Informatique

Gaetan Bisson

https://gaati.org/bisson/

Introduction

Ce cours présente divers certains fondamentaux de l'informatique principalement liés à l'algorithmique et aux structures de données. Divers concepts théoriques sont abordés en s'appuyant sur la pratique de la programmation dans les langages Python, OCaml et SQL.

Cet enseignement unifie deux matières du programme officiel des classes préparatoires, l'informatique commune [38] et l'option informatique [39] qui font l'objet d'une épreuve commune à certains concours et d'épreuves séparées à d'autres.

Table des matières

1 Premier semestre **4**
- 1.1 Programmation impérative . . . 4
  - 1.1.1 Variables et types . . . 5
  - 1.1.2 Structures de contrôle . . . 8
  - 1.1.3 Fonctions . . . 8
  - 1.1.4 Thèmes unidimensionnels . . . 9
- 1.2 Programmation récursive . . . 11
  - 1.2.1 Expressions . . . 11
  - 1.2.2 Fonctions . . . 13
  - 1.2.3 Tuples . . . 15
  - 1.2.4 Filtrage . . . 15
  - 1.2.5 Listes . . . 16
  - 1.2.6 Références . . . 17
  - 1.2.7 Tableaux . . . 18
  - 1.2.8 Types . . . 20

2 Second semestre **22**
- 2.1 Représentation des nombres . . . 22
  - 2.1.1 Nombres entiers . . . 22
  - 2.1.2 Nombres réels . . . 24
  - 2.1.3 Matrices de pixels et images . . . 25
- 2.2 Méthodes de programmation et analyse des algorithmes . . . 28
  - 2.2.1 Méta programmation . . . 28
  - 2.2.2 Terminaison . . . 30
  - 2.2.3 Correction . . . 32
  - 2.2.4 Complexité . . . 33
- 2.3 Structures de données . . . 37
  - 2.3.1 Piles et files . . . 38
  - 2.3.2 Dictionnaires . . . 39
- 2.4 Types récursifs immuables et arbres . . . 41
  - 2.4.1 Listes . . . 41
  - 2.4.2 Arbres . . . 42
  - 2.4.3 Arbres binaires . . . 43
  - 2.4.4 Arbres de recherche . . . 46
- 2.5 Bases des graphes, plus courts chemins . . . 48
  - 2.5.1 Aspects théoriques . . . 48
  - 2.5.2 Aspects effectifs . . . 50
  - 2.5.3 Plus courts chemins . . . 51

2.6 Syntaxe puis sémantique de la logique propositionnelle 52

3 Troisième semestre **55**
3.1 Dictionnaires et programmation dynamique 55
3.1.1 Dictionnaires 55
3.1.2 Programmation dynamique 57
3.2 Applications des arbres 60
3.2.1 Retour sur trace 60
3.2.2 File de priorité 61
3.3 Graphes avancés 65
3.3.1 Arbres couvrants 65
3.3.2 Couplages des graphes bipartis 68
3.4 Bases de données 72
3.4.1 Problématique et langage SQL 72
3.4.2 Sélection et projection 74
3.4.3 Opérations ensemblistes 74
3.4.4 Jointure 75
3.4.5 Fonctions d'agrégats et sous requêtes 75
3.5 Déduction naturelle pour la logique propositionnelle 77
3.6 Algorithmique pour l'intelligence artificielle et l'étude des jeux 79
3.6.1 Intelligence artificielle 80
3.6.2 Étude des jeux 81

4 Quatrième semestre **84**
4.1 Motifs, automates et expressions 84
4.1.1 Motifs 84
4.1.2 Langages rationnels 86
4.1.3 Automates 91

A Feuilles de TD **100**
A.1 Premier semestre 101
A.2 Second semestre 104
A.3 Troisième semestre 109
A.4 Quatrième semestre 111

Bibliographie **115**

Chapitre 1

Premier semestre

Ce semestre est consacré à la pratique de la programmation dans ses paradigmes impératif en langage Python puis récursif en langage OCaml avec pour objectif de se familiariser avec les structures de données élémentaires fournies par les langages.

1.1 Programmation impérative

Aux concours vous programmerez sur papier mais, pendant l'année, lorsque comme aujourd'hui nous voudrons exécuter notre code, nous utiliserons une interface appelée Jupyter qui supporte les trois langages exigés. Connectez-y vous et créer une nouvelle feuille de calcul de type Python. Recopiez-y alors le programme ci-dessus et exécutez le.

```
n = 1
for k in range(1, 60 + 1):
    n = n * k
print(n)
```

Ce programme contient une boucle `for` où la variable n est successivement multipliée par tous les entiers $k = 1, 2, 3, \ldots, 60$; la valeur finalement affichée est ainsi 60!.

Remarque. *L'indentation, c'est-à-dire les espaces placés en début de ligne, joue un rôle essentiellement esthétique dans la plupart des langages de programmation. Ce n'est pas le cas en Python où elle sert à délimiter les blocs de code. Pour éviter des contresens majeurs il vous faudra y être particulièrement vigilant.*

Exercice. *Écrire un programme calculant la somme des entiers compris entre* 0 *et* 100.

Considérons à présent ce programme :

```
for x in range(1, 20):
    for y in range(1, 20):
        for z in range(1, 20):
            if x**2 + y**2 == z**2:
                print(x, y, z)
```

Chacune des trois boucles `for` imbriquées a pour effet de faire prendre successivement les valeurs $1, 2, \ldots, 19 (= 20 - 1)$ à sa variable ; au cœur de ces trois boucles, l'égalité $x^2 + y^2 = z^2$ est ainsi testée pour tous les triplets $(x, y, z) \in \{1, \ldots, 19\}^3$ et ceux la vérifiant sont affichés.

Exercice. *Écrire un programme affichant tous les entiers $n \in \{1, \dots, 100\}$ s'écrivant sous la forme $x^2 + y^2$ avec $(x, y) \in \mathbb{N}^2$.*

Exercice. *Déterminer le nombre de couples d'entiers $(x, y) \in \{0, \dots, 100\}^2$ pour lesquels $x + y$ et $x\,y$ sont simultanément des carrés.*

Votre aptitude à la programmation est certes évaluée aux concours mais c'est surtout le socle sur lequel les aspects théoriques de ce cours reposent ; il est difficile de surestimer à quel point elle conditionne votre réussite aux épreuves d'informatique. Pour progresser en programmation, pas de secret, il faut pratiquer très régulièrement : réécrivez les programmes vus en cours, modifiez les, améliorez les, créez en de nouveaux, etc.

Pour pratiquer hors campus, le plus simple est d'utiliser un interpréteur en ligne, tels :

Python :	`https://www.online-python.com/`
OCaml :	`https://try.ocamlpro.com/`
SQL :	`https://sqliteonline.com/`

Passons à présent en revue les bases du langage Python.

1.1.1 Variables et types

Les variables sont un concept de programmation qui associe à un identifiant une valeur stockée en mémoire. Python est un langage typé, ce qui signifie que chaque variable possède un type contraignant ses valeurs et les opérations qu'on peut lui appliquer. C'est de surcroit un langage orienté objet et appelant ainsi objets ses variables et classes ses types.

```
a = 1
b = 2.3
c = "quatre"
type(a)     # <class 'int'>
type(b)     # <class 'float'>
type(c)     # <class 'str'>
a + b       # 3.3
b + c       # TypeError
```

L'avant-dernière ligne montre que le type d'une valeur est parfois implicitement converti afin de permettre l'évaluation d'une expression.

Sera qualifié d'expression toute combinaison de constantes, variables, opérations et fonctions pouvant être évaluée afin d'obtenir une valeur. Cette notion s'oppose à celle d'instruction qui lui est similaire mais dont l'exécution ne renvoie rien. Par exemple :

```
1 + 1   # expression
a = 1   # instruction
a == 1  # expression
del(a)  # instruction
abs(a)  # expression
```

Seuls les types et opérations ci-après sont supposés connus aux épreuves des concours et il est inutile de pousser plus avant votre connaissance du langage Python. Pour la complétude de ce cours, mentionnons néanmoins la documentation officielle du langage [31] et en l'occurrence à son chapitre *Built-in Types* consultable à l'adresse :

`https://docs.python.org/3/library/stdtypes.html`

Nombres entiers : type `int`. On dispose notamment des opérateurs usuels d'addition (`+`), de soustraction (`-`), de multiplication (`*`), de quotient (`//`), de reste (`%`) et de puissance (`**`). Leur évaluation suit les règles de priorité usuelles. On aura par exemple :

```
1 + 2 * 3   # 7
1 + 2 ** 3  # 9
1 + 2 // 3  # 1
1 + 2 % 3   # 3
```

Nombres flottants : type `float`. Les opérateurs sont identiques à ceux des entiers auxquels s'ajoutent celui de division réelle (`/`). On aura ainsi :

```
1 + 2 / 3     # 1.6666666666666665
2 ** (1 / 2)  # 1.4142135623730951
```

Les opérations sur les nombres entiers et réels sont imparfaites car le nombre d'atomes dans l'univers (donc *a fortiori* dans votre ordinateur) est fini et ne permet donc pas de représenter ne serait-ce que $\mathbb{N}$ fidèlement. Nous décrirons plus tard ces imperfections en détail.

Booléens : type `bool`. Effectue sur les deux valeurs `True` et `False` les opérations logiques `not`, `and` et `or`. Ces deux dernières sont évaluées de manière paresseuse, c'est-à-dire qu'en fonction de la valeur du premier argument le second n'est pas nécessairement évalué : si $f(x) = F$ alors il est inutile d'évaluer $g(x)$ pour déduire $f(x) \wedge g(x) = F$.

```
not True        # False
True and False  # False
True or False   # True
```

On prendra garde à ne pas confondre l'opérateur d'égalité (`==`) avec l'opérateur d'affectation (`=`). On dispose en outre des opérateurs de différence (`!=`), inférieur strict (`<`), supérieur strict (`>`), inférieur large (`<=`) et supérieur large (`>=`). Toute comparaison renvoie un booléen.

```
1 + 2 > 3  # False
```

Contrairement aux types ci-dessus, ceux ci-après ne découlent pas directement des fonctionnalités offertes par le processeur et exigent donc un travail logiciel qui pour l'instant vous est néanmoins caché. On parle alors de structures de données.

Tuples et chaînes de caractères : types `tuple` et `str`. Les tuples implantent la notion mathématique de famille aussi appelée n-uplet. Les chaînes n'en sont que le cas particulier où chaque élément est un caractère, mais elles possèdent une syntaxe légèrement différente. La longueur n d'une telle structure s'obtient par la fonction `len()`. Ses éléments sont indicés par les entiers $k \in \{0, 1, 2, \ldots, n-1\}$ et on y accède directement grâce aux crochets `x[k]`. On parle de structures indicées immuables car on ne peut pas en modifier les éléments après création de la structure.

```
s = "Polynésie"
len(s)          # 9
s[5]            # 'é'
s[9]            # IndexError: string index out of range
t = ("PF", 987, 8.38)
len(t)          # 3
t[2]            # 8.38
t[2] = 119.332  # TypeError: object does not support item assignment
```

Les opérateurs ci-après semblent manipuler ces structures mais, puisqu'aucune modification n'est possible, ils créent en réalité de nouvelles structures de toutes pièces et y recopient alors les éléments des structures de départ ; ces opérations peuvent donc s'avérer coûteuses. Il s'agit de la concaténation « `+` », la répétition « `*` » et de la tranche `x[a:b]` où a et b dénotent deux indices chacun optionnel.

```
t + t   # ('PF', 987, 8.38, 'PF', 987, 8.38)
3 * s   # 'PolynésiePolynésiePolynésie'
s[3:6]  # 'yné'
s[:6]   # 'Polyné'
s[6:]   # 'sie'
```

Listes : type `list`. On crée une liste par extension `[a,b,c]` ou par compréhension `[f(x) for x in S if g(x)]` où la clause `if` est optionnelle. Ce qui précède concernant les structures indicées immuables s'applique identiquement à ceci près qu'on peut de surcroit modifier une liste : en ajoutant un élément à la fin `L.append()` ; en enlevant le dernier élément `L.pop()`.

```
L = [p**2 for p in range(1, 10) if p % 2 == 1]
L              # [1, 9, 25, 49, 81]
len(L)         # 5
L[2]           # 25
L.pop()        # [1, 9, 25, 49]
L.append(91)   # [1, 9, 25, 49, 91]
L[2] = 0       # [1, 9, 0, 49, 91]
L = L + [101]  # [1, 9, 0, 49, 91, 101]
L[1:3]         # [9, 0 ]
```

On gardera à l'esprit que les opérations des structures immuables sont généralement moins efficaces ; la concaténation ou l'extraction de tranche, par exemple, crée une nouvelle liste et y recopie les éléments souhaités. L'affectation se contente quant à elle de passer une référence et il faut explicitement copier une liste `L.copy()` lorsqu'on souhaite en obtenir un nouvel exemplaire indépendant.

```
L = [1, 9, 25, 49, 81]
M = L;        M[2] = 0; L  # [1, 9, 0, 49, 81]
M = L.copy(); M[2] = 0; L  # [1, 9, 25, 49, 81]
```

Ensembles et dictionnaires : types `set` et `dict`. La notion mathématique d'ensemble correspond au type Python éponyme.

```
E = {1, 2, 4, 8, 16}
type(E)      # <class 'set'>
E | {2, 32}  # {16, 1, 2, 32, 4, 8}
E & {2, 32}  # {2}
```

Ce type admet pour extension celui de dictionnaire qui correspond à la notion mathématique de fonction. L'appartenance $x \in \mathrm{dom}(f)$ s'écrit `x in f`, l'évaluation $f(x)$ se note `f[x]` et on étend une fonction par `f[x] = y`.

```
f = {"abc": 3, "de": 2}
f["abc"]    # 3
f["def"]    # KeyError: "def"
f["f"] = 1
for x in f:
    print(x, f[x])
```

1.1.2 Structures de contrôle

Les structures de contrôle déterminent l'ordre d'exécution des instructions. Elles incluent notamment : les branchements conditionnels, les boucles « tant que », les boucles « pour tout » et les fonctions qui feront l'objet de la prochaine section. Rappelons que l'indentation, c'est-à-dire l'ajout d'espaces en début de ligne afin d'obtenir un alignement vertical précis, joue un rôle crucial en Python : c'est ainsi que sont identifiées les instructions assujetties à un branchement ou à une boucle.

Les branchements conditionnels. Ils permettent d'exécuter un bloc d'instructions lorsqu'une condition est ou n'est pas satisfaite ; leur forme la plus générale est la suivante :

```
if expression1:
    instruction_si_expression1_vraie
elif expression2:
    instruction_si_expression2_vraie
elif expression3:
    instruction_si_expression3_vraie
else:
    instruction_sinon
instruction_ensuite
```

L'usage de `elif` garantit que seul un bloc d'instructions conditionné à une expression soit exécuté, ce qui peut ne pas être le cas lorsqu'on utilise des `if` successifs, notamment si les instructions conditionnées à la première expression modifient la valeur de la seconde expression.

Les instructions itératives. Communément appelées boucles, elles exécutent un même bloc de code plusieurs fois. La boucle « `for x in L:` » exécute le bloc qui la suit en donnant successivement à la variable `x` les valeurs des éléments de `L` qui peut être un tuple, une chaîne, une liste ou un dictionnaire. En lieu d'une liste explicite, la fonction `range(a,b)` énumère les entiers $\{a, a+1, \ldots, b-1\}$, avec $a = 0$ lorsque cet argument est omis. La boucle `while` exécute quant à elle le bloc qui la suit tant que la condition est satisfaite.

Exercice. *Construire la liste des nombres premiers inférieurs à 1000.*

```
L = []
n = 1000
for k in range(2, n + 1):
    premier = True
    for d in range(2, k):
        if k % d == 0:
            premier = False
    if premier:
        L.append(k)
```

L'instruction `break` interrompt immédiatement l'exécution de la dernière boucle lancée. On pourrait ici la rajouter après la ligne « `premier = False` » afin d'interrompre la boucle portant sur `d` dès qu'on a trouvé un diviseur car on est alors certain que `k` n'est pas premier.

1.1.3 Fonctions

La notion de fonction en informatique ne présente qu'une ressemblance trompeuse avec son homonyme en mathématiques. Il s'agit en réalité de morceaux de programmes dont on a clairement défini les entrées et les sorties dans le but de les réutiliser aisément. Par exemple :

```
def somme(a, b, c):
    r = a + b + c
    return r
```

Les variables `a`, `b` et `c` portent le nom d'arguments ou encore de paramètres. Remarquer qu'on n'a pas eu besoin d'en définir le type ; cette fonction s'appliquera donc aussi bien à des listes qu'à des entiers. Le mot clef `def` s'utilise pour définir une fonction et `return` renvoie le résultat et termine l'exécution.

Remarquons que toute variable déclarée à l'intérieur d'une fonction lui est locale, c'est-à-dire que la variable `r` définie dans la fonction ci-dessus n'existe pas en dehors de celle-ci. On peut donc réutiliser les noms typiques tels `i` et `n` sans craindre que cela porte à confusion.

Exercice. *Écrire des fonctions qui calculent :*
- *la factorielle ;*
- *la suite de Fibonacci ;*
- *la liste des diviseurs d'un entier ;*
- *la liste des n plus petits nombres premiers.*

On peut écrire des fonctions récursives, c'est-à-dire s'appelant elles-mêmes, telle :

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)
```

Programmer ainsi facilite l'implantation de nombreuses notions mathématiques, mais en complexifie l'exécution, car la gestion de ces appels a un coût, ainsi que l'analyse, car les appels récursifs peuvent ne pas terminer tels `factorielle(-1)` et `factorielle(3/2)`.

Les épreuves écrites des concours imposent généralement de ne pas utiliser de fonctions récursives en Python et de ne pas utiliser d'instruction itérative en OCaml. On s'y attachera dans ce cours sauf indication contraire.

1.1.4 Thèmes unidimensionnels

Afin de consolider votre maîtrise de la programmation en Python nous allons traiter des problèmes variés relevant de thèmes classiques en commençant par les tableaux qui, dans ce langage, sont confondus avec les listes.

Exercice. *Écrire une fonction `appartient(objet,liste)` renvoyant `True` si l'objet est élément de la liste et `False` sinon. Écrire une fonction `maximum(liste)` renvoyant le plus grand élément. Écrire une fonction `secondmaximum(liste)` renvoyant le deuxième plus grand élément.*

Remarquons que le temps d'exécution de ces fonctions est proportionnel à la longueur de la liste puisqu'ils se contentent essentiellement de la parcourir. Rappelons aussi que Python offre directement l'idiome « `objet in liste` » et les fonctions `min()`, `max()`, `sum()`, etc.

Exercice. *Écrire une fonction `plusproches(liste)` renvoyant les deux éléments les plus proches.*

Le temps d'exécution est ici proportionnel au carré de la longueur de la liste.

Lorsque la liste donnée est triée, ce qui peut être réalisé par la méthode `liste.sort()`, on peut réaliser certains opérations telle la recherche d'éléments plus rapidement par la méthode de la dichotomie ; cela consiste à découper la liste en deux, comparer l'objet recherché à l'élément du milieu, puis à chercher l'objet dans la première moitié de la liste s'il lui est inférieur et dans la seconde moitié sinon.

Exercice. *Supposant que la liste donnée est triée, écrire une version plus efficace de la fonction* `appartient(objet,liste)`.

```
def appartient(objet, liste):
    debut = 0
    fin = len(liste)
    while fin - debut > 1:
        milieu = (debut + fin) // 2
        if objet < liste[milieu]:
            fin = milieu
        else:
            debut = milieu
    return objet == liste[debut]
```

Le temps d'exécution est ici proportionnel au logarithme de la longueur de la liste.

Exercice. *Écrire une fonction* `facteur(mot,chaine)` *déterminant si le mot y apparaît.*

Lorsque les consignes le permettront, il pourra être opportun d'avoir recours à la programmation récursive, même en langage Python ; ce serait notamment le cas pour les deux problèmes ci-après qu'il est assez malcommode de résoudre dans le paradigme impératif.

Exercice. *Écrire une fonction qui prend en argument une liste et qui renvoie la liste de ses sous listes.*
Écrire une fonction qui prend en argument une liste et qui renvoie la liste de ses permutations.

Un algorithme glouton résout un problème en faisant, à chaque étape, le choix localement maximal, sans se soucier de son optimalité globale. Par exemple, pour rendre 47 francs en pièces de 1,2,5,10,20 et 50 francs, on choisi d'abord la plus grosse pièce inférieure à ce montant, soit 20 francs ; il reste ainsi 27 francs à rendre et on choisi à nouveau une pièce de 20 francs ; il reste enfin 7 francs à rendre et on choisi alors une pièce de 5 francs puis une de 2 francs.

Exercice. *Écrire un programme qui prend comme argument un entier x et une liste d'entiers et qui renvoie une sous liste dont la somme fait x.*

Restons toutefois conscient du fait que cette approche ne marche pas inconditionnellement : l'entier 6 est la somme d'une sous liste de $[1, 3, 3, 4]$ qu'aucun algorithme glouton ne trouvera.

On discutera des avantages et inconvénients des méthodes de tri ci-après.

Exercice (tri par sélection). *Écrire une fonction déterminant le plus petit élément d'une liste. Échanger cet élément avec celui en première position. Itérer ce processus afin d'obtenir une méthode triant la liste entière.*

Exercice (tri par insertion). *Écrire une fonction insérant un élément dans une liste déjà triée. En déduire une méthode de tri qui procède en insérant le premier élément de la liste donnée dans sa queue récursivement triée.*

Exercice (tri à bulles). *Écrire une fonction parcourant une liste et intervertissant tout couple d'éléments consécutifs rencontrés qui ne sont pas dans l'ordre croissant. En déduire une méthode de tri répétant cette procédure jusqu'à ce que la liste soit triée.*

Exercice (tri rapide). *Écrire une fonction découpant une liste* `L` *en deux sous listes : l'une formée des éléments inférieurs à* `L[len(L) // 2]` *(élément appelé « pivot ») et l'autre formée des éléments supérieurs. En déduire une méthode de tri qui procède en découpant ainsi une liste puis en concaténant les deux sous listes récursivement triées.*

Exercice (tri fusion). *Écrire une fonction découpant une liste en deux. Écrire une fonction fusionnant deux listes déjà triées. En déduire un méthode de tri procédant en découpant la liste donnée puis en fusionnant les deux sous listes récursivement triées.*

1.2 Programmation récursive

En langage Python, nous avons pratiqué la **programmation impérative**, c'est-à-dire que notre code consistait en une suite d'instructions, chacune modifiant l'état du programme. Cette approche est naturelle car c'est ainsi que tout processeur fonctionne.

Certains langages permettent néanmoins de programmer autrement. Avec OCaml [33], nous adopterons certains idiomes de **programmation fonctionnelle** : notre code consistera exclusivement en des définitions de fonctions, leur composition, leur évaluation (notamment récursive).

Remarque. *Aux concours c'est sur le papier que vous mettrez ce langage en œuvre, donc de petites erreurs de syntaxe seront tolérées : c'est l'algorithmique et non pas la programmation qui est évaluée. En revanche, de grosses fautes dénotant une méconnaissance du langage seront pénalisantes.*

1.2.1 Expressions

Le double point-virgule « `;;` » indique la fin d'un bloc de code dans l'interpréteur ; bien qu'inutile partout ailleurs, notamment dans l'interface Jupyter et avec le compilateur, son usage reste répandu aux concours. On écrira donc :

```
# 1 ;;
- : int = 1

# 1.1 ;;
- : float = 1.1
```

La réponse de l'interpréteur prend la forme « `nom : type = valeur` » ; on prendra l'habitude de la consulter attentivement afin de vérifier que notre code a été interprété comme prévu.

```
# 1 + 2. ;;
Error: This expression has type float but
         an expression was expected of type int
```

Passons en revue les principaux types élémentaires.

Entiers relatifs : type `int`. On dispose des opérateurs d'addition (`+`), de soustraction (`-`), de multiplication (`*`), de quotient (`/`) et de reste (`mod`). Leur évaluation suit les règles de priorité.

```
# 4 + 5 / 2 ;;
- : int = 6

# 4 + 5 mod 2 ;;
- : int = 5
```

Nombres réels : type `float`. Ses opérateurs sont distincts de ceux portant sur les entiers et comportent `+.`, `-.`, `*.`, `/.`, `**`, `sqrt`, `sin`, `cos`, `exp` et `log`.

```
# 1. /. 3. ;;
- : float = 0.333333333333333315
```

Rappelons encore que les opérations sur les nombres entiers et réels sont imparfaites car votre ordinateur est une machine finie ; ces imperfections seront détaillées au second semestre.

Chaînes de caractères : type `string`. Ne sont autres que des tableaux de caractères avec un type et une syntaxe spécifiques qui sont toutefois immuables.

```
# String.length "coucou" ;;
- : int = 6

# "coucou".[0] ;;
- : char = 'c'

# "coucou" ^ "toi" ;;
- : string = "coucoutoi"
```

Booléens : type `bool`. Ses éléments se notent `true` et `false` sans majuscules et disposent des opérateurs `not`, `&&` et `||`.

```
# true || not true ;;
- : bool = true
```

Les opérateurs de comparaison, `=`, `<>`, `<=`, `>=`, `<` et `>`, s'appliquent à des arguments de tout type.

```
# 0 <> 1 ;;
- : bool = true

# "abc" < "def" ;;
- : bool = true
```

Remarque. *En langage OCaml, les opérateurs `==` et `!=` sont de faux amis : ils testent respectivement l'égalité et la différence des adresses mémoires.* ***On prendra garde à ne jamais les utiliser !***

Le branchement conditionnel « `if` condition `then` expression `else` expression » fonctionne comme on s'y attendrait ; la condition peut être toute expression de type `bool` et les deux expressions doivent être de même type.

```
# if 1 <> 2 then "trois" else "quatre" ;;
- : string = "trois"
```

Afin de convertir les types, on dispose de fonctions nommées *`type`*`_of_`*`type`*.

```
# int_of_float 65.4321 ;;
- : int = 65
```

On déclare des variables globales par le mot clef `let` et locales en rajoutant `in`.

```
# let x = 2 ;;
val x : int = 2

# x ;;
- : int = 2

# let y = 1 in y + 1 ;;
- : int = 2

# y ;;
Error: Unbound value y
```

Contrairement au paradigme de programmation impérative, les variables ne sont pas mutables, c'est-à-dire qu'on ne peut pas modifier leur valeur, seulement en créer de nouvelles.

1.2.2 Fonctions

Les fonctions se déclarent en invoquant le mot clef `function` comme fait ci-dessous. Elles consomment les expressions qui les suivent en guise d'arguments. Les parenthèses servent quant à elles uniquement à délimiter ces expressions lorsque cela s'avère nécessaire.

```
# let f = function n -> 2 * n + 1 ;;
val f : int -> int = <fun>

# f 2 ;;
- : int = 5

# f 2 + 1 ;;
- : int = 6

# f (2 + 1) ;;
- : int = 7
```

Exemple. *Pour tester la parité d'un entier on pourrait écrire :*

```
# let pair = function n -> if n mod 2 = 0 then true else false ;;
val pair : int -> bool = <fun>
```

Ou encore plus simplement :

```
# let pair = function n -> n mod 2 = 0 ;;
val pair : int -> bool = <fun>
```

Exercice. *Écrire deux fonctions valeur absolue : une pour les entiers, une pour les réels.*

L'idiome « `let f = function x -> ...` » est suffisamment lourd et courant pour mériter la forme abrégée « `let f x = ...` ». Les deux définitions ci-dessous sont ainsi parfaitement équivalentes.

```
# let f = function n -> 2 * n + 1 ;;
val f : int -> int = <fun>

# let f n = 2 * n + 1 ;;
val f : int -> int = <fun>
```

L'intérêt de la forme abrégée est particulièrement clair lorsqu'il s'agit de définir des fonctions admettant plusieurs arguments, comme en témoignent les définitions équivalentes suivantes.

```
# let argument = function x -> function y -> atan (y /. x) ;;
val argument : float -> float -> float = <fun>

# let argument x y = atan (y /. x) ;;
val argument : float -> float -> float = <fun>
```

Le type de la fonction `argument` ainsi déclarée, aussi appelé signature, est un abus de notation pour « `float -> (float -> float)` », OCaml omettant les parenthèses lorsqu'elles donnent priorité aux flèches à droite. Le langage interprète donc cette fonction comme allant de $\mathbb{R}$ vers $\mathbb{R}^{\mathbb{R}}$ alors qu'on aurait plutôt tendance à considérer qu'elle va de $\mathbb{R} \times \mathbb{R}$ vers $\mathbb{R}$. Ces deux points de vue sont équivalents puisque les ensembles $\left(E^F\right)^G$ et $E^{F \times G}$ sont en bijection canonique ; et c'est justement l'essence même de la programmation fonctionnelle que d'adopter le premier point de vue. Ainsi, si `f` est une fonction admettant deux arguments, l'expression « `f a` » dénotera la fonction $b \mapsto f(a, b)$. On aura ici :

```
# let g = argument 1. ;;
val g : float -> float = <fun>
```

```
# g 0. ;;
- : float = 1.57079632679489656
```

On parle de polymorphisme lorsqu'une fonction n'a pas de type spécifié pour un ou plusieurs de ses arguments, ce que la signature indique par des variables de type notées « `'a` », « `'b` » et ainsi de suite.

```
# let identite = function x -> x ;;
val identite : 'a -> 'a = <fun>

# let egalite x y = x = y ;;
val egalite : 'a -> 'a -> bool = <fun>

# let evalue x = function y -> x y ;;
val evalue : ('a -> 'b) -> 'a -> 'b = <fun>
```

Les opérateurs de comparaisons ne sont d'ailleurs autres que des fonctions polymorphes révélées par l'usage de parenthèses :

```
# (<) ;;
- : 'a -> 'a -> bool = <fun>
```

Exemple. *On peut inculquer à OCaml la dérivation des fonctions numériques comme il suit.*

```
# let derivee f =
      let epsilon = 0.0001 in
      function x -> (f (x +. epsilon) -. f x) /. epsilon
  ;;
val derivee : (float -> float) -> float -> float = <fun>

# derivee sqrt 1. ;;
- : float = 0.499987500623966241
```

Exercice. *Déterminer les types des fonctions suivantes :*

```
# let f x = x 1. + 1 ;;

# let g x y = x.[y + 1] ;;

# let compose x y = function z -> x (y z) ;;
```

Les définitions récursives sont un concept central au langage OCaml ; elles doivent néanmoins être explicitement indiquées comme telle par le mot clef `rec`. Par exemple, pour définir la fonction factorielle, on écrirait naturellement :

```
# let rec factorielle n = if n < 2 then 1 else n * factorielle (n - 1) ;;
val factorielle : int -> int = <fun>
```

Exercice. *Écrire une fonction calculant 2^n en fonction de l'entier n.*

Exercice. *Écrire des fonctions calculant :*

- *la suite de Fibonacci ;*
- *le plus grand diviseur commun ;*
- *la quantité x^n en fonction du réel x et de l'entier n ;*
- *si un entier donné est une puissance de deux ;*
- *si un entier donné est premier (utiliser une fonction auxiliaire) ;*

(Dorénavant nous omettrons souvent les éléments d'interface tels que l'invite ou le code de retour afin de nous concentrer sur le code OCaml proprement dit.)

1.2.3 Tuples

Les tuples s'écrivent « `(x,y,z)` », les parenthèses pouvant souvent être omises, et le type produit cartésien associé est noté « `'a * 'b * 'c` ». On peut récupérer les éléments d'un tuple via `let` comme il suit.

```
let x = (1, 2, 'c') ;;
let a, b, c = x in a ;;
```

On s'efforcera d'employer cette notion avec parcimonie car elle entrave l'utilisation de nombreux idiomes de programmation fonctionnelle. Considérons par exemple les deux fonctions :

```
let f a b = a * b ;;
let g (a, b) = a * b ;;
```

Leurs signatures sont « `f : int -> int -> int` » et « `g : int * int -> int` » ; en particulier, seule la première se prête à l'évaluation partielle : on peut écrire « `f 2` » pour obtenir l'opération de doublement mais rien de tel avec `g`.

Tuple vide : type `unit`. Le tuple vide « `()` », unique élément de type `unit`, est utilisé pour dénoter l'absence de valeur. Par exemple, la fonction « `print_string : string -> unit` » affiche une chaîne de caractères à l'écran et renvoie `()`. Les expressions de type `unit` correspondent ainsi aux instructions du langage Python. Afin d'en enchaîner plusieurs, on écrit :

```
let _ = print_string "un\n" in
let _ = print_string "deux\n" in
print_string "trois\n"
;;
```

Le symbole « `_` » est une variable spéciale acceptant tout mais ne stockant rien. Une autre possibilité, plus concise, est d'utiliser le connecteur « `;` ». Le code ci-dessous est interprété comme celui ci-dessus, à ceci près qu'un avertissement est émis lorsqu'une expression précédant un connecteur « `;` » n'est pas de type `unit`.

```
print_string "un\n";
print_string "deux\n";
print_string "trois\n"
;;
```

Le connecteur « `;` » est de priorité plus faible que `if-then-else` ; autrement dit, lorsqu'un tel enchaînement n'est pas délimité par les mots clefs `begin` et `end` ou par des parenthèses, seule la première expression suivant `then` et `else` est assujettie au branchement. Comparer :

```
if 1 = 2 then print_string "un\n"; print_string "deux\n" ;;
if 1 = 2 then ( print_string "un\n"; print_string "deux\n" ) ;;
```

1.2.4 Filtrage

Le filtrage est un mécanisme similaire au branchement conditionnel mais sensiblement plus esthétique et, nous le verrons prochainement, plus puissant. N'écrivons plus :

```
if n = 1 then f(n) else if n = 2 then g(n) else h(n)
```

Écrivons :

```
match n with
| 1 -> f(n)
| 2 -> g(n)
| _ -> h(n)
```

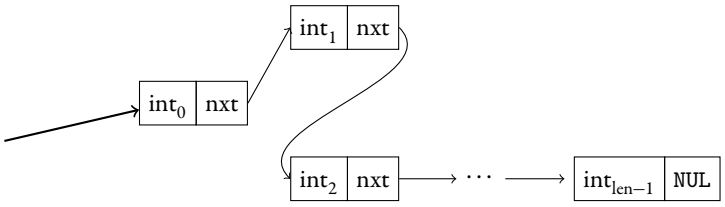


FIGURE 1.1 – Une liste chaînée d'entiers.

Le mot clef `function` effectue un filtrage implicite, c'est-à-dire que l'expression « `function x -> match x with` » est un pléonasme : il suffisait d'écrire « `function` ». Grâce à ce mécanisme, certaines fonctions sont particulièrement agréables à coder.

```
let rec factorielle = function
    | 0 -> 1
    | n -> n * factorielle (n - 1)
;;
```

1.2.5 Listes

En guise de liste, Python offre une structure de données hybride assez complexe dont la polyvalence est certes commode en programmation occasionnelle mais qui rend difficile toute analyse de performance. OCaml dispose quant à lui de véritables listes et tableaux.

Définition. *Une liste chaînée est une famille d'objets de même type stockés en mémoire à des emplacements arbitraires, chacun indiquant l'adresse de son successeur. Elle est identifiée par le type de ses éléments et l'adresse du premier élément. Voir la figure 1.1.*

Cette construction permet, en temps constant, d'ajouter ou d'enlever un élément en tête de liste. En revanche, accéder ou modifier le k^{e} élément nécessite de parcourir préalablement tous ceux qui le précèdent. En langage OCaml, les listes se notent `[a; b; c]` et peuvent aussi être créées par les opérateurs d'ajout « `::` » et de concaténation « `@` ».

```
# let l = 0 :: 1 :: 2 :: [] ;;
val l : int list = [0; 1; 2]
# let m = l @ l ;;
val m : int list = [0; 1; 2; 0; 1; 2]
# (List.hd m, List.tl m) ;;
- : int * int list = (0, [1; 2; 0; 1; 2])
```

Toute liste non vide est de la forme `head::tail` et cette dichotomie est la source du paradigme de manipulation des listes en langage OCaml. Par exemple, une fonction idiomatique est :

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

Exercice. *Écrire des fonctions de signature :*

- *`somme : int list -> int`*
- *`maximum : 'a list -> 'a`*
- *`moyenne : int list -> float`*

— *`appartient : 'a -> 'a list -> bool`*
— *`enlever : int -> 'a list -> 'a list` (enlever l'élément d'indice donné)*
— *`ajouter : int -> 'a -> 'a list -> 'a list` (ajouter à l'indice donné)*

Le langage OCaml dispose des fonctions natives :

```
List.init    : int -> (int -> 'a) -> 'a list
List.length  : 'a list -> int
List.mem     : 'a -> 'a list -> bool
List.exists  : ('a -> bool) -> 'a list -> bool
List.for_all : ('a -> bool) -> 'a list -> bool
List.filter  : ('a -> bool) -> 'a list -> 'a list
List.map     : ('a -> 'b) -> 'a list -> 'b list
List.iter    : ('a -> unit) -> 'a list -> unit
```

Précisons que :
— « `List.init n f` » renvoie la liste $[f(0); f(1); \ldots; f(n-1)]$;
— « `List.exists P L` » détermine si au moins un élément de L satisfait P ;
— « `List.filter P L` » renvoie la liste formée des éléments de L satisfaisant P ;
— « `List.map f L` » renvoie la liste formée des images par f des éléments de L ;
— « `List.iter f L` » évalue l'expression de type `unit` f sur chaque élément de L.

Exercice. *Implanter ces fonctions.*

Armé de fonctions auxiliaires, ce paradigme de manipulation des listes permet de traiter tout problème d'algorithmique. Par exemple, afin de couper une liste en deux.

```
let rec aux a b = function
    | []          -> (a, b)
    | [ x ]       -> (x :: a, b)
    | x :: y :: t -> aux (x :: a) (y :: b) t
;;

let decoupe = aux [] [] ;;
```

Exercice. *Écrire une fonction « `range : int -> int list` » prenant en argument un entier n et renvoyant la liste $[0; 1; 2; \ldots; n-1]$ formée des n plus petits entiers. Écrire ensuite une fonction « `inverse : 'a list -> 'a list` » qui, étant donnée une liste $[x_0; x_1; \ldots; x_n]$, renvoie la liste $[x_n; \ldots; x_1; x_0]$.*

1.2.6 Références

Les variables mutables sont appelées références : elles sont déclarées explicitement par le mot clef `ref` ; l'affectation puis l'accès aux valeurs se fait alors par les opérateurs `:=` et `!` comme ceci.

```
# let a = ref 0 ;;
val a : int ref = {contents = 0}
# a := 1 ;;
- : unit = ()
# !a + 1 ;;
- : int = 2
```

Avec l'usage de références, l'emploi de boucles redevient pertinent. Leur syntaxe en OCaml est « `for var = 0 to 9 do ... done` » où les bornes sont atteintes et « `while condition do ... done` ». On peut donc écrire :

```
let fibo n =
    let a = ref 0 in
    let b = ref 1 in
    for k = 1 to n do
        let c = !a + !b in
        a := !b;
        b := c
    done;
    !a
;;
```

Il est toutefois plus élégant d'adopter une approche fonctionnelle pour réaliser ce calcul.

```
let rec aux a b = function
    | 0 -> a
    | n -> aux b (a + b) (n - 1)
;;

let fibo = aux 0 1 ;;
```

Cette fonction auxiliaire n'étant destinée qu'à l'usage de `decoupe`, on peut la définir à l'intérieur de cette dernière. Cela en restreint la portée afin d'éviter toute confusion dans l'espace global.

```
let fibo =
    let rec aux a b = function
        | 0 -> a
        | n -> aux b (a + b) (n - 1)
    in
    aux 0 1
;;
```

Remarque. *De manière générale, l'usage de variables globales est à proscrire. Leur portée est typiquement dynamique en programmation impérative et statique en programmation fonctionnelle ; concrètement, les programmes Python et OCaml ci-dessous renverront respectivement* 4, 2, 4 *et* 2.

```
a = 1
f = lambda x : a*x
a = 2
f(2)
```

```
let a = 1 ;;
let f x = a*x ;;
let a = 2 ;;
f 2 ;;
```

```
let a = ref 1 ;;
let f x = !a*x ;;
a := 2 ;;
f 2 ;;
```

```
let a = ref 1 ;;
let f x = !a*x ;;
let a = ref 2 ;;
f 2 ;;
```

On peut donc définir de multiples fonctions auxiliaires toutes appelées `aux` *dans l'espace global OCaml à condition que chaque fonction précède directement celle qui l'utilise.*

1.2.7 Tableaux

Définition. *Un tableau est une famille d'objets de même type stockés en mémoire à des emplacements contigus. Il est identifié par le type de ses éléments, l'adresse du premier élément et le nombre total d'éléments appelé longueur du tableau. Voir la figure 1.2.*

Cette construction permet, en temps constant, de consulter ou de modifier des éléments d'indice arbitraire : l'élément d'indice k est directement accessible à l'adresse :

$$\texttt{addr}(x_k) = \texttt{addr}(x_0) + k \cdot \texttt{sizeof}(\text{type})$$

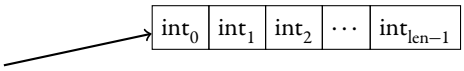


FIGURE 1.2 – Un tableau d'entiers.

En revanche, on ne peut généralement ni ajouter ni enlever des éléments sans procéder à une recopie complète du tableau, opération très coûteuse. En langage OCaml, les tableaux se notent `[|a; b; c|]` et, contrairement aux tuples, sont mutables, c'est-à-dire que leur contenu peut être modifié : le k^{e} élément se note `t.(k)` et sa valeur peut être changée par l'opérateur `<-` comme il suit.

```
# let t = Array.make 5 1 ;;
val t : int array = [|1; 1; 1; 1; 1|]
# for k = 0 to 4 do t.(k) <- k done ;;
- : unit = ()
# t ;;
- : int array = [|0; 1; 2; 3; 4|]
# Array.length t ;;
- : int = 5
```

On peut ainsi écrire :

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

Exercice. *Écrire des fonctions de signature :*

- *`somme : int array -> int`*
- *`maximum : 'a array -> 'a`*
- *`moyenne : int array -> float`*
- *`appartient : 'a -> 'a array -> bool`*
- *`renverser : 'a array -> 'a array` (mêmes éléments en ordre inverse)*

Le langage OCaml dispose des fonctions natives suivantes. On remarquera que, à l'exception de la première, chacune possède un homonyme opérant sur les listes qui a déjà été évoqué.

```
Array.make    : int -> 'a -> 'a array
Array.init    : int -> (int -> 'a) -> 'a array
Array.length  : 'a array -> int
Array.mem     : 'a -> 'a array -> bool
Array.exists  : ('a -> bool) -> 'a array -> bool
Array.for_all : ('a -> bool) -> 'a array -> bool
Array.map     : ('a -> 'b) -> 'a array -> 'b array
Array.iter    : ('a -> unit) -> 'a array -> unit
```

Exercice. *Implanter ces fonctions.*

Exercice. *Écrire des fonctions de signature :*

— *indices : 'a array -> 'a -> int list (liste des indices où l'élément apparait)*
— *concat2 : 'a array -> 'a array -> 'a array (concatène deux tableaux)*
— *concatn : 'a array list -> 'a array (concatène une liste de tableaux)*

Pour des tableaux multidimensionnels attention à ne pas tomber dans le piège suivant.

```
# let x = Array.make 4 (Array.make 4 0) ;;
val x : int array array =
  [|[|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]|]
# x.(1).(1) <- 1 ;;
- : unit = ()
# x ;;
- : int array array =
[|[|0; 1; 0; 0|]; [|0; 1; 0; 0|]; [|0; 1; 0; 0|]; [|0; 1; 0; 0|]|]
```

Le tableau `x` ainsi créé comprend quatre exemplaires du même tableau `[| 0; 0; 0; 0 |]`, ce qui explique le phénomène observé. On s'attendait à avoir quatre tableaux distincts, ce pour quoi il aurait fallu utiliser la commande dédiée `Array.make_matrix`.

Exercice. *Écrire des fonctions calculant la somme, la différence, le produit, le déterminant et l'inverse de matrices carrées.*

1.2.8 Types

L'introduction de nouveaux types permet de résoudre certains problèmes très élégamment et efficacement. Le langage OCaml propose deux mécanismes pour ce faire.

Types enregistrements. On peut définir un nouveau type comme produit cartésien de types existants, c'est-à-dire essentiellement comme un tuple nommé. Cette construction est notamment pertinente pour construire les nombres complexes :

```
type complex = { re : float; im : float } ;;
let i = { re = 0.; im = 1. } ;;
i.re ;;

let produit w z =
    let a = w.re *. z.re -. w.im *. z.im in
    let b = w.re *. z.im +. w.im *. z.re in
    { re = a; im = b }
;;
```

Exercice. *Écrire des fonctions calculant la somme, la différence, le produit, le quotient, le conjugué, le module et l'argument de nombres complexes.*

Types énumérés (ou sommes, ou unions). On peut définir un nouveau type comme union disjointe de types existants. Cette construction est pertinente lorsque les objets du nouveau type peuvent être regroupés en plusieurs catégories distinctes. Prenons pour exemple les droites du plan :

```
type droite_du_plan =
    | Verticale of float        (* abscisse                    *)
    | Oblique of float * float  (* pente, ordonnée à l'origine *)
;;
```

```
let paralleles d d' =
    match (d, d') with
    | Verticale _,    Verticale _    -> true
    | Oblique (p, _), Oblique (p', _) -> p = p'
    | _ -> false
;;
```

On peut enfin mélanger types enregistrements et types énumérés, par exemple pour déterminer si un point du plan d'affixe donné est incident à une droite donnée.

```
let incident z = function
    | Verticale a    -> z.re = a
    | Oblique (p, o) -> z.im = z.re *. p +. o
;;
```

Exercice. *Écrire une fonction déterminant si deux droites du plan sont perpendiculaires. Écrire une fonction renvoyant l'affixe du point d'intersection de deux droites du plan données.*

Lorsqu'il nous est demandé de déterminer le point d'intersection de deux droites parallèles, ou encore le minimum d'une liste vide, renvoyer une erreur (notamment par filtrage partiel) empêche le traitement de cette situation *a posteriori*. On peut plutôt utiliser la construction suivante, déjà implantée dans OCaml, qui sert à indiquer qu'on dispose soit d'une valeur de type `'a` soit d'aucune valeur.

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

let rec minimum = function
    | []          -> None
    | [ x ]       -> Some x
    | x :: y :: t -> minimum (min x y :: t)
;;
```

Tout code utilisant cette fonction récupèrera ainsi un résultat de type `'a option` qu'il s'agira de filtrer pour spécifier que faire en cas d'absence de valeur ou de présence de valeur.

Chapitre 2

Second semestre

Ce semestre reprend les notions de programmation abordées au premier semestre en développant leurs aspects techniques et dans une moindre mesure théoriques. Il doit amener à une prise de recul significative par rapport aux langages de programmation.

2.1 Représentation des nombres

L'information numérique transite sous forme de signaux électriques interprétés comme suites de bits, unité élémentaire d'information prenant pour valeur 0 ou 1. Les bits sont groupés par paquets de huit, appelés octets, chacun admettant $2^8 = 256$ valeurs possibles. Pour en dénoter de grandes quantités, on utilise les préfixes du système international d'unités : une machine de bureau comporte typiquement une mémoire de capacité 1 To, une carte réseau de débit 10 Mo/s et un processeur de fréquence 1 Gop./s.

Pour traiter ces données, le processeur dispose de trois grandes familles d'instructions, chacune opérant sur des mots machines de 64 bits :

— **Les opérations logiques** sont exactes : elles réalisent sur l'ensemble $\{0,1\}^{64}$ la (puissance cartésienne de la) conjonction, la disjonction, la négation, le décalage, etc.
— **Les opérations arithmétiques** imitent l'arithmétique de $\mathbb{Z}$.
— **Les opérations numériques** imitent l'arithmétique de $\mathbb{R}$.

Ceci nécessite de coder des nombres comme éléments de $\{0,1\}^{64}$; on ne peut ainsi réaliser qu'une imparfaite approximation de l'arithmétique des ensembles infinis que sont $\mathbb{Z}$ et $\mathbb{R}$.

2.1.1 Nombres entiers

Afin de coder les entiers naturels comme familles de bits, on a recours au résultat ci-dessous dans le cas particulier $B = 2$ que l'on qualifie de binaire.

Définition. *Soit $B \geqslant 2$ un entier naturel. On appelle représentation ou décomposition de l'entier naturel $n \in \mathbb{N}$ en base B l'unique famille $(c_0, c_1, \dots, c_{k-1}) \in \{0, \dots, B-1\}^k$ vérifiant $c_{k-1} \neq 0$ et*

$$n = c_0 B^0 + c_1 B^1 + \cdots + c_{k-1} B^{k-1} = \sum_{i=0}^{k-1} c_i B^i.$$

L'être humain possède dix doigts et s'est donc habitué à travailler en base dix ; les bits (contraction de l'anglais *binary digit*) sont l'équivalent des chiffres en base deux. Remarquer qu'ici l'ordre des chiffres est inversé par rapport à l'usage courant : les unités se placent en première position, à l'indice zéro.

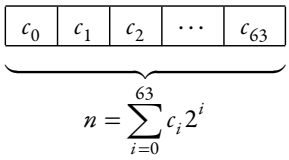


FIGURE 2.1 – Entiers de type `uint64`.

Les processeurs opèrent donc en base deux sur des entiers naturels n pour lesquels $k \leqslant 64$ soit $n \in \{0, \ldots, 2^{64} - 1\}$; ces entiers naturels sont codés suivant le type `uint64` illustré par la figure 2.1. Lorsque le résultat d'une opération dépasse $2^{64} - 1$, c'est son reste modulo 2^{64} que l'on obtient ; autrement dit, c'est l'arithmétique de $\mathbb{Z}/2^{64}\mathbb{Z}$ qui est implantée, non celle de $\mathbb{Z}$. Le langage OCaml utilise directement les entiers du processeur, ce qui donne :

```
# let rec puis = function 0 -> 1 | n -> 2*puis (n-1) in puis 64 ;;
- : int = 0
```

Pour représenter les entiers relatifs, on pourrait réserver le premier bit au stockage du signe et consacrer les 63 suivants à la décomposition binaire de la valeur absolue. La méthode retenue par le type `int64` est plus astucieuse : l'entier $n \in \left\{-2^{63}, \ldots, 2^{63} - 1\right\}$ est représenté comme le codage en `uint64` de n si $n \geqslant 0$ et de $(2^{64} + n)$ si $n < 0$; l'entier zéro n'a ainsi qu'une seule représentation et les opérations d'addition, de soustraction et de multiplication s'effectuent pour le type `int64` avec le même circuit que pour `uint64`.

Ces deux codages sont sujets aux problèmes de dépassement. Les interfaces de plus haut niveau sont quant à elles libres d'adopter l'un des comportements suivants :

— utiliser ces entiers machines tels quels : `2**64==0`
— pareil mais en détectant les dépassements : `2**64==overflow`
— représenter $\mathbb{Z}$ fidèlement en utilisant des listes d'entiers machines

Cette dernière option demande un travail logiciel significatif appelé arithmétique multiprécision, semblable à ce que nous faisons lorsque nous calculons à la main avec des entiers dans le système décimal ; voir [37] pour un ouvrage de référence sur ce sujet.

Exercice. *Écrire une fonction calculant la puissance d'un entier par un autre.*

La méthode naïve calcule x^n comme produit de n copies de x. Pour $n = 2^k$, il est avantageux de procéder par carré successifs, c'est-à-dire avec $x^n = \cdots((x^2)^2)\cdots$, ce qui nécessite $k = \log_2(n)$ multiplications seulement. Pour n quelconque, on se ramène au cas précédent en écrivant la décomposition binaire de n :

$$n = \sum_{i=0}^{k} a_i 2^i \quad \Longrightarrow \quad x^n = \prod_{i \in \{0,\ldots,k\}} x^{a_i 2^i}$$

Cette technique, appelée exponentiation rapide, obtient x^n en au plus $2\log_2(n)$ opérations. On peut l'implanter comme il suit en OCaml.

```
let rec decomposition = function
    | 0 -> []
    | n -> (n mod 2) :: decomposition (n / 2)
;;
```

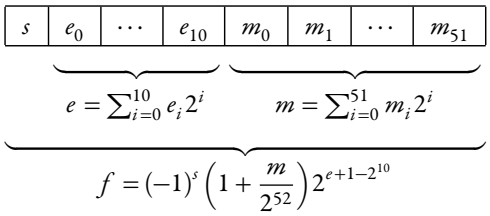


FIGURE 2.2 – Réels de type `binary64`.

```
let rec aux x = function
    | [] -> 1
    | 0 :: t -> aux (x * x) t
    | 1 :: t -> x * aux (x * x) t
;;

let puissance x n = aux x (decomposition n) ;;
```

Ou de manière plus concise en combinant en une ces deux fonctions.

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

2.1.2 Nombres réels

Le codage des nombres réels le plus répandu est régit par le standard IEEE 754 [29] qui stipule qu'un mot de 64 bits se décompose en un bit de signe, onze bits d'exposant et 52 bits de mantisse (avec un bit dominant 1 implicite) comme l'illustre la figure 2.2.

Cette représentation est dite à virgule flottante car, le nombre de bits étant constant, la précision (bits après la virgule) est inversement proportionnelle à la taille des nombres représentés (bits avant la virgule). Plus précisément, les nombres représentés entre 2^{63} et 2^{64} sont exactement les entiers ; entre 2^{62} et 2^{63} ce sont les demi entiers ; entre 2^{61} et 2^{62} les quarts d'entiers, etc. Inversement, autour de zéro, toute la précision porte sur les bits après la virgule.

Les nombres flottants souffrent :

— des problèmes d'erreur d'arrondi
— des problèmes de comparaison au zéro
— des problèmes de dépassement déjà évoqués dans le cas des entiers

Concrètement on aura, dans la très grande majorité des langages (ici Python) :

```
>>> 0.3 - 0.2 - 0.1
-2.7755575615628914e-17
>>> 0.1 ** 1000 > 0.0
False
>>> 10.0 ** 1000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
```

Remarque. *On peut être conduit à prendre ces problèmes à la légère, notamment par habitude d'utiliser une calculatrice. Il est cependant dangereux de les sous-estimer : c'est une erreur de dépassement qui a causé l'explosion de la fusée Ariane 5G le 4 juin 1996.*

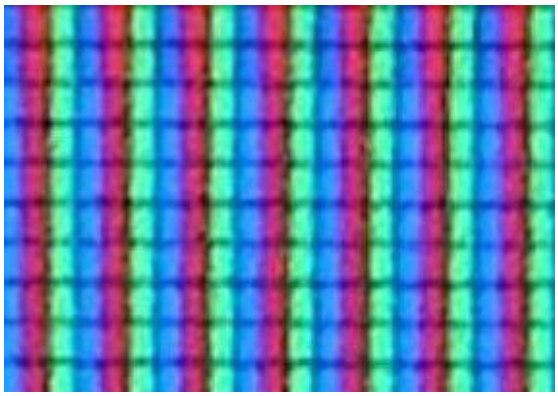


FIGURE 2.3 – Détail d'une matrice de pixels d'un écran LCD.

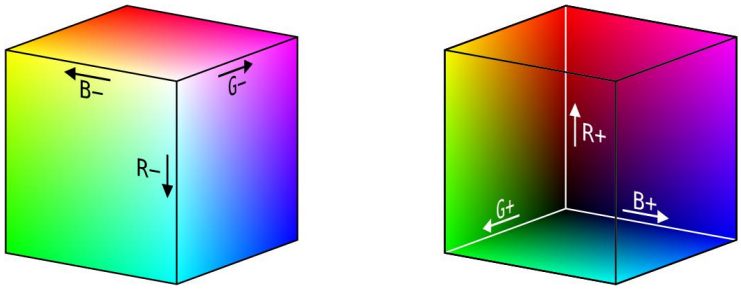


FIGURE 2.4 – Le cube RGB à gauche et ses faces arrières à droite.

2.1.3 Matrices de pixels et images

En guise d'application, on se propose de manipuler des images numériques. Une telle image est représentée comme un tableau bidimensionnel : le coefficient d'indice $[y, x]$ représente la couleur du pixel d'ordonnée y et d'abscisse x, l'origine $[0, 0]$ étant par convention le coin supérieur gauche de l'image.

Le codage RGB est le plus répandu pour représenter les couleurs : les écrans LCD l'utilisent nativement comme on le voit sur la figure 2.3. Il consiste à coder chaque couleur comme un triplet $(r, g, b) \in [0, 1]^3$ donnant les intensités de rouge, de vert et de bleu qui la composent ; cela donne la palette de la figure 2.4. On a par exemple :

$(0, 0, 0)$	noir	$(1, 0, 0)$	rouge	$(1, 1, 0)$	jaune
$(0.5, 0.5, 0.5)$	gris	$(0, 1, 0)$	vert	$(1, 0, 1)$	magenta
$(1, 1, 1)$	blanc	$(0, 0, 1)$	bleu	$(0, 1, 1)$	cyan

Ces intensités sont typiquement stockées sur un octet chacune ; le nombre de couleurs qu'il est ainsi possible de représenter est 256^3 soit plus de seize millions.

On utilisera la bibliothèque `matplotlib` comme il suit pour charger puis afficher une image.

```
import matplotlib.pyplot as plt
import matplotlib.image as img
```

```
a = img.imread("lenna.png")
plt.imshow(a)
plt.show()
```

Et après avoir potentiellement modifié une telle image on pourra la sauvegarder ainsi.

```
img.imsave("lenna2.png", a)
```

Ces images sont stockées comme des tableaux tridimensionnels de type `numpy.ndarray` et de taille *hauteur* × *largeur* × 3 ; chaque coefficient est de type `float32` et représente l'intensité d'une couleur primaire dans un pixel. La taille du tableau `a` est stockée dans l'attribut `a.shape`.

Exercice. *Décrire la transformation que le code ci-dessous applique à l'image.*

```
n, m, c = a.shape
for y in range(n):
    for x in range(100, 200):
        a[y][x][1] = 0
        a[y][x][2] = 0
```

L'objectif de cette section étant la pratique de la programmation, nous adopterons la définition naïve ci-dessous à fins de simplification, même si la réalité est plus complexe notamment car l'œil humain est plus sensible aux nuances de vert que de rouge ou de bleu.

Définition. *La luminosité de la couleur* $(r, g, b) \in \{0, \dots, 255\}^3$ *est la quantité* $\frac{r+g+b}{3}$.

Exercice. *Écrire un programme prenant en argument une image en couleurs et la transformant de sorte qu'elle soit en nuances de gris tout en préservant la luminosité de chaque pixel.*

```
def griser(a):
    n, m, _ = a.shape
    for y in range(n):
        for x in range(m):
            r, g, b = a[y][x]
            l = (r + g + b) / 3
            a[y][x] = (l, l, l)
```

Exercice. *Pour tenter d'améliorer la qualité visuelle d'une image, on se propose d'en augmenter le contraste en appliquant à chaque pixel l'opération* $(r, g, b) \mapsto (n(r), n(g), n(b))$ *avec* $n(x) = \frac{x-\alpha}{\beta-\alpha}$ *où* α *et* β *dénotent respectivement la plus faible et la plus forte des luminosités des pixels de l'image. Implanter cette opération.*

```
def augmenter(a):
    n, m, _ = a.shape
    e, f = (1, 0)
    for y in range(n):
        for x in range(m):
            r, g, b = a[y][x]
            l = (r + g + b) / 3
            e = min(e, l)
            f = max(f, l)
    for y in range(n):
        for x in range(m):
            for p in range(3):
                c = (a[y][x][p] - e) / (f - e)
                a[y][x][p] = min(1, max(0, c))
```

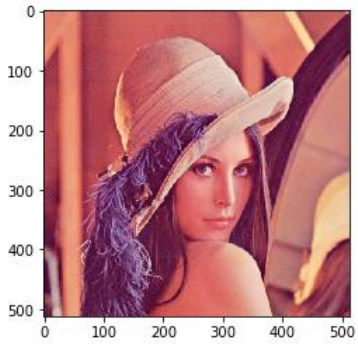
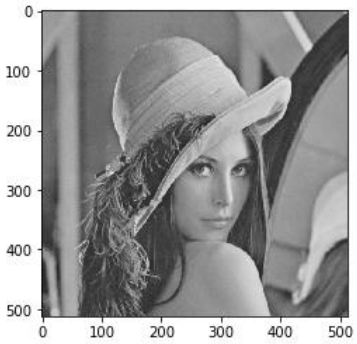
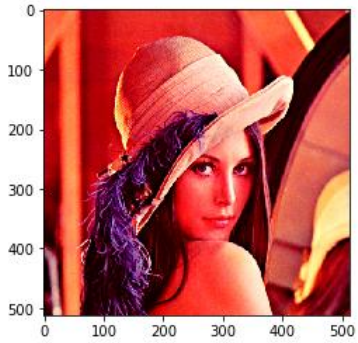


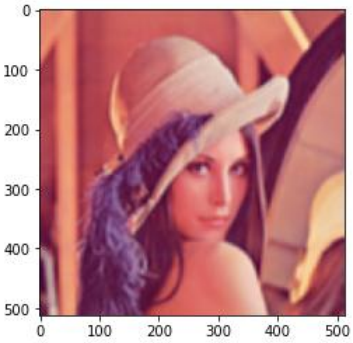
Image originale.



Nuances de gris.



Augmentation du contraste.



Floutage naïf.

FIGURE 2.5 – Quelques opérateurs de traitement d'images appliqués à une célèbre photographie de Lenna Sjööblom.

Exercice. *On propose de flouter une image en remplaçant la couleur de chaque pixel par la moyenne des couleurs des pixels qui en sont à distance au plus s, un entier paramétrable par l'utilisateur. Implanter cette opération.*

```
def flouter(a, s):
    n, m, _ = a.shape
    b = a.copy()
    for y in range(n):
        for x in range(m):
            for p in range(3):
                c, d = (0, 0)
                for j in range(max(0, y - s), min(n, y + s)):
                    t = int(numpy.sqrt(s ** 2 - (y - j) ** 2))
                    for i in range(max(0, x - t), min(m, x + t)):
                        c += b[j][i][p]
                        d += 1
                a[y][x][p] = c/d
```

Voir la figure 2.5.

Exercice. *Implanter des fonctions de redimensionnement. On créera une image vierge avec :*

```
b = numpy.ndarray((n, m, 3), dtype=numpy.float32)
```

2.2 Méthodes de programmation et analyse des algorithmes

Nous allons maintenant présenter des techniques ayant pour objectif d'améliorer la qualité des programmes. Une première approche consiste à rajouter au code des éléments inertes mais le rendant plus contraint et verbeux.

Une seconde approche repose sur des techniques d'analyse théorique. Nous parlerons alors d'algorithmes : un algorithme est l'essence d'un programme indépendamment du langage de programmation dans lequel il est écrit. Étant donné un algorithme $\mathscr{A}$ prenant en argument un paramètre x, nous traiterons trois questions :

— L'exécution de $\mathscr{A}(x)$ termine t-elle ?
— Les données renvoyées par $\mathscr{A}(x)$ sont-elles correctes ?
— Comment varie le temps d'exécution de $\mathscr{A}(x)$ en fonction de x ?

2.2.1 Méta programmation

Les principaux outils qu'offrent les langages de programmation afin d'améliorer la qualité du code (en matière de lisibilité, de maintenance et de correction) sont les suivants.

Fonctions. L'usage de fonctions est primordial : il permet de partitionner une grande quantité de code en une collection de petites procédures intelligibles, indépendantes et réutilisables. Toute fonction Python (resp. OCaml) dépassant dix (resp. cinq) lignes gagnerait probablement à être découpée en plusieurs sous fonctions. Par exemple :

```
def diviseurs(n):
    return [d for d in range(1, n + 1) if n % d == 0]

def premier(n):
    return diviseurs(n) == [1, n]
```

Commentaires. Le mécanisme le plus primitif pour documenter son code sont les commentaires. En Python, ils sont introduits par le caractère « `#` » qui a pour effet de faire ignorer du langage tout ce qui le suit sur sa ligne. En OCaml, on délimite des commentaires par les marqueurs `(*` et `*)`.

Assertions et exceptions. Ce dispositif consiste à vérifier la véracité d'expressions et d'interrompre la chaîne d'exécution si elles sont infirmées. Ceci permet d'éviter qu'une cascade d'erreurs passe inaperçue. Considérons par exemple :

```
def quotient(n, d):
    q = 0
    while n >= d:
        n = n - d
        q = q + 1
    return q
```

Pour prévenir les dysfonctionnements de cette fonction Python en présence d'arguments négatifs, on rajouterait avant la boucle :

```
    assert n >= 0 and d > 0
```

En OCaml, on rajouterait en début de fonction :

```
    if n < 0 || d <= 0 then failwith "argument negatif" else
```

Jeux de tests. Les tests permettent de s'assurer du bon fonctionnement d'un programme et de détecter d'éventuelles régressions en cours de développement. Ils peuvent être automatisés mais nous nous contenterons ici d'une approche naïve. Par exemple, en Python :

```
def verifier(fonction, tests):
    for a, r in tests:
        assert r == fonction(*a)

tests = [((9, 3), 3), ((9, 2), 4), ((2, 9), 0)]
verifier(quotient, tests)
```

On prendra soin d'inclure des tests couvrant largement toutes les parties de l'espace des arguments, notamment les cas limites.

Effets de bord. Une fonction est dite à effet de bord lorsqu'elle modifie des variables extérieures à son environnement. Soient par exemple les deux programmes Python ci-dessous :

```
def ajouter1(L): return L + [1]
L = []
L = ajouter1(L)
```

```
def ajouter2(L): L.append(1)
L = []
ajouter2(L)
```

Tous deux ont pour résultat de modifier la liste de sorte qu'on ait $L = [1]$. La première fonction ne modifie pas L, elle renvoie une nouvelle liste créée comme concaténation de L et de $[1]$ qui est alors affectée explicitement à la variable L. En revanche, la seconde fonction utilise un effet de bord : elle modifie la valeur de la variable L non seulement dans son corps de fonction mais aussi à l'extérieur. On évitera généralement les effets de bords car ils compliquent l'analyse des programmes et peuvent avoir des conséquences perverses.

Bibliothèques. Une bibliothèque logicielle, parfois aussi appelé module, n'est autre qu'une collection de programmes destinés à être utilisés comme sous routines d'autres programmes. En Python, on les charge par l'instruction « `import library` » et ses fonctions sont alors accessibles sous la syntaxe `library.function()`. On dispose aussi de l'instruction « `import library as lib` » qui rend les fonctions accessibles sous la syntaxe `lib.function()`. On peut enfin accéder directement à une fonction spécifique par `function()` grâce à l'appel « `from library import function` ». Certaines constantes et fonctions mathématiques élémentaires sont par exemples accessibles ainsi :

```
>>> import numpy as np
>>> np.pi
3.141592653589793
>>> np.sqrt(2)
1.4142135623730951
>>> np.sin(np.pi)
1.2246467991473532e-16
```

En OCaml, il n'est pas nécessaire de les charger explicitement, on se contente d'appeler directement leurs fonctions par la syntaxe `Module.fonction`.

Fichiers. L'utilisation de fichiers permet de stocker de manière organisée un grand volume d'informations. Leur manipulation se fait de manière globalement similaire quelque soit le langage de programmation : ou ouvre d'abord le fichier, en précisant son nom et si on souhaite le lire, l'écrire ou le créer ; on effectue alors les opérations proprement dites ; on ferme enfin le fichier. En Python on écrirait :

```
f = open("bidule.txt", "x")
f.write("123\n")
f.write("abc\n")
f.close()
f = open("bidule.txt", "r")
contenu = f.read()
f.close()
```

La fonction `open()` accepte comme second argument les modes `"r"` (lecture), `"w"` (écriture), `"x"` (création et écriture) et `"a"` (ajout). La fonction `read()` renvoie l'intégralité du contenu du fichier et l'expression « `contenu.split("\n")` » renvoie alors la liste des lignes.

Exercice. *Écrire une fonction qui prend en argument un entier N et le nom d'un fichier et qui y écrit la liste des nombres premiers inférieurs à N.*

Les fonctions `readline()` et `readlines()` renvoient respectivement la ligne suivante et la liste de toutes les lignes. Toutefois, elles incluent le caractère de retour à la ligne « `\n` » qui les termine. La construction suivante utilise `readline()` en interne :

```
f = open("bidule.txt", "r")
for line in f:
    print(line[:-1])
f.close()
```

Exercice. *Écrire une fonction qui prend en argument le nom d'un fichier contenant un réel par ligne et qui renvoie leur moyenne et leur écart type. On convertira une chaîne de caractères en nombre flottant grâce à la fonction `float()`.*

2.2.2 Terminaison

La seule obstruction potentielle à la terminaison d'un algorithme est la présence d'appels récursifs ou de boucles `while`. Pour les traiter, nous utiliserons le principe d'induction qui généralise la méthode de démonstration par récurrence à des ensembles autres que celui des entiers naturels. Commençons par quelques rappels sur la notion d'ensemble ordonné.

Définition. *Un ordre est une relation binaire réflexive, transitive et antisymétrique.*

Lorsque toute partie non vide admet un plus petit élément, on dit que c'est un bon ordre.

Lorsque toute partie non vide admet un élément minimal, on dit qu'il est bien fondé ; formellement, cela s'écrit :

$$\forall F \in \mathfrak{P}(E) \setminus \{\varnothing\}, \exists x \in F, \forall y \in F, y \leqslant x \Rightarrow y = x$$

Lemme. *Un ordre est bon si et seulement s'il est total et bien fondé.*

Puisque nous serons amenés à travailler avec des ordres partiels, nous parlerons désormais d'ordres bien fondés, y compris dans le cas total.

Exemple. *On a notamment :*

- *L'ordre naturel de $\mathbb{N}$ est bien fondé.*
- *L'ordre naturel de $\mathbb{Z}$ n'est pas bien fondé.*
- *L'ordre naturel de $\mathbb{R}_+$ n'est pas bien fondé.*
- *L'ordre lexicographique de $\mathbb{N}^2$ est bien fondé.*
- *La divisibilité dans $\mathbb{N}^*$ est un ordre bien fondé.*
- *L'inclusion dans $\mathfrak{P}(\{1, \dots, n\})$ est un ordre bien fondé.*

— *L'inclusion dans $\mathfrak{P}(\mathbb{N})$ n'est pas un ordre bien fondé.*

Un critère équivalent mais plus facile à vérifier en pratique est qu'un ensemble ordonné est bien fondé si et seulement s'il n'admet aucune suite infinie strictement décroissante. La preuve de cette équivalence nécessite l'axiome du choix, ce qui dépasse le cadre de ce cours. On l'admettra tout comme le théorème ci-dessous, qui généralise le principe de récurrence.

Théorème (principe d'induction). *Soit un ensemble E muni d'un ordre bien fondé $\leqslant$. Si F est une partie contenant tout élément minimal de E et vérifiant*

$$\forall y \in E, (\forall x \in E, x < y \Rightarrow x \in F) \Rightarrow y \in F$$

alors $E = F$.

Afin de montrer qu'une propriété $P(x)$ est vérifiée pour tout élément x d'un ensemble E muni d'un ordre bien fondé $\leqslant$, on montrera donc : que, pour tout élément minimal $m \in E$, la propriété $P(m)$ est vérifiée ; et que, pour tout autre élément $y \in E$, si $\forall x \in E, x < y \Rightarrow P(x)$, alors $P(y)$ est vérifiée.

En pratique, on montrera la terminaison d'un algorithme $\mathscr{A}$ par induction sur la valeur d'un paramètre (dans le cas d'appels récursifs) ou d'une variable (dans le cas de boucles `while`) $x \in X$. Pour ce faire on procèdera ainsi :

1. Identifier le paramètre x qui conditionne la terminaison de l'algorithme.
2. Identifier l'ensemble X des valeurs de x qu'il s'agit de traiter.
3. Munir cet ensemble d'un ordre bien fondé.
4. Montrer que, pour tout $x \in X$ minimal, l'exécution $\mathscr{A}(x)$ termine.
5. Montrer que, pour tout $x \in X$ autre, l'exécution $\mathscr{A}(x)$ se réduit à un nombre fini d'exécutions $\mathscr{A}(y)$ avec $y < x$.
6. Conclure que $\mathscr{A}(x)$ termine pour tout $x \in X$.

Exemple. *Soit l'algorithme ci-après.*

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

Montrons que `factorielle(n)` termine pour tout $n \in \mathbb{N}$. On procède par induction sur $n \in \mathbb{N}$ muni de son ordre usuel qui est évidemment bien fondé. Lorsque n est minimal, c'est-à-dire pour $n = 0$, l'algorithme termine trivialement. Sinon, `factorielle(n)` appelle récursivement `factorielle(k)` pour $k = n - 1 < n$. Ainsi, par le principe d'induction, `factorielle(n)` termine pour tout $n \in \mathbb{N}$.

Noter l'importance de spécifier l'ensemble des valeurs considérées : l'algorithme ci-dessus ne termine pas pour $n = -1$ quand bien même l'ensemble $\mathbb{N} \cup \{-1\}$ est lui aussi bien fondé.

Exercice. *Déterminer si les fonctions ci-dessous terminent et ce qu'elles renvoient le cas échéant.*

```
def a(n):
    if n == 0:
        return 0
    return a(n - 1)
```

```
def b(n):
    if n == 0:
        return 1
    return b(b(n - 1))
```

```
def c(n):
    if n == 0:
        return 0
    return c(c(n - 1))
```

Exercice. *Montrer la terminaison des algorithmes suivants.*

```
def pgcd(a, b):
    if b == 0:
        return a
    return pgcd(b, a % b)

def log2(n):
    k = 0
    while n > 1:
        n = n // 2
        k = k + 1
    return k

def ackermann(m,n):
    if m == 0:
        return n + 1
    if n == 0:
        return ackermann(m - 1, 1)
    return ackermann(m - 1, ackermann(m, n - 1))

def deuxtrois(n):
    if n % 2 == 1:
        return n
    return deuxtrois(3 * n // 2)
```

Remarque. *On peut montrer par argument diagonal qu'il n'existe aucun algorithme permettant de déterminer si un programme donné termine ou non ; on dit que le problème de l'arrêt est indécidable. Il reste même ouvert dans certains cas : on ne sait prouver ni la terminaison du programme ci-après ni son contraire.*

```
def syracuse(n):
    if n == 1:
        return 0
    elif n % 2 == 0:
        return 1 + syracuse(n // 2)
    else:
        return 1 + syracuse(3 * n + 1)
```

2.2.3 Correction

Soit un algorithme $\mathscr{A}$ dépendant d'un paramètre arbitraire $x \in X$ dont il s'agit de montrer que la valeur renvoyée $\mathscr{A}(x)$ est telle que calculée par une fonction mathématique f. Autrement dit, on souhaite établir que la suite des instructions exécutées par $\mathscr{A}$ calcule f. Pour cela, on démontrera la proposition $\mathscr{A}(x) = f(x)$ pour tout $x \in X$ en procédant par induction comme nous l'avons fait pour la terminaison.

Exemple. *On considère à nouveau l'algorithme ci-dessous dont la terminaison a déjà été établie.*

```
def pgcd(a, b):
    if b == 0:
        return a
    return pgcd(b, a % b)
```

Montrons que `pgcd(a,b)` *est correct pour tout* $(a, b) \in \mathbb{N}^2$*, c'est-à-dire qu'il renvoie* $\text{pgcd}(a, b)$*. On procède par induction sur* $b \in \mathbb{N}$ *muni de son ordre usuel qui est bien fondé. Lorsque* b *est minimal, c'est-à-dire pour* $b = 0$*, l'algorithme est correct. Sinon,* `pgcd(a,b)` *renvoie* `pgcd(b,a%b)`*; puisque* $a \bmod b < b$*, par hypothèse d'induction, cela renvoie* $\text{pgcd}(b, a \bmod b) = \text{pgcd}(a, b)$*. Ainsi, par le principe d'induction,* `pgcd(a,b)` *est correct pour tout* $(a, b) \in \mathbb{N}^2$*.*

Exercice. *Montrer la correction de l'algorithme* `factorielle` *récursif ci-dessus.*

En présence d'instructions itératives, l'induction est rarement aussi évidente car elle porte sur la valeur des variables intervenant dans ces boucles, que nous noterons ici k. Il s'agit alors d'exhiber une proposition $P(k)$ vraie avant la boucle et après chaque itération, ce qui sera alors montré par induction. Une telle proposition s'appelle un invariant de boucle et doit être choisi de sorte à pouvoir en déduire la correction de l'algorithme à la sortie de la boucle.

Exemple. *Soit à monter la correction de l'algorithme ci-dessous.*

```
def maximum(L):
    m = L[0]
    for k in range(1, len(L)):
        if m < L[k]:
            m = L[k]
    return m
```

On considère comme invariant de boucle la proposition « $m = \max_{i \in \{0,\dots,k\}} v_i$ *». Cette assertion est clairement vraie avant la boucle lorsque* $k = 0$ *puisqu'on a alors* $m = v_0$*. Supposant qu'elle est vérifiée après* k *itérations, la valeur de* k *est incrémentée à l'itération suivante et on compare alors* m *à* v_{k+1} *:*

— *Si* $m > v_{k+1}$ *alors* $m = \max_{i \in \{0,\dots,k\}} v_i = \max_{i \in \{0,\dots,k+1\}} v_i$*.*

— *Si* $m \leqslant v_{k+1}$ *alors* $m = v_{k+1} = \max_{i \in \{0,\dots,k+1\}} v_i$*.*

Ceci démontre l'induction par disjonction des cas. En sortie de boucle on a donc bien $m = \max L$ *ce qui démontre la correction de l'algorithme.*

Exercice. *Montrer la correction de l'algorithme* `factorielle` *itératif.*

Montrer la correction de l'algorithme `log2` *de la section précédente.*

2.2.4 Complexité

Les processeurs n'offrent qu'un petit nombre d'instructions et c'est en les combinant méthodiquement que les programmes parviennent à réaliser des calculs complexes. En première approximation, on peut supposer que chaque instruction s'exécute en temps constant ; le nombre de telles instructions utilisées par un programme est ainsi proportionnel à son temps d'exécution. L'objectif de cette section est d'évaluer cette quantité.

Remarque. *La réalité est certes plus complexe ; notamment, les appels mémoire ont un coût. Des modèles moins naïfs existent comme les machines de Turing [4] ou de von Neumann [5] mais les complexités sont pour la plupart asymptotiquement indépendantes de l'architecture sous-jacente.*

Considérons donc le nombre d'instructions utilisées par un algorithmes donné ; cette quantité est une fonction des valeurs des paramètres de l'algorithme et n'est pertinente qu'à un facteur de proportionnalité inconnu près. Nous n'étudierons ainsi que son comportement asymptotique modulo ce facteur. Rappelons à cet usage une notation de Landau.

Définition. *Soient* f *et* g *deux fonctions d'un ensemble quelconque* X *dans* $\mathbb{R}$*. On dit que* f *est dominée par* g *et on note* $f = O(g)$ *lorsque la quantité* f / g *est bornée.*

Munis de cette notation on cherche un majorant de la complexité le plus exact possible.

Définition. *Soit $\mathscr{A}$ un algorithme dépendant d'un paramètre quelconque $x \in X$ et soit $f(x)$ le nombre d'instructions utilisées par l'exécution de $\mathscr{A}(x)$. On appelle borne de complexité de $\mathscr{A}$ toute fonction $g : X \to \mathbb{R}$ vérifiant $f = O(g)$. Une telle borne est dite fine s'il existe une partie infinie $Y \subset X$ pour laquelle $g|_Y = O(f|_Y)$.*

En l'absence d'appels récursifs, une borne de complexité fine s'obtiendra simplement par comptage du nombre d'instructions en prenant bien soin de dérouler les boucles.

Exemple. *Les complexités des programmes ci-dessous sont respectivement $O(1)$ et $O(b)$.*

```
def multiplication(a, b):
    return a * b

def multiplication(a, b):
    x = 0
    for k in range(1, b):
        x = x + a
    return x
```

Lorsqu'à l'instar de l'exemple ci-dessus le nombre d'instructions par itération est constant, on peut se contenter de le multiplier par le nombre d'itérations (c'est-à-dire la longueur de la boucle) afin d'obtenir le nombre total d'instructions de la boucle. Quand ce n'est pas le cas on est réduit à sommer le nombre d'instructions pour chaque itération.

Exemple. *Considérons le programme :*

```
def sommecarre(n):
    x = 0
    for k in range(1, n):
        x = x + multiplication(k, k)
    return x
```

La complexité du programme `multiplication(k,k)` implanté plus haut étant $O(k)$, c'est le nombre d'instructions par itération. Le total est donc de $\sum_{k=1}^{n} O(k)$. Le facteur multiplicatif implicite du « grand O » étant le même pour chaque terme, on peut l'intervertir avec la somme. On obtient alors comme borne de complexité $O(\sum_{k=1}^{n} k) = O(\frac{n(n+1)}{2}) = O(n^2)$.

Remarquer qu'on aurait obtenu la même borne si on avait grossièrement majoré par $O(n)$ le nombre d'instructions par itération, puis multiplié par la longueur de la boucle. Cette approche plus simple peut néanmoins parfois donner des bornes non fines.

Exercice. *Afin de déterminer si un objet appartient à une liste donnée, on peut procéder par la méthode naïve consistant à parcourir la liste ou, lorsque celle-ci est triée, par la méthode de la dichotomie. Déterminer les complexité de ces deux approches.*

```
def appartient(objet, liste):
    for i in range(len(liste)):
        if objet == liste[i]:
            return True
    return False
```

```
def appartient(objet, liste):
    debut = 0
    fin = len(liste)
    while fin - debut > 1:
        milieu = (debut + fin) // 2
        if objet < liste[milieu]:
            fin = milieu
        else:
            debut = milieu
    return objet == liste[milieu]
```

En présence d'appels récursifs, on procèdera en deux temps. D'abord, on écrira une relation de récurrence exprimant la complexité pour un paramètre donné en fonction de celles pour de plus petits paramètres. (Cela suppose implicitement la terminaison.) Ensuite, on résoudra cette relation de récurrence afin d'obtenir le terme général de la complexité. Si cela s'avère difficile on pourra remplacer la relation par un majorant simple, au risque d'obtenir une borne non fine.

Exemple. *Notons c_b la complexité pour $b \in \mathbb{N}$ du programme :*

```
def multiplication(a, b):
    if b == 0:
        return 0
    return a + multiplication(a, b - 1)
```

Elle vérifie $c_b = 3 + c_{b-1}$ et $c_0 = 1$ d'où l'on déduit facilement $c_b = 3b + 1 = O(b)$.

Plus généralement, rappelons qu'une suite $(s_n)_{n\in\mathbb{N}}$ est dite arithmético-géométrique lorsqu'elle satisfait une relation de récurrence du type $s_{n+1} = as_n + b$; son terme général s'écrit alors :

$$s_n = a^n \left(s_0 - \frac{b}{1-a} \right) + \frac{b}{1-a}$$

Considérons à présent les programmes suivants.

```
def fibo1(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibo1(n - 1) + fibo1(n - 2)

def fibo2(n):
    a, b = (0, 1)
    for k in range(n):
        a, b = (b, a + b)
    return a

def mul(x, y):
    a, b, c, d = x
    e, f, g, h = y
    return (a * e + b * g, a * f + b * h, c * e + d * g, c * f + d * h)
```

	`fibo1`		`fibo2`		`fibo3`	
	COMPLEXITÉ	TEMPS	COMPLEXITÉ	TEMPS	COMPLEXITÉ	TEMPS
n	1.618^n		n		$\log(n)$	
10	10^2	0.00003 s				
20	10^4	0.003 s				
30	10^6	0.3 s				
40	10^8	30 s				
50	10^{10}	50 min				
10^2	10^{21}	10^7 ans	10^2	0.00002 s		
10^3			10^3	0.0002 s		
10^4			10^4	0.002 s		
10^5			10^5	0.02 s	5	0.00005 s
10^{10}			10^{10}	30 min	10	0.0001 s
10^{20}			10^{20}	10^6 ans	20	0.0002 s
10^{10^2}					10^2	0.001 s
10^{10^3}					10^3	0.01 s
10^{10^4}					10^4	0.1 s

FIGURE 2.6 – Complexité et estimation du temps d'exécution de différents algorithmes calculant le n^e terme de la suite de Fibonacci modulo un entier fixé.

```
def pow(x, n):
    r = (1, 0, 0, 1)
    while n > 0:
        if n % 2 == 1:
            r = mul(r, x)
        x = mul(x, x)
        n = n // 2
    return r

def fibo3(n):
    return pow((0, 1, 1, 1), n)[1]
```

Supposant que les opérations arithmétiques s'effectuent en temps constant, ce qui serait notamment le cas si les calculs étaient réalisés modulo un entier fixé, les complexités des programmes `fibo1`, `fibo2` et `fibo3` sont respectivement $O(\varphi^n)$, $O(n)$ et $O(\log(n))$, avec $\varphi = \frac{1+\sqrt{5}}{2}$. Par une simple règle de proportionnalité, on peut alors estimer le temps de calcul nécessaire à l'exécution de ces programmes pour de grands paramètres ; voir la figure 2.6.

Remarquons enfin que le temps n'est pas la seule ressource dont on ne dispose qu'en quantité limitée. À tout moment, un ordinateur ne peuvent en effet stocker qu'une quantité finie d'information. Il pourra donc être opportun dans certaines situations de considérer la notion de complexité en mémoire parallèlement à celle de complexité en temps.

Exercice. *Écrire des fonctions calculant le minimum, la moyenne et l'écart type d'une liste de réels. Démontrer leur correction et déterminer leur complexité.*

Exercice. *Démontrer la correction et déterminer la complexité des algorithmes de tris vus au premier semestre : le tri par sélection, le tri par insertion, le tri à bulles, le tri rapide et le tri fusion.*

On trouve que la complexité de trier une liste de longueur n est de $O(n^2)$ pour tous les tris mentionnés ci-dessus à l'exception du tri fusion pour lequel elle est de $O(n \log n)$. Cette dernière complexité est en fait optimale.

Théorème. *Soit un algorithme permettant de trier une liste donnée suivant un ordre arbitraire. Sa complexité est minorée par $n \log_2(n)$ où n dénote la longueur de la liste.*

Démonstration. Une liste constituée de n éléments distincts admet $n!$ permutations et l'algorithme doit ainsi identifier l'unique $\sigma \in \mathfrak{S}_n$ triant la liste donnée. Chaque comparaison lui permet d'apprendre que σ appartient soit à une partie $A \subset \mathfrak{S}_n$ soit à son complémentaire. Dans le pire des cas, chaque comparaison ne permet au mieux d'éliminer que la moitié des possibilités restantes puisque $\max\left\{|A|, |\overline{A}|\right\} \geqslant \frac{n!}{2}$. Le nombre de comparaisons $f(n)$ vérifie alors $2^{f(n)} \geqslant n!$ d'où découle le résultat attendu avec la formule de Stirling. □

Exercice. *Écrire une fonction calculant l'intersection de deux listes données. En déterminer la complexité. Écrire une fonction plus rapide dans le cas où les deux listes sont supposées triées. Qu'en déduire dans le cas général ?*

2.3 Structures de données

Le reste du semestre est consacré à l'application des outils théoriques que nous venons d'introduire à la construction et à l'analyse de structures de données qui s'avèreront être progressivement de plus en plus avancées.

Définition. *Une **structure de données** est un procédé permettant de stocker de l'information de manière organisée afin de la traiter efficacement. Elle fourni des opérations de trois types : **des constructeurs** qui initialisent un nouvel objet ; **des accesseurs** qui en extraient des informations ; **des transformateurs** qui en modifient les informations. Une structure de données n'admettant aucun transformateur est dite **immuable** ; dans le cas contraire, elle est dite **mutable**.*

Les listes vues précédemment sont immuables. On peut certes écrire :

```
let ajout = function liste -> 0 :: liste ;;
let liste = ajout liste ;;
```

Mais la fonction `ajout` se contente de créer puis de renvoyer une nouvelle liste comportant un terme supplémentaire qui est alors substituée à l'ancienne dans la variable `liste` ; l'ancienne liste n'est en réalité jamais modifiée, elle est seulement oubliée lorsqu'aucune variable ne permet d'y accéder. Inversement, les tableaux sont quant à eux mutables :

```
let t = Array.make 2 0 ;;
t.(1) <- 1 ;;
```

L'expression de la seconde ligne est un transformateur : elle est de type `unit` et a pour effet de bord de modifier le contenu du tableau `t`.

La programmation fonctionnelle et, en particulier, le langage OCaml privilégient les types immuables. Les types mutables y sont l'exception plutôt que la règle ; ils incluent notamment :

- les références ;
- les tableaux ;
- les champs des types enregistrements déclarés avec le préfixe `mutable` comme nous allons le voir ci-après.

2.3.1 Piles et files

Abordons à présent les structures de données classiques en commençant par celles techniquement proches des listes et des tableaux que nous fournit le langage.

Définition. *Une **pile** (resp. **file**) est une structure de donnée stockant une collection d'objets et offrant deux opérations, ajouter et enlever, de sorte que l'objet ajouté en dernier (resp. premier) soit le premier enlevé.*

Les systèmes d'exploitation exploitent intensément ces deux structures : les piles sont notamment utilisée pour gérer les appels de fonctions et la récursivité ; les files sont quant à elles utilisées comme mémoire tampon pour les communications entre processus, avec les périphériques et en particulier les réseaux.

Piles comme listes. Pour implanter un type réalisant la structure de pile en OCaml, on s'appuie naturellement sur la structure de liste en se contentant d'habiller légèrement le type correspondant.

```
type 'a pile = { mutable contenu : 'a list } ;;
let pile_vide () = { contenu = [] } ;;
let empiler p x = p.contenu <- x :: p.contenu ;;

let depiler p =
    match p.contenu with
    | []     -> failwith "pile vide"
    | h :: t -> p.contenu <- t; h
;;
```

Les opérations d'empilement et de dépilement se réalisent ainsi en temps constant. Une implantation de cette structure de donnée est déjà disponible en langage OCaml via le module `Stack` qui fournit notamment les fonctions :

```
Stack.create   : unit -> 'a Stack.t        (* constructeur  *)
Stack.push     : 'a -> 'a Stack.t -> unit  (* empilement    *)
Stack.pop      : 'a Stack.t -> 'a          (* dépilement    *)
Stack.is_empty : 'a Stack.t -> bool        (* teste si vide *)
```

Files comme couples de listes. Afin de réaliser des files, une première approche consiste à les représenter comme un couple de listes, la première étant utilisée pour y ajouter des objets et la seconde pour en enlever ; lorsque la seconde est vide, on la remplace par la première renversée.

```
type 'a file = {
    mutable uno : 'a list;
    mutable dos : 'a list;
} ;;

let file_vide () = { uno = []; dos = [] } ;;
let enfiler f x = f.uno <- x :: f.uno ;;

let defiler f =
    if f.dos = [] then begin
        f.dos <- List.rev f.uno;
        f.uno <- []
    end;
    match f.dos with
    | []     -> failwith "file vide"
    | h :: t -> f.dos <- t; h
;;
```

Les appels à `enfiler` s'effectuent clairement en temps constant mais ce n'est pas le cas des appels à `defiler`. On peut toutefois montrer que, partant d'une file vide, la complexité totale de n appels consécutifs à `enfiler` et `defiler` est $O(n)$; ainsi, la complexité moyenne par appel (dit « coût amorti ») est $O(1)$.

Files comme tranches de tableaux. Une seconde approche consiste à représenter une file par une tranche de tableau, c'est-à-dire un triplet (t, s, n) où t est un tableau et s et n deux entiers ; les objets de la file sont les $t_s, t_{s+1}, \ldots, t_{s+n-1}$ où les opérations sur les indices sont réalisées modulo la longueur du tableau.

```
type 'a file = {
    mutable t : 'a array;
    mutable s : int;
    mutable n : int;
} ;;

let vide () = { t = [| 0 |]; s = 0; n = 0 } ;;

let defiler f =
    if f.n = 0 then failwith "file vide";
    let x = f.t.(f.s) in
    f.n <- f.n - 1;
    f.s <- (f.s + 1) mod Array.length f.t;
    x
;;

let enfiler f x =
    if f.n >= Array.length f.t then f.t <- Array.concat [ f.t; f.t ];
    f.t.((f.s + f.n) mod Array.length f.t) <- x;
    f.n <- f.n + 1
;;
```

Pour cette implantation, ce sont les appels à `defiler` qui s'effectuent en temps constant, non ceux à `enfiler`. Comme précédemment, on peut néanmoins montrer que la complexité totale partant d'une file vide est linéaire en le nombre d'appels à ces fonctions. Le coût amorti de chaque appel est donc $O(1)$.

Remarque. *Les deux implantations ci-dessus réalisent la même structure de données et présentent la même interface, c'est-à-dire qu'elles fournissent des fonctions de signature identique. Elles peuvent donc être utilisées de manière interchangeable en faisant abstraction de leurs rouages internes.*

Une implantation de cette structure de donnée est déjà disponible en langage OCaml via le module `Queue` qui fournit notamment les fonctions :

```
Queue.create   : unit -> 'a Queue.t        (* constructeur  *)
Queue.push     : 'a -> 'a Queue.t -> unit  (* enfilement    *)
Queue.pop      : 'a Queue.t -> 'a          (* défilement    *)
Queue.is_empty : 'a Queue.t -> bool        (* teste si vide *)
```

2.3.2 Dictionnaires

Définition. *Un dictionnaire est une structure de données stockant des couples* (*clef*, *valeur*) *où les clefs sont distinctes. Les opérations principales sont :*

— ***constructeur :*** *créer un nouveau dictionnaire ;*

— ***transformateur :*** *ajouter ou enlever un couple (si mutable) ;*
— ***accesseur :*** *déterminer si une clef donnée apparaît dans le dictionnaire ;*
— ***accesseur :*** *lorsque c'est le cas, déterminer la valeur correspondante.*

Exercice. *Implanter une structure de dictionnaire immuable en stockant les clefs dans un tableau trié et effectuant les recherches par dichotomie. Déterminer les complexités des opérations.*

```
type ('a, 'b) dictionnaire = {
    clefs   : 'a array;
    valeurs : 'b array;
} ;;

let dichotomie t x =
    let n = Array.length t in
    let a, b = (ref 0, ref n) in
    while !b - !a > 1 do
        let c = (!a + !b) / 2 in
        if x < t.(c) then b := c else a := c
    done;
    if t.(!a) = x then Some !a else None
;;

let apparait dict clef =
    match dichotomie dict.clefs clef with
    | None   -> false
    | Some _ -> true
;;

let valeur dict clef =
    match dichotomie dict.clefs clef with
    | None   -> failwith "mauvaise clef"
    | Some i -> dict.valeurs.(i)
;;

let creation liste_couples =
    let ordre (c, v) (c', v') = compare c c' in
    let triee = List.sort ordre liste_couples in
    {
        clefs   = Array.of_list (List.map fst triee);
        valeurs = Array.of_list (List.map snd triee);
    }
;;
```

La complexité du constructeur est dominée par l'appel à la fonction de tri et vaut donc $O(n \log n)$ où n désigne le nombre de couples stockés. La complexité des autres opérations est dominée par l'appel à la fonction dichotomie et vaut ainsi $O(\log n)$.

Rappelons que la version mutable de cette structure de données est disponible en langage Python via une syntaxe qui généralise celle des ensembles.

```
D = {"abc": 3, "de": 2}
D["abc"]    # 3
D["def"]    # KeyError: "def"
D["f"] = 1
for x in D:
    print(x, D[x])
```

Elle est également disponible en langage OCaml grâce aux fonctions :

```
Hashtbl.create   : int -> ('a, 'b) Hashtbl.t
Hashtbl.add      : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit
Hashtbl.remove   : ('a, 'b) Hashtbl.t -> 'a -> unit
Hashtbl.mem      : ('a, 'b) Hashtbl.t -> 'a -> bool
Hashtbl.find     : ('a, 'b) Hashtbl.t -> 'a -> 'b
Hashtbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b option
Hashtbl.iter     : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit
```

Nous verrons au semestre prochain comment en réaliser une implantation efficace.

Exercice. *Écrire une fonction de signature « `liste_clefs : ('a, 'b) Hashtbl.t -> 'a list` » renvoyant la liste des clefs de la table de hachage donnée.*

```
let liste_clefs h =
    let l = ref [] in
    let f clef valeur = l := clef :: !l in
    Hashtbl.iter f h;
    !l
;;
```

2.4 Types récursifs immuables et arbres

Les structures de données récemment construites à l'aide de types enregistrements sont mutables et directement apparentées aux listes et aux tableaux offerts par le langage. Nous allons maintenant créer des structures de toutes pièces ; elles seront immuables et définies récursivement via des types énumérés.

2.4.1 Listes

Contrairement aux définitions de fonctions, les définitions de types sont implicitement supposées récursives et proscrivent l'usage du mot clef `rec`. On peut donc recréer les listes ainsi :

```
type 'a liste = Nil | Cons of 'a * 'a liste ;;
```

Ce nouveau type s'utiliserait alors comme ceci :

```
let liste = Cons (1, Cons (2, Cons (3, Nil))) ;;
let rec longueur = function Nil -> 0 | Cons (h,t) -> 1+longueur t ;;
longueur liste ;;
```

Exercice. *Établir la correction et la complexité de la fonction `longueur`.*

Exercice. *Implanter pour ce nouveau type de listes, démontrer la correction puis enfin déterminer la complexité des fonctions `maximum` et `second_maximum`. Faire de même pour une méthode de tri de votre choix.*

La définition ci-dessus décrit la structure des listes par dichotomie : soit vide, soit tête et queue. Tout procédé d'induction portant sur l'ensemble des listes et exploitant l'ordre bien fondé induit par cette structure est appelé induction structurelle.

Remarque. *On peut aussi définir des types (ou des fonctions) mutuellement récursifs grâce au mot clef `and` comme il suit ; cette construction reste néanmoins anecdotique car généralement peu utile hormis pour des pirouettes telles :*

```
type phrase = Simple of sujet * verbe | Complexe of phrase * subordonnee
and subordonnee = ParceQue of phrase | PendantQue of phrase ;;
```

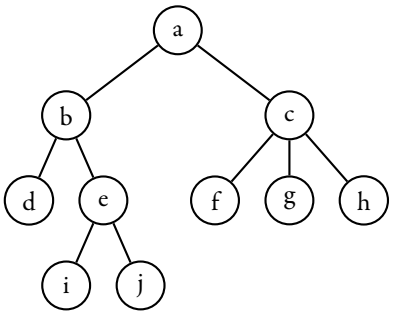


FIGURE 2.7 – Un arbre de taille dix et hauteur trois comportant six feuilles.

```
let rec a n = if n = 0 then 1. else (a (n - 1) +. b (n - 1)) /. 2.
    and b n = if n = 0 then 2. else sqrt (a (n - 1) *. b (n - 1)) ;;
```

2.4.2 Arbres

Les arbres sont des structures de données dont l'organisation rappelle la forme des plantes éponymes. Leur théorie combinatoire est vaste et leurs applications algorithmiques innombrables. Comme toute structure de données de ce cours, ils sont implicitement supposés finis.

Définition. *On appelle arbre toute structure de données présentant les caractéristiques suivantes :*
- *Elle est formée d'objets, appelés nœuds.*
- *Chaque nœud admet une famille (possiblement vide) de nœuds, appelés fils.*
- *Un nœud, appelé racine, n'est le fils d'aucun nœud.*
- *Tout nœud autre que la racine est le fils d'un unique nœud, appelé père.*
- *Tous les nœuds sont des descendants (c'est-à-dire fils de fils de fils, etc.) de la racine.*

La structure d'arbre généralise celle de liste, cas particulier où chaque nœud admet au plus un fils. L'élément minimal ne comportant aucun nœud et généralisant la liste vide porte ici le nom d'arbre vide.

Exemple. *Les notions suivantes présentent généralement une structure d'arbre :*
- *un arbre généalogique (restreint aux pères) ;*
- *l'organigramme d'une entreprise privée ;*
- *l'arborescence de répertoires d'un système de fichiers ;*
- *les classifications classique et phylogénétique des êtres vivants ;*
- *les décisions possibles qu'un acteur peut successivement prendre ;*
- *les opérations, les constantes et les variables d'une expression algébrique ;*

Les arbres sont représentés comme l'illustre la figure 2.7 : à l'envers, c'est-à-dire avec la racine en haut. Comme ces représentations le présupposent, la définition est souvent tacitement amendée dans le sens où la famille des fils de chaque nœud est implicitement munie d'un ordre total, comme c'est par exemple le cas des membres d'une fratrie : aîné, cadet, benjamin, etc.

Définition. *Soit $\mathscr{A}$ un arbre.*
On appelle feuille tout nœud n'admettant aucun fils.
On appelle profondeur d'un nœud la longueur du chemin le reliant à la racine.
On appelle hauteur de $\mathscr{A}$ notée $h(\mathscr{A})$ la plus grande profondeur de ses nœuds.
On appelle taille de $\mathscr{A}$ notée $|\mathscr{A}|$ le nombre de nœuds qu'il comporte.

L'arbre de la figure 2.7 est ainsi de taille dix, de hauteur trois et comporte six feuilles. Certains résultats à venir pourront être démontrés par récurrence sur $h(\mathscr{A})$ ou $|\mathscr{A}|$ plutôt que par induction structurelle sur l'ensemble des arbres.

En OCaml on peut définir un type d'arbre abstrait par la définition récursive :

```
type arbre = Noeud of arbre list ;;
```

L'arbre composé d'une racine admettant deux fils s'écrirait alors :

```
Noeud [ Noeud []; Noeud [] ]
```

Et on procéderait ainsi pour calculer la taille et la hauteur :

```
let rec taille = function
    | Noeud [] -> 1
    | Noeud (h :: t) -> taille h + taille (Noeud t)
;;

let rec hauteur = function
  | Noeud [] -> 0
  | Noeud [ h ] -> 1 + hauteur h
  | Noeud (h :: t) -> max (Noeud [ h ]) (Noeud t)
;;
```

Exercice. *Écrire une fonction renvoyant le nombre de feuilles dans un arbre donné.*
Quelles sont les complexités de cette fonction et de celles implantées ci-dessus ?

Définition. *Soit x un nœud d'un arbre A. On appelle arbre enraciné en x l'arbre formé de x et de ses descendants dans A. On dit qu'un arbre B est un sous arbre de A si c'est un arbre enraciné en un nœud quelconque de A.*

Exercice. *Montrer qu'on définit ainsi un ordre bien fondé sur l'ensemble des arbres.*

Cet ordre bien fondé est celui sous-tendant les preuves par induction structurelle.

Exercice. *Montrer que tout nœud d'un arbre $\mathscr{A}$ de profondeur $h(\mathscr{A})$ est une feuille.*
Montrer que tout arbre $\mathscr{A}$ satisfait $|\mathscr{A}| = h(\mathscr{A}) + 1$ si et seulement si c'est une liste.

2.4.3 Arbres binaires

Nos efforts seront dorénavant focalisés sur le cas particulier que voici.

Définition. *Un arbre est dit binaire lorsque chacun de ses nœuds comporte exactement deux fils, appelés respectivement le fis gauche et le fils droit, chacun étant possiblement vide.*

On peut définir un tel type en OCaml plus commodément sans faire appel aux listes :

```
type arbreB =
    | Vide
    | Noeud of arbreB * arbreB
;;
```

Exercice. *Écrire des fonctions calculant la taille, la hauteur puis le nombre de feuilles d'un objet de type* `arbreB`*. Écrire une fonction déterminant si deux objets de ce type sont égaux.*

La combinatoire des arbres est un thème riche en résultats ; on se contentera dans ce cours de quelques propriétés élémentaires reliant les invariants introduits ci-dessus.

Théorème. *Soit un arbre binaire. Son nombre de feuilles f et sa taille t vérifient $t \geqslant 2f - 1$ avec égalité si et seulement si aucun nœud n'a exactement un fils.*
Sa hauteur h satisfait de surcroit $h < t < 2^{h+1}$.

Démonstration. Prouvons l'inégalité $t \leqslant 2f - 1$ par induction : elle est clairement vérifiée dans le cas minimal de l'arbre réduit à sa racine. Pour un arbre binaire A de taille deux ou plus, la supposant vérifiée pour son fils gauche B et son fils droit C, on vérifie $t_A = t_B + t_C + 1 \leqslant (2f_B - 1) + (2f_C - 1) + 1 = 2(f_B + f_C) - 1 = 2f_A - 1$, d'où le résultat. □

Exercice. *Démontrer pareillement les autres inégalités du théorème.*

Lorsqu'il est possible d'annoter les nœuds, on parlera d'arbres étiquetés ; en informatique cela permet notamment de stocker de l'information dans un arbre et en mathématiques de mettre les nœuds en correspondance avec d'autres objets. On définit ainsi :

```
type 'a arbreBE =
    | Vide
    | Noeud of 'a * 'a arbreBE * 'a arbreBE
;;
```

Exercice. *La numérotation de Sosa–Stradonitz étiquète la racine d'un arbre binaire par l'entier un, puis les fils du nœud étiqueté n par $2n$ pour le fils gauche et $2n + 1$ pour le fils droit. Définir une fonction « `arbreB -> int arbreBE` » calculant cette numérotation. Montrer que cette numérotation donne des étiquettes distinctes aux nœuds d'un même arbre.*

Exercice. *Écrire une fonction qui, étant donné un arbre de type* `arbreBE`*, renvoie la liste des étiquettes de ses nœuds.*

Pour énumérer les étiquettes d'un arbre, autrement dit, pour parcourir ses nœuds, une première approche consiste à explorer les fils d'un nœud avant ses frères ; on obtient ainsi les **parcours en profondeur** qui se déclinent en trois variantes :

```
let rec prefixe = function
    | Vide            -> []
    | Noeud (e, g, d) -> [ e ] @ prefixe g @ prefixe d
;;

let rec infixe = function
    | Vide            -> []
    | Noeud (e, g, d) -> infixe g @ [ e ] @ infixe d
;;

let rec postfixe = function
    | Vide            -> []
    | Noeud (e, g, d) -> postfixe g @ postfixe d @ [ e ]
;;
```

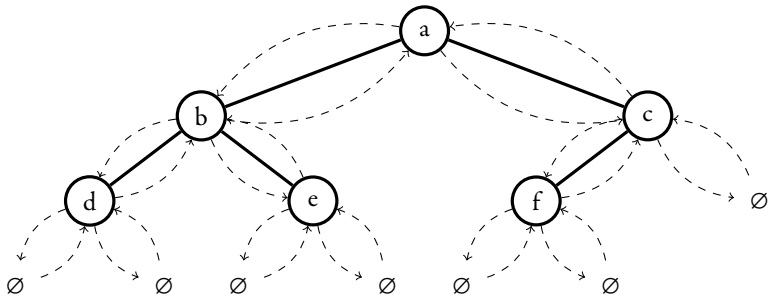


FIGURE 2.8 – Un parcours en profondeur offre trois opportunités de visiter chaque nœud : avant, entre et après ses deux sous arbres ; elles donnent respectivement les parcours préfixe « abdecf », infixe « dbeafc » et postfixe « debfca ».

Comme l'illustre la figure 2.8, un parcours en profondeur visite trois fois chaque nœud de l'arbre ; les variantes préfixe, infixe et postfixe consistent à extraire l'étiquette d'un nœud respectivement lors de sa première, seconde et troisième visite.

Une seconde approche consiste à explorer les fils d'un nœud après ses frères ; on obtient ainsi les **parcours en largeur**. Ci-dessous, la fonction auxiliaire prend comme argument une liste contenant les sous arbres à explorer.

```
let largeur a =
    let rec aux = function
        | []                    -> []
        | Vide :: t             -> aux t
        | Noeud (e, g, d) :: t -> e :: aux (t @ [ g; d ])
    in
    aux [ a ]
;;
```

Une implantation plus efficace stockerait ces sous arbres dans une file.

```
let largeur a =
    let q = Queue.create () in
    let r = ref [] in
    Queue.push a q;
    while not (Queue.is_empty q) do
        match Queue.pop q with
        | Vide -> ()
        | Noeud (e, g, d) ->
            r := e :: !r;
            Queue.push g q;
            Queue.push d q
    done;
    List.rev !r
;;
```

Exercice. *Écrire une fonction de signature « `'a arbreBE -> int -> 'a list` » renvoyant la liste des étiquettes des nœuds de profondeur donnée dans l'arbre donné.*

2.4.4 Arbres de recherche

Pour stocker une collection de n objets de sorte à y effectuer des recherches, nous avons déjà exploité les listes et les tableaux, où la recherche s'effectue naïvement en $O(n)$, et les tableaux triés, où la recherche s'effectue par dichotomie en $O(\log n)$. Les arbres de recherche offrent un moyen de réaliser cette structure de donnée de manière efficace et potentiellement mutable.

Définition. *On appelle arbre de recherche tout arbre binaire dont l'étiquette de chaque nœud est supérieure à celles des nœuds de son sous arbre gauche et inférieure à celles des nœuds de son sous arbre droit.*

Proposition. *Un arbre binaire est un arbre de recherche si et seulement si un parcours infixe énumère ses étiquettes dans l'ordre croissant.*

Démonstration. On procède par induction structurelle sur les arbres binaires. Le cas de l'arbre vide (élément minimal) est trivial et l'induction est immédiate. □

En procédant de manière semblable à la dichotomie, on peut déterminer l'appartenance d'un élément à l'ensemble des étiquettes d'un arbre de recherche en $O(h)$ comparaisons.

```
let rec appartient x = function
    | Vide            -> false
    | Noeud (e, g, d) -> if x < e then appartient x g else
                         if e < x then appartient x d else
                         true
;;
```

Puisque la complexité dépend de la hauteur des arbres considérés, il est opportun d'essayer de la minimiser à taille constante. Pour obtenir un bon rapport entre la hauteur h d'un arbre et sa taille t, nous allons considérer une classe d'arbres pour lesquels l'inégalité $h < t < 2^{h+1}$ peut être significativement raffinée.

Définition. *Un arbre binaire est dit équilibré si, en chacun de ses nœuds, les sous arbres gauche et droite sont de même hauteur à une unité près.*

Théorème. *Un arbre équilibré vérifie $t \geqslant \left(\frac{3}{2}\right)^h$ soit, asymptotiquement, $h = O(\log(t))$.*

Démonstration. On procède par récurrence sur h. L'inégalité est évidente pour $h = 0$. Dans un arbre équilibré de hauteur $h > 0$, les sous arbres enracinés en le fils gauche et le fils droit de la racine sont eux-mêmes équilibrés et de hauteur $h-1$ ou $h-2$. On a ainsi :

$$t = 1 + t_g + t_d \geqslant 1 + \left(\frac{3}{2}\right)^{h-1} + \left(\frac{3}{2}\right)^{h-2} = 1 + \frac{10 \cdot 3^{h-2}}{2^h} \geqslant \left(\frac{3}{2}\right)^h.$$

En réalité, la taille minimale t_h d'un arbre équilibré de hauteur h satisfait $t_{h+2} = 1 + t_{h+1} + t_h$ d'où l'égalité $t_h = \lambda\phi^h + (2-\lambda)(1-\phi)^h - 1$ avec $\phi = \frac{1+\sqrt{5}}{2}$ et $\lambda = 1 + \frac{2}{\sqrt{5}}$. □

L'équilibrage d'un arbre garantit donc la quasi optimalité de sa hauteur. Concentrons nous maintenant sur l'implantation efficace des différentes notions introduites ci-dessus.

Exercice. *Écrire une fonction de signature « `'a arbreBE -> bool` » déterminant si un arbre binaire est équilibré. Quelle est sa complexité ?*

```
let equilibre x =
    let rec aux = function
        | Vide -> true, -1
        | Noeud (g, d) ->
            let eg, hg = aux g in
            let ed, hd = aux d in
            (eg && ed && abs (hg - hd) < 2, 1 + max hg hd)
    in
    fst (aux x)
;;
```

Exercice. *Écrire une fonction de signature « `'a arbreBE -> bool` » déterminant si un arbre binaire étiqueté est un arbre de recherche. Quelle est sa complexité ?*

```
let rec fold f = function
    | Noeud (e, Vide, Vide) -> e
    | Noeud (e, Vide, d) -> f e (fold f d)
    | Noeud (e, g, Vide) -> f e (fold f g)
    | Noeud (e, g, d) -> f e (f (fold f g) (fold f d))
;;

let recherche a =
    let rec aux x z = function
        | Vide -> true
        | Noeud (e, g, d) -> x < e && e < z && aux x e g && aux e z d
    in
    aux (fold min a) (fold max a) a
;;
```

On pourrait aussi se contenter de vérifier qu'un parcours infixe énumère les étiquettes dans l'ordre croissant ; cela nécessite de concaténer des listes qu'il serait donc opportun de remplacer par une structure de donnée davantage adaptée à cette opération.

```
let recherche a = croissant (infixe a) ;;
```

Construisons à présent des instances de cette nouvelle structure de données.

Exercice. *Écrire une fonction renvoyant un arbre de recherche équilibré dont les étiquettes sont exactement les éléments d'un tableau donné que l'on supposera trié.*

```
let array2arbreRE t =
    let rec aux a b =
        let c = (a + b) / 2 in
        if b <= a then Vide else
        Noeud (t.(c), aux a c, aux (c + 1) b)
    in
    aux 0 (Array.length t)
;;
```

Exercice. *Écrire une fonction rajoutant à un arbre de recherche un nœud d'étiquette donnée.*

Cette opération peut toutefois déséquilibrer l'arbre : en chacun des ancêtres du nœud inséré, le sous arbre gauche et celui de droite peuvent alors potentiellement différer d'au plus deux unités de hauteur. En travaux dirigés on pourra concevoir des types d'arbres de recherche rééquilibrant.

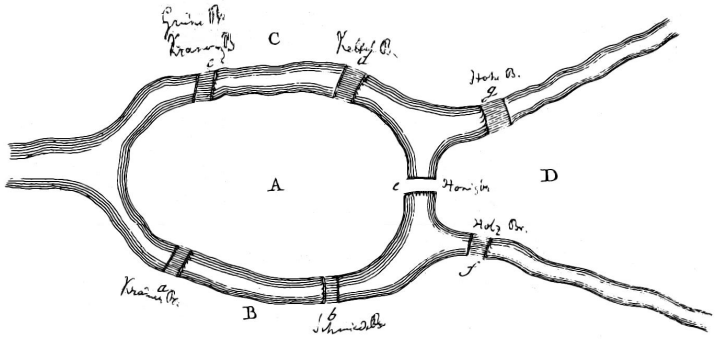


FIGURE 2.9 – Représentation synthétique des cours d'eau et des ponts du centre ville de Königsberg; extrait du manuscrit original d'Euler [1].

2.5 Bases des graphes, plus courts chemins

2.5.1 Aspects théoriques

La notion de graphe formalise les structures que l'on qualifie communément de réseaux ; qu'il s'agisse d'un réseau téléphonique ou routier, un graphe n'est du point de vue conceptuel qu'un ensemble d'arcs reliant des nœuds.

Définition. *Un graphe est un couple (S, A) d'ensembles vérifiant $A \subset S \times S$; les éléments de S sont appelés sommets et ceux de A arêtes. Le graphe est dit d'ordre $|S|$ et de taille $|A|$; il est dit simple s'il ne comporte aucune boucle, c'est-à-dire aucune arête de la forme (x, x) ; il est dit non orienté si A est symétrique, c'est-à-dire si $(x, y) \in A \Leftrightarrow (y, x) \in A$.*

Tous les graphes considérés dans ce cours seront implicitement supposés simples. Parmi les nombreuses applications de cette notion, mentionnons les quelques exemples que voici :

- circuit électrique : $S = \{\text{composants}\}$, $A = \{\text{fils}\}$
- toile sociale : $S = \{\text{individus}\}$, $A = \{\text{fréquentations}\}$
- réseau routier : $S = \{\text{échangeurs}\}$, $A = \{\text{routes}\}$
- mappemonde : $S = \{\text{pays}\}$, $A = \{\text{frontières}\}$

Exercice. *Identifier lesquels de ces exemples sont de nature non orientée.*

Abordons l'étude des graphes par un problème historique.

Problème (sept ponts de Königsberg). *Du temps d'Euler, au XVIIIe siècle, la ville de Königsberg au royaume de Prusse comportait sept ponts disposés comme l'indique la figure 2.9. Existait-il un chemin passant par chaque pont une fois et une seule ?*

Pour attaquer ce problème, la première étape est de n'en garder que les éléments pertinents ; ceux-ci s'expriment naturellement comme le graphe de la figure 2.10. Reste à présent à introduire des outils permettant une analyse rigoureuse des graphes.

Définition. *Un chemin est une famille de sommets $(\sigma_0, \sigma_1, \ldots, \sigma_n)$ vérifiant $(\sigma_k, \sigma_{k+1}) \in A$ pour tout $k \in \{0, \ldots, n-1\}$. On le note $\sigma_0 \to \sigma_1 \to \ldots \to \sigma_n$ et on dit qu'il est de longueur n. Il est qualifié de cycle lorsque $\sigma_0 = \sigma_n$, est dit élémentaire lorsque les sommets σ_k sont distincts pour $k \in \{0, \ldots, n-1\}$ et est dit simple lorsque les arêtes (σ_k, σ_{k+1}) sont distinctes pour $k \in \{0, \ldots, n-1\}$.*

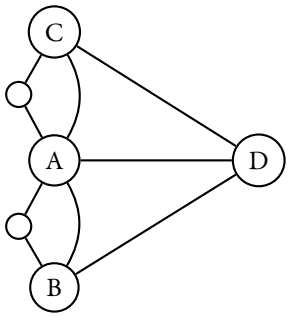


FIGURE 2.10 – Structure de graphe associée à la figure 2.9. Afin d'éviter les arêtes multiples, on rajoute un sommet au milieu des deux ponts de gauche.

Définition. *Soit (S, A) un graphe non orienté. Deux sommets sont dits adjacents lorsqu'ils sont reliés par une arête. Cela induit une relation binaire sur S appelée relation d'adjacence qui est symétrique mais généralement ni réflexive ni transitive.*

Cette relation s'étend par transitivité en $x \mathrel{\mathscr{R}} y \iff$ « il existe un chemin de x à y » qui est quant à elle une relation d'équivalence. Ses classes partitionnent S et sont appelées composantes connexes. Un graphe est dit connexe s'il admet une unique composante connexe.

Exercice. *Quelles définitions similaires pourriez vous proposer pour les graphes orientés ?*

La clef pour percer le problème d'Euler est la notion que voici.

Définition. *On appelle degrés sortant et entrant d'un sommet x les quantités*

$$\deg^+(x) = \#\big\{y \in S : (x, y) \in A\big\} \text{ et}$$
$$\deg^-(x) = \#\big\{y \in S : (y, x) \in A\big\}.$$

L'appellation degré total désigne alors la quantité $\deg(x) = \deg^+(x) + \deg^-(x)$.

Proposition. *On a l'identité* $\sum_{x \in S} \deg(x) = 2|A|$.

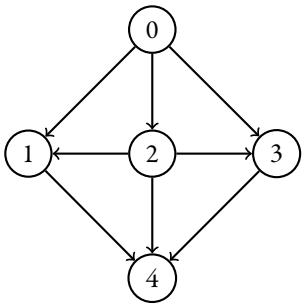
Pour un graphe non orienté, les deux membres de cette égalité subissent un double comptage : à gauche, avec les degrés entrant et sortant ; à droite, avec la transposition des arêtes. On parlera alors indifféremment de degré $\deg(x) = \deg^+(x) = \deg^-(x)$ ***et on considèrera l'ensemble des arêtes à transposition près.*** L'égalité ci-dessus reste ainsi vérifiée avec les quantités moitiés.

Théorème. *Un graphe connexe non orienté admet un chemin Eulérien, c'est-à-dire passant une fois et une seule par chaque arête, si et seulement s'il admet au plus deux sommets de degré impair.*

Esquisse de preuve. Un chemin Eulérien traverse chaque sommet un nombre pair de fois (une en entrant, une en sortant) sauf peut-être ses extrémités ; cela prouve le sens direct.

Réciproquement, soient α et β les deux sommets de degré impair s'ils existent, ou $\alpha = \beta$ un sommet arbitraire sinon. Considérons un chemin simple C de longueur maximale parmi ceux d'extrémités (α, β). Supposons qu'il existe une arête non dans C. Par connexité on peut supposer que l'une de ses extrémités γ est traversée par C. Soit alors un cycle simple D issu de γ et formé d'arêtes de $A \setminus C$. On obtient un chemin plus long que C en y insérant en γ le chemin D, ce qui contredit sa maximalité. Ainsi C est Eulérien. □

Exercice. *Montrer que tout graphe connexe non orienté (S, A) satisfait $|A| \geqslant |S| - 1$.*




```
[| [1;2;3]; [4]; [1,3,4]; [4]; [] |]
```

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

FIGURE 2.11 – Un graphe, sa liste d'adjacence et sa matrice d'adjacence.

2.5.2 Aspects effectifs

Afin de rendre ces considérations effectives, il nous faut représenter les graphes sous forme numérique. Pour cela, fixons en premier lieu une bijection arbitraire $S \simeq \{0, \dots, n-1\}$.

Définition. *On appelle liste d'adjacence d'un graphe le tableau dont l'élément d'indice i est la liste $\{j : (i,j) \in A\}$. On appelle matrice d'adjacence d'un graphe la matrice dont le coefficient d'indice (i,j) vaut 1 si $(i,j) \in A$ et 0 sinon. Voir la figure 2.11.*

Remarque. *Un graphe est non orienté si et seulement si sa matrice d'adjacence est symétrique.*

La pertinence de la notion algébrique de matrice est pleinement justifiée par ce résultat.

Proposition. *La puissance k^e de la matrice d'adjacence a pour coefficient d'indice (x,y) le nombre de chemins de longueur k reliant x à y.*

Exercice. *Pour représenter en langage OCaml les graphes sous forme de listes d'adjacences ou de matrices d'adjacences, on introduit les types suivants :*

```
type graphe_lst = int list array ;;
type graphe_mat = bool array array ;;
```

Écrire, pour chacun de ces deux types, des fonctions permettant de : déterminer si deux sommets sont voisins ; obtenir la liste des voisins d'un sommet ; déterminer si un graphe est non orienté. Comparer leurs complexités.

Choisir une représentation adaptée au problème considéré est crucial pour obtenir des algorithmes efficaces en temps et en mémoire. On pourra notamment observer que, d'un point de vue effectif, la représentation matricielle n'est intéressante que lorsque le nombre d'arêtes est comparable au carré du nombre de sommets.

2.5.3 Plus courts chemins

Soient α et β deux sommets d'une même composante connexe d'un graphe (S, A). Un chemin reliant α à β est dit plus court lorsque sa longueur est minimale. Le problème consistant à calculer de tels chemins est particulièrement pertinent lorsque la notion de longueur est étendue pour assigner à chaque arête une longueur positive arbitraire.

Définition. *Un graphe (S, A) est dit pondéré lorsqu'il est muni d'une application $p : A \to \mathbb{R}_+$. La longueur du chemin $s_0 \to \cdots \to s_n$ est alors la quantité $p(s_0, s_1) + \cdots + p(s_{n-1}, s_n)$.*

Exemple. *Avec l'application constante $p = 1$ on retrouve la notion antérieure de longueur.*

Notons $d(\alpha, \beta)$ la longueur des plus courts chemins de α à β. Pour chaque voisin γ de β, prenons un plus court chemin de α à γ puis rajoutons-y l'arête $\gamma \to \beta$; on obtient ainsi des chemins reliant α à β parmi lesquels se trouve un plus court, ce qui donne l'identité

$$\begin{cases} d(\alpha, \alpha) = 0, \\ d(\alpha, \beta) = \min\left\{d(\alpha, \gamma) + p(\gamma, \beta) : \gamma \in \text{voisins}(\beta)\right\} \quad \text{si } \alpha \neq \beta. \end{cases}$$

Pour calculer cette quantité récursivement, on ordonne les sommets par distance croissante depuis α. Autrement dit, on énumère la composante connexe de α en explorant en priorité les sommets qui en sont les plus proches. On construit ainsi un plus court chemin de α à chaque sommet d'un ensemble croissant.

Algorithme (Dijkstra [11]).

Entrée : *Un sommet α d'un graphe pondéré (S, A, p).*

Sortie : *La longueur des plus courts chemin issus de α.*

1. *Créer un tableau $D = (d_\beta)_{\beta \in S}$ avec coefficients initialisés à ∞.*
2. *Affecter $E \leftarrow S$.*
3. *Affecter $d_\alpha \leftarrow 0$.*
4. *Tant que $E \neq \varnothing$:*
5. *Enlever de E un sommet γ pour lequel d_γ est minimal.*
6. *Pour chaque voisin β de γ :*
7. *Si $d_\beta > d_\gamma + p(\gamma, \beta)$:*
8. *$d_\beta \leftarrow d_\gamma + p(\gamma, \beta)$*
9. *Renvoyer D.*

Chaque itération commence ligne 5 par l'identification d'un somment $\gamma \in E$ minimisant d_γ. Afin d'effectuer cette opération efficacement, on peut stocker les éléments de E dans une file de priorité, donnant au sommet γ la priorité $-d_\gamma$; implantées grâce à une structure de données efficace, les opérations de cette file s'effectueront en temps $O(\log |S|)$.

Proposition. *L'algorithme de Dijkstra est correct. Sa complexité est $O(|A| + |S| \log |S|)$.*

Démonstration. Notons $d(\alpha, \beta)$ la distance des plus courts chemins de α à β. On démontre la correction grâce à l'invariant de boucle suivant, valable en fin d'itération :

$$\begin{aligned} & \forall \beta \notin E, d(\alpha, \beta) = d_\beta \\ \wedge \quad & \forall \beta \in E, d(\alpha, \beta) \leqslant d_\beta = \min\left\{d(\alpha, \gamma) + p(\gamma, \beta) : \gamma \notin E, (\gamma, \beta) \in A\right\}. \end{aligned}$$

□

Exercice. *Modifier cet algorithme afin qu'il renvoie un plus court chemin.*

2.6 Syntaxe puis sémantique de la logique propositionnelle

En logique mathématique, le calcul des propositions étudie exclusivement les expressions composées de connecteurs logiques et de variables propositionnelles pouvant prendre les valeurs de vérité vrai et faux. Il s'oppose au calcul des prédicats, aussi appelé logique du premier ordre, et aux logiques d'ordre supérieur qui font intervenir des quantificateurs portant respectivement sur les éléments et les fonctions d'ensembles.

On rappelle les tables de vérité des connecteurs classiques :

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$	$p \oplus q$
V	V	F	V	V	V	V	F
V	F	F	F	V	F	F	V
F	V	V	F	V	V	F	V
F	F	V	F	F	V	V	F

Dans le cours de logique du premier semestre on a notamment exprimé la redondance entre ces connecteurs sous forme d'identité classiques, telles les règles de De Morgan ($\neg(p \vee q) \equiv \neg p \wedge \neg q$ et $\neg(p \wedge q) \equiv \neg p \vee \neg q$), le tiers exclus ($p \vee \neg p$) et la décomposition de l'implication ($p \Rightarrow q \equiv \neg p \vee q$). Essayons à présent d'inculquer ces notions à une machine.

Exercice. *Définir un type OCaml permettant de représenter des propositions logiques composées de variables x_i avec $i \in \mathbb{N}$, des constantes V et F ainsi que des connecteurs classiques ci-dessus. Écrire une fonction de signature « `bool array -> proposition -> bool` » évaluant une telle proposition en des valeurs de vérité données.*

```
type prop =
    | V
    | F
    | Var of int
    | Non of prop
    | Et of prop * prop
    | Ou of prop * prop
;;
```

Cette définition inculque à OCaml la syntaxe du langage des propositions, c'est-à-dire ses règles d'écriture permettant notamment d'identifier les expressions syntaxiquement correctes.

```
# Ou Non V F V ;;
Error: Syntax error
```

En revanche, OCaml ignore à ce stade la sémantique de ce langage, c'est-à-dire la signification des expressions. La lui inculquer revient à implanter des fonctions interprétant ces expressions.

```
let rec eval t = function
    | V         -> true
    | F         -> false
    | Var i     -> t.(i)
    | Non p     -> not (eval t p)
    | Et (p, q) -> (eval t p) && (eval t q)
    | Ou (p, q) -> (eval t p) || (eval t q)
;;
```

Exercice. *Écrire une fonction calculant la négation d'une proposition. Écrire une fonction simplifiant une proposition.*

On remarquera que les propositions ainsi définies sont représentées sous forme d'arbre et que leurs opérations se traduisent ainsi en termes de manipulation de la structure d'arbre. Ce travail de traduction est interne au langage de programmation.

Définition. *Une proposition P en n variables propositionnelles $x_1, \dots, x_n$ est dite :*
- *une contradiction lorsque $\nexists x \in \{V, F\}^n, P(x)$;*
- *satisfiable lorsque $\exists x \in \{V, F\}^n, P(x)$;*
- *une tautologie lorsque $\forall x \in \{V, F\}^n, P(x)$.*

Exercice. *Écrire une fonction déterminant si une proposition est une tautologie. Si elle est satisfiable. Quels sont leurs complexités ?*

```
let rec maxvar = function
    | V         -> 0
    | F         -> 0
    | Var i     -> i
    | Non p     -> maxvar p
    | Et (p, q) -> max (maxvar p) (maxvar q)
    | Ou (p, q) -> max (maxvar p) (maxvar q)
;;

let rec aux p n t =
    if (eval t p) then true else
    let i = ref 0 in
    while !i < n && t.(!i) do
        t.(!i) <- false;
        incr i
    done;
    if !i = n then false else begin
        t.(!i) <- true;
        aux p n t
    end
;;

let satiasfiable p =
    let n = maxvar p + 1 in
    let t = Array.make n false in
    aux p n t
;;
```

Il est conjecturé qu'il n'existe aucun algorithme déterminant si une proposition est satisfiable en temps polynomial en n. Raffinons maintenant un résultat abordé l'an dernier en exercice du cours de logique et fondements.

Définition. *On appelle littéral toute proposition de la forme « x » ou « $\neg x$ » où x est une variable propositionnelle. On appelle conjonction (resp. disjonction) toute proposition de la forme « $\ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_n$ » (resp. « $\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n$ ») où les ℓ_k sont des littéraux. On appelle forme normale disjonctive (resp. conjonctive), toute proposition de la forme « $C_1 \vee C_2 \vee \cdots \vee C_m$ » où les C_k sont des conjonctions (resp. « $D_1 \wedge D_2 \wedge \cdots \wedge D_m$ » où les D_k sont des disjonctions).*

Proposition. *Toute proposition admet une forme normale disjonctive (resp. conjonctive).*

Démonstration. Par induction structurelle sur l'ensemble des propositions. □

Les formes normales disjonctives se lisent directement sur une table de vérité.

x	y	$P(x,y)$
V	V	V
V	F	F
F	V	V
F	F	V

$$P(x,y) \equiv (x \wedge y) \vee (\neg x \wedge y) \vee (\neg x \wedge \neg y)$$

Exercice. *Écrire une fonction calculant la forme normale disjonctive d'une proposition donnée.*

Chapitre 3

Troisième semestre

3.1 Dictionnaires et programmation dynamique

Nous allons revisiter la notion de dictionnaire en son incarnation mutable et l'appliquer à la résolution de divers problèmes classiques via des techniques de programmation dynamique.

3.1.1 Dictionnaires

Rappelons pour commencer la définition ci-dessous.

Définition. *Un dictionnaire est une structure de données stockant des couples* (*clef*, *valeur*) *où les clefs sont distinctes. Les opérations principales sont :*

- ***constructeur :*** *créer un nouveau dictionnaire ;*
- ***transformateur :*** *ajouter ou enlever un couple ;*
- ***accesseur :*** *déterminer si une clef donnée apparaît dans le dictionnaire ;*
- ***accesseur :*** *lorsque c'est le cas, déterminer la valeur correspondante.*

L'an dernier nous en avions implanté une version immuable en stockant les couples dans un tableau trié : l'indice du couple (x, y) est l'ordinal de la clef x, ce qui permet d'effectuer des recherches rapidement par dichotomie. Un exercice sur les arbres AVL nous a aussi montré comment en réaliser une version mutable.

Les tables de hachage sont une autre implantation de dictionnaires mutables. Comme leur nom l'indique, elles reposent toujours sur un tableau mais l'indice du couple (x, y) est donné par l'image $h(x)$ d'une certaine fonction dite de hachage h : {clefs} $\to \{0, \ldots, \texttt{len(t)} - 1\}$. Celle-ci n'étant pas nécessairement injective, le tableau sera formé de listes de couples. On écrira ainsi :

```
type ('a, 'b) dictionnaire = {
    h : 'a -> int;
    t : ('a * 'b) list array;
} ;;

let constructeur n = {
    h = (function x -> Hashtbl.hash x mod n);
    t = Array.make n [];
} ;;

let apparait d c = List.mem c (List.map fst d.t.(d.h(c))) ;;
let valeur d c = snd (List.find (function x, y -> x = c) d.t.(d.h(c))) ;;
let ajout d c valeur = d.t.(d.h(c)) <- (c, valeur) :: d.t.(d.h(c)) ;;
```

Pour que les fonctions ci-dessus soient efficaces, il faut borner la longueur des listes que contient le tableau ; cela revient à s'assurer que n domine le nombre de couples à stocker et que les images de h soient réparties uniformément.

Définition. *Une fonction $h : \{0,1\}^{(\mathbb{N})} \to \{1,\dots,n\}$ est dite de hachage lorsque :*
— *Il existe un algorithme calculant h efficacement.*
— *Pour tout $y \in \{1,\dots,n\}$, on a :*

$$\lim_{k\to\infty} \mathrm{Prob}\left[h(x) = y : |x| \leqslant k\right] = \frac{1}{n}$$

Exemple. *Soit n un entier. Montrer que l'on définit une fonction de hachage par :*

$$b : \begin{cases} \{0,1\}^{(\mathbb{N})} \longrightarrow \{1,\dots,n\} \\ (c_\ell)_{\ell\in\{1,\dots,k\}} \longmapsto \left(\sum_{\ell=1}^{k} c_\ell 2^{\ell-1}\right) \bmod n \end{cases}$$

La fonction b ci-dessus interprète la clef comme la décomposition binaire d'un entier qui est alors réduit modulo n. Cette structure relativement transparente peut s'avérer problématique : par exemple, si n est pair et que toutes les clefs considérées vérifient $c_0 = 0$, alors au plus la moitié des emplacements du tableau seront utilisés. Ce type de faiblesses est évité par les fonctions de hachage cryptographiques dont la conception est toutefois plus complexe et dépasse largement le cadre de ce cours.

Rappelons enfin que les dictionnaires mutables sont une structure de données directement offerte par les langages et donc librement utilisable aux concours.

En langage Python.

```
dict = {2: 1, 4: 2, 8: 3, 16: 4}
dict[4]       # 2
dict[3]       # KeyError: 3
dict[1] = 0
dict.pop(8)   # 3
dict[32] = 5
dict.keys()   # dict_keys([2, 4, 16, 1, 32])
32 in dict    # true
for clef in dict:
    print(clef)
```

En langage OCaml.

```
Hashtbl.create   : int -> ('a, 'b) Hashtbl.t
Hashtbl.add      : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit
Hashtbl.remove   : ('a, 'b) Hashtbl.t -> 'a -> unit
Hashtbl.mem      : ('a, 'b) Hashtbl.t -> 'a -> bool
Hashtbl.find     : ('a, 'b) Hashtbl.t -> 'a -> 'b
Hashtbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b option
Hashtbl.iter     : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit
```

Exercice. *Le service de la scolarité souhaite se doter de deux tables de hachage : l'une associe au numéro étudiant l'état civil (nom, prénom, date de naissance) ; l'autre associe au numéro étudiant la liste des filières d'inscription. Définir des types et des fonctions OCaml implantant cela. Écrire une fonction renvoyant la listes des prénoms et noms des étudiants d'une filière donnée. Écrire une fonction déterminant le prénom le plus populaire en CUPGE.*

3.1.2 Programmation dynamique

La programmation dynamique est un principe général consistant à résoudre un problème en le décomposant récursivement en sous problèmes, ceux-ci étant alors résolus par taille croissante et leurs résultats combinés afin d'obtenir une solution du problème initial. Le terme « programmation » revêt ici son sens historique antérieur à l'informatique moderne signifiant « organisation » ou « planification » des calculs, tel un programme des réjouissances.

Afin d'illustrer ce principe, considérons le problème consistant à déterminer la valeur du n^{e} terme de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$. Rappelons que la méthode naïve ci-dessous est de complexité $O(\varphi^n)$; il s'agit donc ici de faire mieux.

```
def fibo1(n):
    if n < 2:
        return n
    return fibo1(n - 1) + fibo1(n - 2)
```

Le problème consistant à calculer F_n se décompose naturellement en deux sous problèmes consistant à calculer respectivement F_{n-1} et F_{n-2} ; eux-mêmes se décomposent et on a ainsi une famille de sous problèmes à résoudre récursivement. Il s'agit maintenant de les ordonner afin de les résoudre du plus petit au plus grand. On dispose pour ce faire de deux méthodes.

Méthode ascendante. Elle consiste à identifier un ordre sur les sous problèmes compatible avec la relation de dépendance. Dans le cas de la suite de Fibonacci, il est clair que F_0 et F_1 peuvent se calculer immédiatement, que l'on pourra alors calculer F_2, puis F_3, etc.

```
def fibo2a(n):
    F = (n + 1) * [0]
    F[1] = 1
    for k in range(2, n + 1):
        F[k] = F[k - 1] + F[k - 2]
    return F[n]
```

Méthode descendante, dite mémoïsation. Elle consiste à procéder par récurrence depuis le problème final tout en maintenant un cache des solutions des sous problèmes déjà traités. Ce cache est typiquement stocké dans un dictionnaire. Avant tout calcul, on regarde si la solution n'y est pas déjà. Un ordre compatible avec la relation de dépendance est ainsi automatiquement construit lors de l'exécution du programme.

```
F={}
def fibo2b(n):
    if not n in F:
        if n < 2:
            F[n] = n
        else:
            F[n] = fibo2b(n - 1) + fibo2b(n - 2)
    return F[n]
```

Remarquons que ces techniques donnent des résultats parfaitement équivalents : les deux programmes `fibo2a` et `fibo2b` sont de complexité $O(n)$ et leur seule différence est que l'ordonnancement des calculs est explicite dans le premier alors qu'il est implicite dans le second. Comme l'illustre la figure 3.1, ces méthodes éliminent les calculs redondants effectués par la méthode naïve.

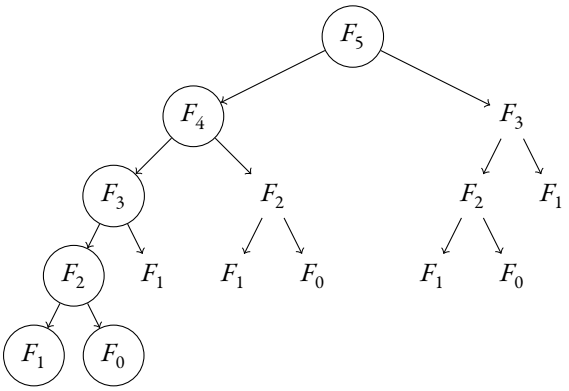


FIGURE 3.1 – Arbre de dépendance des sous problèmes du calcul de F_5 ; la programmation dynamique élimine la redondance en ne traitant que les nœuds entourés.

Définition. *Un problème satisfait la propriété de sous structure optimale lorsqu'une solution du problème peut être calculée à partir de solutions de ses sous problèmes. Un problème présente un chevauchement de sous problèmes lorsque des sous problèmes identiques apparaissant dans les décompositions de deux problèmes distincts.*

Essentiellement, lorsqu'on problème satisfait la propriété de sous structure optimale, on peut le résoudre grâce aux techniques de programmation dynamique ; lorsqu'il présente un chevauchement de sous problèmes, ces techniques seront plus efficaces qu'une approche naïve.

Exemple. *Le problème du calcul de F_n satisfait la propriété de sous structure optimale puisqu'il se décompose en celui de F_{n-1} et de F_{n-2} ; il présente aussi un chevauchement de sous problèmes car les calculs de F_{n-1} et F_{n-2} font tous deux appel au calcul de F_{n-3}. On constate effectivement que la programmation dynamique permet de le résoudre en $O(n)$ opérations arithmétiques contre $O(\varphi^n)$ pour l'approche naïve.*

Armés de cette terminologie, attaquons quelques applications phares de la programmation dynamique. On s'attellera à expliciter des algorithmes conçus par calcul ascendant et par calcul descendant puis à en déterminer les complexités.

Problème du sac-à-dos avec répétitions. Soient n objets indicés par $i \in \{1, \dots, n\}$ possédant chacun une valeur v_i et un poids p_i. On cherche la valeur maximale $m(P)$ réalisable par une sélection d'objets de poids $\leqslant P$. Supposant que l'on dispose d'une infinité d'exemplaires de chaque objet, ce problème satisfait la propriété de sous structure optimale puisqu'on a

$$m(P) = \max\{m(P - p_i) + v_i : i \in \{1, \dots, n\}, p_i \leqslant P\}$$

avec la convention $\max \varnothing = 0$.

Problème du sac-à-dos sans répétitions. Lorsque chaque objet ne peut être sélectionné qu'une fois au plus, déterminer la quantité $m(P)$ n'est plus un problème satisfaisant la propriété de sous structure optimale. Il s'agit alors d'introduire une nouvelle quantité adaptée à cette contrainte supplémentaire. Par exemple, la valeur maximale $m(P, k)$ réalisable par une sélection d'objets de $\{1, \dots, k\}$ de poids $\leqslant P$ satisfait

$$m(P, k) = \max\{m(P - p_k, k - 1) + v_k, m(P, k - 1)\}.$$

Problème des plus longues sous suites croissantes. Soit $[x_0, \dots, x_{n-1}]$ une liste d'entiers et soit à en déterminer une sous suite croissante de longueur maximale. Ce problème ne satisfait pas lui-même la propriété de sous structure optimale : la plus longue sous suite croissante de $[2, 0, 6, 1, 5, 3]$ ne peut se calculer à partir des plus longues sous suites croissantes de $[2, 0, 6]$ et de $[1, 5, 3]$. Il s'agit donc de le généraliser ; une possibilité est d'introduire :

> Calculer la longueur ℓ_k d'une sous suite croissante de longueur maximale dont le dernier terme est x_k.

Ce nouveau problème satisfait la propriété de sous structure optimale par l'identité

$$\ell_k = 1 + \max\{\ell_i : i \in \{0 \dots, k-1\}, x_i \leqslant x_k\}.$$

Exercice. *Quelle complexité obtient on ainsi pour le calcul de la plus longue sous suite croissante par la programmation dynamique ? Quelle aurait été la complexité d'une approche naïve ?*

Problème des plus courts chemins. Soit $G = (S, A, p)$ un graphe orienté pondéré. Sans perte de généralité on suppose l'ensemble des sommets $S = \{1, \dots, n\}$ fixé. Afin de résoudre le problème des plus courts chemins, on introduit le problème plus général que voici :

> Calculer $d_k(i, j)$, la longueur d'un plus court chemin allant de i à j en passant exclusivement par des sommets de $\{1, \dots, k\}$.

Ce nouveau problème se réduit à celui des plus courts chemins lorsque $k = n$ mais tout son intérêt est qu'il satisfait la propriété de sous structure optimale ; en effet, on a l'identité

$$d_k(i, j) = \min\big(d_{k-1}(i, j),\ d_{k-1}(i, k) + d_{k-1}(k, j)\big).$$

L'ordonnancement est clair et on obtient ainsi par calcul ascendant l'algorithme ci-après.

Algorithme (Floyd–Warshall [12, 16, 17])**.**

Entrée : *Un graphe orienté pondéré (S, A, p) avec $S = \{1, \dots, n\}$.*
Sortie : *Une matrice D telle que $d_{i,j}$ est la longueur du plus court chemin de i à j.*

1. *Créer D de taille $n \times n$ avec coefficients initialisés à $+\infty$.*
2. *Pour tout i de* 1 *à n :*
3. $\quad d_{i,i} \leftarrow 0$
4. *Pour tout $(i, j) \in A$:*
5. $\quad d_{i,j} \leftarrow p(i, j)$
6. *Pour k de* 1 *à n :*
7. $\quad$ *Pour i de* 1 *à n :*
8. $\quad\quad$ *Pour j de* 1 *à n :*
9. $\quad\quad\quad$ *Si $d_{i,j} > d_{i,k} + d_{k,j}$:*
10. $\quad\quad\quad\quad d_{i,j} \leftarrow d_{i,k} + d_{k,j}$
11. *Renvoyer D.*

Cet algorithme est grossièrement inadéquat dans le cas de graphes creux pour lesquels une liste d'adjacence est l'option de stockage par excellence ; en effet, il énumère systématiquement chaque triplet (i, j, k) quand bien même A serait significativement plus petit que S^2. En revanche, il est parfaitement adapté à une représentation matricielle des graphes.

Exercice. *Modifier l'algorithme ci-dessus de sorte qu'il calcule la matrice des plus courts chemins et non seulement celle de leurs longueurs.*

3.2 Applications des arbres

Appliquons à deux problèmes concrets les notions vues précédemment.

3.2.1 Retour sur trace

Issu de l'anglais *backtracking*, ce terme désigne la méthode consistant, pour résoudre un problème, à construire des solutions partielles en les étendant progressivement et en rejetant celles ne pouvant être étendues. Elle est adaptée aux problèmes pour lesquels on peut facilement déterminer si une solution partielle peut être étendue, tels les sudoku ou encore les mots croisés. Formellement, cela revient à réaliser un parcours en profondeur de l'arbre des solutions partielles (la racine étant la solution vide et chaque fils étendant la solution père).

Problème (huit dames). *Comment placer huit dames sur un échiquier sans aucune menace ?*

Énumérer tous les placements de huit dames sur un échiquier puis sélectionner ceux sans menaces serait délicat, le nombre de tels placements étant $C_{64}^{8} \approx 2^{32}$. On adopte donc la méthode ci-dessus en plaçant les dames une par une sur un échiquier initialement vide. Remarquant que toute solution place une unique dame par colonne, on placera d'abord la dame de la première colonne, puis celle de la seconde, etc. L'essence du retour sur trace étant que, dès que deux dames se menacent, on élimine la dernière dame placée sans considérer les suivantes.

```
let affiche p =
    let n = Array.length p in
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            print_string (if p.(i) = j then "X " else "- ")
        done;
        print_string "\n"
    done;
    print_string "\n"
;;

let rec menace p k i =
    if i = k then false
    else
        p.(k) - p.(i) = 0
        || p.(k) - p.(i) = k - i
        || p.(k) - p.(i) = i - k
        || menace p k (i + 1)
;;

let dames n =
    let p = Array.make n (-1) in
    let k = ref 0 in
    while !k >= 0 do
        p.(!k) <- p.(!k) + 1;
        if p.(!k) = n then decr k
        else if not (menace p !k 0) then
            if !k = n - 1 then affiche p
            else (incr k; p.(!k) <- -1)
    done
;;
```

```
000: def factorielle(n):
001:     m=1
002:     if n>2: m=factorielle(n-1)
003:     return m*n
```

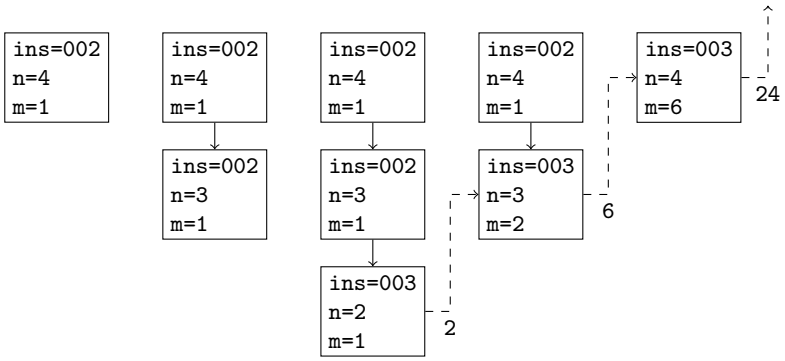


FIGURE 3.2 – Évolution de la pile d'exécution pour `factorielle(4)`.

Exercice. *Un labyrinthe est codé par une matrice de taille $n \times n$ dont les coefficients nuls correspondent à des couloirs et les autres à des murs. Appliquer le principe du retour sur trace afin de trouver un chemin reliant $(0,0)$ à $(n-1, n-1)$.*

La structure de donnée facilitant le retour sur trace est la pile : elle stocke les solutions partielles, empilant leurs extensions et dépilant celles ne pouvant être étendues. Lorsqu'on résout un tel problème par un programme récursif, la pile généralement est cachée dans les appels de fonction.

Les systèmes d'exploitation gèrent en effet les appels de fonctions à l'aide d'une structure de donnée appelée pile d'exécution. La pile d'exécution d'un processus référence tous les appels de fonctions en cours ; chaque objet de la pile correspond à un appel non encore achevé et stocke :

— les valeurs des arguments et variables ;
— l'adresse de l'instruction courante.

Lorsque l'instruction courante est un appel de fonction, le système ajoute une entrée pour cet appel et l'exécute immédiatement ; lorsque l'appel termine il supprime son entrée, met à jour les variables de l'appel précédent et reprend son exécution à l'instruction suivante. Voir les figures 3.2 et 3.3 pour des illustrations dans le cas de fonctions respectivement simplement et doublement récursives.

Exercice. *Écrire un programme énumérant tous les sudokus.*

3.2.2 File de priorité

L'algorithme de Dijkstra vu l'an dernier reposait sur une structure de donnée alors non implantée : la file de priorité. Il s'agit de stocker une collection d'éléments chacun étiqueté par un nombre réel appelé priorité de sorte que l'on puisse rapidement identifier et enlever l'élément de priorité maximale. Pour réaliser cette structure de données, approfondissons quelques notions des arbres.

Définition. *Un arbre binaire est dit posséder la propriété de tas si l'étiquette de chacun de ses nœuds est supérieure à celles de ses fils.*

```
001: def fibonacci(n):
002:     a=0
003:     b=0
004:     if n<2: return n
005:     a=fibonacci(n-1)
006:     b=fibonacci(n-2)
007:     return a+b
```

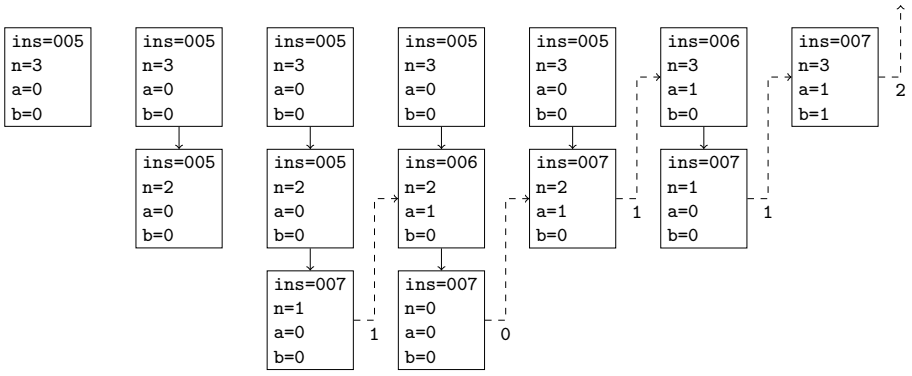


FIGURE 3.3 – Évolution de la pile d'exécution pour `fibonacci(3)`.

Exercice. *Écrire une fonction de signature « `float arbreBE -> bool` » déterminant si un arbre étiqueté par des réel possède la propriété de tas.*

```
let prop_tas =
    let rec aux m = function
      | Nil -> true
      | Noeud (e, g, d) -> e <= m && aux e g && aux e d
    in
    aux 99999
;;
```

Cette propriété facilite évidemment l'identification de l'élément d'étiquette maximale : c'est la racine ! Afin de l'enlever en préservant la propriété de tas, il s'agit de remonter récursivement celui des deux fils de plus grande étiquette.

Exercice. *Écrire une fonction de signature « `float arbreBE -> float arbreBE` » qui, étant donné un arbre possédant la propriété de tas, renvoi un nouvel arbre possédant cette même propriété dont l'ensemble des nœuds est celui de l'arbre original privé de sa racine.*

```
let rec enlever = function
    | Nil -> Nil
    | Noeud (e, g, Nil) -> g
    | Noeud (e, Nil, d) -> d
    | Noeud (e, g, d) -> (
        match (g, d) with
        | Noeud (e, _, _), Noeud (f, _, _) ->
            if e < f then Noeud (f, g, enlever d)
            else Noeud (e, enlever g, d))
;;
```

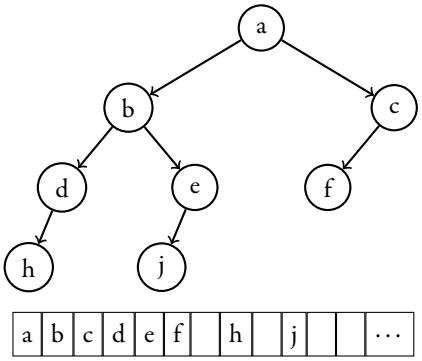


FIGURE 3.4 – Un arbre binaire non presque complet et sa représentation tabulaire.

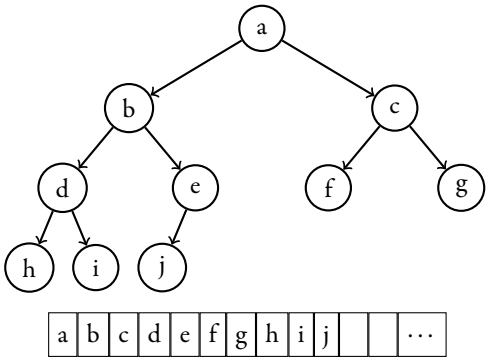


FIGURE 3.5 – Un arbre binaire presque complet et sa représentation tabulaire.

Ce processus, appelé percolation, est de complexité $O(h)$. Rappelons que tout arbre binaire satisfait $h < t < 2^{h+1}$. Il est donc opportun, comme nous l'avions fait pour les arbres de recherche, de minimiser la hauteur à taille constante. Pour cela, on peut ici faire bien mieux que d'exiger que les arbres soient équilibrés.

Définition. *On dit que le tableau T représente l'arbre binaire A lorsque :*
- *L'étiquette de la racine de A est stockée à l'indice 0 dans T.*
- *Si l'étiquette d'un nœud est stockée à l'indice n, alors :*
  - *Celle de son fils gauche est stockée à l'indice $2n + 1$.*
  - *Celle de son fils droit est stockée à l'indice $2n + 2$.*

L'arbre A est alors dit presque complet lorsque T est connexe. Voir les figures 3.4 et 3.5.

Définition. *On appelle tas tout arbre binaire presque complet possédant la propriété de tas.*

Les tas sont généralement implantés via leur représentation tabulaire.

Exercice. *Écrire une fonction « `enlever : float array -> unit` » identique à la précédente mais opérant directement sur l'opération tabulaire. On considèrera comme libre toute case du tableau contenant le flottant `neg_infinity`.*

```
let vide = neg_infinity ;;
```

```
let enlever t =
    let rec aux k =
        let g = (2 * k) + 1 in
        let d = (2 * k) + 2 in
        if t.(g) = vide && t.(d) = vide then t.(k) <- vide
        else (
            let f = if t.(g) < t.(d) then d else g in
            t.(k) <- t.(f);
            aux f
        )
    in
    aux 0
;;
```

Reste à construire un tas à partir d'un tableau d'éléments arbitraires. L'approche naïve consistant à répéter insertions à la racine et percolations ne peut garantir que l'arbre obtenu soit presque complet. On construira donc un tas non pas « par le haut » mais « par le bas » comme il suit.

Exercice. *Écrire une fonction « `ajouter : float array -> float -> unit` » qui rajoute le nombre flottant donné à la représentation tabulaire du tas donné. On insèrera pour cela le flottant donné à la première case libre du tableau puis on l'échangera récursivement avec son père tant qu'il lui est supérieur.*

```
let rec ajouter t k x =
    let p = (k - 1) / 2 in
    if k = 0 || x < t.(p) then t.(k) <- x
    else (
        t.(k) <- t.(p);
        ajouter t p x
    )
;;
```

Exercice. *Écrire une fonction « `tas : float list -> float array` » renvoyant la représentation tabulaire d'un tas contenant les éléments de la liste donnée. On procèdera par appels répétés à `ajout`. Quelle est la complexité de cette fonction ?*

```
let tas l =
    let n = List.length l in
    let t = Array.make ((2 * n) + 1) vide in
    let rec aux k = function
        | [] -> ()
        | x :: y ->
            ajouter t k x;
            aux (k + 1) y
    in
    aux 0 l;
    t
;;
```

Un fois ce tas créé, on peut en enlever de manière itérée la racine afin d'en obtenir les éléments dans l'ordre décroissant. Cette méthode porte le nom de tri par tas [19] et est de complexité $O(n \log n)$.

```
let tri l =
    let t = tas l in
    let rec aux () =
        if t.(0) = vide then []
        else
            let x = t.(0) in
            x :: aux (enlever t)
    in
    List.rev (aux ())
;;
```

3.3 Graphes avancés

L'objectif de cette section est d'approfondir nos connaissances sur la structure de graphe, notamment en exploitant ses interactions avec la structure d'arbre. On considèrera ici exclusivement le cas des graphes non orientés.

3.3.1 Arbres couvrants

Commençons par raffiner la notion de cycle.

Définition. *On appelle cycle primitif d'un graphe tout chemin $s_0 \to s_1 \to \cdots \to s_n$ de longueur $n \geqslant 3$ vérifiant $s_0 = s_n$ et dont les sommets $s_0, s_1, \ldots, s_{n-1}$ sont distincts. Un graphe est dit acyclique s'il ne possède pas de cycle primitif.*

Définition. *Soit $G = (S, A)$ un graphe non orienté. On appelle sous graphe de G tout graphe non orienté (T, B) vérifiant $T \subset S$ et $B \subset A \cap T^2$. Ce sous graphe est qualifié de couvrant lorsque $S = T$. Ce sous graphe est dit arbre couvrant s'il est couvrant, connexe et acyclique.*

Intuitivement, un arbre couvrant relie tous les sommets du graphe et possède une structure d'arbre déracinée : le choix arbitraire d'une racine parmi les sommets suffit à lui conférer une structure d'arbre ; on restera néanmoins conscient que celle-ci n'est pas nécessairement binaire et que les fratries ne sont pas munies d'un ordre. Précisons le lien de cette notion avec la connexité.

Lemme. *Si un graphe non orienté admet un arbre couvrant, il est connexe.*

Démonstration. Soit G un graphe non orienté, H un arbre couvrant et α et β deux sommets de G. Puisque H est couvrant, α et β sont dans H. Puisque H est connexe, il existe un chemin reliant α à β dans H. Et puisque H est un sous graphe de G, ce chemin est encore dans G. □

Proposition. *Un graphe non orienté connexe est acyclique si et seulement s'il est son unique sous graphe couvrant connexe.*

Démonstration. Soit $G = (S, A)$ un graphe non orienté connexe. S'il est acyclique et que (S, B) est un sous graphe couvrant connexe avec $B \subsetneq A$, soit une arête $(\alpha, \beta) \in B \setminus A$. Par connexité de (S, B), il existe un chemin $\alpha \to \cdots \to \beta$; en le concaténant avec l'arête $\beta \to \alpha$, on obtient un cycle primitif contredisant l'acyclicité de (S, A). On en déduit $A = B$.

Réciproquement, si G n'est pas acyclique, soit $\alpha \to \cdots \to \beta \to \alpha$ un cycle primitif. Alors le sous graphe $(S, A \setminus \{(\alpha, \beta)\})$ est couvrant, connexe et distinct de (S, A) lui-même, qui n'est ainsi pas unique. □

Ces propriétés fournissent des caractérisations alternatives des arbres couvrants.

Proposition. *Soit H un sous graphe d'un graphe connexe non orienté G. Les trois propositions ci-après sont équivalentes.*

1. *H est un arbre couvrant de G.*
2. *H est maximal parmi les sous graphes acycliques de G.*
3. *H est minimal parmi les sous graphes couvrants connexes de G.*

La maximalité et minimalité de $H = (T, B)$ est à interpréter au sens de l'inclusion sur T et B.

Démonstration. Montrons $(1) \Rightarrow (2)$. Soit $H = (S, B)$ un arbre couvrant et soit $H' = (S, B')$ un sous graphe acyclique vérifiant $B \subsetneq B'$. Soit une arête $(\alpha, \beta) \in B' \setminus B$. Par connexité de H, il existe un chemin $\alpha \to \cdots \to \beta$ dans H ; en le concaténant avec l'arête $\beta \to \alpha$, obtient un cycle primitif contredisant l'acyclicité de H'. On en déduit $B = B'$ et H est maximal.

Montrons $(2) \Rightarrow (3)$. Soit $H = (S, B)$ un sous graphe acyclique maximal. Soient α et β deux sommets. Par connexité de G, il existe un chemin $\alpha \to s_1 \to \cdots \to s_n = \beta$ dans G. Soit k le plus petit entier pour lequel s_k n'est pas dans la composante connexe de α dans H. Le sous graphe $(S, B \cup \{(s_{k-1}, s_k)\})$ est alors acyclique, ce qui contredit la maximalité de H. L'entier k n'existe donc pas. Ainsi, H est connexe et la proposition précédente s'applique.

Montrons $(3) \Rightarrow (1)$. Soit $H = (S, B)$ minimal parmi les sous graphes couvrants connexes de G. Si $s_0 \to s_1 \to \cdots s_n = s_0$ est un cycle dans H, alors le sous graphe $(S, B \setminus \{s_{n-1}, s_n\})$ est encore couvrant et connexe ce qui contredit la minimalité de H. Ainsi H est acyclique et c'est un arbre couvrant. □

Corollaire. *Tout graphe connexe non orienté admet un arbre couvrant.*

Exercice. *Concevoir un algorithme qui détermine si un graphe connexe non orienté est acyclique.*

Une idée naïve est d'exploiter la proposition ci-dessus en vérifiant qu'aucun sous graphe couvrant ne soit connexe. On obtient ainsi un algorithme de complexité $O(|S|^2 \log |S|)$.

Algorithme (acyclicité naïve).
Entrée : *Un graphe connexe non orienté (S, A).*
Sortie : *S'il est acyclique.*

1. *Pour toute arête $(\alpha, \beta) \in A$:*
2. *Si le graphe $(S, A \setminus \{(\alpha, \beta)\})$ est connexe, renvoyer* `faux`.
3. *Renvoyer* `vrai`.

Une meilleure idée consiste à parcourir les sommets du graphe en partant d'un sommet initial arbitraire et vérifiant qu'aucun sommet n'est atteint par deux chemins distincts. L'algorithme obtenu est de complexité $O(|S| \log |S|)$.

Algorithme (acyclicité).
Entrée : *Un graphe connexe non orienté (S, A).*
Sortie : *S'il est acyclique.*

1. *Construire un ensemble V contenant un unique sommet arbitraire de S.*
2. *Tant que $A \neq \emptyset$:*
3. *Sélectionner une arête $(\alpha, \beta) \in A$ avec $\alpha \in V$.*
4. *Si $\beta \in V$, renvoyer* `faux`.
5. *Rajouter β à V et enlever (α, β) de A.*
6. *Renvoyer* `vrai`.

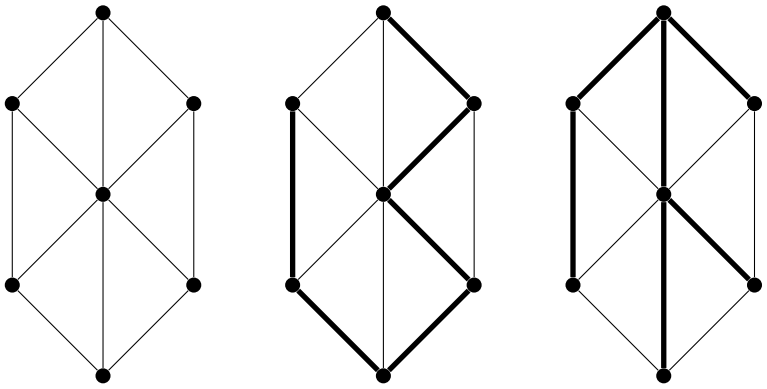


FIGURE 3.6 – Un graphe et, superposés en gras sur la figure du milieu et sur celle de droite, des arbres couvrants obtenus comme arborescences respectivement d'un parcours en profondeur et d'un parcours en largeur.

La construction d'un arbre couvrant permet de ramener certains problèmes de graphes à leurs homologues plus faciles sur les arbres. Pour ce faire, l'approche la plus directe consiste à réaliser un parcours de l'arbre.

Définition. *On appelle parcours d'un graphe tout procédé qui en énumère les sommets de proche en proche. Plus précisément, cela consiste à sélectionner, étant donnés les sommets déjà visités, le prochain sommet à visiter. Un parcours est dit en largeur (resp. profondeur) lorsqu'il visite en premier les voisins des sommets visités en premier (resp. dernier).*

Exemple. *Sur un graphe non orienté avec des arêtes de poids unité, l'algorithme de Dijkstra réalise un parcours en largeur.*

Cette notion généralise celle déjà vue dans le cas particulier des arbres. Voir la figure 3.6. Rappelons que les parcours en largeur s'implantent typiquement à l'aide d'une file et les parcours en profondeur à l'aide d'une pile.

Définition. *On appelle arborescence d'un parcours le sous graphe constitué des arêtes ayant permis de sélectionner chaque sommet visité.*

Remarquer que deux parcours distincts peuvent avoir la même arborescence ; par exemple, si le graphe initial est un arbre, tout parcours donnera la même arborescence.

Proposition. *L'arborescence d'un parcours d'un graphe connexe non orienté est un arbre couvrant.*

Exercice. *Démontrer la proposition ci-dessus.*

Parmi tous les arbres couvrants, qui s'obtiennent donc comme arborescences de parcours, on peut en rechercher présentant des propriétés particulières et notamment d'optimalité.

Définition. *Soit un graphe pondéré non orienté $G = (S, A, p)$. On appelle poids du sous graphe (S, B) la quantité $\sum_{a \in B} p(a)$.*

La construction d'un arbre couvrant de poids minimal se prête à un algorithme glouton : à chaque itération, on sélectionne l'arête de plus petits poids ; si on obtient un cycle en rajoutant cette arête à l'arbre couvrant en cours de construction cyclique, on la jette ; sinon, on l'y rajoute.

Algorithme (Kruskal [7]).

Entrée : *Un graphe pondéré connexe non orienté* (S, A, p).

Sortie : *Un arbre couvrant* (S, B) *de poids minimal.*

1. *Soit* E *l'ensemble des sous graphes* $(\{s\}, \varnothing)$ *lorsque* s *parcours* S.
2. *Tant que* $|E| > 1$ *:*
3. *Enlever de* A *l'arête* a *de poids minimal.*
4. *Si ses extrémités appartiennent à deux éléments distincts* $H_1 \neq H_2$ *de* E *:*
5. *Enlever* H_1 *et* H_2 *de* E *et y rajouter* $H_1 \cup H_2 \cup a$.
6. *Renvoyer l'unique élément de* E.

Afin de réaliser efficacement l'étape 3 on peut soit utiliser une file de priorité ou, puisqu'aucun élément n'est jamais rajouté à cette file, préalablement trier suivant p l'ensemble des arêtes A.

Proposition. *L'algorithme de Kruskal est correct et a pour complexité* $O(|A| \log |A|)$.

Démonstration. Montrer que la proposition « il existe un arbre couvrant de poids minimal obtenu en complétant $\bigcup_{H \in E} H$ par des arêtes de A » est un invariant de boucle. □

Exercice. *Démontrer que, si les poids des arêtes sont tous distincts, l'arbre couvrant de poids minimal est unique. Que dire de la réciproque ?*

3.3.2 Couplages des graphes bipartis

Considérons les deux problèmes isomorphes ci-après.

Problème (des mariages). *Étant donnés un ensemble de garçons, un ensemble de filles et l'ensemble des couples* (*garçon*, *fille*) *compatibles, combien peut on célébrer de mariages monogames ?*

Problème (d'affectation). *Étant donnés un ensemble d'agents, un ensemble de tâches et l'ensemble des couples* (*agent*, *tâche*) *compétents, combien de tâches peut on réaliser en parallèle ?*

Considérant les graphes dont les arêtes sont les couples évoqués ci-dessus, ces problèmes peuvent être formalisés comme ceci.

Définition. *Soit* $G = (S, A)$ *un graphe non orienté. Deux arêtes de* A *sont dites indépendantes si elles n'ont pas d'extrémité commune. On appelle couplage de* G *toute partie* $C \subset A$ *formée d'arêtes deux-à-deux indépendantes.*

Problème. *Trouver un couplage de cardinal maximum dans un graphe non orienté.*

La notion que voici va nous permettre de construire des couplages de cardinal croissant afin d'espérer en atteindre un de cardinal maximum.

Définition. *Soit* C *un couplage d'un graphe* G. *Un sommet* s *est dit couplé par* C *si c'est l'extrémité d'une arête de* C. *Un chemin est dit alternant pour* C *s'il passe alternativement par des arêtes de* C *et de* $\overline{C}$. *Un chemin augmentant pour* C *est un chemin alternant dont le premier et le dernier sommet sont distincts et non couplés.*

Proposition. *Soit* P *un chemin augmentant pour un couplage* C. *L'ensemble d'arêtes* $(C \setminus P) \cup (P \setminus C)$ *est un couplage de cardinal strictement supérieur à celui de* C. *On appelle cet ensemble l'augmentation du couplage* C *par le chemin augmentant* P.

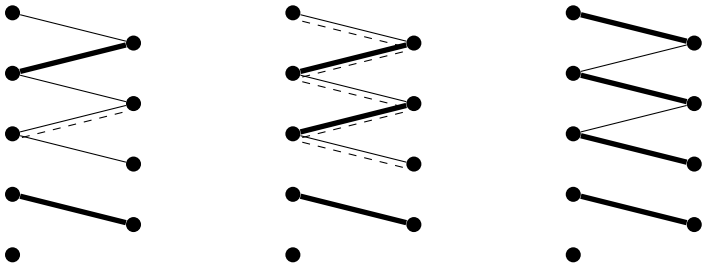


FIGURE 3.7 – Trois exemplaires d'un même graphe (arêtes fines) accompagnés chacun d'un couplage (arêtes grasses) et d'un chemin augmentant (arêtes pointillées décalées). L'augmentation du chemin de gauche (resp. milieu) donne le couplage du milieu (resp. droite) ; le couplage de droite est de cardinalité maximum et n'admet donc aucun chemin augmentant.

Démonstration. Notons $P = s_0 \to s_1 \to \cdots \to s_n$; puisque c'est un chemin augmentant, l'entier n est impair et les arêtes (s_{2k}, s_{2k+1}) sont dans $\overline{C}$ alors que celles (s_{2k+1}, s_{2k+2}) sont dans C. On a ainsi $P \smallsetminus C = \{(s_0, s_1), (s_2, s_3), \ldots\}$. Soient deux arêtes de $(C \smallsetminus P) \cup (P \smallsetminus C)$; on montre qu'elles sont indépendantes si elles sont toutes deux dans C, toutes deux dans $P \smallsetminus C$ ou encore si l'une est dans C et l'autre dans $P \smallsetminus C$. □

Voir la figure 3.7 pour une illustration de cette construction.

Lemme (Berge [8]). *Un couplage est de cardinalité maximum si et seulement s'il n'admet aucun chemin augmentant.*

Démonstration. Le sens direct découle de la proposition ci-dessus. Réciproquement, soit C un couplage n'admettant aucun chemin augmentant. Étant donné un second couplage D, on considère le sous graphe $(C \smallsetminus D) \cup (D \smallsetminus C)$. Chaque sommet y étant de degré au plus deux, ses composantes connexes sont des cycles ou des chemins où les arêtes de C et D alternent. Tout cycle est donc de longueur pair et contient autant d'arêtes de C que de D. Si un chemin contenait davantage d'arêtes de D que de C il serait augmentant pour C. On en conclue que C est de cardinal supérieur ou égal à D. □

La construction d'un couplage maximum se ramène donc à celle de chemin augmentant.

Algorithme (Couplage maximum).

ENTRÉE : *Un graphe.*

SORTIE : *Un couplage de cardinal maximum.*

1. *Soit C le couplage vide.*
2. *Tant que C admet un chemin augmentant P :*
3. *Remplacer C par son couplage augmenté suivant P.*
4. *Renvoyer C.*

L'algorithme d'Edmonds [20] permet de construire un chemin augmentant pour un couplage donné dans un graphe non orienté arbitraire. Il est cependant assez complexe en toute généralité. Aussi, nous n'en considérerons ici qu'un cas particulier beaucoup plus abordable qui permet de traiter les problèmes des mariages et d'affectation évoqués plus haut.

Définition. *On appelle bipartition d'un graphe non orienté $G = (S, A)$ toute partition $S = S_1 \sqcup S_2$ vérifiant $A \subset (S_1 \times S_2) \cup (S_2 \times S_1)$. Un graphe est dit biparti s'il admet une bipartition.*

Considérons le graphe biparti $S_1 = \{\text{garçons}\}$, $S_2 = \{\text{filles}\}$ et $A = \{\text{couples compatibles}\}$; un couplage met en relation certains éléments de S_1 avec certains de S_2 de sorte qu'aucun ne soit en relation deux fois ou plus. Cela traduit donc le problème des mariages qui se réduit ainsi à la recherche d'un couplage de cardinal maximum dans un graphe biparti.

On admettra le théorème ci-après qui n'a pour vocation que d'illustrer à quel point la notion de graphe biparti est naturelle.

Théorème. *Soit G un graphe non orienté. Les trois propositions ci-après sont équivalentes.*

1. *G est biparti.*
2. *G n'admet aucun cycle de longueur impaire.*
3. *Le spectre de la matrice d'adjacence de G est symétrique.*

Démonstration. Montrons $1 \Rightarrow 2$. Soit $(x_0, x_1, \dots, x_n = x_0)$ un cycle d'un graphe biparti $(S_1 \sqcup S_2, A)$. Supposant $x_0 \in S_1$, on démontre par récurrence $x_{2k} \in S_1 \wedge x_{2k+1} \in S_2$. Puisque $x_n = x_0 \in S_1$, la longueur n est paire. On traite pareillement le cas $x_0 \in S_2$.

Montrons $2 \Rightarrow 1$. Soit T un ensemble contenant exactement un sommet de chaque composante connexe de G. Notons S_1 (resp. S_2) l'ensemble des sommets x pour lesquels il existe un chemin de longueur impaire (resp. paire) reliant x à un sommet de T. On montre alors $S = S_1 \cup S_2$ puis $S_1 \cap S_2 = \varnothing$ et enfin $A \subset S_1 \times S_2 \cup S_2 \times S_1$. □

Exercice. *Écrire un programme déterminant si un graphe donné par sa liste d'adjacence est biparti et renvoyant une bipartition le cas échéant. En déterminer la complexité.*

Algorithme (biparti)**.**

Entrée : *Un graphe non orienté (S, A).*
Sortie : *S'il est biparti.*

1. *Soient $S_1 \leftarrow \varnothing$ et $V \leftarrow \varnothing$.*
2. *Soit S_2 un ensemble contenant un sommet de chaque composante connexe.*
3. *Tant que $S_1 \cup S_2 \not\subset V$:*
4. *Si $S_1 \not\subset V$: $i \leftarrow 1$, $j \leftarrow 2$*
5. *Si $S_2 \not\subset V$: $i \leftarrow 2$, $j \leftarrow 1$*
6. *Enlever un sommet x de $S_i \setminus V$ et le rajouter à V.*
7. *Pour tout voisin y de x :*
8. *Si $y \in S_i$: renvoyer* `faux`.
9. *Rajouter y à S_j.*
10. *Renvoyer* `vrai`.

Avec des structures de données permettant de représenter efficacement des ensembles, à savoir, des tables de hachages ou des arbres binaires de recherche auto-équilibrants, la complexité en temps de cet algorithme est de $O(|S| + |A| \log |S|)$.

L'algorithme d'Edmonds consiste, partant de sommets non couplés, à effectuer un parcours en profondeur. Dans l'arborescence de ce parcours, il s'agit d'identifier un chemin alternant allant d'une racine à un autre sommet non couplé. On exige pour cela que les arêtes formant cette arborescence soient alternativement dans $\overline{C}$ et dans C. Un chemin augmentant est ainsi obtenu en arrivant sur un nœud de profondeur impair non couplé.

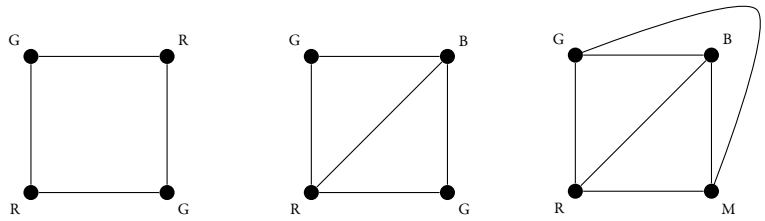


FIGURE 3.8 – Trois graphes munis chacun d'une coloration par un sous ensemble minimal de $\{R, G, B, M\}$; leurs nombres chromatiques sont respectivement 2, 3 et 4.

Algorithme (Edmonds [20]).

ENTRÉE : *Un couplage C non maximal d'un graphe biparti $(S_1 \sqcup S_2, A)$.*

SORTIE : *Un chemin augmentant.*

1. *Soit $W \leftarrow S_1 \setminus C$ l'ensemble des sommets à explorer.*
2. *Soit $V \leftarrow \varnothing$ l'ensemble des sommets visités.*
3. *Tant que $W \neq \varnothing$:*
4. *Enlever un sommet s de W et le rajouter à V.*
5. *Pour toute arête $(s, t) \notin C$ avec $t \notin V$:*
6. *Si t n'est pas couplé, on a trouvé un chemin augmentant.*
7. *Pour toute arête $(t, u) \in C$ avec $u \notin V$:*
8. *Rajouter t à V et rajouter u à W.*

Exercice. *Expliciter les structures de données pertinentes pour cet algorithme, en analyser la complexité puis le modifier afin de renvoyer explicitement le chemin augmentant obtenu.*

Compléments de culture générale. Considérons le problème consistant à colorier chaque sommet d'un graphe de sorte que deux extrémités d'une même arête ne soient jamais de la même couleur. Par exemple, pour la mappemonde représentée par le graphe (S, A) où $S = \{\text{pays}\}$ et $A = \{\text{couples de pays limitrophes}\}$, il s'agit de colorier chaque pays de sorte à pouvoir en distinguer les frontières.

Définition. *Soit $G = (S, A)$ un graphe non orienté et C un ensemble. On appelle coloration de G par C toute fonction $f : S \to C$ vérifiant $\forall (x, y) \in A, f(x) \neq f(y)$. On dit alors que $f(x)$ est la couleur du sommet x.*

Voir la figure 3.8 pour quelques exemples de colorations. Notons que, du point de vue de la théorie des graphes, l'ensemble des couleurs est arbitraire : seul son cardinal est pertinent.

Définition. *On appelle nombre chromatique d'un graphe non orienté G et l'on note $\mu(G)$ le plus petit entier n pour lequel il existe une coloration de G par un ensemble de cardinal n.*

Cette notion généralise celle de bipartition. En effet, à toute bipartition $S = S_1 \sqcup S_2$ correspond la coloration donnée par la fonction indicatrice de S_1 ; Inversement, à toute coloration $f : S \to \{a, b\}$ correspond la bipartition $S = f^{-1}(\{a\}) \sqcup f^{-1}(\{b\})$. Les graphes bipartis sont donc exactement ceux dont le nombre chromatique est inférieur ou égal à deux.

Définition. *Un graphe $G = (S, A)$ est dit planaire s'il existe une fonction injective $f : S \to \mathbb{R}^2$ telle que, pour tout couple d'arêtes (a, b) et (x, y) vérifiant $\{a, b\} \neq \{x, y\}$, l'intersection des segments $[f(a), f(b)]$ et $[f(x), f(y)]$ est vide ou réduite à l'éventuelle extrémité commune.*

Un graphe est donc planaire s'il peut être représenté dans le plan sans en croiser les arêtes.

Théorème (des quatres couleurs [24, 25, 34, 35]). *Le nombre chromatique d'un graphe planaire est inférieur ou égal à quatre.*

Ce célèbre théorème est l'aboutissement de longs travaux en théorie des graphes et fut historiquement le premier résultat démontré à l'aide d'un ordinateur. Sa preuve nous est hors de portée mais on pourra le démontrer dans les cas particuliers des graphes ci-après.

Définition. *Soient ℓ et m deux entiers positifs. Le graphe complet K_ℓ a pour ensemble de sommets $S = \{1, 2, \dots, \ell\}$ et pour ensemble d'arêtes $A = S^2 \setminus \{(x, x) : x \in S\}$. Le graphe biparti complet $K_{\ell,m}$ a pour ensemble de sommets $S = S_1 \sqcup S_2$ avec $S_1 = \{1, \dots, \ell\}$ et $S_2 = \{\ell + 1, \dots, \ell + m\}$ et pour ensemble d'arêtes $A = S_1 \times S_2 \cup S_2 \times S_1$.*

Proposition. *Le graphe K_ℓ est planaire si et seulement si $\ell \leqslant 4$. Le graphe $K_{\ell,m}$ est planaire si et seulement si $\ell \leqslant 2$ ou $m \leqslant 2$.*

3.4 Bases de données

Ce semestre se conclura par deux sections correspondant aux parties les plus fantaisistes du programme. Avant cela, il nous faut traiter de notions moins exotiques mais en lien très direct avec les épreuves des concours.

3.4.1 Problématique et langage SQL

Les bases de données sont des structures de données permettant de stocker de manière organisée de grands volumes d'information. Elles fournissent trois opérations : la création d'une base (en spécifiant l'organisation des données qu'elle sera amenée à stocker), l'insertion de données dans une base et l'accès à ces données ; les principaux enjeux étant :

- **Fiabilité :** On accède bien aux données qui ont été insérées.
- **Performance :** Les insertions et les accès s'exécutent rapidement.
- **Concurrence :** On peut effectuer des insertions et des accès simultanément.
- **Fonctionnalité :** En ce qui nous concerne (et ce sera le seul enjeu abordé dans ce cours), il s'agit de pouvoir effectuer des recherches dans la base.

Ces bases se déclinent en une multitude d'implantations, chacune ciblant des applications spécifiques. Parmi les plus célèbres citons SQLite qui prend la forme d'une bibliothèque portable dont font usage notamment Chrome, Firefox et Safari, ainsi que MariaDB qui a une architecture client/server performante utilisée notamment par Google et Wikipedia.

L'objectif est ici d'apprendre les rudiments du *Structured Query Language* (SQL) conçu pour interagir avec les bases de données dont l'organisation suit le modèle relationnel [22].

Définition. *Une base de données suit le modèle relationnel lorsque ses données sont organisées sous forme de **relations**. Chaque relation est représentée par une table dont les colonnes représentent des **attributs** et les lignes des **tuples**. Différentes relations interagissent par la présence d'attributs communs. Le **domaine** d'un attribut n'est autre que son type. Le **schéma** d'une relation est l'ensemble des couples $(nom, domaine)$ pour chaque attribut. On appelle **clef primaire** d'une relation tout attribut dont la valeur permet d'identifier de manière unique chaque tuple.*

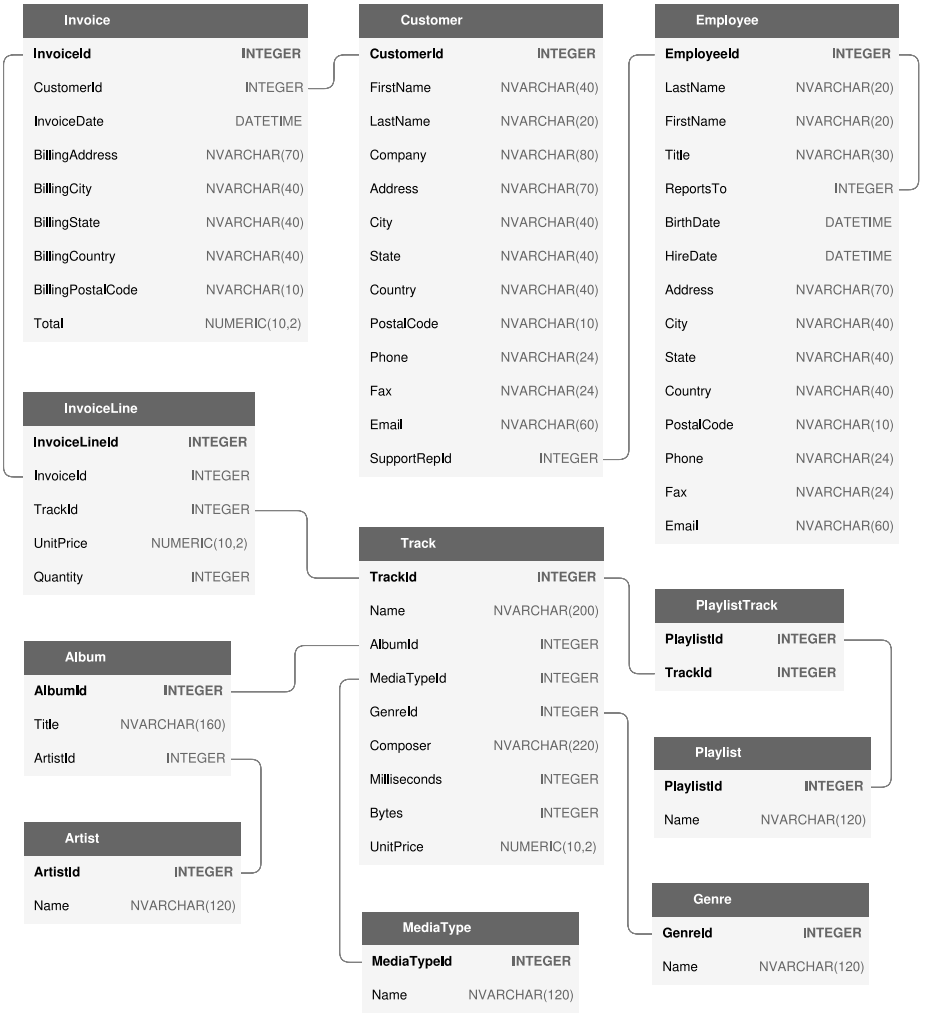


FIGURE 3.9 – Relations de la base de données « Chinook » [36].

Comme c'est l'usage courant, on utilisera de manière interchangeable les termes tables et relations, colonnes et attributs, lignes et tuples. La convention veut aussi que les tables comportent presque systématiquement une clef primaire, quitte à ce que ce soit la seule utilité de l'attribut correspondante.

Exemple. *Considérons la base de données d'un disquaire fictif dont les onze relations sont illustrées par la figure 3.9. La relation « Track » comporte notamment l'attribut « TrackId » de type entier qui joue le rôle de clef primaire, l'attribut « Name » de type chaîne de caractères, etc.*

Charger cette base de données dans une feuille de calcul de type `Sqlite3` en exécutant :

```
.open /home/bisson/chinook.db
```

3.4.2 Sélection et projection

Seule au programme, la commande `SELECT` permet en langage SQL d'effectuer des requêtes, c'est-à-dire d'extraire des données d'une base. Sa syntaxe minimaliste est « `SELECT attributs FROM tables` », les attributs pouvant être spécifiées par leur nom « `Nom` » ou de manière plus complète sous la forme « `Table.Nom` » ; on peut utiliser l'astérisque pour dénoter tous les attributs, par exemple « `Table.*` » ou encore « `*` ».

Remarque. *L'usage veut que les mots clefs SQL s'écrivent entièrement en majuscules.*

Afin d'obtenir les noms et prénoms des clients, on écrit :

```
SELECT LastName,FirstName FROM Customer;
```

On peut compléter cette requête d'une clause « `WHERE condition` » afin de sélectionner uniquement les tuples satisfaisant la condition donnée. Pour obtenir les noms et prénoms des clients résidant en France on écrit :

```
SELECT LastName,FirstName FROM Customer WHERE Country="France";
```

Exercice. *Afficher les noms et prénoms des employés résidant à Calgary.*

Les requêtes SQL peuvent être raffinées par les mots clefs suivants :
- `DISTINCT` se préfixe à un attribut afin d'en éliminer les doublons :
- `ORDER BY` se suffixe à la requête afin d'ordonner les résultats suivant un attribut donné ; on peut préciser l'ordre ascendant `ASC` (par défaut) ou descendant `DESC`.
- `LIMIT` et `OFFSET` se suffixent à la requête afin de n'en afficher qu'un extrait : au plus `limit` tuples à compter de `offset`.

Par exemple :

```
SELECT DISTINCT Country
FROM Customer
ORDER BY Country ASC
LIMIT 5 OFFSET 0;
```

3.4.3 Opérations ensemblistes

Assimilons une relation à l'ensemble de ses tuples. Deux relations de même schéma peuvent être combinées par union, intersection et différence, notées `UNION`, `INTERSECT` et `EXCEPT` en langage SQL. On écrirait par exemple « `SELECT * FROM Table1 UNION Table2` ». Ces opérations sont toutefois rarement utilisées ; d'ailleurs, aucun couple de relations ne s'y prête dans notre base de données exemple. La construction que voici est bien plus courante.

Définition. *Soient R et S deux relations de schémas respectifs $(a_1, \dots, a_k)$ et $(b_1, \dots, b_\ell)$. Leur produit cartésien est la relation $R \times S$ de schéma $(a_1, \dots, a_k, b_1, \dots, b_\ell)$ vérifiant*

$$R \times S = \{(r_1, \dots, r_k, s_1, \dots, s_\ell) : (r_1, \dots, r_k) \in R, (s_1, \dots, s_\ell) \in S\}.$$

Le produit cartésien des relations `R` et `S` se note « `R,S` » en langage SQL. Remarquons que cette version diffère subtilement du produit cartésien ensembliste par son associativité ; en effet, en algèbre relationnel, on a $(r, (s, t)) = (r, s, t)$.

Exemple. *Pour savoir quel employé suit quel client, on écrit :*

```
SELECT Customer.LastName,Employee.LastName
FROM Customer,Employee
WHERE Customer.SupportRepId=Employee.EmployeeId;
```

Exemple. *Pour afficher les noms et prénoms des clients suivis par Jane, on peut écrire :*

```
SELECT Customer.LastName,Customer.FirstName
FROM Customer,Employee
WHERE Customer.SupportRepId=Employee.EmployeeId
AND Employee.FirstName="Jane";
```

Exercice. *Extraire les noms et prénoms des clients ayant un facture datée de 2013.*

```
SELECT DISTINCT Customer.LastName,Customer.FirstName
FROM Customer,Invoice
WHERE Invoice.CustomerId=Invoice.CustomerId
AND Invoice.InvoiceDate>='2013-01-01'
AND Invoice.InvoiceDate<'2014-01-01';
```

3.4.4 Jointure

La mise en relation de deux tables par produit cartésien suivi d'une sélection est inefficace : cela génère tous les couples de tuples quand bien même seule une fraction sera ensuite sélectionnée. Le mécanisme de jointure est une alternative plus efficace énumérant uniquement les couples pour lesquels des attributs spécifiés coïncident ; sa complexité est ainsi quasi linéaire plutôt que quadratique.

Définition. *Soit R et S deux relations. La jointure $R \bowtie S$ est la relation ayant pour tuples les couples (r, s) lorsque r et s parcours les tuples de R et S dont les attributs communs coïncident.*

En SQL cette opération s'écrit « `Table1 JOIN Table2 ON Attribut1=Attribut2` ».

Exemple. *Pour afficher les noms et prénoms des clients suivis par Jane, on peut écrire :*

```
SELECT Customer.LastName,Customer.FirstName
FROM Customer JOIN Employee
ON Customer.SupportRepId=Employee.EmployeeId
WHERE Employee.FirstName="Jane";
```

Exercice. *Extraire les noms et prénoms des clients ayant un facture datée de 2013.*

3.4.5 Fonctions d'agrégats et sous requêtes

On peut davantage raffiner les requêtes SQL par les mécanismes que voici.

Projections étendues. Cela consiste à renvoyer non pas directement les valeurs des attributs sélectionnés mais le résultat d'expressions arithmétiques évaluées en ces valeurs. On dispose notamment des opérateurs `+`, `-`, `*`, `/` et des constantes numériques. On peut ainsi écrire :

```
SELECT UnitPrice*Quantity
FROM InvoiceLine;
```

Renommage. Cette clause définit un nouvel attribut qui peut alors être utilisé. Sa syntaxe est « `Attribut AS NouveauNom` ». On écrira notamment :

```
SELECT UnitPrice*Quantity AS PrixHT
FROM InvoiceLine
ORDER BY PrixHT DESC
LIMIT 5;
```

Exercice. *Modifier cette requête pour qu'elle renvoie non seulement le montant des ventes mais aussi le nom de chaque piste concernée.*

Agrégats. Les tuples de chaque relation sont partitionnés en paquets appelés agrégats. Par défaut, tous sont regroupés dans un unique agrégat ; nous en étions jusqu'alors inconscient car nous n'utilisions aucune fonction d'agrégats. La clause « `GROUP BY expression` » crée des agrégats regroupant les tuples pour lesquels la valeur de l'expression est identique.

Fonctions d'agrégats. Les fonctions d'agrégat, notées `AGR(expr)`, sont évaluées sur chaque agrégat ; elles incluent notamment `COUNT`, `MIN`, `MAX`, `SUM` et `AVG` qui calculent le nombre de valeurs non nulles, le minimum, le maximum, la somme et la moyenne. Plutôt que `COUNT(expr)` on préfèrera généralement utiliser `COUNT(*)` qui renvoie simplement le nombre de tuples de l'agrégat. Pour obtenir le montant de la facture la plus élevée, on écrit :

```
SELECT MAX(Total) FROM Invoice;
```

Et pour obtenir le montant de la facture la plus élevée pour chaque client, on écrit :

```
SELECT MAX(Total) FROM Invoice GROUP BY CustomerId;
```

Et si l'on souhaite afficher les noms et prénoms des clients on écrit :

```
SELECT MAX(Total),LastName,FirstName
FROM Invoice JOIN Customer ON Invoice.CustomerId=Customer.CustomerId
GROUP BY Invoice.CustomerId;
```

La clause « `HAVING condition` » est similaire à la clause `WHERE` sauf qu'elle opère sur les agrégats plutôt que les tuples. Pour identifier les employés en charge de vingt clients ou plus :

```
SELECT Employee.LastName
FROM Employee JOIN Customer ON SupportRepId=EmployeeId
GROUP BY EmployeeId
HAVING COUNT(*)>=20;
```

Exercice. *Identifier les clients ayant dépensé quarante dollars ou plus.*

```
SELECT LastName,FirstName
FROM Customer JOIN Invoice ON Invoice.CustomerId=Customer.CustomerId
GROUP BY Customer.CustomerId
HAVING SUM(Total)>=40;
```

Sous requêtes. Le résultat d'une sous requête SQL peut être substitué dans une autre requête via l'usage de parenthèses. Par exemple, afin d'identifier la composition de l'album ayant le plus de pistes :

```
SELECT Name
FROM Track
WHERE AlbumId=(
    SELECT AlbumId
    FROM Track
    GROUP BY AlbumId
    ORDER BY COUNT(*) DESC
    LIMIT 1
);
```

Ce qui peut aussi s'effectuer de manière équivalente :

```
SELECT Name
FROM Track JOIN (
    SELECT AlbumId AS id
    FROM Track
    GROUP BY AlbumId
    ORDER BY COUNT(*) DESC
    LIMIT 1
) ON id=AlbumId;
```

Exercice. *Déterminer les quantités suivantes :*

- *le nombre de clients ;*
- *le chiffre d'affaire ;*
- *l'artiste ayant le plus d'albums ;*
- *l'artiste ayant le plus de chansons ;*
- *l'artiste ayant vendu le plus de chansons ;*
- *l'artiste ayant rapporté le plus au commerce ;*
- *le plus gros client du commerce dans chacun des genres.*

3.5 Déduction naturelle pour la logique propositionnelle

Le calcul propositionnel a été abordé l'an dernier par une approche directe mais relativement inefficace. Par exemple, l'algorithme que nous avons conçu pour déterminer la satisfiabilité des propositions logiques était de complexité exponentielle. On ne peut espérer faire mieux dans le cas général puisque ce problème est conjecturé ne pas admettre de solution polynômiale. Nous allons donc cibler un contexte plus précis.

La déduction naturelle est un système permettant de formaliser la construction de preuves mathématiques. Il consiste à manipuler des objets appelés séquents que l'on note

$$p, q, r \vdash s, t$$

et interprète intuitivement comme signifiant que les hypothèses p, q et r entraînent au moins l'une des conclusions s et t ; autrement dit, le séquent ci-dessus représente la véracité de la proposition « $p \wedge q \wedge r \Rightarrow s \vee t$ ».

Des règles de calcul permettent de déduire (on dit parfois aussi dériver) de nouveaux séquents à partir de séquents déjà démontrés. Ces règles se notent

$$\frac{s \quad t}{u}$$

signifiant que, de démonstrations des séquents s et t, on déduit une démonstration du séquent u ; la ligne horizontale est dite d'inférence ou de déduction. Ces règles de calcul, garantes de la correction de la preuve, reposent sur des identités logiques. Voir la figure 3.10 qui énumère les règles d'un système couramment utilisé.

Remarque. *Nous adoptons ici la notation standard en logique, le symbole « $\rightarrow$ » désignant l'implication communément notée « $\Rightarrow$ » et le symbole « $\bot$ » désignant la proposition fausse communément notée « F ».*

Dans ce formalisme, chaque démonstration consiste, partant de séquents admis, à appliquer des règles de calcul successives afin d'obtenir un séquent de la forme « $\vdash s$ » où s est la proposition qu'il s'agissait de démontrer, prouvant ainsi sa véracité inconditionnelle. On construit ainsi un arbre ayant pour feuilles les hypothèses, dont chaque branche correspond à l'application d'une règle de calcul et dont la racine est ce qu'il fallait démontrer.

RÈGLES D'IDENTITÉ

$$\frac{}{A \vdash A} \qquad \frac{A \vdash p, B \quad p, C \vdash D}{A, C \vdash B, D}$$

RÈGLES STRUCTURELLES

$$\frac{A \vdash B}{p, A \vdash B} \qquad \frac{A \vdash B}{A \vdash p, B}$$

$$\frac{p, p, A \vdash B}{p, A \vdash B} \qquad \frac{A \vdash p, p, B}{A \vdash p, B}$$

$$\frac{A, p, q, B \vdash C}{A, q, p, B \vdash C} \qquad \frac{A \vdash B, p, q, C}{A \vdash B, q, p, C}$$

RÈGLES LOGIQUES

$$\frac{A \vdash p, B}{A, \neg p \vdash B} \qquad \frac{p, A \vdash B}{A \vdash \neg p, B}$$

$$\frac{p, q, A \vdash B}{p \wedge q, A \vdash B} \qquad \frac{A \vdash p, B \quad A \vdash q, B}{A \vdash p \wedge q, B}$$

$$\frac{p, A \vdash B \quad q, A \vdash B}{p \vee q, A \vdash B} \qquad \frac{A \vdash p, q, B}{A \vdash p \vee q, B}$$

$$\frac{A \vdash p, B \quad q, A \vdash B}{p \to q, A \vdash B} \qquad \frac{p, A \vdash q, B}{A \vdash p \to q, B}$$

FIGURE 3.10 – Règles de calcul des séquents du système LK [3] formalisant la logique du premier ordre, restreintes ici à la logique propositionnelle. Les lettres latines minuscules et majuscules dénotent respectivement des propositions et des familles de propositions arbitraires.

Exemple. *Dans ce formalisme, le* modus ponens *« $p, p \to q \vdash q$ » s'obtient comme il suit, sans aucune hypothèse.*

$$\dfrac{\dfrac{\overline{p \vdash p}}{p \vdash p, q} \quad \dfrac{\overline{q \vdash q}}{p, q \vdash q}}{p, p \to q \vdash q}$$

Exercice. *Dans ce formalise, démontrer le syllogisme Barbara « $p \to q, q \to r \vdash p \to r$ » puis le principe de contraposition « $p \to q \vdash \neg q \to \neg p$ ».*

Dans le cas particulier du calcul propositionnel, le calcul des séquents permet de démontrer la correction des règles bien connues et notamment celles d'introduction et d'élimination des opérateurs logiques énumérées ci-après.

$$\dfrac{p \quad \neg p}{\bot} \qquad \dfrac{\bot}{p}$$

$$\dfrac{p \quad q}{p \wedge q} \qquad \dfrac{p \wedge q}{p}$$

$$\dfrac{p}{p \vee q} \qquad \dfrac{p \vee q \quad p \to r \quad q \to r}{r}$$

Exercice. *Définir un type OCaml représentant les séquents ; implanter quelques règles de la figure 3.10.*

En implantant ce formalisme et ses règles de calcul, on pourrait concevoir un programme qui, partant des axiomes, explorerait automatiquement le graphe de toutes les démonstrations possibles ; on énumèrerait ainsi les énoncés démontrables. Cela serait toutefois peu satisfaisant à deux titres.

Premièrement, la taille du graphe étant exponentielle, la complexité du programme le serait également. Cependant, si l'on ne souhaite qu'obtenir une preuve rigoureuse d'une proposition donnée, on peut guider l'exploration du graphe en spécifiant des sommets intermédiaires correspondant à certaines étapes clefs du raisonnement mathématique sous-jacent. Ceci limite la recherche exhaustive aux chemins d'un sommet intermédiaire à un autre et la complexité s'en retrouve amoindrie. Cette approche, alliant l'ingénuité de l'homme avec la rigueur de la machine, est au cœur des assistants de preuve.

Deuxièmement, le théorème ci-dessous révèle que le vrai et le démontrable sont deux notions bien distinctes.

Théorème (Gödel [2]). *Tout système formel (axiomes et règles de calcul) cohérent dans lequel on peut démontrer l'arithmétique de Peano admet des propositions dont ni la véracité ni la fausseté ne sont démontrables.*

3.6 Algorithmique pour l'intelligence artificielle et l'étude des jeux

Remarque. *Cette partie du programme est invraisemblablement extravagante. Votre enseignant fera néanmoins de son mieux pour en présenter les notions de manière sobre et pragmatique.*

3.6.1 Intelligence artificielle

L'intelligence artificielle rassemble les techniques informatiques visant à imiter les processus cognitifs humains. Parmi celles-ci, les réseaux neuronaux sont des programmes informatiques qui simulent le fonctionnement des structures biologiques éponymes. Ils ont bouleversé les domaines de la vision par ordinateur, de l'art génératif, du jeu de go, etc.

L'enjeu majeur est de construire des réseaux neuronaux permettant de résoudre un problème donné. Pour cela, on a généralement recours à l'apprentissage automatique (*machine learning*) : on améliore des réseaux neuronaux par itérations successives en les modifiant aléatoirement et en sélectionnant ceux qui présentent les meilleures performances sur un jeu de données représentatif. On construit ainsi des programmes informatiques de manière probabiliste qui, bien que d'efficacité parfois impressionnante, sont inscrutables.

Notons E l'espace des paramètres du problème considéré et F celui de ses solutions et soit $f : E \to F$ la fonction optimale qu'il s'agit d'approcher ; elle associerait par exemple à chaque image un mot la décrivant. Notons enfin $T = (x_i)_{i\in\{1,\dots,n\}} \in E^n$ le jeu de données choisi. On distingue deux approches :

— **l'apprentissage supervisé,** où l'on dispose des solutions $f(x_i)$ des données du jeu, tel en reconnaissance vocale où la transcription des enregistrements audio est connue ;
— **l'apprentissage non supervisé,** où l'on ignore les solutions $f(x_i)$ et dispose uniquement d'une fonction évaluant la qualité d'une approximation, par exemple dans le cas du jeu de go où l'on ne connaît pas nécessairement le meilleur coup à jouer.

Nous allons maintenant présenter quelques outils élémentaires d'apprentissage. On se place dans $E = \mathbb{R}^d$ muni de sa structure euclidienne et on suppose que l'ensemble F est fini, situation que l'on qualifie parfois de « problème de classification ».

Méthode des k plus proches voisins. En apprentissage supervisé, on connaît $f(x_i)$ pour chaque donnée x_i du jeu. Étant donné $x \in E$, on peut adopter comme approximation de $f(x)$ la valeur $f(x_i)$ prépondérante pour les k vecteurs x_i les plus proches de x. L'entier k est fixé au préalable notamment en fonction de n et $\#F$.

Exercice. *Écrire une fonction Python qui, étant donnés un entier k, une famille de vecteurs $T \in \left(\mathbb{R}^d\right)^n$ et un vecteur $x \in \mathbb{R}^d$, renvoie les indices des k vecteurs de T les plus proches de x.*

```
def plusproches(k, T, x):
    inf = float("inf")
    D = k * [inf]
    I = k * [-1]
    for i in range(len(T)):
        d = distance(x, T[i])
        j = k
        while j > 0 and d < D[j - 1]:
            j = j - 1
        if j < k:
            D = D[0:j] + [d] + D[j:-1]
            I = I[0:j] + [i] + I[j:-1]
    return I
```

Exercice. *L'algorithme ci-dessus est de complexité $O(n\,d\,k)$; discuter des améliorations qui pourraient y être apportées pour abaisser cette grandeur à $O(n\,d\log k)$, notamment en matière de structures de données.*

Méthodes des k moyennes. En apprentissage non supervisé, une première étape consiste à déterminer une approximation $\widehat{f}(x_i)$ de $f(x_i)$ pour tout $x_i \in T$. Cela revient à partitionner T en $k = \#F$ parties $c(y) = \widehat{f}|_T^{-1}(y)$. Supposant qu'on adoptera ensuite la méthode des plus proches voisins pour approcher $f(x)$, il est pertinent de minimiser la distance entre vecteurs au sein de chaque partie. Autrement dit, on souhaite « détecter » k amas de vecteurs dans $T \subset \mathbb{R}^d$ c'est-à-dire qu'on souhaite minimiser la quantité

$$\sum_{y \in F} \sum_{x \in c(y)} \|x - \mu(y)\| \qquad \text{où} \qquad \mu(y) = \frac{1}{\#c(y)} \sum_{x \in c(y)} x$$

désigne la moyenne des vecteurs de la partie $c(y)$ d'image y. C'est l'objectif de la méthode heuristique ci-après.

Algorithme (méthode des k moyennes)**.**

Entrée : *Un entier k et une partie finie T de $\mathbb{R}^d$.*

Sortie : *Une k-partition de T.*

1. *Choisir k points $\mu_1, \dots, \mu_k$ de $\mathbb{R}^d$ aléatoirement.*
2. *Répéter autant de fois qu'utile :*
3. *Pour tout $i \in \{1, \dots, k\}$: affecter $c_i \leftarrow \{x \in T : i = \operatorname{argmin}_j \|x - \mu_j\|\}$.*
4. *Pour tout $i \in \{1, \dots, k\}$: affecter $\mu_i \leftarrow \left(\sum_{x \in c_i} x\right)/\#c_i$.*
5. *Renvoyer la partition correspondante.*

Exercice. *Écrire une fonction Python implantant cette méthode.*

Matrice de confusion. Les constructions qui précèdent fournissent des approximations $\widehat{f}$ à la solution f. Afin d'évaluer leur qualité sur un ensemble test $T \subset \mathbb{R}^d$ fini, on fixe une énumération $F = \{y_1, \dots, y_r\}$ et on définit la matrice de confusion

$$\left(\#\left\{ x \in T' : f(x) = y_i \wedge \widehat{f}(x) = y_j \right\} \right)_{(i,j) \in \{1,\dots,r\}^2}$$

L'approximation $\widehat{f}$ est d'autant bonne qualité que les coefficients non diagonaux de cette matrice sont négligeables.

3.6.2 Étude des jeux

Considérons deux joueurs qui s'affrontent à un jeu de plateau en en modifiant l'état alternativement. Les règles du jeu dictent les modifications possibles et certains états sont gagnants pour le premier joueur et d'autres pour le second. On pensera notamment au jeu d'échecs, au jeu de go ou encore aux jeux de Nim. Modélisons cette situation à l'aide de graphes.

Définition. *On appelle jeu la donnée de :*

— *un graphe orienté $G = (S, A)$ muni d'une bipartition $S = S_1 \sqcup S_2$, c'est-à-dire vérifiant $A \subset (S_1 \times S_2) \cup (S_2 \times S_1)$;*

— *deux sous-ensembles disjoints F_1 et F_2 de l'ensemble F des sommets de degré sortant nul.*

On appelle partie de ce jeu tout chemin menant à un sommet de F. Cette partie est dite gagnante pour le joueur $i \in \{1, 2\}$ lorsque son sommet d'arrivée est dans F_i.

Intuitivement, le premier joueur contrôle S_1, le second S_2 et ils jouent alternativement : le premier joueur choisi une arête (x_1, x_2), le second une arête (x_2, x_3), le premier une arête (x_3, x_4), etc. Chacun tentant de diriger le chemin vers un sommet gagnant pour lui.

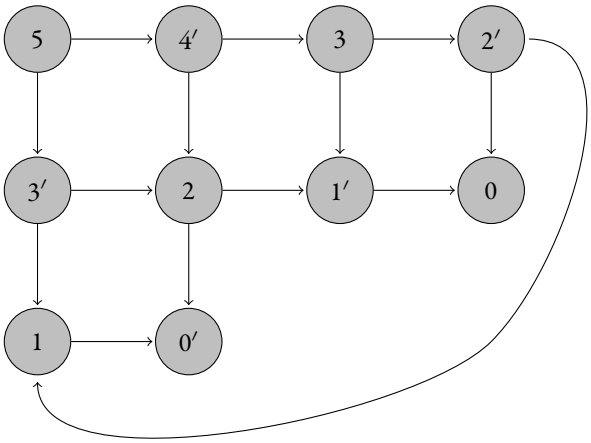


FIGURE 3.11 – Graphe modélisant un jeu de Nim à cinq bâtons, chaque joueur devant en retirer un ou deux à son tour. L'étiquette de chaque sommet indique le nombre de bâtons restants. On a $S_1 = \{5, 3, 2, 1, 0\}$, $S_2 = \{4', 3', 2', 1', 0'\}$, $F_1 = \{0\}$ et $F_2 = \{0'\}$.

Exemple. *La figure 3.11 modélise un jeu de Nim avec quatre bâtons, chaque joueur devant en retirer un ou deux à chaque tour.*

On se placera désormais implicitement du point de vue du premier joueur. On définira donc les stratégies gagnantes, les positions gagnantes, sans autre précision ; des notions similaires pourraient évidemment être définies du point de vue du second joueur.

Définition. *On appelle stratégie sans mémoire toute fonction $f : S_1 \to S_2$ vérifiant $\forall x \in S_1, (x, f(x)) \in A$. Cette stratégie est dite gagnante depuis x_1 si tous les chemins de la forme*

$$x_1 \to f(x_1) \to x_3 \to f(x_3) \to x_5 \to f(x_5) \to \cdots$$

sont gagnants (pour le premier joueur) quels que soient les sommets $x_3, x_5, \ldots$ Le sommet x_1 est alors qualifié de position gagnante.

Remarquons que la notion de position gagnante suppose que le premier joueur joue parfaitement mais ne fait aucune hypothèse sur le second joueur : il est quant à lui libre de jouer parfaitement ou non.

Par exemple, sur le jeu de la figure 3.11, les positions gagnantes sont : l'état 0, l'état 2 (stratégie $2 \mapsto 1'$ gagnante), l'état 3 (stratégie $3 \mapsto 1'$ gagnante) et l'état 5 (stratégie $5 \mapsto 4'$ gagnante).

Exercice. *On considère un jeu de Nim à n bâtons, chaque joueur devant à chaque tour en retirer au moins un et au plus k. Déterminer l'ensemble des positions gagnantes.*

Afin d'énumérer les positions gagnantes, on introduit la notion suivante.

Définition. *Soit (S, A) un graphe muni d'une bipartition $S = S_1 \sqcup S_2$. Pour toute partie $E_0 \subset S$ on considère la suite croissante définie par*

$$\begin{aligned} E_{k+1} = E_k &\cup \{x \in S_1 : \exists (x, y) \in A, y \in E_k\} \\ &\cup \{x \in S_2 : \forall (x, y) \in A, y \in E_k\} ; \end{aligned}$$

l'ensemble $\overline{E} = \bigcup_{k \in \mathbb{N}} E_k$ est dit attracteur de E_0.

Proposition. *L'attracteur de F_1 est l'ensemble des positions gagnants.*

Exercice. *Concevoir un algorithme calculant l'attracteur à partir de la matrice d'adjacence ; quelle est sa complexité ? L'implanter en langage Python puis OCaml.*

Comme l'étude de complexité ci-dessus le révèle, il n'est faisable de calculer explicitement l'attracteur que lorsque le graphe est particulièrement petit. Une approche heuristique alternative consiste à se munir d'une fonction $e : S \to [0, 1]$ estimant la « probabilité » que la position soit gagnante. Pour approcher une stratégie gagnante, on explorera une partie du graphe guidé par cette fonction, sachant que le premier joueur chercher à la maximiser et son adversaire à la minimiser.

Cela donne naissance à la famille des algorithmes minimax, parfois aussi appelés min-max, qui consistent à raffiner l'estimation par exploration successive comme ceci.

Algorithme (minimax)**.**

Entrée : *Un graphe orienté $G = (S, A)$ muni d'une bipartition $S = S_1 \sqcup S_2$, une fonction d'estimation du gain $e : S \to [0, 1]$ et un sommet x.*

Sortie : *Un raffinement de cette estimation.*

1. *Choisir un sous arbre H de G enraciné en x.*
2. *Pour chaque nœud $y \in H$ par profondeur décroissante :*
3. *Si $y \in S_1$:*
4. $e(y) \leftarrow \max\{e(z) : z \in \mathit{fils}(y)\}$
5. *Si $y \in S_2$:*
6. $e(y) \leftarrow \min\{e(z) : z \in \mathit{fils}(y)\}$

Toute la difficulté réside désormais dans le choix initial de cette fameuse fonction d'estimation ainsi que dans celui de l'arbre à explorer : taille, hauteur, choix des fils, etc.

Exercice. *Mettre cette approche en œuvre pour le jeu du morpion.*

Chapitre 4

Quatrième semestre

4.1 Motifs, automates et expressions

Cette dernière section attaque un problème classique, celui de la reconnaissance de motifs, par trois angles distincts : les techniques *ad hoc*, la théorie des langages rationnels puis enfin la théorie des automates.

4.1.1 Motifs

Au sens très large, un motif désigne une forme à rechercher automatiquement dans un espace de données. Il peut par exemple s'agir de reconnaître des gènes dans un génome, la parole dans un enregistrement sonore ou encore des codes-barres, des textes manuscrits voire des personnes sur une image. Sous toutes ses formes, le problème de la reconnaissance de motifs est omniprésent dans notre monde numérique.

Ce problème ne sera ici abordé que dans un cadre strictement délimité : d'une part, on se restreindra aux espaces de dimension un ; d'autre part, on ne considèrera que des motifs exacts, c'est-à-dire dont la présence ou absence peut être déterminée incontestablement, ce qui élimine notamment tous les exemples ci-dessus. On adopte pour cela le formalisme suivant.

Définition. *On appelle alphabet tout ensemble fini Σ ni vide ni singleton. On appelle lettre tout élément de Σ. On appelle mot sur Σ toute famille de lettres ; on note $|m|$ la longueur du mot m. Le mot vide noté ε est l'unique mot de longueur nulle. On appelle langage ou motif sur Σ tout ensemble de mots sur Σ. Le monoïde libre sur Σ noté $\Sigma^\star$ est l'ensemble de tous les mots sur Σ.*

L'alphabet est généralement fixé en début de problème et vaut typiquement $\{0, 1\}$ ou $\{a, b, c, \dots, z\}$. Un mot est noté tout simplement en juxtaposant ses éléments. Le langage des mots de longueur trois sur l'alphabet $\{0, 1\}$ s'écrit donc

$$\{000, 001, 011, 010, 110, 111, 101, 100\}.$$

On munit $\Sigma^\star$ de la structure que voici.

Définition. *On appelle concaténation de deux mots $x = x_1 \cdots x_n$ et $y = y_1 \cdots y_m$ sur un même alphabet le mot $xy = x \cdot y = x_1 \cdots x_n y_1 \cdots y_m$.*

Proposition. *La concaténation vérifie $|xy| = |x| + |y|$.*

La concaténation est une loi de composition sur l'ensemble $\Sigma^\star$. Elle est associative et admet le mot vide comme élément neutre ; on dit ainsi que $(\Sigma^\star, \cdot)$ est un monoïde, soit plus familièrement un groupe sans inverse. Attention toutefois à la non commutativité de cette opération : les mots ab et ba sont bien distincts.

Définition. *On dit qu'un mot x est un préfixe (resp. suffixe) d'un mot m s'il existe un mot u tel que $m = xu$ (resp. $m = ux$). On dit qu'un mot x est un facteur d'un mot m s'ils existent des mots u et v tels que $m = uxv$.*

Proposition. *Si x est un facteur de y alors $|x| \leqslant |y|$.*

Tout préfixe ou suffixe d'un mot en est un facteur. Plus généralement, les facteurs sont exactement les suffixes des préfixes ou, de manière équivalente, les préfixes des suffixes.

Remarque. *Les facteurs sont parfois aussi appelés sous mots ou sous chaînes mais certains petits malins donnent à ces termes un sens complètement distinct (et inutile) calqué sur la notion de sous suite ; on évitera donc ces appellations.*

Exercice. *Montrer que, sur l'ensemble $\Sigma^\star$, la relation $x \mathrel{\mathscr{R}} y \iff$ « x est un facteur de y » est un ordre bien fondé.*

Exercice. *Soient x et y deux mots vérifiant $xy = yx$. Montrer qu'il existe un mot z et deux entiers naturels k et ℓ vérifiant $x = z^k$ et $y = z^\ell$. On procèdera par induction.*

Abordons le problème de la reconnaissance de motifs dans un cas élémentaire : celui des langages formés de tous les mots admettant un facteur x donné. Reconnaître le motif $\Sigma^\star x \Sigma^\star$, c'est-à-dire déterminer si un mot donné y appartient à ce langage, revient ainsi à déterminer si x est un facteur de y.

Exercice. *Écrire une fonction OCaml prenant en argument deux mots et déterminant si le premier est un facteur du second. À votre convenance les mots seront représenté par des listes ou par des chaînes de caractères. Démontrer sa terminaison et sa correction ; borner sa complexité.*

```
let rec facteur x =
    let rec prefixe = function
        | [], _      -> true
        | h :: t, [] -> false
        | h :: t, i :: u -> h = i && prefixe (t, u)
    in
    function
    | []     -> x = []
    | h :: t -> prefixe (x, h :: t) || facteur x t
;;

let facteur x y =
    let m = String.length x in
    let n = String.length y in
    let r = ref false in
    let i = ref 0 in
    while (not !r) && !i + m <= n do
        let j = ref 0 in
        while !j < m && x.[!j] = y.[!i + !j] do incr j done;
        if !j = m then r := true;
        incr i
    done;
    !r
;;
```

La complexité de cette approche naïve est évidemment $O(mn)$ où m et n dénotent les longueurs respectives du motif et du texte. Dans le cas moyen, elle dépend entièrement du nombre d'itérations qu'effectue la boucle `while`. Si l'on suppose chaque lettre uniformément distribuée dans l'alphabet alors avec grande probabilité cette boucle est courte et la complexité dans le cas moyen est $O(m+n)$. Cette hypothèse est souvent justifiée et on retiendra donc qu'à l'instar de certaines méthodes de tri non optimales l'algorithme ci-dessus peut être un choix judicieux en pratique.

Remarque. *Les algorithmes de Knuth–Morris–Pratt [27] et de Boyer–Moore [26] atteignent la complexité optimale de $O(m+n)$ dans le cas le pire. Toutefois, la théorie que nous allons maintenant développer fourni une solution alternative bien plus satisfaisante d'un point de vue théorique.*

4.1.2 Langages rationnels

On ne peut espérer inculquer à une machine la reconnaissance de langages arbitraires ; en effet, s'agissant de parties du monoïde libre $\Sigma^\star$, il existe une infinité non dénombrable de langages distincts. C'est pourquoi on commence cette section en introduisant un formalisme permettant de représenter sous forme finie une classe importante de motifs infinis.

Définition. *Soient L et M deux langages. On appelle opérations rationnelles :*
- *la somme $L+M=L\cup M$;*
- *le produit $L\cdot M=\{x\cdot y : x\in L, y\in M\}$;*
- *l'étoile $L^\star=\bigcup_{k\in\mathbb{N}} L^k$ où l'on a posé $L^0=\{\varepsilon\}$ et $L^{k+1}=L^k\cdot L$.*

La somme $L+M$ de deux langages et aussi parfois appelée union et notée $L|M$. Remarquons notamment la cohérence des notations $\Sigma^\star$ et $L^\star$ dans le sens où, en identifiant les lettres avec les mots de longueur unité, la seconde généralise la première.

Exemple. *L'étoile du langage vide est $\varnothing^\star=\{\varepsilon\}$. Si σ désigne une lettre, le langage $\{\sigma\}^\star=\{\varepsilon,\sigma,\sigma^2,\sigma^3,\ldots\}$ est celui des mots ne comportant aucune lettre autre que σ ; le langage $\{\sigma\sigma\}^\star=\{\varepsilon,\sigma^2,\sigma^4,\ldots\}$ est celui des mots comportant un nombre pair de σ et aucune autre lettre.*

Exercice. *Soient les langages $L=\{u\in\Sigma^\star : |u| \text{ pair}\}$ et $M=\{u\in\Sigma^\star : |u| \text{ impair}\}$. Montrer que leur somme vaut $L+M=\Sigma^\star$ et leurs produits $LL=L$, $LM=ML=M$ et $MM=L\setminus\{\varepsilon\}$. Établir, pour tout $k\geqslant 1$, que leurs puissances valent $L^k=L$ et $M^k=X\cap\{u\in\Sigma^\star : |u|\geqslant k\}$ avec $X=L$ si k est pair et $X=M$ sinon. En déduire que leurs étoiles valent $L^\star=L$ et $M^\star=\Sigma^\star$.*

Proposition. *Pour tous langages L, M et N on a les identités :*

$$\begin{aligned} L+(M+N)&=(L+M)+N & L+\varnothing&=\varnothing+L=L\\ L\cdot(M\cdot N)&=(L\cdot M)\cdot N & L\cdot\{\varepsilon\}&=\{\varepsilon\}\cdot L=L\\ L+M&=M+L & L\cdot\varnothing&=\varnothing\cdot L=\varnothing\\ L\cdot(M+N)&=L\cdot M+L\cdot N & (L^\star)^\star&=L^\star\\ (L+M)\cdot N&=L\cdot N+M\cdot N \end{aligned}$$

Forts de ces opérations, on définit une classe très importante de langages.

Définition. *On appelle expression rationnelle toute combinaison finie syntaxiquement correcte d'opérations rationnelles et des langages $\varnothing$ et $\{\sigma\}$ pour $\sigma\in\Sigma$. On note $\mathscr{L}(E)$ le langage décrit par l'expression rationnelle E. Un langage L est dit rationnel s'il admet une expression rationnelle E le décrivant, c'est-à-dire vérifiant $\mathscr{L}(E)=L$.*

L'adjectif « régulier » se substitut parfois à « rationnel » ; certains auteurs parlent ainsi d'opérations régulières, d'expressions régulières et de langages réguliers.

Exemple. *Les langages ci-dessous sont rationnels :*
- $\{\varepsilon\} = \varnothing^\star$
- $\Sigma = \{\sigma_1\} + \cdots + \{\sigma_{|\Sigma|}\}$
- *tout singleton :* $\{x\} = \{x_1\} \cdot \{x_2\} \cdots \{x_{|x|}\}$
- *tout ensemble fini*
- *les mots de longueur paire :* $(\Sigma\Sigma)^\star$
- *les mots de longueur impaire :* $\Sigma(\Sigma\Sigma)^\star$
- *les mots de longueur k ou plus :* $\Sigma^k\Sigma^\star$
- *les mots de longueur k ou moins :* $\bigcup_{\ell \leqslant k} \Sigma^\ell$
- *les mots admettant un préfixe x donné :* $\{x\}\Sigma^\star$
- *les mots admettant un facteur x donné :* $\Sigma^\star\{x\}\Sigma^\star$

Remarque. *Des expressions rationnelles distinctes peuvent parfaitement décrire le même langage qui peut d'ailleurs aussi être décrit par des expressions non rationnelles. Le langage des mots de longueur non nulle satisfait par exemple* $\Sigma \cdot \Sigma^\star = \Sigma^\star \cdot \Sigma = \Sigma^\star \setminus \{\varepsilon\}$.

Nous ne disposons pas encore d'outil assez puissant pour démontrer qu'un langage donné n'est pas rationnel. Plus tard, nous établirons que c'est notamment le cas de $\{a^k\, b^k : k \in \mathbb{N}\}$. Pour l'instant, nous devrons nous contenter du résultat existentiel suivant.

Proposition. *Il existe des langages non rationnels.*

Démonstration. L'ensemble des expressions rationnelles est dénombrable ; il se surjecte sur l'ensemble des langages rationnels qui est donc lui aussi dénombrable. L'ensemble de tous les langages, $\mathfrak{P}\left(\Sigma^{(\mathbb{N})}\right)$, est quant à lui indénombrable et donc nécessairement distinct. □

Les expressions rationnelles admettent certaines simplifications évidentes.

Exercice. *En procédant par induction sur l'ensemble des expressions rationnelles, montrer que tout langage rationnel est décrit par une expression d'une des formes*

$$\varnothing, \quad \varnothing^\star, \quad \varnothing^\star + E, \quad E$$

où E est une expression rationnelle ne faisant pas intervenir le symbole $\varnothing$.

Exercice. *Transcrire naïvement en OCaml les définitions ci-dessus en représentant un langage par sa fonction indicatrice. Quelle est la complexité des opérations de produit et d'étoile ?*
(Nous verrons plus tard comment remédier à cette inefficacité.)

```
type langage = string -> bool ;;

let vide = function x -> false ;;
let pair = function x -> String.length x mod 2 = 0 ;;
let singleton c = function x -> String.length x = 1 && x.[0] = c ;;
let somme a b = function x -> a x || b x ;;

let produit a b = function x ->
    let n = String.length x in
    let r = ref false in
    for i = 0 to n do
        let y = String.sub x 0 i in
        let z = String.sub x i (n - i) in
        if a y && b z then r := true
    done;
    !r
;;
```

```
let rec etoile a = function x ->
    let n = String.length x in
    if n = 0 then true else
    let r = ref false in
    for i = 1 to n do
        let y = String.sub x 0 i in
        let z = String.sub x i (n - i) in
        if a y && etoile a z then r := true
    done;
    !r
;;

type expression_rationnelle =
    | Vide
    | Lettre of char
    | Somme of expression_rationnelle * expression_rationnelle
    | Produit of expression_rationnelle * expression_rationnelle
    | Etoile of expression_rationnelle
;;

let rec evaluer = function
    | Vide           -> vide
    | Lettre c       -> singleton c
    | Somme (a, b)   -> somme (evaluer a) (evaluer b)
    | Produit (a, b) -> produit (evaluer a) (evaluer b)
    | Etoile a       -> etoile (evaluer a)
;;
```

Nous démontrerons plus tard la proposition ci-dessous en utilisant des résultats théoriques profonds sur les langages rationnels qui sont hors de portée à ce stade du cours.

Proposition. *Le complémentaire de tout langage rationnel est un langage rationnel.*

Corollaire. *L'intersection et la différence de deux langages rationnels sont des langages rationnels.*

Démonstration. L'intersection de deux langages rationnels est le complémentaire de l'union de leurs complémentaires ; c'est donc un langage rationnel. La différence de deux langages rationnels est l'intersection du premier avec le complémentaire du second ; c'est donc un langage rationnel. □

Pour alléger l'écriture, on commettra désormais l'abus de notation consistant à omettre les accolades, identifiant ainsi la lettre $\sigma \in \Sigma$ au langage $\{\sigma\}$ contenant l'unique mot de longueur un « σ ». On écrira donc $a^\star + (ab)^\star$ plutôt que $\{a\} + (\{a\}\{b\})^\star$.

Exercice. *Sur l'alphabet usuel $\Sigma = \{a, b\}$, écrire des expressions rationnelles décrivant :*
- *le langage des mots ne comportant pas la lettre « a » ;*
- *le langage des mots comportant au plus une fois la lettre « a » ;*
- *le langage des mots comportant au moins deux fois la lettre « a » ;*
- *le langage des mots comportant un nombre impair de fois la lettre « a » ;*
- *le langage des mots commençant et terminant par la même lettre ;*
- *le langage des mots admettant « ab » et « ba » comme facteurs ;*
- *le langage des mots n'admettant pas « aa » comme facteur ;*
- *le langage des mots n'admettant pas « aaa » comme facteur.*

Considérons à présent une classe de langages adaptée à la reconnaissance effective.

Définition. *Un langage sur l'alphabet Σ est dit local s'il est de la forme*

$$\mathscr{L}(\Sigma, P, S, F) = (P\Sigma^\star \cap \Sigma^\star S) \setminus (\Sigma^\star F \Sigma^\star)$$

ou de la forme $\{\varepsilon\} \cup \mathscr{L}(\Sigma, P, S, F)$ pour certaines parties $P, S \subset \Sigma$ et $F \subset \Sigma^2$; autrement dit, si ses mots sont entièrement caractérisés par leur première lettre, leur dernière lettre et les facteurs de longueur deux interdits.

Exemple. *Les langages $\varnothing$ et $\{\sigma\}$ sont locaux ; on a en effet :*

$$\varnothing = \mathscr{L}(\Sigma, \varnothing, \varnothing, \varnothing)$$
$$\{\sigma\} = \mathscr{L}(\Sigma, \{\sigma\}, \{\sigma\}, \Sigma\Sigma)$$

Proposition. *Tout langage local est rationnel.*

Démonstration. Découle des résultats précédents sur l'intersection et la différence. □

Afin de déterminer l'appartenance d'un mot $m = m_0 m_1 \cdots m_{n-1} \in \Sigma^\star$ au langage local $\mathscr{L}(\Sigma, P, S, F)$, la méthode évidente est la suivante.

1. Vérifier que m_0 est dans P.
2. Vérifier que m_{n-1} est dans S.
3. Pour tout $k \in \{0, 1, \ldots, n-2\}$:
4. Vérifier que $m_k m_{k+1}$ n'est pas dans F.

En représentant les ensembles P, S et F par des structures de données adaptées, cette méthode est de complexité $O(n \log |\Sigma|)$, ce qui n'est autre que la longueur de m en bits ; elle est donc optimale.

Exercice. *Quelles structures de données sont adaptées pour représenter les ensembles P, S et F ? Implanter un type OCaml pour les langages locaux et une fonction permettant de déterminer si un mot donné appartient à un langage local donné.*

```
type langage_local = {
    p : (char, unit) Hashtbl.t;
    s : (char, unit) Hashtbl.t;
    f : (char * char, unit) Hashtbl.t;
} ;;

let appartient l = function m ->
    let n = String.length m in
    n > 0 &&
    Hashtbl.mem l.p m.[0] &&
    Hashtbl.mem l.s m.[n - 1] &&
    let r = ref true in
    for k = 0 to n - 2 do
        if Hashtbl.mem l.f (m.[k], m.[k + 1]) then r := false
    done;
    !r
;;
```

Plus tard, nous réduirons le problème consistant à reconnaître efficacement les langages rationnels au cas des langages locaux. Afin d'avancer dans cette direction, il nous faut développer plus avant les concepts théoriques associés.

Proposition. *La classe des langages locaux est stable par intersection et par étoile.*
La somme de deux langages locaux sur des alphabets disjoints est un langage local.
Le produit de deux langages locaux sur des alphabets disjoints est un langage local.

Démonstration. On vérifie facilement :

$$\mathscr{L}(\Sigma, P, S, F) \cap \mathscr{L}(\Sigma, P', S', F') = \mathscr{L}(\Sigma, P \cap P', S \cap S', F \cup F')$$
$$\mathscr{L}(\Sigma, P, S, F)^\star = \{\varepsilon\} \cup \mathscr{L}(\Sigma, P, S, F \setminus SP)$$

Soit maintenant Σ' disjoint de Σ ; on a alors :

$$\mathscr{L}(\Sigma, P, S, F) + \mathscr{L}(\Sigma', P', S', F') = \mathscr{L}\left(\Sigma \cup \Sigma', P \cup P', S \cup S', F \cup F' \cup \Sigma'\Sigma \cup \Sigma\Sigma'\right)$$
$$\mathscr{L}(\Sigma, P, S, F) \cdot \mathscr{L}(\Sigma', P', S', F') = \mathscr{L}\left(\Sigma \cup \Sigma', P, S', F \cup F' \cup \Sigma'\Sigma \cup (\Sigma\Sigma' \setminus SP')\right)$$

On dispose d'égalités semblables pour les langages locaux contenant le mot vide. □

Si l'on pouvait s'affranchir de l'hypothèse de disjonction des alphabets on aurait ainsi démontré que tout langage rationnel est local, ce qui est bien évidemment faux ; on pourra par exemple vérifier que le langage $a^\star + (ab)^\star$ n'est pas local.

Définition. *Une expression rationnelle est dite linéaire lorsque chaque lettre de l'alphabet y apparaît au plus une fois.*

Proposition. *Le langage d'une expression linéaire est local.*

Démonstration. On procède par induction sur l'expression linéaire E. Pour les éléments minimaux $E = \varnothing$ et $E = \{\sigma\}$ le résultat est évident. Tout autre expression linéaire est de la forme $E + E'$, $E \cdot E'$ ou $E^\star$ avec E et E' linéaires et vérifiant donc le résultat par hypothèse d'induction. Puisque l'expression est elle-même linéaire, E et E' s'expriment sur des alphabets distincts ; la proposition précédente s'applique donc ce qui achève la preuve. □

La réciproque est fausse : par exemple, le langage $a^\star a$ est local mais n'admet pas d'expression linéaire ; en effet, il s'écrit sur l'alphabet singleton $\{a\}$ mais n'est ni un minimal ni une étoile.

Exercice. *Écrire une fonction déterminant si un objet du type* `expression_rationnelle` *défini plus haut correspond à une expression linéaire.*

```
let lineaire e =
    let rec lettres = function
        | Vide           -> []
        | Lettre x       -> [ x ]
        | Somme (e, f)   -> lettres e @ lettres f
        | Produit (e ,f) -> lettres e @ lettres f
        | Etoile e       -> lettres e
    in
    let rec doublons = function
        | []          -> false
        | [ h ]       -> false
        | h :: i :: t -> h = i || doublons (h :: t) || doublons (i :: t)
    in
    not (doublons (lettres e))
;;
```

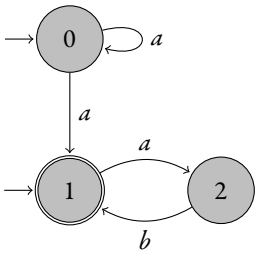


FIGURE 4.1 – Automate reconnaissant le langage $a^\star(ab)^\star$.

Exercice. *Écrire un programme prenant en argument une expression rationnelle supposée linéaire et calculant l'ensemble des préfixes de longueur un, des suffixes de longueur un, et des facteurs de longueur deux interdits.*

4.1.3 Automates

La notion d'automate donne un cadre formel pour décrire les méthodes de reconnaissances efficaces que nous avons entrevues ci-dessus dans le cas particulier des langages locaux.

Définition. *On appelle automate tout quintuplet $\mathscr{A} = (\Sigma, E, T, I, F)$ où :*
- *Σ est un alphabet (les lettres) ;*
- *E est un ensemble fini (les états) ;*
- *T est une partie de $E \times \Sigma \times E$ (les transitions) ;*
- *I est une partie de E (les états initiaux) ;*
- *F est une partie de E (les états finaux).*

Un calcul de $\mathscr{A}$ sur le mot $m = \sigma_1 \cdots \sigma_n$ est une famille d'états $(e_0, e_1, \ldots, e_n)$ vérifiant $e_0 \in I$ et $(e_{k-1}, \sigma_k, e_k) \in T$ pour tout k. Un mot m est dit accepté par l'automate s'il admet un calcul pour lequel $e_n \in F$. L'ensemble des mots acceptés est appelé langage reconnu par $\mathscr{A}$ et noté $\mathscr{L}(\mathscr{A})$. Deux automates sont dits équivalents s'ils reconnaissent le même langage.

Un automate est représenté comme le graphe obtenu en associant à chaque état un sommet et à toute transition (e, σ, f) une arête $e \to f$ étiquetée par σ. Les états initiaux sont typiquement indiqués par une flèche entrante et les états finaux par un cerclage double.

Exemple. *L'automate ayant pour alphabet $\Sigma = \{a, b\}$, états $E = \{0, 1, 2\}$, transitions $T = \{(0, a, 0), (0, a, 1), (1, a, 2), (2, b, 1)\}$, états initiaux $\{0, 1\}$ et états finaux $\{1\}$ est représenté sur la figure 4.1. Il reconnaît le langage $a^\star(ab)^\star$.*

Exercice. *Sur l'alphabet $\Sigma = \{a, b\}$, concevoir des automates reconnaissant :*
- *les mots comportant un nombre pair de lettres a ;*
- *les mots comportant la même parité de lettres a que de lettres b ;*
- *les mots admettant $abba$ comme facteur.*

Exercice. *Montrer que les automates de la figure 4.2 reconnaissent les décompositions binaires des multiples de trois et de cinq. Généraliser cette construction.*

Pour les implantations en OCaml, on fixe dorénavant le type `char` comme alphabet et le type `int` comme ensemble d'états. On pourra ainsi représenter les automates comme il suit.

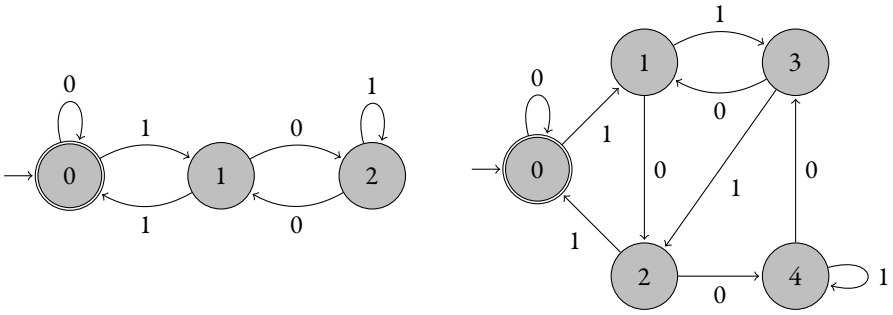


FIGURE 4.2 – Automates reconnaissant les multiples de trois et de cinq écrits en binaire.

```
type automate = {
    transitions : (int * char * int) list;
    initiaux : int list;
    finaux : int list;
} ;;

let fig41 = {
    transitions = [ (0, 'a', 0); (0, 'a', 1); (1, 'a', 2); (2, 'b', 1) ];
    initiaux = [ 0; 1 ];
    finaux = [ 1 ];
} ;;
```

Déterminer si un mot m est accepté par un automate $\mathscr{A}$ est alors relativement direct.

```
let suivants e c = function
    | [] -> []
    | (x, y, z) :: t ->
        if x = e && y = c then z :: suivants e c t
        else suivants e c t
;;

let rec calcul a m k = function
    | [] -> false
    | e :: f ->
        calcul a m k f ||
        if k = String.length m then List.mem e a.finaux
        else calcul a m (k + 1) (suivants e m.[k] a.transitions)
;;

let accepte a m = calcul a m 0 a.initiaux ;;
```

Exercice. *Quelle est la complexité du programme ci-dessus ?*
Quelles structures de données et quels algorithmes permettraient de l'améliorer ?

L'approche ci-dessus est grossièrement inefficace mais pas uniquement du fait de la naïveté des structures de données et des algorithmes qu'elle emploie. L'évaluation de multiples calculs au sein d'un même automate est en soi problématique. Nous allons donc caractériser les automates pour lesquels chaque état contribue aux calculs acceptants et pour lesquels chaque mot admet un unique calcul ; nous construirons alors des automates équivalents vérifiant ces propriétés.

Définition. *Un état e d'un automate est dit :*
- *accessible s'il existe un calcul allant d'un état initial à e ;*
- *co-accessible s'il existe un calcul allant de e à un état final.*

Un automate est dit émondé si chacun de ses états est accessible et co-accessible.

Proposition. *Soit $\mathscr{A} = (\Sigma, E, T, I, F)$ un automate. On note E' la partie de E formée des états accessibles et co-accessibles. Alors l'automate $\mathscr{A}' = (\Sigma, E', T \cap E' \times \Sigma \times E', I \cap E', F \cap E')$ est émondé et équivalent à $\mathscr{A}$.*

Démonstration. À chaque calcul acceptant de $\mathscr{A}'$ correspond un calcul acceptant de $\mathscr{A}$. Inversement, tout calcul acceptant de $\mathscr{A}'$ ne passe que par des états accessibles et co-accessibles, par définition de ces derniers, et il lui correspond donc un calcul acceptant dans $\mathscr{A}'$. □

L'émondage d'un automate consiste donc à n'en garder que les états accessibles et co-accessibles ; ceux ci sont facilement énumérés en procédant comme pour calculer la composante connexe des ensembles I et F dans le graphe associé.

Définition. *Un automate (Σ, E, T, I, F) est dit déterministe (resp. complet) lorsque $|I| \leqslant 1$ (resp. $|I| \geqslant 1$) et que l'application $(e, \sigma, f) \in T \mapsto (e, \sigma) \in E \times \Sigma$ est injective (resp. surjective).*

Proposition. *Si $\mathscr{A}$ est un automate déterministe (resp. complet) alors tout mot $m \in \Sigma^\star$ admet au plus (resp. au moins) un calcul dans $\mathscr{A}$.*

La construction d'un automate complet équivalent est aisée et consiste uniquement en l'ajout d'un état supplémentaire appelé puits servant de destination des transitions jusqu'alors manquantes.

Définition. *Soit $\mathscr{A} = (\Sigma, E, T, I, F)$ un automate et soit $p \notin E$. On appelle complété de $\mathscr{A}$ l'automate $\overline{\mathscr{A}} = (\Sigma, E \cup \{p\}, T', I', F)$ avec*

$$T' = T \cup \left\{(e, \sigma, p) : e \in E \cup \{p\}, \sigma \in \Sigma, \forall f \in E, (e, \sigma, f) \notin T\right\}$$

et $I' = I$ si cet ensemble est non vide et $I' = \{p\}$ sinon.

Le puits n'étant pas co-accessible, l'automate obtenu n'est pas émondé.

Proposition. *L'automate $\overline{\mathscr{A}}$ est complet et équivalent à $\mathscr{A}$. Il est déterministe si et seulement $\mathscr{A}$ l'est.*

Avant d'aborder la construction d'un automate déterministe équivalent, adoptons une notation permettant de manipuler plus aisément la notion de transition : pour toute lettre $\sigma \in \Sigma$ et tout ensemble d'états $H \subset E$ on pose

$$\sigma H = \{f \in E : \exists e \in H, (e, \sigma, f) \in T\};$$

autrement dit, l'ensemble σH est celui des états auxquels on accès par une transition étiquetée par σ en partant de l'un des états de H.

Lemme. *Le mot $\sigma_1\sigma_2\cdots\sigma_n$ est reconnu par l'automate $\mathscr{A} = \{\Sigma, E, T, I, F\}$ si et seulement si les ensembles F et $\sigma_n\cdots\sigma_2\sigma_1 I$ ne sont pas disjoints.*

Démonstration. Soit $(e_0, e_1, \ldots, e_n)$ un calcul acceptant le mot $m = \sigma_1\cdots\sigma_n$. Alors $e_0 \in I$ d'où $e_1 \in \sigma_1 I$ puis $e_2 \in \sigma_2\sigma_1 I$ et, par récurrence, $e_n \in \sigma_n\cdots\sigma_1 I$ ce qui démontre le résultat puisque $e_n \in F$.

Réciproquement, soit e_n un état dans $\sigma_n\cdots\sigma_1 I$ et dans F. Il existe alors un état e_{n-1} dans $\sigma_{n-1}\cdots\sigma_1 I$ tel que $(e_{n-1}, \sigma_n, e_n) \in T$ puis, par récurrence, un état $e_0 \in I$ tel que $(e_0, \sigma_1, e_1) \in T$ et la famille $(e_0, e_1, \ldots, e_n)$ est alors un calcul acceptant m. □

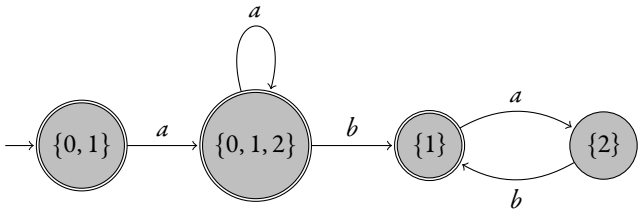


FIGURE 4.3 – Automate des parties émondé de l'automate de la figure 4.1.

Définition. *On appelle automate des parties de* $\mathscr{A} = (\Sigma, E, T, I, F)$ *l'automate*

$$\begin{aligned} \mathfrak{P}(\mathscr{A}) &= \big(\Sigma, \mathfrak{P}(E), T', \{I\}, F'\big) \\ \textit{avec} \quad T' &= \{(H, \sigma, \sigma H) : H \in \mathfrak{P}(E), \sigma \in \Sigma\} \\ \textit{et} \quad F' &= \{H \in \mathfrak{P}(E) : F \cap H \neq \varnothing\}. \end{aligned}$$

Proposition. *L'automate* $\mathfrak{P}(\mathscr{A})$ *est déterministe, complet et équivalent à* $\mathscr{A}$.

Démonstration. Ceci découle de la structure de T' et du lemme ci-dessus. □

L'automate des parties est de taille $|\mathfrak{P}(\mathscr{A})| = 2^{|\mathscr{A}|}$; aussi sa construction peut être longue. On peut néanmoins se limiter à ne calculer que les états accessibles en commençant par I et en rajoutant successivement les états auxquels mènent les transitions étiquetées par chaque lettre σ. L'état $\varnothing$ est un puits et peut donc être omis.

Exemple. *Considérons l'automate de la figure 4.1. On calcule successivement :*

$$\begin{aligned} I &= \{0, 1\} & & \\ a\{0, 1\} &= \{0, 1, 2\} & a\{1\} &= \{2\} \\ b\{0, 1\} &= \varnothing & b\{1\} &= \varnothing \\ a\{0, 1, 2\} &= \{0, 1, 2\} & a\{2\} &= \varnothing \\ b\{0, 1, 2\} &= \{1\} & b\{2\} &= \{1\} \end{aligned}$$

Outre l'état puits $\varnothing$*, l'automate des parties compte donc quatre états accessibles correspondant aux ensembles ci-dessus ; représenté sur la figure 4.3, il est déterministe et reconnaît aussi le langage* $a^\star(ab)^\star$.

Cette méthode permet non seulement de calculer un automate déterministe équivalent, mais aussi d'effectuer de manière efficace le calcul d'un automate (supposé non déterministe) sur un mot $\sigma_1 \cdots \sigma_n$: en stockant à chaque étape un ensemble d'états plutôt qu'un état individuel, à savoir $\sigma_k \cdots \sigma_1 I$ pour k allant de zéro à n.

Lorsque l'on considère exclusivement des automates déterministes et complets, la notion d'ensemble de transitions peut être simplifiée en celle de fonction de transition qui est indéniablement plus intuitive.

Définition. *Soit* $\mathscr{A} = (\Sigma, E, T, I, F)$ *un automate déterministe complet. On appelle fonction de transition de* $\mathscr{A}$ *l'unique fonction* $\delta : E \times \Sigma \to E$ *vérifiant*

$$\forall e \in E, \forall \sigma \in \Sigma, \big(e, \sigma, \delta(e, \sigma)\big) \in T.$$

On pourra donc occasionnellement voir des automates déterministes complets définis comme étant des quintuplets $(\Sigma, Q, \delta, i_0, F)$ avec Q l'ensemble des états, δ la fonction de transition et i_0 l'unique état initial.

Exercice. *Soit L le langage reconnu par un automate $\mathscr{A}$. Construire un automate $\mathscr{A}'$ reconnaissant son langage des préfixes $L' = \{x \in \Sigma^\star, \exists y \in \Sigma^\star, x\,y \in L\}$.*

Exercice. *Soit L le langage reconnu par un automate $\mathscr{A}$. Construire un automate $\mathscr{A}'$ reconnaissant son langage verlan $L' = \{\sigma_n \cdots \sigma_1 : \sigma_1 \cdots \sigma_n \in L\}$.*

Théorème (Kleene [6]). *Les langages rationnels sont exactement ceux reconnus par les automates.*

Ce célèbre théorème admet de multiples démonstrations ; nous donnerons ici deux preuves du sens direct et une du sens réciproque, chacune constructive. Commençons par le sens réciproque pour lequel nous empruntons une méthode due à McNaughton et Yamada [13] qui reprend la structure de l'algorithme de Floyd–Warshall vu à la section 3.1.2 pour le calcul des plus courts chemins dans un graphe.

Démonstration du sens réciproque du théorème de Kleene. Soit $\mathscr{A} = (\Sigma, E, T, I, F)$ un automate ; on suppose $E = \{1, \ldots, n\}$ sans perte de généralité. Pour tout $k \in \mathbb{N}$ et $(e, f) \in E^2$ on note $L_k(e, f)$ le langage des mots admettant un calcul dans $\mathscr{A}$ allant de e à f en passant exclusivement par des états de $\{1, \ldots, k\}$. Le langage reconnu par $\mathscr{A}$ admet alors l'expression

$$\mathscr{L}(\mathscr{A}) = \sum_{e \in I} \sum_{f \in F} L_n(e, f).$$

Construisons maintenant des expressions rationnelles pour $L_k(e, f)$ par récurrence sur k. Pour $k = 0$, aucun état intermédiaire n'intervient donc les calculs ne sont que de simples transitions :

$$L_0(e, f) = \sum_{(e, \sigma, f) \in T} \{\sigma\} + \begin{cases} \varnothing^\star & \text{si } e = f \\ \varnothing & \text{si } e \neq f \end{cases}$$

Pour $k \geqslant 1$, les calculs de $L_k(e, f)$ sont soit restreints aux états $\{1, \ldots, k-1\}$, soit passent par l'état k ; on a donc

$$L_k(e, f) = L_{k-1}(e, f) + L_{k-1}(e, k)\big(L_{k-1}(k, k)\big)^\star L_{k-1}(k, f)$$

ce qui achève la preuve du sens réciproque. □

Remarque. *Notons qu'en toute généralité l'expression rationnelle ainsi obtenue est de longueur exponentielle en le nombre d'états de l'automate.*

Afin d'établir le sens direct, l'approche la plus naturelle consiste à combiner les constructions suivantes de sorte à obtenir un automate reconnaissant une expression rationnelle donnée

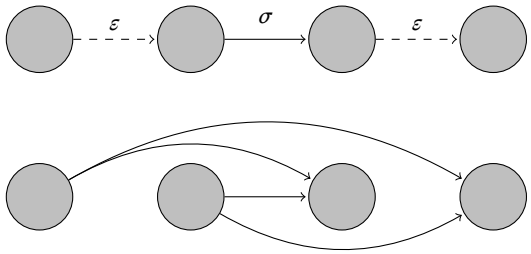


FIGURE 4.4 – Construction d'un automate équivalent sans transition spontanée.

Automate produit. Si $\mathscr{A}$ et $\mathscr{B}$ sont deux automates complets sur un même alphabet Σ alors l'automate $(\Sigma, E_{\mathscr{A}} \times E_{\mathscr{B}}, T', I', F')$ reconnaît le langage $\mathscr{L}(\mathscr{A}) + \mathscr{L}(\mathscr{B})$ où l'on a posé

$$T' = \big\{\big((e,f),\sigma,(g,h)\big) : (e,\sigma,g) \in T_{\mathscr{A}} \wedge (f,\sigma,h) \in T_{\mathscr{B}}\big\},$$
$$I' = I_{\mathscr{A}} \times I_{\mathscr{B}}, \qquad F' = (F_{\mathscr{A}} \times E_{\mathscr{B}}) \cup (E_{\mathscr{A}} \times F_{\mathscr{B}}).$$

Calculer dans l'automate produit revient à calculer parallèlement dans $\mathscr{A}$ et dans $\mathscr{B}$.

Il nous sera commode de faire usage de transitions spontanées, appelées aussi ε-transitions. Cela consiste à étendre la définition des automates pour permettre $T \subset E \times (\Sigma \cup \{\varepsilon\}) \times E$, les transitions de la forme (e,ε,f) pouvant intervenir dans les calculs sans « consommer » de lettre : un calcul arrivant en e pouvant alors se poursuivre, au choix, depuis e ou f.

Automate étoile. Si l'automate (Σ, E, T, I, F) reconnaît le langage L alors l'automate (Σ, E, T', I, F) reconnaît le langage $L^\star$ où l'on a posé

$$T' = T \cup \big\{(f,\varepsilon,i) : f \in F, i \in I\big\}.$$

Les transitions rajoutées permettent de recommencer en un état initial tout calcul acceptant.

Automate union. Si $\mathscr{A}$ et $\mathscr{B}$ sont deux automates sur un même alphabet Σ dont les ensembles d'états $E_{\mathscr{A}}$ et $E_{\mathscr{B}}$ sont disjoints, alors l'automate $(\Sigma, E_{\mathscr{A}} \cup E_{\mathscr{B}}, T', I_{\mathscr{A}}, F_{\mathscr{B}})$ reconnaît le langage $\mathscr{L}(\mathscr{A}) \cdot \mathscr{L}(\mathscr{B})$ où l'on a posé

$$T' = T_{\mathscr{A}} \cup \big\{(f,\varepsilon,i) : f \in F_{\mathscr{A}}, i \in I_{\mathscr{B}}\big\} \cup T_{\mathscr{B}}.$$

Tout automate $\mathscr{A} = (\Sigma, E, T, I, F)$ comportant des transitions spontanées est équivalent à un automate n'en comportant pas. En effet, sauf peut-être pour ce qui concerne le mot vide, le langage reconnu est inchangé lorsqu'on substitut T par l'ensemble des triplets $(e,\sigma,f) \in E \times \Sigma \times E$ pour lesquels le mot σ admet un calcul dans $\mathscr{A}$ allant de e à f en passant éventuellement par des transitions spontanées. Voir la figure 4.4.

La figure 4.5 récapitule les propriétés de ces constructions. En les combinant on obtient la preuve ci-dessous, souvent appelée *algorithme de Thompson* [21].

Démonstration du sens direct du théorème de Kleene. Afin de construire un automate reconnaissant le langage d'une expression rationnelle donnée, procédons par induction structurelle sur l'ensemble des expressions rationnelles.

Pour les éléments minimaux, c'est-à-dire $\varnothing$ et σ, le résultat est clair. Toute autre expression rationnelle M est de la forme $M = L + L'$ ou $M = L \cdot L'$ ou $M = L^\star$ avec L et L' reconnues

AUTOMATE	LANGAGE RECONNU	NOMBRE D'ÉTATS	DÉTERMINISTE
parties de $\mathscr{A}$	$\mathscr{L}(\mathscr{A})$	$2^{\lvert\mathscr{A}\rvert}$	toujours
produit $\mathscr{A} \times \mathscr{B}$	$\mathscr{L}(\mathscr{A}) \cup \mathscr{L}(\mathscr{B})$	$\lvert\mathscr{A}\rvert \times \lvert\mathscr{B}\rvert$	si $\mathscr{A}$ et $\mathscr{B}$ le sont
union $\mathscr{A} \cup \mathscr{B}$	$\mathscr{L}(\mathscr{A}) \cdot \mathscr{L}(\mathscr{B})$.	$\lvert\mathscr{A}\rvert + \lvert\mathscr{B}\rvert$	pas en général
étoile $\mathscr{A}^\star$	$\mathscr{L}(\mathscr{A})^\star$	$\lvert\mathscr{A}\rvert + 1$	pas en général

FIGURE 4.5 – Tableau récapitulatif des opérations élémentaires sur les automates.

par des automates $\mathscr{A}$ et $\mathscr{B}$ d'après l'hypothèse d'induction ; les opérations décrites ci-dessus montrent alors que M est lui aussi reconnu par un automate. □

Cette approche fourni une méthode de reconnaissance de motifs extrêmement efficace : pour rechercher une expression rationnelle de longueur n dans un texte de longueur m, on construit un automate déterministe reconnaissant le langage de cet expression puis on effectue le calcul du texte dans cet automate. La complexité totale est ainsi $O(2^n)+O(m)$ ce qui explique les deux étapes proposées par Python au travers des fonctions `re.compile()` et `re.search()`.

Exercice. *Par cette méthode, construire un automate reconnaissant le langage* $(a + b\,a\,b)^\star$. *Faire ensuite de même pour le langage* $((a\,b)^\star a + b)a$.

Corollaire. *Le complémentaire de tout langage rationnel est rationnel.*

Démonstration. Soit L un langage rationnel. D'après le sens direct du théorème de Kleene, il est reconnu par un automate $\mathscr{A} = (\Sigma, E, T, I, F)$ que l'on peut supposer complet et déterministe. Un mot $m \in \Sigma^\star$ est ainsi dans L si et seulement si son unique calcul dans $\mathscr{A}$ termine sur un état final. Alors l'automate $(\Sigma, E, T, I, E \setminus F)$ reconnaît le langage $\Sigma^\star \setminus L$ qui est donc lui aussi rationnel d'après le sens réciproque du théorème de Kleene. □

Entreprenons une seconde preuve du sens direct du théorème de Kleene, moins naturelle mais plus élégante, en reprenant la théorie des langages locaux.

Définition. *Un automate* (Σ, E, T, I, F) *est dit local s'il est déterministe et satisfait* $|\{f : (e, \sigma, f) \in T\}| \leqslant 1$ *pour tout* $\sigma \in \Sigma$ *; autrement dit, si toutes les transitions étiquetées par* σ *mènent à un unique état.*

Proposition. *Les langages locaux sont exactement ceux reconnus par les automates locaux.*

Démonstration. Le langage local $(P\Sigma^\star \cap \Sigma^\star S) \setminus (\Sigma^\star F \Sigma^\star)$ est reconnu par l'automate local

$$(\Sigma, \Sigma \cup \{\varepsilon\}, T, \{\varepsilon\}, S) \quad \text{avec} \quad T = \{(\varepsilon, \sigma, \sigma) : \sigma \in P\} \cup \{(\sigma, \tau, \tau) : \sigma\tau \notin F\}.$$

Réciproquement, soit (Σ, E, T, I, F) un automate local. On désigne par ε son état initial et par e_σ l'unique état où les transitions étiquetées par σ mènent. Soient alors $P = \{\sigma : (\varepsilon, \sigma, e_\sigma) \in T\}$, $F' = \{\sigma\tau : (e_\sigma, \tau, e_\tau) \notin T\}$. Un mot est reconnu si et seulement si sa première lettre est dans P, sa dernière lettre dans F et aucun de ses facteurs de longueur deux n'est dans F' ; autrement dit, le langage reconnu est local. □

Appelée algorithme de Glushkov [15, 30], la preuve ci-dessous construit un automate reconnaissant le langage d'une expression rationnelle donnée en différentiant artificiellement ses lettres afin de se ramener au cas d'expressions linéaires.

Démonstration du sens direct du théorème de Kleene. Soit L un langage rationnel ; rappelons qu'il admet une expression de la forme $\varnothing, \varnothing^\star, \varnothing^\star + E$ ou E pour une expression rationnelle E ne faisant pas intervenir le symbole $\varnothing$.

Notons alors $(\sigma_1, \dots, \sigma_n)$ les lettres de E dans l'ordre d'apparition ; l'expression E' sur l'alphabet $\Sigma' = \{1, \dots, n\}$ obtenue en remplaçant la k^e lettre de E par k est évidemment linéaire. Son langage est local donc reconnu par un automate local $\mathscr{A}'$; en y remplaçant chaque étiquette de transition k par σ_k on obtient un automate $\mathscr{A}$ reconnaissant E. Si $\varepsilon \in L$ il reste alors à rendre l'état initial final. □

L'automate ainsi obtenu possède $|E| + 1$ états mais est non déterministe. S'il peut être obtenu en $O(|E|)$ opérations, le calcul d'un automate déterministe équivalent nécessite $O(2^{|E|})$ opérations.

Terminons par un résultat permettant d'établir la non rationalité.

Lemme (de l'étoile ou de pompage). *Tout langage rationnel L admet un entier N tel que tout mot $m \in L$ de longueur supérieure ou égale à N admet une décomposition $m = xyz$ vérifiant $|xy| \leqslant N$, $|y| \geqslant 1$ et $\forall k \in \mathbb{N}, xy^k z \in L$.*

Démonstration. Soit L un langage rationnel. Par le théorème de Kleene, il existe un automate (Σ, E, T, I, F) reconnaissant L ; on pose alors $N = |E|$. Si un mot $m = \sigma_1 \cdots \sigma_n$ est dans L alors il admet un calcul acceptant $(e_0, e_1, \dots, e_n)$. Supposant $n \geqslant N$, par le principe des tiroirs, il existe deux indices $i < j$ dans $\{0, \dots, N\}$ tels que $e_i = e_j$. On obtient alors le résultat voulu avec les facteurs $x = \sigma_1 \cdots \sigma_i$, $y = \sigma_{i+1} \cdots \sigma_j$ et $z = \sigma_{j+1} \cdots \sigma_n$. □

L'implication pour $k = 0$ est parfois aussi utile que pour $k \to \infty$. Elle permet notamment de montrer que si un automate de taille n ne reconnaît aucun mot de longueur strictement inférieure à n, alors son langage est vide.

Exemple. *Le langage $\{a^k b^k : k \in \mathbb{N}\}$ n'est pas rationnel.*

Exercice. *Montrer que le langage des mots comportant davantage de lettres a que de lettres b n'est pas rationnel.*

Rajoutons en complément du programme deux méthodes de construction de l'automate minimal.

Langages résiduels. Fixons un alphabet Σ et définissons, pour tout langage L, une relation d'équivalence sur les mots de $\Sigma^\star$.

Définition. *Pour tout mot x et langage L on pose $x^{-1}L = \{y \in \Sigma^\star : xy \in L\}$. Sur le monoïde libre $\Sigma^\star$ on défini une relation d'équivalence $\mathscr{R}_L$ par $x \mathscr{R}_L z \iff x^{-1}L = z^{-1}L$.*

Théorème (Myhill–Nerode [9, 10]). *Le langage L est rationnel si et seulement si le quotient $\Sigma^\star/\mathscr{R}_L$ est fini. La quantité $|\Sigma^\star/\mathscr{R}_L|$ est alors le plus petit nombre d'états des automates déterministes complets reconnaissant L.*

Démonstration. Si L est rationnel alors soit (Σ, E, T, I, F) un automate déterministe complet le reconnaissant. Pour tout état accessible $e \in E$ on note $\overline{e}$ l'ensemble des mots dont un calcul commençant en e termine sur un état final ; c'est une classe pour $\mathscr{R}_L$ et l'application $e \in E \mapsto \overline{e} \in \Sigma^\star/\mathscr{R}_L$ est surjective.

Réciproquement, si $\Sigma^\star/\mathscr{R}_L$ est fini alors soit l'automate $(\Sigma, \Sigma^\star/\mathscr{R}_L, T, I, F)$ avec :

$$\begin{aligned} T &= \left\{\left(x^{-1}L, \sigma, (x\sigma)^{-1}L\right) : x \in \Sigma^\star, \sigma \in \Sigma\right\} \\ I &= \left\{\varepsilon^{-1}L\right\} \\ F &= \left\{x^{-1}L : x \in L\right\} \end{aligned}$$

Son langage reconnu n'est autre que L et, d'après la preuve du sens direct ci-dessus, il admet un nombre d'états minimal. □

Il découle de cette preuve que l'on peut calculer le quotient $\Sigma^\star/\mathscr{R}_L$ et ainsi l'automate minimal en partant d'un automate arbitraire reconnaissant L puis en identifiant les couples d'états (e, f) vérifiant $\overline{e} = \overline{f}$.

Dérivation d'expressions rationnelles. Une autre approche, due à Antimirov [32], consiste à ne travailler qu'avec des expressions rationnelles.

Définition. *La dérivée d'une expression rationnelle E par rapport à une lettre σ est l'ensemble d'expressions rationnelles noté $\partial_\sigma E$ défini par induction suivant les règles :*

$$\partial_\sigma \varnothing = \{\varnothing\} \qquad \partial_\sigma(E+F) = \partial_\sigma E \cup \partial_\sigma F$$

$$\partial_\sigma\{\tau\} = \begin{cases} \{\varnothing^\star\} & \text{si } \sigma = \tau \\ \{\varnothing\} & \text{si } \sigma \neq \tau \end{cases} \qquad \partial_\sigma(E\cdot F) = \begin{cases} \partial_\sigma E \cdot \{F\} & \text{si } \varepsilon \notin \mathscr{L}(E) \\ \partial_\sigma E \cdot \{F\} \cup \partial_\sigma F & \text{si } \varepsilon \in \mathscr{L}(E) \end{cases}$$

$$\partial_\sigma(E^\star) = \partial_\sigma E \cdot \{E^\star\}$$

où les opérations produit et dérivée ci-dessus sont implicitement étendues aux ensembles d'expressions rationnelles par les formules $A \cdot B = \{E \cdot F : E \in A, F \in B\}$ et $\partial_\sigma A = \bigcup_{E\in A} \partial_\sigma E$.

Pour tout mot $m = \sigma_1 \cdots \sigma_n$ on pose alors $\partial_m E = \partial_{\sigma_n} \cdots \partial_{\sigma_1} E$.

Lemme. *On a l'identité $\mathscr{L}(\partial_m E) = m^{-1}\mathscr{L}(E)$.*

Démonstration. On commence par démontrer ce résultat dans le cas $m = \sigma$ en procédant par induction sur l'ensemble des expressions rationnelles. Le cas général s'ensuit par récurrence sur la longueur de m. □

La définition ci-dessus semble inutilement lourde car les unions d'ensembles d'expressions expriment le même langage que leur somme, elle-même une expression et non un ensemble. C'est justement là l'idée clef d'Antimirov qui ne permet à chaque expression rationnelle de n'avoir qu'un nombre fini de dérivées.

Proposition. *L'ensemble $\{\partial_m E : m \in \Sigma^\star\}$ est fini.*

Démonstration. Par induction sur l'ensemble des expressions rationnelles. □

Exercice. *Concevoir puis implanter en langage OCaml une fonction calculant par cette approche un automate reconnaissant une expression rationnelle donnée. Quelle est sa complexité ?*

Annexe A

Feuilles de TD

A.1 Premier semestre

Programmation impérative (Python)

1. *On note c_r le nombre de points du plan à coordonnées entières situés sur le disque de rayon r centré à l'origine. Calculer c_{10^2} puis c_{10^4}. Déterminer un équivalent de c_r lorsque $r \to \infty$.*

2. *Un théorème de Lagrange stipule que tout entier naturel n peut s'exprimer comme la somme de quatre carrés. On a par exemple $31 = 5^2 + 2^2 + 1^2 + 1^2$. Exprimer $n = 1234$ comme somme de quatre carrés. Combien $n = 1234$ admet-il de telles décompositions ? Quel est le nombre moyen de telles décompositions lorsque n parcours $\{1, \dots, 1234\}$?*

3 (suites de Syracuse). *Soit s une suite à valeurs entières vérifiant, pour tout $k \in \mathbb{N}$, la relation*

$$s_{k+1} = \begin{cases} s_k/2 & \text{si } s_k \bmod 2 = 0, \\ 3s_k + 1 & \text{si } s_k \bmod 2 = 1. \end{cases}$$

Vérifier que, pour tout $s_0 \in \left\{1, 2, \dots, 10^4\right\}$, il existe k tel que $s_k = 1$. Quelle est la valeur moyenne du plus petit tel entier k lorsque s_0 parcours $\left\{1, 2, \dots, 10^4\right\}$? Son écart type ?

4 (régression linéaire). *Soient $X = (x_1, \dots, x_n)$ et $Y = (y_1, \dots, y_n)$ deux familles de réels. La droite $y = ax + b$ dont la somme des carrés des distances aux points (x_i, y_i) est minimum vérifie :*

$$a = \frac{E(XY) - E(X)E(Y)}{E(X^2) - E(X)^2}, \quad b = \frac{E(X^2)E(Y) - E(XY)E(X)}{E(X^2) - E(X)^2}.$$

Écrire un programme Python qui, étant données deux listes X et Y, calcule ces coefficients. Choisir dix points puis les tracer ainsi que cette droite. On pourra s'inspirer du code suivant :

```
X = [1, 2]
Y = [3, 4]
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.scatter(X, Y)
ax.plot(X, Y)
plt.show()
```

5 (suite logistique). *La suite logistique de paramètre $\lambda \in [0, 4]$ est définie par la relation de récurrence $x_{n+1} = \lambda x_n(1 - x_n)$; on prendra ici la condition initiale $x_0 = \frac{1}{2}$. Observer son comportement pour $\lambda = 8/3$ puis $\lambda = 10/3$. Écrire un programme calculant (une approximation de) l'ensemble des valeurs d'adhérence de x_n en fonction de λ. Tracer cet ensemble afin d'obtenir la figure A.1.*

Programmation récursive (OCaml)

6. *Écrire une fonction déterminant si un entier donné est premier. Écrire ensuite une fonction renvoyant la liste des facteurs premiers d'un entier donné.*

7 (segmentation d'une liste). *Soit p un entier et L une liste d'entiers. Écrire une fonction* `decoupe` *qui, étant donnés p et L, renvoie deux listes M et N et un entier k tels que M (resp. N) contient les éléments de L strictement inférieurs (resp. supérieurs) à p et que k est le nombre d'occurrences de p dans L.*

En déduire une fonction déterminant si une liste donnée est sans répétition. En déduire une fonction déterminant un élément d'une liste donnée ayant le nombre maximal d'occurrences. En déduire une fonction déterminant si deux listes données sont disjointes.

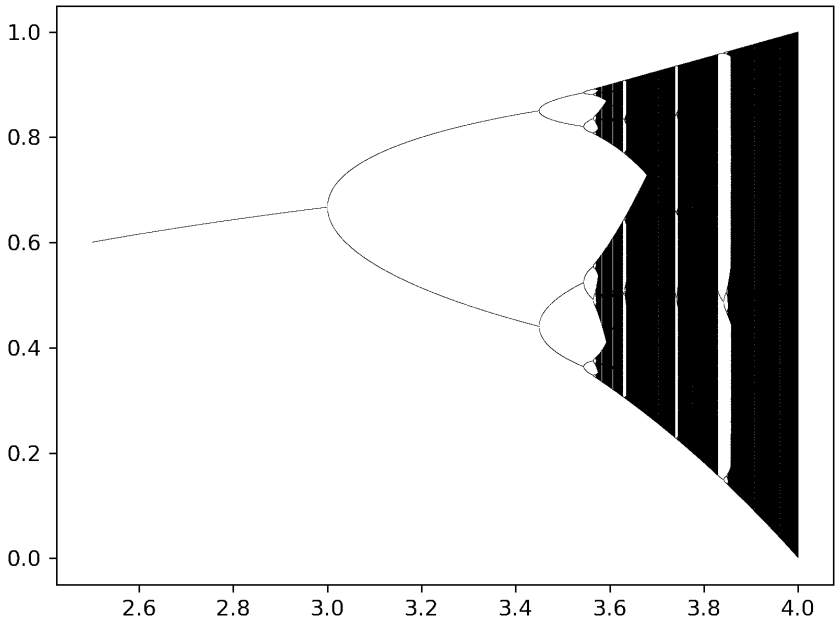


FIGURE A.1 – Valeurs d'adhérence de la suite logistique pour $\lambda \in [5/2, 4]$.

8. *Écrire un programme renvoyant la liste des sous suites d'une liste donnée ; par exemple, étant donnée* `[1;2]`*, il renverrait* `[ []; [1]; [2]; [1;2] ]` *à permutation près. En déduire un programme déterminant la plus longue sous suite croissants d'une liste donnée.*

On suppose maintenant que la suite `L` *est donnée sous forme d'un tableau et on se propose d'en déterminer la plus longue sous suite croissante plus efficacement en utilisant deux tableaux auxiliaires* `N` *et* `P` *: on stockera en* `N.(k)` *la longueur de la plus longue sous suite croissance se terminant en* `L.(k)` *et en* `P.(k)` *l'avant dernier élément de la plus longue sous suite croissance se terminant en* `L.(k)`*. Calculer ces tableaux en commençant par* $k = 0$ *puis en incrémentant* k*.*

9. *Écrire un programme affichant les n premières lignes du triangle de Pascal. On pourra utiliser les fonctions* `print_int`*,* `print_string` *et* `print_newline`*.*

10. *Montrer que la fonction* $f : (x, y) \mapsto x + (x + y)(x + y + 1)/2$ *réalise une bijection de* $\mathbb{N}^2$ *vers* $\mathbb{N}$*. L'implanter en OCaml ainsi que sa réciproque. Implanter alors une fonction de signature «* `int list -> int` *» qui, pour tout* $k \in \mathbb{N}^\star$*, réalise une bijection de* $\mathbb{N}^k$ *vers* $\mathbb{N}$*.*

11 (arithmétique de Peano)**.** *Un entier naturel est soit nul soit le successeur d'un autre entier naturel ; on définit ainsi leur ensemble par «* `type naturel = Zero | S of naturel ;;` *». L'entier* 4 *s'écrit alors* `S(S(S(S Zero)))`*. Écrire des fonctions de conversions* `naturel_of_int` *et* `int_of_naturel`*. Écrire des fonctions calculant l'addition, le produit et la soustraction de deux entiers de type* `naturel`*. Faire enfin de même pour le quotient et le reste.*

12 (calcul formel)**.** *On définit en OCaml un type pour une classe d'expressions mathématiques :*

```
type expression =
    | Constante of float
    | Variable of string
    | Somme of expression * expression
```

```
    | Difference of expression * expression
    | Produit of expression * expression
    | Quotient of expression * expression
;;
```

Écrire une fonction permettant d'évaluer une telle expression, supposant qu'elle ne fasse intervenir aucune variable. Écrire une fonction permettant de dériver une telle expression par rapport à une variable donnée. Écrire enfin une fonction convertissant une chaîne de caractères du type `(x+1)/2` *en une expression comme ci-dessus.*

13. *Programmer le jeu de Nim. Voir* `https://en.wikipedia.org/wiki/Nim`*. On pourra utiliser les fonctions* `read_int` *et* `Random.int`*.*

A.2 Second semestre

Représentation des nombres (Python)

14. *En calculant avec des nombres flottants, on obtient les résultats ci-dessous ; expliquer pourquoi l'associativité n'est pas respectée.*

```
0.1 + (0.2 + 0.3) == 0.599999999999999978
(0.1 + 0.2) + 0.3 == 0.600000000000000089
```

15. *Comment effectueriez vous des calculs avec des nombres rationnels ? Présenter les avantages et inconvénients de chaque approche envisagée.*

Méthodes de programmation et analyse des algorithmes (Python)

16. *Soit un fichier* `rec.txt` *contenant une liste de noms de fichiers, stockés ligne par ligne. Deux fichiers de noms distincts peuvent avoir un contenu identique. Écrire un nouveau fichier* `uniq.txt` *contenant la même liste de noms de fichiers mais en se limitant à un seul exemplaire pour chaque contenu.*

17 (conjecture de Goldbach). *On conjecture que tout entier pair $n \geqslant 3$ peut s'écrire comme la somme de deux nombres premiers. On a par exemple* $42 = 5 + 37$. *Décomposer* 123456 *en somme de deux nombres premiers. Cette conjecture a été vérifiée par ordinateur pour tous les entiers $n < 4 \cdot 10^{18}$. Combien de temps environ cette vérification prendrait-elle avec votre programme ?*

On note $a(n)$ le nombre de décomposition de n en somme de deux nombres premiers. Calculer $a(123456)$. Pour $n = 2p$ avec p premier, on conjecture l'équivalent $a(n) \sim \lambda \frac{n}{\ln(n)^2}$ lorsque $n \to \infty$. Calculer une valeur approchée de λ.

18. *Écrire des algorithmes permettant de calculer l'addition de deux matrices de $\mathcal{M}_n(\mathbb{C})$. En démontrer la terminaison, la correction et en déterminer une borne de complexité. Faire de même pour la multiplication puis l'inversion.*

Structures de données

19. *Plutôt que de servir ses clients par ordre d'arrivée, une boulangerie décide de s'occuper en priorité des derniers arrivés, c'est-à-dire utilisant une pile plutôt qu'une file, dans l'espoir de satisfaire le plus grand nombre en réduisant leur temps d'attente au détriment d'un petit nombre.*

On suppose que l'unique vendeuse met une minute à servir chaque client et que celui-ci repart satisfait si son temps d'attente n'a pas dépassé cinq minutes. La boulangerie améliore t-elle son taux de satisfaction si, une heure durant, sept clients arrivent simultanément toutes les cinq minutes ? Et pour d'autres rythmes d'arrivée ?

20 (jeu des tours de Hanoï). *Pour un entier $n \in \mathbb{N}$ fixé on considère le jeu suivant :*

- *Soient trois piles, la première initialement $[n, n-1, \dots, 3, 2, 1]$ et les deux autres vides.*
- *On effectue une succession d'étapes, chacune consistant à dépiler un entier d'une pile et à l'empiler sur une autre en préservant la décroissance de chaque pile.*
- *L'objectif est de vider complètement la première et seconde pile.*

On appelle état du jeu le contenu de ces trois piles ; écrire un programme prenant comme argument un état et renvoyant la liste des états après chaque étape possible. Écrire un programme déterminant la stratégie gagnante utilisant le moins d'étapes possible. Admettant qu'elle utilise $2^n - 1$ étapes, quelle est la complexité de votre programme ?

Une autre stratégie consiste à déplacer récursivement les $n-1$ premiers entiers sur la seconde pile de sorte à obtenir l'état $([n], [n-1, \dots, 3, 2, 1], [])$, puis à déplacer n sur la troisième pile,

puis à re-déplacer les $n-1$ premiers entiers sur la troisième pile. Implanter cette stratégie sous forme de programme récursif. Quelle est le nombre d'étapes utilisées ?

21. *On considère une pile à deux extrémités : c'est un tableau linéaire auquel on peut ajouter un élément à gauche, ajouter un élément à droite, enlever l'élément le plus à gauche, enlever l'élément le plus à droite. Proposer différentes implantations de cette structure de données et donner pour chacune la complexité de ces quatre opérations.*

22 (jeu de la vie [23]). *On appelle cellule tout élément de $\mathbb{Z}^2$ représenté comme une grille planaire ; les voisins de la cellule (α, β) sont les huit cellules $(\alpha+\varepsilon, \beta+\delta)$ avec $(\varepsilon, \delta) \in \{-1, 0, +1\}^2 \setminus \{(0,0)\}$. À l'instant $t=0$, chaque cellule est soit morte soit vivante. À l'instant $t+1$, une cellule est vivante si et seulement si :*

— *Au temps t elle était morte et trois de ses voisines étaient vivantes.*

— *Au temps t elle était vivante et deux ou trois de ses voisines aussi.*

Programmer l'évolution de ce système pour différents états initiaux.

23. *On se propose d'énumérer toutes les permutations σ de $\{1, \dots, n\}$ en représentant chacune par le tableau $[|\sigma(1), \sigma(2), \dots, \sigma(n)|]$; la méthode que voici construit le successeur de la permutation $[|x_1, x_2, \dots, x_n|]$ pour l'ordre lexicographique :*

1. *Déterminer le plus petit $k \in \{1, \dots, n\}$ pour lequel $x_k > x_{k+1} > \cdots > x_n$.*
2. *Échanger x_{k-1} avec le plus petit x_ℓ vérifiant $x_\ell > x_{k-1}$ et $\ell > k-1$.*
3. *Trier dans l'ordre croissant les $n-k+1$ derniers éléments de x.*

Écrire une fonction de signature « `int -> ()` » affichant à l'écran les $n!$ permutations de $\{1, \dots, n\}$. Montrer sa correction. Borner sa complexité.

Types récursifs immuables et arbres

24 (arbres de Fibonacci). *On considère la suite d'arbres $(A_k)_{k\in\mathbb{N}}$ pour laquelle A_0 et A_1 sont réduits à leurs racines et A_{k+2} est l'arbre binaire ayant A_{k+1} comme sous arbre gauche et A_k comme sous arbre droit. Écrire une fonction OCaml calculant ces arbres. Déterminer la taille de A_k. Déterminer la hauteur de A_k puis montrer qu'en tout nœud la hauteur du sous arbre gauche et celle du sous arbre droit diffèrent au plus d'une unité.*

25. *Pour tout $n \in \mathbb{N}$, on note b_n le nombre d'arbres binaires de taille n ; on a par exemple $b_0 = 1$ (arbre vide), $b_1 = 1$, $b_2 = 2$ et $b_3 = 5$. Établir que la suite (b_n) satisfait la relation de récurrence $b_{n+1} = \sum_{k=0}^{n} b_k b_{n-k}$. Vérifier alors que la série formelle $f(x) = \sum_{n\in\mathbb{N}} b_n x^n$ satisfait l'équation fonctionnelle $x f(x)^2 = f(x) - 1$; en résolvant classiquement ce trinôme du second degré puis en développant en série la solution trouvée, obtenir l'identité $b_n = \frac{1}{n+1} C_{2n}^n$.*

26. *On modélise par un arbre de décision un algorithme triant une liste de n éléments distincts $(x_1, \dots, x_n)$. Donner un minorant de sa hauteur. On considère maintenant le cas $n = 4$.*

Supposant qu'à l'exception d'un unique élément la liste soit déjà triée, on souhaite que l'algorithme effectue en priorité les comparaisons $x_i < x_j$ les plus révélatrices. On mesure la pertinence d'une comparaison (i, j) par la quantité $E_{(i,j)} = -p\log(p) - q\log(q)$ où p et q dénotent la proportion de listes vérifiant respectivement $x_i < x_j$ et $x_i > x_j$. Déterminer un couple pour lequel cette quantité est maximale. Créer alors un arbre de décision en plaçant cette comparaison comme racine puis en itérant ce procédé pour construire ses sous arbres gauche et droite. Quelle est la complexité de l'algorithme correspondant ?

27. *Soit un arbre de recherche stockant les valeurs $x_1 < \cdots < x_n$. Montrer que si le nœud x_i a deux fils, alors x_{i+1} n'a pas de fils gauche. Donner un contre exemple lorsque x_i n'a qu'un seul fils.*

28 (arbre AVL [18]). *On se propose de concevoir un type d'arbre de recherche automatiquement équilibré.*

Définir un type OCaml identique à celui d'arbre de recherche mais munissant chaque nœud d'une donnée supplémentaire appelée facteur d'équilibrage destinée à contenir la différence entre la hauteur du sous arbre droit et celle du sous arbre gauche.

On y insère, cherche et supprime des éléments comme dans un arbre de recherche classique tout en prenant soin de mettre à jour les facteurs d'équilibrage. Lorsque le facteur d'équilibrage d'un nœud sort de l'intervalle $\{-1, 0, +1\}$ *on effectue la transformation de la figure A.2, celle de la figure A.3 ou l'une de leurs symétriques.*

Implanter ces opérations d'insertion, recherche et suppression puis en borner la complexité.

Bases des graphes, plus courts chemins

29. *On appelle diamètre d'un graphe connexe* $G = (S, A)$ *la quantité* $\max_{(\alpha,\beta)\in S^2} d(\alpha, \beta)$ *où* $d(\alpha, \beta)$ *dénote la longueur d'un plus court chemin reliant les sommets* α *et* β*. Donner des graphes à n sommets pour lesquels cette quantité est extrémale. Concevoir un algorithme permettant de calculer le diamètre d'un graphe donné. Analyser sa complexité.*

30. *On dit qu'un graphe* $G = (S, A)$ *est connexe à l'ordre k lorsque tous les graphes* $(S, A \setminus T)$ *sont connexes pour* T *parcourant les parties de* A *de cardinal* $|T| < k$*. On note* $\omega(G)$ *le plus grand entier* k *pour lequel* G *est connexe à l'ordre* k*. Donner un minorant et un majorant de* $\omega(G)$ *en fonction de* $|S|$ *et* $|A|$*; on pourra notamment montrer l'inégalité* $\omega(G) \leqslant 1 + 2|A|/|S|$*. Écrire un programme calculant* ω*. Démontrer sa correction et borner sa complexité.*

31 (isomorphisme). *On dit que deux graphes* (S, A) *et* (S', A') *sont isomorphes s'il existe une bijection* $f : S \to S'$ *telle que* $(x, y) \in A \Leftrightarrow (f(x), f(y)) \in A'$*. Donner un minorant et un majorant du nombre de graphes d'ordre n à isomorphisme près. Écrire un programme déterminant si deux graphes donnés sont isomorphes. Démontrer sa correction et borner sa complexité.*

32. *Soit n un entier strictement positif. Pour quelles parties* G *de* $\mathbb{Z}/n\mathbb{Z}$ *le graphe non orienté* $(\mathbb{Z}/n\mathbb{Z}, A)$ *est-il connexe avec* $A = \{x + g : x \in \mathbb{Z}/n\mathbb{Z}, g \in G\}$*? On commencera par le cas où* G *est un singleton.*

33. *Soit n un entier naturel et* $\lambda \in [0, 1]$ *un réel. On considère le graphe* (S, A) *avec* $S = \{1, \dots, n\}$ *et* A *une variable aléatoire à valeur dans* $\mathfrak{P}(S^2)$ *pour laquelle les évènements* $(x, y) \in A$ *sont tous de probabilité* λ *et indépendants lorsque* (x, y) *parcours* S^2*. Montrer que, pour tout* λ*, la probabilité que ce graphe soit connexe tend vers zéro lorsque* $n \to \infty$*. A-t-on un comportement similaire pour la probabilité que la plus grande composante connexe soit de cardinal* $\geqslant n/2$*?*

Syntaxe puis sémantique de la logique propositionnelle

34 (connecteur de Sheffer). *Si* P *et* Q *sont deux propositions on pose* $P|Q \equiv (\neg P) \vee (\neg Q)$*. Montrer les équivalences* $\neg P \equiv P|P$ *puis* $P \vee Q \equiv (P|P)|(Q|Q)$*. En déduire que toute proposition admet une forme équivalente ne faisant intervenir que des variables propositionnelles, le connecteur « | » et des parenthèses. Majorer le nombre minimum de connecteurs de Sheffer composant cette forme équivalente en fonction du nombre de connecteurs de la proposition initiale.*

35 (connecteurs universels). *Montrer que le connecteur logique non-et noté* $\overline{\wedge}$ *et défini par l'équivalence* $p \overline{\wedge} q \Leftrightarrow \neg(p \wedge q)$ *est universel, c'est-à-dire que toute proposition en deux variables équivaut à une expression ne faisant intervenir que les variables, le connecteur* $\overline{\wedge}$ *et des parenthèses. Généraliser ce résultat à un nombre arbitraire de variables. Donner un majorant du nombre de connecteurs* $\overline{\wedge}$ *nécessaire pour écrire ainsi une proposition en k variables.*

Montrer que la disjonction exclusive notée $\oplus$ *et définie par* $p \oplus q \iff (p \wedge q) \vee (\neg p \wedge \neg q)$ *n'est pas un connecteur universel.*

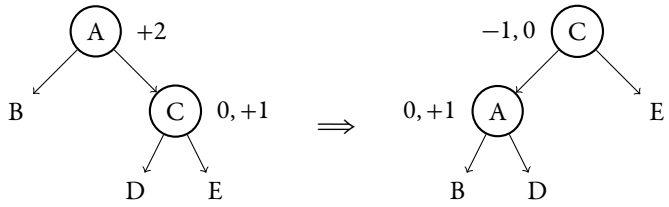


FIGURE A.2 – Équilibrage d'un nœud par rotation simple.

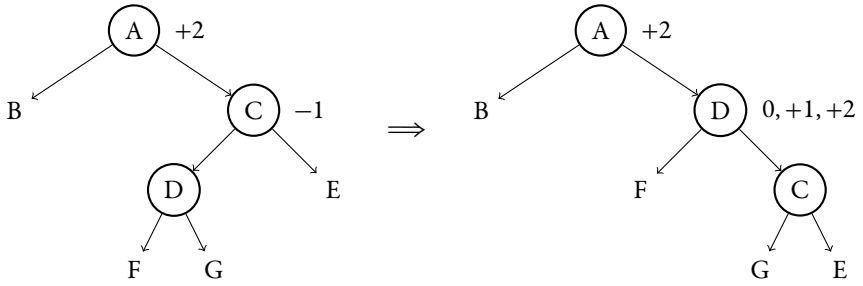


FIGURE A.3 – Équilibrage d'un nœud par rotation double. Ramène au cas de la figure A.2.

A.3 Troisième semestre

Le terme « mot » désigne ici toute chaîne de caractères.

Dictionnaires et programmation dynamique

36 (plus grand sous mot commun). *Montrer que le problème consistant à déterminer le plus grand sous mot commun de deux mots donnés ne satisfait pas la propriété de sous structure optimale. Notant $s(x, y)$ le plus grand suffixe commun aux mots x et y, montrer que pour toutes lettres σ et μ on a*

$$s(x\sigma, y\mu) = \begin{cases} 1 + s(x, y) & \text{si } \sigma = \mu, \\ 0 & \text{sinon.} \end{cases}$$

En déduire un algorithme de complexité $O(|x| \cdot |y|)$ permettant de déterminer le plus grand sous mot commun.

37 (distance de Levenstein). *On appelle distance de Levenshtein entre deux mots x et y le nombre minimal d'opérations successives à effectuer afin de transformer x en y, les opérations autorisées étant : changer une lettre en une autre, insérer une lettre, supprimer une lettre. On la note $d(x, y)$. Montrer que, pour toutes lettres σ et μ, on a l'identité*

$$d(\sigma x, \mu y) = \begin{cases} d(x, y) & \text{si } \sigma = \mu, \\ 1 + \min\{d(x, \mu y), d(\sigma x, y), d(x, y)\} & \text{sinon.} \end{cases}$$

En déduire que le problème du calcul de $d(x, y)$ satisfait la propriété de sous structure optimale puis implanter une fonction OCaml calculant cette quantité.

Applications des arbres

38. *La boulangerie de l'exercice 19 fait de nouveau polémique en adoptant une nouvelle politique de gestion des clients. Désormais, afin de s'occuper en priorité des clients lucratifs, elle stockera chaque commande en attente comme nœud d'un tas étiqueté par son montant. L'unique vendeuse enlèvera la racine, traitera la commande correspondante, puis percolera le tas avant de recommencer.*

On suppose qu'elle met une minute à traiter chaque commande et qu'un client non servi dans les cinq minutes suivant son arrivée annule sa commande et quitte l'établissement insatisfait. Avec ce système, de combien la boulangerie améliore-t-elle son chiffre d'affaire si, une heure durant, sept clients arrivent simultanément toutes les cinq minutes avec des commandes de montants respectifs 100, 100, 200, 200, 200, 200 *et* 400 *? Et pour d'autres rythmes et montants ?*

Graphes avancés

39. *Borner le nombre d'arbres couvrants d'un graphe $G = (S, A)$ en fonction de $|S|$ et $|A|$.*

40. *Démontrer que, si un graphe connexe non orienté n'admet pas deux arêtes de même poids, l'arbre couvrant de poids minimal est unique. Que dire de la réciproque ?*

41. *Soit un graphe non orienté $G = (S, A)$. Montrer que, parmi les propositions suivantes, toute paire implique la troisième : G est connexe ; G est acyclique ; $|A| = |S| - 1$.*

42 (plus petit surmot commun). *On appelle surmot d'un ensemble S de mots, tout mot admettant chaque élément de S comme facteur. Tout surmot s'obtient en concaténant les mots de S puis éliminant la redondance, c'est-à-dire tout suffixe d'un mot préfixe du suivant. Notons $m(x, y)$ la*

longueur du plus long suffixe de x qui est aussi préfixe de y. Si $(s_i)_{i\in\{1,\dots,\#S\}}$ sont les mots de S dans l'ordre de concaténation, la longueur du surmot obtenu est

$$\sum_{s\in S}|s| - \sum_{i=1}^{\#S-1}|m(s_i, s_{i+1})|$$

et le plus petit surmot s'obtient donc en maximisant la seconde somme. Montrer que cela revient à chercher, dans le graphe orienté pondéré (S, S^2, p) avec $p : (x, y) \mapsto |m(x, y)|$, un chemin Hamiltonien, c'est-à-dire passant une unique fois par chaque sommet, le plus long. La recherche de tels chemins étant NP-complet, concevoir un algorithme glouton approchant la solution : à chaque étape, sélectionner l'arête la plus longue et la rajouter au chemin en cours de construction.

Bases de données (SQL)

Déduction naturelle pour la logique propositionnelle

Algorithmique pour l'intelligence artificielle et l'étude des jeux

A.4 Quatrième semestre

Motifs, automates et expressions (OCaml)

43 (puzzle MU [28]). *Sur l'alphabet $\Sigma = \{M, I, U\}$ montrer qu'il est impossible d'obtenir le mot MU en partant du mot MI et en appliquant des transformations du type $xI \mapsto xIU$, $M \mapsto Mxx$, $xIIIy \mapsto xUy$ et $xUUy \mapsto xy$ quels que soient les mots x et y. On pourra considérer le résidu modulo trois du nombre de lettres I des mots considérés.*

44. *Soit $(f_k)_{k\in\mathbb{N}}$ la suite de mots définie par $f_{k+2} = f_{k+1}f_k$ avec $f_1 = 1$ et $f_0 = 0$. Montrer que le nombre de lettres 1 dans le mot f_k est pair si et seulement si k est divisible par trois. Pour $k \geqslant 2$, montrer que f_k admet 10 (resp. 01) comme suffixe lorsque k est pair (resp. impair). Montrer que, privé de ce suffixe, le mot f_k est un palindrome, c'est-à-dire qu'il reste inchangé si l'on écrit ses lettres dans l'ordre inverse.*

45 (mots bien parenthésés). *Soit Σ l'alphabet binaire dont les lettres sont la parenthèse ouvrante « (» et la parenthèse fermante «) ». Un mot sur Σ est dit bien parenthésés si aucun de ses préfixes ne contient plus de «) » que de « (» et si lui-même en contient autant. Par exemple, « (()) » est bien parenthésé mais pas « ())(» ni « ((() ».*

Montrer que l'ensemble $\mathscr{P}$ des mots bien parenthésés est le plus petit langage sur Σ contenant le mot vide et tel que $\forall (x, y) \in \mathscr{P}^2$, $(x)y \in \mathscr{P}$. En déduire que la suite $c_n = |\mathscr{P} \cap \Sigma^{2n}|$ satisfait la relation de récurrence $c_{n+1} = \sum_{k=0}^{n} c_k c_{n-k}$. Vérifier alors que la série formelle $f(x) = \sum_{n\in\mathbb{N}} c_n x^n$ satisfait l'équation fonctionnelle $xf(x)^2 = f(x) - 1$; en résolvant classiquement ce trinôme du second degré puis en développant en série la solution trouvée, obtenir l'identité $c_n = \frac{1}{n+1} C_{2n}^n$.

46 (arbre des suffixes). *FIXME*

47 (plus court surmot commun). *Soit $S = \{s_1, \ldots, s_n\}$ un langage fini sur un alphabet Σ. Démontrer l'existence d'un mot m de longueur minimale admettant chaque mot de S comme facteur. Pour tous mots x et y, notons $c(x, y)$ le plus long suffixe de x qui est aussi préfixe de y. En construisant des surmots comme concaténation des mots de S puis en enlevant la redondance qui apparaît entre deux mots successifs, établir l'égalité*

$$|m| = \sum_{k=1}^{n} |s_k| - \min_{\sigma\in\mathfrak{S}_n} \sum_{k=1}^{n-1} \left| c\left(s_{\sigma(k)}, s_{\sigma(k+1)}\right)\right|.$$

Montrer que déterminer une permutation maximisant cette somme revient à trouver un chemin Hamiltonien de poids maximal dans le graphe (S, S^2) muni de la fonction de poids $(x, y) \to |c(x, y)|$. Concevoir un algorithme déterminant un long chemin Hamiltonien si ce n'est le plus long.

48. *Étant donnés deux langages arbitraires L et M, déterminer toutes les relations d'inclusion entre les langages ci-dessous.*

$$L+M \quad L\cdot M \quad (L+M)^\star \quad (L\cdot M)^\star \quad L^\star + M^\star \quad L^\star \cdot M^\star \quad (L^\star + M^\star)^\star \quad (L^\star \cdot M^\star)^\star$$

49 (lemme d'Arden [14]). *Soit A et B deux langages. Montrer que le langage $A^\star B$ est la plus petite solution X de l'équation $X = AX + B$. Montrer que si $\varepsilon \notin A$ alors c'est l'unique solution.*

50. *Montrer que si L est un langage rationnel, alors le langage de ses préfixes, celui de ses suffixes et celui de ses facteurs le sont aussi.*

51. *Montrer que si L est un langage rationnel, alors $\sqrt{L} = \{m \in \Sigma^\star : m^2 \in L\}$ aussi.*

52. *Montrer que les langages rationnels sont exactement les langages reconnus par les automates dont les transitions sont étiquetées par des expressions rationnelles plutôt que par des lettres.*

53. *Sur l'alphabet $\Sigma = \{a, b\}$ notons $v_a(m)$ le nombre d'occurrences de la lettre a dans le mot m. On considère le langage $L_k = \{m \in \Sigma^\star : k | v_a(m) \wedge k | v_b(m)\}$. Donner des expressions rationnelles pour L_k et son complémentaire $\Sigma^\star \setminus L_k$ et comparer leurs longueurs. On pourra considérer le cas $k = p_1 \cdots p_n$ où p_i désigne le i^e nombre premier.*

Bibliographie

Afin de laisser transparaître la dimension historique du développement des concepts de l'informatique théorique et des notions de l'informatique pratique, les références ci-dessous sont énumérées dans l'ordre chronologique.

[1] Leonhard Euler. « Solutio problematis ad geometriam situs pertinentis ». *Commentarii Academiæ Scientiarum Imperialis Petropolitanæ* 8 (1736), pages 128-140.

[2] Kurt Gödel. « Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I ». *Monatshefte für Mathematik und Physik* 38.1 (1931), pages 173-198. doi : 10.1007/BF01700692.

[3] Gerhard Gentzen. « Untersuchungen über das logische Schließen, I ». *Mathematische Zeitschrift* 39 (1933), pages 172-210. doi : 10.1007/BF01201353.

[4] Alan Mathison Turing. « On computable numbers, with an application to the Entscheidungsproblem ». *Proceedings of the London Mathematical Society* 42.2 (1937), pages 230-265. doi : 10.1112/plms/s2-42.1.230.

[5] John von Neumann. *First Draft of a Report on the EDVAC*. Rapport technique. Contract No. W-670-ORD-4926. University of Pennsylvania, 1945.

[6] Stephen Cole Kleene. *Representation of events in nerve nets and finite automata*. Research Memorandum RM-704. Project RAND, U.S. Air Force, 1951.

[7] Joseph Bernard Kruskal. « On the shortest spanning subtree of a graph and the traveling salesman problem ». *Proceedings of the American Mathematical Society* 7 (1956), pages 48-50. doi : 10.1090/S0002-9939-1956-0078686-7.

[8] Claude Berge. « Two theorems in graph theory ». *Proceedings of the National Academy of Sciences of the United States of America* 43.9 (1957), pages 842-844. doi : 10.1073/pnas.43.9.842.

[9] John R. Myhill. *Finite automata and the representation of events*. Technical Report 57-624. Wright Air Development Division, U.S. Air Force, 1957, pages 112-137.

[10] Anil Nerode. « Linear Automaton Transformations ». *Proceedings of the AMS* 9.4 (1958), pages 541-544. doi : 10.2307/2033204.

[11] Edsger Wybe Dijkstra. « A note on two problems in connexion with graphs ». *Numerische Mathematik* 1.1 (1959), pages 269-271. doi : 10.1007/BF01386390.

[12] Bernard Roy. « Transitivité et connexité ».
Comptes rendus de l'académie des sciences de Paris 249 (1959), pages 216-218.

[13] Robert Forbes McNaughton et Hisao Yamada.
« Regular expressions and state graphs for automata ».
IRE Transactions on Electronic Computers EC-9.1 (1960), pages 39-47.
doi : DOI10.1109/TEC.1960.5221603.

[14] Dean Norman Arden. « Delayed-logic and finite-state machines ».
Switching Circuit Theory and Logical Design — SWCT 1961. IEEE, 1961,
pages 133-151. doi : 10.1109/FOCS.1961.13.

[15] Виктор Михайлович Глушков. « Абстрактная теория автоматов ».
Успехи Математических Наук 16.5 (1961), pages 3-62.

[16] Robert Willoughby Floyd. « Algorithm 97 : shortest path ».
Communications of the ACM 5.6 (1962), page 345. doi : 10.1145/367766.368168.

[17] Stephen Warshall. « A theorem on Boolean matrices ».
Journal of the ACM 9.1 (1962), pages 11-12. doi : 10.1145/321105.321107.

[18] Георгий Максимович Адельсон-Вельский et Евгений Михайлович Ландис.
« Один алгоритм организации информации ».
Доклады Академии Наук 146.2 (1962), pages 263-266.

[19] John William Joseph Williams. « Algorithm 232 : heapsort ».
Communications of the ACM 7.6 (1964), pages 347-348.
doi : 10.1145/512274.512284.

[20] Jack R. Edmonds. « Paths, trees, and flowers ».
Canadian Journal of Mathematics 17 (1965), pages 449-467.
doi : 10.4153/CJM-1965-045-4.

[21] Kenneth Lane Thompson. « Regular expression search algorithm ».
Communications of the ACM 11.6 (1968), pages 419-422.
doi : 10.1145/363347.363387.

[22] Edgar Frank Codd. « A relational model of data for large shared data banks ».
Communications of the ACM 13.6 (1970), pages 377-387.
doi : 10.1145/362384.362685.

[23] Martin Gardner. « Mathematical Games — The fantastic combinations of John
Conway's new solitaire game 'life' ». *Scientific American* 223 (1970), pages 120-123.

[24] Kenneth Appel et Wolfgang Haken.
« Every planar map is four colorable, part I : discharging ».
Illinois Journal of Mathematics 21.3 (1977), pages 429-490.
doi : 10.1215/ijm/1256049011.

[25] Kenneth Appel, Wolfgang Haken et John Koch.
« Every planar map is four colorable, part II : reducibility ».
Illinois Journal of Mathematics 21.3 (1977), pages 491-567.
doi : 10.1215/ijm/1256049012.

[26] Robert Stephen Boyer et J Strother Moore. « A Fast String Searching Algorithm ».
Communications of the ACM 20.10 (1977), pages 762-772.
doi : 10.1145/359842.359859.

[27] Donald Ervin Knuth, James Hiram Morris et Vaughan Ronald Pratt.
« Fast Pattern Matching in Strings ».
SIAM Journal on Computing 6.2 (1977), pages 323-350. doi : 10.1137/0206024.

[28] Douglas Richard Hofstadter. *Gödel, Escher, Bach. An Eternal Golden Braid*. Basic Books, 1979.

[29] IEEE. *Binary Floating-Point Arithmetic*. IEEE Standard. 1985. isbn : 0-7381-1165-1. doi : 10.1109/IEEESTD.1985.82928.

[30] Gérard Berry et Ravi Sethi. « From regular expressions to deterministic automata ». *Theoretical Computer Science* 48 (1986), pages 117-126. doi : 10.1016/0304-3975(86)90088-5.

[31] Guido van Rossum et al. *Python Documentation*. Python Software Foundation. 1990. url : https://docs.python.org/.

[32] Valentin Antimirov. « Partial derivatives of regular expressions and finite automaton constructions ». *Theoretical Computer Science*. Tome 155. 2. 1996, pages 291-319. doi : 10.1016/0304-3975(95)00182-4.

[33] Xavier Leroy, Jérôme Vouillon, Damien Doligez, Didier Rémy et Ascánder Suárez. *OCaml*. An industrial strength programming language supporting functional, imperative and object-oriented styles. INRIA. 1996. url : http://ocaml.org/.

[34] Neil Robertson, Daniel Sanders, Paul Seymour et Robin Thomas. « The Four-Colour Theorem ». *Journal of Combinatorial Theory, Series B* 70.1 (1997), pages 2-44. doi : 10.1006/jctb.1997.1750.

[35] Georges Gonthier. « Formal proof, the four-color theorem ». *Notices of the American Mathematical Society* 55.11 (2008), pages 1382-1393.

[36] Luis Rocha. *Chinook Sample Database*. 2008. url : https://github.com/lerocha/chinook-database.

[37] Richard Peirce Brent et Paul Zimmermann. *Modern Computer Arithmetic*. Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010. isbn : 0-521-19469-5.

[38] *Classes préparatoires aux grandes écoles. Programme d'informatique commune*. Arrêté du 5 janvier 2021. Ministère de l'enseignement supérieur, 7 fév. 2021. url : https://www.enseignementsup-recherche.gouv.fr/pid20536/bulletin-officiel.html?cid_bo=156662.

[39] *Classes préparatoires aux grandes écoles. Programme de l'option informatique en MPSI/MP*. Arrêté du 5 janvier 2021. Ministère de l'enseignement supérieur, 7 fév. 2021. url : https://www.enseignementsup-recherche.gouv.fr/pid20536/bulletin-officiel.html?cid_bo=156675.