Informatique

Gaetan Bisson

https://gaati.org/bisson/

Introduction

Ce cours aborde certains fondamentaux de l'informatique qui relèvent essentiellement de l'algorithmique et des structures de données. Différents concepts théoriques sont abordés en s'appuyant sur la pratique de la programmation dans les langages Python, Caml et SQL.

Cet enseignement unifie deux matières du programme officiel des classes préparatoires, *l'informatique générale* [33] et *l'option informatique* [34] qui font l'objet d'une épreuve commune à certains concours et d'épreuves séparées à d'autres. Le chapitre 1 puis les sections 3.1 et 3.5 relèvent de la première matière et le reste de ce document de la seconde.

Table des matières

Chapitre 1

Premier semestre

Avant de commencer. *Lorsqu'un compte informatique vous est octroyé, la première chose à faire est d'en changer le mot de passe : connectez vous pour ce faire au serveur avec l'application Putty puis lancez la commande* `passwd`.

Ce semestre est consacré à la prise en main de l'outil informatique. Il consiste en un socle de connaissances générales nécessaires à la pratique de la programmation impérative. Ces techniques seront alors appliquées à divers problèmes relevant notamment des structures de données élémentaires ou encore du calcul numérique.

1.1 Introduction

Une fois replongés dans l'univers de la programmation, nous présenterons synthétiquement les couches matérielles et logicielles sous-jacentes à notre environnement informatique, à savoir l'ordinateur et son interface utilisateur.

1.1.0 Prélude

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 tous les langages demandés. Connectez-y vous et créer une nouvelle feuille de calcul de type Python. Tapez alors votre premier 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)
```

Ce programme comporte trois boucles imbriquées, chacune ayant pour effet de faire prendre successivement les valeurs $1, 2, \dots, 19$ $(= 20 - 1)$ à sa variable ; au cœur de ces trois boucles on teste donc l'égalité $x^2 + y^2 = z^2$ pour tous les triplets $(x, y, z) \in \{1, 19\}^3$ et affiche ceux pour lesquels elle est vérifiée.

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 : c'est ainsi que les trois boucles ci-dessus sont identifiées comme imbriquées. Pour éviter des contresens majeurs il vous faudra être particulièrement vigilant.*

Essayer maintenant de comprendre ce que fait ce second programme :

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

Il crée une variable n, lui donne la valeur 1, puis la multiplie par k pour k prenant successivement les valeurs $1, 2, \ldots, 60$; lorsque ce programme termine on a donc $n = 60!$ et cette quantité s'affiche alors à l'écran.

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

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

Exercice. *Déterminer tous les entiers entre* 1 *et* 100 *qui ne peuvent pas s'écrire comme la somme de deux carrés non nuls. Que remarquez-vous ?*

Votre aptitude à la programmation est non seulement évaluée aux concours, c'est le socle pratique sur lequel tous 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 autant que possible : réécrivez les programmes vus en cours, modifiez les, améliorez les, créez en de nouveaux, etc.

1.1.1 Ordinateurs

Quel type de machine peut exécuter les programmes ci-dessus ? Wikipedia définit un ordinateur comme « une machine électronique programmable qui fonctionne par lecture séquentielle d'un ensemble d'instructions, organisées en programmes, qui lui font exécuter des opérations logiques et arithmétiques. »

Dans la société moderne, les ordinateurs sont omniprésents : on a évidemment les ordinateurs portables, de bureau et serveurs, mais aussi les tablettes, téléphones, montres, GPS, etc. La plupart sont livrés avec un ensemble de programmes (appelé système d'exploitation) présentant une interface conviviale permettant de réaliser les taches les plus courantes ; la grande majorité des utilisateurs s'en contente : ils pratiquent la bureautique, non la programmation.

D'un point de vue purement matériel, un ordinateur se compose :

- **d'une unité centrale** contenant :
  - le processeur, qui exécute les opérations proprement dites
  - la carte mère, qui relie le processeur aux autres composants
  - la mémoire (vive et dure), qui stocke l'information
  - l'alimentation, qui distribue l'électricité aux différents composants
  - les ports de communication, qui relient la carte mère aux périphériques
- **de périphériques d'entrée–sortie** dont typiquement :
  - un écran
  - un clavier (tactile ou non)
  - une carte réseau (filaire ou wifi)
  - une carte son
  - une caméra

Chaque composant n'est qu'un simple automate : il reçoit des instructions et des données, les traite, ceci résultant éventuellement en l'envoi de nouvelles instructions et données à d'autres composants. Les informations (instructions et données) sont transmises sous la forme de

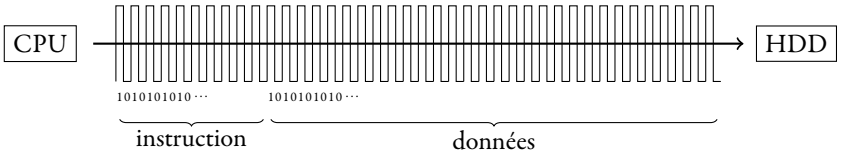


FIGURE 1.1 – Processeur transmettant une instruction d'écriture de données au disque dur.

PRÉFIXE	SYMBOLE	QUANTITÉ
kilo	k	10^3
méga	M	10^6
giga	G	10^9
téra	T	10^{12}

FIGURE 1.2 – Préfixes utilisés couramment en informatique.

signaux électriques. Ces signaux analogiques sont interprétés comme une suite de bits, unité élémentaire d'information numérique pouvant prendre les valeurs 0 ou 1. Voir la figure 1.1.

Les bits, ainsi nommés par contraction de *binary digit*, viennent généralement par paquets de huit, appelés octets. Un octet peut donc prendre $2^8 = 256$ valeurs possibles. Pour dénoter de grandes quantités d'octets, on utilise les préfixes du système international d'unités (SI) rappelés sur la figure 1.2. On pourra notamment garder en tête les ordres de grandeurs suivants pour un ordinateur de bureau :

— mémoire vive : 10 Go
— mémoire dure : 1 To
— débit réseau : 1 Mo/s
— processeur : 1 Gop./s

Un programme n'est autre qu'une suite d'instructions destinées au processeur. Parmi les instructions dont disposent les processeurs, on distingue trois grandes familles, chacune opérant sur des *mots machines* composés de 64 bits (soit huit octets) :

— **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 $\mathbb{R}$.

Ces deux dernières familles nécessitent de coder des nombres (entiers ou réels) comme éléments de $\{0,1\}^{64}$. Elles ne peuvent donc réaliser qu'une imparfaite approximation de l'arithmétique des ensembles infinis que sont $\mathbb{Z}$ et $\mathbb{R}$. Nous allons en discuter.

1.1.2 Nombres entiers

Afin de coder les nombres entiers en signaux numériques, c'est-à-dire comme suite de bits, la technique de la représentation binaire s'impose naturellement.

Définition. *On appelle représentation binaire (ou écriture binaire) d'un entier naturel $n \in \mathbb{N}$ l'unique famille $(c_0, c_1, \ldots, c_{k-1}) \in \{0,1\}^k$ vérifiant $c_{k-1} = 1$ et*

$$n = c_0 2^0 + c_1 2^1 + \cdots + c_{k-1} 2^{k-1} = \sum_{i=0}^{k-1} c_i 2^i .$$

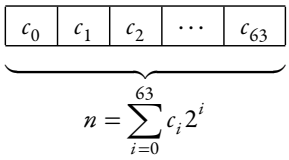


FIGURE 1.3 – Entiers de type `uint64`.

Si la mémoire peut être supposée infinie, les processeurs n'opèrent que sur des mots de 64 bits et ne peuvent donc manipuler que des entiers naturels n pour lesquels $k \leqslant 64$ soit $n \in \{0, \ldots, 2^{64} - 1\}$; ils sont codés suivant le type `uint64` illustré par la figure 1.3. Lorsque le résultat d'une opération arithmétique dépasse $2^{64} - 1$, c'est son reste modulo 2^{64} que l'on obtient ; autrement dit, c'est l'arithmétique de $\mathbb{Z}/2^{64}\mathbb{Z}$ et non celle de $\mathbb{Z}$ qui est réalisée. Le langage Caml utilise directement les entiers du processeur, ce qui donne :

```
# 4*4611686018427387904 ;;
- : 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 $(2^{64} - n)$ sinon ; l'entier zéro n'a ainsi qu'une seule représentation et on peut effectuer les opérations d'addition, de soustraction et de multiplication sur le type `int64` avec le même circuit que pour `uint64`. Ce codage est évidemment sujet aux comportements que l'on pourra observer en Caml :

```
# 4611686018427387904 ;;
- : int = -4611686018427387904
```

Les interfaces de plus haut niveau peuvent 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

Ce dernier cas utilise une technique appelée arithmétique multiprécision qui demande un travail logiciel significatif ; c'est le choix fait par Python. En revanche, Caml adhère à la première option et la majorité des calculatrices à la seconde.

1.1.3 Nombres réels

Le codage des nombres réels le plus répandu est régit par le standard IEEE 754 [25] 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 1.4.

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

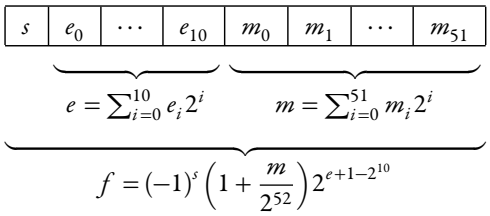


FIGURE 1.4 – Réels de type `binary64`.

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.
False
>>> 10.**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.*

1.1.4 Systèmes d'exploitation

Deux grandes familles de systèmes d'exploitation règnent aujourd'hui : les descendants de CP/M qui comptent notamment QDOS puis Windows et les descendants d'UNIX parmi lesquels figurent BSD, Linux, MacOS et Android. Avant leur invention dans les années soixante, les ordinateurs ne pouvaient exécuter qu'un programme à la fois et ce dernier devait s'adresser directement au processeur pour toute opération. Cette situation rigoureuse a été dépassée grâce à quelques innovations matérielles mais surtout de grandes innovations logicielles.

Définition. *Un système d'exploitation est une collection de logiciels avec pour fonctions principales :*
- *De présenter aux utilisateurs une interface indépendante du matériel sous-jacent.*
- *D'allouer équitablement les ressources matérielles à de multiples utilisateurs.*

Autrement dit, un système d'exploitation présente aux utilisateurs un environnement idéalisé et se charge entièrement de le simuler sur le matériel sous-jacent. Chaque aspect de cet environnement n'est donc qu'une *abstraction logicielle*, dont on peut citer notamment :
- **la notion de processus :** programme en cours d'exécution ; possibilité d'interrompre puis reprendre l'exécution, multiples processus en parallèle sur un processeur.
- **la notion de fichier :** unité de stockage ; possibilité de lire, écrire, copier, renommer, supprimer ; organisé en arborescence de répertoires.
- **la notion de droit :** limitation des accès et opérations de certains utilisateurs vis-à-vis de certains processus et fichiers.

Les notions de droit et de processus sont hors programme mais pas celle de fichier qui doit impérativement être maîtrisée.

Les fichiers forment les feuilles d'une arborescence de répertoires, c'est-à-dire que leur organisation est hiérarchique et que les fichiers en sont les terminaisons. Ce concept perdure depuis bientôt soixante ans [16] et s'est même généralisé à d'autres adressages telles les URL.

On spécifie l'emplacement d'un fichier ou répertoire par un chemin, c'est-à-dire une suite de répertoires, séparés par le caractère `/`, chacun contenant son successeur et finissant par le fichier ou répertoire dont il est question ; on écrira donc :

le répertoire `/nombres/premiers/`
contient le répertoire `/nombres/premiers/impairs/`
ainsi que le fichier `/nombres/premiers/deux`

Les chemins ci-dessus sont *absolus* car ils partent du répertoire racine ; on peut aussi spécifier des chemins *relatifs*, c'est-à-dire partant du répertoire courant. Le répertoire courant est souvent `/home/user/` où `user` est votre nom d'utilisateur ; ces chemins désignent alors le même fichier :

- `info/seance4.py`
- `/home/user/info/seance4.py`
- `info/./seance4.py`
- `info/../info/seance4.py`
- `/home/user/./info/seance4.py`
- `/home/user/../user/info/seance4.py`

Les quatre derniers chemins utilisent les répertoires spéciaux « `.` » et « `..` » qui désignent respectivement le répertoire actuel et son père.

Remarque. *Ce principe régit aussi les URL (adresses Internet) à l'ajout près du protocole ; on écrira donc* `file:///nombres/premiers/deux`. *La syntaxe DOS commence quant à elle par l'identifiant de l'unité de stockage puis utilise le caractère* `\` *et donnerait donc* `C:\nombres\premiers\deux`. *Enfin, pour aider certains programmes à identifier la nature des fichiers, on ajoute parfois au nom un suffixe appelé* extension *et commençant par un point, par exemple* `texte.txt` *ou encore* `image.jpeg`.

La procédure pour manipuler des fichiers est relativement indépendante du langage de programmation choisi : ou ouvre d'abord le fichier, en précisant si c'est pour le lire, l'écrire, le créer, etc ; on effectue alors les opérations proprement dites ; et on ferme enfin le fichier. En Python on a par exemple :

```
f = open("bidule.txt", "x")
f.write("123\n")
f.write("abc\n")
f.close()
f = open("bidule.txt", "r")
f.read()                    # '123\nabc\n'
f.close()
f = open("bidule.txt", "r")
for line in f:
    print(line, end='')
```

La fonction `open()` accepte notamment comme second argument les modes `"r"` (lecture), `"w"` (écriture) et `"x"` (création et écriture).

1.2 Algorithmique et programmation I (Python)

1.2.1 Expressions et instructions

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 (entier, réel, caractère, etc.) contraignant ses valeurs et les opérations qu'on peut lui appliquer. Il est de surcroit un langage dit *orienté objet*, nous le verrons ainsi parfois appeler *objets* ses variables et *classes* ses types.

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

On dispose notamment des types suivants qui sont qualifiés de *simples* car ils correspondent directement aux fonctionnalités offertes par le matériel. Nous verrons plus tard qu'ils s'opposent aux *structures de données* qui demandent un travail important côté logiciel (par exemple les ensembles `set` ou encore les dictionnaires `dict`).

`bool` : les booléens `True` et `False`

`int`, `float`, `complex` : les nombres dont `1`, `1.1` et `1+1j`

`list`, `tuple`, `range` : les familles telles `[1,2]`, `(1,2)`, et `range(1,2)`

`str` : les chaînes de caractères comme `"allô?"`

Le type d'une valeur est parfois implicitement converti afin de permettre l'évaluation d'une expression. On prendra garde aux conséquences surprenantes que cela peut avoir, notamment :

```
1+1.1      # 2.1
not 1      # False
0<True     # True
False*[1]  # []
```

Afin d'apprendre les usages et spécificités de chaque type au delà du survol qu'effectue ce cours, nous invitons le lecteur à recourir librement à la documentation officielle du langage [27] et en l'occurrence à son chapitre *Built-in Types* consultable à l'adresse :

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

On qualifiera 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. Tous les exemples ci-dessus sont des expressions à l'exception de l'affectation. On a encore :

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

Les opérations usuelles comportent notamment, pour les nombres :

```
1+2+3   # 6
1+2*3   # 7
1+2**3  # 9
1+2/3   # 1.6666666666666665
1+2//3  # 1
1+2%3   # 3
```

Pour les booléens :

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

Pour les familles (et pareillement les chaînes de caractères) :

```
[1,2][0]    # 1
len([1,2])  # 2
[1,2]+[3]   # [1, 2, 3]
2*[1,2]     # [1, 2, 1, 2]
```

La programmation impérative se caractérise par l'emploi des instructions `if`, `for` et `while`.

Les branchements conditionnels. Ils exécutent un bloc d'instructions uniquement lorsqu'une expression booléenne donnée est vraie. Leur structure générale est :

```
if expression_booleenne:
        instruction_si_vraie_1
        instruction_si_vraie_2
else:
        instruction_si_fausse_1
        instruction_si_fausse_2
instruction_suivante
```

Rappelons à nouveau qu'en Python c'est l'indentation seule qui délimite les blocs d'instructions ; on y portera une attention toute particulière pour éviter les faux sens.

Les instructions itératives. Communément appelées *boucles*, elles exécutent un même bloc de code pour plusieurs valeurs d'une variable. La boucle « `for x in L:` » exécute le bloc qui la suit en donnant successivement à la variable `x` les valeurs de la liste `L`. Par exemple, pour afficher les carrés des entiers de 1 à 9 :

```
for i in [1,2,3,4,5,6,7,8,9]:
    print(i**2)
```

Au lieu d'une liste explicite, on utilisera souvent la fonction `range(a,b)` qui énumère les entiers $\{a, a+1, \ldots, b-1\}$, avec $a = 0$ lorsque cet argument est omis. Pour calculer la factorielle de 42 :

```
f=1
for i in range(1,43):
    f=f*i
```

La boucle `while` exécute quant à elle le bloc qui la suit tant que la condition est satisfaite. Par exemple, pour calculer la partie entière du logarithme en base deux de 1234 :

```
r=1234
n=0
while r>1:
    r=r//2
    n=n+1
```

Combinons ces instructions afin de calculer les nombres premiers inférieurs à n.

```
for k in range(2,n+1):
    premier=True
    for d in range(2,k):
        if k%d==0:
            premier=False
    if premier:
        print(k)
```

On aimerait pouvoir interrompre les tests `k%d==0` dès qu'on a trouvé un diviseur ; c'est justement pour cela que l'instruction `break` existe : elle interrompt l'exécution de la dernière boucle lancée. On pourrait donc écrire :

```
for k in range(2,n+1):
    premier=True
    for d in range(2,k):
        if k%d==0:
            premier=False
            break
    if premier:
        print(k)
```

1.2.2 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. L'instruction `return` renvoi le résultat et termine l'exécution de la fonction.

L'un des grands avantages de cette construction est 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'est pas définie en dehors de celle-ci. On peut donc réutiliser les noms typiques `i`, `k`, `n` et autres sans craindre que cela porte à confondre des variables déclarées dans des fonctions distinctes.

Exercice. *Écrire des fonctions qui calculent :*
- *la factorielle ;*
- *la suite de Fibonacci ;*
- *la division Euclidienne ;*
- *le plus grand diviseur commun ;*
- *le plus petit multiple commun ;*
- *la liste des diviseurs d'un entier.*

L'usage de fonctions est primordial en programmation : cela permet de partitionner une quantité de code arbitrairement grande en une collection de procédures intelligibles dont l'interaction est claire. C'est là l'objectif officiel de votre apprentissage de la programmation que d'écrire des ensembles de fonctions :
- **modulaires,** c'est-à-dire petites et réalisant chacune une fonctionnalité bien définie, quitte à en combiner plusieurs pour répondre à un problème complexe ;

— **structurées,** c'est-à-dire claires et faciles à comprendre, présentant un flux de contrôle aussi simple que possible ;
— **documentées,** c'est-à-dire faisant très librement usage de commentaires afin d'en rendre la lecture limpide à tout autre programmeur.

Ces vertus seront évidemment appréciées des correcteurs mais vous permettront surtout d'avancer plus rapidement dans votre maitrise de la programmation.

Remarque. *En Python comme dans la majorité des langages de programmation on peut écrire des fonctions dites récursives, c'est-à-dire s'appelant elles-mêmes ; par exemple :*

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

La programmation récursive facilite grandement l'implantation de nombreuses notions mathématiques. Elle rend en contrepartie leur exécution significativement plus complexe :
— *Les appels récursifs peuvent ne pas terminer. Considérer par exemple l'évaluation de* `factorielle(-1)` *et* `factorielle(3/2)`*.*
— *La gestion de ces appels a un coût.*

On préfère en règle générale écrire des fonctions impératives plutôt que récursives lorsque cela ne présente aucune difficulté supplémentaire. Les épreuves écrites des concours imposent habituellement de programmer impérativement en Python et récursivement en Caml ; on gardera donc les techniques de programmation récursive pour le second semestre.

1.2.3 Structures de données

Les types simples présentés plus haut sont essentiellement des fonctionnalités du processeur que le langage nous présente telles quelles. Python fait néanmoins quelques efforts pour pallier les limitations du matériel, notamment en ce qui concerne le type `int`.

Les types avancés font quant à eux l'objet d'une implantation logicielle significative, ce qui leur permet de réaliser des structures de données non triviales comme les listes ou encore les ensembles. Soyons donc conscient qu'une instruction Python faisant intervenir l'un de ces types peut dérouler des milliards d'instructions processeurs, avec les conséquences que cela implique en terme de temps de calcul. Il sera particulièrement important de garder cela à l'esprit lorsque nous étudierons la complexité des algorithmes au second semestre.

Tableaux. Un tableau est une famille finie d'objets de même type stockés en mémoire à des emplacements contigus comme l'illustre la figure 1.5. Un tableau est identifié par : le type de ses éléments, l'adresse mémoire du premier élément et le nombre total d'éléments, appelé longueur du tableau. On peut accéder directement à l'élément d'indice `i` en allant à l'adresse

```
tableau[i] == tableau[0] + i * sizeof(type)
```

mais on ne peut facilement ni rajouter ni enlever des éléments à un tableau.

Listes. Une liste est une famille finie d'objets de même type stockés en mémoire à des emplacements arbitraires, chaque élément indiquant l'adresse de son successeur comme l'illustre la figure 1.6 ; cette construction précise porte le nom de *liste chaînée*. Une liste est identifiée par : le type de ses éléments et l'adresse mémoire du premier élément. Afin d'accéder au k^{e} élément on doit ainsi parcourir les $k-1$ premiers, mais cela permet de rajouter en d'enlever facilement des éléments à un emplacement arbitraire dans la liste.

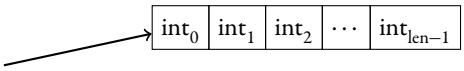


FIGURE 1.5 – Un tableau d'entiers.

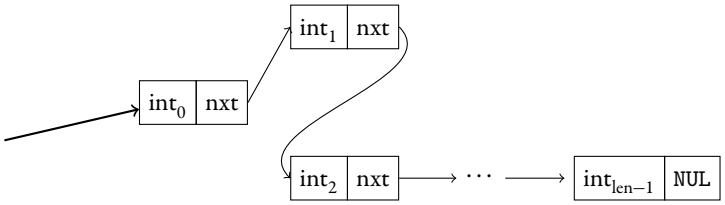


FIGURE 1.6 – Une liste chaînée d'entiers.

Chaînes de caractères. En première approximation, un caractère n'est autre qu'un octet interprété comme l'indique la figure 1.7. (C'était historiquement le cas avant que le support des caractères accentués et non latins complique la situation.) Une chaîne de caractères est alors tout simplement un tableau de caractères.

En Python. Ce langage choisi d'offrir par le biais de son type `list` une structure de donnée sensiblement plus complexe que celles présentées ci-dessus mais présentant certains avantages à la fois des tableaux et des lites : on peut directement accéder, rajouter ou enlever un élément d'indice arbitraire. (Attention, ce ne sera pas le cas en Caml qui possède deux types bien distincts correspondant respectivement aux tableaux et aux listes élémentaires ci-dessus.)

```
t=[1,2,3.4]
len(t)          # 3
t[0]            # 1
t[3]            # IndexError
t[1]=0
t               # [1, 0, 3.4]
t.insert(1,2)
t               # [1, 2, 0, 3.4]
t.append(5)
t               # [1, 2, 0, 3.4, 5]
t.pop(1)        # 1
t               # [1, 0, 3.4, 5]
```

Les chaînes de caractères se manipulent pareillement :

```
s="coucou c'est moi"
len(s)              # 16
s[15]               # 'i'
s[7:12]             # "c'est"
s=="coucou"         # False
s+" encore"         # "coucou c'est moi encore"
```

Précisons que les listes sont grandement mises en avant par le langage Python : on dispose notamment du constructeur suivant qui permet de les manipuler avec une aisance fort appréciable.

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
10	LF	SUB	*	:	J	Z	j	z
11	VT	ESC	+	;	K	[	k	{
12	FF	FS	,	<	L	\	l	|
13	CR	GS		=	M	]	m	}
14	SO	RS	.	>	N	^	n	~
15	SI	US	/	?	O	_	o	DEL

FIGURE 1.7 – Le codage de caractères ASCII : le caractère d'abscisse i et d'ordonnée j est codé par l'octet $16i + j$; les octets compris entre 128 et 255 sont réservés pour des extensions.

```
[x**2 for x in range(1,17)]
[x**2 for x in range(1,17) if x%2==1]
```

Exercice. *Trivialiser les programmes écrits jusqu'à maintenant en utilisant ce constructeur.*

En remplaçant les crochets « `[` » et « `]` » par les accolades « `{` » et « `}` », on obtient des structures de données bien connues mais moins évidentes à implanter ; nous y reviendrons mais mentionnons dès maintenant qu'il s'agit, d'une part, des ensembles.

```
a={1,2,3,1}
len(a)                                  # 3
a.union({4})                            # {1, 2, 3, 4}
{ k%10 for k in range(1,100) if k%6==0 }  # {0, 2, 4, 6, 8}
```

Et, d'autre part, des dictionnaires, qui étendent les ensembles en associant à chaque élément un objet arbitraire ; cela réalise donc la notion mathématique de fonction sur un ensemble fini.

```
s={k**2:k for k in range(0,10)}
s[25]                           # 5
s[27]                           # KeyError
```

1.2.4 Algorithmique

Exercice. *Écrire une fonction* `appartient(objet,liste)` *renvoyant* `True` *si l'objet est élément de la liste et* `False` *sinon. Écrire une fonction* `indice(objet,liste)` *renvoyant un indice auquel on trouve l'objet dans la liste si c'en est un élément et* `-1` *sinon. Écrire une fonction* `indices(objet,liste)` *renvoyant la liste des indices auxquels on trouve l'objet dans la liste.*

En Python on a les expressions « `objet in liste` » et `liste.index(objet)`.

Exercice. *Écrire une fonction `maximum(liste)` calculant le plus grand élément d'une liste d'entiers. Faire de même pour le plus petit élément.*

En Python on a les fonctions `min()` et `max()`.

Exercice. *Écrire une fonction `moyenne(liste)` calculant la moyenne des éléments d'une liste de réels. Faire de même pour la variance.*

En Python on a la fonction `sum()`.

Appliquer maintenant à votre liste de test la méthode `liste.sort()` ; elle est désormais triée. Cela permet de réaliser certains opérations beaucoup plus rapidement, notamment la recherche d'éléments, en utilisant une technique connue sous le nom de dichotomie.

Exercice. *Supposant que la liste donnée est triée, écrire des versions plus efficaces des fonctions `appartient(objet,liste)`, `indice(objet,liste)` et `indices(objet,liste)`.*

Les chaînes de caractères ne sont qu'une légère variation sur le thème des listes.

Exercice. *Écrire une fonction `sousmot(mot,chaine)` déterminant si le mot apparaît dans la chaîne. Écrire ensuite une fonction `indice(mot,chaine)` renvoyant le plus petit indice auquel le mot apparaît dans la chaîne (et `-1` ou `None` s'il n'apparaît pas). Écrire enfin une fonction `indices(mot,chaine)` renvoyant la liste des indices auxquels le mot apparaît dans la chaîne.*

En Python on a les expressions « `mot in chaine` » et `chaine.index(mot)`.

1.3 Ingénierie numérique et simulation (Python)

Nous allons maintenant présenter les principales extensions de Python qui en font un langage très réputé pour le calcul numérique et plus particulièrement ses applications aux sciences molles.

1.3.1 Bibliothèques de calcul scientifique

On appelle bibliothèque logicielle tout ensemble de programmes destinés à être utilisés comme sous routines d'autres programmes. Votre navigateur Web utilise par exemple les bibliothèques `freetype2` et `openssl` respectivement pour l'affichage des polices de caractères et la sécurisation des données. Il en va de même du panneau de configuration ainsi que d'innombrables autres applications.

En Python, on charge une bibliothèque 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` », voire même toutes les fonctions de la bibliothèque avec « `from library import *` ».

La bibliothèque NumPy. Elle offre des fonctionnalités en calcul numérique ; remarquons d'abord qu'elle fournit certaines constantes et fonctions mathématiques élémentaires.

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

Au cœur de NumPy se trouve le type `np.ndarray` qui implémente des tableaux multidimensionnels. Contrairement au type hybride `list`, il s'agit là de véritables tableaux avec l'efficacité que cela implique lorsque les éléments sont même type et les dimensions fixées ; ces informations se retrouvent respectivement dans les attributs `dtype` et `shape`. Le constructeur `np.array()` permet de convertir une liste en tableau ; on dispose aussi de constructeurs spécifiques. Créeons par exemple un tableau à trois colonnes, deux lignes et deux étages avec coefficients nuls.

```
>>> M=np.zeros((2,2,3))
>>> M[0][1][1]=17
>>> M[(1,0,1)]=42
>>> M
array([[[ 0.,  0.,  0.],
        [ 0., 17.,  0.]],

       [[ 0., 42.,  0.],
        [ 0.,  0.,  0.]]])
>>> M.dtype
dtype('float64')
>>> M.shape
(2, 2, 3)
```

Ce type effectue la majorité des opérations classiques coefficient par coefficient :

```
M=np.array([[1,2,3],[4,5,6],[7,8,9]])
M+M
3*M
M*M
np.log(M)
```

Des méthodes explicites implantent de surcroit certaines opérations matricielles :

```
np.dot(M,M)
np.linalg.det(M)
np.linalg.eig(M)
```

NumPy possède enfin bien d'autres sous bibliothèques telle `random` :

```
np.random.randint(a,b)  # entier de [a,b[
np.random.random()      # réel de [0,1[
```

La bibliothèque SciPy. Elle s'appuie sur NumPy pour construire un environnement de calcul scientifique offrant notamment des fonctionnalités en algèbre linéaire, statistiques, traitement du signal, etc. Nous reviendrons sur ses usages lorsque cela sera pertinent et nous contentons ici d'en donner deux exemples. Premièrement, pour le calcul d'intégrales :

```
import numpy as np
from scipy import integrate
f=lambda x : np.exp(-x**2)
integrate.quad(f,-9,9)
```

Deuxièmement, pour la recherche de zéros :

```
import numpy as np
from scipy import optimize
g=lambda x : x*np.exp(x)-1
optimize.newton(g,0)
```

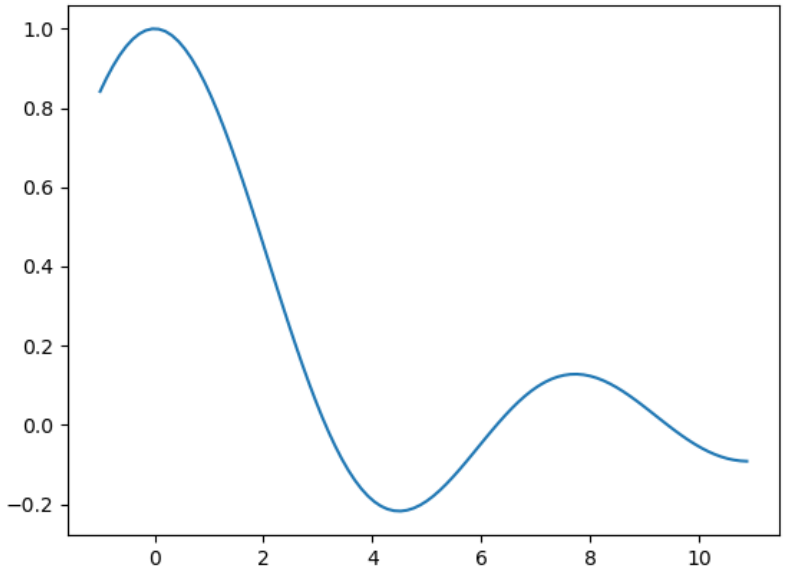


FIGURE 1.8 – Objet renvoyé par Matplotlib.

La bibliothèque Matplotlib. Elle permet de tracer des graphiques. Ses fonctionnalités pourront notamment s'avérer utiles pour vos TIPE et pendant certaines épreuves orales des concours.

```
import numpy as np
import matplotlib.pyplot as plt
f=lambda x : np.sin(x)/x
n=100
X=[-1+12*k/n for k in range(n)]
Y=[f(x) for x in X]
plt.plot(X,Y)
plt.show()
```

On obtient alors l'objet de type `matplotlib.lines.Line2D` de la figure 1.8.

Exercice. *Tracer pareillement le graphe de la fonction* $\begin{cases} [-1,1] \longrightarrow \mathbb{R} \\ x \longmapsto \sqrt{1-x^2} \end{cases}$.

Calculer une valeur approchée de son intégrale.

Rappelons enfin que ce cours n'a aucunement vocation à se substituer à la documentation officielle du langage : c'est elle qui fait autorité et on ne peut qu'en louer la clarté. En l'occurrence, on consultera avec profit :

— NumPy : `https://numpy.org/doc/`
— SciPy : `https://docs.scipy.org/doc/`
— Matplotlib : `https://matplotlib.org/contents.html`

1.3.2 Sommation sur un segment

On considère ici le problème consistant à calculer une valeur approchée de l'intégrale d'une fonction réelle f donnée sur un segment $[a,b]$ donné. Une première approche, populairement baptisée *méthode des rectangles*, repose sur le théorème suivant vu en cours d'analyse.

Théorème (sommes de Riemann). *Soit f une fonction de classe $\mathscr{C}^1([a,b],\mathbb{R})$. On a*

$$\left|\varepsilon\sum_{k=0}^{\left\lfloor\frac{b-a}{\varepsilon}\right\rfloor} f(a+k\varepsilon)-\int_a^b f(x)dx\right| \leqslant \varepsilon\left(\frac{b-a}{2}\|f'\|_\infty+\|f\|_\infty\right)$$

c'est-à-dire que la somme tend vers l'intégrale avec un terme d'erreur en $O(\varepsilon)$.

Démonstration. Pour tout $\alpha\in[a,b]$ on a

$$\begin{aligned}\left|\varepsilon f(\alpha)-\int_\alpha^{\alpha+\varepsilon} f(x)dx\right| &= \left|\int_\alpha^{\alpha+\varepsilon}(f(\alpha)-f(x))\,dx\right|\\ &\leqslant \int_\alpha^{\alpha+\varepsilon}\|f'\|_\infty|\alpha-x|\,dx\\ &\leqslant \|f'\|_\infty\frac{\varepsilon^2}{2}\end{aligned}$$

et il ne reste qu'à sommer ces majorations pour $\alpha=a+k\varepsilon$ lorsque k parcours $\{0,\dots,n\}$ avec $n=\left\lfloor\frac{b-a}{\varepsilon}\right\rfloor$ puis d'y ajouter le terme en $\|f\|_\infty$ qui borne l'intégrale restante $\int_{a+n\varepsilon}^b f(x)dx$. □

Remarque. *Si f est seulement supposée continue alors la somme de Riemann converge toujours vers l'intégrale (en utilisant l'uniforme continuité plutôt que les accroissements finis) mais on ne peut pas exprimer facilement le terme d'erreur ; ce cas est donc d'intérêt moindre en calcul numérique.*

Exercice. *Écrire une fonction* `integrale(f,a,b)` *calculant une approximation de l'intégrale de la fonction réelle f sur le segment $[a,b]$ en utilisant la méthode des rectangles. Déduire une valeur approchée de π en utilisant $f(x)=\sqrt{1-x^2}$.*

Pour raffiner cette technique d'intégration numérique on peut approcher f sur $[\alpha,\alpha+\varepsilon]$ non pas par la constance $f(\alpha)$ mais par la fonction affine

$$x\longmapsto f(\alpha)+\frac{x-\alpha}{\varepsilon}(f(\alpha+\varepsilon)-f(\alpha));$$

on obtient ainsi la méthode dite *des trapèzes*.

Théorème. *Soit f une fonction de classe $\mathscr{C}^2([a,b],\mathbb{R})$. On a*

$$\left|\varepsilon\sum_{k=0}^{\left\lfloor\frac{b-a}{\varepsilon}\right\rfloor}\left(\frac{f(a+k\varepsilon)+f(a+(k+1)\varepsilon)}{2}\right)-\int_a^b f(x)dx\right| = O(\varepsilon^2).$$

Remarque. *La somme ci-dessus est fortuitement identique à celle correspondant à la méthode des rectangles, aux termes en $f(\alpha)$ et $f(\alpha+n\varepsilon)$ près. Ces termes améliorent donc à eux seuls la convergence de linéaire en quadratique. La méthode des trapèzes forme le premier échelon d'une famille de méthodes consistant à approcher f par des polynômes de degré k fixé : pour $k=0$ on a la méthode des rectangles, pour $k=1$ celle des trapèzes et pour $k=2$ celle des paraboles communément attribuée à Simpson, etc.*

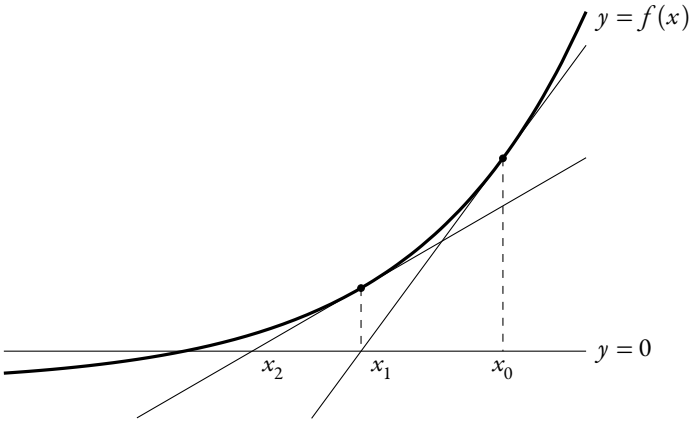


FIGURE 1.9 – Deux itérations de la méthode de Newton.

1.3.3 Problème stationnaire à une dimension

L'objectif de cette section est de résoudre numériquement une équation du type $f(x) = 0$ où f dénote une fonction réelle d'une variable réelle que l'on sait évaluer. On cherche ainsi un nombre réel α vérifiant $0 \in f\,(]\alpha - \varepsilon, \alpha + \varepsilon[)$ où la quantité $\varepsilon > 0$ dénote l'erreur tolérée.

Méthode de la dichotomie. La première méthode que nous allons décrire s'appuie sur le théorème des valeurs intermédiaires et fonctionne donc dès que la fonction f est continue. Son principe est identique à celui que nous avons vu dans le cadre de la recherche d'un élément dans une liste triée : diviser par deux l'espace de recherche à chaque itération.

Afin de chercher un zéro de f sur l'intervalle $[a, b]$ à ε près on procède alors ainsi :

```
def dichotomie(f, a, b, epsilon):
    # On suppose f(a)*f(b)<=0.
    while b-a>epsilon:
        c=(a+b)/2
        if f(a)*f(c)<=0:
            b=c
        else:
            a=c
    return a
```

Chaque itération divise par deux l'encadrement de la racine et augmente ainsi la précision d'un bit significatif. Il faut donc $\log\left(\frac{b-a}{\varepsilon}\right)$ itérations pour obtenir une approximation à ε près.

Exercice. *Utiliser cette méthode pour calculer des valeurs approchées de $\sqrt{2}$ et de π.*

Méthode de Newton. Cette seconde méthode exploite la formule de Taylor à l'ordre un et fonctionne dès que la fonction f est de classe $\mathscr{C}^2$. Son principe est d'approcher f par ses tangentes et donc d'approcher le zéro recherché par les zéros de ces droites. Voir la figure 1.9.

Définition. *Soit f une fonction réelle de classe $\mathscr{C}^2$. On appelle suite de Newton pour cette fonction toute suite vérifiant la relation de récurrence $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.*

Théorème. *Soit α un zéro d'une fonction f de classe $\mathscr{C}^2$ pour lequel $f'(\alpha) \neq 0$. Il existe un voisinage V de α tel que toute suite de Newton à valeur initiale dans V converge quadratiquement vers α.*

Démonstration. Le développement de Taylor de f en α donne l'existence d'une suite y telle que

$$0 = f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2} f''(y_n)(\alpha - x_n)^2;$$

divisant par $f'(x_n)$ on obtient

$$\frac{f(x_n)}{f'(x_n)} + (\alpha - x_n) = -\frac{1}{2}\frac{f''(y_n)}{f'(x_n)}(\alpha - x_n)^2$$

c'est-à-dire

$$\alpha - x_{n+1} = -\frac{1}{2}\frac{f''(y_n)}{f'(x_n)}(\alpha - x_n)^2$$

or, par continuité, f'' et $1/f'$ sont bornés au voisinage de α. □

On cherche donc un zéro de f (de dérivée g) au voisinage de a en n itérations ainsi :

```
def newton(f, g, a, n):
    for i in range(0,n):
        a=a-f(a)/g(a)
    return a
```

La preuve ci-dessus montre que la précision en nombre de bits significatifs double à chaque itération. Autrement dit, il faut $c + \log\left(\log\left(\frac{1}{\varepsilon}\right)\right)$ itérations pour obtenir une approximation à ε près.

Exercice. *Écrire un programme calculant en fonction de $k, n > 0$ une approximation de $\sqrt[k]{n}$.*

Exercice. *Montrer que la suite définie par $x_0 = 1$ et $x_{n+1} = \cos(x_n)$ converge. Calculer une approximation de sa limite. Comparer les vitesses de convergence de la dichotomie et de Newton.*

Les suites de Newton sont utiles même lorsque les hypothèses du théorème ci-dessus ne sont pas vérifiées. Par exemple, si la dérivée s'annule, alors la converge est toujours possible, même si pas nécessairement quadratique.

Ces suites sont encore utiles lorsque la dérivée g de f n'est pas efficacement calculable ; on peut alors l'approcher en posant $g(a) = \frac{1}{\varepsilon}(f(a+\varepsilon) - f(a))$. On pourra en exercice adapter le théorème et sa preuve à ce cas.

1.3.4 Problème dynamique à une dimension

L'approximation de fonctions par leurs tangentes permet aussi de résoudre numériquement les équations différentielles ordinaires, technique connue sous le nom de méthode d'Euler.

Théorème. *Soit une fonction y vérifiant l'équation différentielle ordinaire $y'(x) = f(x, y(x))$ et la condition initiale $y(x_0) = y_0$. Si f est de classe $\mathscr{C}^1$ alors il existe $\eta > 0$ tel que, pour tout $\varepsilon > 0$, les suites définies par*

$$\begin{cases} x_{n+1} = x_n + \varepsilon \\ y_{n+1} = y_n + \varepsilon f(x_n, y_n) \end{cases}$$

vérifient $|y_n - y(x_n)| < \varepsilon$ pour tout $n \in \left\{1, \ldots, \left\lfloor \frac{\eta}{\varepsilon} \right\rfloor\right\}$.

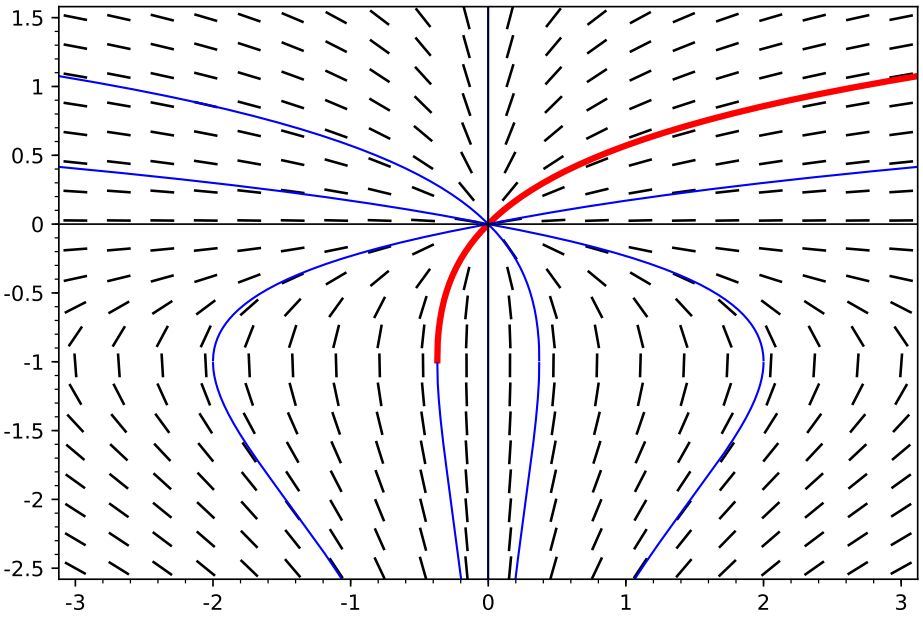


FIGURE 1.10 – Champ de vecteurs et solutions de l'équation différentielle $y' = \frac{y}{x(1+y)}$. En rouge, la fonction W de Lambert.

Exercice. *Calculer ainsi une approximation de la fonction exponentielle.*

Exercice. *Calculer des fonctions y solutions de l'équation différentielle $y' = \frac{y}{x(1+y)}$. Qu'observez-vous lorsque $x \to 0$? Et lorsque $y \to -1$? Voir la figure 1.10.*

Exercice (méthode de Runge–Kutta). *Refaire les exercices précédents avec l'approximation :*

$$y(t+\varepsilon) \approx y(t) + \varepsilon \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \qquad \text{où} \qquad \begin{cases} k_1 = f(t, y(t)) \\ k_2 = f(t + \frac{\varepsilon}{2}, y(t) + \frac{\varepsilon}{2} k_1) \\ k_3 = f(t + \frac{\varepsilon}{2}, y(t) + \frac{\varepsilon}{2} k_2) \\ k_4 = f(t + \varepsilon, y(t) + \varepsilon k_3) \end{cases}$$

Comparer les vitesses de convergence de la méthode d'Euler et de cette nouvelle méthode.

1.3.5 Problème multidimensionnel linéaire

L'étude de ce problème est repoussé au second semestre où nous serons bien plus à même de l'entreprendre avec rigueur. Voir la section 2.2.7.

Chapitre 2

Second semestre

Ce semestre est une introduction aux aspects théoriques de l'informatique. Nous y aborderons, d'une part, de nouveaux paradigmes de programmation (fonctionnel avec Caml puis déclaratif avec SQL) et, d'autre part, les outils rigoureux d'analyse d'algorithme.

2.1 Méthodes de programmation

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. C'est ainsi que tout processeur fonctionne ; cette approche naturelle a donc pour avantage de ne demander qu'un modeste travail de traduction du code en instructions processeurs.

Certains langages dits *de plus haut niveau* permettent, par un travail de traduction plus complexe, de programmer autrement. En langage Caml, nous adopterons certains idiomes de **programmation fonctionnelle** : notre code consistera en des définitions de fonctions (en un sens proche des mathématiques), leur composition et évaluation ; en particulier, nous ne modifierons pas la valeur des variables.

La version de ce langage au programme des classes préparatoires [35] fut longtemps l'obsolète Caml Light [26] mais est désormais OCaml [30] qui, pour notre usage, n'en diffère que marginalement. Garde toutefois à ce détail si vous consultez des resources antérieures à 2018 sur l'option informatique en classes préparatoires scientifiques.

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.*

2.1.1 Expressions

Même si l'interface que nous utilisons est clémente, l'interpréteur Caml exige que toute expression se conclue par « `;;` » et votre correcteur aux concours aussi ! On écrira donc :

```
# 1 ;;
- : int = 1
# 1.1 ;;
- : float = 1.1
```

Observons qu'un résultat est affiché sous la forme « `nom : type = valeur` » ; on prendra l'habitude de consulter attentivement cette information pour vérifier que notre code a été compris comme on s'y attendait, d'autant que le respect des types est crucial en Caml :

```
# 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 (`int`). Comme évoqué au premier semestre, ce type réalise l'arithmétique de $\mathbb{Z}/2^{64}\mathbb{Z}$ avec comme représentants les entiers de $\left\{-2^{63}, 2^{63}-1\right\}$. On dispose notamment des opérateurs classiques `+`, `-`, `*`, `/` et `mod`.

```
# 8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8*8 ;;
- : int = 0
```

Nombres réels (`float`). Ce type adhère à la norme IEEE 754 décrite au premier semestre, dont nous avions omis certains cas particuliers. Ses opérateurs sont distincts de ceux portant sur les entiers et comportent `+.`, `-.`, `*.`, `/.`, `**`, `sqrt`, `sin`, `cos`, `exp` et `log`.

```
# 1. /. 0. ;;
- : float = infinity
# infinity -. infinity ;;
- : float = nan
```

Chaînes de caractères (`string`). Il ne s'agit là que de tableaux de caractères mais avec un type et une syntaxe spécifiques.

```
# String.length "coucou" ;;
- : int = 6
# "coucou".[0] ;;
- : char = 'c'
# "coucou"^"toi" ;;
- : string = "coucoutoi"
```

Booléens (`bool`). Ses éléments se notent `true` et `false` sans majuscules, contrairement au cas du langage Python. Ils disposent des opérateurs `not`, `&&` et `||`.

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

Les opérateurs de comparaison, `=`, `<>`, `<=`, `>=`, `<` et `>`, sont quant à eux polymorphes, c'est-à-dire qu'ils peuvent être appliqués à des types arbitraires, cela jusqu'aux confins du raisonnable.

```
# 0 = 1 ;;
- : bool = false
# 0. <> 1. ;;
- : bool = true
# 0 <= 1 ;;
- : bool = true
# 0. <= 1. ;;
- : bool = true
# "abc" < "def" ;;
- : bool = true
# false < true ;;
- : bool = true
```

Remarque. *En langage Caml, 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 !***

Afin de convertir les types, on dispose de fonctions nommées `type_of_type`.

```
# int_of_float 65.4321 ;;
- : int = 65
# char_of_int 65 ;;
- : char = 'A'
```

On déclarera 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
```

Une différence considérable avec le paradigme de la programmation impérative est que les variables ne sont pas mutables, c'est-à-dire qu'on ne peut pas modifier leur valeur. En l'absence du mot clef `let`, Caml interprète l'opérateur = comme un test d'égalité.

```
# let x = 1 ;;
val x : int = 1
# x = 2 ;;
- : bool = false
```

Les branchements conditionnels empruntent la structure `if-then-else` classique. On peut délimiter des blocs d'instructions par les mots clefs `begin` et `end`, faute de quoi seule la première instruction suivant `then` et `else` est sujette au branchement.

```
# if 1 <> 1 then print_int 2; print_int 3 ;;
3- : unit = ()
# if 1 <> 1 then begin print_int 2; print_int 3 end ;;
- : unit = ()
# if 1 <> 1 then print_int 2; print_int 3 else print_int 4 ;;
Error: Syntax error
```

2.1.2 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
# f (f 2) ;;
- : int = 11
```

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>
```

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

Exercice. *Écrire une fonction prenant en argument une chaîne de caractère et renvoyant sa première lettre si celle-ci est égale à sa dernière et renvoyant la lettre du milieu si ce n'est pas le cas.*

L'idiome « `let f = function x -> ...` » est lourd et suffisamment courant pour admettre 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 (x /. y) ;;
val argument : float -> float -> float = <fun>
# let argument x y = atan (x /. y) ;;
val argument : float -> float -> float = <fun>
```

Le type (parfois aussi appelé signature) de la fonction `argument` ainsi déclarée est un abus de notation pour « `float -> (float -> float)` », Caml 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 ne requiert aucun type particulier pour un ou plusieurs de ses arguments, ce que la signature indique par des *variables de type* qui en Caml sont notées « `'a` », « `'b` » et ainsi de suite.

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

Exemple. *On peut inculquer à Caml 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 Caml ; 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 :*

1. *la quantité x^n en fonction du réel x et de l'entier n ;*
2. *le plus grand diviseur commun de deux entiers donnés ;*
3. *si un entier donné est une puissance de deux ;*
4. *si un entier donné est premier (penser à utiliser une fonction auxiliaire) ;*
5. *si un entier donné est une puissance d'un nombre premier.*

Exercice. *Écrire une fonction calculant $\sqrt[n]{x}$ à ε près en utilisant le principe de dichotomie. En déduire une fonction plus efficace que la précédente déterminant si un entier est une puissance d'un nombre premier donné.*

2.1.3 Tuples

On s'efforcera d'employer la notion de produit cartésien avec parcimonie en Caml car elle entrave l'utilisation de nombreux idiomes de programmation fonctionnelle. On pourra néanmoins écrire :

```
# let x = 1,2,'c' ;;
val x : int * int * char = (1, 2, 'c')
# let a,b,c = x in a ;;
- : int = 1
# let y = 1,(2,'c') ;;
val y : int * (int * char) = (1, (2, 'c'))
```

De même pour les fonctions :

```
# let f a b = a*b ;;
val f : int -> int -> int = <fun>
# let g (a,b) = a*b ;;
val g : int * int -> int = <fun>
# f = g ;;
Error: This expression has type int * int -> int
       but an expression was expected of type int -> int -> int
       Type int * int is not compatible with type int
```

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

2.1.4 Aiguillage

Les aiguillages sont un élément central du langage Caml ; ils offrent les mêmes fonctionnalités que les branchements conditionnels classiques mais leur sont préférables, en premier lieu pour des raisons esthétiques. Plutôt que d'écrire :

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

On écrira de manière parfaitement équivalente :

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

De surcroit, l'idiome « `function x -> match x with | ...` » admet lui-même une forme abrégée, à savoir « `function | ...` » ; on gagnera donc en lisibilité en écrivant :

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

2.1.5 Listes

Les listes se notent `[a; b; c]` et leurs éléments doivent être de même type. Il s'agit de véritables listes chaînées (voir figure 1.6) et pour les manipuler nous n'avons essentiellement à notre disposition que 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])
```

Les types avancés telles les listes sont en Caml implantés dans des *modules*, constructions qui s'apparentent aux bibliothèques logicielles dont nous discuterons plus tard ; pour la programmation, on se contentera de retenir que les fonctions associées s'appellent suivant la convention `Module.fonction` comme ci-dessus.

Toute liste non vide est donc de la forme `head::tail` ; cette dichotomie est l'origine du paradigme de manipulation des listes cher au langage Caml. Par exemple, pour calculer la longueur d'une liste on écrira :

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

(Heureusement déjà disponible en la fonction `List.length`.)

Exercice. *Écrire une fonction « `somme : int list -> int` » calculant la somme d'une liste d'entiers donnée.*

Écrire une fonction « `max : 'a list -> 'a` » renvoyant le plus grand élément de la liste donnée. Faire de même pour la moyenne.

Exercice. *Écrire une fonction « `appartient : 'a -> 'a list -> bool` » déterminant si un élément donné appartient à une liste donnée.*

Exercice. *Écrire une fonction « `enlever : int -> 'a list -> 'a list` » enlevant l'élément d'indice donné dans la liste donnée.*

Écrire une fonction « `inserer : 'a -> int -> 'a list -> 'a list` » insérant l'élément donné à l'indice donné dans la liste donnée.

Exercice. *Écrire une fonction « `applique : ('a -> 'b) -> 'a list -> 'b list` » qui étant donnés une fonction f et une liste $[x_0; x_1; x_2; \dots]$ renvoie la liste $[f(x_0); f(x_1); f(x_2); \dots]$.*

Exercice. *Écrire une fonction « `existe : ('a -> bool) -> 'a list -> bool` » qui détermine si l'un des éléments de la liste donnée vérifie la propriété donnée.*

Écrire une fonction « `pourtout : ('a -> bool) -> 'a list -> bool` » qui détermine si tout les éléments de la liste donnée vérifient la propriété donnée.

Armé de fonctions auxiliaires, ce paradigme peut être employé afin de traiter tout problème d'algorithmique. Par exemple, afin de couper une liste en deux, on pourra procéder ainsi :

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

Exercice. *Écrire une fonction « `range : int -> int list` » prenant en argument un entier n et renvoyant la liste $[0; 1; 2; \dots; n-1]$ formée des n plus petits entiers.*

Exercice. *Écrire une fonction « `inverse : 'a list -> 'a list` » qui étant donnée une liste $[x_0; x_1; \dots; x_n]$ renvoie la liste $[x_n; \dots; x_1; x_0]$.*

2.1.6 Références

Même si leur usage n'est pas préconisé en Caml, les variables mutables existent. On doit les déclarer explicitement par le mot clef `ref` ; l'affectation puis l'accès aux valeurs se fait alors par les opérateurs `:=` et `!` comme il suit.

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

L'affectation d'une valeur à notre référence ne renvoie rien ; elle se contente de modifier la référence en question. En Python on parlerait d'instruction ; en Caml on parle d'expression *de type `unit`*.

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

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

Tout l'intérêt de Caml est cependant d'adopter une approche fonctionnelle ; on s'en efforcera autant que faire se peut. Par exemple, pour calculer la suite de Fibonacci aussi rapidement que la fonction impérative ci-dessus on pourra écrire :

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

Remarque. *On évitera l'usage de variables globales : leur portée est généralement dynamique en programmation impérative et statique en programmation fonctionnelle ; concrètement, les programmes Python et Caml ci-dessous renverront respectivement* 4, 2, 4 *et* 2.

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

2.1.7 Tableaux

Les tableaux se notent `[|a; b; c|]` et leurs éléments doivent être de même type. Il s'agit de véritables tableaux (voir figure 1.5) et leur taille est donc fixe. Contrairement aux tuples, c'est une structure mutable, c'est-à-dire qui peut être modifiée : le k^{e} élément se dénote `v.(k)` et on peut en changer la valeur 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) 1 in
    for k = 2 to n do
        t.(k) <- t.(k - 1) + t.(k - 2);
    done;
    t.(n)
;;
```

Évidemment rien ne nous oblige à nous cantonner aux entiers.

```
let f n x = x+n ;;
let t = Array.make 5 (f 0) ;;
for k=0 to 4 do t.(k) <- f k done ;;
```

Le tableau `t` est alors de type « `(int -> int) array` ».

Exercice. *Écrire une fonction qui détermine si un élément apparaît dans un tableau.*

Exercice. *Écrire une fonction qui inverse l'ordre des éléments d'un tableau.*

Exercice. *Écrire une fonction qui renvoie la liste des indices d'un élément dans un tableau.*

Exercice. *Écrire une fonction qui détermine si une chaîne de caractères est un palindrome.*

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.*

2.1.8 Types

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

Types produits. On peut définir un nouveau type comme produit cartésien de types existants, c'est-à-dire essentiellement comme un tuple nommé. Cette construction s'illustre notamment avec les nombres complexes :

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

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

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

Types sommes. 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, ce qui est le cas des droites du plan :

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

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

Exercice. *Écrire une fonction déterminant si deux droites du plan sont perpendiculaires.*

Exercice. *Écrire un type permettant de représenter les 52 cartes à jouer usuelles. Implanter le jeu de la bataille. S'attaquer ensuite au blackjack.*

Contrairement aux fonctions, les définitions des types peuvent être récursives sans indication particulière ; c'est une fonctionnalité indispensable à la construction de structures de données non triviales. Nous y reviendrons et nous contenterons pour l'instant de cet exemple :

```
type 'a liste = Nil | Cons of 'a * 'a liste ;;
Cons(1,Cons(2,Cons(3,Nil))) ;;
```

2.2 Structures de données et algorithmes I

L'objectif de cette section est l'étude rigoureuse de trois aspects élémentaires concernant l'exécution des programmes. Pour ce faire nous préférerons parler 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 quelconque, nous traiterons trois questions :

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

2.2.1 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, c'est-à-dire une application*

$$\leqslant : \begin{cases} E \times E \longrightarrow \{vrai, faux\} \\ (x, y) \longmapsto x \leqslant y \end{cases}$$

vérifiant, pour tout triplet $(x, y, z) \in E^3$ *:*

— $x \leqslant x$,
— $x \leqslant y \wedge y \leqslant z \Longrightarrow z \leqslant z$,
— $x \leqslant y \wedge y \leqslant x \Longrightarrow x = y$.

Définition. *Lorsque toute partie non vide admet un plus petit élément on dit que* $(E, \leqslant)$ *est bien ordonné.*

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

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

Les bons ordres sont exactement les ordres biens fondés totaux. La plupart des ordres que nous rencontrerons dans ce cours seront partiels et c'est pourquoi nous allons nous concentrer sur la notion d'ordre bien fondé.

Exemple. *Ordres totaux :*
— *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.*
Ordres partiels :
— *La divisibilité dans* $\mathbb{N}^*$ *est un ordre bien fondé.*
— *L'inclusion dans* $\mathfrak{P}(\mathbb{N})$ *n'est pas un ordre bien fondé.*

Un critère équivalent, en un sens plus explicite, 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. À l'instar du théorème ci-dessous, on l'admettra comme généralisation du 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 $E = F$.

Montrer qu'une propriété $P(x)$ est vérifiée pour tout élément x d'un ensemble E muni d'un ordre bien fondé $\leqslant$ équivaut donc à montrer :

1. Que, pour tout élément minimal $m \in E$, la propriété $P(m)$ est vérifiée.
2. Que, pour tout $y \in E$, si $\forall x \in E, x < y \Rightarrow P(x)$, alors $P(y)$ est vérifiée.

Remarque. *Pour un bon ordre, l'élément minimal* m *est unique ; l'initialisation se borne alors à vérifier* $P(m)$. *Ce n'est pas le cas en général où de multiples éléments minimaux* m *coexistent ; il faudra donc prendre soin de vérifier* $P(m)$ *pour chacun d'entre eux.*

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 arbitraire (dans le cas de boucles `while`) $x \in X$. Pour ce faire on montrera indépendamment trois assertions :

1. L'ensemble X est muni d'un ordre bien fondé.
2. L'exécution $\mathscr{A}(x)$ termine pour tout élément minimal $x \in X$.
3. Toute exécution $\mathscr{A}(x)$ se réduit à un nombre fini d'exécutions $\mathscr{A}(y)$ avec $y < x$.

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

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

L'ensemble $\mathbb{N}$ muni de son ordre naturel 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 $k = n - 1 < n$. Par induction, on obtient donc que `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.*

```
let rec a n = if n=0 then 0 else n - a(n-1) ;;
let rec b n = if n=0 then 1 else n - b(b(n-1)) ;;
let rec c n = if n=0 then 0 else n - c(c(n-1)) ;;
```

Exercice. *Montrer la terminaison des algorithmes suivants.*

```
let rec pgcd a =
    function
    | 0 -> a
    | b -> pgcd b (a mod b)
;;

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

let log2 n =
    let k = ref 0 in
    let x = ref n in
    while !x > 1 do
        x := !x / 2;
        k := !k + 1;
    done;
    !k
;;
```

Parfois, la difficulté consiste à identifier l'ordre bien fondé lui-même :

Exercice. *Montrer la terminaison des algorithmes suivants.*

```
let rec ackermann =
    function
    | 0,n -> n+1
    | m,0 -> ackermann (m-1,1)
    | m,n -> ackermann (m-1,ackermann(m,n-1))
;;
```

```
let rec deuxtrois n =
    if n mod 2 = 1 then n
    else deuxtrois (3*(n/2))
;;

let rec mul =
    function
    | [] -> [1]
    | 0::t -> []
    | h::t -> (mul t)@(mul ((h-1)::t))
;;
```

Remarque. *Le problème de l'arrêt, consistant à déterminer si un algorithme donné termine, est extrêmement difficile en toute généralité. D'une part, il est indécidable, c'est-à-dire qu'on peut montrer (par argument diagonal) qu'il n'existe aucun algorithme permettant de le résoudre ; d'autre part, il est ouvert pour certaines classes d'algorithmes, c'est-à-dire qu'on ne sait prouver ni leur terminaison ni son contraire, par exemple pour :*

```
let rec syracuse =
    function
    | 1 -> ()
    | n -> if n mod 2 = 0
        then syracuse (n/2)
        else syracuse (3*n+1)
;;
```

2.2.2 Correction

Considérons à présent un algorithme $\mathscr{A}$ dépendant d'un paramètre arbitraire $x \in X$ et attelons nous à montrer que la valeur $\mathscr{A}(x)$ qu'il renvoie est telle que calculée par une fonction mathématique f. Il s'agit donc de considérer la suite des instructions exécutées par $\mathscr{A}$ et de montrer qu'elle débouche sur $f(x)$. On met pour cela à nouveau en œuvre le principe d'induction afin de montrer la proposition $\mathscr{A}(x) = f(x)$ pour tout $x \in X$. On procèdera donc en trois temps :

1. Montrer que l'ensemble des valeurs est muni d'un ordre bien fondé.
2. Montrer que l'algorithme est correct pour toute valeur minimale.
3. Montrer que si l'algorithme est correct pour tout $x < y$ alors il l'est pour y.

Ces preuves sont généralement transparentes dans le cas de fonctions récursives pures, c'est-à-dire sans boucles, puisque ces valeurs ne sont alors autres que les paramètres de l'algorithme.

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

```
let rec pgcd a =
    function
    | 0 -> a
    | b -> pgcd b (a mod b)
;;
```

La quantité $\mathrm{pgcd}(a,b)$ *est préservée à chaque appel récursif de la fonction grâce à l'identité* $\mathrm{pgcd}(a,b) = \mathrm{pgcd}(b, a \bmod b)$. *L'algorithme termine lorsque* $b = 0$, *cas où l'on a clairement* $\mathrm{pgcd}(a,b) = a$. *Cette fonction renvoie donc bien* $\mathrm{pgcd}(a,b)$.

Exercice. *Montrer la correction des algorithmes* `factorielle` *et* `longueur` *ayant fait l'objet de preuves de terminaison dans la précédente section. Montrer aussi celle de* `mul` *après avoir identifié ce qu'il calcule.*

En présence d'instructions itératives (familièrement appelées *boucles*) 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.*

```
let maximum v =
    let m = ref v.(0) in
    for k = 1 to Array.length v - 1 do
        if !m < v.(k) then m := v.(k);
    done;
    !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'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 bien $m = \max v$.

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

2.2.3 Complexité

Les processeurs n'offrent qu'un (petit) nombre fini d'instructions et c'est uniquement en les combinant méthodiquement que les programmes parviennent à réaliser des calculs complexes. En première approximation, on peut supposer que chaque instruction processeur 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. *Pour les besoins de ce cours, les processeurs seront des boîtes noires arbitraires offrant un jeu d'instructions minimal et toute autre considération sera ignorée, notamment le coût des appels mémoire. Une approche plus sérieuse consisterait à introduire un formalisme abstrait modélisant le calcul sur ordinateur telles les machines de Turing [2], mais on obtient généralement des complexités comparables quel que soit le modèle retenu.*

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 $O(b)$.*

```
let multiplication a b = a*b ;;
```

```
let multiplication a b =
    let x = ref 0 in
    for k = 1 to b do
        x := !x + a;
    done;
    !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 :*

```
let sommecarre n =
    let x = ref 0 in
    for k = 1 to n do
        x := !x + multiplication k k;
    done;
    !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.

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 :*

```
let rec multiplication a =
    function
    | 0 -> 0
    | b -> 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 :

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

Exercice. *Donner la complexité de tous les programmes implantés précédemment.*

Considérons à présent les programmes suivants.

```
let rec fibo1 =
    function
    | 0 -> 0
    | 1 -> 1
    | n -> fibo (n-1) + fibo (n-2)
;;

let fibo2 =
    let rec aux a b =
        function
        | 0 -> a
        | n -> aux b (a+b) (n-1)
    in
    aux 0 1
;;

let fibo3 n =
    let mul a b =
        [| a.(0)*b.(0)+a.(1)*b.(2);
           a.(0)*b.(1)+a.(1)*b.(3);
           a.(2)*b.(0)+a.(3)*b.(2);
           a.(2)*b.(1)+a.(3)*b.(3) |]
    in
    let rec pow n x t =
        if n = 0 then t else if n mod 2 = 1
        then pow (n/2) (mul x x) (mul t x)
        else pow (n/2) (mul x x) t
    in
    (pow n [|0;1;1;1|] [|1;0;0;1|]).(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 2.1.

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.

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

2.2.4 Applications aux entiers

L'archétype des opérations en cachant d'autres, ce qui constitue un piège récurrent et impitoyable pour l'évaluation de la complexité, sont celles portant sur les entiers de taille arbitraire.

Rappelons que les processeurs opèrent sur des mots de 64 bits et que, afin de représenter les entiers naturels, ils disposent d'un type `uint64` suivant lequel l'entier $\sum_{i=0}^{63} c_i 2^i$ est stocké comme le tableau de bits $[c_0, c_1, \ldots, c_{63}]$. Cette représentation est naturellement limitée à l'ensemble $\{0, \ldots, 2^{64}-1\}$ et le processeur implante donc l'arithmétique de $\mathbb{Z}/2^{64}\mathbb{Z}$. C'est pourquoi on observe en Caml :

```
# let rec puissance x n = if n=0 then 1 else x * puissance x (n-1) ;;
val puissance : int -> int -> int = <fun>
# puissance 2 64 ;;
- : int = 0
```

Pour que le processeur puisse calculer la multiplication de deux entiers et renvoyer un résultat exact, il faut donc que ces derniers soient restreints à l'ensemble $\{0, \ldots, 2^{32}-1\}$. Les opérations arithmétiques **sur cet ensemble borné** sont des instructions processeurs et s'exécutent donc bien en temps constant.

La technique connue sous le nom d'arithmétique multiprécision permet d'opérer de manière exacte sur des entiers arbitrairement grands en les représentant comme des tableaux d'entiers machines, c'est-à-dire de `uint64`. Plus précisément, l'entier N dont la décomposition en base $B = 2^{32}$ est $N = \sum_{i=0}^{n-1} x_i B^i$ sera représenté comme le tableau $[|x_0; x_1; \ldots; x_{n-1}|]$. Toute la difficulté consiste alors à réduire les opérations arithmétiques portant sur N en une suite d'instructions processeurs portant sur les x_i.

Remarque. *C'est exactement ce que nous faisons lorsque nous calculons à la main sur des entiers décimaux ! Tout problème auquel nous pourrions faire face peut donc être attaqué avec du papier*

et un crayon pour $B = 10$ et de petites valeurs de n ; il suffira ensuite de généraliser la méthode obtenue à $B = 2^{32}$ et n arbitraire.

Commençons par un petit échauffement sans grande difficulté.

Exercice. *Écrire une fonction prenant en argument un entier machine et renvoyant le grand entier correspondant (sous forme d'un tableau d'entiers machines).*

```
let base = puissance 2 32 ;;
let grand n = [| n / base; n mod base |] ;;
```

Dès à présent et jusqu'à la fin de ce chapitre on s'attachera à montrer la terminaison et la correction puis à évaluer la complexité de tout algorithme conçu. Pour l'implantation, consulter au besoin la documentation du langage :

`https://caml.inria.fr/pub/docs/manual-ocaml/`

Exercice. *Écrire une fonction comparant deux grands entiers et renvoyant $+1$, 0 et -1 lorsque le premier est respectivement plus petit, égal et plus grand que le second.*

```
let compare x y =
    let a = ref (Array.length x - 1) in
    let b = ref (Array.length y - 1) in
    while !a>0 && x.(!a)=0 do decr a done;
    while !b>0 && y.(!b)=0 do decr b done;
    if !a < !b then +1 else
    if !b > !a then -1 else
    let r = ref 0 in
    while !r=0 && !a>=0 do
        if x.(!a) < y.(!a) then r:=+1 else
        if x.(!a) > y.(!a) then r:=-1;
        decr a;
    done;
    !r
;;
```

Exercice. *Écrire une fonction calculant la somme de deux grands entiers.*

```
let addition x y =
    let a = Array.length x in
    let b = Array.length y in
    let z = Array.make (max a b + 1) 0 in
    for i = 0 to Array.length z - 1 do
        if i < a then z.(i) <- z.(i) + x.(i);
        if i < b then z.(i) <- z.(i) + y.(i);
        if z.(i) >= base then begin
            z.(i+1) <- z.(i+1) + z.(i)/base;
            z.(i) <- z.(i) mod base;
        end
    done;
    z
;;
```

Rappelons que la méthode naïve consiste à calculer les produits par le développement classique

$$\sum_{i=0}^{n-1} x_i B^i \cdot \sum_{j=0}^{m-1} x_j B^j = \sum_{k=0}^{nm-1} \left(\sum_{i+j=k} x_i x_j \right) B^k$$

et il ne reste alors qu'à propager les retenues, problème néanmoins plus délicat que dans le cas d'une somme.

Exercice. *Écrire une fonction calculant ainsi le produit de deux grands entiers.*

```
let multiplication x y =
    let a = Array.length x in
    let b = Array.length y in
    let z = Array.make (a+b) 0 in
    for i = 0 to a - 1 do
        for j = 0 to b - 1 do
            let k = ref (i+j) in
            z.(!k) <- z.(!k) + x.(i)*y.(j);
            while z.(!k) >= base do
                z.(!k+1) <- z.(!k+1) + z.(!k)/base;
                z.(!k) <- z.(!k) mod base;
                incr k;
            done;
        done;
    done;
    z
;;
```

Afin de calculer $(aB+b)\cdot(cB+d) = \left(acB^2 + (ad+bc)B + bd\right)$ la méthode ci-dessus réalise quatre multiplications d'entiers machines. Cependant, une fois ac et bd calculés, le terme $ad+bc$ peut s'obtenir au prix d'une seule multiplication sous la forme $(a+b)(c+d)-ac-bd$; la multiplication étant une opération plus coûteuse que l'addition, il est avantageux d'exploiter cette formule qui utilise trois multiplications plutôt que quatre. En l'appliquant récursivement aux grands entiers, on obtient la méthode de multiplication de Karatsuba [14] dont la complexité est $O(n^{\log_2(3)})$.

Exercice. *Écrire une fonction calculant ainsi le produit de deux grands entiers.*

```
let rec karatsuba x y =
    let a = Array.length x in
    let b = Array.length y in
    let k = a / 2 in
    if k < 3 then multiplication x y else
    let x0 = Array.sub x 0 k in
    let y0 = Array.sub y 0 k in
    let x1 = Array.sub x k a in
    let y1 = Array.sub y k b in
    let z0 = karatsuba x0 y0 in
    let z2 = karatsuba x1 y1 in
    let s = karatsuba (addition x0 x1) (addition y0 y1) in
    let z1 = soustraction (soustraction s z0) z2 in
    let z = Array.make 0 (a+b+1) in
    for i = 0 to a+b do
        if i     < Array.length z0 then z.(i) <- z.(i) + z0.(i);
```

```
        if i-k   < Array.length z1 then z.(i) <- z.(i) + z1.(i-k);
        if i-k-k < Array.length z2 then z.(i) <- z.(i) + z1.(i-k-k);
        if z.(i) >= base then begin
            z.(i+1) <- z.(i+1) + z.(i)/base;
            z.(i) <- z.(i) mod base;
        end
    done;
    z
;;
```

Remarque. *L'algorithme de Schönhage–Strassen [21] exploite la transformée de Fourier dans les corps finis afin de multiplier deux entiers de n bits en temps* $O(n\log(n)\log(\log(n)))$*; des avancées successives débouchèrent récemment [37] sur l'obtention d'un algorithme de complexité asymptotique* $O(n\log(n))$*, quantité conjecturée optimale.*

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

```
let rec puissance x n =
    if n = 0 then [|1|] else
    multiplication x (puissance x (n-1)) ;;
;;
```

Cet algorithme naïf calcule x^n comme produit de n copies de x. Un méthode plus rapide est relativement évidente dans le cas $n = 2^k$ où elle consiste à obtenir cette puissance par carrés successifs :

$$x^{2^k} = \left(\cdots\left(x^2\right)^2\cdots\right)^2$$

Cela nécessite $k = \log_2(n)$ multiplications. Pour n quelconque, on se ramène au cas précédent en écrivant la décomposition binaire de n :

$$n = \sum_{i=0}^{k} a_i 2^i \quad \Longrightarrow \quad x^n = \prod_{\substack{i\in\{0,\ldots,k\}\\ a_i=1}} x^{2^i}$$

Cette technique est connue sous le nom de méthode de l'exponentiation rapide :

```
let rec puissance x n =
    if n = 0 then [|1|] else
    let y = puissance (multiplication x x) (n/2) in
    if n mod 2 = 0 then y else multiplication x y
;;
```

Lorsque le résultat d'un calcul n'est demandé que modulo un certain entier N, on gagnera à réduire les résultats intermédiaires par leur représentants dans $\{0,\ldots,N-1\}$ afin de limiter la taille des nombres manipulés et donc la complexité du calcul. Dans ce contexte, la méthode de l'exponentiation rapide sera particulièrement bénéfique car sa complexité sera alors logarithmique en n.

La division euclidienne est la dernière opération manquante afin d'avoir une boîte à outil rudimentaire mais complète pour l'arithmétique des (grands) entiers. Nous n'en discuterons pas car elle est extrêmement pénible à réaliser. Pour un ouvrage de référence sur l'arithmétique multiprécision, on consultera avec profit [32].

2.2.5 Applications aux listes

Familiarisons nous avec la manipulation de listes en Caml en reprenant certains problèmes que nous avons traités au semestre dernier en Python. Rappelons à nouveau qu'on attend une démonstration de la terminaison et de la correction de chaque algorithme, ainsi qu'une évaluation de sa complexité.

Exercice. *Écrire des fonctions calculant le minimum, le maximum, la moyenne et enfin l'écart type d'une liste de réels.*

Exercice. *Écrire une fonction calculant le nombre de couples d'indices $i < j$ pour lesquels $L_i > L_j$ dans une liste L donnée.*

Exercice. *Écrire une fonction qui, étant données deux listes, détermine si la première est une sous liste de la seconde. (Nous répondrons à ce problème de manière bien plus satisfaisante l'an prochain quand nous aborderons la théorie des langages.)*

En algorithmique le problème le plus classique est certainement celui consistant à trier une liste donnée. Wikipedia recense une vingtaine d'approches différentes, chacune réalisant des complexités diverses et des propriétés variées. Nous nous bornerons ici à discuter trois approches.

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.*

```
let rec tri =
    let rec insertion x =
        function
        | []   -> [x]
        | y::t -> if x<y then x::y::t else y::insertion x t
    in
    function
    | []   -> []
    | x::t -> insertion x (tri t)
;;
```

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.*

```
let rec tri =
    let rec decoupe a b =
        function
        | []      -> a,b
        | x::[]   -> x::a,b
        | x::y::t -> decoupe (x::a) (y::b) t
    in
    let rec fusion a b =
        match a,b with
        |   [],_    -> b
```

```
        |    _,[]   -> a
        | x::t,y::s -> if x<y then x::(fusion t b) else y::(fusion a s)
    in
    function
    | []  -> []
    | [x] -> [x]
    | l   -> let a,b = decoupe [] [] l in
         fusion (tri a) (tri b)
;;
```

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.*

```
let rec tri l =
    let rec bulle =
        function
        | []      -> [],true
        | x::[]   -> x::[],true
        | x::y::t -> if x>y
              then let r,s=bulle(x::t) in y::r,false
              else let r,s=bulle(y::t) in x::r,s
    in
    let r,s = bulle l in
    if s then r else tri r
;;
```

Exercice. *Discuter des avantages et des inconvénients de chacun de ces tris.*

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. □

Concernant la complexité dans le cas le pire, on ne peut donc pas faire mieux que le tri fusion. Remarquons cependant que les algorithmes exploitant des propriétés intrinsèques d'un ordre spécifique sortent du cadre de ce théorème ; c'est par exemple le cas du *tri par paquet* :

```
let tri x =
    let rec max =
        function
        | [] -> 0
        | h::t -> let m = max t in if h>m then h else m
    in
    let m = max x + 1 in
    let p = Array.make m 0 in
    let rec fill =
        function
        | [] -> ();
```

```
        | h::t -> p.(h) <- p.(h)+1; fill t
    in
    fill x;
    let r = ref [] in
    for i=(m-1) downto 0 do
        for j=1 to p.(i) do r:=i::!r; done;
    done;
    !r
;;
```

Notons enfin que cette approche suppose un accès mémoire en $O(1)$ à toute case du tableau mais que cette hypothèse n'est réaliste uniquement lorsque l'élément maximal est négligeable devant la longueur de la liste.

Pour conclure cette section, présentons deux structures de données dérivées des listes ; elles sont relativement primitives mais omniprésentes dans tout système informatique. Chacune stocke une collection d'objets et offre deux opérations : ajouter et enlever un objet à la collection. Elles diffèrent par l'ordre dans lequel les éléments sont enlevés.

Définition. *On appelle **pile** (resp. **file**) toute 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.

Exercice. *Définir un type Caml représentant les piles et implanter les opérations associées.*

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

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

Les opérations d'empilement et de dépilement se réalisent ainsi en temps constant.

Exercice. *On peut représenter une file par un couple de listes, l'une est utilisée pour ajouter des objets et l'autre pour les enlever ; lorsque la seconde liste est vide, on la remplace par la première renversée. Définir un type Caml adoptant cette représentation et implanter les opérations associées.*

```
type 'a file = { mutable a:'a list; mutable b:'a list } ;;
let file_vide () = { a=[]; b=[] } ;;
let enfiler f x = f.a <- x::f.a ;;

let renverse l =
    let rec aux a = function
        | [] -> a
        | h::t -> aux (h::a) t
    in
    aux [] l
;;
```

```
let defiler f =
    if f.b=[] then begin
        f.b <- renverse f.a;
        f.a <- [];
    end;
    match f.b with
    | [] -> failwith "file vide"
    | h::t -> f.b <- 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 de n appels consécutifs à `enfiler` et `defiler` est en $O(n)$.

Exercice. *Une file peut être représentée comme un triplet (v, s, n) où v est un vecteur et s et n deux entiers. L'entier n dénote le nombre d'objets de la file et s l'indice de l'objet à enlever en premier. Les objets de la file sont ainsi les $v_s, v_{s+1}, \ldots, v_{s+n-1}$ où les opérations sur les indices doivent être réalisées modulo la longueur du vecteur. Implanter une file suivant ce principe.*

```
type 'a file = { mutable v:'a array; mutable s:int; mutable n:int; } ;;
let vide = { v=[|0|]; s=0; n=0; } ;;

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

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

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 globale partant d'une file vide est linéaire en le nombre d'appels à ces fonctions.

2.2.6 Applications aux 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 sont innombrables ; ils forment aussi une étape clef vers les graphes, structure que nous étudierons l'an prochain. Précisons enfin que les arbres et, plus généralement, toutes les structures de données considérées dans ce cours sont implicitement supposés finis.

Définition. *On appelle arbre toute structure de données présentant les caractéristiques suivantes :*
- *Elle est formé d'un ensemble d'objets appelés* nœuds.
- *Chaque nœud admet une famille de nœuds appelés* fils.
- *Tous les nœuds descendent d'un unique nœud appelé* racine.
- *Chaque nœud autre que la racine est le fils d'un unique nœud appelé* père.

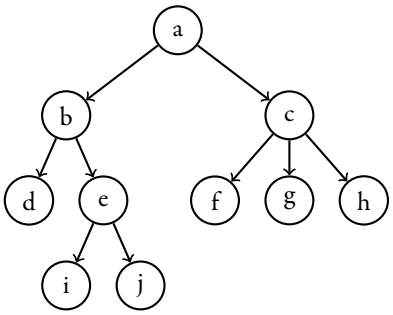


FIGURE 2.2 – Un arbre de taille dix et hauteur trois comportant six feuilles.

La structure d'arbre généralise celle de liste chaînée (revoir la figure 1.6), cas particulier où chaque nœud a au plus un fils. L'élément minimal qu'est la liste vide (celle ne comportant aucun nœud) s'appellera dans ce contexte *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 matchs (finale, demi-finale, etc.) d'une compétition de votre sport favori.*

Les arbres sont représentés comme l'illustre la figure 2.2 : à l'envers, c'est-à-dire avec la racine en haut. Comme ces représentations le présupposent, la définition ci-dessus sera souvent (silencieusement !) amendée en exigeant que l'ensemble des fils de chaque nœud soit implicitement muni d'un ordre total, comme c'est par exemple le cas des membres d'une fratrie (aîné, cadet, benjamin, etc.).

En Caml 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 []]
```

Définition. *On appelle taille d'un arbre le nombre de nœuds qu'il comporte.*
On appelle profondeur d'un nœud la longueur du chemin le reliant à la racine.
On appelle hauteur d'un arbre la plus grande profondeur de ses nœuds.
On appelle feuille tout nœud n'admettant aucun fils.

Exercice. *Soit x un nœud d'un arbre $\mathscr{A}$. On suppose que la profondeur de x est la hauteur de $\mathscr{A}$. Montrer que x est une feuille.*

En langage Caml, pour le type défini ci-dessus, on écrirait donc :

```
let taille a =
    let rec aux =
        function
```

```
        | [] -> 0
        | Noeud(f)::t -> aux t + aux f + 1
    in
    aux [a]
;;

let hauteur a =
    let rec aux =
        function
        | [] -> -1
        | Noeud(f)::t -> max (aux t) (aux f + 1)
    in
    aux [a]
;;
```

Exercice. *Écrire une fonction renvoyant le nombre de feuilles dans un arbre donné.*

Exercice. *Écrire une fonction déterminant si deux arbres de type* `arbre` *sont égaux.*

Les arbres sont nettement plus intéressants lorsqu'on annote leurs nœuds ; on parle alors 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. Définissions donc en langage Caml le type suivant :

```
type 'a arbreE = Noeud of 'a * 'a arbreE list ;;
```

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

On peut procéder de deux manières naturelles distinctes : en effectuant un **parcours en profondeur**, c'est-à-dire en explorant systématiquement les fils d'un nœud avant ses frères ; ou en effectuant un **parcours en largeur**, c'est-à-dire en explorant les fils d'un nœud après ses frères. Pour le type défini ci-dessus on peut écrire :

```
let profondeur a =
    let rec aux =
        function
        | [] -> []
        | Noeud(e,f)::t -> e::(aux(f@t))
    in
    aux [a]
;;

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

Sur l'arbre de la figure 2.2 la fonction `profondeur` renvoie `[a;b;d;e;i;j;c;f;g;h]` alors que `largeur` renvoie `[a;b;c;d;e;f;g;h;i;j]`. On pourra montrer que ces méthodes sont toutes deux de complexité optimale en exercice.

Exercice. *Écrire une fonction Caml de type « `string arbreE -> ()` » affichant à l'écran une représentation graphique de l'arbre donné. Elle pourra se comporter comme il suit.*

```
# let n = Noeud("entiers",[
      Noeud("premiers",[
          Noeud("pairs",[
              Noeud("deux",[]);
          ]);
          Noeud("impairs",[
              Noeud("trois",[]);
              Noeud("cinq",[]);
              Noeud("...",[]);
          ]);
      ]);
      Noeud("composes",[
          Noeud("quatre",[]);
          Noeud("six",[]);
          Noeud("...",[]);
      ]);
  ]) ;;
val n : string arbreE = ...
# affiche n ;;
└── entiers
    ├── premiers
    │   ├── pairs
    │   │   └── deux
    │   └── impairs
    │       ├── trois
    │       ├── cinq
    │       └── ...
    └── composes
        ├── quatre
        ├── six
        └── ...
- : unit = ()
```

```
let affiche a =
    let rec pr p x =
        match p with
        | true ::t -> print_string "│   "; ligne t x
        | false::t -> print_string "    "; ligne t x
        | []       -> print_endline x
    in
    let rec aux p =
        function
        | Noeud(e,[])::[] -> pr p ("└── "^e)
        | Noeud(e,f )::[] -> pr p ("└── "^e); aux (p@[false]) f
        | Noeud(e,[])::t  -> pr p ("├── "^e);                    aux p t
        | Noeud(e,f )::t  -> pr p ("├── "^e); aux (p@[true ]) f; aux p t
    in
    aux [] [a]
```

```
;;
```

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 Caml plus commodément sans faire appel aux listes :

```
type arbreB =
    | Nil
    | 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. Avant toute chose il nous faut nous munir d'un ordre bien fondé indispensable à la réalisation de preuves par induction.

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 bon ordre sur l'ensemble des arbres binaires.*

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.*

On utilisera à présent le type d'arbre binaire étiqueté défini ci-dessous.

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

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

Exercice. *La numérotation de Sosa–Stradonitz (appelée* Ahnentafel *de l'allemand en anglais) 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 en Caml un type d'arbre binaire étiqueté puis une fonction « `arbreB -> int arbreBE` » calculant cette numérotation. Montrer que cette numérotation donne des étiquettes distinctes aux nœuds d'un même arbre.*

2.2.7 Applications aux matrices

On considère ici le problème consistant à résoudre un système de n équations linéaires en k inconnues, c'est-à-dire un système du type :

$$\begin{cases} c_{11}x_1 & + & c_{12}x_2 & + & \cdots & + & c_{1k}x_k & = & d_1 \\ c_{21}x_1 & + & c_{22}x_2 & + & \cdots & + & c_{2k}x_k & = & d_2 \\ & & & & & & \vdots & & \\ c_{n1}x_1 & + & c_{n2}x_2 & + & \cdots & + & c_{nk}x_k & = & d_n \end{cases}$$

Ce problème a été abordé d'un point de vue théorique dans le cours de logique et fondements où nous avons obtenu l'algorithme ci-dessous permettant de se ramener au cas d'un système triangulaire supérieur, c'est-à-dire vérifiant $c_{ij} = 0$ lorsque $i > j$.

Algorithme (pivot de Gauss).

Entrée : *Un système d'équations (E_i) avec coefficients $(c_{ij})_{\substack{i\in\{1,\dots,n\}\\ j\in\{1,\dots,k\}}}$.*

Sortie : *Un système triangulaire admettant les mêmes solutions.*

1. *Poser* $\ell \leftarrow 1$.
2. *Pour* m *de* 1 *à* k *:*
3. *Pour* i *de* ℓ *à* n *:*
4. *Si* $c_{im} \neq 0$ *:*
5. *Appliquer* $(E_i) \leftrightarrow (E_\ell)$.
6. *Si* $c_{\ell m} \neq 0$ *:*
7. *Appliquer* $(E_\ell) \leftarrow \frac{1}{c_{\ell m}}(E_\ell)$.
8. *Pour* i *de* $\ell + 1$ *à* n *:*
9. *Appliquer* $(E_i) \leftarrow (E_i) - c_{im}(E_\ell)$.
10. *Poser* $\ell \leftarrow \ell + 1$.

La variable m dénote l'indice de l'inconnue en cours d'élimination ; les lignes restant à traiter sont celles d'indices $> \ell$. Pour vérifier la correction de cet algorithme on peut prendre comme invariant de boucle à l'étape 3 la proposition « la sous matrice d'indices $(i, j) \in \{1, \dots, \ell\} \times \{1, \dots, m\}$ est triangulaire supérieure ».

Attention, l'existence de solutions n'équivaut pas à l'inégalité $k > n$. Toutefois la méthode du pivot de Gauss éliminera naturellement la redondance entre les équations de sorte que :

— Si $k < n$ mais qu'une solution existe, on obtiendra des lignes de la forme $0 = 0$.

— Si $k > n$ mais qu'aucune solution n'existe, on obtiendra une ligne de la forme $0 = 1$.

Exercice. *Démontrer la terminaison, la correction et évaluer la complexité de cet algorithme.*

Exercice. *Implanter cet algorithme en langage Caml. La fonction ainsi réalisée acceptera comme arguments deux tableaux $C = \left(c_{ij}\right)_{\substack{i\in\{1,\dots,n\}\\ j\in\{1,\dots,k\}}}$ et $D = (d_i)_{i\in\{1,\dots,n\}}$ décrivant un système linéaire comme ci-dessus et en renverra une solution.*

Les calculs décrits plus haut s'effectuent sans encombre de manière exacte. Menés numériquement, ils sont en revanche sujets aux limitations des nombres flottants. En particulier, la condition $c_{\ell m} \neq 0$ n'est plus satisfaisante : si $c_{\ell m}$ est suffisamment petit, son inverse sera trop grand et ruinera le reste des calculs. On remplacera alors avantageusement la condition $c_{im} \neq 0$ à l'étape 4 par $|c_{im}| > |c_{\ell m}|$ afin d'obtenir à l'étape 6 un *pivot* $c_{\ell m}$ le moins nul possible.

Exercice. *Comparer les solutions des deux systèmes ci-dessous.*

$$\begin{cases} x + \quad y = 1 \\ x + (1+\varepsilon)y = 1 \end{cases} \qquad \begin{cases} x + \quad y = 1 \\ x + (1+\varepsilon)y = 1 + \varepsilon \end{cases}$$

Chapitre 3

Troisième semestre

3.1 Algorithmique et programmation II (Python)

Cette section est l'aboutissement du cours d'informatique générale. Elle consiste en l'application à des problèmes concrets des différents outils théoriques vus en première année. Ces techniques pourront tout particulièrement être mises à profit dans le cadre de vos TIPE.

3.1.1 Piles

La notion de pile a été brièvement introduite au semestre dernier et nous l'avions implantée en Caml en nous reposant sur le système de typage. Nous allons ici rappeler les caractéristiques de cette structure de donnée puis l'implanter de manière plus pragmatique en Python avant de considérer quelques unes de ses applications.

Définition. *Une pile 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 soit le premier à être enlevé.*

Concrètement une pile peut être implantée comme une liste chaînée (revoir la figure 1.6) à laquelle on rajoute ou enlève le premier objet. En Python, on pourrait naïvement écrire :

```
def empiler(L,x):
    return [x]+L
```

```
def depiler(L):
    return L[0],L[1:]
```

Attention toutefois : Python est ainsi fait que ces opérations recopient entièrement la liste à chaque appel et incombent ainsi une une complexité en $O(n)$. Une implantation optimale avec insertion et suppression en $O(1)$, comme décrit ci-dessus avec les listes chaînées, est disponible en Python par des méthodes dédiées :

```
L=[]
L.append(42)
L.pop()
```

Exercice. *Que fait le programme suivant ?*

```
def mystere(L):
    M=[]
    while L: M.append(L.pop())
    return M
```

Exercice. *Soit une pile contenant initialement* $[1, 2, 3, \ldots, n]$. *Tant qu'elle n'est pas réduite à un singleton, on dépile deux entiers x et y puis on empile $x + y$. Quel entier contient-elle alors ?*

Exercice. *Soit un programme prenant comme argument un arbre, créant une pile contenant initialement sa racine puis, tant qu'elle n'est pas vide, dépile un nœud et empile ses fils. Montrer que cela réalise un parcours en profondeur.*

Profitons de cette section pour rappeler la notion de file, structure de donnée en tous points identique à celle de pile excepté en ce que les objets sont enlevés dans l'ordre d'ajout et non pas dans l'ordre inverse. Nous l'avions l'an dernier implantée à l'aide d'un couple de listes, ce que nous proposons maintenant de réaliser à nouveau en Python.

Exercice. *Implanter une structure de file en Python.*

```
def enfiler(F,x):
    a,b=F
    a.append(x)

def defiler(F):
    a,b=F
    if not b:
        while a: b.append(a.pop())
    x=b.pop()
    return x
```

3.1.2 Récursivité

Les systèmes d'exploitation utilisent notamment des piles pour gérer les appels de fonctions, en particulier lorsque celles-ci sont récursives. Nous avions présenté l'an dernier ce paradigme de programmation permettant d'écrire des fonctions s'utilisant elles-mêmes. En pratique il est réalisé en s'appuyant sur une structure de donnée appelée *pile d'exécution*.

Définition. *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.

Exemple. *Une fonction factorielle naïve aurait la pile d'exécution de la figure 3.1.*

Nous ne préciserons pas davantage les considérations pratiques liés à la mise en œuvre de cette technique, mais mentionnerons qu'elle permet de traiter un nombre arbitraire d'appels à un nombre arbitraire de fonctions, ces quantités n'étant limitées que par l'espace mémoire alloué à la pile. Voir notamment la figure 3.2 pour un exemple avec récursion double. On peut facilement afficher la pile d'exécution de tout processus, typiquement afin de diagnostiquer un problème :

```
#0 0x00007fcd05287d1c in __strncpy_ssse3 ()
#1 0x0000000000473507 in strncpy (__len=256, __src=0x0, __dest=0x7fff...
#2 mutt_encode_path (dest=0x7fff6846be70 "", dlen=256, src=<optimized...
#3 0x000000000048d401 in bcache_path (account=<optimized out>, mailbo...
#4 0x000000000048d50c in mutt_bcache_open (account=0x7fff6846c010, ma...
#5 0x000000000048971e in pop_open_mailbox (ctx=0x1204d10)
#6 0x0000000000443b92 in mx_open_mailbox (path=path@entry=0x7fff6846c...
#7 0x0000000000408de0 in main (argc=1, argv=<optimized out>)
```

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

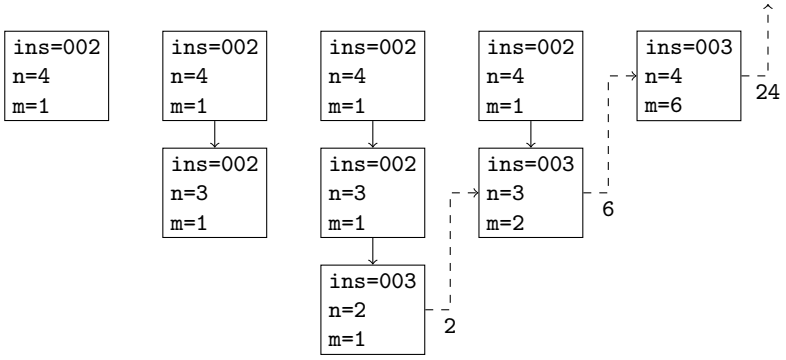


FIGURE 3.1 – Évolution de la pile d'exécution pour `factorielle(4)`.

```
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
```

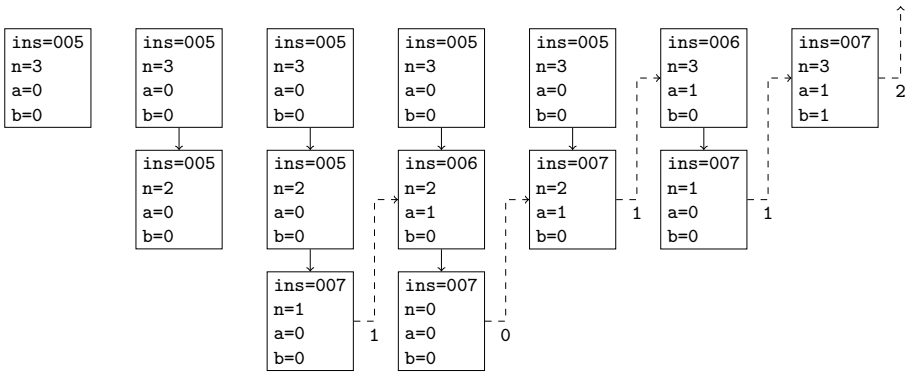


FIGURE 3.2 – Évolution de la pile d'exécution pour `fibonacci(3)`.

On retiendra que, comme tout autre paradigme de programmation, la récursivité n'a rien de mystérieux : son exécution nécessite un travail logiciel s'appuyant sur des structures de données non triviales. Aussi, si elle facilite l'écriture de programmes, elle entrave leur analyse. En présence de tels paradigmes on procèdera donc avec beaucoup de soin afin, par exemple, de ne pas méprendre comme linéaire la complexité d'une fonction doublement récursive.

Exercice. *Écrire un programme récursif calculant la factorisation d'un nombre entier donné. À l'aide d'une pile, en écrire alors une version non recursive.*

Exercice. *Déterminer la complexité du programme suivant.*

```
def recursif(n):
    r=1
    for i in range(1,n):
        r+=recursif(i)
    return r
```

Écrire un programme plus rapide calculant la même quantité.

3.1.3 Tris

L'an dernier nous avions déjà étudié et implanté certaines méthodes de tri. Nous allons ici les revoir en Python ainsi qu'en introduire d'autres avant de comparer ces méthodes entre elles et de présenter diverses applications.

Exercice (tri par sélection). *Le tri par sélection consiste à rechercher le plus petit élément dans la liste donnée, à le placer en première position, puis à trier récursivement le reste de la liste. Écrire un programme Python non récursif implantant cette méthode.*

```
def tri(L):
    for n in range(0,len(L)):
        p=n
        for k in range(n+1,len(L)):
            if L[p]>L[k]: p=k
        (L[p],L[n])=(L[n],L[p])
    return L
```

Exercice (tri par insertion). *Le tri par insertion consiste à insérer le n^e élément de la liste donnée dans la sous liste $L[0:n]$ au préalable récursivement triée. Écrire un programme Python non récursif implantant cette méthode.*

```
def tri(L):
    for n in range(1,len(L)):
        i=n
        while i>0 and L[i]<L[i-1]:
            (L[i],L[i-1])=(L[i-1],L[i])
            i=i-1
    return L
```

Exercice (tri fusion). *Le tri fusion consiste à découper la liste donnée en deux, trier récursivement chaque moitié, puis fusionner les deux listes triées. Écrire un programme Python récursif implantant cette méthode.*

```
def fusion(L,M):
    R=[]
    while L or M:
```

```
        if not L: x=M.pop()
        elif not M: x=L.pop()
        elif L[-1]<M[-1]: x=M.pop()
        else: x=L.pop()
        R.append(x)
    R.reverse()
    return R

def tri(L):
    n=len(L)
    if n<2: return L
    A=tri(L[0:n//2])
    B=tri(L[n//2:n])
    return fusion(A,B)
```

Exercice (tri rapide). *Le tri rapide consiste à sélectionner un élément de la liste donnée appelé pivot, typiquement $L[n/2]$, à partitionner la liste en ses éléments inférieurs, égaux et supérieurs au pivot, puis à trier récursivement chaque partie. Écrire un programme Python récursif implantant cette méthode.*

```
def tri(L):
    n=len(L)
    if n<2: return L
    p=L[n//2]
    A=tri([x for x in L if x<p])
    B=tri([x for x in L if x>p])
    return A+[x for x in L if x==p]+B
```

Nous allons à présent comparer ces quatre méthodes de tri suivant trois métriques subtilement différentes, chacune reflétant un aspect particulier. En effet, dans certaines applications, la liste donnée est déjà triée ou presque et il est alors intéressant de choisir un algorithme optimisé pour cette situation.

Définition. *Soit $\mathscr{A}$ un algorithme dépendant d'un paramètre $x \in X$. On note C_n l'ensemble formé du nombre d'instructions utilisées par l'exécution de $\mathscr{A}(x)$ lorsque x parcours l'ensemble des paramètres de longueur n. On appelle :*

- ***complexité dans le cas le meilleur*** *le minimum de C_n ;*
- ***complexité dans le cas moyen*** *la moyenne de C_n ;*
- ***complexité dans le cas le pire*** *le maximum de C_n.*

Cette dernière notion s'apparente avec celle de borne de complexité fine *définie l'an dernier.*

La figure 3.3 indique les complexités des algorithmes de tri vus en cours ; elle inclue notamment la notion de complexité dans le cas moyen qui est hors programme mais néanmoins mentionnée par soucis de complétude.

Exercice. *Démontrer les résultats de la seconde et quatrième colonne du tableau ci-dessus.*

Remarque. *La rapidité du tri rapide est cachée dans le grand O : une bonne implantation lui permet, dans le cas moyen, d'effectuer environ trois fois moins d'opérations que le tri fusion.*

Rappelons enfin le résultat ci-dessous obtenu l'an dernier et justifiant pourquoi aucune complexité dans le pire cas ne peut être être inférieure à $O(n\log(n))$, borne néanmoins atteinte par le tri fusion et d'autres tris hors programmes.

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.*

ALGORITHME	MEILLEUR CAS	MOYEN CAS	PIRE CAS
tri par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$
tri par insertion	$O(n)$	$O(n^2)$	$O(n^2)$
tri fusion	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$
tri rapide	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$

FIGURE 3.3 – Complexités de différents algorithmes de tri.

Concluons cette partie avec quelques applications directes des méthodes de tri.

Exercice. *Écrire un programme calculant la médiane d'une liste donnée.*

Exercice. *Écrire un programme aussi efficace que possible prenant en argument deux listes et renvoyant leur intersection.*

3.1.4 Applications

Pour clore le programme d'informatique générale, on se propose d'implanter quelques fonctionnalités très basiques d'un logiciel de traitement d'image. On utilisera à ces fins 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)
```

Une image digitale est stockée nativement comme un tableau bidimensionnel : le coefficient d'indice $[y, x]$ du tableau 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. Les ordonnées croissent donc vers le bas comme le montre la figure 3.4.

Le codage RGB est le plus répandu pour représenter les couleurs ; en particulier, les écrans LCD l'utilisent nativement comme on peut le constater sur la figure 3.5. 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 3.6. 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.

Dans le cas précis de la bibliothèque `matplotlib`, les images sont représentées par 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`.

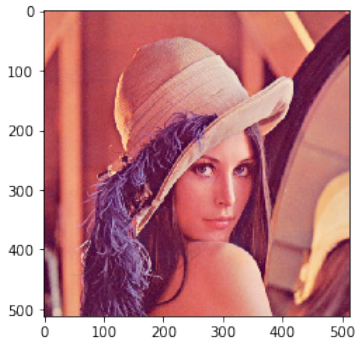
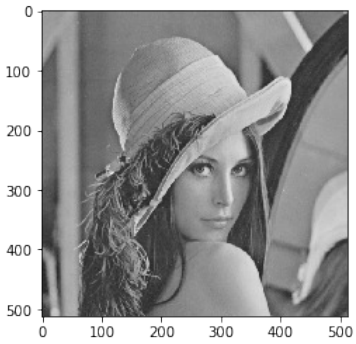
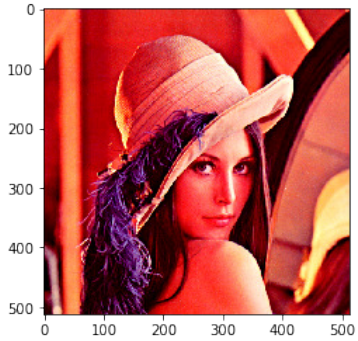


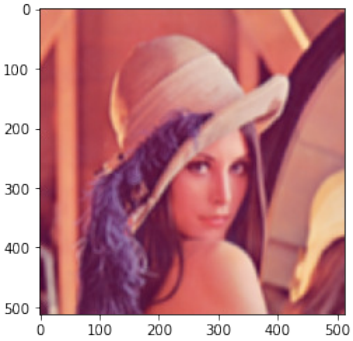
Image originale.



Nuances de gris.



Augmentation du contraste.



Floutage naïf.

FIGURE 3.4 – Quelques opérateurs de traitement d'images appliqués à une célèbre photographie de Lenna Sjööblom.

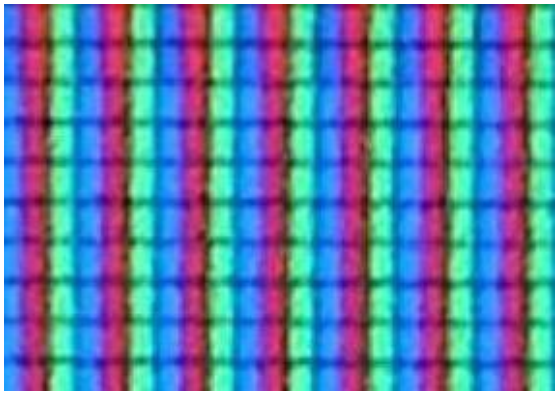


FIGURE 3.5 – Détail d'une matrice de pixels d'un écran LCD.

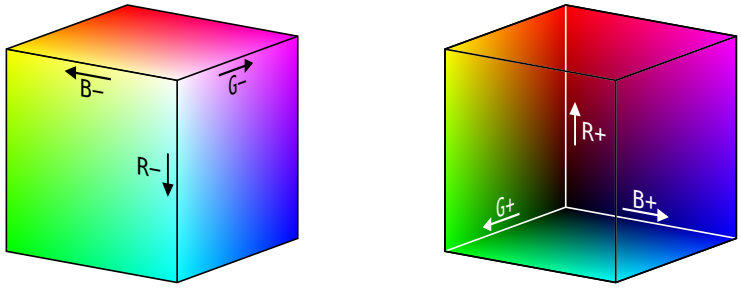


FIGURE 3.6 – Le cube RGB à gauche et ses faces arrières à droite.

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

```
n,m,c = a.shape
assert(c==3)
for y in range(150,200):
    for x in range(0,m):
        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* $\left\lfloor \frac{r+g+b}{3} \right\rfloor$.

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
            if l<e: e=l
            if l>f: 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))
```

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
```

Finissons cette section avec quelques problèmes plus ambitieux.

Exercice. *Proposer puis implanter un programmer permettant de redimensionner des images. On pourra par exemple interpoler linéairement la couleur des pixels lorsqu'il s'agira d'agrandir l'image et moyenner les couleurs des pixels adjacents lorsqu'il s'agira de la réduire. On pourra d'abord créer une image vierge de taille voulue par les incantations :*

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

Exercice. *L'utilisateur souhaite effectuer une série d'opérations consécutives ; plutôt que de les appliquer directement à l'image, il aimerait disposer d'un système lui permettant, à tout moment, d'annuler la dernière opération sans pour autant perdre les précédentes. Expliquer comment une pile permet de répondre à ce besoin. Implanter un tel système.*

Exercice. *On considère que deux couleurs sont indistinguables par l'œil humain lorsque leurs triplets (r, g, b) sont à distance au plus s, un entier paramétrable par l'utilisateur. Proposer puis implanter une méthode de compression d'image ne stockant qu'une fois la couleur d'un ensemble de pixels indistinguables. Quelle est le taux de compression moyen pour $s = 2$? Pour $s = 5$?*

3.2 Notions de logique

Nous allons maintenant faire d'une pierre trois coups en traitant des problèmes qui nous permettront de retrouver nos aptitudes de l'année dernière concernant la logique, les arbres et Caml. Nous en profiterons pour introduire les notions duales que sont la syntaxe et la sémantique.

3.2.1 Calcul propositionnel

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 (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

Notant « $\equiv$ » l'équivalence des propositions, on rappelle les identités classiques :

$$
\begin{aligned}
p \wedge p &\equiv p & \neg(p \wedge q) &\equiv \neg p \vee \neg q \\
p \vee p &\equiv p & \neg(p \vee q) &\equiv \neg p \wedge \neg q \\
p \wedge q &\equiv q \wedge p & p \wedge (\neg p) &\equiv F \\
p \vee q &\equiv q \vee p & p \vee (\neg p) &\equiv V \\
p \wedge (q \wedge r) &\equiv (p \wedge q) \wedge r & \neg(\neg p) &\equiv p \\
p \vee (q \vee r) &\equiv (p \vee q) \vee r & p \Rightarrow q &\equiv \neg p \vee q \\
p \wedge (q \vee r) &\equiv (p \wedge q) \vee (p \wedge r) & p \Leftrightarrow q &\equiv (p \wedge q) \vee (\neg p \wedge \neg q) \\
p \vee (q \wedge r) &\equiv (p \vee q) \wedge (p \vee r) & p \oplus q &\equiv (p \wedge \neg q) \vee (\neg p \wedge q)
\end{aligned}
$$

Si la maîtrise du calcul des propositions et des prédicats enseigné l'an dernier est l'un des prérequis de ce chapitre ainsi que, soit dit en passant, de votre réussite aux concours, il n'a pas pour autant vocation à être repris. L'objectif est ici de développer ces concepts dans une nouvelle direction, plus formelle et qui nous amènera à distinguer deux concepts clefs en informatique présentant la même dualité que le fond et la forme en littérature.

Définition. *On appelle sémantique d'un langage l'étude de la signification de ses écrits.*
On appelle syntaxe d'un langage l'étude de ses règles d'écriture.

Par exemple, en langage Python, les écrits ne sont autres que les programmes et une erreur de type `IndentationError` dénote un problème de syntaxe alors qu'une erreur de type `IndexError` relève quant à elle de la sémantique. En calcul propositionnel, les écrits sont les propositions, le bon parenthésage relève de la syntaxe et les tables de vérité de la sémantique.

Exercice. *Définir un type Caml 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 type « `proposition -> bool array -> bool` » évaluant une telle proposition en des valeurs de vérité données.*

```
type prop =
| V
| F
| Var of int
| Non of prop
```

```
let rec eval v = function
| V -> true
| F -> false
| Var(i) -> v.(i)
| Non(p) -> not (eval v p)
```

```
| Et of prop*prop                | Et(p,q) -> (eval v p) && (eval v q)
| Ou of prop*prop                | Ou(p,q) -> (eval v p) || (eval v q)
;;                               ;;
    (* syntaxe *)                    (* sémantique *)
```

Exercice. *Écrire une fonction calculant la négation d'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 $x_1, \ldots, x_n$ 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.*
Écrire une fonction déterminant si une proposition est satisfiable.
Quelles 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 puissance x = function
| 0 -> 1
| n -> let y = puissance (x*x) (n/2) in
       if n mod 2 = 1 then x*y else y
;;

let binaire n k =
    let v = Array.make n false in
    let x = ref k in
    for i=0 to n-1 do
        if !x mod 2 = 1 then v.(i) <- true;
        x := !x / 2;
    done;
    v
;;

let satiasfiable p =
    let n = maxvar p + 1 in
    let r = ref false in
    for k=0 to (puissance 2 n - 1) do
        let v = binaire n k in
        if (eval v p) then r:=true;
    done;
    !r
;;
```

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) équivalente.*

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 équivalente à une proposition donnée.*

3.2.2 Calcul formel

Les techniques mises en œuvre ci-dessus pour représenter et manipuler des propositions logiques s'appliquent plus généralement à toute expression symbolique. Elles forment l'un des piliers du calcul formel, domaine ayant pour objectif de développer et d'étudier les méthodes permettant de manipuler des objets mathématiques de manière exacte, par opposition au calcul approché. Ce domaine est très vaste, aussi nous concentrerons notre étude sur deux cas particuliers, celui des expressions arithmétiques et celui des fonctions élémentaires.

Exercice. *Définir un type Caml permettant de représenter les nombres rationnels.*

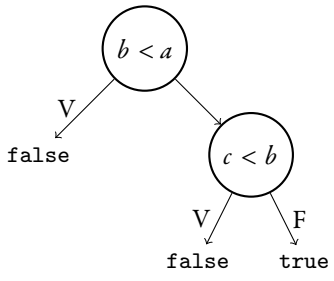
Exercice. *Définir un type Caml permettant de représenter les expressions arithmétiques composées de variables x_i avec $i \in \mathbb{N}$, de constantes rationnelles, ainsi que des opérations d'addition, soustraction, de division et de multiplication.*

Écrire une fonction évaluant une telle expression en des valeurs données.

Exercice. *Écrire une fonction s'efforçant de simplifier une expression arithmétique.*

On relèvera à nouveau la différence et dualité entre la syntaxe, matérialisée par la structure arborescente, et la sémantique, matérialisée par le processus d'évaluation qui interprète les opérations afin d'assigner des valeurs aux expressions.

Définition. *On qualifie d'élémentaire toute fonction d'une variable réelle construite en composant un nombre fini de constantes, fonctions identités, exponentielles, logarithmes, sinus, cosinus, sommes, différences, quotients et produits.*




```
let croissant a b c =
    if b<a then false else
        if c<b then false else true
;;
```

FIGURE 3.7 – Un algorithme et son arbre de décision.

Exercice. *Définir un type Caml permettant de représenter cette classe de fonctions.*
Écrire une fonction évaluant une telle fonction en un réel donné.
Écrire une fonction dérivant une fonction donnée.

Le problème de l'intégration est de difficulté incomparable, comme vous avez pu vous en apercevoir en cours d'analyse. Informons le lecteur curieux que le problème de déterminer, lorsqu'il en existe, une primitive élémentaire d'une fonction élémentaire a été complètement résolu [20] ; les techniques mathématiques mises en œuvre pour ce faire sont toutefois profondes et dépassent largement le cadre du programme des classes préparatoires. Abordons néanmoins ce problème en nous restreignant à une sous classe de fonctions bien choisies.

Exercice. *Écrire une fonction déterminant si une fonction élémentaire donnée est un polynôme.*
Écrire une fonction renvoyant le tableau des coefficients d'un polynôme donné sous cette forme.
Écrire enfin une fonction intégrant une fonction élémentaire donnée lorsque c'est un polynôme.

3.3 Structures de données et algorithmes II

Poursuivons maintenant notre étude des arbres initiée en première année.

3.3.1 Arbres de décision

Commençons par introduire un outil descriptif utile pour l'analyse d'algorithmes.

Définition. *On appelle arbre de décision d'un algorithme l'arbre dont chaque nœud correspond à un état d'exécution (instruction courante et valeur des variables) précédant un branchement conditionnel, ses fils correspondants aux états pouvant résulter de ce branchement. La racine correspond au premier branchement conditionnel et les feuilles aux valeurs de retours de l'algorithme.*

Voir la figure 3.7 pour un exemple simple.

Ces objets facilitent notamment les raisonnements élégants sur des algorithmes composés essentiellement de branchements conditionnels. Pour illustrer cela nous allons démontrer à nouveau le résultat ci-dessus.

Théorème. *Soit un algorithme permettant de trier les éléments d'un tableau suivant un ordre arbitraire. Sa complexité est minorée par $n \log_2(n)$ où n dénote la longueur du tableau.*

Démonstration. On modélise l'exécution de l'algorithme par un arbre de décision : chaque nœud correspond à un état de l'algorithme menant à une comparaison entre deux éléments du tableau (x et y) ; il possède deux fils correspondant chacun à un résultat possible ($x < y$

et $y < x$). Cet arbre possède au moins $n!$ feuilles, chacune correspondant à une permutation possible du tableau. Sa hauteur, c'est-à-dire la complexité de l'algorithme dans le cas le pire, est donc minorée par $\log_2(n!)$ d'où le résultat. □

3.3.2 Arbres de recherche

Le penchant effectif des arbres de décision (concept essentiellement descriptif) est celui d'arbre de recherche, une structure de donnée réalisant de manière plus explicite la méthode de recherche par dichotomie vue au premier semestre.

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.*

Rappelons le type Caml suivant défini l'an dernier :

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

La structure d'arbre de recherche permet, en procédant de manière semblable à la dichotomie, de déterminer l'appartenance d'un élément à l'ensemble des étiquettes en $O(h)$ comparaisons dans le cas le pire.

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

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

```
let recherche =
    let rec aux a b =
        function
        | Nil -> false
        | Noeud(e,g,d) -> a<=e && e<=b && aux a e g && aux e b d
    in
    aux 0 99999
;;
```

3.3.3 Tas

La classe d'arbres ci-dessous est elle aussi particulièrement intéressante.

Définition. *On appelle tas tout arbre binaire dont l'étiquette de chaque nœud est supérieure à celles de ses fils.*

Cette structure permet d'obtenir directement le nœud d'étiquette maximale : c'est la racine ! Elle est donc particulièrement adaptée à l'implantation de files de priorité où il s'agit de traiter un ensemble de tâches en commençant par celle de priorité maximum.

Exercice. *Écrire une fonction déterminant si un arbre quelconque est un tas.*

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

Lorsqu'on ajoutera (resp. enlèvera) un nœud à un tas, afin d'en préserver la structure, on prendra soin de faire récursivement redescendre (resp. remonter) les nœuds d'étiquette minimale (resp. maximale). Ce processus est appelé appelé *percolation*.

Exercice. *Écrire une fonction de type « `'a -> 'a arbreBE -> 'a arbreBE` » renvoyant un tas dont l'ensemble des étiquettes est identique à celui du tas donné mais avec un nœud en plus.*

Exercice. *Écrire une fonction de type « `'a arbreBE -> 'a arbreBE` » renvoyant un tas dont l'ensemble des étiquettes est identique à celui du tas donné mais avec sa racine en moins.*

```
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)
;;
```

Extraire dans l'ordre décroissant les étiquettes d'un tas nécessite ainsi $O(ht)$ opérations. Comme dans le cas des arbres de recherche, la complexité dépend de la hauteur des arbres considérés et il est donc opportun d'essayer de la minimiser à taille constante.

3.3.4 Arbres équilibrés

Pour obtenir un bon rapport entre la hauteur h d'un arbre et sa taille t (autrement dit, le cardinal de l'ensemble des étiquettes), nous allons maintenant 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.*

Exercice. *Écrire une fonction déterminant si un arbre binaire est équilibré.*

```
let equilibre =
    let rec aux =
        function
        | Nil -> -1
        | Noeud(e,g,d) ->
            let x = aux g in
```

```
            let y = aux d in
            if abs(x-y)<2 then max x y + 1 else 99999
    in
    function x -> aux x < 99999
;;
```

L'équilibrage d'un arbre est donc un critère facile à vérifier effectivement et, nous allons le voir, qui garantit la propriété escomptée concernant l'optimalité de sa hauteur.

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

Démonstration. Soit E_h un arbre équilibré de hauteur h et de taille minimale ; l'arbre constitué d'une racine ayant E_h comme sous arbre gauche et E_{h+1} pour sous arbre droit est équilibré, de hauteur $h+2$ et minimal. La taille minimale t_h d'un arbre équilibré de hauteur h vérifie donc $t_{h+2} = t_{h+1} + t_h + 1$. Cela revient à $(t_{h+2}+1) = (t_{h+1}+1) + (t_h+1)$ d'où on déduit $t_h + 1 = \lambda\phi^h + \mu(1-\phi)^h$ pour $\phi = \frac{1+\sqrt{5}}{2}$ avec $\lambda = 1 + \frac{2}{\sqrt{5}}$ et $\mu = 1 - \frac{2}{\sqrt{5}}$. □

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

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 auto-ré-équilibrants.

3.3.5 Arbres tassés

On peut faire mieux que les arbres équilibrés en ce qui concerne le rapport hauteur/taille au prix de perdre les facilités de ré-équilibrage après insertion.

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* $2n+2$.

On dit alors qu'un arbre est tassé à gauche *lorsque sa représentation tabulaire est connexe. Voir les figures 3.8 et 3.9.*

De nombreux auteurs considèrent implicitement que tout tas est tassé à gauche et nous adopterons dorénavant cette convention.

Exercice. *Écrire une fonction de type « `'a arbreBE -> 'a array` » renvoyant la représentation tabulaire d'un arbre binaire. En déduire une fonction de type « `'a arbreBE -> bool` » déterminant si un arbre binaire est tassé.*

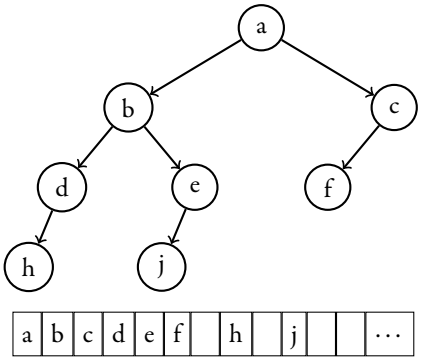


FIGURE 3.8 – Un arbre binaire non tassé et sa représentation tabulaire.

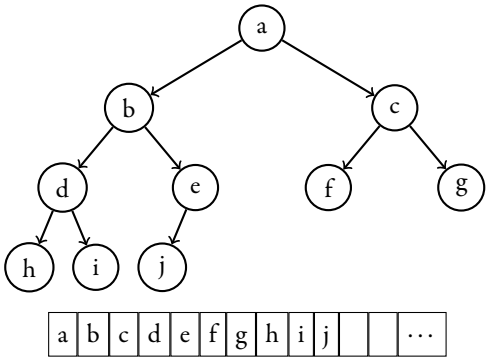


FIGURE 3.9 – Un arbre binaire tassé et sa représentation tabulaire.

```
let tableau a =
    let h = hauteur a in
    let n = puissance 2 (h+1) in
    let v = Array.make n (-1) in
    let rec aux i =
        function
        | Nil          -> ()
        | Noeud(e,g,d) ->
              v.(i) <- e;
              aux (2*i+1) g;
              aux (2*i+2) d;
    in
    aux 0 a;
    v
;;

let tasse a =
    let v = tableau a in
    let n = Array.length v in
    let t = ref true in
    for i=1 to n-1 do
        if v.(i-1)<0 && v.(i)>=0 then t:= false;
    done;
    !t
;;
```

Exercice. *Écrire une fonction de type « `'a array -> ()` » percolant un tas binaire donné par sa représentation tabulaire.*

```
let percoler v =
    let i = ref 0 in
    while v.(!i)>=0 do
        let g=v.(!i*2+1) in
        let d=v.(!i*2+2) in
        if g>d then begin
            v.(!i) <- g;
            i:=!i*2+1;
        end else begin
            v.(!i) <- d;
            i:=!i*2+2;
        end
    done;
    v.(!i/2) <- -1;
;;
```

3.4 Graphes

3.4.1 Aspects élémentaires

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 conceptuellement qu'un ensemble d'arcs reliant des nœuds.

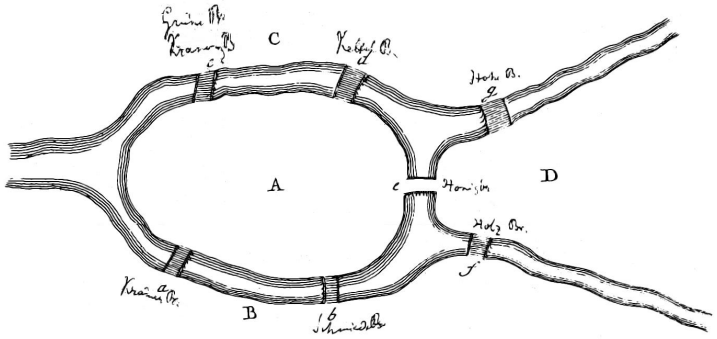


FIGURE 3.10 – Représentation synthétique des cours d'eau et des ponts du centre ville de Königsberg; extrait du manuscrit original d'Euler [1].

Définition. *On appelle graphe tout couple* (S, A) *vérifiant* $A \subset S \times S$ *; les éléments de* S *sont appelés les sommets et ceux de* A *les arêtes. Par convention aucun graphe considéré ne comportera d'arête (dite* boucle*) du type* (x, x)*.*

L'ensemble A s'interprète aussi comme une relation binaire (irréflexive) sur S.

Définition. *Un graphe est dit non orienté lorsque la relation induite est symétrique, c'est-à-dire lorsqu'il vérifie* $(x, y) \in A \Rightarrow (y, x) \in A$*.*

On se convainc aisément du large champ d'application que cette notion peut avoir mais mentionnons néanmoins quelques exemples concrets parmi les plus pertinents :

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

Exercice. *Déterminer lesquels de ces exemples sont non orientés. Donner un ordre de grandeur pour le cardinal de* S *dans chacun des cas.*

Problème (sept ponts de Königsberg)**.** *Du temps d'Euler, au* XVIII^{e} *siècle, la ville de Königsberg au royaume de Prusse comportait sept ponts disposés comme l'indique la figure 3.10. 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 3.11. Il nous faut à présent introduire quelques définitions et développer quelques outils afin d'analyser ce graphe en vue de résoudre le problème posé par Euler.

Définition. *On appelle degré sortant d'un sommet* x *le nombre d'arêtes de la forme* (x, y)*.*
On appelle degré entrant d'un sommet y *le nombre d'arêtes de la forme* (x, y)*.*

Dans le cas non orienté ces notions coïncident et on parle simplement de *degré*.

Proposition. *Si* (S, A) *est un graphe non orienté alors* $\sum_{s \in S} \deg(s) = 2|A|$.

Formalisons aussi la notion de chemin.

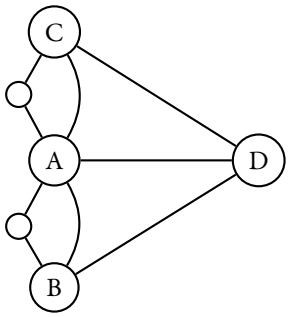


FIGURE 3.11 – Structure de graphe associée à la figure 3.10. Afin d'éviter les arêtes multiples, on rajoute un sommet au milieu des deux ponts de gauche.

Définition. *Un chemin est une famille finie de sommets* $(s_0, s_1, \dots, s_n)$ *vérifiant* $(s_k, s_{k+1}) \in A$ *pour tout* k. *On le note* $s_0 \to s_1 \to \dots \to s_n$ *et dit qu'il est de longueur* n.

Définition. *Soit* (S, A) *un graphe. On définit une relation d'équivalence* $\mathscr{R}$ *sur* S *en posant* $x \mathscr{R} y$ *lorsqu'il existe un chemin de* x *à* y *et un chemin de* y *à* x. *Ses classes partitionnent* S *et sont appelées composantes connexes fortes. Un graphe est dit fortement connexe s'il admet une unique composante connexe forte.*

Pour un graphe non orienté, $x \mathscr{R} y$ équivaut à l'existence d'un chemin de x à y *ou* d'un chemin de y à x ; ce n'est en revanche pas nécessairement le cas pour un graphe orienté et on introduit donc une seconde notion de complexité qui n'est autre que la transitivisée de ce « ou ».

Définition. *Soit* (S, A) *un graphe. On définit une relation d'équivalence* $\mathscr{R}$ *sur* S *en posant* $x \mathscr{R} y$ *lorsqu'il existe une famille* $(s_0, s_1, \dots, s_n)$ *vérifiant*

$$s_0 = x \quad \wedge \quad s_n = y \quad \wedge \quad \forall k \in \{0, \dots, n-1\}, \big((s_k, s_{k+1}) \in A \vee (s_{k+1}, s_k) \in A\big).$$

Ses classes partitionnent S *et sont appelées composantes connexes faibles. Un graphe est dit faiblement connexe s'il admet une unique composante connexe faible.*

Dans le cas non orienté ces notions coïncident et on parle simplement de *connexité*.

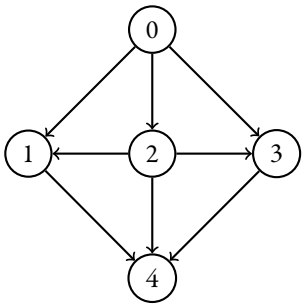
Théorème. *Un graphe connexe non orienté admet un chemin Eulérien, c'est-à-dire passant une fois et une seule par chaque arête, si et seulement s'il admet au plus deux sommets de degré impair.*

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

3.4.2 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 $S \simeq \{0, \dots, n-1\}$.




```
[| [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 3.12 – Un graphe, sa liste d'adjacence et sa matrice d'adjacence.

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 3.12.*

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

Exercice. *Soit un graphe donné par sa liste d'adjacence. Écrire des programmes permettant d'ajouter et de supprimer une arête. Écrire des programmes permettant d'ajouter et de supprimer un sommet. Mêmes questions pour un graphe donné par sa matrice d'adjacence. Borner la complexité de ces huit programmes.*

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.

Exercice. *Écrire un programme renvoyant la liste des composantes connexes faibles d'un graphe donné. Même question concernant la liste des composantes connexes fortes. Borner la complexité de ces deux programmes.*

On pourra employer, comme pour les arbres, la terminologie *parcours en largeur* et *parcours en profondeur* ; les premiers s'implantant typiquement à l'aide d'une file et les seconds d'une pile.

Exercice. *Écrire un programme renvoyant un chemin Eulérien d'un graphe non orienté donné. Borner en la complexité.*

3.4.3 Aspects algébriques

Les résultats élémentaires que voici sont hors programme mais complètent grandement ce qui précède notamment afin de mettre en lumière la pertinence de la représentation matricielle.

Lemme. *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.*

Corollaire. *Soit k le degré du polynôme minimal de la matrice d'adjacence d'un graphe fortement connexe. Tout couple de sommets peut être relié par un chemin de longueur au plus $k-1$.*

Démonstration. Par le lemme ci-dessus, s'il existe un plus petit chemin de longueur k, les matrices A^i pour $i \in \{0, \dots, k\}$ seraient linéairement indépendantes, ce qui est absurde puisqu'elles engendrent un sous espace vectoriel de dimension k. □

3.4.4 Recherche de plus courts chemins

Soient s et t deux sommets d'une même composante connexe d'un graphe (S, A). Un chemin de s à t 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 $s_0 \to s_1 \to \dots \to s_n$ est alors la quantité $p(s_0, s_1) + p(s_1, s_2) + \dots + p(s_{n-1}, s_n)$.*

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

Décrivons puis analysons deux solutions au problème de recherche de plus courts chemins.

Algorithme de Dijkstra [6]. Notons $d(r, t)$ la longueur d'un plus court chemin de r à t. Pour chaque voisin s de t, prenons un plus court chemin de r à s puis rajoutons-y l'arête $s \to t$ afin d'obtenir un chemin de r à t ; le plus court est parmi ceux-ci, ce qui donne l'identité

$$\begin{cases} d(r, r) = 0, \\ d(r, t) = \min\limits_{\substack{s \in S \\ s \neq t}} d(r, s) + p(s, t) \quad \text{si } r \neq t. \end{cases}$$

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

Algorithme (Dijkstra)**.**

Entrée : *Deux sommets α et β d'un graphe pondéré (S, A, p).*

Sortie : *La longueur d'un plus court chemin de α à β.*

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

Chaque itération commence ligne 5 par l'identification d'un somment $s \in E$ minimisant d_s. Afin d'effectuer cette opération efficacement, on peut stocker les éléments de E dans un tas étiquetant le nœud s par la quantité $-d_s$; il sera percolé à chaque suppression d'élément ou modification d'étiquette. Avec un arbre auto-ré-équilibrant, par exemple de type AVL, ces opérations sont de complexité $O(\log|S|)$.

Proposition. *La complexité de l'algorithme de Dijkstra est $O(|A| + |S|\log|S|)$.*

Proposition. *L'algorithme de Dijkstra est correct.*

Démonstration. Notons $d(\alpha, t)$ la distance d'un plus court chemin de α à t. On dispose de l'invariant de boucle suivant, valable en fin d'itération :

$$\begin{aligned} & \forall t \notin E, d(\alpha,t) = d_t \\ \wedge \quad & \forall t \in E, d(\alpha,t) \leqslant d_t = \min\big\{d(\alpha,s) + p(s,t) : s \notin E, (s,t) \in A\big\}. \end{aligned}$$

□

Exercice. *Modifier cet algorithme afin qu'il renvoie non seulement sa longueur mais aussi un plus court chemin.*

Remarquer que l'hypothèse $p(s,t) \geqslant 0$ est cruciale pour la correction de cette méthode. Ceci la distingue de l'approche ci-dessous qui pourra quant à elle être aussi bien appliquée en présence de poids.

Algorithme de Floyd–Warshall [7, 11, 12]. Cette méthode détermine simultanément, pour chaque couple de sommets d'un graphe pondéré donné, la longueur du plus court chemin les reliant. Supposons $S = \{1, \dots, n\}$. L'idée principale est d'introduire la quantité $d_k(i,j)$, longueur du plus court chemin de i à j passant exclusivement par des sommets de $\{1, \dots, k\}$. L'algorithme consiste à calculer cette grandeur récursivement en exploitant l'identité

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

Algorithme (Floyd–Warshall).

Entrée : *Un graphe 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.

Proposition. *La complexité de l'algorithme de Floyd–Warshall est $O(|S|^3)$.*

Exercice. *Démontrer la correction de cet algorithme.*

Exercice. *Modifier cet algorithme pour qu'il renvoie les chemins, non seulement leurs longueurs.*

3.5 Initiation aux bases de données (SQL)

3.5.1 Problématique et langage SQL

Les bases de données sont des structures de données complexes 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.

La place qu'occupe une base de données au sein d'une application logicielle est précisée par le modèle appelé *architecture à trois couches* ; il décrit une application comme la conjugaison de trois modules bien distincts : la couche de *présentation des données*, à laquelle incombe la mise en forme, l'affichage et l'interaction avec l'utilisateur ; la couche de *traitement des données*, c'est-à-dire le cœur de l'application proprement dite ; et la couche d'*accès aux données*, chargée

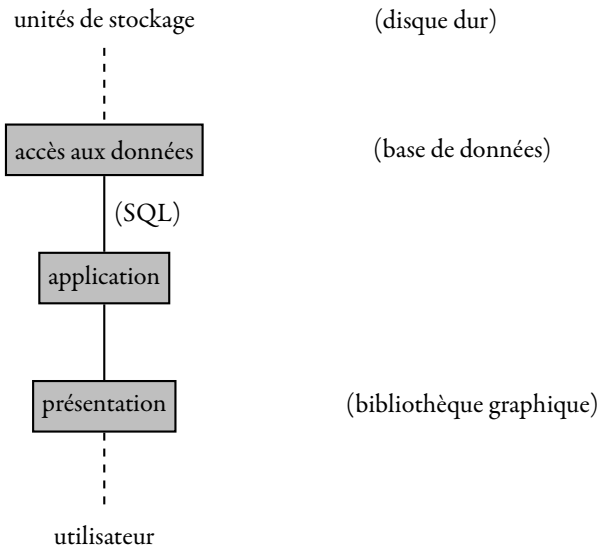


FIGURE 3.13 – Modèle d'architecture à trois couches.

de leur stockage et transmission. Comme la figure 3.13 l'illustre, c'est cette dernière couche, la plus distante de l'utilisateur, qui incorpore typiquement une base de données.

Chaque couche interagit avec ses voisines et l'objectif de cette section est d'apprendre un protocole de communication particulièrement répandu entre la couche applicative et la couche d'accès aux données : le langage SQL (Structured Query Language) inventé en 1970 pour interagir avec les bases de données structurées suivant le *modèle relationnel*, un mode d'organisation des données alors nouvellement inventé [18]. Des constructions plus anciennes adoptaient notamment le modèle hiérarchique et le modèle réseau où les données sont respectivement organisées sous forme d'arbre et de graphe ; elles ont depuis été largement supplantées par le modèle relationnel.

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.*

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. *La base de données utilisée par Twitter pourrait ressembler à la figure 3.14. Dans la relation* `Users`*, les attributs* `Id` *et* `Pseudo` *pourraient chacun servir de clef primaire, mais pas les attributs* `Nom` *ou* `Prenom`*.*

Dans une feuille de calcul de type Sqlite3 exécuter les instructions suivantes :

```
.open twitter.db
```

Relation `Users`

ID	NOM	PRENOM	PSEUDO
44	Obama	Barack	BarackObama
45	Trump	Donald	realDonaldTrump

Relation `Events`

CLEF	DATE	HEURE
896523232098078720	2017-08-12	14:06
516382177798680576	2014-09-28	21:21
257552283850653696	2012-10-14	08:43

Relation `Tweets`

CLEF	ID	MESSAGE
896523232098078720	44	No one is born hating another person because of the color of his skin.
516382177798680576	45	Windmills are the greatest threat in the US to both bald and golden eagles.
257552283850653696	45	I have never seen a thin person drinking Diet Coke.

FIGURE 3.14 – Base de donnée pouvant ressembler à celle utilisée par Twitter.

```
CREATE TABLE Users (Id INTEGER, Nom TEXT, Prenom TEXT, Pseudo TEXT);
INSERT INTO Users VALUES(44,'Obama','Barack','BarackObama');
INSERT INTO Users VALUES(45,'Trump','Donald','realDonaldTrump');
UPDATE Users SET Prenom='Donald J.' WHERE Id=45;
SELECT Prenom,Nom FROM Users WHERE Id=45;
```

La première ligne indique que l'on souhaite travailler sur une base de donnée stockée dans un fichier, plutôt qu'en mémoire vive ; cela nous permettra de reprendre notre travail plus tard. Chaque ligne suivante est une transaction, c'est-à-dire une instruction que la base de données exécute en garantissant que, soit elle échoue et la base n'est pas modifiée, soit elle réussit et la base est modifiée exactement comme spécifié.

Exercice. *Sachant que les domaines supportés incluent notamment `BOOLEAN`, `INTEGER`, `FLOAT`, `DATE`, `TIME`, `TEXT` et `BINARY`, créer des relations `Events` et `Tweets` comme ci-dessus.*

Remarque. *Pour consulter l'état d'une base de données on peut à tout moment utiliser les commandes « `.tables` » qui énumère les relations et « `.dump table` » qui affiche les commandes SQL nécessaire à recréer de zéro l'état actuel d'une relation.*

Afin de modifier les données d'une base, on dispose des transactions SQL suivantes :

— « `INSERT INTO table (attributs) VALUES (valeurs)` » ajoute un tuple à la relation. Lorsqu'une liste d'attributs est donnée elle est utilisée pour placer les valeurs.
— « `UPDATE table SET attribut=valeur WHERE condition` » affecte la valeur à l'attribut pour tous les tuples de la relation vérifiant la condition.
— « `DELETE FROM table WHERE condition` » supprime les tuples concernés.

La transaction la plus importante est cependant celle permettant d'extraire des données d'une ou plusieurs relations. Sa syntaxe épurée est « `SELECT attributs FROM tables` ». Les attributs peuvent être spécifiées par leur nom `Nom` ou de manière plus complète sous la forme `Users.Nom` et peuvent utiliser l'astérisque pour dénoter toutes les possibilités `Users.*` ou simplement `*`. Cette requête peut être raffiné en la suffixant des clauses suivantes :

— « `WHERE conditions` » sélectionne les tuples vérifiant la condition donnée.
— « `GROUP BY expression` » agrège les tuples dont l'expression est identique.
— « `HAVING conditions` » sélectionne les agrégats vérifiant la condition donnée.
— « `ORDER BY expression` » trie les résultats suivant l'expression croissante.

3.5.2 Algèbre relationnel

Avant d'aller plus loin, formalisons algébriquement la notion de requête. On notera pour ce faire une relation comme l'ensemble de ses tuples. Commençons par définir les opérations ensemblistes déjà bien connus.

Définition. *Soient R et S deux relations de même schéma. Leur union est la relation $R \cup S = \{t : t \in R \vee t \in S\}$. Leur intersection est la relation $R \cap S = \{t : t \in R \wedge t \in S\}$. Leur différence est la relation $R \setminus S = \{t : t \in R \wedge t \notin S\}$.*

Ces opérations correspondent respectivement, en langage SQL, aux mots clefs `UNION`, `INTERSECT` et `EXCEPT` ; elles seront toutefois rarement utilisées.

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 afficher le prénom et le nom de tout utilisateur ayant posté un message de longueur inférieure à soixante caractères, on écrira en langage SQL :*

```
SELECT Prenom,Nom FROM Users,Tweets
WHERE Users.Id=Tweets.Id
AND LENGTH(Message)<60;
```

Exercice. *Écrire une requête SQL affichant le prénom de tout utilisateur ayant posté après 2015.*

```
SELECT Prenom FROM Users,Tweets,Events
WHERE Users.Id=Tweets.Id
AND Tweets.Clef=Events.Clef
AND Date>'2015-12-31';
```

Passons à présent aux opérations spécifiques aux bases de données relationnelles.

Définition. *Soit R et S deux relations. La sélection $\sigma_P(R)$ est la relation formée des tuples de R qui satisfont la condition P. La projection $\pi_A(R)$ est la relation R restreinte aux attributs du sous-ensemble A. La renommée $\rho_{b/a}(R)$ est la relation R avec l'attribut a renommée en b. 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.*

Nous avons déjà manipulé les deux premières opérations et allons prochainement développer les deux suivantes. En langage SQL, ces quatre opérations correspondent respectivement aux mots clefs `WHERE`, `SELECT`, `AS` et `JOIN`. L'expression de l'algèbre relationnel

$$\pi_{\texttt{Prenom,Nom}}(\sigma_{\texttt{Id=45}}(\texttt{Users}))$$

correspond donc à la transaction du langage SQL :

```
SELECT Prenom,Nom FROM Users WHERE Id=45;
```

L'opération de jointure est équivalente à la composition d'une sélection avec un produit cartésien. Elle est toutefois bien plus efficace car elle fusionne plusieurs relations en temps quasi linéaire en leurs longueurs. La requête ci-dessus affichant le prénom de tout utilisateur ayant posté après 2015 peut donc être écrit de manière équivalente mais plus efficace :

```
SELECT Prenom FROM Users
JOIN Tweets ON Users.Id=Tweets.Id
JOIN Events ON Tweets.Clef=Events.Clef
WHERE Date>'2015-12-31';
```

3.5.3 Autres opérations

Le langage SQL dispose aussi d'opérations qui sortent du cadre strict de l'algèbre relationnel. Nous en détaillerons ici deux sortes qui sont particulièrement répandues.

Projections étendues. Ces opérations consistent à renvoyer non pas directement les valeurs des attributs sélectionnés mais le résultat d'expressions arithmétiques évaluées en ces valeurs. On peut par exemple écrire :

```
SELECT Prenom, Nom, Credit-Debit AS Solde FROM Banque;
SELECT Produit, PrixHT*(1+TVA) AS PrixTTC FROM Facture;
```

Fonctions d'agrégats. Une fonction d'agrégat notée `AGR(expr)` est évaluée, pour chaque agrégat, sur l'ensemble des valeurs prises par l'expression `expr` dans cet agrégat. En SQL on dispose notamment des mots clefs `COUNT`, `MIN`, `MAX`, `SUM` et `AVG` qui calculent notamment le nombre de valeurs non nulles, le minimum, le maximum, la somme et la moyenne. Plutôt que `COUNT(expr)` et préfèrera généralement utiliser `COUNT(*)` qui renvoie simplement le nombre de tuples par agrégat. On peut donc écrire :

```
SELECT AVG(Credit),AVG(Debit) FROM Banque;
```

Ces fonctions renvoient une ligne par agrégat. Par défaut, un unique agrégat regroupe tous les tuples et une seule ligne est donc renvoyée. On peut toutefois partitionner les tuples en agrégats plus fins par la clause `GROUP BY`. Afin d'obtenir le solde de chaque famille on écrirait :

```
SELECT SUM(Credit-Debit) FROM Banque GROUP BY Nom;
```

Exercice. *Afficher le prénom et le nom de tout utilisateur ayant posté au moins deux message.*

```
SELECT Prenom,Nom FROM Users
JOIN Tweets ON Users.Id=Tweets.Id
GROUP BY Tweets.Id HAVING COUNT(*)>=2;
```

On peut enfin combiner des requêtes SQL en utilisant des parenthèses comme il suit.

```
SELECT SUM(PrixTTC) FROM (
    SELECT Produit, PrixHT*(1+TVA) AS PrixTTC FROM Facture
);
```

Exemple. *Afin d'obtenir l'identité de l'utilisateur ayant posté le plus grand nombre de tweets, on écrirait :*

```
SELECT Prenom,Nom,Pseudo FROM Users WHERE Id=(
    SELECT Id FROM Tweets GROUP BY Id HAVING COUNT(*) = (
        SELECT MAX(Nombre) FROM (
            SELECT COUNT(*) AS Nombre FROM Tweets GROUP BY Id
        )
    )
);
```

Finissons par manipuler une base de données plus fournie et vraisemblable que nos exemples ci-dessus. Cette base organise les données (fictives) d'un commerce de vente de musique en ligne ; ses onze relations sont illustrées par la figure 3.15. On la chargera en exécutant :

```
.open /home/prof/chinook.db
```

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.*

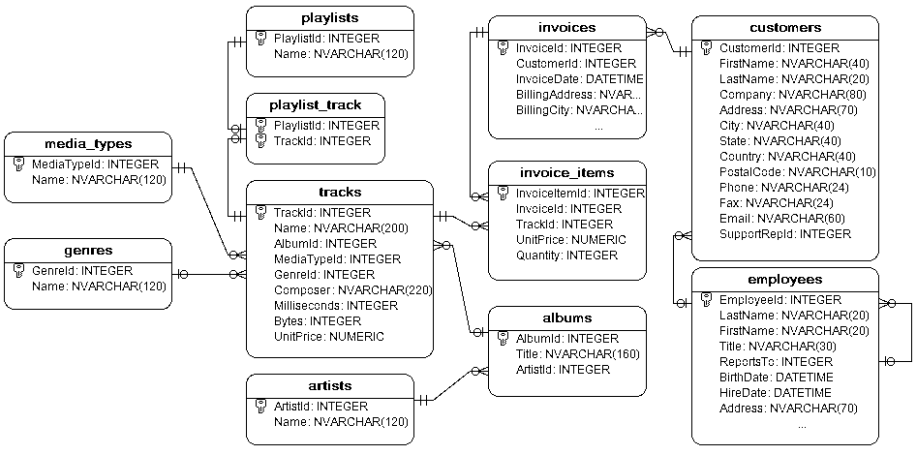


FIGURE 3.15 – Relations de la base de données chinook [36].

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 dont nous montrerons qu'elle n'est en réalité qu'une forme effective de la théorie des langages rationnels.

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 dit *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 finie 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, muni de la relation $x \mathcal{R} y \iff$ « x est un facteur de y », le monoïde libre $\Sigma^\star$ est bien ordonné.*

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.*

Si jusqu'ici seules ces modestes notions ont été introduites c'est que nous allons maintenant aborder le problème de la reconnaissance de motifs en l'attaquant dans un cas bien particulier : celui des langages formés de tous les mots admettant un facteur donné.

Algorithme naïf. Adoptons pour commencer l'approche directe consistant à comparer une à une les lettres du mot avec celles du facteur recherché.

Exercice. *Écrire une fonction Caml 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 r = ref false in
    let m = String.length x in
    let n = String.length y in
    for i=0 to n-m do
        let j = ref 0 in
        while !j<m && x.[!j]=y.[i+!j] do j:=!j+1; done;
        if !j=m then r:=true;
    done;
    !r
;;
```

Dans le cas le pire, 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 l'algorithme ci-dessus peut être un choix judicieux dans certains cas pratiques.

Nous allons néanmoins décrire deux méthodes de complexité $O(m+n)$ dans le cas le pire ; on notera comme ci-dessus $x_1x_2\cdots x_m$ le facteur à rechercher (ou motif) et $y_1y_2\cdots y_n$ le mot dans lequel le rechercher (ou texte).

Algorithme de Knuth–Morris–Pratt [23]. L'algorithme naïf ci-dessus accède à la valeur y_{j+i} pour de multiples couples (i,j) de même somme. Pour éliminer cette redondance, nous allons examiner la structure du motif plus en détail. Supposons que la comparaison entre le texte et le motif échoue après j comparaisons :

$$y_i = x_0 \quad \wedge \quad y_{i+1} = x_1 \quad \wedge \quad \cdots \quad \wedge \quad y_{i+j-1} = x_{j-1} \quad \wedge \quad y_{i+j} \neq x_j$$

Notant v_j la longueur du plus grand suffixe de $x_1x_2\cdots x_{j-1}$ qui est un préfixe de x, on reprend alors les comparaisons en testant l'égalité $y_{i+j} = x_{v_j}$. On obtient ainsi l'algorithme suivant :

```
def facteur(x,y):
    (m,n)=(len(x),len(y))
    (i,j)=(0,0)
    v=precalcul(x)
    while i+j<n and j<m:
        if y[i+j]==x[j]: j=j+1
        elif j==0: i=i+1
        else: (i,j)=(i+j-v[j],v[j])
    return j==m
```

Cette même approche permet de calculer efficacement les v_j pour $j \in \{1,\ldots,m\}$ comme ceci :

```
def precalcul(x):
    m=len(x)
    (i,j)=(1,0)
    v=(m+1)*[0]
    while i+j<m:
        if x[i+j]==x[j]: j=j+1; v[i+j]=j
        elif j==0: i=i+1; v[i+j]=j
        else: (i,j)=(i+j-v[j],v[j])
    return v
```

Pour borner la complexité on remarquera que la quantité $2i + j$ croit strictement.

Algorithme de Boyer–Moore [22]. L'efficacité de cette approche est due à son usage de tests d'appartenance à un ensemble de lettres et pas seulement de tests d'égalité entre lettres. Elle a par ailleurs pour originalité de comparer d'abord les dernières lettres du motif. Plus formellement, notons y_i la lettre du texte en cours de considération.

— Si $y_i = x_m$ on poursuit la comparaison des lettres précédentes.
— Si $y_i \notin \{x_1, \dots, x_m\}$ on poursuit en considérant y_{i+m} soit m caractères plus loin.
— Si $y_i \in \{x_1, \dots, x_m\}$ on détermine les décalages possibles du motif.

Le calcul de ce décalage repose sur le pré calcul de deux vecteurs (d_σ) et (e_i) indicés respectivement par les lettres $\sigma \in \Sigma$ et les entiers $i \in \{1, \dots, m\}$ et définis par

$$d_\sigma = \min\left\{k \in \{0, \dots, m\} : k = m \vee x_{m-k} = \sigma\right\},$$
$$e_i = \min\left\{k \in \{1, \dots, i\ \} : k = i\ \ \vee \forall \ell \in \{i+1, \dots, m\}, x_{\ell-k} = x_\ell\right\} + m - i.$$

L'algorithme s'implante alors comme il suit :

```
def facteur(x,y):
    (m,n)=(len(x),len(y))
    (d,e)=precalcul(x)
    j=m-1
    while j<n:
        i=m-1
        while i>=0 && x[i]=y[j]: (i,j)=(i-1,j-1)
        if j==0: return True
        else: j=j+max(d[y[k]],e[i])
    return False
```

Exercice. *Implanter la méthode* `precalcul(x)` *utilisée ci-dessus. Quelle en est la complexité ?*

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 $L^{k+1} = L^k \cdot L$.*

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. *Soient les langages $L = \{u \in \Sigma^\star : |u|\ pair\}$ et $M = \{u \in \Sigma^\star : |u|\ impair\}$. Leur somme vaut $L + M = \Sigma^\star$ et leurs produits $LL = L$, $LM = ML = M$ et $MM = L \setminus \{\varepsilon\}$. Pour $k \geqslant 1$, 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. Ainsi leurs étoiles sont $L^\star = L$ et $M^\star = \Sigma^\star$.*

Proposition. *Pour tous langages L, M et N on a les identités :*

$$L + (M + N) = (L + M) + N \qquad L + \varnothing = \varnothing + L = L \qquad L + M = M + L$$
$$L \cdot (M \cdot N) = (L \cdot M) \cdot N \qquad L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$$
$$L \cdot (M + N) = L \cdot M + L \cdot N \qquad (L^\star)^\star = L^\star$$

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 $\mathscr{L}(E) = L$.*

Par exemple, les expressions rationnelles $\Sigma \cdot \Sigma^\star$ et $\Sigma^\star \cdot \Sigma$ sont distinctes mais décrivent le même langage rationnel, celui des mots de longueur non nulle. Par ailleurs, l'expression non rationnelle $\Sigma^\star \setminus \{\varepsilon\}$ décrit elle aussi ce langage.

Exemple. *Les langages ci-dessous sont rationnels :*
- $\{\varepsilon\} = \varnothing^\star$
- $\Sigma = \{\sigma_1\} + \cdots + \{\sigma_{|\Sigma|}\}$
- *tout ensemble fini de mots*
- *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 :* $\Sigma^\star x \Sigma^\star$
- *les mots formés de blocs de lettres de longueurs paires :* $\left(\bigcup_{\sigma\in\Sigma} \sigma\sigma\right)^\star$

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 ; c'est donc a fortiori le cas de l'ensemble des langages rationnels puisque ce premier se surjecte sur ce second. L'ensemble de tous les langages, à savoir $\mathfrak{P}\left(\Sigma^{(\mathbb{N})}\right)$, est quant à lui indénombrable et donc nécessairement distinct. □

En Caml, on peut transcrire les définitions ci-dessus dans le système de typage en représentant un langage par sa fonction indicatrice. Restons néanmoins conscient que si cette transcription est directe elle est hautement inefficace.

```
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 expressionRationnelle =
    | Vide
    | Lettre of char
    | Somme of expressionRationnelle*expressionRationnelle
    | Produit of expressionRationnelle*expressionRationnelle
    | Etoile of expressionRationnelle
;;

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)
;;
```

Les expressions rationnelles admettent parfois des simplifications évidentes.

Exercice. *En procédant par induction sur l'ensemble des expressions rationnelles, montrer que tout langage rationnel est décrit par une expression rationelle admettant l'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 $\varnothing$.

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. □

Il est ainsi opportun de généraliser la notion d'expression rationnelle afin de permettre l'usage de ces opérations. Si toutefois cela est sans conséquence sur les langages décrits, la distinction est fort pertinente s'agissant de l'efficacité de telles expressions : comme nous le verrons plus tard, le complémentaire d'une expression rationnelle de longueur n s'exprime, en toute généralité, comme une expression rationnelle de longueur doublement exponentielle, c'est-à-dire $\exp(\exp(O(n)))$.

Définition. *On appelle expression rationnelle étendue toute combinaison finie syntaxiquement correcte d'opérations rationnelles, d'intersections, de différences, et des langages $\varnothing$ et $\{\sigma\}$ pour $\sigma \in \Sigma$.*

Lorsqu'on fait référence à la notion effective de motif en programmation plutôt qu'au concept théorique sous-jacent, on parle souvent d'expression régulière. Nous allons présenter cette notion telle que spécifiée par le standard POSIX [28, §2.8] comme *Extended Regular Expressions (ERE)* qui est notamment utilisé par le langage Python et plus précisément sa bibliothèque `re`. On restera cependant conscient du fait que d'autres implantations de ce même concept théorique adoptent pour certaines des syntaxes subtilement différentes.

Remarque. *En anglais, on parle de* regular expression *pour désigner tant le concept théorique que la notion effective ; très souvent, la notion effective est abrégée* regexp *voire* regex.

On adopte dorénavant et jusqu'à la fin de cette section l'alphabet $\Sigma = \{0,1\}^8$ dont les lettres, dans ce contexte aussi appellées caractères, sont les octets interprétés par le codage ASCII que décrit la figure 1.7.

Définition. *Les caractères dits spéciaux sont `#\|{}*+?.()[]^$`. Les expressions régulières sont listées ci-dessous, où `a` et `b` dénotent deux caractères, m et n deux entiers et `e` et `f` deux expressions régulières reconnaissant respectivement les langages E et F.*

- *L'expression `a` (avec `a` non spécial) reconnaît $\{a\}$.*
- *L'expression `\a` (avec `a` spécial) reconnaît $\{a\}$.*
- *L'expression `e|f` reconnaît la somme $E + F$.*
- *L'expression `ef` reconnaît le produit $E \cdot F$.*
- *L'expression `e{m,n}` reconnaît les puissances $\bigcup_{m \leqslant k \leqslant n} E^k$.*
- *L'expression `e*` reconnaît l'étoile $E^\star$.*
- *L'expression `e+` reconnaît $E \cdot E^\star$.*
- *L'expression `e?` reconnaît $E + \{\varepsilon\}$.*
- *L'expression `.` reconnaît Σ.*
- *L'expression `(e)` reconnaît E et groupe cette sous expression.*
- *L'expression `[ab]` reconnaît $\{a, b\}$.*
- *L'expression `[^ab]` reconnaît $\Sigma \setminus \{a, b\}$.*
- *L'expression `^` reconnaît le début d'une ligne.*
- *L'expression `$` reconnaît la fin d'une ligne.*

La bibliothèque `re` de Python offre trois fonctions dénommées `fullmatch`, `match` et `search` permettant respectivement de déterminer si une chaîne de caractère, l'un de ses préfixes ou l'un de ses facteurs est reconnu par une expression régulière donnée. On a par exemple :

```
>>> import re
>>> re.fullmatch("b","abc")
>>> re.match("b","abc")
>>> re.search("b","abc")
<re.Match object; span=(1, 2), match='b'>
```

Ces fonctions renvoient un objet de type `re.Match` (d'évaluation booléenne `True`) lorsqu'une chaîne est reconnue et l'objet `None` (d'évaluation booléenne `False`) sinon. Les objets de type `re.Match` jouissent des méthodes suivantes :

— `group()` renvoie la chaîne reconnue ;
— `start()` renvoie l'indice de son début ;
— `end()` renvoie l'indice de sa fin.

On aura donc :

```
>>> m=re.search("b.d","abcdef")
>>> if m:
...     print(m.start(),m.end(),m.group())
...
1 4 bcd
```

Pour des raisons qui deviendront prochainement claires, lorsqu'une même expression régulière sera utilisée de manière répétée, il sera plus efficace d'écrire :

```
p=re.compile("b.d")
m=p.search("abcdef")
m=p.search("uvwxyz")
```

Remarque. *Le langage Python interprète lui aussi le caractère* `\` *spécialement afin de spécifier certains octets comme le saut de ligne noté* `\n` *(codé* `0a` *en ASCII). Ce comportement est malvenu pour l'écriture d'expressions régulières et on le désactivera donc en utilisant des chaînes de caractères brutes telle* `r"expression régulière"`*, le préfixe* `r` *étant l'abréviation de* raw *en anglais.*

Pour davantage d'informations le lecteur est invité à se référer à la documentation officielle du langage Python [27] et tout particulièrement à ses chapitres *Regular Expression HOWTO* et `re` *— Regular expression operations*.

Exercice. *Écrire des expressions régulières reconnaissant :*

— *les chaînes de caractères de longueur impair ;*
— *les chaînes de caractères ne contenant aucune voyelle ;*
— *les chaînes de caractères contenant au plus une voyelle ;*
— *les chaînes de caractères contenant un nombre impair de voyelles ;*
— *les chaînes de caractères admettant à la fois* `ab` *et* `ba` *comme facteurs ;*
— *les chaînes de caractères n'admettant ni* `ab` *ni* `ba` *comme facteurs ;*
— *les adresses email vous semblant bien formatées ;*
— *les phrases conjuguées à la première personne ;*
— *les entiers naturels inférieurs à* 1729.

Revenons à présent sur la théorie des langages rationnels avec pour objectif de développer les techniques permettant de les mettre en œuvre en pratique de manière efficace. Nous allons pour ce faire commencer en définissant une classe de langages particulièrement adaptée à la reconnaissance effective.

Définition. *Un langage L sur l'alphabet Σ est dit local s'il est de la forme*

$$L \setminus \{\varepsilon\} = (P\Sigma^\star \cap \Sigma^\star S) \setminus (\Sigma^\star F \Sigma^\star)$$

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.

Pour tester l'appartenance d'un mot à un tel langage, la méthode évidente est d'efficacité optimale : un mot de longueur n admet $n-1$ facteurs de longueur deux et, avec des structures de données adaptées, tester leur appartenance à F nécessite $O(n \log |\Sigma|)$ opérations.

Exercice. *Quelles structures de données sont adaptées pour stocker P, S et F ? Écrire en Python puis en Caml une fonction prenant comme argument P, S, F et $m \in \Sigma^\star$ et déterminant si m appartient au langage local défini par ces ensembles.*

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. Posons $\mathscr{L}(\Sigma, P, S, F) = (P\Sigma^\star \cap \Sigma^\star S) \setminus (\Sigma^\star F \Sigma^\star)$. On vérifie facilement :

$$\mathscr{L}(\Sigma, P, S, F) \cap \mathscr{L}(\Sigma, P', S', F') = \mathscr{L}(\Sigma, P \cap P', S \cap S', F \cup F')$$
$$\mathscr{L}(\Sigma, P, S, F)^\star = \mathscr{L}(\Sigma, P, S, F \setminus SP)$$

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 $\mathscr{L}$ 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 `expressionRationnelle` 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))
;;
```

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 Caml, 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.

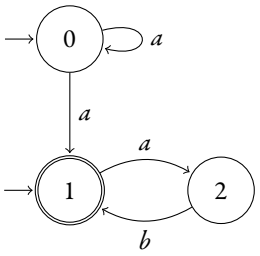


FIGURE 4.1 – Automate reconnaissant le langage $a^\star(ab)^\star$.

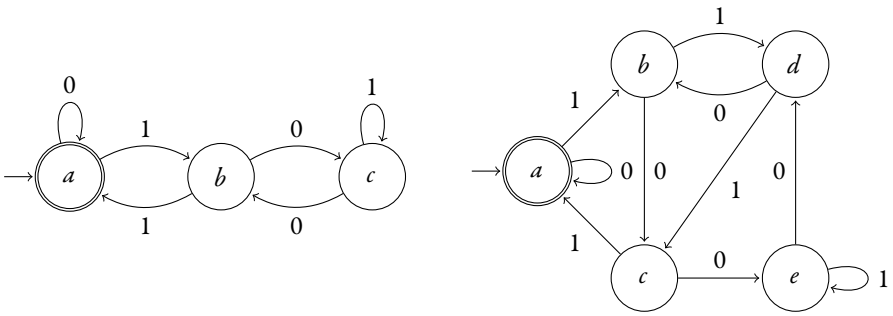


FIGURE 4.2 – Automates reconnaissant les multiples de trois et de cinq écrits en binaire.

```
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 accepte a m =
    let rec appartient x =
        function
        | [] -> false
        | h::t -> x=h || appartient x t
    in
    let rec suivants e c =
        function
        | []         -> []
        | (x,y,z)::t -> let s = suivants e c t in
                        if e=x && c=y then z::s else s
    in
    let n = String.length m in
```

```
    let rec calcul =
        function
        | _,[]   -> false
        | k,e::t -> calcul (k,t) || if k=n then appartient e a.finaux
                    else calcul (k+1,suivants e m.[k] a.transitions)
    in
    calcul (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 une tache simple : on rajoute un état supplémentaire appelé *puits* et des transitions $(e, \sigma, \text{puits})$ pour tout couple (e, σ) vérifiant $\nexists f \in E, (e, \sigma, f) \in T$; intuitivement, le puits capte toutes les transitions non existantes et complète donc l'automate. Le puits n'étant pas co-accessible, l'automate obtenu n'est pas émondé.

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.

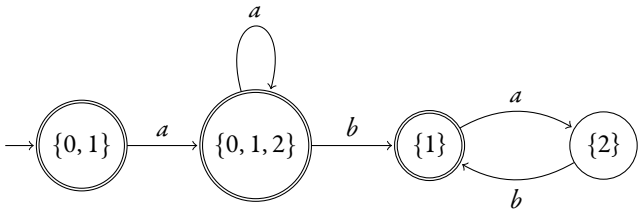


FIGURE 4.3 – Automate des parties émondé de l'automate de la figure 4.1.

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. □

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 et reconnaît $\mathscr{L}(\mathfrak{P}(\mathscr{A})) = \mathscr{L}(\mathscr{A})$.*

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 :*

$$\begin{aligned} I &= \{0,1\} & & \\ aI &= \{0,1,2\} & abaI &= \{2\} \\ bI &= \varnothing & bbaI &= \varnothing \\ aaI &= \{0,1,2\} & aabaI &= \varnothing \\ baI &= \{1\} & babaI &= \{1\} \end{aligned}$$

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.

Exercice. *Écrire une fonction de type « `automate -> string -> bool` » mettant en œuvre cette méthode ; on pourra repartir de l'implantation naïve `accepte()` vue plus haut.*

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.

Théorème (Kleene [3])**.** *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 [8], qui reprend la structure de l'algorithme de Floyd–Warshall vu à la section 3.4.4 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 $E = \{1, \dots, n\}$ 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 dont un calcul dans $\mathscr{A}$ commence en e, termine en f et passe exclusivement par des états de $\{1, \dots, k\}$. 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 $\{1, \dots, k-1\}$, 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 construire un automate reconnaissant le langage d'une expression rationnelle donnée en composant les opérations suivantes.

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
étoile $\mathscr{A}^\star$	$\mathscr{L}(\mathscr{A})^\star$	$\lvert\mathscr{A}\rvert$	pas en général
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

FIGURE 4.4 – Tableau récapitulatif des opérations élémentaires sur les automates.

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 $\mathscr{B}$.

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 $L\cdot L^\star$ où l'on a posé

$$T' = T\cup\big\{(f,\sigma,e) : f\in F, \sigma\in\Sigma, e\in E, \exists i\in I, (i,\sigma,e)\in T\big\}.$$

Les transitions rajoutées permettent de recommencer en un état initial un calcul acceptant. On obtient enfin un automate pour l'étoile en écrivant $L^\star = \{\varepsilon\}+L\cdot L^\star$.

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 $(\Sigma, E_{\mathscr{A}}\cup E_{\mathscr{B}}, T', I', F')$ reconnaît le langage $\mathscr{L}(\mathscr{A})\cdot\mathscr{L}(\mathscr{B})$ où l'on a posé

$$T' = T_{\mathscr{A}}\cup T_{\mathscr{B}}\cup\big\{(f,\sigma,e) : f\in F_{\mathscr{A}}, \sigma\in\Sigma, e\in E_{\mathscr{B}}, \exists i\in I_{\mathscr{B}}, (i,\sigma,e)\in T_{\mathscr{B}}\big\},$$
$$I' = I_{\mathscr{A}}, \qquad F' = \begin{cases} F_{\mathscr{B}} & \text{si } \varepsilon\notin\mathscr{L}(\mathscr{B}) \\ F_{\mathscr{B}}\cup F_{\mathscr{A}} & \text{si } \varepsilon\in\mathscr{L}(\mathscr{B}) \end{cases}.$$

La figure 4.4 récapitule les propriétés de ces constructions. En les combinant on obtient la preuve ci-dessous, souvent appelée *algorithme de Thompson* [17].

Démonstration du sens direct du théorème de Kleene. Afin de construire un automate reconnaissant le langage d'une expression rationnelle donnée, procédons par induction sur l'ensemble des expressions rationnelles muni du bon ordre

$$\begin{aligned} L\leqslant M \iff \exists L',\ & M = L+L' \\ & \vee\ M = L\cdot L' \\ & \vee\ M = L'\cdot L \\ & \vee\ M = L^\star. \end{aligned}$$

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 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()`. Cette méthode est implantée, directement ou par le biais de bibliothèques telles `pcre` et `re2`, dans la majorité des logiciels pertinents, notamment les langages de programmation, les éditeurs de texte et les utilitaires comme `grep` et `awk`.

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 $(\Sigma,E,T,I,E\smallsetminus F)$ reconnaît le langage $\Sigma^\star\smallsetminus L$ qui est donc lui aussi rationnel d'après le sens réciproque du théorème de Kleene. □

Entreprenons pour finir 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 $(P\Sigma^\star\cap\Sigma^\star S)\smallsetminus(\Sigma^\star F\Sigma^\star)$ 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* [10], 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.

Rajoutons en complément du programme les éléments suivants.

Lemme (dit *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 $(e_0, e_1, \ldots, e_n)$. Supposant $n \geqslant N$, par le principe des tiroirs, il existe deux indices $i < j$ dans $\{0, \ldots, N\}$ 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.*

Afin de mieux décrire la structure des langages rationnels et des automates qui les reconnaissent, fixons un alphabet Σ et définissons, pour tout langage L, une relation d'équivalence sur les mots de $\Sigma^\star$. (On précise que la même approche pourrait être menée sur les états d'un automate donné.)

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 [4, 5]). *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 reconnaissant L.*

Démonstration. Si L est rationnel alors soit (Σ, E, T, I, F) un automate déterministe 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 $\overline{e} = \overline{f}$. Une autre approche, due à Antimirov [29], 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 :*

$$\partial_\sigma \varnothing = \{\varnothing\} \qquad \partial_\sigma(E+F) = \partial_\sigma E \cup \partial_\sigma F$$

$$\partial_\sigma\{\tau\} = \begin{cases} \{\varnothing^\star\} & si\ \sigma = \tau \\ \{\varnothing\} & si\ \sigma \neq \tau \end{cases} \qquad \partial_\sigma(E\cdot F) = \begin{cases} \partial_\sigma E \cdot \{F\} & si\ \varepsilon \notin \mathscr{L}(E) \\ \partial_\sigma E \cdot \{F\} \cup \partial_\sigma F & si\ \varepsilon \in \mathscr{L}(E) \end{cases}$$

$$\partial_\sigma(E^\star) = \partial_\sigma E \cdot \{E^\star\}$$

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 Caml 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

Introduction

1. *On souhaite simuler un système physique atome par atome. Supposant que représenter un atome prenne un octet de mémoire, quel masse d'atomes peut on simuler sur un ordinateur de bureau ? On rappelle qu'une masse de 12 grammes de carbone 12 comprend environ* $6 \cdot 10^{23}$ *atomes.*

2. *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
```

3. *Comment effectueriez vous des calculs avec des nombres rationnels ? Présenter les avantages et inconvénients de chaque approche envisagée.*

Algorithmique et programmation I (Python)

4. *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* $r \to \infty$*.*

5. *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, en moyenne, le nombre de telles décompositions pour les entiers* $n \in \{1, \dots, 1234\}$ *?*

6. *Pour tout nombre premier* p *on considère la quantité*

$$a_p = \left| \left\{ (x, y) \in \{0, \dots, p-1\}^2 : y^2 = x^3 + x + 1 \bmod p \right\} \right|.$$

On a par exemple $a_3 = 3$ *car seuls les trois couples* $(0,1)$*,* $(0,2)$ *et* $(1,0)$ *satisfont l'équation pour* $p = 3$*. Que vaut* a_{13} *? Quelle est la moyenne de la suite* $\frac{a_p}{p}$ *pour* $p \in \mathscr{P} \cap \{1, \dots, 10^3\}$ *? Quel est l'écart type de la suite* $\frac{a_p - p}{\sqrt{p}}$ *pour* $p \in \mathscr{P} \cap \{1, \dots, 10^3\}$ *?*

7 (suites de Syracuse)**.** *Soit* s *une suite à valeurs entières vérifiant, pour tout* $k \in \mathbb{N}$*, la relation*

$$s_{k+1} = \begin{cases} s_k/2 & \text{si } s_k \bmod 2 = 0, \\ 3s_k + 1 & \text{si } s_k \bmod 2 = 1. \end{cases}$$

Si $s_0 = 6$*, que vaut* s_{11} *? Vérifier que, quel que soit* $s_0 \in \left\{1, 2, \dots, 10^4\right\}$*, il existe* k *tel que* $s_k = 1$*. Quelle est la moyenne du plus petit tel entier* k *pour* $s_0 \in \left\{1, 2, \dots, 10^4\right\}$ *? Et son écart type ?*

8 (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*.*

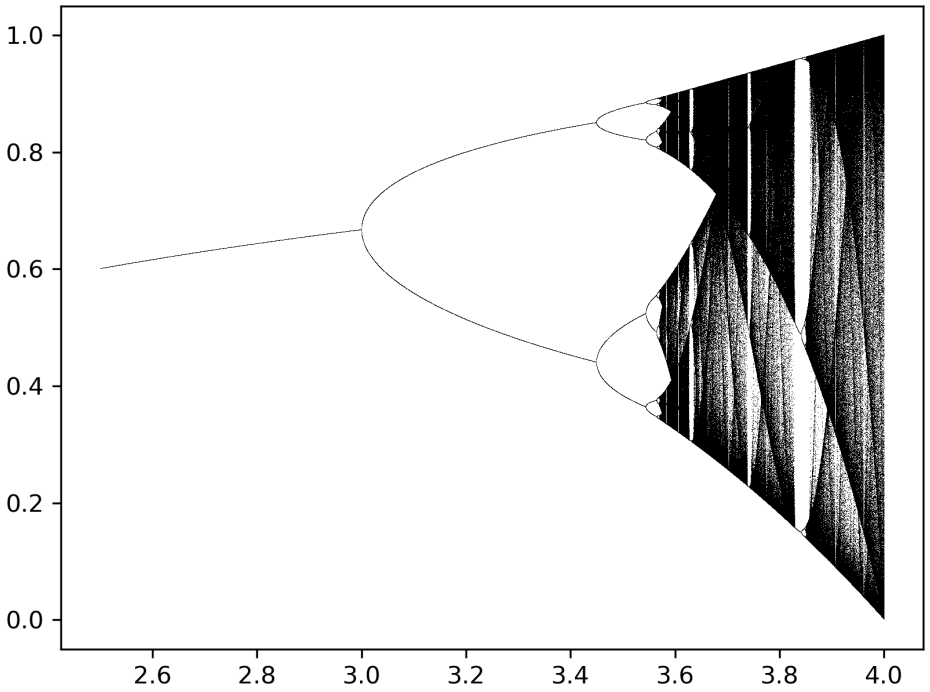


FIGURE A.1 – Valeurs d'adhérence de la suite logistique pour $\lambda \in [5/2, 4]$.

Ingénierie numérique et simulation (Numpy/Scipy)

9 (régression linéaire). *Soient $X = (x_1, \ldots, x_n)$ et $Y = (y_1, \ldots, y_n)$ deux familles de réels. La droite $y = ax + b$ pour laquelle 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 les coefficients de cette droite. Choisir dix points arbitraires puis les tracer ainsi que cette droite.

10 (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.*

A.2 Second semestre

Méthodes de programmation

11. *Écrire une fonction renvoyant les facteurs premiers d'un un entier donné.*

12. *Écrire un programme affichant les n premières lignes du triangle de Pascal. On pourra utiliser les fonctions* `print_int`, `print_string` *et* `print_newline`.

13 (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.

14. *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.*

15 (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.*

16 (calcul formel). *On définit en Caml 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.*

17. *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 Caml ainsi que sa réciproque. Implanter alors une fonction de type «* `int list -> int` *» qui, pour tout $k \in \mathbb{N}^\star$, réalise une bijection de $\mathbb{N}^k$ vers $\mathbb{N}$.*

18. *Programmer le jeu de Nim. Voir* `https://en.wikipedia.org/wiki/Nim`*. On pourra utiliser les fonctions* `read_int` *et* `Random.int`*.*

Structures de données et algorithmes I

19 (tables de hachage). *FIXME d'abord uniquement pour déterminer l'appartenance à un ensemble ensuite pour des fonctions*

20. *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 type « `int -> ()` » affichant à l'écran les $n!$ permutations de $\{1,\dots,n\}$. Montrer sa correction. Borner sa complexité.

21 (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 Caml 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.*

22. *Écrire un programme qui, étant donnés deux nœuds x et y d'un arbre binaire donné $\mathcal{A}$, détermine l'ancêtre commun à x et y de profondeur maximale. Prouver la correction et déterminer la complexité de l'algorithme utilisé.*

23. *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 $b_n=\frac{1}{n+1}C_{2n}^n$.*

A.3 Troisième semestre

Algorithmique et programmation II (Python)

24. *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 ?

25 (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 ?

26. *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 donnée et donner pour chacune la complexité de ces quatre opérations.*

27 (jeu de la vie [19]). *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.

Notions de logique

28 (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.*

29 (connecteurs universels). *Montrer que le connecteur logique non-et noté $\overline{\wedge}$ et défini par l'équivalence $p \overline{\wedge} q \iff \neg(p \wedge q)$ 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.

Structures de données et algorithmes II

30. *On modélise par un arbre de décision un algorithme triant une liste de n éléments distincts $(x_1, \ldots, x_n)$. 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 priorise 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 ?

31. *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.*

32 (arbre AVL [13])**.** *On se propose de concevoir un type d'arbre de recherche auto-ré-équilibrant.*

Définir un type Caml 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.

33 (tri par tas [15])**.** *Écrire une fonction triant un tableau d'entiers de la manière suivante : on insère d'abord ses n éléments comme étiquettes dans un tas vide puis on extrait les racines successives de ce tas. En utilisant la représentation tabulaire du tas, implanter cette méthode* en place*, c'est-à-dire en n'utilisant que le tableau donné et un nombre fini de variables, à l'exclusion de tout autre structure de données.*

34. *La boulangerie de l'exercice 24 fait de nouveau le buzz 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

35. *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.*

36 (coloriage). *On appelle coloriage d'un graphe $G = (S, A)$ avec k couleurs toute application $f : S \to \{1, \dots, k\}$ vérifiant $f(x) \neq f(y)$ pour tout couple de sommets adjacents $(x, y) \in A$. On appelle nombre chromatique $\chi(G)$ le plus petit nombre de couleurs nécessaires au coloriage de G. Donner un minorant et un majorant de $\chi(G)$ en fonction de $|S|$ et $|A|$. Écrire un programme calculant χ. Démontrer sa correction et borner sa complexité.*

37 (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 de taille n à isomorphisme près. Écrire un programme déterminant si deux graphes donnés sont isomorphes. Démontrer sa correction et borner sa complexité.*

38. *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 $\geqslant n/2$?*

Initiation aux bases de données (SQL)

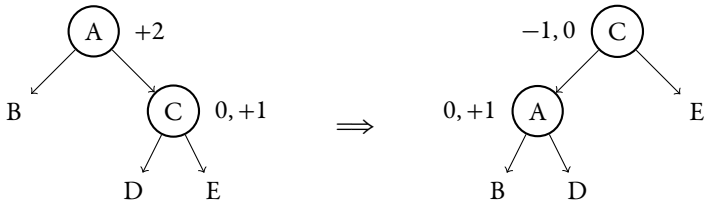


FIGURE A.2 – Ré-équilibrage d'un nœud par rotation simple.

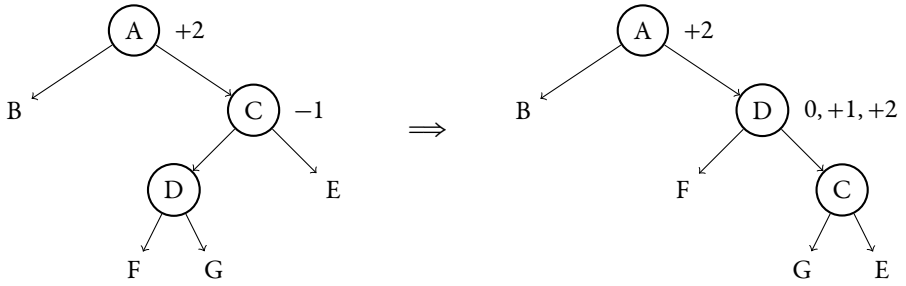


FIGURE A.3 – Ré-équilibrage d'un nœud par rotation double. Ramène au cas de la figure A.2.

A.4 Quatrième semestre

Motifs, automates et expressions

39 (puzzle MU [24]). *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.*

40. *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.*

41 (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 $c_n = \frac{1}{n+1} C_{2n}^n$.

42 (arbre des suffixes). *FIXME*

43 (plus court surmot commun). *Voir [31, §1.4].*

44. *Étant donnés deux langages arbitraires L et M, déterminer toutes les relations d'inclusion entre les langages ci-dessous.*

$$L+M \quad L\cdot M \quad (L+M)^\star \quad (L\cdot M)^\star \quad L^\star + M^\star \quad L^\star \cdot M^\star \quad (L^\star + M^\star)^\star \quad (L^\star \cdot M^\star)^\star$$

45 (lemme d'Arden [9]). *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.*

46. *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.*

47. *Montrer que si L est un langage rationnel, alors $\sqrt{L} = \{m \in \Sigma^\star : m^2 \in L\}$ aussi.*

48. *Montrer que les langages rationnels sont exactement les langages reconnus par les autoamtes dont les transitions sont étiquetées par des expressions rationelles plutôt que par des lettres.*

49. *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] 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.

[3] Stephen Cole Kleene. *Representation of events in nerve nets and finite automata*. Research Memorandum RM-704. Project RAND, U.S. Air Force, 1951.

[4] 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.

[5] Anil Nerode. « Linear Automaton Transformations ». *Proceedings of the AMS* 9.4 (1958), pages 541-544. doi : 10.2307/2033204.

[6] Edsger Wybe Dijkstra. « A note on two problems in connexion with graphs ». *Numerische Mathematik* 1.1 (1959), pages 269-271. doi : 10.1007/BF01386390.

[7] Bernard Roy. « Transitivité et connexité ». *Comptes Rendus de l'Académie des Sciences de Paris* 249 (1959), pages 216-218.

[8] 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.

[9] 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.

[10] Виктор Михайлович Глушков. « Абстрактная теория автоматов ». *Успехи Математических Наук* 16.5 (1961), pages 3-62.

[11] Robert Willoughby Floyd. « Algorithm 97 : Shortest Path ». *Communications of the ACM* 5.6 (1962), page 345. doi : 10.1145/367766.368168.

[12] Stephen Warshall. « A theorem on Boolean matrices ». *Journal of the ACM* 9.1 (1962), pages 11-12. doi : 10.1145/321105.321107.

[13] Георгий Максимович Адельсон-Вельский et Евгений Михайлович Ландис.
« Один алгоритм организации информации ».
Доклады Академии Наук 146.2 (1962), pages 263-266.

[14] Анатолий Алексеевич Карацуба.
« Умножение многозначных чисел на автоматах ».
Доклады Академии Наук 145.2 (1962), pages 293-294.

[15] John William Joseph Williams. « Algorithm 232 : Heapsort ».
Communications of the ACM 7.6 (1964), pages 347-348.
DOI : 10.1145/512274.512284.

[16] Robert C. Daley et Peter Gabriel Neumann.
« A general-purpose file system for secondary storage ».
Fall Joint Computer Conference. Édité par Robert W. Rector. Tome 1.
American Federation of Information Processing Societies. Association for Computing Machinery, 1965, pages 213-229. DOI : 10.1145/1463891.1463915.

[17] Kenneth Lane Thompson. « Regular expression search algorithm ».
Communications of the ACM 11.6 (1968), pages 419-422.
DOI : 10.1145/363347.363387.

[18] 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.

[19] Martin Gardner. « Mathematical Games — The fantastic combinations of John Conway's new solitaire game 'life' ». *Scientific American* 223 (1970), pages 120-123.

[20] Robert Henry Risch. « The solution of the problem of integration in finite terms ».
Bulletin of the American Mathematical Society 76.3 (1970), pages 605-608.
DOI : 10.1090/S0002-9904-1970-12454-5.

[21] Arnold Schönhage et Volker Strassen.
« Schnelle Multiplikation großer Zahlen ». *Computing* 7.3-4 (1971), pages 281-292.
DOI : 10.1007/BF02242355.

[22] 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.

[23] 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.

[24] Douglas Richard Hofstadter. *Gödel, Escher, Bach. An Eternal Golden Braid*.
Basic Books, 1979.

[25] IEEE. *Binary Floating-Point Arithmetic*. IEEE Standard. 1985. ISBN : 0738111651.
DOI : 10.1109/IEEESTD.1985.82928.

[26] Xavier Leroy.
Caml Light. A lightweight, portable implementation of the core Caml language.
INRIA. 1989. URL : https://caml.inria.fr/caml-light/.

[27] Guido van Rossum et al. *Python Documentation*. Python Software Foundation. 1990.
URL : https://docs.python.org/.

[28] IEEE. *Portable Operating System Interfaces (POSIX). Part 2 : Shell and Utilities*.
IEEE Standard. 1993. ISBN : 073811376X. DOI : 10.1109/IEEESTD.1993.6880751.

[29] 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.

[30] 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/.

[31] Gaetan Bisson. *Pépites algorithmiques. Algorithmique des graphes*.
Semester 1, graduate engineering. École des Mines de Nancy, 2010.
url : https://gaati.org/bisson/tea/pepites-graph.pdf.

[32] Richard Peirce Brent et Paul Zimmermann. *Modern Computer Arithmetic*.
Monographs on Applied and Computational Mathematics.
Cambridge University Press, 2010. isbn : 0521194695.

[33] *Classes préparatoires scientifiques. Programme d'informatique*. Arrêté du 4 avril 2013.
NOR : ESRS1306084A. Ministère de l'enseignement supérieur, 4 avr. 2013.
url : http:
//www.education.gouv.fr/pid285/bulletin_officiel.html?cid_bo=71586.

[34] *Classes préparatoires scientifiques.*
Programme de l'option informatique en classe de MPSI et MP. Arrêté du 4 avril 2013.
NOR : ESRS1306085A. Ministère de l'enseignement supérieur, 4 avr. 2013.
url : http:
//www.education.gouv.fr/pid285/bulletin_officiel.html?cid_bo=71588.

[35] *Enseignement de l'option informatique en classes préparatoires scientifiques*.
Note de service numéro 2017-182 du 27 novembre 2017. NOR : ESRS1732186N.
Ministère de l'enseignement supérieur, 27 nov. 2017. url : https:
//www.education.gouv.fr/pid285/bulletin_officiel.html?cid_bo=123904.

[36] SQLite Tutorial. *Chinook SQLite sample database*. 2017.
url : http://www.sqlitetutorial.net/sqlite-sample-database/.

[37] David Harvey et Joris Van Der Hoeven. *Integer multiplication in time O(n log n)*.
2019. url : https://hal.archives-ouvertes.fr/hal-02070778.