et un seul parcours à chaque fois pour calculer la longueur de la liste au début). On obtient donc la relation $T(n) \leq 2T(n/2) + An$ pour la complexité de `nb_inv`. C'est exactement la même relation que pour le tri fusion, et la complexité est donc identique : $O(n \log n)$.

PROGRAMMATION DYNAMIQUE : LE PROBLÈME DU SAC À DOS

Présentation du problème

On considère n objets numérotés de 0 à $n - 1$ ayant chacun un poids $p_i \in \mathbb{N}$ et une valeur $v_i \in \mathbb{N}$. On dispose d'un sac à dos pouvant contenir un nombre quelconque d'objets tant que la somme de leurs poids ne dépasse pas une constante P connue. On souhaite choisir un sous-ensemble des objets de valeur totale maximale parmi ceux rentrant dans le sac à dos.

Ce problème, dont le nom usuel est 0, 1-KNAPSACK, est un exemple célèbre de problème NP-complet, ce qui signifie en particulier que l'on ne connaît pas de solution en temps polynomial en n . Cependant, nous allons voir qu'il est possible de trouver une solution par programmation dynamique en temps polynomial en n et p .

Dans tout le problème, les poids et les valeurs seront donnés sous forme de deux **int array** que l'on appellera p et v . On notera en OCaml p_{max} le poids autorisé P . Comme d'habitude, on commencera par tenter de calculer la valeur optimale et l'on ne s'intéressera qu'ensuite au problème de la recherche d'une solution optimale.

Exercice XXXIV.1 – Force brute

Si l'on souhaite procéder en énumérant toutes les solutions possibles :

1. combien faut-il en considérer ?
2. que complexité peut-on raisonnablement attendre ?

I Une relation de récurrence

On définit $f(k, d)$ la valeur maximale que l'on peut obtenir si le poids disponible est d et que l'on se limite à choisir des objets dont l'indice est strictement inférieur à k .

Exercice XXXIV.2

1. Quelle valeur de f veut-on calculer ?
2. Exprimer $f(1, d)$ en fonction de d , p_0 et v_0 .
3. Pour $k \geq 1$, donner une relation entre $f(k + 1, d)$ et les $f(k, d')$ (faisant bien sûr intervenir les poids et les valeurs des objets).
4. Quelle valeur peut-on raisonnablement donner à $f(0, d)$ (avec $d \geq 0$) ? Quel intérêt cela présente-t-il ?
5. Peut-on de même définir $f(k, d)$ quand $d < 0$?

Exercice XXXIV.3 – Solution récursive naïve

Écrire une fonction $\text{sac } p \ v \ p_{\text{max}}$ répondant au problème posé. On utilisera une fonction auxiliaire récursive $f \ k \ d$ correspondant à la définition vue plus haut.

2 Programmation dynamique ascendante et descendante

Exercice XXXIV.4 – Chevauchement de sous-problèmes

Comme nous l'avons vu en cours, deux conditions doivent être réunies pour que l'approche par programmation dynamique ait un intérêt : la présence de sous-structures optimales, que nous avons mise en évidence à la partie précédente, et le chevauchement de sous-problèmes. Contrairement aux exemples que nous avons vus jusqu'à présent, il n'est pas évident ici de déterminer *a priori* combien de fois chaque appel à $f(n, d)$ sera effectué. Nous allons donc utiliser une approche expérimentale.

1. Pour les p_{ex} et v_{ex} fournis et en prenant $P = 100$, combien y a-t-il au maximum d'appels distincts à $f(n, d)$? *On veut juste une majoration, le nombre exact d'appels distincts dépend du contenu de p_{ex} et de v_{ex} .*
2. Écrire une fonction `sac_instrumente p v pmax` effectuant le même travail que `sac` mais comptant en parallèle le nombre total d'appels à f effectués. Le plus simple est de définir une variable mutable que l'on incrémentera à chaque appel.
3. Dans le cas du 1, que peut-on dire du nombre moyen de fois que l'on effectue chaque appel (distinct)?

Exercice XXXIV.5

Écrire deux fonctions `sac_mem p v pmax` et `sac_dyn p v pmax` implémentant respectivement les versions descendantes et ascendantes de la solution par programmation dynamique du problème. Pour créer un tableau de tableaux (*i.e.* une matrice) de dimensions $n \times p$ initialisée avec la valeur x , on pourra utiliser `Array.make_matrix n p x`.

Comme souvent, on prendra une table de type `int array array` dans le cas ascendant et `int option array array` dans le cas descendant.

Exercice XXXIV.6

Déterminer (ou majorer de manière pas trop grossière) les complexités en temps et en espace des deux fonctions de la question précédente.

3 Reconstruction de la solution

Exercice XXXIV.7

Reconstruire une solution optimale demande de retrouver la série de choix qui a permis d'obtenir la valeur optimale. Souvent, le plus simple est de stocker le choix fait (on prend l'objet ou pas) dans chaque case de la table.

- Dans le cas descendant, on peut stocker dans chaque case un couple ^a $(f(k, d), s)$ où s est une liste correspondant à une solution optimale pour les objets 0 à k avec un poids maximal de d . Il n'y a alors pas de phase de reconstruction à proprement parler : on construit la solution au fur et à mesure des appels récursifs.
- Dans le cas ascendant, il est plus naturel d'utiliser un `(int * bool) array array`, où le booléen indique si l'objet a été choisi. Il faut alors reconstruire la liste solution dans un deuxième temps.

Implémenter l'une de ces approches (au choix). On renverra un couple $(\text{valeur}_{\text{opt}}, \text{solution}_{\text{opt}})$, où la solution est donnée sous forme d'une liste d'entiers.

^a. ou plutôt une option sur un tel couple

Exercice XXXIV.8

En réalité, on peut ici assez facilement retrouver la série de choix effectués sans stocker d'information supplémentaire (c'est-à-dire en travaillant avec les mêmes tables qu'à la partie précédente). Expliquer comment on procéderait.

4 De l'intérêt des deux approches**Exercice XXXIV.9 – Intérêt de la solution mémoisée**

Déterminer le nombre de valeurs de $f(k, d)$ que la fonction `sac_mem` calcule pour $P = 100$ avec les p et v donnés en exemple. Que constate-t-on ?

Exercice XXXIV.10 – Intérêt de la solution *bottom-up*

Expliquer comment diminuer la complexité en espace de la fonction `sac_dyn` (et le faire si vous avez le temps). Quel problème cela pose-t-il ?

MAXIMISATION D'UNE SOMME

I Introduction

Le but de ce TD est de résoudre le problème 345 du Projet Euler (<http://projecteuler.net>), à savoir : étant donné une matrice carrée $M = (m_{i,j}) \in \mathcal{M}_n(\mathbb{N})$, calculer

$$s(M) = \max_{\sigma \in S_n} \sum_{i=0}^{n-1} m_{i,\sigma(i)}$$

où S_n désigne l'ensemble des permutations de $E_n = \{0 \dots n - 1\}$.

Pour la matrice M_1 ci-dessous, on obtient $s(M_1) = 3315$.

$$M_1 = \begin{pmatrix} 7 & 53 & 183 & 439 & 863 \\ 497 & 383 & 563 & 79 & 973 \\ 287 & 63 & 343 & 169 & 583 \\ 627 & 343 & 773 & 959 & 943 \\ 767 & 473 & 103 & 699 & 303 \end{pmatrix}$$

Le but est de calculer $s(M_2)$ où M_2 est la matrice 15×15 définie par :

$$M_2 = \begin{pmatrix} 7 & 53 & 183 & 439 & 863 & 497 & 383 & 563 & 79 & 973 & 287 & 63 & 343 & 169 & 583 \\ 627 & 343 & 773 & 959 & 943 & 767 & 473 & 103 & 699 & 303 & 957 & 703 & 583 & 639 & 913 \\ 447 & 283 & 463 & 29 & 23 & 487 & 463 & 993 & 119 & 883 & 327 & 493 & 423 & 159 & 743 \\ 217 & 623 & 3 & 399 & 853 & 407 & 103 & 983 & 89 & 463 & 290 & 516 & 212 & 462 & 350 \\ 960 & 376 & 682 & 962 & 300 & 780 & 486 & 502 & 912 & 800 & 250 & 346 & 172 & 812 & 350 \\ 870 & 456 & 192 & 162 & 593 & 473 & 915 & 45 & 989 & 873 & 823 & 965 & 425 & 329 & 803 \\ 973 & 965 & 905 & 919 & 133 & 673 & 665 & 235 & 509 & 613 & 673 & 815 & 165 & 992 & 326 \\ 322 & 148 & 972 & 962 & 286 & 255 & 941 & 541 & 265 & 323 & 925 & 281 & 601 & 95 & 973 \\ 445 & 721 & 11 & 525 & 473 & 65 & 511 & 164 & 138 & 672 & 18 & 428 & 154 & 448 & 848 \\ 414 & 456 & 310 & 312 & 798 & 104 & 566 & 520 & 302 & 248 & 694 & 976 & 430 & 392 & 198 \\ 184 & 829 & 373 & 181 & 631 & 101 & 969 & 613 & 840 & 740 & 778 & 458 & 284 & 760 & 390 \\ 821 & 461 & 843 & 513 & 17 & 901 & 711 & 993 & 293 & 157 & 274 & 94 & 192 & 156 & 574 \\ 34 & 124 & 4 & 878 & 450 & 476 & 712 & 914 & 838 & 669 & 875 & 299 & 823 & 329 & 699 \\ 815 & 559 & 813 & 459 & 522 & 788 & 168 & 586 & 966 & 232 & 308 & 833 & 251 & 631 & 107 \\ 813 & 883 & 451 & 509 & 615 & 77 & 281 & 613 & 459 & 205 & 380 & 274 & 302 & 35 & 805 \end{pmatrix}$$

On se donne la contrainte que le programme doit résoudre le problème en moins d'une minute.

On représentera ces deux matrices en OCaml par un tableau de lignes, chaque ligne étant représentée par un tableau ; elles sont définies dans le fichier joint.

2 Algorithme naïf

L'algorithme naïf consiste à énumérer toutes les permutations de S_n et à calculer pour chacune la somme des éléments.

Exercice XXXVI.

1. En ne comptant que les additions entre éléments de la matrice, combien d'opérations faut-il faire avec cette méthode pour traiter une matrice de taille n ?
2. En utilisant la « recette de cuisine » du cours sur la complexité, donner alors une fourchette réaliste pour le temps d'exécution sur des matrices de taille 8, 10, 12, 15 et 20. Que peut-on conclure pour notre problème ?

3 Récursion naïve

Une matrice carrée de taille M et une partie S de E_n étant données, on définit

$$\text{eval}(M, S) := \max \left\{ \sum_{i=0}^{|S|-1} m_{i,f(i)} \mid f : [0 \dots |S|-1] \rightarrow S \text{ avec } f \text{ bijective} \right\}$$

Exercice XXXV.2 – Solution récursive naïve

1. Que valent $\text{eval}(M, \emptyset)$? $\text{eval}(M, E_n)$?
2. Que représente $\text{eval}(M, S)$? *Faire un dessin!*
3. Pour $S \neq \emptyset$, exprimer $\text{eval}(M, S)$ en fonction des $\text{eval}(M, S \setminus \{i\})$ pour $i \in S$ et des coefficients de M .
4. Écrire une fonction `range` : `int -> int -> int list` telle que `range a b` renvoie la liste `[a; a + 1; ...; b - 1]`.
5. Écrire une fonction `prive_de` : `int list -> int list` telle que l'appel `prive_de u x` renvoie une liste ayant les mêmes éléments que `u`, sauf que la première occurrence de `x` a été supprimée :

```
# prive_de [1; 3; 5; 7] 3;;
- : int list = [1; 5; 7]
```

6. Écrire une fonction `max_liste` : `int list -> int` calculant le maximum d'une liste d'entiers.
7. Écrire une fonction `eval` : `int array array -> int list -> int` calculant récursivement $\text{eval}(M, S)$. Le premier argument de `eval` est la matrice M , le deuxième l'ensemble S représenté par la liste de ses éléments.
8. Vérifier qu'elle calcule le résultat correct pour M_1 .

Exercice XXXV.3 – Analyse de la solution récursive naïve

1. On note $\varphi(k)$ le nombre total d'appels effectués lors du calcul de $\text{eval}(M, S)$ quand $|S| = k$. Justifier que $\varphi(k) \geq k!$.
2. Combien de valeurs *differentes* S prend-il lors de ces appels ?

4 Solution en programmation dynamique

Comme nous venons de le voir, il n'y a que 2^n valeurs de $\text{eval}(M, S)$ à calculer pour une matrice M donnée de taille n . En mémoisant ces appels, on peut donc espérer réduire la complexité à « un peu plus » de 2^n (peut-être 2^n , peut-être $n2^n$, peut-être $n^22^n\dots$). Or, autant $15!$ est « vraiment grand », autant 2^{15} (qui vaut 32 768) ne l'est pas : pour *notre problème précis*, ça devrait marcher !

Exercice XXXV.4

M étant fixée, on va utiliser un dictionnaire dans lequel on va stocker les associations $(S, \text{eval}(M, S))$ au fur et à mesure du calcul. Le principe de calcul de $\text{eval}(M, S)$ sera donc le suivant :

- Si la table globale contient déjà une entrée (S, v) on sait que v est la valeur cherchée pour $\text{eval}(M, S)$: on la renvoie directement.
- Si aucune entrée de cette forme (S, v) n'existe dans la table, on calcule $\text{eval}(M, S)$ par la méthode récursive vue plus haut : on calcule récursivement les $\text{eval}(M, S \setminus \{i\})$ (ces appels récursifs ajouteront eux-mêmes leur résultat dans la table). On peut alors calculer la valeur v de $\text{eval}(M, S)$. On note (S, v) dans la table et on retourne v .

Pour réaliser le dictionnaire, on va utiliser les tables de hachage qui sont prédefinies en OCaml. On aura besoin des fonctions suivantes :

- `Hashtbl.create` : `int` -> (`'a`, `'b`) `Hashtbl.t` qui crée une table de hachage vide (l'entier passé en argument donne la taille initiale de la table, il est souhaitable qu'il soit de l'ordre du nombre d'éléments que l'on compte stocker dans la table);
- `Hashtbl.mem` : (`'a`, `'b`) `Hashtbl.t` -> `'a` -> `bool` telle que `Hashtbl.mem t k` renvoie `true` si `t` a une valeur associée à `k`, `false` sinon;
- `Hashtbl.find` : (`'a`, `'b`) `Hashtbl.t` -> `'a` -> `'b` qui prend en argument une table et une clé et renvoie la valeur associée à la clé (ou une exception `Not_found` s'il n'y a pas de telle valeur);
- `Hashtbl.add` : (`'a`, `'b`) `Hashtbl.t` -> `'a` -> `'b` -> `unit` telle que `Hashtbl.add t k v` rajoute l'association `(k, v)` à la table `t`.

1. Écrire une fonction `eval_mem` et calculer $s(M_2)$. Il faudra une fonction principale, non récursive, dont le travail sera simplement de créer une table vide et d'appeler la fonction auxiliaire. Cette fonction auxiliaire sera, elle, récursive, et procédera comme décrit ci-dessus.

Ici, il est très important que les listes correspondant aux ensembles S soient triées, sinon un même ensemble pourrait être représenté par deux listes différentes et on ne retrouverait pas nos données dans la table.

2. **Question délicate que je vous invite à sauter dans un premier temps.** En supposant que l'accès à un élément de la table se fasse en temps constant, quelle est la complexité de la fonction `eval_mem`?

5 Optimisation

On peut représenter le sous-ensembles S de $\llbracket 0, n \rrbracket$ par l'entier $\sum_{i \in S} 2^i$. Ainsi, les parties de $\llbracket 0, n \rrbracket$ sont représentées par des entiers compris entre 0 et $2^n - 1$ inclus. On n'a alors plus besoin d'une table de hachage : on peut utiliser un tableau de 2^n entiers à la place.

Écrire une version de l'algorithme dynamique précédent en utilisant cette structure de données ; le plus simple est d'utiliser une approche *top-down*, mais du *bottom-up* est également possible (et, bien programmé, est même encore un peu plus rapide).

Il existe en fait une solution en temps polynomial à ce problème, mais l'algorithme est plus compliqué et ne se comprend bien qu'avec un peu de théorie des graphes. Nous aurons sans doute l'occasion de revenir dessus en deuxième année.

PLUS LONGUE SOUS-SÉQUENCE CROISSANTE

Définitions et notations

Dans tout le sujet, on s'intéresse à des séquences $s = s_0, \dots, s_{n-1}$ d'entiers. Ces séquences peuvent être représentées en Caml soit par des listes (type `int list`), soit par des tableaux (type `int array`).

Les fonctions que l'on vous demande d'écrire accepteront en fait des types plus généraux (`'a list` et `'a array`), car les seules opérations que l'on fera sur les éléments sont des comparaisons, qui sont polymorphes en OCaml. Il n'y a pas de problème : si l'énoncé vous demande une fonction $f : \text{int list} \rightarrow \text{bool}$ et que vous écrivez une fonction $f : 'a list \rightarrow \text{bool}$ qui a le comportement attendu quand elle est appelée sur une liste d'entiers, vous avez évidemment répondu à la question.

Étant donnée une séquence $s = s_0, \dots, s_{n-1}$:

- la *longueur* de s , notée $|s|$, est son nombre d'éléments n ;
 - s est dite croissante si $s_0 \leq s_1 \leq \dots \leq s_{n-1}$;
 - une *sous-séquence* de longueur k de s est une séquence $s_{\varphi(0)}, \dots, s_{\varphi(k-1)}$ où φ est strictement croissante. Si $k = 0$, la sous-séquence est vide.
- Par exemple, $u = 7, 2, 8$ est une sous-séquence de $s = 7, 1, 2, 6, 4, 5, 8$ (avec $\varphi(0) = 0$, $\varphi(1) = 2$ et $\varphi(3) = 6$). En revanche, ni $v = 2, 8, 1$ ni $w = 7, 7, 6$ ne sont des sous-séquences de s .
- on notera $l_{\text{seq}}(s)$ la longueur maximale d'une sous-séquence croissante de s . Autrement dit :

$$l_{\text{seq}}(s) \stackrel{\text{déf.}}{=} \max(|u|, u \text{ sous-séquence de } s \text{ et } u \text{ croissante})$$

Ce problème porte sur le calcul efficace de l_{seq} ainsi que sur l'extraction d'une sous-séquence croissante maximale. Pour $s = 7, 1, 2, 6, 4, 5, 8$, on a $l_{\text{seq}}(s) = 5$ réalisé pour $u = 1, 2, 4, 5, 8$:

I Méthode par énumération

Dans cette partie, on représente les séquences par des listes d'entiers.

- ▶ **Question 1** Exprimer en fonction de $|s|$ le nombre de sous-séquences de s (en supposant que les éléments de s sont deux à deux distincts).
- ▶ **Question 2** Écrire une fonction `est_croissante : int list -> bool` qui renvoie `true` si son argument est une liste croissante, `false` sinon. La liste vide sera considérée comme croissante.
- ▶ **Question 3** Écrire une fonction `prefixe : 'a -> 'a list list -> 'a list list` telle que `prefixe x [u_1 ; ... ; u_n]` renvoie `[x :: u_1 ; ... ; x :: u_n]`.
- ▶ **Question 4** Écrire une fonction `sous_sequences : (s : 'a list) : 'a list list` qui renvoie la liste de toutes les sous-séquences de s . On doit donc avoir, à l'ordre près :
`sous_sequences [8; 3; 5] = [[8; 3; 5]; [8; 3]; [8; 5]; [8]; [3; 5]; [3]; [5]; []]`.
- ▶ **Question 5** Écrire une fonction `l_seq_naif (s : int list) : int` qui renvoie $l_{\text{seq}}(s)$. On procédera de manière brutale en générant toutes les sous-séquences de s .
- ▶ **Question 6** Quelle est la complexité de `l_seq_naif` (en fonction de $|s|$) ?

2 Méthode par programmation dynamique

Dans cette partie, on représente une séquence $s = s_0, \dots, s_{n-1}$ par un $s : \text{int array}$ de taille n . On associe à s un tableau longueurs de taille n tel que $\text{longueurs}.(k)$ soit la longueur de la plus longue sous-séquence croissante de la forme $s_{\varphi(0)}, \dots, s_{\varphi(i)}$ avec $\varphi(i) = k$ (autrement dit, la longueur de la plus grande sous-séquence croissante de s se terminant exactement en s_k). On peut par exemple avoir :

s	10	12	2	8	3	11	7	14	9	4
longueurs	1	2	1	2	2	3	3	4	4	3

► **Question 7** Montrer que, pour $0 \leq k < n - 1$, on a :

$$\text{longueurs}.(k + 1) = \begin{cases} 1 & \text{si } s_{k+1} < \min(s_0, \dots, s_k) \\ 1 + \max\{\text{longueurs}.(i) \mid 0 \leq i \leq k \text{ et } s_i \leq s_{k+1}\} & \text{sinon} \end{cases}$$

► **Question 8** En déduire une fonction $\text{aux_dyn}(s : \text{int array}) : \text{int array}$ qui renvoie le tableau longueurs associé à s . On demande une complexité en $O(|s|^2)$, que l'on justifiera.

► **Question 9** Écrire alors une fonction $\text{l_seq_dyn}(s : \text{int array}) : \text{int}$ calculant $\text{l}_{\text{seq}}(s)$ et préciser sa complexité.

► **Question 10** Écrire une fonction $\text{sous_sequence_dyn}(s : \text{int array}) : \text{int array}$ qui renvoie une sous-séquence croissante de s de longueur maximale, et préciser sa complexité.
On reconstruira la séquence à partir du tableau longueurs.

3 Méthode de la patience

L'algorithme présenté dans cette partie est inspiré d'un jeu de cartes (d'une réussite, plus précisément) appelé *patience*, dont le principe est exposé ci-dessous.

On dispose d'un paquet de n cartes numérotées (les numéros sont des entiers, plusieurs cartes peuvent éventuellement porter le même numéro). On prend les cartes une par une, depuis le sommet du paquet, et l'on doit les organiser en piles en respectant les règles suivantes :

- au début du jeu, il n'y a aucune pile;
- quand on tire une nouvelle carte, on peut :
 - soit la mettre sur une nouvelle pile (que l'on placera à droite de toutes les piles existantes);
 - soit la rajouter au sommet d'une pile existante, à condition qu'elle soit strictement plus petite que la carte actuellement au sommet de cette pile.

Par exemple, supposons que l'on tire une carte 17 et que l'on soit dans la configuration suivante :

18	20	25	13
20	17	28	
23	19		

17 p₁ p₂ p₃ p₄

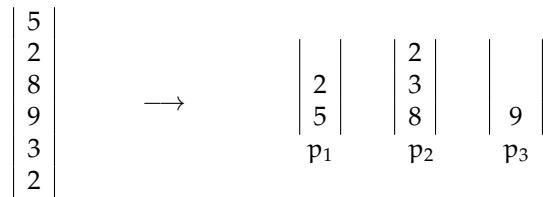
On peut rajouter le 17 au sommet de p_1 , au sommet de p_3 ou au sommet d'une nouvelle pile p_5 , mais pas au sommet de p_2 ni de p_4 .

Le but du jeu est de terminer le paquet en utilisant le moins de piles possibles. Pour ce faire, on utilise la stratégie *gloutonne*.

Stratégie gloutonne : à chaque fois qu'on tire une nouvelle carte, on la place le plus à gauche possible.

Dans l'exemple ci-dessus, on ajouterait donc le 17 au sommet de la pile p_1 .

- **Question 11** Vérifier qu'en utilisant la stratégie gloutonne, on a la transformation suivante (du paquet initial vers l'état en fin de partie) :



- **Question 12** Montrer que si l'on suit la stratégie gloutonne, alors après chaque étape les sommets des piles, lus de gauche à droite, forment une suite croissante.

On note $s = a_0, \dots, a_{n-1}$ la séquence correspondant à un paquet (a_0 est la première carte tirée).

- **Question 13** Montrer que quelle que soit la stratégie utilisée, on utilise au moins $l_{seq}(s)$ piles.

- **Question 14** Montrer que la stratégie gloutonne utilise exactement $l_{seq}(s)$ piles (et est donc optimale).

Étant donnée une séquence s , on peut donc déterminer $l_{seq}(s)$ en « jouant une partie » avec le paquet s en suivant la stratégie gloutonne et en comptant le nombre de piles utilisées.

La séquence s est donnée sous forme de liste, et l'on choisit d'utiliser le type suivant pour représenter les configurations :

```
type config = int list array
```

- **Question 15** Écrire une fonction `patience` : `int list` → `config` qui prend en argument un paquet (sous la forme d'une liste) et renvoie la configuration obtenue à la fin du jeu en suivant la stratégie gloutonne. On doit donc avoir `patience [5; 2; 8; 9; 3; 2] = [[2; 5]; [2; 3; 8]; [9]; []; []; []]]`. On demande une complexité en $O(|s|^2)$ dans le pire des cas, ce que l'on justifiera.

- **Question 16** Comment pourrait-on utiliser le résultat de la question 12 pour abaisser la complexité de la fonction `patience` ?

À cette question, on demande seulement de donner l'idée de l'algorithme.

- **Question 17** Écrire une fonction `patience_opt` : `int list` → `config` ayant la même spécification que `patience` mais une complexité en $O(|s| \cdot \log |s|)$, que l'on justifiera.

- **Question 18** Écrire une fonction `l_seq_patience` ($s : \text{int list}$) : `int` qui calcule $l_{seq}(s)$ en temps $O(|s| \cdot \log |s|)$.

- **Question 19** Expliquer comment modifier la manière de stocker les configurations pour pouvoir reconstruire à la fin du calcul une sous-séquence croissante de longueur maximale.

- **Question 20** Écrire une fonction `sous_sequence_patience` ($s : \text{int list}$) : `int list` qui renvoie une sous-séquence croissante de s de longueur maximale en temps $O(|s| \ln |s|)$.

4 Bonus

- **Question 21** Démontrer le théorème suivant :

Théorème – Erdős-Szekeres

Toute séquence de $n^2 + 1$ entiers contient une sous-séquence monotone de longueur $n + 1$.

Solutions

► **Question 1** Choisir une sous-séquence de s , c'est choisir une partie de $\llbracket 0, |s|-1 \rrbracket$: il y a $2^{|s|}$ sous-séquences.

► **Question 2**

```
let rec est_croissante u =
  match u with
  | [] | [_] -> true
  | a :: b :: xs -> a <= b && est_croissante (b :: xs)
```

► **Question 3**

```
let prefixe x u =
  List.map (fun v -> x :: v) u
```

Il serait plus idiomatique d'écrire `let prefixe x = List.map (fun v -> x :: v)`.

► **Question 4** On a $\mathcal{P}(\{x\} \cup A) = \mathcal{P}(A) \cup \{\{x\} \cup B, B \in \mathcal{P}(A)\}$, et l'union est disjointe si $x \notin A$.

```
let rec sous_sequences u =
  match u with
  | [] -> [[]]
  | x :: xs ->
    let sans_x = sous_sequences xs in
    let avec_x = prefixe x sans_x in
    avec_x @ sans_x
```

► **Question 5**

```
(* max_croissante : 'a list list -> int
   max_croissante [u_1; ...; u_n] renvoie le max des
   longueurs de u_k pour u_k croissante *)
let rec max_croissantes u =
  match u with
  | [] -> 0
  | x :: xs when est_croissante x -> max (List.length x) (max_croissantes xs)
  | x :: xs -> max_croissantes xs

(* l_seq_naif : 'a list -> 'a list *)
let l_seq_naif u =
  max_croissantes (sous_sequences u)
```

► **Question 6**

- `max_croissantes [u_1; ... ; u_n]` effectue un parcours de chacune des listes u_1, \dots, u_n et a donc une complexité en $O(\sum_{i=1}^n |u_i|)$.
- Quand on l'appelle sur toutes les sous-séquences de u , on obtient donc (sachant qu'il y a $\binom{|s|}{k}$ sous-séquences de longueur k) :

$$O\left(\sum_{k=0}^{|s|} k \binom{|s|}{k}\right) = O\left(\sum_{k=0}^{|s|} |s| \binom{|s|-1}{k-1}\right) = O\left(|s| \cdot 2^{|s|-1}\right) = O\left(|s| \cdot 2^{|s|}\right)$$

- Il reste à prendre en compte le temps de construction des sous-séquences, mais il est clairement dominé par $O(|s| \cdot 2^{|s|})$.

Remarque

Montrer que ce coût est en fait un $O(2^{|s|})$ est un bon exercice de calcul de complexité. Il faut appliquer le même genre de technique que pour les algorithmes « diviser pour régner ».

La complexité de `l_seq_naif` est donc en $O(|s| \cdot 2^{|s|})$.

► Question 7 Soit $0 \leq k \leq n - 2$.

- Si $s_{k+1} < \min(s_0, \dots, s_k)$, alors la seule sous-séquence croissante se terminant en s_{k+1} est celle réduite à s_{k+1} , qui est de longueur 1.
- Sinon, il y a au moins un i vérifiant $0 \leq i \leq k$ et $s_i \leq s_{k+1}$.
 - Pour chacun de ces i , on peut obtenir une sous-séquence croissante de taille $1 + \text{longueur}[i]$ se terminant en s_{k+1} en prenant une sous-séquence maximale se terminant en s_i et en lui rajoutant s_{k+1} . On a donc $\text{longueur}[k+1] \geq 1 + \max\{\text{longueur}[i] \mid 0 \leq i \leq k \text{ et } s_i \leq s_{k+1}\} (\geq 2)$.
 - Inversement, toute sous-séquence croissante de longueur supérieure ou égale à 2 se terminant en s_{k+1} est de la forme \dots, s_i, s_{k+1} pour un certain i vérifiant $0 \leq i \leq k$ et $s_i \leq s_{k+1}$. La sous-séquence \dots, s_i est croissante, donc de longueur au plus $\text{longueur}[i]$, et la séquence \dots, s_{k+1} est donc de longueur au plus $1 + \text{longueur}[i]$. On a donc l'autre inégalité.

On a donc bien

$$\text{longueur}[k+1] = \begin{cases} 1 & \text{si } s_{k+1} < \min(s_0, \dots, s_k) \\ 1 + \max\{\text{longueur}[i] \mid 0 \leq i \leq k \text{ et } s_i \leq s_{k+1}\} & \text{sinon} \end{cases}$$

► Question 8 Traduction immédiate de la récurrence :

```
(* aux_dyn : 'a array -> int array
   A l'étape i, longueur.(k) contient la longueur de la plus
   longue sous-suite croissante se terminant en k (ou 0 si k > i) *)
let aux_dyn t =
  let n = Array.length t in
  let longueur = Array.create n 0 in
  for i = 0 to n - 1 do
    let maxi = ref 1 in
    for k = 0 to i - 1 do
      if t.(k) <= t.(i) then maxi := max !maxi (longueur.(k) + 1)
    done;
    longueur.(i) <- !maxi
  done;
  longueur
```

Le corps de la boucle interne est en temps constant, donc la boucle interne est en $O(i)$ et la boucle externe en $O\left(\sum_{i=0}^{n-1} i\right) = O(n^2)$.

L'initialisation est en $O(n)$, on a bien une complexité totale en $O(n^2)$.

► Question 9 Il suffit de prendre le maximum du tableau `longueur`. La fonction `ind_et_max` servira réellement à la question suivante.

```
(* indice d'une occurrence du maximum et valeur de ce maximum *)
let ind_et_max t =
  let ind_maxi = ref 0 in
  for i = 1 to Array.length t - 1 do
    if t.(i) > t.(!ind_maxi) then ind_maxi := i
  done;
!ind_maxi, t.(!ind_maxi)

(* l_seq_dyn : 'a array -> int *)
let l_seq_dyn t = snd (ind_et_max (aux_dyn t))
```

► Question 10

```
(* On détermine à quel endroit se termine la plus grande sous-suite croissante
(l'une des, en fait), ainsi que sa longueur. On parcourt le tableau vers la
gauche à partir de ce point, en prenant à chaque fois un élément pour
lequel la longueur est égale au nombre d'éléments restant à choisir et la
valeur est inférieure à celle du dernier élément choisi. *)
let sous_sequence_dyn t =
  let tab_longueur = aux_dyn t in
  let ind_dernier, longueur = ind_et_max tab_longueur in
  let sous_suite = Array.create longueur t.(ind_dernier) in
  let k = ref (ind_dernier - 1) in
  let a_choisir = ref (longueur - 1) in
  while !a_choisir > 0 do
    if tab_longueur.(!k) = !a_choisir && t.(!k) <= sous_suite.(!a_choisir) then
      begin
        a_choisir := !a_choisir - 1;
        sous_suite.(!a_choisir) <- t.(!k)
      end;
    decr k (* équivaut à k := !k - 1 *)
  done;
sous_suite
```

La reconstruction rajoute simplement un parcours du tableau en $O(n)$, la complexité reste donc en $O(n^2)$.

► Question 11 On le vérifie...

► Question 12 C'est bien sûr vrai au départ (suite vide), supposons que ce soit vrai au moment de tirer une carte x et notons $s_1 \leq \dots \leq s_p$ les sommets.

- Si $x \geq s_p$, on rajoute une pile et l'on obtient $s_1 \leq \dots \leq s_p \leq s_{p+1} = x$.
- Si $x < s_1$, on place x sur la première pile et obtient $s'_1 = x < s_2 \leq \dots \leq s_p$.
- Sinon, on trouve l'unique i tel que $s_i \leq x < s_{i+1}$ et l'on place x sur la pile $i + 1$. On obtient alors $s_1 \leq \dots \leq s_i \leq s'_{i+1} = x < s_{i+2} \leq \dots$

Par récurrence, la suite des sommets est croissante à tout instant.

► Question 13 Remarquons d'abord que les règles imposent que chaque pile, lire de bas en haut, soit strictement décroissante. Comme les cartes sont empilées dans l'ordre de tirage, cela implique que si $i < j$ et $a_i \leq a_j$, les deux cartes ne peuvent être dans la même pile. Ainsi, les cartes d'une sous-séquence croissante sont nécessairement rangées dans des piles deux à deux distinctes. En considérant une sous-séquence croissante de longueur maximale, on en déduit qu'il faut au moins $l_{seq}(s)$ piles.

► **Question 14** Imaginons qu'à chaque fois que l'on place une carte sur une pile, on rajoute un pointeur de cette carte vers le sommet de la pile située immédiatement à gauche de cette pile (si la carte est placée sur la première pile, disons qu'on ne crée pas de pointeur).

- Toute carte (sauf celles de la première pile) pointe vers une autre carte.
- Si a_i pointe vers a_j , alors :
 - $i > j$ (on pointe vers une carte déjà placée);
 - $a_i \geq a_j$ (si on avait pu placer a_i au-dessus de a_j , on l'aurait fait puisqu'on utilise la stratégie gloutonne).

Si l'on prend l'une des cartes situées sur la pile la plus à droite et que l'on suit les pointeurs jusqu'à arriver à la pile la plus à gauche, on obtient donc une sous-séquence croissante de longueur égale au nombre de piles.

Par maximalité de $l_{seq}(s)$, on a donc au plus $l_{seq}(s)$ piles pour la stratégie gloutonne, et en combinant avec la question précédente, la stratégie gloutonne utilise exactement $l_{seq}(s)$ piles.

► **Question 15** Il n'y a pas de difficulté particulière. Pour chaque carte, on parcourt les sommets de pile en cherchant le premier que l'on puisse utiliser puis l'on place la carte. Cela prend au pire un temps proportionnel au nombre de piles à cet instant, qui est majoré par $|s|$.

Il y a $|s|$ cartes, la complexité est donc bien un $O(|s|^2)$.

Ce grand- O est clairement optimal si l'on considère un paquet de départ croissant.

```
type config = int list array
let patience (cartes : int list) : config =
  let piles = Array.create (List.length cartes) [] in
  let rec empile cartes actuel =
    match cartes, piles.(actuel) with
    | [], _ -> piles
    | x :: xs, y :: ys when x >= y -> empile cartes (actuel + 1)
    | x :: xs, u -> piles.(actuel) <- x :: u; empile xs 0
  in
  empile cartes 0
```

► **Question 16** Les sommets des piles étant en ordre croissant, on peut procéder par dichotomie. La recherche de la pile est alors logarithmique en le nombre de piles et donc en $|s|$, ce qui, répété $|s|$ fois, donne une complexité totale en $O(|s| \cdot \ln |s|)$.

► **Question 17**

```
(* Version avec recherche dichotomique de la première pile légale. *)
let patience_opt (cartes : int list) : config =
  let n = List.length cartes in
  let piles = Array.create n [] in
  (* Renvoie le + petit i tq deb <= i <= fin et
     (carte < List.hd piles.(i)) OU piles.(i) = [] *)
  let rec num_pile carte deb fin =
    if fin = deb then
      fin
    else
      let mil = (deb + fin) / 2 in
      match piles.(mil) with
      | x :: xs when carte >= x -> num_pile carte (mil + 1) fin
      | _ -> num_pile carte deb mil in
  let rec empile cartes =
    match cartes with
    | [] -> piles
    | x :: xs -> let k = num_pile x 0 (n - 1) in
      piles.(k) <- x :: piles.(k);
      empile xs in
  empile cartes
```

► **Question 18** Il suffit de compter le nombre de piles utilisées (c'est-à-dire non vides). Une solution parmi d'autres :

```
(* nb_si_filtre : 'a array -> ('a -> bool) -> int
   Compte le nombre d'élément de t vérifiant le prédictat filtre
*)
let nb_si_filtre t filtre =
  let nb = ref 0 in
  for k = 0 to Array.length t - 1 do
    if filtre t.(k) then nb := !nb + 1
  done;
  !nb

let l_seq_patience_1 u =
  nb_si_filtre (patience u) (fun v -> v <> [])

let l_seq_patience u =
  nb_si_filtre (patience_opt u) (fun v -> v <> [])
```

► **Question 19** Il faut stocker les pointeurs imaginés plus haut. On n'empile donc plus des entiers, mais des cellules constituées d'un entier et d'un pointeur vers une autre cellule : autrement dit, des listes.

► **Question 20** Pour éviter d'avoir un cas particulier pour la première pile (pas de pointeur), on rajoute une pile fictive tout à gauche constituée uniquement d'un sommet égal à la liste vide (cette pile vaut donc `[]`). Ainsi, quand on rajoute un élément sur la « vraie » pile la plus à gauche, on peut le faire pointer vers le sommet de cette pile fictive. Il faut modifier un peu la dichotomie puisque la numérotation des piles va maintenant de 1 à n.

```
type pile_avec_pointeurs = int list list
type config_avec_pointeurs = pile_avec_pointeurs array

(* Renvoie le numéro de pile (entre deb et fin) sur lequel il
faut empiler carte. *)
let rec num_pile_pointeurs carte deb fin (piles : config_avec_pointeurs) =
  if fin = deb then
    fin
  else
    let mil = (deb + fin) / 2 in
    match List.hd piles.(mil) with
    | x :: xs when carte >= x -> num_pile_pointeurs carte (mil + 1) fin piles
    | _ -> num_pile_pointeurs carte deb mil piles

let rec empile_pointeurs cartes (piles: config_avec_pointeurs) : unit =
  match cartes with
  | [] -> ()
  | x :: xs ->
    let k = num_pile_pointeurs x 1 (Array.length piles) piles in
    piles.(k) <- (x :: List.hd piles.(k - 1)) :: piles.(k);
    empile_pointeurs xs piles

let sous_sequence_patience_1 cartes =
  let piles = Array.create (List.length cartes + 1) [] in
  let rec recuperation piles actuel k =
    match piles.(k) with
    | [] -> actuel
    | x :: xs -> recuperation piles x (k + 1) in
  recuperation cartes piles;
  List.rev (recuperation piles [] 1)
```

Ce code n'est pas très satisfaisant (en particulier tous les `List.hd`). On peut en fait faire plus simple en remarquant que les cartes qui ne sont pas au sommet d'une pile sont « mortes » : on n'aura plus jamais besoin de créer un pointeur vers une telle carte. On obtient le code suivant :

```
let rec num_pile carte deb fin piles =
  if fin = deb then
    fin
  else
    let mil = (deb + fin) / 2 in
    match piles.(mil) with
    | x :: xs when carte >= x -> num_pile carte (mil + 1) fin piles
    | _ -> num_pile carte deb mil piles

let rec empile cartes piles =
  match cartes with
  | [] -> ()
  | x :: xs -> let k = num_pile x 1 (Array.length piles) piles in
    piles.(k) <- x :: piles.(k - 1);
    empile xs piles

let sous_sequence_patience_2 cartes =
  let piles = Array.create (List.length cartes + 1) [] in
  let rec recuperation piles actuel k =
    match piles.(k) with
    | [] -> actuel
    | _ -> recuperation piles (piles.(k)) (k + 1) in
  recuperation cartes piles;
  List.rev (recuperation piles [] 1)
```

► **Question 21** Jouons une partie en utilisant la stratégie gloutonne avec ce paquet de $n^2 + 1$ cartes. Notons p la hauteur maximale des piles et q le nombre de piles.

- On a $pq \geq n^2 + 1$ (les cartes doivent rentrer dans un rectangle de taille p, q), et donc $\max(p, q) \geq n + 1$.
- Une pile, lue de bas en haut, fournit une sous-séquence décroissante (nous l'avons déjà remarqué plus haut), il y a donc une sous-séquence décroissante de longueur p .
- D'après la partie précédente, il y a une sous-séquence croissante de longueur q .

On a donc bien une sous-séquence monotone de longueur supérieure ou égale à $n + 1$.

GRAPHES EULÉRIENS

Définition XXXVII.I – Chemin eulérien, circuit eulérien

Soit $G = (V, E)$ un graphe non orienté.

- Un *chemin eulérien* de G est un chemin simple (c'est-à-dire constitué d'arêtes distinctes) reliant deux sommets de G et utilisant toutes les arêtes. Autrement dit, c'est un chemin simple de longueur $|E|$.
- Un *circuit eulérien* est un chemin eulérien dont les deux extrémités sont les mêmes. Autrement dit, c'est un cycle de longueur $|E|$.
- Un *graphe eulérien* est un graphe possédant un circuit eulérien.

► **Question 1** Montrer que le graphe suivant possède un chemin eulérien.

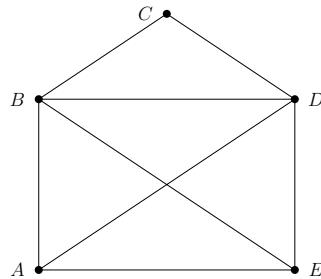


FIGURE XXXVII.1 – Le graphe G_1 .

► **Question 2** Montrer que le graphe suivant possède un circuit eulérien.

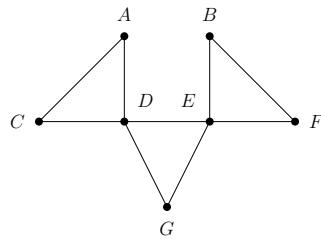


FIGURE XXXVII.2 – Le graphe G_2 .

I Condition nécessaire

► **Question 3** Soit G un graphe eulérien. Montrer que tous les sommets de G sont de degré pair.

► **Question 4** Énoncer une condition nécessaire similaire à celle de la question précédente pour qu'un graphe G possède un chemin eulérien.

2 Condition suffisante

► **Question 5** Soit G un graphe dont tous les sommets sont de degré pair. On considère le processus suivant :

- on part d'un sommet x quelconque ;
- on suit des arêtes à partir de x , sans les réutiliser, jusqu'à se retrouver bloqué. Ici, *bloqué* signifie que l'on est sur un sommet depuis lequel il n'y a plus d'arête disponible.

Montrer que le processus se termine nécessairement au sommet x .

► **Question 6** On suppose avoir construit un cycle en suivant le processus de la question précédent. Montrer que, si le graphe est connexe, on est forcément dans l'un des deux cas suivants :

- soit le cycle construit est un circuit eulérien ;
- soit il existe un sommet du cycle qui dispose encore d'une arête disponible.

► **Question 7** En déduire qu'un graphe connexe est eulérien si et seulement si tous ses sommets sont de degré pair.

► **Question 8** Énoncer et démontrer une propriété similaire pour l'existence d'un chemin eulérien.

► **Question 9** Le problème historique qui a donné naissance à la notion de graphe eulérien est celui des *ponts de Königsberg*. La ville de Königsberg¹ possédait à l'époque d'Euler sept ponts, disposés comme suit :

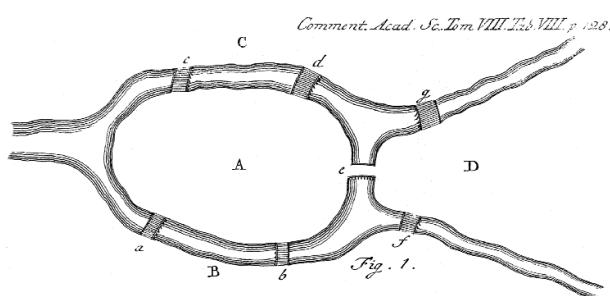


FIGURE XXXVII.3 – Les sept ponts de Königsberg.

Le problème est de savoir s'il est possible de traverser successivement tous ces ponts sans repasser deux fois par le même. Proposer une extension des résultats précédents aux *multigraphes* (graphes dans lesquels on peut avoir plusieurs arêtes entre deux sommets donnés) permettant de répondre à cette question.

3 Construction de chemins eulériens

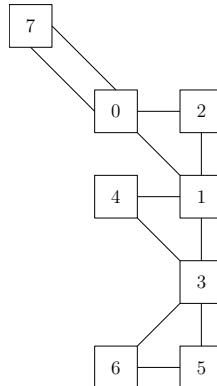
On s'intéresse dans cette partie à la construction effective d'un circuit eulérien dans un graphe. L'algorithme utilisé, dit *algorithme de Hierholzer*, suit essentiellement la démonstration faite plus haut pour la condition suffisante.

On maintient deux piles, une nommée *actuel* correspondant au chemin en cours d'exploration, et l'autre *euler* correspondant au circuit eulérien que l'on construit.

- On commence à un sommet x_0 quelconque.
- On suit des arêtes pour former un chemin, en supprimant au fur et à mesure les arêtes du graphe et en empilant les sommets visités sur *actuel*.
- Si à un moment on arrive sur un sommet ne disposant plus d'arête disponible, on dépile des sommets de *actuel* jusqu'à retomber sur un sommet qui dispose encore d'une arête. Ces sommets sont empilés sur *euler* au fur et à mesure.
- L'algorithme se termine quand *actuel* est vide : *euler* contient alors un circuit eulérien si le graphe de départ était eulérien.

1. Aujourd'hui Kaliningrad, en Russie.

► **Question 10** Simuler à la main l'exécution de l'algorithme sur le multigraphe suivant, en supposant que, quand il y a plusieurs arêtes x, y disponibles depuis un certain sommet x , on traite d'abord celle pour laquelle y est minimal. On le fera une fois en commençant par le sommet 0 et une fois en commençant par le sommet 4.

FIGURE XXXVII.4 – Le graphe G_3 .

Le fichier `stack.h` contient l'interface d'une structure de pile mutable, et le fichier `graph.h` l'interface d'une structure de graphe. On donne les garanties suivantes sur la complexité des fonctions² :

- toutes les opérations élémentaires sur les piles sont en temps constant, l'initialisation d'une pile en temps proportionnel à sa capacité;
- `build_graph`, avec $\text{nb_vertex} = n$ et $\text{nb_edges} = p$, est en $O(n + p)$;
- `get_edge`, `has_available_edge` et `delete_edge` sont en temps constant.

► **Question 11** Écrire une fonction lisant des données sur l'entrée standard au format suivant :

- la première ligne contient deux entiers n et p séparés par une espace : n sera le nombre de sommets, p le nombre d'arêtes;
- les p lignes suivantes contiennent chacune deux entiers de $[0 \dots n - 1]$ séparées par une espace : les deux sommets incidents à une arête.

La fonction renverra un tableau `edges` de longueur $2p$ tel que `edges[2 * i]` et `edges[2 * i + 1]` soient les deux sommets constituant l'arête i . Elle modifiera également les valeurs pointées par les arguments de sortie `nb_vertex` et `nb_edges`.

```
int *read_data(int *nb_vertex, int *nb_edges);
```

► **Question 12** Écrire une fonction `euler_tour` qui renvoie un circuit eulérien du graphe g (en supposant qu'un tel circuit existe), sous forme d'un pointeur vers une pile de sommets.

```
stack *euler_tour(graph g);
```

► **Question 13** Créer un fichier correspondant au graphe G_3 et vérifier que l'on obtient bien ce qui était prévu.

► **Question 14** Déterminer la complexité totale de l'algorithme (lecture des données, construction du graphe, construction du circuit eulérien).

► **Question 15** Si le graphe possède un chemin eulérien mais pas de circuit eulérien, est-il garanti que cet algorithme le trouve ? Si ce n'est pas le cas, indiquer la modification qu'il faudrait apporter pour traiter correctement ce cas.

2. Pour certaines fonctions, la réalité est un peu plus complexe mais cela n'affecte pas le résultat final.

Solutions

► Question 1

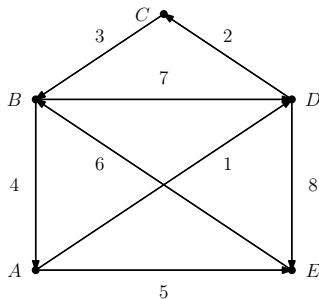


FIGURE XXXVII.5 – Un chemin eulérien dans G_1 .

► Question 2

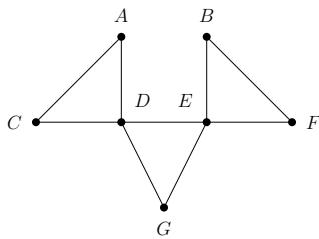


FIGURE XXXVII.6 – Un circuit eulérien dans G_2 .

► **Question 3** Considérons un cycle eulérien $x_0, x_1, \dots, x_p = x_0$ (avec p le nombre d'arêtes du graphe). Pour chaque occurrence d'un sommet x dans ce cycle, on a deux arêtes incidentes à x : une pour y entrer et une pour en sortir. De plus, le graphe n'ayant pas de boucle, on ne compte jamais une arête deux fois (il n'y a pas de $x - x$ dans le cycle). Le nombre d'arêtes incidentes à x présentes dans le cycle est donc pair, et comme toutes les arêtes du graphe sont dans le cycle une et une seule fois, le degré de x est pair. D'autre part, un sommet n'apparaissant pas dans le cycle est nécessairement de degré nul.

Dans un graphe eulérien, tous les sommets sont de degré pair.

► **Question 4** Le raisonnement précédent reste valable pour un chemin eulérien, sauf pour les deux extrémités. Si un graphe possède un chemin eulérien, il a au plus deux sommets de degré impair.

► **Question 5** Supposons qu'on finisse bloqué en un sommet $y \neq x$. Alors on est « entré » dans y une fois de plus qu'on en est « sorti », et l'on a ce faisant utilisé toutes les arêtes incidentes à y . Donc y est de degré impair, ce qui est absurde. On termine donc nécessairement en x .

► **Question 6** S'il n'y a plus d'arête disponible dans les sommets du cycle, cela signifie que ces sommets forment une composante connexe. Le graphe étant connexe, cela signifie que tous les sommets sont dans le cycle, et donc qu'on a épousé les arêtes du graphe. Ainsi, le cycle est un circuit eulérien.

► **Question 7** Soit G un graphe connexe dont tous les sommets sont de degré pair. On considère le processus suivant :

- on part d'un sommet v quelconque et l'on construit un cycle C comme décrit plus haut
- tant que C n'est pas un circuit eulérien :
 - on trouve un sommet x de C possédant encore une arête disponible
 - on construit un cycle C' autour de x (par le même processus, le graphe ayant toujours ses sommets de degré pair en tenant compte des arêtes retirées)
 - on remplace C par sa fusion avec C' : en écrivant $C : x = x_0, x_1, \dots, x_{k-1} = x$ et $C' : x = y_0, y_1, \dots, y_{l-1} = x$, on pose $C : x = x_0, x_1, \dots, x_{k-1}, y_1, \dots, y_{l-1} = x$.

À chaque étape C est un cycle simple, et le nombre d'arêtes disponibles décroît strictement. Le processus se termine donc avec un cycle simple C , qui est un circuit eulérien d'après la question précédente. On en déduit que la condition nécessaire trouvée en 3 est en fait suffisante :

un graphe connexe possède un circuit eulérien si et seulement si tous ses sommets sont de degré pair.

► **Question 8** Si le graphe ne possède pas de sommet de degré impair, il a un circuit (et donc un chemin) eulérien d'après la question précédente. S'il possède deux sommets de degré impair, on commence le processus précédent à l'un de ces deux sommets. On termine forcément à l'autre, puis on greffe des cycles supplémentaires sur ce chemin comme précédemment, jusqu'à épuisement des arêtes. Notons qu'il ne peut y avoir un unique sommet de degré impair. Combiné à la question 4, on peut conclure que

G possède un chemin eulérien si et seulement si il a au plus deux sommets de degré impair.

► **Question 9** Les résultats précédents s'étendent tels quels aux multigraphes (en définissant bien sûr le degré d'un nœud comme le nombre d'arêtes incidentes à ce nœud) : il n'y a rien à changer dans la démonstration. Le problème se réduit à la recherche d'un chemin eulérien dans le graphe ci-dessous :

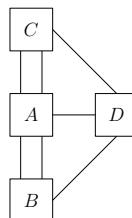


FIGURE XXXVII.7 – Le graphe des ponts de Königsberg.

Il y a quatre sommets de degré impair, un tel chemin n'existe donc pas.

► **Question 10**

En partant de 0

- On commence par empiler 0, 1, 2, 0, 7, 0 dans actuel.
- On dépile 0, 7, 0, 2 pour arriver en 1 qui a encore des arêtes disponibles.
- On a alors (sommet à droite) courant = 0, 1 et euler = 0, 7, 0, 2.
- On empile dans courant jusqu'à obtenir courant = 0, 1, 3, 4, 1.
- On dépile pour obtenir courant = 0, 1, 3 et euler = 0, 7, 0, 2, 1, 4.
- On empile jusqu'à courant = 0, 1, 3, 5, 6, 3.
- On dépile jusqu'à courant = \emptyset et euler = 0, 7, 0, 2, 1, 4, 3, 6, 5, 3, 1, 0

En partant de 4

- On empile 4, 1, 0, 2, 1, 3, 4.
- On dépile 4, euler = 4
- On empile 5, 6, 3, courant = 4, 1, 0, 2, 1, 3, 5, 6, 3.
- On dépile 3, 6, 5, 3, 1, 2. On a courant = 4, 1, 0 et euler = 4, 3, 6, 5, 3, 1, 2.
- On empile 7, 0, courant = 4, 1, 0, 7, 0.
- On dépile tout, euler = 4, 3, 6, 5, 3, 1, 2, 0, 7, 0, 1, 4.

► Question 11

```
int *read_data(int *nb_vertex, int *nb_edges){  
    scanf("%d %d", nb_vertex, nb_edges);  
    int *data = malloc(2 * *nb_edges * sizeof(int));  
    for (int i = 0; i < *nb_edges; i++) {  
        int x, y;  
        scanf("\n%d %d", &x, &y);  
        data[2 * i] = x;  
        data[2 * i + 1] = y;  
    }  
    return data;  
}
```

► Question 12 C'est une traduction directe de l'algorithme, les structures de données étant fournies.

```
stack *euler_tour(graph g){  
    int p = g.nb_edges;  
    stack *euler = stack_new(2 * p);  
    stack *current = stack_new(2 * p);  
    stack_push(current, 0);  
    while (!stack_is_empty(current)) {  
        int v = stack_peek(current);  
        if (has_available_edge(g, v)) {  
            edge e = get_edge(g, v);  
            stack_push(current, e.to);  
            delete_edge(g, e);  
        } else {  
            stack_pop(current);  
            stack_push(euler, v);  
        }  
    }  
    stack_free(current);  
    return euler;  
}
```

► Question 13 On peut écrire la fonction main suivante :

```
int main(void){  
    int nb_vertex, nb_edges;  
    int *data = read_data(&nb_vertex, &nb_edges);  
  
    graph g = build_graph(data, nb_vertex, nb_edges);  
    stack *tour = euler_tour(g);  
  
    while (!stack_is_empty(tour)) {  
        int v = stack_pop(tour);  
        printf("%d ", v);  
    }  
    printf("\n");  
  
    stack_free(tour);  
    graph_free(g);  
    return 0;  
}
```

Comme on affiche le circuit en le dépliant, on l'obtient « à l'envers » (ce qui ne serait gênant que si le graphe était orienté). On obtient bien :

```
$ ./euler < g3.txt
0 1 3 5 6 3 4 1 2 0 7 0
```

► **Question 14** Avec les garanties données par le sujet sur la complexité des différentes opérations, on obtient en notant n le nombre de sommets et p le nombre d'arêtes :

- lecture des données en $O(p)$;
- construction du graphe en $O(n + p)$;
- construction du parcours en $O(p)$ (le corps de la boucle est en temps constant, et l'on y passe deux fois pour chaque arête).

Au total, on a donc une complexité en $O(n + p)$.

► **Question 15** Pour un chemin eulérien, l'algorithme n'a aucune raison de fonctionner si l'on part d'un sommet quelconque (si le graphe ne possède pas de circuit eulérien). En revanche, si l'on fait en sorte de partir de l'un des deux sommets de degré impair, on va commencer par construire un chemin se terminant à l'autre sommet de degré impair, puis ensuite greffer des cycles sur ce chemin. Il suffit donc de commencer par identifier un sommet de degré impair et par commencer l'exploration à partir de ce sommet.

PARCOURS DE GRAPHES EN OCAML

Les exercices marqués d'une flèche ► doivent être vus comme faisant partie du cours.

I Parcours

Dans tout le sujet, les graphes sont supposés donnés sous forme d'un tableau de listes d'adjacence :

```
type sommet = int
type graphe = sommet list array
```

1.1 Parcours en profondeur

► Exercice XXXVIII.1 – Parcours en profondeur récursif

p. 690

Écrire une fonction `dfs` telle que `dfs pre post g x0` effectue un parcours en profondeur du graphe `g` à partir du sommet `x0`, en exécutant `pre x` à l'ouverture du sommet `x` et `post x` à la fermeture.

Il s'agit simplement de traduire le pseudo-code donné dans le cours en OCaml.

```
dfs : (sommet -> unit) -> (sommet -> unit) -> graphe -> sommet -> unit
```

Exercice XXXVIII.2 – Sur un exemple

p. 690

On considère les graphes suivants :

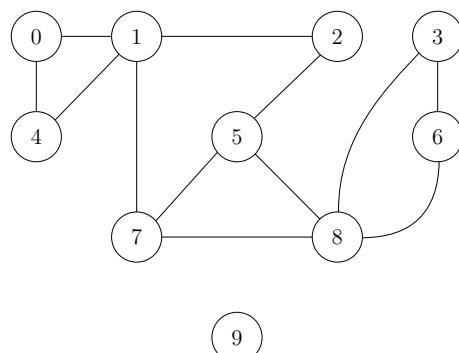
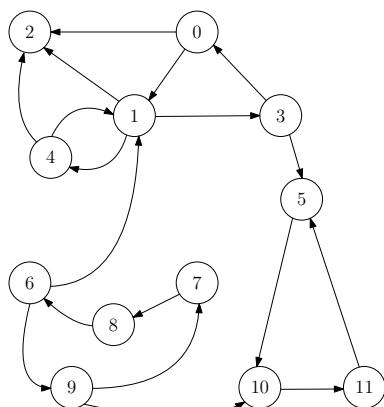


FIGURE XXXVIII.2 – Le graphe `g1`.

FIGURE XXXVIII.1 – Le graphe `g0`

Ces graphes sont stockés de façon à ce que les listes d'adjacence soient *en ordre croissant*.
On définit :

```
let ouvre x = Printf.printf "Ouverture %d\n" x
let ferme x = Printf.printf "Fermeture %d\n" x
```

Déterminer à la main l'affichage produit par `dfs ouvre ferme g0 0` et par `dfs ouvre ferme g1 5` puis vérifier sur l'ordinateur.

1.2 Parcours en largeur

► Exercice XXXVIII.3 – Parcours en largeur à l'aide d'une file

p. 691

1. Écrire une fonction `bfs` effectuant un parcours en largeur. On traduira le pseudo-code du cours en utilisant le module `Queue`; le premier argument de `bfs` correspond à la fonction `TRAITEMENT` et doit être appelé au moment où l'on extrait un sommet de la file.

```
Queue.create : unit -> 'a Queue.t (* crée une file vide *)
Queue.is_empty : 'a Queue.t -> bool
Queue.pop : 'a Queue.t -> 'a
Queue.push : 'a -> 'a Queue.t -> unit
```

```
bfs : (sommet -> unit) -> graphe -> sommet -> unit
```

2. Vérifier que `bfs ouvre g0 0` et `bfs ouvre g1 5` donnent bien ce que vous pensiez.
 3. Si l'on ne souhaite pas utiliser le module `Queue`, comment peut-on réaliser de manière efficace une file impérative? fonctionnelle? On ne demande pas d'implémenter ces structures mais simplement de se remémorer les techniques vues depuis le début de l'année.

► Exercice XXXVIII.4 – Parcours en largeur avec frontière explicite

p. 691

Écrire une fonction de parcours en largeur utilisant le principe suivant :

- on a toujours un ensemble *vis* codé par un tableau de booléens;
- on utilise deux listes appelées `frontiere` et `nouveaux`;
- au début d'une itération, `frontiere` contient tous les sommets situés à une certaine distance `k` du sommet initial et `nouveaux` est vide;
- on parcourt `frontiere`, et pour chaque sommet on rajoute ses voisins non explorés à `nouveaux`;
- à la fin de ce parcours, on passe à l'itération suivante avec la liste `nouveaux` qui devient la nouvelle liste `frontiere`.

Remarque

La manière la plus simple de coder cette fonction en OCaml est purement fonctionnelle.

2 Accessibilité, composantes connexes

Exercice XXXVIII.5 – Accessibilité

p. 692

1. Écrire une fonction `accessible` telle que l'appel `accessible g x y` détermine si `y` est accessible depuis `x` dans le graphe (*a priori* orienté) `g`, à l'aide d'un parcours en profondeur.

```
accessible : graphe -> sommet -> sommet -> bool
```

2. Modifier cette fonction pour qu'elle réponde dès que possible.

Remarque

Le plus simple est d'utiliser une exception.

► Exercice XXXVIII.6 – Composantes connexes

1. Écrire une fonction `tab_composantes : graphe -> int array` qui prend en entrée un graphe *supposé non orienté* et renvoie un tableau `t` tel que $t_i = t_j$ si et seulement si les sommets x_i et x_j sont dans la même composante connexe du graphe.
2. Écrire une fonction `listes_composantes : graphe -> sommet list list` qui renvoie les composantes connexes sous forme de liste de listes.
On ne réutilisera pas la fonction précédente et on n'hésitera pas à passer par des list ref.

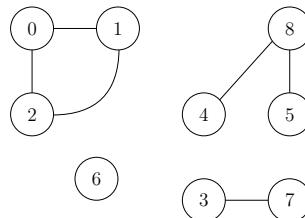


FIGURE XXXVIII.3 – Graphe g2.

```

utop[68]> liste_composantes g2;;
- : sommet list list = [[6]; [5; 8; 4]; [7; 3]; [2; 1; 0]]
utop[69]> tab_composantes g2;;
- : sommet array = [|0; 0; 0; 3; 4; 4; 6; 3; 4|]
  
```

2.1 Construction de l'arborescence d'un parcours

Il est très souvent utile de générer explicitement l'arborescence (l'arbre enraciné) associé à un parcours de graphe : en particulier, cela permet ensuite de reconstituer facilement des chemins. La manière la plus simple de procéder est généralement de stocker l'arbre *orienté des feuilles vers la racine* : cela peut se faire facilement à l'aide d'un tableau de taille n (nombre de sommets) :

- si le sommet i n'est pas dans l'arborescence, la case i du tableau contiendra une valeur particulière (`None`, ou -1 , ou...);
- si le sommet i est un nœud autre que la racine de l'arbre, la case i contiendra l'indice du parent de i ;
- si le sommet i est la racine de l'arbre, la case i contiendra i .

► Exercice XXXVIII.7 – Arbre de parcours, reconstitution de chemins

On reprend les types utilisés plus haut :

```

type sommet = int
type graphe = sommet list array
  
```

1. Écrire une fonction `arbre_dfs` effectuant un parcours en profondeur d'un graphe G à partir d'un sommet x_0 passé en argument, et renvoyant un tableau `t` codant l'arbre de parcours en profondeur associé de la manière suivante :
 - `t` est de longueur $|V|$;
 - `t.(x0) = x0`;
 - si x n'est pas accessible depuis x_0 , `t.(x) = -1`;
 - si x a été exploré depuis y , `t.(x) = y`.

```

arbre_dfs : graphe -> sommet -> sommet array
  
```

2. Écrire une fonction `arbre_bfs` ayant les mêmes spécifications que `arbre_dfs` (sauf bien sûr que l'on renverra un arbre de parcours en largeur).

3. Écrire une fonction `chemin` qui prend en entrée un arbre enraciné en x_0 comme ci-dessus et un sommet x , et renvoie un chemin `Some [x0; ... ; x]` s'il en existe un, `None` sinon.

Remarque

Il est inutile de passer x_0 en argument : c'est le seul indice pour lequel $t.(i) = i$.

```
chemin : sommet array -> sommet -> sommet list option
```

4. Que peut-on dire du chemin renvoyé par la fonction `chemin` si l'arbre utilisé est issu de la fonction `arbre_bfs` ?

3 Variantes des parcours

Exercice XXXVIII.8 – Parcours en profondeur itératif

p. 694

1. Compléter cette fonction pour qu'elle réalise un parcours en profondeur de g à partir de i :

```
let dfs_pile pre g i =
  let visites = Array.make g.nb_sommets false in
  let rec traite pile = match pile with
    | [] -> ...
    | x :: xs when not visites.(x) -> ...
    | x :: xs -> ... in
  ...
```

La fonction `traite` devra être récursive terminale. On évitera d'utiliser @ qui n'est pas terminal^a, mais on n'hésitera pas à faire appel à la fonction `List.rev_append`, qui l'est.

Il n'y a pas cette fois d'argument `post`, car il n'y a pas de moyen évident de savoir quand le traitement d'un noeud est terminé.

2. Décrire l'évolution de `pile` au cours de l'appel `dfs_pile g1 0`.

3. Justifier que la taille de `pile` (et donc la complexité spatiale de `dfs_pile`) peut être de l'ordre de $|E|$.

Remarque

Le remède semble évident mais ne l'est pas tant que ça : cf exercices XXXVIII.9 et 15.23.

4. Une fonction récursive terminale peut très facilement être transformée en une fonction non récursive^b. Il suffit ici d'utiliser une pile impérative à la place d'une pile fonctionnelle (*i.e.* d'une liste).

On pourrait réaliser une pile impérative à l'aide d'une `list ref`, mais le plus simple est d'utiliser le module `Stack` :

```
Stack.create : unit -> 'a Stack.t (* crée une pile vide *)
Stack.is_empty : 'a Stack.t -> bool
Stack.pop : 'a Stack.t -> 'a
Stack.push : 'a -> 'a Stack.t -> unit
```

Écrire une fonction `dfs_it` ayant les mêmes spécifications (et donc les mêmes arguments) que `dfs_pile` mais purement itérative.

a. On pourrait presque systématiquement le faire puisque la profondeur de récursion serait majorée par le degré maximal du graphe, qui est rarement gigantesque.

b. C'est tellement facile que le compilateur le fait automatiquement.

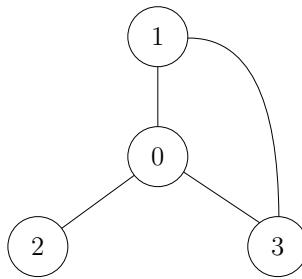
Exercice XXXVIII.9 – Parcours en pseudo-profondeur

p. 696

1. Modifier la fonction `dfs_it` de l'exercice XXXVIII.8 de manière à ce qu'un sommet soit rajouté au plus une fois sur la pile. On appellera la fonction obtenue `pseudo_dfs`. Que remarque-t-on par rapport au parcours en largeur écrit à l'exercice XXXVIII.3 ?

Ce parcours est très couramment utilisé car il est rapide, peu gourmand en mémoire, et ne risque pas de résulter en un `stack overflow`. Cependant, il ne s'agit pas d'un vrai parcours en profondeur, comme le montrent les questions suivantes. Souvent, on souhaite juste parcourir le graphe et cela ne nous dérange nullement; parfois, les propriétés du parcours en profondeur sont cruciales et le parcours en « pseudo-profondeur » ne convient pas.

2. On fait un parcours du graphe représenté ci-dessous à partir du nœud 0 (et l'on suppose que les listes de voisins sont stockées dans l'ordre croissant). Dans quel ordre les nœuds sont-ils traités (un nœud x est traité quand on appelle `pre x`) par `pseudo_dfs`? Est-ce le même que pour `dfs_pile`? que pour `dfs`?



3. On se limite aux graphes non orientés pour simplifier. Montrer que l'arbre associé au parcours en pseudo-profondeur d'un graphe G depuis un sommet x est un arbre de parcours en profondeur si et seulement si la composante connexe de x dans G est un arbre.

Solutions

Correction de l'exercice XXXVIII.1 page 685

Il faut écrire des variantes de cette fonction jusqu'à ce que ça vous semble aussi naturel que de calculer la longueur d'une liste.

```
let dfs pre post g i =
  let n = Array.length g in
  let visites = Array.make n false in
  let rec visite j =
    if not visites.(j) then begin
      visites.(j) <- true;
      pre j;
      List.iter visite g.(j);
      post j
    end in
  visite i
```

Correction de l'exercice XXXVIII.2 page 685

dfs ouvre ferme g0 0

Ouverture 0
Ouverture 1
Ouverture 2
Fermeture 2
Ouverture 3
Ouverture 5
Ouverture 10
Ouverture 11

Fermeture 11
Fermeture 10
Fermeture 5
Fermeture 3
Ouverture 4
Fermeture 4
Fermeture 1
Fermeture 0

dfs ouvre ferme g1 5

Ouverture 5
Ouverture 2
Ouverture 1
Ouverture 0
Ouverture 4
Fermeture 4
Fermeture 0
Ouverture 7
Ouverture 8

Ouverture 3
Ouverture 6
Fermeture 6
Fermeture 3
Fermeture 8
Fermeture 7
Fermeture 1
Fermeture 2
Fermeture 5

Correction de l'exercice XXXVIII.3 page 686

1.

```

let bfs pre g initial =
  let vus = Array.make (Array.length g) false in
  let file = Queue.create () in
  let ajoute x =
    if not vus.(x) then begin
      pre x;
      vus.(x) <- true;
      Queue.push x file
    end in
  ajoute initial;
  while not (Queue.is_empty file) do
    let x = Queue.pop file in
    List.iter ajoute g.(x);
  done

```

2. Pour `bfs ouvre g0 0`, on doit obtenir dans l'ordre 0,1,2,3,4,5,10,11. Pour `bfs ouvre g1 5`, on doit obtenir 5,2,7,8,1,3,6,0,4.
3. Une file impérative (de capacité fixée à la construction) peut être réalisée par un tableau circulaire, par une structure contenant un pointeur vers chaque extrémité, d'une liste simplement chaînée mutable, par une liste doublement chaînée... Une file fonctionnelle peut être réalisée par un couple de listes.

Correction de l'exercice XXXVIII.4 page 686

```

let bfs_avec_frontiere pre g x0 =
  let vus = Array.make (Array.length g) false in
  vus.(x0) <- true;
  let rec ajoute voisins nouveaux =
    match voisins with
    | [] -> nouveaux
    | x :: xs when vus.(x) -> ajoute xs nouveaux
    | x :: xs ->
        vus.(x) <- true;
        pre x;
        ajoute xs (x :: nouveaux) in
  let rec loop frontiere nouveaux =
    match (frontiere, nouveaux) with
    | [], [] -> ()
    | [], _ -> loop nouveaux []
    | x :: xs, _ -> loop xs (ajoute g.(x) nouveaux) in
  loop [x0] []

```

L'appel `ajoute voisins nouveaux` marque les sommets non encore visités de `voisins` et les « ajoute » à `nouveaux` (c'est-à-dire renvoie la concaténation de ces sommets non visités avec `nouveaux`).

Dans la boucle principale (fonction `loop`), la liste `frontiere` représente les sommets vus mais pas encore traités à distance k de $x0$ et la liste `nouveaux` les sommets vus situés à distance $k+1$.

Correction de l'exercice XXXVIII.5 page 686

1.

```
let accessible g i j =
  let vus = Array.make (Array.length g) false in
  let rec explore v =
    if not vus.(v) then begin
      vus.(v) <- true;
      List.iter explore g.(v)
    end in
  explore i;
  vus.(j)
```

2. Pour s'arrêter dès que possible, le plus simple est d'utiliser une exception :

```
exception Trouve

let accessible_exn g i j =
  let vus = Array.make (Array.length g) false in
  let rec explore v =
    if v = j then raise Trouve;
    if not vus.(v) then begin
      vus.(v) <- true;
      List.iter explore g.(v)
    end in
  try
    explore i;
    false
  with
    | Trouve -> true
```

On peut bien sûr faire sans, comme dans la version ci-dessous. Deux remarques qui peuvent aider à la compréhension de ce code :

- on utilise de manière cruciale le fait que la branche droite d'un « ou » n'est évaluée que si la branche gauche est fausse – au besoin, vous pouvez ré-écrire le code avec des `if..then..else` pour mieux comprendre ce qui se passe;
- les fonctions `cherche` et `traite` sont *mutuellement récursives*, elles doivent donc être définies simultanément par un `let rec cherche v = ... and traite =`

```
let accessible_bis g i j =
  let vus = Array.make (Array.length g) false in
  let rec cherche v =
    (v = j) || (vus.(v) <- true; traite g.(v))
  and traite = function
    | [] -> false
    | x :: xs when vus.(x) -> traite xs
    | x :: xs -> cherche x || traite xs in
  cherche i
```

Correction de l'exercice XXXVIII.6 page 687

```
1. let tab_composantes graphe =
  let n = Array.length graphe in
  let t = Array.make n (-1) in
  let rec assigne i sommet =
    if t.(sommet) = -1 then begin
      t.(sommet) <- i;
      List.iter (assigne i) graphe.(sommet)
    end in
  for i = 0 to n - 1 do
    assigne i i (* sans effet si i est déjà dans une composante *)
  done;
  t
```

L'appel `assigne i` effectue un parcours en profondeur à partir du sommet `s`, en affectant tous les sommets rencontrés à la composante `i`.

2.

```
let liste_composantes graphe =
  let n = Array.length graphe in
  let vus = Array.make n false in
  (* la liste des composantes *)
  let composantes = ref [] in
  (* la composante actuelle *)
  let c = ref [] in
  let rec explore i =
    if not vus.(i) then begin
      vus.(i) <- true;
      c := i :: !c;
      List.iter explore graphe.(i)
    end in
  for i = 0 to n - 1 do
    if not vus.(i) then begin
      c := [];
      explore i;
      composantes := !c :: !composantes
    end
  done;
  !composantes
```

Correction de l'exercice XXXVIII.7 page 687

1.

```
let arbre_dfs g x0 =
  let parent = Array.make (Array.length g) (-1) in
  let rec explore u =
    let f v =
      if parent.(v) = -1 then (parent.(v) <- u; explore v) in
      List.iter f g.(u) in
    parent.(x0) <- x0;
    explore x0;
    parent
```

2.

```
let arbre_bfs g x0 =
  let parent = Array.make (Array.length g) (-1) in
  let file = Queue.create () in
  let ajoute pere x =
    if parent.(x) = -1 then begin
      parent.(x) <- pere;
      Queue.push x file
    end in
  ajoute x0 x0;
  while not (Queue.is_empty file) do
    let x = Queue.pop file in
    List.iter (ajoute x) g.(x);
  done;
  parent
```

3.

```
let chemin parent i =
  let rec aux i acc =
    if parent.(i) = i then i :: acc
    else aux parent.(i) (i :: acc) in
  if parent.(i) = -1 then None
  else Some (aux i [])
```

4. Si l'on utilise un arbre de parcours en largeur, le chemin renvoyé sera systématiquement un plus court chemin.

Correction de l'exercice XXXVIII.8 page 688

1.

```
let dfs_pile pre g i =
  let visites = Array.make (Array.length g) false in
  let rec traite = function
    | [] -> ()
    | x :: xs when not visites.(x) ->
      visites.(x) <- true;
      pre x;
      traite (List.rev_append g.(x) xs)
    | x :: xs -> traite xs in
  traite [i]
```

2. Pour bien pouvoir observer le comportement de la fonction, on affiche un message "Traitement de ..." quand on dépile un sommet pour la première fois et un message "Déjà traité..." quand on dépile un sommet déjà traité. On affiche la pile à chaque fois qu'on y ajoute des sommets.

```

let dfs_pile_avec_affichage g i =
  let visites = Array.make (Array.length g) false in
  let rec traite = function
    | [] -> ()
    | x :: xs when not visites.(x) ->
      visites.(x) <- true;
      let nv_pile = List.rev_append g.(x) xs in
      Printf.printf "Traitement de %d\n" x;
      Printf.printf "État de la pile : ";
      affiche nv_pile;
      traite (List.rev_append g.(x) xs)
    | x :: xs -> Printf.printf "Déjà traité : %d\n" x; traite xs in
      traite [i]
  
```

```
dfs_pile_avec_affichage g1 0
```

```

Traitement de 0
État de la pile : 4 1
Traitement de 4
État de la pile : 1 0 1
Traitement de 1
État de la pile : 7 4 2 0 0 1
Traitement de 7
État de la pile : 8 5 1 4 2 0 0
↪ 1
Traitement de 8
État de la pile : 7 6 5 3 5 1 4
↪ 2 0 0 1
Déjà traité : 7
Traitement de 6
État de la pile : 8 3 5 3 5 1 4
↪ 2 0 0 1
Déjà traité : 8
Traitement de 3
État de la pile : 8 6 5 3 5 1 4
↪ 2 0 0 1
Déjà traité : 8
  
```

```

Déjà traité : 6
Traitement de 5
État de la pile : 8 7 2 3 5 1 4
↪ 2 0 0 1
Déjà traité : 8
Déjà traité : 7
Traitement de 2
État de la pile : 5 1 3 5 1 4 2
↪ 0 0 1
Déjà traité : 5
Déjà traité : 1
Déjà traité : 3
Déjà traité : 5
Déjà traité : 1
Déjà traité : 4
Déjà traité : 2
Déjà traité : 0
Déjà traité : 0
Déjà traité : 1
  
```

3. Imaginons qu'on parcourt un graphe complet à n sommets, avec des listes d'adjacence croissantes. On note x_1, \dots, x_n les sommets dans l'ordre de leur traitement (*i.e.* de leur ajout à vus). Quand on traite x_i , on ajoute ses $n - 1$ voisins à la pile ; avant de traiter x_{i+1} , on aura éliminé au plus $i - 1$ sommets de la pile (les sommets x_1, \dots, x_{i-1} qui ont déjà été traités). Juste après avoir traité x_n , la pile contient donc au moins $n(n - 1) - \sum_{i=1}^n (i - 1) = \frac{n(n-1)}{2}$ sommets. La complexité spatiale peut donc être de l'ordre de n^2 (et pas plus, puisque la complexité temporelle est en $O(n + p) = O(n^2)$).

Remarque

Le fait que la complexité *temporelle* soit en n^2 pour un graphe complet est normal (ce serait le cas avec n'importe quel parcours, car la taille du graphe est de cet ordre). La complexité *spatiale*, en revanche, serait en $O(n)$ pour un parcours en profondeur récursif (ou pour un parcours en largeur).

- 4.

```

let dfs_it pre g i =
  let visites = Array.make (Array.length g) false in
  let p = Stack.create () in
  Stack.push i p;
  while not (Stack.is_empty p) do
    let x = Stack.pop p in
    if not visites.(x) then begin
      visites.(x) <- true;
      pre x;
      List.iter (fun v -> Stack.push v p) g.(x);
    end
  done

```

Correction de l'exercice XXXVIII.9 page 689

- On obtient exactement la même fonction qu'à l'exercice XXXVIII.3, sauf que l'on a remplacé la file par une pile.

```

let pseudo_dfs pre g x0 =
  let vus = Array.make (Array.length g) false in
  let pile = Stack.create () in
  let rec ajoute x =
    if not vus.(x) then begin
      vus.(x) <- true;
      Stack.push x pile
    end in
  ajoute x0;
  while not (Stack.is_empty pile) do
    let x = Stack.pop pile in
    pre x;
    List.iter ajoute g.(x)
  done

```

- Pour pseudo_dfs, les sommets seront traités dans l'ordre 0, 3, 2, 1 car 1 n'est pas rajouté au sommet de la pile quand on traite 3 (il a déjà été marqué pendant le traitement de 0). Cet ordre est impossible pour un parcours en profondeur, qui impose de terminer l'exploration à partir de 3 avant de commencer celle à partir de 2.
- Supposons que la composante connexe de x ne soit pas un arbre. Dans ce cas, tout arbre de parcours en profondeur à partir de x doit nécessairement comporter au moins un arc arrière reliant un sommet t à l'un de ses ancêtres y (autre que son père z). Il est impossible d'obtenir un tel arbre avec pseudo_dfs : en effet, t aurait été marqué lors du traitement de y (au plus tard), et n'aurait donc pas été ajouté à la pile lors du traitement de z.

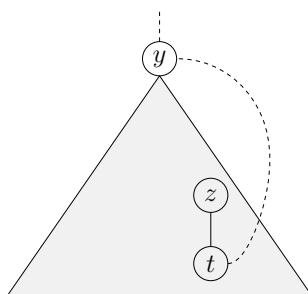


FIGURE XXXVIII.5 – Arc arrière, impossible avec un parcours en pseudo-profondeur.

CLÔTURE TRANSITIVE

Ce sujet est directement adapté d'un problème posé au concours Mines-Ponts en 2014 (reformulé de manière parfaitement équivalente en termes de graphes, et traduit en C.)

On considère un graphe orienté $G = (V, E)$, et l'on note $n = |V|$ son nombre de sommets. On se fixe une numérotation x_0, \dots, x_{n-1} des sommets.

- Pour $x, y \in V$, on note $x \Rightarrow_k y$ s'il existe $k + 1$ sommets y_0, \dots, y_k tels que :
 - $y_0 = x$;
 - $y_k = y$;
 - $(y_i, y_{i+1}) \in E$ pour tout $i \in [0 \dots k - 1]$.
 On a donc $x \Rightarrow_0 y$ si et seulement si $x = y$.
- On suppose que pour tout sommet x , la boucle (x, x) fait partie de l'ensemble E des arcs. On a donc également $x \Rightarrow_1 x$ pour tout sommet x .
- On note $x \Rightarrow y$ s'il existe un entier k tel que $x \Rightarrow_k y$.

Exemples On se référera aux deux graphes ci-dessous dans le problème :

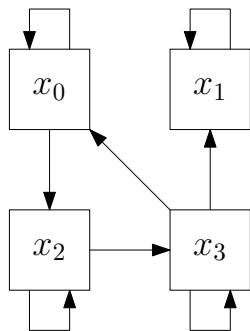


FIGURE XXXIX.1 – Le graphe G_1 .

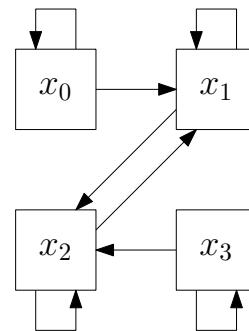


FIGURE XXXIX.2 – Le graphe G_2 .

- **Question 1** Soient $x, y \in V$ et $h \leq k$ deux entiers naturels. Montrer que si $x \Rightarrow_h y$, alors $x \Rightarrow_k y$.
- **Question 2** Soient $x, y \in V$. Montrer que l'on a $x \Rightarrow y$ si et seulement si $x \Rightarrow_{n-1} y$.

Matrice booléenne Une matrice booléenne est une matrice dont les coefficients sont à valeurs dans {vrai, faux}. Le produit de deux matrices booléennes s'obtient selon la formule habituelle du produit matriciel, en prenant pour somme de deux valeurs booléennes le « ou logique » (noté \vee) et pour produit de deux valeurs booléennes le « et logique » (noté \wedge). Dans toute la suite, le produit matriciel, et les puissances d'une matrice, sont à comprendre avec cette définition.

On a par exemple :

$$\begin{pmatrix} \text{vrai} & \text{vrai} \\ \text{faux} & \text{vrai} \end{pmatrix} \cdot \begin{pmatrix} \text{faux} & \text{vrai} \\ \text{vrai} & \text{faux} \end{pmatrix} = \begin{pmatrix} \text{vrai} & \text{vrai} \\ \text{vrai} & \text{faux} \end{pmatrix}$$

Programmation En C, une matrice booléenne M de dimensions (n, p) sera représentée par un M de type `bool**`. M pointera vers un bloc alloué de n pointeurs de type `bool*` : $M[i]$ sera un pointeur vers un bloc alloué de p booléens correspondant à la ligne i de la matrice (les lignes étant numérotées de 0 à $n - 1$).

► **Question 3** Écrire une fonction `matrix_new` renvoyant une nouvelle matrice de la taille spécifiée, initialisée à `false`.

```
bool **matrix_new(int n, int p);
```

► **Question 4** Écrire une fonction `matrix_free` qui libère toute la mémoire associée à une matrice. On pourra supposer que la matrice a été créée par un appel à `matrix_new` (on a ensuite pu modifier ses coefficients, mais pas les pointeurs). Seul le nombre de lignes est pris en paramètre, le nombre de colonnes étant superflu.

```
void matrix_free(bool **m, int n);
```

► **Question 5** Écrire une fonction `identity` renvoyant la matrice carrée de taille n contenant des `true` sur la diagonale et des `false` ailleurs.

```
bool **identity(int n);
```

► **Question 6** Écrire une fonction `product` renvoyant le produit des deux matrices passées en argument.

```
bool **product(bool **a, bool **b, int n, int p, int q);
```

Préconditions : on pourra supposer (sans le vérifier) que a est de dimensions (n, p) et b de dimensions (p, q) (et que tous ces entiers sont strictement positifs).

On associe à un graphe G sa matrice d'adjacence, vue comme une matrice booléenne, que l'on notera A . On a donc $A[i, j]$ qui vaut vrai si et seulement si $x_i \Rightarrow_1 x_j$ (on rappelle que $x_i \Rightarrow_1 x_i$ pour tout i).

► **Question 7** Montrer que pour tous $i, j \in [0 \dots n - 1]$ et pour tout $k > 0$, on a $A^k[i, j] = \text{vrai}$ si et seulement si $x_i \Rightarrow_k x_j$.

► **Question 8** Montrer que pour $k \geq n - 1$, on a $A^k = A^{n-1}$.

Clôture transitive On appelle *clôture transitive* de la matrice A la matrice $CT(A) = A^{n-1}$.

► **Question 9** Écrire une fonction `closure` qui prend en entrée une matrice carrée A de taille n et renvoie sa clôture transitive, calculée à l'aide de multiplications de matrices.

```
bool **closure(bool **a, int n);
```

► **Question 10** Déterminer la complexité de la fonction `closure`.

► **Question 11** Écrire une fonction `accessible` qui prend en entrée une matrice carrée A de taille n et un entier $i \in [0 \dots n - 1]$ et renvoie un pointeur `arr` vers un bloc alloué de n booléens tel que $arr[j]$ soit égal à `true` si $x_i \Rightarrow x_j$, `false` sinon.

```
bool *accessible(bool **a, int n, int i);
```

On exige une complexité temporelle en $O(n^2)$, que l'on justifiera.

- **Question 12** Écrire une nouvelle version de la fonction `closure` ayant une complexité temporelle en $O(n^3)$.

```
bool **closure(bool **a, int n);
```

Axiome On dit qu'un sommet x du graphe est un *axiome* s'il possède la propriété suivante : pour tout sommet y tel que $y \Rightarrow x$, on a $x \Rightarrow y$.

- **Question 13** Donner tous les axiomes du graphe G_1 .

- **Question 14** Donner tous les axiomes du graphe G_2 .

- **Question 15** Écrire une fonction `is_axiom` qui prend en entrée la matrice $B = CT(A)$ et un sommet i , et indique si ce sommet est un axiome.

```
bool is_axiom(bool **b, int n, int i);
```

Suite unidirectionnelle de sommets On appelle *suite unidirectionnelle de sommets* une suite finie y_0, \dots, y_h (avec $h \in \mathbb{N}$) de sommets vérifiant :

- pour $i \in [0, h - 1]$, $y_i \Rightarrow y_{i+1}$;
- pour $i \in [0, h - 1]$, $y_{i+1} \not\Rightarrow y_i$ (y_i n'est pas accessible depuis y_i).

- **Question 16** Montrer que les sommets d'une suite unidirectionnelle sont deux à deux distincts.

- **Question 17** Soit y un sommet. Montrer qu'il existe un axiome x tel que $x \Rightarrow y$.

- **Question 18** Donner les composantes fortement connexes du graphe G_2 .

- **Question 19** On considère une composante fortement connexe C contenant un axiome. Montrer que tous les sommets de C sont des axiomes.

Composante source On dit qu'une composante fortement connexe est une *composante source* si elle contient un axiome (ce qui revient à dire que tous ses éléments sont des axiomes, d'après la question précédente).

Système d'axiomes On dit qu'une partie X de V est un *système d'axiomes* si, pour tout sommet $y \in V$, il existe un sommet $x \in X$ tel que $x \Rightarrow y$.

- **Question 20** Montrer qu'on obtient un système d'axiomes de cardinal minimum en choisissant un et un seul sommet dans chacune des composantes source.

- **Question 21** Écrire une fonction `axiom_system` prenant en entrée la matrice $B = CT(A)$ et renvoyant un système d'axiomes de cardinal minimum. On renverra ce système sous forme d'un bloc alloué s de n booléens, tel que $s[i]$ soit vrai si et seulement si le sommet i fait partie du système d'axiomes.

```
bool *axiom_system(bool **b, int n);
```

Solutions

► **Question 1** On peut compléter le chemin de longueur h dont on dispose par $k - h \geq 0$ arcs $y \rightarrow y$. On obtient ainsi $x \Rightarrow_k y$.

► **Question 2** Si $x \Rightarrow_{n-1} y$, on a bien sûr $x \Rightarrow y$.

Si $x \Rightarrow y$, considérons le plus petit h tel que $x \Rightarrow_h y$.

- Si $h \leq n - 1$, alors on conclut par la question précédente.
- si $h \geq n$, alors on remarque que le chemin fait intervenir $h + 1 \geq n + 1$ sommets. Comme $|V| = n$, il est nécessairement de la forme $x = y_0 \rightarrow \dots y_j \rightarrow \dots y_k \rightarrow \dots y_h = y$ avec $y_j = y_k$ et $j < k$. Mais alors $x = y_0 \rightarrow \dots y_j \rightarrow y_{k+1} \rightarrow \dots y_h = y$ est un chemin de longueur $h - (k - j) < h$ de x à y , ce qui contredit la minimalité de h .

Donc $x \Rightarrow y$ si et seulement si $x \Rightarrow_{n-1} y$.

► **Question 3**

```
bool **matrix_new(int n, int p){  
    bool **m = malloc(n * sizeof(bool*));  
    for (int i = 0; i < n; i++){  
        m[i] = malloc(p * sizeof(bool));  
        for (int j = 0; j < p; j++) {  
            m[i][j] = false;  
        }  
    }  
    return m;  
}
```

► **Question 4**

```
void matrix_free(bool **m, int n){  
    for (int i = 0; i < n; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

► **Question 5**

```
bool **identity(int n){  
    bool **m = matrix_new(n, n);  
    for (int i = 0; i < n; i++) {  
        m[i][i] = true;  
    }  
    return m;  
}
```

► Question 6

```

bool **product(bool **a, bool **b, int n, int p, int q){
    bool **c = matrix_new(n, q);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < q; j++) {
            for (int k = 0; k < p; k++) {
                c[i][j] = c[i][j] || (a[i][k] && b[k][j]);
            }
        }
    }
    return c;
}

```

► Question 7 On procède par récurrence sur k .

- Pour $k = 1$, la propriété demandée est exactement la définition de A .
- On suppose la propriété vérifiée pour $k \geq 1$. On a (pour tout i, j) :

$$x_i \Rightarrow_{k+1} x_j \text{ssi } \exists 0 \leq i_1, \dots, i_{k-1} < n, x_i \Rightarrow_1 x_{i_1} \dots \Rightarrow_1 x_{i_{k-1}} \Rightarrow_1 x_j$$

$$\text{ssi } \exists 0 \leq l < n, x_i \Rightarrow_k x_l \wedge x_l \Rightarrow_1 x_j$$

$$\text{ssi } \exists 0 \leq l < n, A^k[i, l] \wedge A[l, j]$$

hypothèse de récurrence

$$\text{ssi } \bigvee_{l=0}^{n-1} (A^k[i, l] \wedge A[l, j])$$

$$\text{ssi } A^{k+1}[i, j]$$

par définition du produit matriciel

On a donc bien $A^k[i, j] = \top$ ssi $x_i \Rightarrow_k x_j$.

► Question 8 Soit $k \geq n - 1$ et $0 \leq i, j < n$.

- Si $A^k[i, j] = \top$, alors $x_i \Rightarrow_k x_j$ d'après la question 7 et donc $x_i \Rightarrow x_j$. Donc d'après 2, on a $x_i \Rightarrow_{n-1} x_j$ et donc $A^{n-1}[i, j] = \top$ d'après 7.
- Si $A^{n-1}[i, j] = \top$, alors $x_i \Rightarrow_{n-1} x_j$ et, comme $k \geq n - 1$, $x_i \Rightarrow_k x_j$ d'après 1. Donc $A^k[i, j] = \top$.

► Question 9 On procède par exponentiation rapide (ce n'était pas nécessaire) et l'on traite correctement le cas $n = 1$ (probablement pas utile non plus). Petite difficulté supplémentaire liée au fait qu'on fait du C : ne pas oublier de libérer les matrices intermédiaires créées.

```

bool **power(bool **m, int n, int pow){
    if (pow == 0) return identity(n);
    bool **m2 = product(m, m, n, n, n);
    bool **b = power(m2, n, pow / 2);
    if (pow % 2 == 0) {
        matrix_free(m2, n);
        return b;
    } else {
        bool **result = product(b, m, n, n, n);
        matrix_free(m2, n);
        matrix_free(b, n);
        return result;
    }
}

bool **closure(bool **a, int n){
    return power(a, n, n - 1);
}

```

► **Question 10** La fonction `product` a clairement une complexité en $O(npq)$, donc $O(n^3)$ ici. Sachant qu'on effectue $O(\log n)$ multiplications avec notre méthode, on obtient une complexité en $O(n^3 \log n)$ pour `closure`.

► **Question 11** On pourrait extraire la ligne i de la matrice $CT(A)$, mais on ne respecterait pas la contrainte sur la complexité. On fait donc un parcours en profondeur :

```
void explore(bool **a, int n, int i, bool *known){
    known[i] = true;
    for (int j = 0; j < n; j++){
        if (a[i][j] && !known[j]) explore(a, n, j, known);
    }
}

bool *accessible(bool **a, int n, int i){
    bool *known = malloc(n * sizeof(bool));
    for (int i = 0; i < n; i++){
        known[i] = false;
    }
    explore(a, n, i, known);
    return known;
}
```

► **Question 12** On calcule simplement ligne par ligne.

```
bool **closure_dfs(bool **a, int n){
    bool **tc = malloc(n * sizeof(bool *));
    for (int i = 0; i < n; i++){
        tc[i] = accessible(a, n, i);
    }
    return tc;
}
```

Remarque

Si l'on procède par exponentiation de matrice, on obtient une complexité en $O(f(n) \ln n)$, où $f(n)$ est la complexité du calcul d'un produit de matrices $n \times n$. Avec la méthode naïve que nous avons employée, on a $f(n) = n^3$ et la complexité est donc moins bonne que celle obtenue ici. En utilisant, par exemple, l'algorithme diviser pour régner de Strassen en $O(n^{2.81})$, on obtient en revanche une meilleure complexité que celle de `closure_dfs`. La situation est différente si le graphe est creux et donné par un tableau de listes d'adjacence.

► **Question 13** Dans le graphe G_1 , x_0, x_2 et x_3 sont des axiomes (depuis chacun de ces sommets, on peut atteindre tous les sommets). En revanche, x_1 n'est pas un axiome (car $x_0 \Rightarrow x_1$ mais $x_1 \not\Rightarrow x_0$).

► **Question 14** Dans le graphe G_2 :

- depuis x_0 , tout le monde est accessible sauf x_3 , et $x_3 \not\Rightarrow x_0$, donc x_0 est un axiome;
- on a $x_0 \Rightarrow x_1$ et $x_0 \Rightarrow x_2$ mais x_0 n'est accessible ni depuis x_2 ni depuis x_2 , donc x_1 et x_2 ne sont pas des axiomes;
- pour x_3 , la situation est symétrique de celle de x_0 .

Les axiomes du graphe G_2 sont x_0 et x_3 .

- **Question 15** En partant de la clôture transitive, il n'y a aucune difficulté : on vérifie que « $x_j \Rightarrow x_i$ implique $x_i \Rightarrow x_j$ » est vrai pour tout j.

```
bool is_axiom(bool **b, int n, int i){  
    for (int j = 0; j < n; j++) {  
        if (b[j][i] && !b[i][j]) return false;  
    }  
    return true;  
}
```

Remarque

On n'utilise en fait qu'une ligne et une colonne de la matrice B, donc si l'on part de A il est plus efficace de ne calculer que cela. En termes de graphes, cela revient à calculer les sommets accessibles depuis i et ceux depuis lesquels i est accessible .

- **Question 16** Soit une suite $y_0 \Rightarrow y_1 \dots \Rightarrow y_h$, supposons qu'on ait $y_i = y_j$ avec $i \leq j - 1$. On a $y_i \Rightarrow y_{j-1}$ par transitivité de \Rightarrow et donc $y_j \Rightarrow y_{j-1}$. La suite n'est donc pas unidirectionnelle (point 3).

Dans une suite unidirectionnelle, les propositions sont deux à deux distinctes.

- **Question 17** On considère l'algorithme suivant :

Entrée : une sommet x

Sortie : un axiome y tel que $y \Rightarrow x$

Initialisation : $y \leftarrow x$

Tant que y n'est pas un axiome :

 Trouver z tel que $z \Rightarrow y$ et $y \not\Rightarrow z$.

$y \leftarrow z$

Renvoyer y

On prouve la correction et la terminaison, ce qui donne le résultat demandé :

- l'étape « trouver z » ne peut échouer car y n'est pas, à cet instant, un axiome;
- la suite $y_h, \dots, y_1, y_0 = x$ des valeurs successives de y (après h passages dans la boucle) est unidirectionnelle par construction. Donc :
 - cette suite est de longueur au plus n car constituée de sommets distincts, ce qui prouve la terminaison.
 - à tout instant, on a $y = y_h \Rightarrow x$;
- quand on sort de la boucle, on a donc bien y axiome et $y \Rightarrow x$.

- **Question 18** Les composantes fortement connexes sont $\{0\}, \{1, 2\}$ et $\{3\}$.

- **Question 19** Soit $x \in C$ un axiome, y un élément de C et $z \in V$ tel que $z \Rightarrow y$. On a $y \Rightarrow x$ car $y \in C$, donc $z \Rightarrow x$. Comme x est un axiome, on en déduit $x \Rightarrow z$ et donc $y \Rightarrow x \Rightarrow z$. Donc y est un axiome.

- **Question 20**

Minimalité : soit X une axiomatique, C une classe source et $y \in C$. Il existe $x \in X$ tel que $x \Rightarrow y$. Mais y est un axiome (il est dans une classe source), donc $y \Rightarrow x$. Donc x et y sont dans la même composante connexe : $x \in C$. Ainsi, X contient au moins un élément de chaque classe source.

Caractère suffisant : soit X constituée d'un élément de chaque classe source, et soit $y \in C$. D'après 17, il existe un axiome x tel que $x \Rightarrow y$. En notant z l'élément de X appartenant à la classe source contenant x, on a $z \Rightarrow x \Rightarrow y$, ce qui montre que X est une axiomatique.

- **Question 21** Pas de difficulté particulière, on utilise le fait que si x_i est un axiome et $x_j \Rightarrow x_i$, alors $x_j \Leftrightarrow x_i$.

```
bool *axiom_system(bool **b, int n){
    bool *system = malloc(n * sizeof(bool));
    for (int i = 0; i < n; i++){
        system[i] = true;
    }
    for (int i = 0; i < n; i++){
        if (system[i] && is_axiom(b, n, i)) {
            // eliminate all the other elements in the scc of i
            for (int j = i + 1; j < n; j++){
                if (b[j][i]) system[j] = false;
            }
        } else {
            system[i] = false;
        }
    }
    return system;
}
```

FICHIERS

Sujet dû à Nicolas Pécheux (Lycée Carnot). Ce sujet est long mais pas difficile : il faudra donc le finir chez vous.

Le programme encourage explicitement l'utilisation de fichiers pour stocker durablement des données sous différents formats. L'objectif de ce TP est de découvrir et d'expérimenter la gestion des fichiers en C et en OCaml. Dans ce TP, on s'intéresse à la lecture d'un fichier texte et à l'écriture de données textuelles, mais nous verrons plus tard que l'on peut aussi traiter de nombreux types de données : audio, vidéo, images, etc.

I Compléments sur le langage C

I.1 Chaînes de caractères

En C, une chaîne de caractères est simplement un tableau de **char** (octets, essentiellement) terminé par un zéro. Attention, il s'agit d'un « caractère nul » (octet égal à zéro) et pas du tout du caractère 0 (dont le code ASCII est 48).

Chaînes littérales Il est possible de définir une chaîne de caractères avec la syntaxe suivante :

```
char *string = "toto";
```

Dans ce cas, **string** pointe vers un bloc de *cinq* octets : le caractère nul final a été ajouté automatiquement. Une telle chaîne ne peut pas être modifiée : elle est typiquement stockée dans une zone mémoire en « lecture seule ».

Longueur d'une chaîne Une chaîne connaît implicitement sa longueur : c'est le nombre de caractères du pointeur fourni jusqu'au premier caractère nul, exclu. La fonction **strlen** est prédéfinie :

```
size_t strlen(char *string);
```

```
char *string = "toto";
char arr[8] = {'t', 'o', 't', 'o', 't', 'a', 't', 'a'};
printf("%d\n", strlen(string)); // OK, affiche 4

// Un appel strlen(arr) provoquerait une erreur : on dépasserait
// du tableau en cherchant un caractère nul.

arr[3] = '\0';
printf("%s : longueur %d\n", arr, strlen(arr));
// OK : affiche "tot : longueur 3"

printf("%s : longueur %d\n", &arr[3], strlen(&arr[3]));
// OK : affiche " : longueur 0"

arr[7] = '\0';
printf("%s : longueur %d\n", arr, strlen(arr));
// OK : affiche "tot : longueur 3" (on s'arrête au premier caractère nul)

printf("%s : longueur %d\n", &arr[4], strlen(&arr[4]));
// OK : affiche "tat : longueur 3"
```

Remarque

La fonction `strlen` parcourt la chaîne jusqu'à trouver un caractère nul : elle a donc une complexité linéaire en la taille de la chaîne. De plus, elle est extrêmement dangereuse puisqu'elle peut facilement provoquer un comportement non défini si la chaîne n'est pas correctement terminée. Une version plus récente existe :

```
size_t strlen_s(char *str, size_t strsz);
```

Cette version lit au maximum `strsz` octets, et renvoie `strsz` si elle ne trouve pas de caractère nul dans cette portion.

Autres fonctions sur les chaînes

La fonction `strcmp` permet de comparer deux chaînes de caractères :

```
int strcmp(char *lhs, char *rhs);
```

Elle renvoie :

- zéro si les deux chaînes sont égales ;
- une valeur strictement négative si `lhs` est avant `rhs` dans l'ordre lexicographique (par exemple "abc" par rapport à "abca", ou "azzz" par rapport à "ba") ;
- une valeur strictement négative si `rhs` est avant `lhs` dans l'ordre lexicographique.

La fonction `strcpy` permet de copier une chaîne de caractères dans un tableau de caractères :

```
char *strcpy(char *dest, char *src);
```

- La fonction copie les caractères (jusqu'au premier caractère nul inclus) de `src` dans le tableau pointé par `dest`. Si ce tableau n'est pas assez grand, le comportement est indéfini.
- La valeur de retour peut typiquement être ignorée : c'est un pointeur vers la copie, donc égal à `dest`.

1.2 Fichiers

La structure `FILE` permet de gérer un flux de données (un fichier, au sens large). Nous l'avons déjà utilisé implicitement, puisque trois objets de type `FILE*` sont prédéfinis :

- `stdin` qui correspond à l'entrée standard (le clavier, sauf si l'on a fait une redirection `./program < data_in.txt`), et qui est utilisé implicitement par `scanf` ;
- `stdout` qui correspond à la sortie standard (la console, sauf si l'on a fait une redirection `./program > data_out.txt`), et qui est utilisé implicitement par `printf` ;
- `stderr` (sortie standard d'erreur, dirigé par défaut vers la console comme `stdout`).

On peut aussi obtenir un `FILE*` à partir d'un fichier présent sur l'ordinateur grâce à la fonction suivante :

```
FILE *fopen(char *filename, char *accessMode);
```

- `filename` est une chaîne de caractères indiquant le chemin d'accès du fichier : "data.txt", ou "data/file1.csv" (chemin relatif dans les deux cas), ou "/home/toto/image.ppm" (chemin absolu).
- `accessMode` indique le mode d'ouverture du fichier :
 - "r" pour *read* (le fichier est ouvert en lecture, et doit exister préalablement) ;
 - "w" pour *write* (le fichier est ouvert en écriture : le fichier est écrasé s'il existe déjà, créé sinon) ;
 - "a" pour *append* (le fichier est ouvert en mode ajout : il est créé s'il n'existe pas, sinon la position courante est fixée à la fin du fichier et l'on peut ajouter des données).

Plusieurs autres modes existent (en particulier pour ouvrir des fichiers en mode binaire, tous les modes présentés étant pour des fichiers en mode texte), mais nous n'en aurons pas besoin aujourd'hui.

Un fichier ouvert par `fopen` doit ensuite être fermé par `fclose` :

```
FILE *f = fopen("data.txt", "r");
// on traite le fichier
fclose(f);
```

Remarque

Le terme de *fichier* doit ici être compris en un sens très large : il peut certes désigner un fichier au sens courant, mais aussi un flux d'entrée ou de sortie associé à un périphérique (clavier, console, imprimante), une zone mémoire permettant la communication avec d'autres processus...

Ces différents types de flux n'autorisent d'ailleurs pas les mêmes opérations : tous permettent une lecture linéaire (c'est le comportement par défaut), certains permettent de se déplacer à une position arbitraire alors que pour d'autres (entrée sur le clavier, par exemple) cela n'a aucun sens...

Fonctions `fscanf` et `fprintf` Les fonctions `fscanf` et `fprintf` fonctionnent exactement comme `scanf` et `printf`, sauf qu'elles prennent un argument supplémentaire indiquant sur quel flux doit se faire la lecture ou l'écriture :

```
int fprintf(FILE *stream, char *format, ...);
int fscanf(FILE *stream, char *format, ...);
```

- Pour `fprintf`, le flux `stream` doit avoir été ouvert en écriture ; pour `fprintf`, en lecture.
- Un appel à `printf` équivaut à un appel à `fprintf` avec le paramètre `stream` fixé à `stdout`.
- Un appel à `scanf` équivaut à un appel à `fscanf` avec le paramètre `stream` fixé à `stdin`.

Nous avons surtout utilisé `scanf` pour lire des nombres jusqu'à présent, mais dans ce TP il faudra lire des caractères et des chaînes. Les formats utiles sont :

- `fscanf(f, "%c", char_pointer)` qui lit un caractère ;
- `fscanf(f, "%s", char_pointer)` qui lit une chaîne de caractères en s'arrêtant juste avant le premier caractère d'espacement rencontré (espace, tabulation ou retour à la ligne). Comportement non défini si le tableau `char_pointer` n'est pas assez grand pour contenir la chaîne lue (y compris le caractère nul rajouté automatiquement à la fin).

Autres fonctions Même si la bibliothèque standard du C est assez rudimentaire, elle fournit bien sûr un certain nombre d'autres fonctions pour interagir avec les flux. Ces fonctions ne sont pas au programme, ce qui ne nous empêchera pas nécessairement de nous en servir (`fseek` dans ce sujet, par exemple), mais signifie que vous n'avez pas à les maîtriser.

On peut cependant mentionner une fonction qui peut être utile pour certaines questions :

```
char *fgets(char *str, int count, FILE *stream);
```

Reads at most `count - 1` characters from the given file stream and stores them in the character array pointed to by `str`. Parsing stops if a newline character is found, in which case `str` will contain that newline character, or if end-of-file occurs. If bytes are read and no errors occur, writes a null character at the position immediately after the last character written to `str`.

2 Gestion des fichiers (en OCaml et en C)

Cette partie est à faire tout d'abord en OCaml puis à reprendre entièrement en C.

► **Question 22** En ouvrant en écriture puis fermant immédiatement, créer un fichier vide. Vérifier avec les commandes UNIX que cela a bien fonctionné.

- **Question 23** Écrire les lignes suivantes dans le fichier que vous venez de créer précédemment, en écrasant son ancien contenu (qui était vide, ce n'est donc pas bien grave)

Première ligne
Deuxième ligne

- **Question 24** Afficher à l'écran le contenu du fichier précédent.

- **Question 25** (En C uniquement) Ajouter une troisième ligne de votre choix dans ce fichier, en utilisant le mode "a". Vérifier.

- **Question 26** Créer un fichier de nom `plot.txt` qui contient le tableau de valeurs de la fonction $x \mapsto x^2$ avec 10 000 lignes de la forme ci-dessous. Vérifier que le fichier est bien présent et contient bien ce qu'il faut.

0 0
1 1
2 4
3 9
...

- **Question 27** Écrire un programme `copy` qui prend deux arguments sur la ligne de commande `nom1` et `nom2` et qui copie le contenu du fichier `nom1` dans un fichier de nom `nom2`. Si le deuxième fichier existait déjà, son contenu est écrasé. Faire quelques tests.

Remarque

En OCaml, on pourra procéder ligne par ligne. En C, sans connaître la longueur maximale des lignes, c'est nettement plus problématique : le plus simple est de procéder caractère par caractère.

- **Question 28** Modifier le programme précédent pour que les lettres « a » et « e » soient échangées lors de cette copie corrompue.

3 Prénoms français (en C)

Le fichier `prenoms.txt` dans le répertoire `data/prenoms` contient les prénoms donnés aux petites françaises et aux petits français, à raison d'un prénom par ligne. Dans toute cette partie on demande d'écrire en C une fonction ou un programme permettant de répondre aux questions posées.

- **Question 29** Déterminer le nombre de prénoms.
- **Question 30** Combien il y a-t-il de prénoms qui commencent par une lettre donnée, par exemple par « e » ?
- **Question 31** Quels sont les prénoms les plus longs ? Quels sont les prénoms les plus courts ?
- **Question 32** Est-ce qu'un prénom a été donné, par exemple le vôtre ?
- **Question 33** Quelle proportion de prénoms comportent une lettre donnée, par exemple la lettre « e » ?
Remarque : cette proportion doit être inférieure à 100% !
- **Question 34** Quels sont les prénoms palindromiques ?
- **Question 35** Inventer une question intéressante et y répondre.

4 Cent mille milliards de poèmes (en OCaml)

En 1961, l'écrivain et poète Raymond Queneau proposa un recueil de sonnets potentiels à composer par le lecteur. Un sonnet est composé de 14 vers — ici des alexandrins — séparés en deux groupes de 4 vers (les quatrains avec une succession de rimes ABAB) et deux groupes de 3 vers (les tercets de rime AAB et CCB). Pour chaque vers Queneau propose 10 possibilités. Ces possibilités sont données dans les fichiers q1v1.txt (premier quatrain, premier vers), q1v2.txt (premier quatrain, deuxième vers), ..., t2v3 (deuxième tercet, troisième vers), qui se trouvent dans le répertoire data/queneau.

Par exemple, le premier poème est :

Le roi de la pampa retourne sa chemise
pour la mettre à sécher aux cornes des taureaux
le cornédbif en boîte empeste la remise
et fermentent de même et les cuirs et les peaux

Je me souviens encor de cette heure exequise
les gauchos dans la plaine agitaient leurs drapeaux
nous avions aussi froid que nus sur la banquise
lorsque pour nous distraire y plantions nos tréteaux

Du pôle à Rosario fait une belle trotte
aventures on eut qui s'y pique s'y frotte
lorsqu'on boit du maté l'on devient argentin

L'Amérique du Sud séduit les équivoques
exaltent l'espagnol les oreilles baroques
si la cloche se tait et son terlintintin

► **Question 36** Écrire une fonction `premier_poeme : unit -> unit` qui affiche ce premier poème à partir des fichiers sources des vers.

Pour la question suivante, on utilisera la fonction `Random.int : int -> int` du module `Random` que l'on initialisera avec `Random.self_init : unit -> unit`.

► **Question 37** Écrire une fonction `compose_poeme : unit -> unit` qui affiche une possibilité de poème choisie aléatoirement parmi les 10^{14} poèmes possibles. On pourra commencer par écrire le premier quatrain. Tester plusieurs fois et admirer.

► **Question 38** Écrire une fonction `sauvegarde_poemes : unit -> unit` qui écrit dans le répertoire courant cent poèmes choisis aléatoirement de noms `poeme00.txt`, `poeme01.txt`, ..., `poeme99.txt`.

Queneau ajoute : « En comptant 45 s pour lire un sonnet et 15 s pour changer les volets à 8 heures par jour, 200 jours par an, on a pour plus d'un million de siècles de lecture, et en lisant toute la journée 365 jours par an, pour 190 258 751 années plus quelques plombes et broquilles (sans tenir compte des années bissextiles et autres détails) ».

5 Décimales de π (en C)

Le fichier `pi.txt`, qui se trouve dans le répertoire `data/pi/` contient les 10 premiers millions de décimales de π . Par exemple, les 50 premières décimales de π sont :

```
> head -c50 data/pi/pi.txt
14159265358979323846264338327950288419716939937510
```

► **Question 39** Écrire en C un programme `decimal_de_pi.c` qui, une fois compilé, prend en argument un entier $n \in \mathbb{N}^*$ sur la ligne de commande et qui affiche la n -ème décimale de pi. Si le fichier n'est pas lisible, si l'argument en ligne de commande n'est pas donné ou si celui-ci ne correspond pas à une décimale que l'on peut trouver dans le fichier, on affichera sur la sortie d'erreur un message et on arrêtera le programme.

Remarques

- On se déplacera séquentiellement dans le fichier jusqu'à la décimale voulue.
- On rappelle que atoi permet de convertir une chaîne de caractères en entier.
Par exemple :

```
> ./decimale_de_pi 939  
7
```

► **Question 40** Écrire en C un programme histogramme_decimales_de_pi.c qui, une fois compilé, affiche l'histogramme des fréquences d'apparition des décimales de π , toujours en gérant correctement les éventuelles erreurs.

On doit obtenir :

```
> ./histogramme_decimales_de_pi  
0 : 0.099944  
1 : 0.099933  
2 : 0.100031  
3 : 0.099996  
4 : 0.100109  
5 : 0.100047  
6 : 0.099934  
7 : 0.100021  
8 : 0.099981  
9 : 0.100004
```

Remarque

Il semblerait que π soit un nombre *équiréparti*. On conjecture que π est un nombre *univers* c'est-à-dire que l'on peut trouver dans son développement décimal toute suite finie de chiffres. On pense même que π est un nombre *normal* c'est-à-dire que toute suite finie de décimales consécutives de longueur k apparaît avec une fréquence limite de $\frac{1}{10^k}$.

Par exemple, on peut trouver le nombre 9265 dans les décimales de π puisque $\pi = 1415926535\dots$

Dans la question suivante, on pourra utiliser les fonctions suivantes :

```
long ftell(FILE *stream);
```

Returns the file position indicator for the file stream stream.

If the stream is open in binary mode, the value obtained by this function is the number of bytes from the beginning of the file.

If the stream is open in text mode, the value returned by this function is unspecified and is only meaningful as the input to fseek().

```
int fseek(FILE *stream, long offset, int whence);
```

Sets the file position indicator for the file stream stream to the value pointed to by offset.

If the stream is open in binary mode, the new position is exactly offset bytes measured from the beginning of the file if origin is SEEK_SET, from the current file position if origin is SEEK_CUR, and from the end of the file if origin is SEEK_END. Binary streams are not required to support SEEK_END, in particular if additional null bytes are output.

If the stream is open in text mode, the only supported values for offset are zero (which works with any origin) and a value returned by an earlier call to ftell on a stream associated with the same file (which only works with origin of SEEK_SET).

Returns 0 upon success, nonzero value otherwise.

► **Question 41** Écrire en C un programme `indice_dans_decimales_de_pi` qui, une fois compilé, prend en argument un entier $n \in \mathbb{N}$ sur la ligne de commande et qui affiche l'indice de la position de la première occurrence de cet entier n dans les décimales de π , en commençant à l'indice 1 pour la première décimale. Si aucune occurrence n'est trouvée (vraisemblablement parce que 10 millions de décimales ce n'est pas assez), on affichera -1 . Comme ci-dessus, on gérera les erreurs éventuelles.

Par exemple :

```
> ./indice_dans_decimales_de_pi 9265
5
```

6 Nombre de langages informatiques (langage libre)

Le fichier `liste_langages.txt` dans le répertoire `donnees/langages` est tiré de [la page correspondante de Wikipedia](#).

► **Question 42** Donner le nombre de langages de programmation répertoriés dans ce fichier.

7 Admissibles aux mines (langage libre)

Le fichier `mines.tsv` qui se trouve dans le répertoire `donnees/mines` contient les résultats des admissibilités des candidats au concours MINES-PONTS 2015. Il est issu d'un pdf disponible sur le site du concours¹.

Dans ce fichier, sur chaque ligne figurent les champs : numéro de candidat, nom et prénom, résultat, série d'oral si admissible. Chaque champ est séparé par une tabulation '\t'. On appelle ce format *Tab-separated values* d'où l'extension.

Le fichier étant issu d'un fichier pdf, il a été nettoyé, mais il reste les numéros des pages initiales sur certaines lignes.

► **Question 43** Afficher avec un programme C ou OCaml le contenu de ce fichier.

► **Question 44** Écrire une fonction `nettoyage` créant un fichier dans le même répertoire dont le nom est `mines_propre.tsv` dans lequel ne figurent plus ces numéros de ligne.

► **Question 45** Écrire une fonction `nb_admissibles` renvoyant le nombre d'admissibles.

► **Question 46** Écrire une fonction `separe_series` créant, toujours dans le bon répertoire, un fichier pour chaque série d'oral : `serie1.tsv`, `serie2.tsv`, `serie3.tsv` et `serie4.tsv`. Chaque fichier contenant le numéro et l'identité des candidats de la série correspondante, au format *Tab-separated values*.

1. <https://mines-ponts.fr/>

MANIPULATION DE FORMULES LOGIQUES

On considère des formules propositionnelles définies par le type suivant :

```
type formule =
| Const of bool
| Var of string
| Et of formule * formule
| Ou of formule * formule
| Non of formule
```

I Affichage d'une formule logique

Exercice XLI.I

p. 715

- Écrire une fonction `string_of_formule` renvoyant la représentation infixée d'une expression sous la forme d'une chaîne de caractères. Le parenthésage doit être suffisant pour qu'il n'y ait pas d'ambiguïté (il peut être excessif).

```
string_of_formule : formule -> string
```

```
# string_of_formule (Ou (Et (Var "y", Const false), Non (Var "z")));;
- : string = "(y et false) ou non z"
```

- Pour minimiser le nombre de parenthèses utilisées, on définit les priorités suivantes :

- **Non** est prioritaire sur **Et** et **Ou**;
- **Et** est prioritaire sur **Ou**.

Ainsi, "`non x_1 et x_2 ou non x_3`" signifie "`((non x_1) et x_2) ou (non x_3)`".

On en profitera également pour se débarrasser des parenthèses rendues inutiles par l'associativité des différents opérateurs : on écrira "`x_1 et x_2 et x_3`" plutôt que "`x_1 et (x_2 et x_3) ou (x_1 et x_2) et x_3`".

On donne la fonction `priorite : formule -> int` suivante :

```
let priorite = function
| Var _ | Const _ -> max_int
| Ou _ -> 0
| Et _ -> 1
| Non _ -> 2
```

Écrire une fonction `string_priorite` n'utilisant que les parenthèses nécessaires.

```
string_priorite : formule -> string
```

```
# string_priorite antinomie;;
- : string = "non x_0 et (x_2 et x_0 ou x_1) et non x_1 et non x_2"
```

2 Égalité syntaxique modulo associativité et commutativité

Dans cette section, on considère qu'une formule logique est un arbre de type `formule`, mais que deux formules peuvent être syntaxiquement égales sans correspondre au même arbre. Plus précisément, deux formules seront dites syntaxiquement égales si l'on peut passer de l'une à l'autre par l'application répétée des règles suivantes :

- $A \wedge (B \wedge C) \simeq (A \wedge B) \wedge C$ (Asso- \wedge)
- $A \vee (B \vee C) \simeq (A \vee B) \vee C$ (Asso- \vee)
- $A \wedge B \simeq B \wedge A$ (Com- \wedge)
- $A \vee B \simeq B \vee A$ (Com- \vee)

Exercice XLI.2 – Traitement artisanal de la commutativité

p. 715

Dans cet exercice, on ne considère que les deux règles (Com- \vee) et (Com- \wedge).

1. Dans chacun des cas suivants, indiquer si les deux expressions sont égales modulo les transformations considérées :
 - a. $(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$ et $(x_2 \wedge x_3) \wedge (x_0 \wedge x_1)$;
 - b. $(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$ et $(x_2 \wedge x_3) \wedge (x_1 \wedge x_0)$;
 - c. $(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$ et $(x_2 \wedge x_0) \wedge (x_1 \wedge x_3)$.
2. Écrire une fonction `egal_com` décidant si deux formules sont égales modulo la commutativité.

```
egal_com : formule -> formule -> bool
```

On devrait avoir :

```
# egal_commut ex2 ex3;;
- : bool = true
# egal_commut gros_ex1 gros_ex2;;
- : bool = false
```

Gérer l'associativité, et plus encore la combinaison de la commutativité avec l'associativité, est plus délicat, surtout si l'on veut un temps de calcul raisonnable. Pour commencer, on ne peut plus se contenter d'arbres binaires. On décide donc de définir un nouveau type :

```
type formule_asso =
| C of bool
| V of int
| EtA of formule_asso list
| OuA of formule_asso list
| N of formule_asso
```

L'idée est ensuite de définir un représentant canonique pour chaque classe d'équivalence modulo associativité et commutativité, et une manière efficace de calculer ce représentant.

En OCaml, tous les types à l'exception des types fonctionnels sont dotés d'une relation d'ordre (totale). Cette relation est définie sur les types de base (pour `bool`, on a `false < true`) et ensuite étendue aux types algébriques de la manière suivante :

- pour un type produit $t = a_1 * a_2 * \dots * a_n$, on prend l'ordre lexicographique induit par les ordres pré-existants sur a_1, a_2, \dots, a_n ;
- pour un type somme $t = A_1 \text{ of } t_1 | \dots | A_n \text{ of } t_n$, on a $A_1 x_1 < A_2 x_2 < \dots < A_n x_n$ quels que soient x_1, \dots, x_n (et bien sûr $A_i x < A_j y$ si $x < y$).

Autrement dit, l'ordre induit sur un type somme est déterminé par l'ordre dans lequel les variantes apparaissent dans la définition du type.

Une expression de type `formule_asso` sera dite *canonique* si elle vérifie les conditions suivantes :

- aucun noeud `EtA` n'a d'enfant de la forme `EtA xs`;
- aucun noeud `OuA` n'a d'enfant de la forme `OuA xs`;
- pour chaque noeud de la forme `EtA enfants` ou `OuA enfants`, la liste `enfants` est triée par ordre croissant.

Exercice XLI.3 – Fonctions préliminaires

p. 716

1. Écrire une fonction `insere` : `'a -> 'a list -> 'a list` insérant un élément dans une liste supposée triée.
2. Écrire une fonction `fusionne` : `'a list -> 'a list -> 'a list` fusionnant deux listes supposées triées.

Exercice XLI.4 – Égalité syntaxique

p. 716

1. Écrire une fonction canonique : `formule -> formule_asso` mettant une expression sous forme canonique.
2. Écrire une fonction `egal_syntaxe` : `formule -> formule -> bool` décidant l'égalité syntaxique (modulo associativité et commutativité) de deux expressions logiques.

Solutions

Correction de l'exercice XLI.1 page 712

1. Pas de difficulté particulière :

```
let rec string_of_formule f =
  match f with
  | Const b -> sprintf "%b" b
  | Var s -> s
  | Et (f1, f2) ->
    sprintf "(%s et %s)" (string_of_formule f1) (string_of_formule f2)
  | Ou (f1, f2) ->
    sprintf "(%s ou %s)" (string_of_formule f1) (string_of_formule f2)
  | Non f1 ->
    sprintf "(non %s)" (string_of_formule f1)
```

```
2. let rec string_priorite formule =
(* La seule possibilité pour priorité expr = prio_parent est que
 * expr et son parent aient le même opérateur à la racine.
 * - si c'est Non (Non f), il est inutile de parenthésier
 *   (opérateur unaire) ;
 * - si c'est Et (Et (f, g), h), il est également inutile de
 *   parenthésier puisque Et est associatif ;
 * - de même pour Ou. *)
let parenthese f prio_parent =
  if priorite f >= prio_parent then
    string_priorite f
  else
    sprintf "(%s)" (string_priorite f) in
match formule with
  | Const b -> string_of_bool b
  | Var s -> s
  | Non f -> sprintf "non %s" (parenthese f 2)
  | Et (ga, dr) ->
    sprintf "%s et %s" (parenthese ga 1) (parenthese dr 1)
  | Ou (ga, dr) ->
    sprintf "%s ou %s" (parenthese ga 0) (parenthese dr 0)
```

Correction de l'exercice XLI.2 page 713

1. a. Oui, il suffit d'appliquer la règle Com- \wedge à la racine.
b. Oui, avec deux applications de la règle Com- \wedge .
c. Non, on aurait besoin de l'associativité ici.
2. À chaque nœud pertinent, on essaie les deux possibilités :

```
let rec egal_commut a b =
  match a, b with
  | Non ex, Non ex' -> egal_commut ex ex'
  | Var i, Var j -> i = j
  | Const b, Const b' -> b = b'
  | Et (ga, dr), Et (ga', dr') | Ou (ga, dr), Ou (ga', dr') ->
    (egal_commut ga ga' && egal_commut dr dr')
    || (egal_commut ga dr' && egal_commut ga' dr)
  | _ -> false
```

Au premier abord, la complexité de cette fonction semble être exponentielle en la taille de la formule (au moins dans le pire cas). Cependant, je ne suis pas du tout convaincu que ce soit vrai : si vous arrivez à construire une famille de formules pour laquelle on a effectivement une complexité exponentielle (ou au contraire à prouver que la complexité est polynomiale dans le pire cas,) je suis intéressé.

Correction de l'exercice XLI.3 page 714

- À savoir faire, bien évidemment :

```
let rec insere formule liste =
  match liste with
  | [] -> [formule]
  | x :: xs when formule <= x -> formule :: x :: xs
  | x :: xs -> x :: insere formule xs
```

- Tout aussi classique :

```
let rec fusionne enfants enfants' =
  match enfants, enfants' with
  | [], _ -> enfants'
  | _, [] -> enfants
  | x :: xs, y :: ys when x <= y -> x :: fusionne xs enfants'
  | _, y :: ys -> y :: fusionne enfants ys
```

Correction de l'exercice XLI.4 page 714

- Le code n'est pas compliqué, mais il faut bien réfléchir à ce qu'on fait dans les cas **Et** et **Ou**. Notez que `fusionne [ga'] [dr']` est juste une manière concise de dénoter la liste `[ga'; dr']` ou `[dr'; ga']` (celle des deux qui est croissante).

```
let rec canonique = function
| Const b -> C b
| Var s -> V s
| Non ex -> N (canonique ex)
| Et (ga, dr) ->
  begin
    let ga', dr' = canonique ga, canonique dr in
    match ga', dr' with
    | EtA enfants_g, EtA enfants_d -> EtA (fusionne enfants_g
      ↪ enfants_d)
    | EtA enfants_g, _ -> EtA (insere dr' enfants_g)
    | _, EtA enfants_d -> EtA (insere ga' enfants_d)
    | _, _ -> EtA (fusionne [ga'] [dr'])
  end
| Ou (ga, dr) ->
  begin
    let ga', dr' = canonique ga, canonique dr in
    match ga', dr' with
    | OuA enfants_g, OuA enfants_d -> OuA (fusionne enfants_g
      ↪ enfants_d)
    | OuA enfants_g, _ -> OuA (insere dr' enfants_g)
    | _, OuA enfants_d -> OuA (insere ga' enfants_d)
    | _, _ -> OuA (fusionne [ga'] [dr'])
  end
```

2. Le problème se ramène à l'égalité structurelle des deux représentants canoniques.

```
let egal_syntaxe f1 f2 =
  canonique f1 = canonique f2
```

AUTOUR DE DIJKSTRA

I File de priorité

Pour implémenter l'algorithme de Dijkstra, nous allons utiliser la structure de file de priorité enrichie de l'opération DECREASEPRIO vue en cours. Pour simplifier, nous allons nous limiter au cas où :

- la capacité N de la file est fixée à la création;
- les clés sont des entiers positifs entre 0 et N – 1;
- les priorités sont des flottants.

Pour représenter cette structure, on utilisera le type suivant :

```
type t =
  {mutable last : int;
   priorities : float array;
   keys : int array;
   mapping : int array}
```

- Les trois tableaux `priorities`, `keys` et `mapping` sont de même taille N : la capacité de la file, fixée à la création.
- La partie « active » du tas est contenue dans la partie des tableaux `keys` et `priorities` située entre les indices 0 et `last` (inclus).
- On utilise, comme d'habitude, un tas binaire sous la forme d'un arbre binaire complet gauche stocké implicitement dans un (ou ici deux) tableaux, avec la racine à l'indice zéro.
- Le tableau `mapping` vérifie l'invariant suivant :
 - si la clé `i` n'est pas présente dans le tas, alors `mapping.(i) = -1`;
 - sinon, `keys.(mapping.(i)) = i`, et la priorité correspondante est dans `priorities.(mapping.(i))`. Autrement dit, le tableau `mapping` permet de retrouver l'emplacement d'une clé dans le tas (ce qui est nécessaire lorsque l'on souhaite diminuer la priorité associée).

Pour une fois, nous allons définir un module pour regrouper les différentes fonctions agissant sur une file de priorité. La syntaxe est la suivante :

```
module PrioQ :
sig
  type t
  val get_min : t -> (int * float)
  ...
  val mem : t -> int -> bool
end = struct
  type t = {mutable last : int; ... }

  let get_min q = ...
  ...
  let mem q x = ...
end
```

Après cette définition, les types et fonctions déclarées dans la signature sont disponibles, en les préfixant avec le nom du module (`PrioQ.t` pour le type, `PrioQ.get_min...`). On peut définir des fonctions auxiliaires supplémentaires dans la partie `struct`, mais ces fonctions seront « privées » (inaccessibles depuis l'extérieur).

Remarque

Si vous ne vous souvenez pas très bien du fonctionnement de la structure de tas (enrichie ou non), n'hésitez pas à consulter le cours. La seule différence est qu'on utilise ici deux tableaux `keys` et `priorities` au lieu d'un seul tableau contenant des couples (clé, priorité). Il est cependant toujours profitable d'essayer de retrouver les algorithmes par vous-mêmes.

► **Question 1** Redonner les formules permettant de retrouver les indices du fils gauche, du fils droit et du père de i dans un arbre binaire complet gauche implicite.

► **Question 2** Écrire une fonction `full_swap` telle que `full_swap q i j` échange les positions dans le tas des clés `q.keys.(i)` et `q.keys.(j)`, en maintenant les invariants (il faudra donc aussi agir sur les tableaux `q.priorities` et `q.mapping`). On pourra supposer sans le vérifier que i et j sont des indices valides (compris entre 0 et `q.last`).

```
full_swap : t -> int -> int -> unit
```

► **Question 3** Écrire la fonction `sift_up` effectuant la percolation vers le haut d'une clé du tas.

```
sift_up : t -> int -> unit
```

Remarque

À nouveau (et il en ira de même pour toutes les questions suivantes), on prendra garde à maintenir tous les invariants de la structure.

► **Question 4** Écrire la fonction `insert` réalisant l'insertion d'une clé dans la file (avec une priorité associée). On pourra supposer sans le vérifier que la clé n'est pas déjà présente dans la file.

```
insert : t -> (int * float) -> unit
```

► **Question 5** Écrire la fonction `sift_down` effectuant la percolation vers le bas d'une clé.

```
sift_down : t -> int -> unit
```

► **Question 6** Écrire la fonction `extract_min` réalisant l'extraction du minimum.

```
extract_min : t -> int * float
```

► **Question 7** Écrire la fonction `decrease_priority` qui diminue la priorité associée à une clé. On vérifiera que la clé est présente et que la nouvelle priorité est bien inférieure à l'ancienne (et lèvera une exception sinon).

```
decrease_priority : t -> int * float -> unit
```

► **Question 8** Que faudrait-il modifier si l'on souhaitait pouvoir *augmenter* la priorité d'une clé existante ?

► **Question 9** Déterminer la complexité des opérations `insert`, `extract_min`, `mem` et `decrease_prio`.

2 Algorithme de Dijkstra

Dans cette partie, on considère des graphes *a priori* orientés et pondérés par des flottants positifs ou nuls, représentés sous forme de tableaux de listes d'adjacence :

```
type weighted_graph = (int * float) list array
```

► **Question 10** Écrire sous forme de pseudo-code l'algorithme de Dijkstra, puis comparer avec le cours.

► **Question 11** Écrire une fonction `dijkstra` prenant en entrée un graphe à n sommets (numérotés $0, \dots, n-1$) et un indice x_0 de sommet, et renvoyant un tableau `dist` de taille n telle que $\text{dist} . (j)$ soit le poids d'un plus court chemin de x_0 à j .

Remarque

Si j n'est pas accessible depuis x_0 , alors $\text{dist} . (j)$ vaudra `infinity`.

```
dijkstra : weighted_graph -> int -> float array
```

► **Question 12** Modifier la fonction `dijkstra` en une fonction `dijkstra_tree` qui renvoie également un tableau `tree` codant l'arbre de parcours associé. Autrement dit, on devra avoir (en notant x_0 le sommet initial passé en argument à la fonction) :

- `tree . (i) = None` si i n'est pas accessible depuis x_0 ;
- `tree . (x_0) = Some x_0`;
- `tree . (i) = Some j` si le plus court chemin trouvé par l'algorithme pour aller de x_0 à i se termine par l'arc $j \rightarrow i$.

```
dijkstra_tree : weighted_graph -> int -> (float array * int option array)
```

► **Question 13** Écrire une fonction `reconstruct_path` prenant en entrée un tableau codant un arbre de parcours comme ci dessus et un indice de sommet `goal` et renvoyant un plus court chemin de x_0 à `goal` sous forme d'une liste de sommets.

Remarques

- La fonction ne prend pas x_0 en entrée puisque ce n'est pas nécessaire : c'est le seul sommet qui est son propre père.
- On lèvera une exception s'il n'existe pas de chemin.

```
reconstruct_path : int option array -> int -> int list
```

3 Calcul d'itinéraire pour misanthropes

3.1 Graphe des communes de France

Les deux fichiers `communes.csv` et `adjacence.csv` contiennent des informations sur les communes françaises :

- `communes.csv` contient, pour chaque commune, un identifiant entier unique, le code INSEE (identifiant alphanumérique unique), le nom, le département et la population;
- `adjacence.csv` contient la liste des communes immédiatement adjacentes (l'identifiant utilisé est l'identifiant entier unique du fichier `communes.csv`). Chaque paire de communes adjacentes n'est présente qu'une seule fois (s'il y a une ligne pour x, y , la ligne y, x n'est pas présente).

► **Question 14** Déterminer la structure des fichiers. On pourra utiliser les commandes `head` et `tail` en ligne de commande (par défaut, elles affichent respectivement les dix premières et dix dernières lignes d'un fichier).

On définit le type suivant pour représenter une commune :

```
type commune =  
{id : int;  
insee : string;  
nom : string;  
pop : int;  
dep : string}
```

Remarque

`insee` et `dep` sont de type `string` (et pas `int`) à cause des communes corses (départements 2A et 2B).

► **Question 15** Créer à partir du fichier `communes.csv` un tableau `tab_communes` contenant à l'indice `i` la commune dont l'`id` vaut `i`.

```
lire_communes : string -> communes array
```

Pour lire une ligne, on pourra utiliser :

- `input_line` pour récupérer la ligne (sans le `\n` final) sous forme d'une chaîne de caractères ;
- `Scanf.sscanf` pour en extraire les données. Pour lire une chaîne de caractères jusqu'à la première occurrence d'un certain caractère, le code de format est `%s@;` (pour lire jusqu'au premier `;`, ce qui sera le cas ici). Le code suivant, par exemple, lit une chaîne de caractère et deux entiers, séparés par des virgules, et renvoie le couple constitué de la chaîne de caractères et de la somme des deux entiers :

```
Scanf.sscanf s "%s@,%d,%d" (fun s x y -> (s, x + y))
```

► **Question 16** Créer à partir du fichier `adjacences.csv` le graphe (non orienté et non pondéré) défini par ces adjacences.

```
lire_graphe : int -> string -> int list array
```

Remarque

L'argument entier est le nombre de communes, qui correspond par exemple à la taille du tableau de communes créé par le code fourni.

On suppose à présent que l'on dispose de deux variables globales `tab_communes` et `g_adj` correspondant respectivement au résultat de l'appel à la fonction `lire_communes` et `lire_graphe`.

3.2 Saute canton

Le jeu `Saute canton` consiste à partir d'une commune aléatoire et à passer de commune adjacente en commune adjacente en essayant d'arriver le plus rapidement possible à une commune d'au moins 50 000 habitants.

► **Question 17** Écrire une fonction `saute_canton` qui renvoie un chemin de longueur minimale reliant la commune passée en argument à une commune (quelconque) d'au moins 50 000 habitants. Le chemin sera donné sous forme d'une liste d'identifiants de communes.

```
saute_canton : int -> int list
```

Remarques

- On arrêtera le parcours dès que possible, en utilisant par exemple une exception (d'autres solutions sont possibles).
- Il n'existe pas toujours de chemin gagnant (certaines composantes connexes ne contiennent pas de communes de plus de 50 000 habitants). On pourra renvoyer un chemin vide dans ce cas.

► **Question 18** Déterminer la (ou une des) commune la plus « perdue » de France suivant le critère de saute canton (celle pour laquelle le chemin minimal vers une « grande » commune est le plus long possible).

3.3 Le saute canton du misanthrope

On s'intéresse désormais au cas d'un voyageur misanthrope : il souhaite voyager d'une commune A à une commune B (toutes deux fixées), mais tient absolument à rencontrer le moins de personnes possible en route. Autrement dit, il cherche un chemin minimisant la somme des populations des communes traversées.

► **Question 19** Expliquer comment construire un graphe permettant de résoudre ce problème à l'aide de l'algorithme de Dijkstra.

► **Question 20** Quel chemin conseillez-vous au misanthrope pour relier Villeurbanne à La Mulatière ? Montrouge à Aubervilliers ?

Remarque

Les chemins les plus courts sont respectivement Villeurbanne - Lyon - La Mulatière et Montrouge - Paris - Aubervilliers, mais notre ami misanthrope est prêt à de très longs détours !

Solutions

- **Question 1** Les enfants sont en $2i + 1$ et $2i + 2$ (sous réserve d'existence), le parent en $\lfloor (i - 1)/2 \rfloor$.
- **Question 2** On suppose qu'une fonction swap est préalablement définie (c'est le cas dans le fichier fourni).

```
let full_swap q i j =
  swap q.keys i j;
  swap q.priorities i j;
  swap q.mapping q.keys.(i) q.keys.(j)
```

Pour améliorer la lisibilité, on définit :

```
let left i = 2 * i + 1
let right i = 2 * i + 2
let parent i = (i - 1) / 2
```

- **Question 3**

```
let rec sift_up q i =
  let j = parent i in
  if i > 0 && q.priorities.(i) < q.priorities.(j) then begin
    full_swap q i j;
    sift_up q j
  end
```

- **Question 4**

```
let insert q (x, prio) =
  if length q = capacity q then failwith "insert"
  else begin
    let l = q.last + 1 in
    q.keys.(l) <- x;
    q.priorities.(l) <- prio;
    q.mapping.(x) <- l;
    q.last <- l;
    sift_up q q.last
  end
```

- **Question 5**

```
let rec sift_down q i =
  let prio = q.priorities in
  let smallest = ref i in
  if left i <= q.last && prio.(left i) < prio.(i) then
    smallest := left i;
  if right i <= q.last && prio.(right i) < prio.(!smallest) then
    smallest := right i;
  if !smallest <> i then begin
    full_swap q i !smallest;
    sift_down q !smallest
  end
```

- **Question 6** Ici, il ne suffit pas de faire un `full_swap` : il faut aussi penser à marquer la clé extraite comme absente.

```
let extract_min q =
  if q.last < 0 then
    failwith "extract_min"
  else
    begin
      let key = q.keys.(0) in
      let prio = q.priorities.(0) in
      full_swap q 0 q.last;
      q.mapping.(key) <- -1;
      q.last <- q.last - 1;
      sift_down q 0;
      key, prio
    end
```

- **Question 7**

```
let decrease_priority q (x, prio) =
  let i = q.mapping.(x) in
  assert (mem q x && prio <= q.priorities.(i));
  q.priorities.(i) <- prio;
  sift_up q i
```

- **Question 8** Si l'on veut augmenter une priorité, il faut faire percoler la clé associée vers le bas. Ce n'est pas un problème, on pourrait facilement définir une fonction `update_priority` qui effectuerait l'une ou l'autre percolation suivant les cas.

- **Question 9** La fonction `mem` est, de manière évidente, en temps constant. Les trois autres fonctions effectuent une percolation (vers le haut ou vers le bas) et quelques opérations en temps constant. Les percolations sont clairement en temps $O(h)$, et dans le cas d'un arbre complet gauche on a $h = O(\log n)$. Par conséquent, les complexités sont en $O(\log n)$.

- **Question 10** C'est dans le cours...

- **Question 11** C'est une traduction vraiment directe du pseudo-code. On a juste remplacé le test $\text{dist}[k] \neq \infty$ par un `PrioQ.mem`, mais cela n'a pas d'importance.

```
let dijkstra g i =
  let n = Array.length g in
  let dist = Array.make n infinity in
  let queue = PrioQ.make_empty n in
  PrioQ.insert queue (i, 0.);
  dist.(i) <- 0.;
  while PrioQ.length queue >= 0 do
    let (j, d) = PrioQ.extract_min queue in
    let update (k, x) =
      let new_d = d +. x in
      if new_d < dist.(k) then begin
        dist.(k) <- new_d;
        if PrioQ.mem queue k then PrioQ.decrease_priority queue (k, new_d)
        else PrioQ.insert queue (k, new_d)
      end in
      List.iter update g.(j)
    done;
  dist
```

► Question 12 À nouveau, c'est essentiellement du cours.

```
let dijkstra_tree g i =
  let n = Array.length g in
  let dist = Array.make n infinity in
  let p = Array.make n None in
  let queue = PrioQ.make_empty n in
  PrioQ.insert queue (i, 0.);
  dist.(i) <- 0.;
  p.(i) <- Some i;
  while PrioQ.length queue >= 0 do
    let (j, d) = PrioQ.extract_min queue in
    let update (k, x) =
      let new_d = d +. x in
      if new_d < dist.(k) then begin
        dist.(k) <- new_d;
        p.(k) <- Some j;
        if PrioQ.mem queue k then PrioQ.decrease_priority queue (k, new_d)
        else PrioQ.insert queue (k, new_d)
      end in
    List.iter update g.(j)
  done;
  dist, p
```

► Question 13 C'est typiquement une question qui ne pose aucun problème sur machine mais sur laquelle il est facile de faire une erreur sur papier (chemin dans le mauvais ordre, extrémité manquante ou présente deux fois). Il faut s'entraîner à détecter ce type de problème en faisant les tests « de tête ».

```
let reconstruct_path p goal =
  let rec aux current =
    match p.(current) with
    | None -> failwith "no path"
    | Some i when i = current -> [current]
    | Some i -> current :: aux i in
  List.rev (aux goal)
```

► Question 14 Pour le fichier communes.csv, quelques commandes permettent d'obtenir l'ordre des colonnes et le délimiteur, et indiquent que les id correspondent aux numéros de ligne (en commençant à zéro) :

```
$ head --lines 4 communes.csv
id;insee;nom;departement;population
0;01001;L'Abergement-Clémenciat;1;784
1;01002;L'Abergement-de-Varey;1;221
2;01004;Ambérieu-en-Bugey;1;13835
$ tail --lines 3 communes.csv
35843;2B364;Zuani;2B;35
35844;2B365;San-Gavino-di-Fiumorbo;2B;174
35845;2B366;Chisa;2B;100
$ wc --lines communes.csv
35847 communes.csv
```

► Question 15 Plusieurs solutions sont possibles (faire une passe pour compter le nombre de lignes, créer un tableau ayant la bonne taille et le remplir dans un deuxième temps...). Ici, on a choisi d'utiliser une liste et de tirer parti du fait que les communes apparaissent dans l'ordre de leur id dans le fichier.

```
let lire_communes nom_fichier =
  let ic = open_in nom_fichier in
  let liste = ref [] in
  try
    (* On saute la première ligne (entête). *)
    let _ = input_line ic in
    while true do
      let s = input_line ic in
      let f id insee nom dep pop =
        {id; insee; nom; dep; pop} in
      let c = Scanf.sscanf s "%d;%s@;%s@;%s@;%d" f in
      liste := c :: !liste
    done;
    assert false
  with
  | End_of_file -> close_in ic; Array.of_list (List.rev !liste)
```

Remarque

On écrirait normalement `{id = id; insee = insee; ...}`, mais dans le cas où les noms de variables correspondent exactement aux noms des champs de l'enregistrement OCaml accepte la version plus courte. Ce n'est pas à retenir.

► **Question 16** Il faut simplement penser à ajouter l'arête aux deux listes d'adjacence, le graphe n'étant pas orienté.

```
let lire_graphe nb_communes fichier_adjacence =
  let ic = open_in fichier_adjacence in
  let g = Array.make nb_communes [] in
  try
    let _ = input_line ic in
    while true do
      let s = input_line ic in
      let x, y = Scanf.sscanf s "%d;%d" (fun x y -> (x, y)) in
      g.(x) <- y :: g.(x);
      g.(y) <- x :: g.(y)
    done;
    assert false
  with
  | End_of_file ->
    close_in ic;
    g
```

► **Question 17** On pourrait utiliser Dijkstra avec un graphe non pondéré (en mettant un poids unitaire à toutes les arêtes), mais c'est assez nettement moins efficace qu'un simple parcours en largeur (il y a un facteur $\log |V|$, et $|V|$ est de l'ordre de 200 000 ici). On fait donc un parcours en largeur, avec une exception pour s'arrêter dès qu'on trouve une commune acceptable.

```

exception Gagne of int

let saute_canton g init tab_communes =
  let f = Queue.create () in
  let n = Array.length g in
  let vus = Array.make n false in
  let arbre = Array.make n None in
  arbre.(init) <- Some init;
  Queue.add init f;
  vus.(init) <- true;
  try
    while not (Queue.is_empty f) do
      let i = Queue.pop f in
      if tab_communes.(i).pop >= 50_000 then raise (Gagne i);
      let rec ajoute j =
        if not vus.(j) then begin
          vus.(j) <- true;
          Queue.add j f;
          arbre.(j) <- Some i
        end in
        List.iter ajoute g.(i)
    done;
  []
with
| Gagne i -> reconstruct_path arbre i
  
```

► Question 18

```

let commune_perdue g tab_communes =
  let dmax = ref (-1) in
  let chemin_max = ref [] in
  let n = Array.length tab_communes in
  for i = 0 to n - 1 do
    let chemin = saute_canton g i tab_communes in
    if List.length chemin > !dmax then begin
      dmax := List.length chemin;
      chemin_max := chemin
    end
  done;
affiche tab_communes !chemin_max
  
```

On obtient un chemin de longueur 32 :

```

# commune_perdue g_adj tab_communes;;
Auderville (50) : 267
Jobourg (50) : 501
Omonville-la-Petite (50) : 137
Digulleville (50) : 291
Omonville-la-Rouge (50) : 524
...
Saint-Manvieu-Norrey (14) : 1729
Carpiquet (14) : 2374
Caen (14) : 108954
  
```

Remarque

Ce résultat n'est malheureusement pas à jour, la commune de Cherbourg-Octeville (par laquelle passe ce chemin) ayant récemment fusionné avec ses voisines pour devenir Cherbourg-en-Cotentin et dépasser ainsi la barre des 50 000 habitants. L'histoire ne dit pas comment les habitants d'Auderville ont réagi à la perte de leur titre de gloire.

► **Question 19** On construit un graphe orienté pondéré de la manière suivante : l'arc (x, y) a pour poids la population de la commune y . Le problème se ramène alors clairement à une recherche de chemin de poids minimal dans ce graphe, qui vérifie les conditions d'application de l'algorithme de Dijkstra (poids positifs).

► **Question 20** Faisons les choses bien, en créant un dictionnaire ayant pour clés les couples (nom, département) et pour valeurs les communes correspondantes (l'utilisation du département permet de s'affranchir des homonymes) :

```
let cree_dictionnaire_communes () =
  let dict = Hashtbl.create 30_000 in
  let ajoute c =
    if Hashtbl.mem dict (c.nom, c.dep) then failwith "couples non uniques ?";
    Hashtbl.add dict (c.nom, c.dep) c in
  Array.iter ajoute tab_communes;
  dict

let dict_communes = cree_dictionnaire_communes ()
```

On crée ensuite le graphe pondéré décrit à la question suivante :

```
let cree_graphe_pondere () =
  let n = Array.length tab_communes in
  let g = Array.make n [] in
  let traite_arc i j =
    g.(i) <- (j, float tab_communes.(j).pop) :: g.(i) in
  for i = 0 to n - 1 do
    List.iter (traite_arc i) g_adj.(i)
  done;
  g

let g_pond = cree_graphe_pondere ()
```

La fonction est alors immédiate à écrire :

```
let misanthrope init but =
  let c_init = Hashtbl.find dict_communes init in
  let c_but = Hashtbl.find dict_communes but in
  let _, arbre = dijkstra_tree g_pond c_init.id in
  reconstruct_path arbre c_but.id
```

On obtient :

```
# affiche (misanthrope ("Villeurbanne", "69") ("La Mulatière", "69"));;
Villeurbanne (69) : 145150
Rillieux-la-Pape (69) : 29952
Cailloux-sur-Fontaines (69) : 2480
Mionnay (1) : 2077
Civrieux (1) : 1362
Parcieux (1) : 1095
Quincieux (69) : 3002
Lucenay (69) : 1752
Marcy (69) : 634
Charnay (69) : 1076
Bagnols (69) : 695
Le Breuil (69) : 454
Bully (69) : 2069
Savigny (69) : 1938
Chevinay (69) : 543
Courzieu (69) : 1161
Yzeron (69) : 1036
Thurins (69) : 2917
Soucieu-en-Jarrest (69) : 3769
Chaponost (69) : 7978
Sainte-Foy-lès-Lyon (69) : 21742
La Mulatière (69) : 6480
```

Pour Montrouge-Aubervilliers, on va se contenter de la longueur :

```
# List.length (misanthrope ("Montrouge", "92") ("Aubervilliers", "93"));;
- : int = 67
```

Table des matières

Cours	6
1 Introduction au langage OCaml	6
1 Opérateurs de base	7
2 Variables	8
3 Définitions de fonctions	9
4 Fonctions récursives	10
5 Listes	11
2 Aspects fonctionnels de OCaml	13
1 Types de base	13
2 Types structurés	17
3 Filtrage	21
4 Définition de types	26
Exercices Supplémentaires	32
1 Version récursive des boucles <code>while</code>	32
2 Entraînement sur les listes	33
3 Approfondissement	35
Solutions	37
3 Aspects impératifs de OCaml	43
1 Type <code>unit</code>	43
2 Variables mutables	44
3 L'opérateur <code>;</code>	47
4 Boucles	48
5 Tableaux	50
Exercices	54
Solutions	56
4 Correction, terminaison	63
1 Spécification d'une fonction	63
2 Correction partielle, correction totale	64
3 Programmes itératifs	65
4 Programmes récursifs	69
Exercices Supplémentaires	72
Solutions	73
5 Introduction à la complexité	75
1 Notations mathématiques	75
2 Types de ressources, niveau de détail	76
3 Complexité en temps	77
4 Calculs de complexité	78
5 Complexité en espace	85
Exercices	87
Solutions	90
6 Structures de données	92
1 Type abstrait et implémentation	92
2 Structures de données fonctionnelles et impératives	95

TABLE DES MATIÈRES

3	Définition en OCaml	97
4	Exemple de spécification plus complexe	97
	Solutions	98
7	Piles et files	101
1	Piles	101
2	Files	103
3	File de priorité	107
4	Deque	107
	Solutions	108
8	Gestion de la mémoire	110
1	Mémoire d'un ordinateur	110
2	Organisation de la mémoire d'un processus	111
3	Portée d'un identifiant	113
4	Pointeurs en C	115
5	Durée de vie d'un objet	117
6	Pile d'appels	118
7	Allocation dynamique	125
	Exercices	133
1	Récursion terminale	133
9	Représentation des données	136
1	Entiers	136
2	Caractères et chaînes de caractères	146
3	Nombres en virgule flottante	148
	Solutions	157
10	Arbres	158
1	Introduction	158
2	Définitions	161
3	Parcours d'arbres binaires	164
4	Un peu de dénombrement	166
5	Arbres non binaires	169
	Exercices	172
	Solutions	174
11	Dictionnaires	180
1	Ensembles, dictionnaires	180
2	Arbres binaires de recherche	182
3	Réalisation d'un dictionnaire par un ABR	191
4	Arbres auto-équilibrés	192
5	Tables de hachage	203
	Exercices	207
	Solutions	208
12	Tas et files de priorité	221
1	Structure abstraite de file de priorité	221
2	Tas binaire	222
3	Tas impératif à l'aide d'un arbre implicite	224
4	Opérations sur un tas binaire	226
5	Tri par tas	229
	Solutions	230
13	Familles d'algorithmes	235
1	Diviser pour régner	235
2	Programmation dynamique	244
3	Algorithmes gloutons	254
4	Exemple de solutions exactes : problèmes d'ordonnancement	255

TABLE DES MATIÈRES

Exercices	258
Solutions	261
14 Graphes : aspects théoriques	266
1 Introduction	267
2 Graphes non orientés	273
3 Graphes orientés	280
Exercices	284
Solutions	288
15 Algorithmes sur les graphes	295
1 Représentation informatique d'un graphe	295
2 Parcours d'un graphe	298
3 Test d'acyclicité et tri topologique	307
4 Plus court chemin dans un graphe pondéré	309
Exercices	317
Solutions	321
16 Logique	333
1 Syntaxe des formules propositionnelles	333
2 Sémantique des formules propositionnelles	335
3 Formes normales	341
4 Problème SAT	343
5 Introduction à la logique du premier ordre	344
Solutions	345
Travaux pratiques	351
I Initiation à OCaml	351
1 Quelques fonctions élémentaires	351
2 Fonctions récursives	351
3 Manipulation de listes	352
4 Pour ceux qui ont fini	353
Solutions	355
II Manipulation de listes	361
Solutions	364
III Tableaux, ordre supérieur	370
1 Manipulations élémentaires de tableaux	370
2 Fonctions d'ordre supérieur : <code>fold</code>	372
3 Si vous avez fini	372
Solutions	374
IV Tri insertion, tri fusion	379
1 Recherche dichotomique	379
2 Tri	380
3 Tri fusion	384
4 Bonus	386
Solutions	387
V Analyse d'algorithmes	391
1 Suite de Fibonacci	391
2 Fonctions de comparaison	392
3 Étude de l'algorithme d'Euclide	393
4 Inégalité arithmético-géométrique	394
Solutions	395
VI Exercices d'entraînement	400

1	Coefficients binomiaux	400
2	Recherche dans une matrice triée	401
3	Combinatoire	401
	Solutions	402
VII	Ensembles et combinatoire	404
1	Ensembles représentés par des listes	404
2	Combinatoire	405
3	Pour chercher	407
	Solutions	408
VIII	Écrivons des programmes!	414
1	Introduction à la ligne de commande	414
2	Compilation et exécution d'un programme OCaml	416
3	Arguments en ligne de commande	417
IX	Petits problèmes d'algorithmique	418
1	Calculs élémentaires	418
2	Tri sélection d'une liste	419
3	Palindromes	420
4	Sections équilibrées	420
5	Question bonus	421
X	Piles, Files	422
1	Files fonctionnelles	422
2	Piles et files impératives	423
XI	Introduction au langage C	427
1	Hello, World!	427
2	Un programme plus complet	429
3	Instructions conditionnelles	431
4	Boucles	432
5	Petits exercices	434
6	Pour chercher	434
	Solutions	435
XII	Boucles et tableaux statiques	441
1	Boucles	441
2	Tableaux statiques	442
3	Arguments en ligne de commande	443
XIII	Un peu de dessin	445
1	Image RGB	445
2	Types utilisés	446
3	Format d'image utilisé	446
4	Création du fichier	447
5	Primitives simples	448
6	Mélange de couleurs	449
7	Tracé de lignes par l'algorithme de Bresenham	450
	Solutions	454
XIV	Pointeurs	463
1	Manipulation de pointeurs	463
2	Fonction scanf	465
3	Petits problèmes	467
XV	Pile d'appel, ensemble de Mandelbrot	469
1	Retour sur scanf	469
2	Quelques expériences sur la pile et les pointeurs	469
3	Fractale de Mandelbrot	472

4	Pour chercher	475
	Solutions	476
XVI	Tableaux dynamiques	481
1	Types enregistrement en C : les struct	481
2	Un tableau qui connaît sa taille	483
3	Tableaux dynamiques (ou vecteurs)	484
	Solutions	489
XVII	Exceptions en OCaml	495
1	Principe des exceptions	495
2	Exemple d'utilisation : lecture d'un fichier	497
3	Utilisation dans le flot de contrôle	499
4	Rien à voir	501
	Solutions	503
XVIII	Programmation d'un allocateur mémoire	507
1	Allocateur linéaire	508
2	Réservations de blocs de taille fixe	508
3	Zones mémoire avec en-tête et pied de page	510
4	Chaînage explicite des portions libres	511
XIX	Tri radix	513
1	Tri stable	513
2	Tri radix	515
	Solutions	519
XX	Expressions arithmétiques	524
1	Arbre d'une expression arithmétique	524
2	Différentes notations pour les expressions arithmétiques	525
3	Expressions avec variables	526
4	Forme normale pour l'associativité	527
	Solutions	528
XXI	Promenade sylvestre	532
1	Fonctions élémentaires sur les arbres	532
2	Parcours d'arbres	533
3	Adressage	536
4	Reconstruction d'un arbre	537
5	Au cas où	538
XXII	Structures de données chaînées en C	539
1	Listes simplement chaînées	539
2	Piles et files	542
3	Retour sur les listes chaînées	544
	Solutions	545
XXIII	Ce duc y parle	555
1	Structure de trie	555
2	Fonctions utilitaires	557
3	Opérations élémentaires sur les tries	557
4	Lecture de fichier	558
5	Filtrage	559
6	Décomposition en anagrammes	559
	Solutions	561
XXIV	Listes doublement chaînées	569
1	Première version	569
2	Version avec sentinelle	570
3	Nombres chanceux	571

TABLE DES MATIÈRES

Solutions	572
XXV Listes à accès direct	575
1 Nombres en binaire	575
2 Listes binaires à accès direct	576
3 Nombres en binaire décentré	578
Solutions	581
XXVI Arbres binaires de recherche en C	587
Solutions	590
XXVII Arbres binaires de recherche en OCaml	595
1 Fonctions élémentaires	595
2 Fonctions supplémentaires	596
3 Structure de multi-ensemble ordonné	596
4 Un problème pour finir	598
Solutions	599
XXVIII Arbres rouge-noir en OCaml	606
1 Insertion	606
2 Suppression	607
Solutions	610
1 Insertion	610
2 Suppression	611
XXIX Seam carving	614
1 Travailler avec des images	614
2 Détection de bords	616
3 Deux approches naïves	617
4 Seam carving	618
Solutions	620
XXX Arbres de Braun	627
1 Définition et propriétés élémentaires	627
2 Calcul de la taille	627
3 Réalisation d'un tas fonctionnel par un arbre de Braun	628
Solutions	629
XXXI Adressage ouvert	632
1 Constructeur, destructeur et recherche d'éléments	634
2 Parcours de la table	634
3 Ajout d'éléments	635
4 Suppression d'éléments	635
5 La liste des adresses IP	636
6 Une meilleure fonction de hachage	636
7 Sondage quadratique	637
Solutions	638
XXXII Autour du tri rapide	646
1 Tri rapide en C	646
2 Quickselect	646
3 Introsort	647
Solutions	648
XXXIII Deux exemples de diviser pour régner	653
1 Paire la plus proche	653
2 Nombre d'inversions	655
Solutions	656
XXXIV Programmation dynamique : le problème du sac à dos	662

TABLE DES MATIÈRES

1	Une relation de récurrence	662
2	Programmation dynamique ascendante et descendante	663
3	Reconstruction de la solution	663
4	De l'intérêt des deux approches	664
XXXV	Maximisation d'une somme	665
1	Introduction	665
2	Algorithme naïf	665
3	Récursion naïve	666
4	Solution en programmation dynamique	666
5	Optimisation	667
XXXVI	Plus longue sous-séquence croissante	668
1	Méthode par énumération	668
2	Méthode par programmation dynamique	669
3	Méthode de la <i>patience</i>	669
4	Bonus	670
	Solutions	671
XXXVII	Graphes eulériens	678
1	Condition nécessaire	678
2	Condition suffisante	679
3	Construction de chemins eulériens	679
	Solutions	681
XXXVIII	Parcours de graphes en OCaml	685
1	Parcours	685
2	Accessibilité, composantes connexes	686
3	Variantes des parcours	688
	Solutions	690
XXXIX	Clôture transitive	697
	Solutions	700
XL	Fichiers	705
1	Compléments sur le langage C	705
2	Gestion des fichiers (en OCaml et en C)	707
3	Prénoms français (en C)	708
4	Cent mille milliards de poèmes (en OCaml)	709
5	Décimales de π (en C)	709
6	Nombre de langages informatiques (langage libre)	711
7	Admissibles aux mines (langage libre)	711
XLI	Manipulation de formules logiques	712
1	Affichage d'une formule logique	712
2	Égalité syntaxique modulo associativité et commutativité	713
	Solutions	715
XLII	Autour de Dijkstra	718
1	File de priorité	718
2	Algorithme de Dijkstra	720
3	Calcul d'itinéraire pour misanthropes	720
	Solutions	723
	Table des matières	730