Informatique

Gaetan Bisson

https://gaati.org/bisson/

Introduction

Ce cours a pour objectif d'inculquer les notions élémentaires d'informatique en s'appuyant sur la pratique (manipulation d'outils) pour aborder la théorie (aspects scientifiques). À ce titre, il unifie deux matières du programme officiel des classes MPSI/MP : l'informatique générale (chapitre 1 et sections 3.5 et 3.1) et l'option informatique (autres sections). Conformément, on utilisera les langages de programmation Python, Numpy/Scipy et SQL en informatique générale et Caml Light en option informatique.

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 usuelles 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 « `for` » imbriquées, chacune ayant pour effet de faire prendre successivement les valeurs $1, 2, \ldots, 19$ $(= 20 - 1)$ 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.*

Essayez 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* xy *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

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 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.

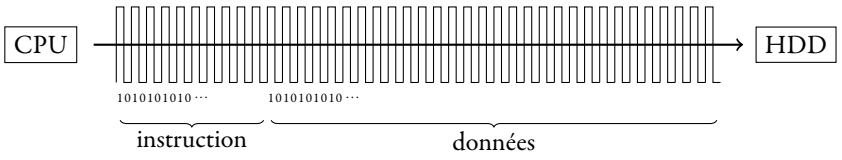


FIGURE 1.1 – Processeur transmettant une instruction d'écriture de données au disque dur.

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.

Par exemple, les processeurs disposent de trois grandes familles d'instructions, chacune opérant sur des « mots » de 64 bits :

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

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.

Un programme n'est qu'une suite d'instructions destinées au processeur. En première approximation, on pourra supposer que toutes s'exécutent en temps constant (de l'ordre de la nanoseconde). Plus tard nous apprendrons ainsi à estimer le temps d'exécution d'un programme en comptant le nombre d'opérations élémentaires qu'il réalise.

Ordres de grandeurs. Si les bits sont une quantité indivisible d'information numérique, on les trouve rarement seuls : ils viennent généralement par paquets de huit, appelés octets. Un octet peut donc prendre $2^8 = 256$ valeurs possibles. Pour un ordinateur de bureau, on gardera en tête les ordres de grandeurs suivants :

— mémoire vive : 8 Go
— mémoire dure : 1 To
— débit réseau : 1 Mo/s
— processeur : 1 Gop./s

Voir la figure 1.2 pour un rappel des préfixes du système international d'unités (SI) pertinents.

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 de l'entier* $n \in \mathbb{N}$ *l'unique famille* $(c_0, c_1, \ldots, c_{k-1}) \in$

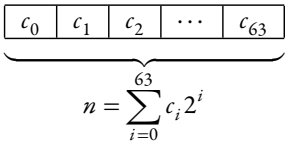


FIGURE 1.3 – Entiers de type `uint64`.

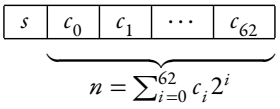


FIGURE 1.4 – Entiers de type `int64`.

$\{0,1\}^k$ *vérifiant* $c_{k-1} = 1$ *et*

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

Toutefois, si la mémoire peut être supposée infinie, les processeurs ne peuvent opérer que sur des mots de 64 bits. Les entiers naturels dits « machines » se limitent donc au cas $k \leqslant 64$ soit $n \in \{0, \ldots, 2^{64} - 1\}$. Voir 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 ; nous dirons plus tard que cela réalise l'arithmétique de $\mathbb{Z}/2^{64}\mathbb{Z}$. Le langage Caml utilise directement les entiers du processeur et on aura donc :

```
# let sq n = n*n in sq (sq (sq (sq (sq 2)))) ;;
- : int = 0
```

Pour représenter les entiers relatifs, on réserve le premier bit au stockage du signe et consacre les soixante trois autres à la décomposition binaire de la valeur absolue. Cela permet de coder les entiers de l'ensemble $\left\{-\left(2^{63} - 1\right), 2^{63} - 1\right\}$. Voir la figure 1.4.

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

Cette dernière possibilité exploite une technique connue sous le nom d'arithmétique multiprécision ; elle est évidemment plus coûteuse en temps processeur. C'est le choix fait par le langage de programmation Python. Le langage Caml adhère quant à lui au premier comportement et la majorité des calculatrices au second.

1.1.3 Nombres réels

Le codage numérique des nombres réels le plus courant est régit par le standard IEEE 754 qui stipule qu'un mot machine (soit 64 bits) se décompose en un bit de signe, onze bits d'exposant et 52 bits de mantisse (avec un bit dominant « 1 » implicite). Voir la figure 1.5. 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).

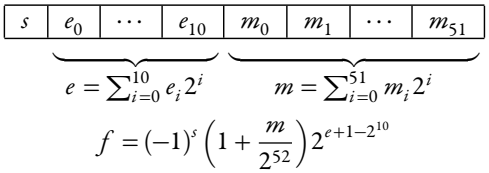


FIGURE 1.5 – Réels de type `binary64`.

Plus précisément, les nombres représentés entre 2^{63} et 2^{64} sont exactement les entiers ; entre 2^{62} et 2^{63} il s'agit des demi entiers ; entre 2^{61} et 2^{62} il s'agit des quarts d'entiers, etc. Inversement, autour de zéro, toute la précision porte sur les bits après la virgule.

Les nombres flottants souffrent :

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

En langage Caml, qui utilise directement les flottants du processeur, on a :

```
# 2.**1100. ;;
- : float = inf.0
# 0.3 -. 0.2 -. 0.1 ;;
- : float = -2.77555756156e-17
# 0.5**1100. >. 0. ;;
- : bool = false
```

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 dont la charge utile était estimée à plusieurs centaines de millions d'euros.*

On peut encore répondre à ces problèmes par des techniques d'arithmétique multiprécision et d'arrondi correct au prix d'un temps de calcul accru.

1.1.4 Systèmes d'exploitation

Avant l'invention des systèmes d'exploitation dans les années soixantes, les ordinateurs ne pouvaient exécuter qu'un programme à la fois et ce dernier devait s'adresser directement au processeur pour toute opération. Si aujourd'hui nous avons dépassé cette situation rigoureuse, c'est grâce à quelques innovations matérielles mais surtout de grandes innovations logicielles.

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.

En d'autres termes, le système 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 droits :** limitation des accès et opérations de certains utilisateurs vis-à-vis de certains processus et fichiers.

Les fichiers forment les feuilles d'une arborescence de répertoires. 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`

Il s'agit ci-dessus de chemins *absolus*, c'est-à-dire partant du répertoire racine ; on peut aussi spécifier des chemins *relatifs*, c'est-à-dire partant du répertoire courant. Sur les systèmes de type Unix, le répertoire courant est souvent « `/home/user/` » où « `user` » désigne votre nom d'utilisateur ; les chemins suivants 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 URI (adresses Internet) à l'ajout près du protocole ; on écrira donc* « `protocol:///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 ; pn 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 comme second argument les modes suivants :

— « `r` » : lecture
— « `w` » : écriture
— « `x` » : création et écriture

Sous Linux, la notion de droit d'accès est particulièrement explicite :

```
$ ls -l /home/bisson
total 104
-rw-r--r--  1 bisson users     0 Dec  9  2013 -i
lrwxrwxrwx  1 bisson users    20 Dec  9  2013 bin -> var/bin
drwx------  2 bisson users  4096 Jul 26 16:19 tmp
drwxr-xr-x 11 bisson users  4096 Jul 20 09:14 var
```

De même pour les processus :

```
$ ps auxw
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
[...]
bisson   22661  5.0  0.2  91288 11052 ?        Rs   17:08   0:00 xterm
bisson   22663  1.0  0.1  23856  5260 pts/5    Ss   17:08   0:00 bash
bisson   22666  0.0  0.0  36008  3140 pts/5    R+   17:08   0:00 ps auxw
```

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. Python 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"`

Nous allons bien sûr présenter chaque type, ses principales propriétés et opérations, mais ne pourrons en aucun cas être exhaustif. Le lecteur gagnera à recourir librement à la documentation officielle du langage et en l'occurrence à cette section ci :

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

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]  # []
```

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. Récapitulons :

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

Apprenons à présent quelques opérations usuelles purement par l'exemple. 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 trois instructions que nous allons maintenant présenter : « `if` », « `for` » et « `while` ». Un *branchement conditionnel* exécute un bloc d'instructions lorsqu'une expression booléenne donnée est vraie. La 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
```

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 », 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. En Python on écrira 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 aussi d'avancer plus rapidement dans votre maitrise de la programmation et, à terme, dans le développement de vos programmes.

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 toutefois 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 du fait 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.

Chaînes de caractères. En première approximation, un caractère n'est autre qu'un octet interprété comme l'indique la figure 1.6. (C'était historiquement le cas avant que le support des caractères accentués et non latins complique la situation.) Une chaîne de caractères est donc typiquement stockée en mémoire comme la suite des caractères qui la composent, suivie de l'octet `NUL` pour indiquer la fin de la chaîne. Voir la figure 1.7.

Cela permet de manipuler les chaînes de caractères comme on s'y attendrait :

```
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"
```

Nous allons maintenant présenter les tableaux et les listes, deux notions fondamentales très complémentaires. On ne les retrouve toutefois pas nécessairement dans les langages modernes comme Python, ceux-ci préférant généralement offrir en leur place des structures de données plus complexes mais réunissant les avantages de chacune.

Tableaux. Un tableau est une suite finie d'éléments d'un même type stockés en mémoire à des emplacements contigus. Lorsque le type est simple, cette construction est très similaire à celle des chaînes de caractères. Voir les figures 1.8 et 1.9. Remarquer que rien n'indique la longueur d'un tableau : celle-ci est typiquement supposée connue du programmeur.

Cela permet d'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 suite finie d'éléments stockés en mémoire à des emplacements arbitraires, chaque élément indiquant l'emplacement de son successeur. Voir la figure 1.10 dont la construction précise porte le terme technique de « liste chaînée ».

Afin d'accéder au k^{e} élément on doit donc parcourir les $k-1$ premiers, mais cela permet de rajouter en d'enlever facilement des éléments à un emplacement arbitraire dans la liste.

En Python le type « `list` » est sensiblement plus complexe que présenté ci-dessus, mais présente les avantages des tableaux et des listes : on peut directement accéder, rajouter ou enlever un élément d'ordre arbitraire.

```
t=[1,2,"abc"]
t[0]              # 1
t[3]              # IndexError
t[1]=0
t                 # [1, 0, 'abc']
t.insert(1,2)
t                 # [1, 2, 0, 'abc']
t.append('d')
```

	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
a	LF	SUB	*	:	J	Z	j	z
b	VT	ESC	+	;	K	[	k	{
c	FF	FS	,	<	L	\	l	|
d	CR	GS		=	M	]	m	}
e	SO	RS	.	>	N	^	n	~
f	SI	US	/	?	O	_	o	DEL

FIGURE 1.6 – Table résumant le codage de caractères ASCII. L'ordonnée et l'abscisse donnent respectivement le premier et le second coefficient de l'indice du caractère en base hexadécimale. Les caractères d'indice compris entre 127 et 255 sont réservés.

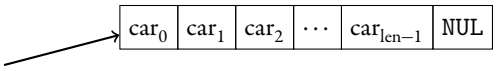


FIGURE 1.7 – Une chaîne de caractères.

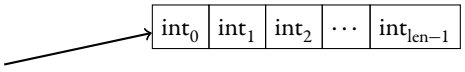


FIGURE 1.8 – Un tableau d'entiers.

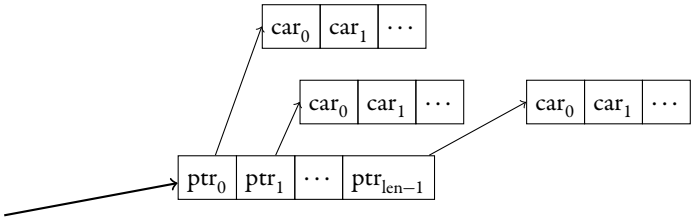


FIGURE 1.9 – Un tableau de chaînes de caractères.

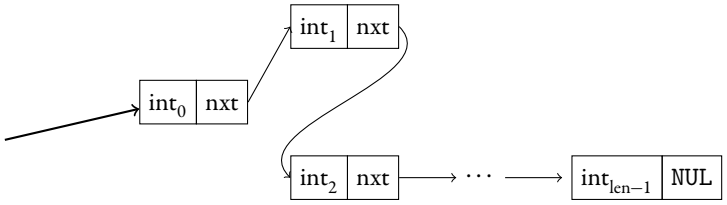


FIGURE 1.10 – Une liste chaînée d'entiers.

```
t               # [1, 2, 0, 'abc', 'd']
t.pop(1)        # 1
t               # [1, 0, 'abc', 'd']
```

Les listes sont très nettement un *first-class citizen* de Python. On dispose notamment du constructeur suivant qui permet de les manipuler avec une aisance fort appréciable.

```
[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 les ensembles, structure de données réalisant la notion mathématique éponyme. En Python :

```
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}
```

La notion de dictionnaire étend celle d'ensemble en associant à chaque élément un objet arbitraire. Elle correspond donc à la notion de fonction sur un ensemble fini. En Python :

```
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,chaîne)` » *déterminant si le mot est un sous mot de la chaîne de caractères. Écrire une fonction* « `indice(mot,chaîne)` » *renvoyant un indice auquel on trouve le mot dans la chaîne de caractères s'il s'y trouve et* « `-1` » *sinon. Écrire une fonction* « `indices(mot,chaîne)` » *renvoyant la liste des indices auxquels on trouve le mot dans la chaîne de caractères.*

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

1.3 Ingénierie numérique et simulation (Numpy/Scipy)

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 La bibliothèque Numpy/Scipy

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.

Dans le cadre de ce cours nous travaillerons essentiellement avec les bibliothèques NumPy et Scipy. La bibliothèque NumPy permet de manipuler des tableaux multidimensionnels dans le langage Python. Scipy s'appuie sur NumPy pour construire un environnement de calcul scientifique complet, offrant notamment des fonctionnalités en algèbre linéaire, statistiques, traitement du signal, etc. Cette section n'a vocation qu'à survoler ces deux bibliothèques ; nous reviendrons ensuite sur leurs usages au fur et à mesure lorsque cela sera pertinent.

En Python, on charge une bibliothèque par l'instruction « `import bibliothèque` » ; ses fonctions sont alors accessibles sous la syntaxe « `bibliothèque.fonction()` ». Alternativement, on pourra aussi utiliser « `from bibliothèque import *` » afin d'accéder à toutes les fonctions de cette bibliothèque directement comme « `fonction()` ». Écrivons donc :

```
>>> from numpy import *
```

Le type « `array` » représente des tableaux sous leur forme primitive décrite plus haut. Le constructeur « `array()` » permet de convertir en tableau n'importe quelle liste Python dont tous les éléments sont du même type. On dispose aussi de constructeurs spécifiques :

```
>>> zeros((2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> diag([1,1])
array([[1, 0],
```

```
       [0, 1]])
>>> fromfunction(lambda a,b:a+b, (2,3))
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  3.]])
```

Ce type supporte toutes les opérations matricielles usuelles :

```
>>> M=fromfunction(lambda a,b:a+b, (3,3))
>>> M+M
>>> 3*M
>>> M*M      # produit coefficient par coefficient
>>> dot(M,M) # produit matriciel
>>> linalg.det(M)
>>> linalg.eig(M)
```

La bibliothèque NumPy possède des sous bibliothèques telles « `random` » :

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

On peut enfin charger la bibliothèque SciPy.

```
>>> from scipy import *
>>> integrate.quad(f,a,b)  # intégrale de f sur [a,b]
>>> optimize.newton(f,a)   # recherche d'un zéro de f en partant de a
```

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

`https://docs.scipy.org/doc/scipy/reference/tutorial/index.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

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

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 trapèzes, aux termes en $f(\alpha)$ et $f(\alpha+n\varepsilon)$ près. Ces termes à eux seuls améliorent donc la convergence de linéaire en quadratique...*

Si cela peut paraître astucieux, la méthode des trapèzes forme le premier échelon d'une famille de méthodes bien générales 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.

Exercice. *Calculer le volume $V_n(r)$ d'une boule de rayon r en dimension n.*
Il vérifie les relations $V_n(r)=r^nV_n(1)$ et $V_{n+1}(r)=\int_{-r}^{r}V_n(\sqrt{1-x^2})dx$.
Quel est le volume des sphères correspondantes ?

Exercice. *Mesurer la superficie de la partie du plan (x,y) définie par $x^2+y^4<1$.*

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):
    fa=f(a)
    fb=f(b)
    while b-a>epsilon:
        c=(a+b)/2
        fc=f(c)
```

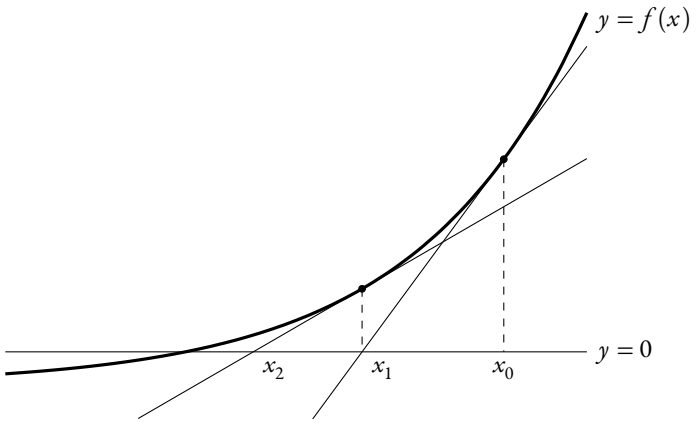


FIGURE 1.11 – Deux itérations de la méthode de Newton.

```
        if fa*fc<0:
            b=c
            fb=fc
        else:
            a=c
            fa=fc
    return a
```

Chaque itération divise par deux la longueur de l'intervalle contenant la racine cherchée et augmente ainsi la précision d'un bit significatif. Il faut donc $\log\left(\frac{b-a}{\varepsilon}\right)$ itérations afin d'obtenir le résultat désiré.

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 1.11.

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. □

Afin de chercher un zéro de f (de dérivée g) au voisinage de a en n itérations on procède alors 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, lorsque a est suffisamment proche du zéro recherché α, la précision en nombre de bits significatifs double à chaque itération. Il faut donc $c + \log\left(\log\left(\frac{1}{\varepsilon}\right)\right)$ itérations pour obtenir le résultat désiré, où la constante c ne dépend que de f et de α.

Remarque. *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.*

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.*

Lorsque la dérivée g de f n'est pas efficacement évaluable on peut 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.

Exercice. *Pour $s \in]1, \infty[$ on pose $\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$. Calculer une approximation de l'unique préimage de 2 par ζ. Comparer les vitesses de convergence de la dichotomie et de Newton.*

1.3.4 Problème dynamique à une dimension

Le principe consistant à approcher une fonction par ses 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 $n \in \left\{1, \ldots, \left\lfloor \frac{\eta}{\varepsilon} \right\rfloor\right\}$.

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 1.12.*

Cette méthode s'applique aussi aux équations différentielles ordinaires d'ordre supérieur en les écrivant comme des équations différentielles ordinaires du premier ordre vectorielles.

Exercice. *Calculer ainsi une approximation du sinus.*

Exercice (méthode de Runge–Kutta)**.** *Refaire les exercices précédents en exploitant l'approximation d'ordre supérieur :*

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

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.

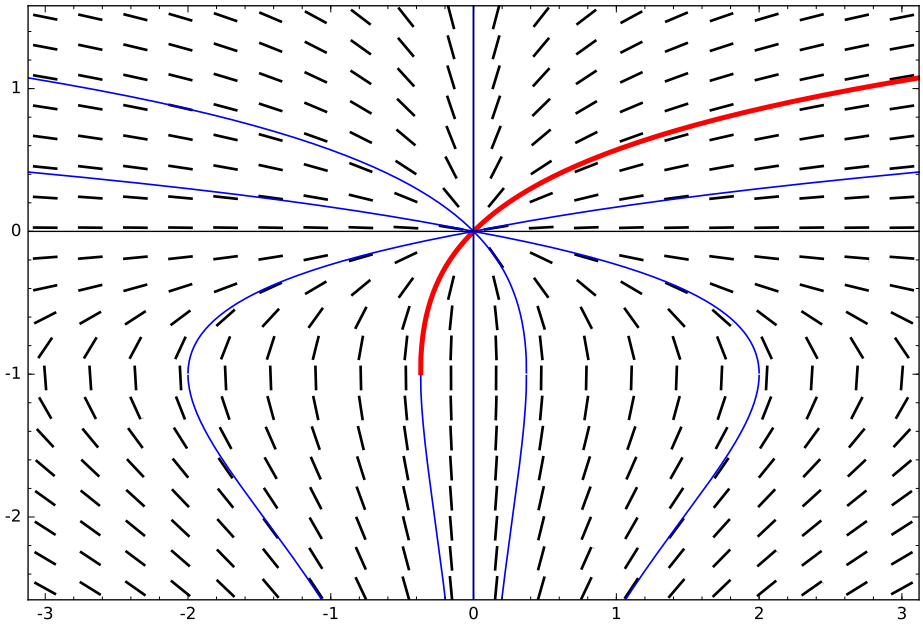


FIGURE 1.12 – Champ de vecteurs et solutions de l'équation différentielle $y' = \frac{y}{x(1+y)}$. En rouge, la fonction W de Lambert.

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

Au premier semestre, 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 tous les processeurs fonctionnent, aussi le travail de traduction du code en instructions processeurs était relativement modeste, à l'exception notable des types avancés comme les listes.

Certains langages dits « de plus haut niveau » permettent, par un travail de traduction plus complexe, de programmer autrement. Ce semestre, en langage Caml, nous adopterons certains idiomes de **programmation fonctionnelle**. Notre code consistera alors en des définitions de fonctions (au sens mathématique), leur composition et évaluation ; en particulier, nous ne modifierons pas la valeur des variables, nous nous contenterons de leur appliquer des fonctions.

La version de ce langage au programme des classes préparatoires [5] est Caml Light [3] ; cette implantation obsolète depuis quinze ans présente l'unique avantage de disposer d'un exécutable Windows tout-en-un :

`https://caml.inria.fr/pub/distrib/caml-light-0.74/cl74win.exe`

Nous présenterons d'abord sa syntaxe puis aborderons un à un ses aspects fonctionnels.

Remarque. *Au concours vous mettrez ce langage en œuvre principalement sur papier et éventuellement au tableau, 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

Toute expression se termine par « ; ; ».

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

Caml résume le résultat de toute expression sous la forme « `nom : type = valeur` » ; on prendra l'habitude de consulter attentivement cette information pour vérifier que Caml a bien compris ce qu'on voulait.

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

```
# let x = 2 ;;
x : int = 2
# x ;;
- : int = 2
# let y = 1 in y+1 ;;
- : int = 2
# y ;;
Toplevel input:
> y ;;
> ^
The value identifier y is unbound.
```

Première différence majeure avec Python : en Caml, les variables ne sont pas mutables ; en l'absence du mot clef « `let` » l'opérateur « `=` » est interprété comme un test d'égalité :

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

Remarque. *Caml possède aussi un opérateur noté « `==` » mais dont la signification est subtilement différente : il teste l'égalité des adresses mémoire. On veillera donc à* ***ne jamais l'utiliser !***

Caml respecte très fortement le typage :

```
# 1. + 2. ;;
Toplevel input:
> 1. + 2. ;;
> ^^
This expression has type float,
but is used with type int.
# 1.+.2. ;;
- : float = 3.0
```

Heureusement, les opérateurs conditionnels sont eux surchargés :

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

Le type « `int` » est celui du processeur :

```
# let x = 2 ;;
x : int = 2
# let x = x*x ;;
x : int = 4
# let x = x*x ;;
x : int = 16
```

```
# let x = x*x ;;
x : int = 256
# let x = x*x ;;
x : int = 65536
# let x = x*x ;;
x : int = 4294967296
# let x = x*x ;;
x : int = 0
```

Cependant Caml apporte quelques extensions aux types numériques :

```
# 1 / 0 ;;
Uncaught exception: Division_by_zero
# let x = 1. /. 0. ;;
x : float = inf.0
# x -. x ;;
- : float = -nan.0
```

Les branchements conditionnels ont la structure « `if-then-else` » classique à ceci près qu'ils n'admettent pas de terminateur « `end if` » (donné par l'indentation en Python) contrairement aux boucles qui utilisent « `do-done` » comme nous le verrons plus bas. On utilisera donc les mots clefs « `begin-end` » pour délimiter tout bloc d'instructions :

```
# 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; print_int 5 ;;
Toplevel input:
>if 1=1 then print_int 2; print_int 3 else print_int 4; print_int 5 ;;
>                                       ^^^^
Syntax error.
```

2.1.2 Fonctions

Les fonctions se définissent par l'une des deux manières équivalentes que voici :

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

Caml parse « `function` » et « `->` » donc l'usage de parenthèses n'est pas nécessaire ; elles servent uniquement à regrouper des expressions. On a alors :

```
# f 1 ;;
- : int = 3
# f (1) ;;
- : int = 3
# f (f 1) ;;
- : int = 7
```

Voici par exemple une fonction permettant de tester la parité d'un entier :

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

En retour, Caml indique toujours le type (aussi appelé signature) de la fonction définie, ici « `int -> bool` » ; cette information est cruciale !

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

Lorsqu'une fonction n'est pas restreinte à un type donné on aura :

```
# let id x = x ;;
id : 'a -> 'a = <fun>
```

Les fonctions définies récursivement doivent être explicitement marquées comme telles :

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

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

La signature est une information particulièrement pertinente dans le cas de fonctions en plusieurs variables ; considérons l'exemple ci-dessous.

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

Remarquer que Caml interprète cette fonction comme allant de $\mathbb{R}$ vers $\mathbb{R}^{\mathbb{R}}$ alors qu'on aurait plutôt tendance à la considérer comme allant de $\mathbb{R} \times \mathbb{R}$ vers $\mathbb{R}$. Ces deux points de vue sont parfaitement équivalents puisque les ensembles $\left(E^F\right)^G$ et $E^{F\times G}$ sont en bijection canonique. C'est l'essence même de la programmation fonctionnelle que d'adopter le premier point de vue. Ainsi, si $f : \mathbb{Z}^2 \to \mathbb{Z}$ est une fonction de deux variables, l'expression « $f\,a$ » dénotera la fonction $b \mapsto f(a, b)$. Ici on a :

```
# let g = argument 1. ;;
g : float -> float = <fun>
# g 0. ;;
- : float = 1.57079632679
```

Concernant la syntaxe, on retiendra que les fonctions consomment leurs arguments. On écrit donc « `f a b` » pour $f(a, b)$ ou encore « `f a (b+c) - d` » pour $f(a, b + c) - d$.

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

La notion de produit cartésien est cependant disponible en Caml. On l'emploiera avec parcimonie, d'une part, parce qu'elle entravera l'utilisation des idiomes de programmation fonctionnelle et, d'autre part, parce qu'elle est moins évidente à maîtriser. Notamment, le produit cartésien n'est pas une opération associative :

```
# let x = 1,2,3 ;;
x : int * int * int = 1, 2, 3
# let a,b,c = x in a ;;
- : int = 1
# let y = 1,(2,3) ;;
y : int * (int * int) = 1, (2, 3)
# x = y ;;
Toplevel input:
> x = y ;;
>     ^
This expression has type int * (int * int),
but is used with type int * int * int.
```

De même pour les fonctions :

```
# let f a b = a*b ;;
f : int -> int -> int = <fun>
# let g (a,b) = a*b ;;
g : int * int -> int = <fun>
# f = g ;;
Toplevel input:
>f = g ;;
>    ^
This expression has type int * int -> int,
but is used with type int -> int -> 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 une structure équivalente aux branchements « `if-then-else` » classiques mais préférable pour des raisons purement esthétiques. Plutôt que d'écrire

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

et surtout de se demander comment formater cette expression agréablement sur trois lignes, on écrira :

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

Cette structure est intégrée avec celle de fonction :

```
let square x = x*x ;;
let square = function | x -> x*x ;;
let rec fact = function
```

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

2.1.5 Listes

Les listes se notent « `[a; b; c]` » ; leurs éléments doivent être de même type. Il s'agit de véritables listes chaînées (voir figure 1.10) et pour les manipuler nous n'avons essentiellement à notre disposition que l'opérateur de concaténation « `::` ».

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

Toute liste non vide est donc de la forme « `head::tail` » ; cette dichotomie est la source du paradigme de manipulation des listes en Caml. Par exemple, pour calculer la longueur d'une liste on écrirait :

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

(Heureusement déjà disponible en la fonction « `list_length` ».)

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.*

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

Fort de fonctions auxiliaires, on peut adapter ce paradigme à tout problème ; par exemple pour couper une liste en deux :

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

Exercice. *Écrire une fonction calculant le plus grand élément d'une liste d'entiers. Faire de même pour la moyenne puis enfin pour la variance.*

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

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.*

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

2.1.6 Références

Même si elles ne sont pas préconisées 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 « `!` ».

```
# let a = ref 0 ;;
a : int ref = ref 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
  let aux = ref 1 in
  for k = 1 to n do
    aux := !a;
    a := !b;
    b := !b + !aux;
  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 faire :

```
let rec fibo01 a b = function
| 0 -> a
| n -> fibo01 b (a+b) (n-1) ;;
let fibo = fibo01 0 1 ;;
```

2.1.7 Vecteurs

Les vecteurs se notent « `[|a; b; c|]` » ; leurs éléments doivent être de même type. Il s'agit de véritables tableaux (voir figure 1.9) 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 « `<-` ».

```
# let t = make_vect 5 1 ;;
t : int vect = [|1; 1; 1; 1; 1|]
# for k=0 to 4 do t.(k) <- k done ;;
- : unit = ()
# t ;;
```

```
- : int vect = [|0; 1; 2; 3; 4|]
# vect_length t ;;
- : int = 5
```

On peut ainsi écrire :

```
let fibo n =
  let t = make_vect (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 = make_vect 5 (f 0) ;;
for k=0 to 4 do t.(k) <- f k done ;;
```

Le vecteur « `t` » est alors de type « `(int -> int) vect` ».

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

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

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

Les chaînes de caractères ne sont autres que des vecteurs de caractères ; elles disposent néanmoins d'un type et d'une syntaxe spécifiques :

```
# let s = "coucou" ;;
s : string = "coucou"
# string_length s ;;
- : int = 6
# s.[0] ;;
- : char = `c`
# s.[0] <- `b` ;;
- : unit = ()
```

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

Pour des vecteurs multidimensionnels attention à ne pas tomber dans le piège suivant :

```
# let x = make_vect 4 (make_vect 4 0) ;;
 [|[|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]|]
# x.(1).(1) <- 1 ;;
 - : unit = ()
# x ;;
 - : int vect vect =
 [|[|0; 1; 0; 0|]; [|0; 1; 0; 0|]; [|0; 1; 0; 0|]; [|0; 1; 0; 0|]|]
```

Le vecteur « `x` » ainsi créé comprend quatre exemplaires du même vecteur « `[|0; 0; 0; 0|]` », ce qui explique le phénomène observé. On aurait voulu avoir quatre vecteurs « `[|0; 0; 0; 0|]` » distincts, ce pour quoi il faut utiliser la commande dédiée « `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 bien plus élégamment et efficacement. On peut définir de nouveaux types de deux façons en Caml : comme produit cartésien de types existants, d'une part, et comme union de types existants, d'autre part.

Cette première construction s'illustre notamment avec les nombres complexes :

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

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

La construction d'un type union est pertinente lorsque les objets peuvent être regroupés en plusieurs catégories distinctes, par exemple dans le cas de types finis :

```
type tirage = Pile | Face ;;
```

On peut aussi paramétrer chaque catégorie, par exemple afin de représenter l'ensemble des droites du plan :

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

Les commentaires ne sont là que pour éclairer le lecteur sur la convention choisie. On manipule alors ce type ainsi :

```
let paralleles = function
| Verticale(_),Verticale(_) -> true
| Oblique(a,_),Oblique(b,_) -> a=b
| _ -> false ;;
```

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 d'un exemple simple.

```
type 'a Liste = Fin | Cons of 'a * 'a Liste ;;
let x = Cons(1,Cons(2,Cons(3,Fin))) ;;
```

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 de boucles « tant que » ou d'appels récursifs. Pour traiter chacun de ces cas, nous utiliserons un principe appelé « induction » qui généralise la méthode de démonstration par récurrence à des ensembles autres que celui des entiers naturels ; à ce titre, cette approche porte aussi le nom de « récurrence transfinie ».

Rappelons en premier lieu ces notions vues au premier semestre.

Définition. *Un ordre sur un ensemble E est une relation binaire réflexive, transitive et symétrique, 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 $(x, y, z) \in E^3$:

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

Un ordre est dit bien fondé lorsque toute partie non vide admet un élément minimal, c'est-à-dire :

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

La notion d'ordre bien fondé étend celle de bon ordre aux ordres partiels. Le cas des ordres totaux est donc parfaitement classique.

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 $\mathbb{N}^\star$ est un ordre bien fondé.*
— *L'inclusion dans $\mathfrak{P}(\mathbb{N})$ n'est pas un ordre bien fondé.*
— *Dans $\mathbb{R}[X]$ l'ordre $P \mathcal{R} Q \iff \deg(P) \leqslant \deg(Q)$ est 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 ce résultat nécessite l'axiome du choix, ce qui dépasse le cadre de ce cours. À l'instar du théorème ci-dessous, on se contentera donc de l'admettre comme généralisation du principe de récurrence.

Théorème (principe d'induction)**.** *Soit $\leqslant$ un ordre bien fondé sur un ensemble E. 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. *Dans le cas d'un ordre total l'élément minimal m est unique (c'est notamment le cas de la récurrence classique sur $\mathbb{N}$ pour lequel $m = 0$) et l'initialisation se borne alors à vérifier $P(m)$. Dans le cas général, en revanche, il s'agira de vérifier $P(m)$ pour chaque élément minimal m.*

En pratique, on montrera la terminaison d'un algorithme par induction portant sur la valeur d'une variable arbitraire (dans le cas de boucles « tant que ») ou d'un paramètre (dans le cas d'appels récursifs). Pour ce faire on montrera indépendamment trois assertions :

1. L'ensemble des valeurs est muni d'un ordre bien fondé.
2. L'algorithme termine lorsque la valeur est minimale.
3. Toute exécution avec valeur y se ramène à un ensemble d'exécutions avec valeurs $x < y$.

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* $n \in \mathbb{N}$*.*

On notera qu'il est d'importance toute particulière de spécifier l'ensemble des valeurs des arguments 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. *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, toute la difficulté consiste à identifier l'ordre bien fondé rendant la preuve de terminaison possible.

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 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 les valeurs en question 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 $\text{pgcd}(a, b)$ *est préservée à chaque appel récursif de la fonction puisqu'on a* $\text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b)$. *L'algorithme termine lorsque* $b = 0$, *cas où l'on a clairement* $\text{pgcd}(a, b) = a$. *Cette fonction renvoie donc bien* $\text{pgcd}(a, b)$.

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 vect_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 on a $m = \max_{i\in\{0,\dots,k\}} v_i = \max_{i\in\{0,\dots,k+1\}} v_i$.*

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

Ceci démontre l'induction par disjonction des cas. En sortie de boucle on a bien $m = \max \mathtt{V}$.

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

2.2.3 Complexité

Les processeurs n'offre 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. *Nous considérons ici des processeurs arbitraires et ignorons sciemment les accès mémoire. Une approche rigoureuse consisterait à introduire des ordinateurs idéalisés appelés « machines de Turing » ; ce modèle diffère sensiblement du matériel existant mais on obtient généralement des complexités comparables quel que soit le modèle de machine 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 $g(x)$ le nombre d'instructions utilisées par l'exécution de $\mathscr{A}(x)$. On appelle borne de complexité de $\mathscr{A}$ toute fonction $f : X \to \mathbb{R}$ vérifiant $g = O(f)$. Une telle borne est dite fine s'il existe une partie infinie $Y \subset X$ pour laquelle on a $f|_Y = O(g|_Y)$.*

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 rec 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 $s_{n+1} = a s_n + b$; son terme général s'écrit alors :

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

Exercice. *Évaluer la complexité des programmes ci-dessous.*

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

let fibo n =
let rec fibo01 a b = function
| 0 -> a
| n -> fibo01 b (a+b) (n-1) in
fibo01 0 1 n ;;
```

```
let fibo 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) ;;
```

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 $\{0, \ldots, 2^{64}-1\}$. 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 vecteurs 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 vecteur $[|x_0; x_1; \ldots; x_{n-1}|]$. 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 vecteur d'entiers machines).*

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-caml-light/`

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

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

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

À la main, on effectue 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.

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 par récurrence aux grands entiers, on obtient la méthode de multiplication de Karatsuba dont la complexité est $O(n^{\log_2(3)})$.

Remarque. *L'algorithme de Schönhage–Strassen exploite la transformée de Fourier dans les corps finis afin de multiplier deux entiers de n bits en temps* $O(n\log(n)\log(\log(n)))$*; c'était asymptotiquement le plus rapide connu jusqu'en 2007 lorsque des améliorations sur le terme* log log *ont été obtenues.*

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

L'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.

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 boite 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 [2].

Proposition de correction.

```
let B = 4294967296 ;;
let grand n = [| n / B; n mod B |] ;;

let compare x y =
  let a = ref (vect_length x - 1) in
```

```
  let b = ref (vect_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;;

let addition x y =
  let a = vect_length x in
  let b = vect_length y in
  let z = make_vect (max a b + 1) 0 in
  for i = 0 to vect_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) >= B then begin
      z.(i+1) <- z.(i+1) + z.(i)/B;
      z.(i) <- z.(i) mod B;
    end
  done;
  z ;;

let multiplication x y =
  let a = vect_length x in
  let b = vect_length y in
  let z = make_vect (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) >= B do
        z.(!k+1) <- z.(!k+1) + z.(!k)/B;
        z.(!k) <- z.(!k) mod B;
        incr k;
      done
    done
  done;
  z ;;

let rec karatsuba x y =
  let a = vect_length x in
  let b = vect_length y in
  let k = a / 2 in
  if k < 3 then multiplication x y else
  let x0 = sub_vect x 0 k in
  let y0 = sub_vect y 0 k in
  let x1 = sub_vect x k a in
  let y1 = sub_vect 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 = make_vect 0 (a+b+1) in
  for i = 0 to a+b do
    if i     < vect_length z0 then z.(i) <- z.(i) + z0.(i);
    if i-k   < vect_length z1 then z.(i) <- z.(i) + z1.(i-k);
    if i-k-k < vect_length z2 then z.(i) <- z.(i) + z1.(i-k-k);
    if z.(i) >= B then begin
      z.(i+1) <- z.(i+1) + z.(i)/B;
      z.(i) <- z.(i) mod B;
    end
  done;
  z ;;

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 ;;
```

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 on a $L_i > L_j$ 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 insertion). *Écrire une fonction insérant un élément dans une liste déjà triée. En déduire une méthode de tri qui procède en insérant le premier élément de la liste donnée dans sa queue récursivement triée.*

Exercice (tri 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.*

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.*

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 = make_vect 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.

Proposition de correction.

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

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

```
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 ;;
let rec tri l =
  let r,s=bulle l in
  if s then r else tri r ;;

let rec prefixe = function
| [], _    -> true
| _,   []   -> false
| g::s,h::t -> if g=h then prefixe(s,t) else false ;;
let rec cherche s = function
| []   -> s=[]
| h::t -> prefixe(s,h::t) or cherche s t ;;
```

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 facilite deux opérations : « ajouter » et « enlever » un objet à la collection. Elles diffèrent par l'ordre dans lequel les éléments sont récupérés :

— **Les piles.** (En anglais, « stack » ou « LIFO » pour « Last In First Out ».)
L'élément ajouté en dernier est le premier à être enlevé. Cette structure de donnée est notamment utilisée par le compilateur pour gérer les appels de fonctions et la récursivité.

— **Les files.** (En anglais, « queue » ou « FIFO » pour « First In First Out ».)
L'élément ajouté en premier est le premier à être enlevé. Cette structure de donnée est notamment utilisée comme mémoire tampon pour les communications entre processus, avec les périphériques et en particulier les réseaux.

Exercice. *Pour chacune, définir un type Caml et implanter les opérations associées.*

Proposition de correction.

```
type 'a Pile = Rien | Cons of 'a * 'a Pile ;;
let empiler p x = Cons(x, p) ;;
let depiler = function
| Rien -> failwith "Pile vide"
| Cons(x,p) -> x,p ;;

type 'a File = { mutable v:'a vect; mutable s:int; mutable n:int } ;;
let vide = { v=[|0|]; s=0; n=0 } ;;
let enfiler f x =
  if f.n >= vect_length f.v then f.v <- concat_vect f.v f.v;
  f.v.((f.s+f.n) mod vect_length f.v) <- x;
  f.n <- f.n + 1 ;;
let defiler f =
  if f.n=0 then failwith "File vide";
  let x=f.v.s in
  f.n <- f.n - 1;
  f.s <- f.s + 1;
  x ;;
```

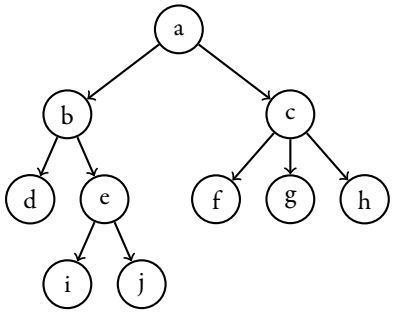


FIGURE 2.1 – Un arbre de taille dix et hauteur trois comportant six feuilles.

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. Précisions 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 ».*

La structure d'arbre généralise celle de liste chaînée (revoir la figure 1.10), 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 ;*
- *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 2.1 : « à l'envers » 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.

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 2.1 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 en exercice que ces méthodes sont toutes deux de complexité optimale.

Définition. *Un arbre est dit binaire lorsque chacun de ses nœuds admet au plus deux fils : le fils gauche, le fils droit, l'un des deux ou aucun.*

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.*

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. *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.*

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

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

Pour démontrer les assertions de ce théorème nous allons procéder par induction sur l'ensemble des arbres binaires muni de l'ordre partiel bien fondé suivant :

Définition. *Un arbre A est dit sous arbre de B lorsque ces deux conditions sont réunies :*
— *La racine de A est un descendant de la racine de B.*
— *Tout descendant dans B de la racine de A est dans A.*

Exercice. *Montrer que l'ordre ainsi défini sur l'ensemble des arbres est bien fondé.*

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 a $t_A = t_B + t_C + 1 \leqslant (2f_B-1)+(2f_C-1)+1 = 2(f_B+f_C)-1 = 2f_A-1$, d'où le résultat. □

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

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 :

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

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) et coefficients (c_{ij}) avec $i \in \{1,\dots,n\}$ et $j \in \{1,\dots,k\}$.*

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.*

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

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 1.10) à laquelle on rajoute ou enlève le premier élément. 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
```

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

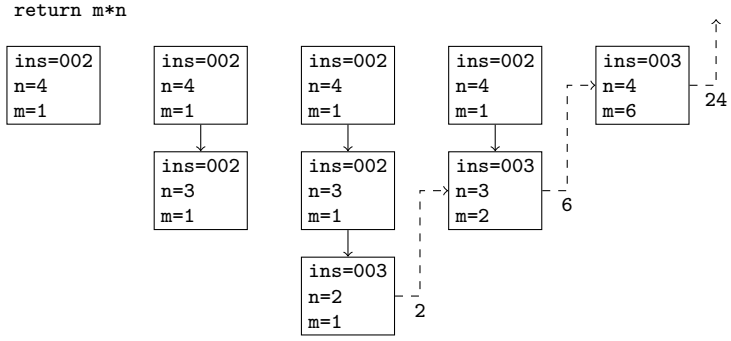


FIGURE 3.1 – Évolution de la pile d'exécution pour `factorielle(4)`.

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. *Soient deux piles contenant chacune initialement* $[1, 2, 3, \ldots, n]$*. Tant qu'aucune n'est vide, on note* x *et* y *leurs derniers éléments et on dépile la première si* $xy^2 + 1$ *admet un nombre pair de facteurs premiers et la seconde sinon. Pour chaque* $n \in \{1, \ldots, 50\}$*, déterminer laquelle des deux piles se vide en premier.*

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.*

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 élément 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. Sous Linux 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=0x7fff6846b...
#2 mutt_encode_path (dest=0x7fff6846be70 "", dlen=256, src=<optimized out>)
#3 0x000000000048d401 in bcache_path (account=<optimized out>, mailbox=<op...
#4 0x000000000048d50c in mutt_bcache_open (account=0x7fff6846c010, mailbox...
#5 0x000000000048971e in pop_open_mailbox (ctx=0x1204d10)
#6 0x0000000000443b92 in mx_open_mailbox (path=path@entry=0x7fff6846c6e0 "...
#7 0x0000000000408de0 in main (argc=1, argv=<optimized out>)
```

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 recrusivement 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 $\mathcal{A}$ 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 $\mathcal{A}(x)$ 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 notion de complexité dans le cas moyen est hors programme mais nous l'indiquons néanmoins dans le tableau ci-dessous par soucis de complétude.

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	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$

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.*

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 basiques d'un logiciel de traitement d'image. On utilisera des bibliothèques Python comme il suit afin de charger, afficher et sauvegarder une image.

```
from matplotlib import pyplot
from scipy import misc

def show(f):
    pyplot.imshow(f)
    pyplot.show()

f = misc.imread('b.png')
show(f)
misc.imsave('c.png', f)
```

La représentation native d'une image digitale est un tableau bidimensionnel de couleurs. La couleur d'indice $[y, x]$ est celle 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).

Pour les couleurs, nous emploierons le codage RGB ; c'est le plus répandu car utilisé nativement par les écrans LCD comme l'illustre la figure 3.3. Il consiste à coder chaque couleur comme un triplet $(r, g, b) \in \{0, \ldots, 255\}^3$ donnant les intensités de rouge, de vert et de bleu qui la composent ; voir la figure 3.4. On a notamment comme couleurs remarquables :

$(0, 0, 0)$	noir	$(255, 0, 0)$	rouge	$(255, 255, 0)$	jaune
$(127, 127, 127)$	gris	$(0, 255, 0)$	vert	$(255, 0, 255)$	magenta
$(255, 255, 255)$	blanc	$(0, 0, 255)$	bleu	$(0, 255, 255)$	cyan

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

```
n,m,_ = f.shape
for x in range(0,m):
    f[99][x][1] = 0
```

```
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
```

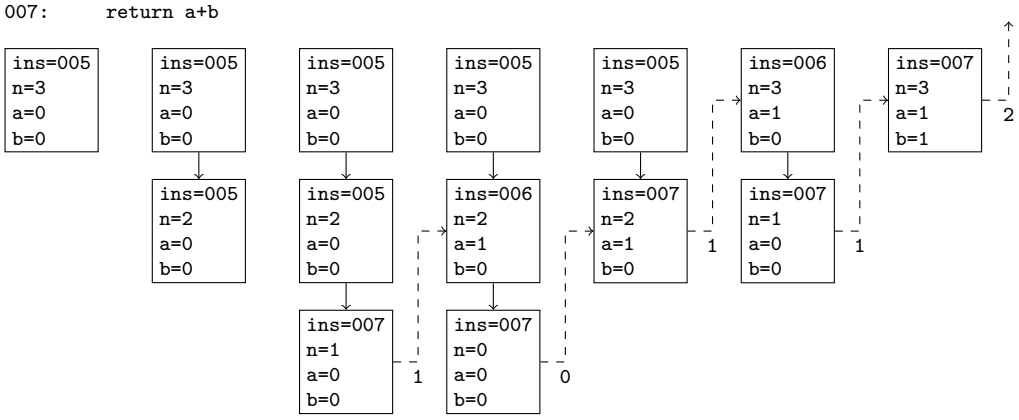


FIGURE 3.2 – Évolution de la pile d'exécution pour `fibonacci(3)`.

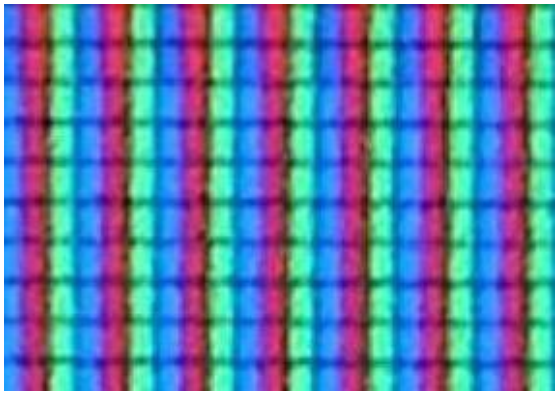


FIGURE 3.3 – Détail d'une matrice de pixels d'un écran LCD.

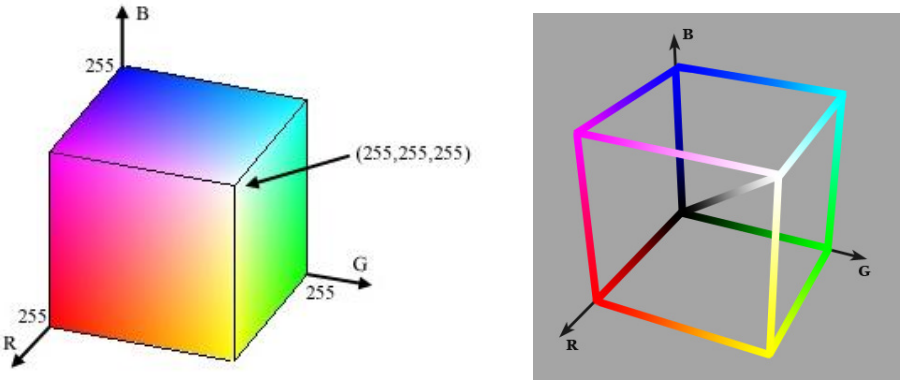


FIGURE 3.4 – Les couleurs du cube RGB ; à droite sur fond gris, ses arêtes uniquement.

Exercice. *Écrire un programme redimensionnant une image suivant la taille donnée. On pourra interpoler linéairement la couleur des pixels, notamment lorsqu'il s'agira d'un agrandissement.*

L'objectif assumé de cette section étant la pratique de la programmation, nous remercions le lecteur d'adopter la définition ci-dessous à fins de simplification, même si elle déforme grossièrement la réalité et notamment le fait que l'œil humain est bien plus sensible aux nuances de vert que de rouge et 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 transformant une image en couleurs en nuances de gris en préservant la luminosité de chaque pixel.*

Exercice. *Écrire un programme ajustant la luminosité d'une image par une constante* γ *donnée ; une couleur* (r, g, b) *sera ajustée en* $(\gamma r, \gamma g, \gamma b)$.

Exercice. *Afin d'améliorer la qualité visuelle d'une image on se propose de la « normaliser » en appliquant à chaque pixel l'opération* $(r, g, b) \mapsto (n(r), n(g), n(b))$ *pour*

$$n(x) = \max\left(255, \left\lfloor 256 \frac{r-\alpha}{\beta-\alpha} \right\rfloor\right)$$

où α *(resp.* β*) dénote la luminosité la plus basse (resp. élevée) des pixels de l'image. Implanter cette transformation.*

Exercice. *Écrire un programme prenant en argument une image et affichant l'histogramme de la fonction qui associe à chaque entier* $\alpha \in \{0, \dots, 255\}$ *le nombre de pixels de l'image ayant* α *comme luminosité.*

En déduire un programme normalisant une image comme ci-dessus mais en prenant pour α *(resp.* β*) la luminosité la plus élevée (resp. basse) parmi les* 1% *des pixels les moins (resp. plus) lumineux de l'image.*

Exercice. *L'utilisateur souhaite effectuer une série de transformations 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 transformation sans pour autant perdre les précédentes. Expliquer comment une pile permet de répondre à ce besoin. Implanter un tel système.*

On se propose enfin de concevoir puis d'implanter une méthode de compression d'images.

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 ne stockant qu'une fois la couleur d'un ensemble de pixels indistinguables. Quelle est le taux de compression typiquement obtenu pour $s = 2$? Et 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 concepts duaux que sont la syntaxe et la sémantique ; cela semble tenir à cœur au programme officiel même si, au delà de la terminologie qu'ils fournissent, ces concepts n'ont essentiellement aucune utilité.

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 l'étude de la signification des écrits d'un langage.*
On appelle syntaxe le respect des règles formelles d'écriture d'un langage.

En informatique, nous y viendrons, les écrits sont les programmes d'un langage de programmation et la sémantique étudie ce qui résulte de leur exécution. En logique, les écrits sont les propositions et la sémantique correspond alors à l'assignation de leurs valeurs de vérité.

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 vect -> bool` » évaluant une telle proposition en des valeurs de vérité données.*

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 $\forall x \in \{V, F\}^n, P(x)$.*

On pourra facilement illustrer chacune des catégories ci-dessus par des exemples.

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

Exercice. *Écrire une fonction s'efforçant de trouver une proposition plus courte et équivalente à une proposition donnée.*

Concluons cette section par un point de culture informatique.

Théorème (Cook–Levin). *Le problème consistant à déterminer si une proposition logique donnée est satisfiable est NP-complet, c'est-à-dire que toute solution peut être vérifiée en temps polynomial et que tout autre problème présentant cette caractéristique s'y réduit en temps polynomial.*

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
;;
```

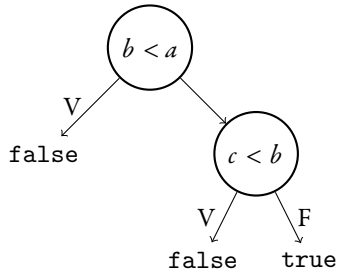


FIGURE 3.5 – Un algorithme et son arbre de décision.

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 [8] ; 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 vecteur 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 est l'état initial et les feuilles étiquetées par les valeurs de retour.*

Voir la figure 3.5 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 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. 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 de la liste (x et y) ; il possède deux fils correspondant chacun à un résultat possible ($x < y$ et

$y < x$). Cet arbre possède $n!$ feuilles, chacune correspondant à une permutation possible de la liste. Sa hauteur, c'est-à-dire la complexité de l'algorithme dans le cas le pire, est donc minorée par $\log(n!)$ 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 élégante 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 de son sous arbre gauche et inférieure à celles 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 ;;
```

Exercice. *Écrire une fonction de type « `int ArbreBE -> bool` » déterminant si un arbre binaire étiqueté est un arbre de recherche.*

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

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

Pour exprimer de manière satisfaisante la hauteur h en fonction de la 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.*

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 vecteur donné que l'on supposera trié.*

```
let rec ArbreRE v =
    let t = vect_length v in
    if t=0 then Nil else
    let e = t / 2 in
    let g = sub_vect v 0 e in
    let d = sub_vect v (e+1) (t-e-1) in
    Noeud(v.(e),ArbreRE g,ArbreRE d) ;;
```

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. On peut concevoir des types d'arbres de recherche auto-ré-équilibrants mais cela nous entraînerai bien au delà des limites du programme.

3.3.3 Arbres tassés

Considérons maintenant une classe d'arbres en un certain sens orthogonale à celle d'arbre de recherche.

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.*

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. Afin de préserver la structure de tas on fera récursivement « remonter » les nœuds d'étiquette maximale, processus appelé « percolation ».*

```
let rec percoler = function
| Nil                                  -> Nil
| Noeud(a,Nil,         Nil)            -> Nil
| Noeud(a,Noeud(b,c,d),Nil)            -> Noeud(b,c,d)
| Noeud(a,Nil,         Noeud(b,c,d))   -> Noeud(b,c,d)
| Noeud(a,Noeud(b,c,d),Noeud(e,f,g)) -> if b<e then
  Noeud(e,Noeud(b,c,d),percoler(Noeud(e,f,g))) else
  Noeud(b,percoler(Noeud(b,c,d)),Noeud(e,f,g)) ;;
```

Extraire dans l'ordre décroissant les étiquettes d'un tas nécessite ainsi $O(ht)$ opérations. Il est donc opportun de minimiser la hauteur d'un tas lorsqu'on le crée ; on utilisera pour cela des arbres « tassés » :

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 3.6 et 3.7.

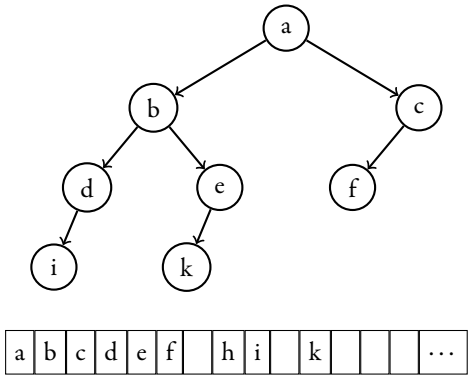


FIGURE 3.6 – Un arbre binaire non tassé et sa représentation tabulaire.

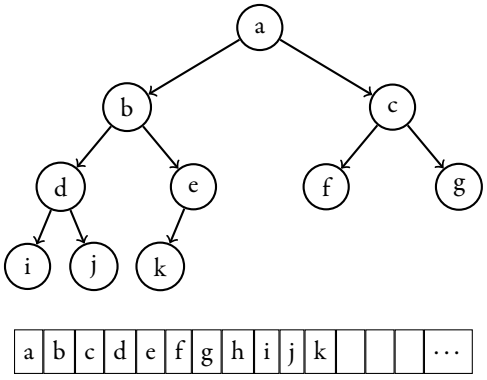


FIGURE 3.7 – Un arbre binaire tassé et sa représentation tabulaire.

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 -> int * 'a vect` *renvoyant la représentation tabulaire d'un arbre binaire tassé ainsi que le plus grand indice du tableau utilisé.*

```
let rec aux v i = function
| Nil          -> ()
| Noeud(a,b,c) ->
    v.(i) <- a;
    aux v (2*i+1) b;
    aux v (2*i+2) c;
in
let tableau a =
    let n = taille a in
    let v = make_vect n (-1) in
    aux v 0 a;
    n,v
;;
```

Exercice. *Écrire une fonction de type* `'a -> int -> 'a vect -> ()` *insérant un nœud d'étiquette donnée dans le tas donné par sa représentation tabulaire et son indice maximal. On prendra soin de préserver la qualité de tas.*

```
let inserer x n v =
    let i = ref 0 in
    let y = ref x in
    while !i < n do
        if !y > v.(!i) then begin
            let z = !y in
            y := v.(!i);
            v.(!i) <- z;
        end;
        i := !i*2+1;
    done;
    v.(n) <- !y;
    i := n;
    while !i>0 && v.(!i)>v.(!i/2) do
        let z = v.(!i) in
        v.(!i) <- v.(!i/2);
        v.(!i/2) <- z;
    done
;;
```

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.

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 3.8. Existait-t-il un chemin passant par chaque pont une fois et une seule ?*

Ce problème, un très grand classique de la théorie des graphes, sort du cadre du programme officiel qui, à l'instar de la définition précédente, se limite à une seule arête par couple de sommets. Nous l'utiliserons néanmoins afin de motiver la formalisation des notions ci-dessous que le lecteur dégourdi n'aura aucun mal à généraliser au « multi-arêtes ».

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 deux notions coïncident et on parle simplement de « degré ».

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

Définition. *Soit (S, A) un graphe. On définit une relation d'équivalence $\mathcal{R}$ sur S en posant $x \mathcal{R} y$ 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.*

Pour un graphe non orienté, $x \mathcal{R} y$ 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 $\mathcal{R}$ sur S en posant $x \mathcal{R} y$ lorsqu'il existe une famille $(s_0, s_1, \ldots, s_n)$ vérifiant*

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

Ses classes partitionnent S et sont appelées composantes connexes faibles.

Dans le cas non orienté ces deux 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.*

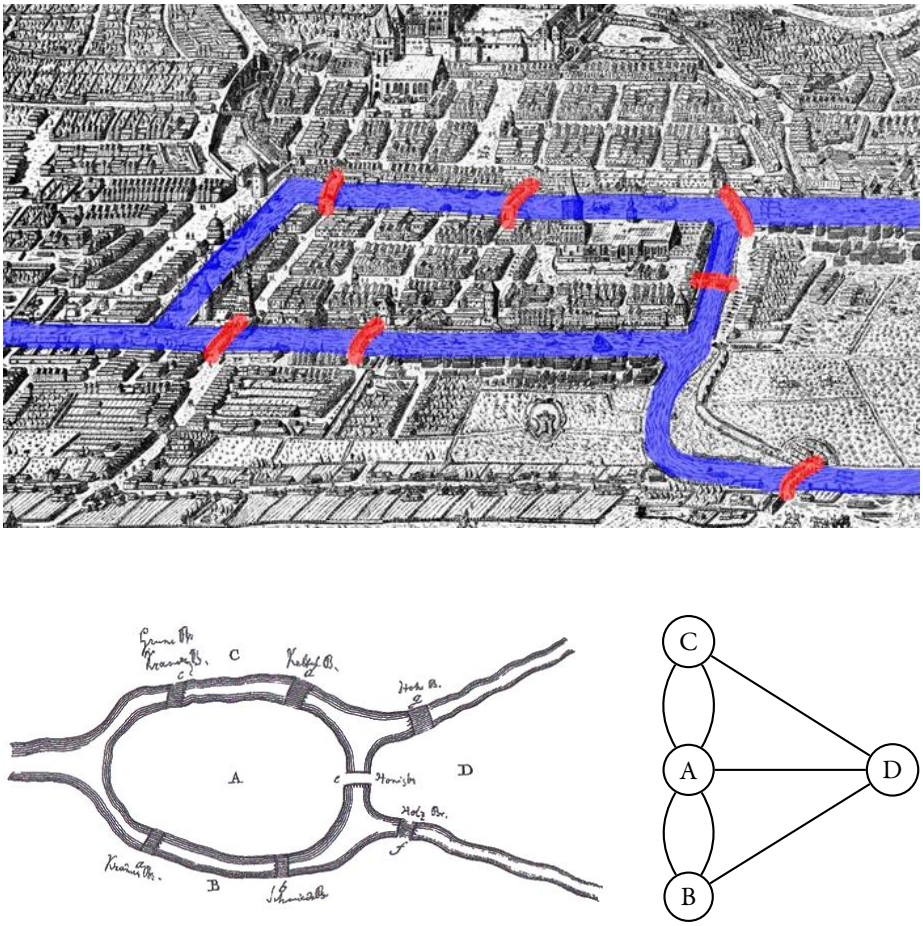


FIGURE 3.8 – Centre ville de Königsberg; carte, représentation et graphe.

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

Définition. *On appelle liste d'adjacence d'un graphe le vecteur 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 l'arête $i \to j$ existe et 0 sinon. Voir la figure 3.9.*

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 porgramme 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 du plus court chemin

Le problème de la recherche du plus court chemin est tout particulièrement pertinent pour les graphes pondérés, c'est-à-dire les graphes dont les arêtes sont étiquetés par des réels positifs.

Définition. *On dit qu'un graphe (S, A) est pondéré lorsqu'il est muni d'une application $p : A \to \mathbb{R}_+$. Dans ce cas, on appelle longueur du chemin $s_0 \to s_1 \to \ldots \to s_n$ la quantité $p(s_0, s_1) + p(s_1, s_2) + \ldots + p(s_{n-1}, s_n)$.*

L'algorithme de Floyd–Warshall. Cette méthode détermine simultanément, pour chaque couple de sommets d'un graphe pondéré donné, la longueur du plus court chemin les reliant. Pour $S = \{1, \ldots, n\}$ notons $m^k_{i,j}$ la longueur du plus court chemin de i à j passant exclusivement par des sommets de $\{1, \ldots, k\}$. L'algorithme consiste à calculer cette quantité itérativement en exploitant l'identité :

$$m^k_{i,j} = \min\left(m^{k-1}_{i,j},\ m^{k-1}_{i,k} + m^{k-1}_{k,j}\right)$$

Algorithme (Floyd–Warshall).

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

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

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. *Modifier cet algorithme afin qu'il renvoit non seulement leurs longueurs mais aussi chaque plus court chemin.*

Algorithme de Dijkstra. Cette méthode détermine le plus court chemin reliant deux sommets a et b donnés. Pour cela, elle construit le plus court chemin entre a et chaque sommet d'un ensemble croissant, incorporant progressivement leurs voisins en commençant par les plus proches.

Algorithme (Dijkstra).
Entrée : *Deux sommets a et b d'un graphe pondéré (S, A, p).*
Sortie : *La longueur du plus court chemin de a à b.*

1. *Créer un vecteur $D = (d_s)_{s\in S}$ indexé par S avec coefficients initialisés à $+\infty$.*
2. *Soit E un ensemble initialisé à S.*
3. *Affecter $d_a \leftarrow 0$.*
4. *Tant que $b \in E$:*
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_b.*

Afin de déterminer efficacement un somment $s \in E$ minimisant d_s à chaque itération, on pourra utiliser une file de priorité implantée à l'aide d'un tas.

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

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

3.5 Initiation aux bases de données (SQL)

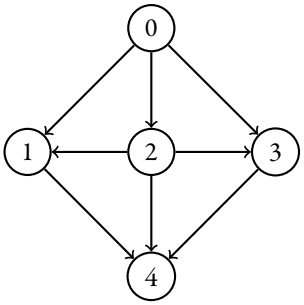
Les bases de données sont des structures de données complexes permettant de stocker de manière organisée de grands volumes d'information. Elles fournissent trois opérations majeures : la création d'une base (en spécifiant l'organisation des données qu'elle sera amenée à stocker), l'insertion de données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

— **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.

Pour comprendre la place qu'occupe une base de données au sein d'une application logicielle, nous pouvons utiliser le modèle appelé « architecture à trois couches » qui 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 de leur stockage et transmission. Comme la figure 3.10. l'illustre, c'est cette dernière couche, la plus distante de l'utilisateur, qui incorpore typiquement une base de données.

Chaque couche communique évidemment 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. Il s'agit du langage SQL (Structured Query Language) inventé en 1970 pour interagir avec les bases de données structurées suivant le « modèle relationnel », un schéma d'organisation des données alors nouvellement inventé [4]. De nos jours, la majorité des bases de données se conforme à ce modèle et supporte ce langage.




```
[| [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 3.9 – Un graphe, sa liste d'adjacence et sa matrice d'adjacence.

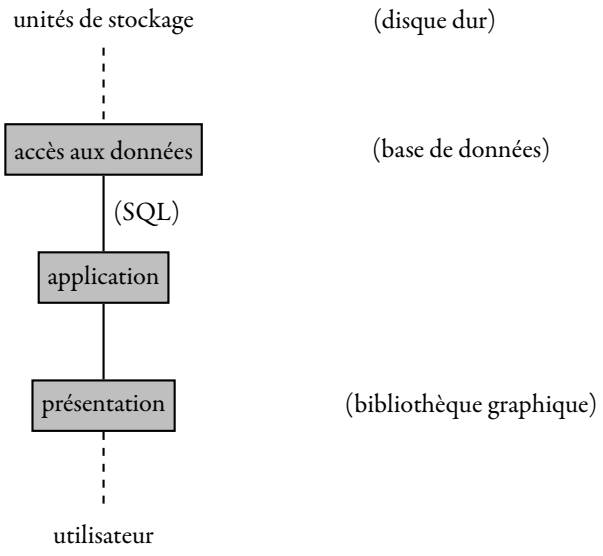


FIGURE 3.10 – Modèle d'architecture à trois couches.

Définition. *On dit qu'une base de données suit le modèle relationnel lorsqu'elle structure ses données en **tables** dont les colonnes représentent des **attributs** et les lignes des **tuples**, différentes tables étant mises en relation par la présence d'attributs communs.*

Les constructions plus anciennes de bases de données comptaient notamment le modèle hiérarchique et le modèle réseau dont les données étaient organisées sous forme d'arbre et de graphe. Elles ont depuis été largement supplantées par le modèle relationnel, ce dernier étant beaucoup plus flexible et simple même si moins performant puisqu'il impose de stocker en mémoire la table complète des tuples.

Exemple. *La base de données utilisée par Twitter pourrait ressembler à la figure 3.11.*

Remarque. *Comme ci-dessus, les tables comportent presque toujours une colonne appellée « clef primaire » de type « `uint64` » permettant d'identifier de manière unique chaque tuple de la table, quitte à ce que ce soit là sa seule utilité.*

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

```
.open twitter.db;
CREATE TABLE Users (Clef INTEGER, Nom TEXT, Prenom TEXT, Identifiant TEXT);
INSERT INTO "Users" VALUES(44,'Obama','Barack','BarackObama');
INSERT INTO "Users" VALUES(45,'Trump','Donald','realDonaldTrump');
UPDATE Users SET Prenom='Donald J.' WHERE Clef=45;
SELECT Prenom,Nom FROM Users WHERE Clef=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 types de colonnes supportés incluent notamment « `BOOLEAN` », « `INTEGER` », « `FLOAT` », « `DATE` », « `TIME` », « `TEXT` » et « `BINARY` », créer des tables « `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 liste les tables et « `.dump table` » qui affiche les commandes SQL nécessaire à recréer de zéro l'état actuel d'une table.*

Les principales transactions du langage SQL sont les suivantes :

- « `INSERT INTO table (colonnes) VALUES (valeurs)` » ajoute un nouveau tuple à la table. Lorsqu'une liste de colonnes est donnée elle est utilisée pour placer les valeurs.
- « `UPDATE table SET colonne=valeur WHERE condition` » affecte la valeur à la colonne pour tous les tuples de la table vérifiant la condition.
- « `DELETE FROM table WHERE condition` » supprime de la table les tuples vérifiant la condition.
- « `SELECT colonnes FROM tables` » extrait des données d'une ou plusieurs tables. Les colonnes 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 « `*` ». Les tables admettent deux séparateurs, « `,` » et « `JOIN` », le premier signifiant un produit cartésien et le second une union disjointe. 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.

Exercice. *Afficher le prénom et le nom de tout utilisateur ayant posté un message de longueur inférieure à soixante caractères.*

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

Exercice. *Afficher le prénom de tout utilisateur ayant posté après 2015.*

```
SELECT Prenom FROM Users,Tweets,Events
WHERE Users.Identifiant=Tweets.Identifiant
AND Tweets.Clef=Events.Clef
AND Date>2015;
```

Nous avons ici réalisé le produit cartésien de trois tables afin d'en sélectionner les attributs correspondants ; on imagine bien que cette approche passe difficilement à l'échelle. Aussi dispose t-on de la clause « `JOIN` » qui fusionne deux tables en une suivant une condition donnée ; elle s'utilise ainsi :

```
SELECT Prenom FROM Users
INNER JOIN Tweets ON Users.Identifiant=Tweets.Identifiant
INNER JOIN Events ON Tweets.Clef=Events.Clef
WHERE Date>2015;
```

Enfin, le langage SQL offre aussi des fonctions permettant de « résumer » les lignes sélectionnées, notamment « `COUNT` », « `MIN` », « `MAX` », « `SUM` » et « `AVG` ». Par exemple, la transaction « `SELECT COUNT(colonnes) FROM tables WHERE ...` » aura pour effet d'afficher le nombre total de lignes sélectionnées par cette requête.

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

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

Les exercices ci-après portent sur une base de données plus fournie et vraisemblable que nos exemples ci-dessus. Elle organise les données (fictives) d'un commerce de vente de musique en ligne ; ses onze tables et leurs relations sont illustrées par la figure 3.12. Avant tout chose, charger cette base de données 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.*

Table « Users »

Clef	Nom	Prenom	Identifiant
45	Trump	Donald	realDonaldTrump
44	Obama	Barack	BarackObama

Table « Events »

Clef	Date	Heure
896523232098078720	2017-08-12	14:06
516382177798680576	2014-09-28	21:21
257552283850653696	2012-10-14	08:43

Table « Tweets »

Clef	Identifiant	Message
896523232098078720	BarackObama	No one is born hating another person because of the color of his skin.
516382177798680576	realDonaldTrump	Windmills are the greatest threat in the US to both bald and golden eagles.
257552283850653696	realDonaldTrump	I have never seen a thin person drinking Diet Coke.

FIGURE 3.11 – Base de donnée pouvant ressembler à celle utilisée par Twitter.

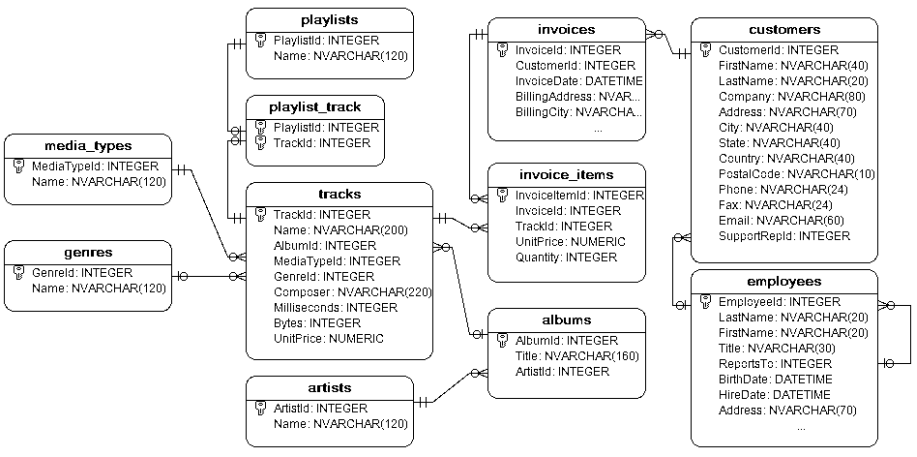


FIGURE 3.12 – Tables et relations de la base de données chinhook [10].

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

$$\{000, 001, 011, 010, 110, 111, 101, 100\}.$$

Mathématiquement, le monoïde libre $\Sigma^\star$ n'est autre que l'ensemble $\Sigma^{(\mathbb{N})}$ des familles finies de lettres ; alors, un langage n'est qu'un élément de $\mathfrak{P}\left(\Sigma^{(\mathbb{N})}\right)$. L'identification avec ces concepts classiques s'arrête cependant ici et 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 $xy = x \cdot y = x_1 \cdots x_n y_1 \cdots y_m$.*

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 une sous chaîne (ou sous mot) d'un mot m s'ils existent des mots u et v tels que $m = uxv$.*

Les sous chaînes sont ainsi exactement les suffixes des préfixes ou, de manière équivalente, les préfixes des suffixes.

Exercice. *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.*

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 une sous chaîne donnée.

Algorithme naïf. Adoptons pour commencer l'approche directe consistant à comparer une à une les lettres du mot avec celles de la sous chaîne recherchée.

Exercice. *Écrire une fonction Caml prenant en argument deux mots et déterminant si le premier est une sous chaîne 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 prefixe = function
| [],_ -> true
| h::t, [] -> false
| h::t, i::u -> h=i && prefixe (t,u)
in
let rec souschaine x = function
| [] -> x=[]
| h::t -> prefixe (x,h::t) || souschaine x t
;;
```

```
let souschaine x y =
    let r = ref false in
    let m = string_length x in
    let n = string_length y in
    for i=0 to n-1 do
        let j = ref 0 in
        while !j<m && !j+i<n && x.[!j]=y.[!j+i] 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 $x_1x_2\cdots x_m$ la sous chaîne à rechercher (ou motif) et $y_1y_2\cdots y_n$ le mot dans lequel la rechercher (ou texte).

Algorithme de Knuth–Morris–Pratt [6]. 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 souschaine(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.

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* $L \cdot M = \{x \cdot y : x \in L, y \in M\}$;
— *la puissance* $L^0 = \{\varepsilon\}$ *et* $L^{k+1} = L^k L$ *pour* $k \in \mathbb{N}$;
— *l'étoile* $L^\star = \bigcup_{k\in\mathbb{N}} L^k$.

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* $L = \{u \in \Sigma^\star : |u| \text{ pair}\}$ *et* $M = \{u \in \Sigma^\star : |u| \text{ impair}\}$. *Leur somme vaut* $L + M = \Sigma^\star$ *et leurs produits* $LL = L$, $LM = ML = M$ *et* $MM = L \smallsetminus \{\varepsilon\}$. *Pour* $k \geqslant 1$, *leurs puissances valent alors* $L^k = L$ *et* $M^k = L \cap \{u \in \Sigma^\star : |u| \geqslant k\}$ *si* k *est pair ; et pareillement avec « $M \cap \ldots$ » pour* k *impair. Ainsi leurs étoiles sont* $L^\star = L$ *et* $M^\star = \Sigma^\star$.

Fort 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* $\sigma \in \Sigma$. *Un langage est dit rationnel s'il admet une expression rationnelle.*

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

Exemple. *Les langages ci-dessous sont rationnels :*
— $\{\varepsilon\} = \varnothing^\star$
— *tout ensemble fini de mots*
— *les mots de longueur pair :* $(\Sigma\Sigma)^\star$
— *les mots de longueur impair :* $\Sigma(\Sigma\Sigma)^\star$
— *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* s *donné :* $s\Sigma^\star$
— *les mots admettant une sous chaine* s *donnée :* $\Sigma^\star s \Sigma^\star$
— *les mots formés de blocs de lettres de longueur pairs :* $\left(\bigcup_{\sigma\in\Sigma} \sigma\sigma\right)^\star$

Nous ne disposons pas encore d'outil assez puissant pour démontrer qu'un langage n'est pas rationnel. Plus tard, nous établirons que c'est notamment le cas de $\{a^k b^k : k \in \mathbb{Z}\}$. 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.

```
let somme1 a b x =
    a(x) || b(x) ;;

let produit1 a b x =
    let n = string_length x in
    let r = ref false in
```

```
    for i = 0 to n do
        let y = sub_string x 0 i in
        let z = sub_string x i (n-i) in
        if a(y) && b(z) then r:=true;
    done;
    !r ;;

let rec etoile1 a x =
    let n = string_length x in
    if n<=1 then a(x) else
    let r = ref false in
    for i = 1 to n-1 do
        let y = sub_string x 0 i in
        let z = sub_string x i (n-i) in
        if a(y) && etoile1 a z then r:=true;
    done;
    !r ;;

type Langage = L of (string -> bool) ;;
let somme = function L(a) -> function L(b) -> L(somme1 a b) ;;
let produit = function L(a) -> function L(b) -> L(produit1 a b) ;;
let etoile = function L(a) -> L(etoile1 a) ;;

type ExpressionRationnelle =
| Vide
| Lettre of char
| Somme of ExpressionRationnelle*ExpressionRationnelle
| Produit of ExpressionRationnelle*ExpressionRationnelle
| Etoile of ExpressionRationnelle ;;

let rec evaluer = function
| Vide -> L(function x -> false)
| Lettre(a) -> L(function x-> string_length x=1 && x.[0]=a)
| Somme(a,b) -> somme (evaluer a) (evaluer b)
| Produit(a,b) -> produit (evaluer a) (evaluer b)
| Etoile(a) -> etoile (evaluer a) ;;
```

Nous démontrerons plus tard la proposition ci-dessous en utilisant des résultats théoriques profonds sur les langages rationnels qui sont hors de portée à ce stade du cours.

Proposition. *Le complémentaire de tout langage rationnel est un langage rationnel.*

Corollaire. *L'intersection et la différence de deux langages rationnels sont des langages rationnels.*

Démonstration. L'intersection de deux langages rationnels est le complémentaire de l'union de leurs complémentaires ; c'est donc un langage rationnelle. La différence de deux langages rationnels est l'intersection du premier avec le complémentaire du second ; c'est donc un langage rationnelle. □

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* $\sigma \in \Sigma$.

Lorsque l'on fera référence à la notion effective de motif en programmation plutôt qu'au concept théorique sous-jacent, on parlera ici d'expression régulière. Nous allons présenter cette notion telle que spécifiée par le standard POSIX [7, §2.8] comme « Extended Regular Expressions (ERE) » qui est notamment utilisée 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 ».*

Par simplicité, 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 1.6.

Définition. *Les caractères dits spéciaux sont ceux de «* `\|{}*+?.()[]^$` *». 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.*

Exemple Python, indique à quel(s) indice(s) l'expression est reconnue dans le texte donné.
FIXME
example : (simplified) regexp for email addresses
FIXME
Exemples. (phrases contenant à la fois les 3 mots fils, fille, mère mais pas le mot père...)

Définition. *Un langage local est défini par la donnée des préfixes de longueur 1, des suffixes de longueur 1 et des facteurs de longueur 2 interdits.*

Définition. *Expressions rationnelles linéaires (chaque lettre apparaît au plus une fois dans l'expression).*

Proposition. *Le langage d'une expression rationnelle linéaire est local.*

Passage d'une expression rationnelle standard à une expression linéaire par marquage.

On donne les algorithmes permettant de calculer les ensembles P (préfixes), S (suffixes) et F (facteurs) définissant le langage local associé à une expression linéaire. On précise la complexité de ces algorithmes. On remarque qu'il y a des langages locaux qui ne sont pas linéaires.

4.1.3 Automates

On introduit un outil effectif simple permettant de reconnaître certains langages. Il s'avèrera que ce sont exactement les langages rationnels.

Définition. *Automates finis non déterministes.*
transition, calcul, mots reconnus/acceptés, langage reconnu

exemples

Définition. *automate déterministe automate complet état accessible*

opérations : - automate union - automate concatenation - automate étoile - automate produit pour l'intersection
Déterminisation d'un automate.

Définition. *Un automate local est un automate déterministe (en général non complet) tel que pour chaque lettre a, toutes les transitions étiquetées par a arrivent dans un même état. Il est standard si aucune transition n'arrive sur l'état initial.*

Construction de l'automate local associé à un langage local. Complexité.
Construction d'un automate reconnaissant une expression régulière donnée.

Méthode de Glushkov. L'automate de Glushkov s'obtient par linéarisation de l'expression, calcul des ensembles définissant le langage local, construction de l'automate local, suppression des marques utilisées pour la linéarisation. (Cette procédure est aussi connue sous le nom d'algorithme de Berry-Sethi.) Complexité.

Méthode de Thompson [9]. Voir : `https://swtch.com/~rsc/regexp/regexp1.html`
Et la réciproque :

Théorème (Kleene)**.** *Les langages reconnus par les automates sont exactement les langages rationnels.*

Démonstration. Étant donné un automate, on construit une expression régulière reconnaissant les mots qu'il accepte. □

Corollaire. *Le complémentaire de tout langage rationnel est un langage rationnel.*

Démonstration. Soit L un langage rationnel. Étant donné un automate $\mathscr{A} = (Q, A, E, I, F)$ reconnaissant L, l'automate $(Q, A, E, I, Q \setminus F)$ reconnaît son complémentaire $\Sigma^\star \setminus L$. □

Lemme (dit « de l'étoile » ou « de pompage »)**.** *FIXME*

Corollaire. *Le langage* $\{a^k b^k : k \in \mathbb{Z}\}$ *n'est pas rationnel.*

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

Algorithmique et programmation I (Python)

3. *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* $r \to \infty$*.*

4. *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* $n \in \{1, \dots, 1234\}$ *?*

5. *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* $p \in \mathscr{P} \cap \{1, \dots, 10^3\}$ *?*

6 (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 ?*

7 (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, on a* $42 = 5 + 37$*. 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*.*

Ingénierie numérique et simulation (Numpy/Scipy)

8 (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.*

Faire de même pour la suite de Hénon définie par $x_{n+1} = 1 - a x_n^2 + b\, x_{n-1}$*. Voir la figure A.2.*

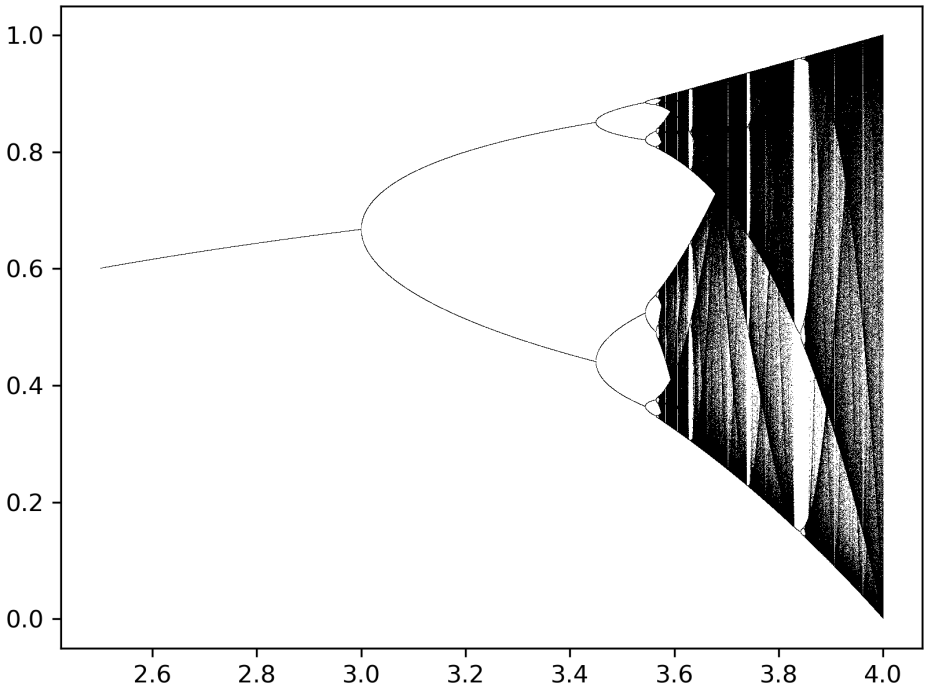


FIGURE A.1 – Valeurs d'adhérence de la suite logistique pour $\lambda \in [5/2, 4]$.

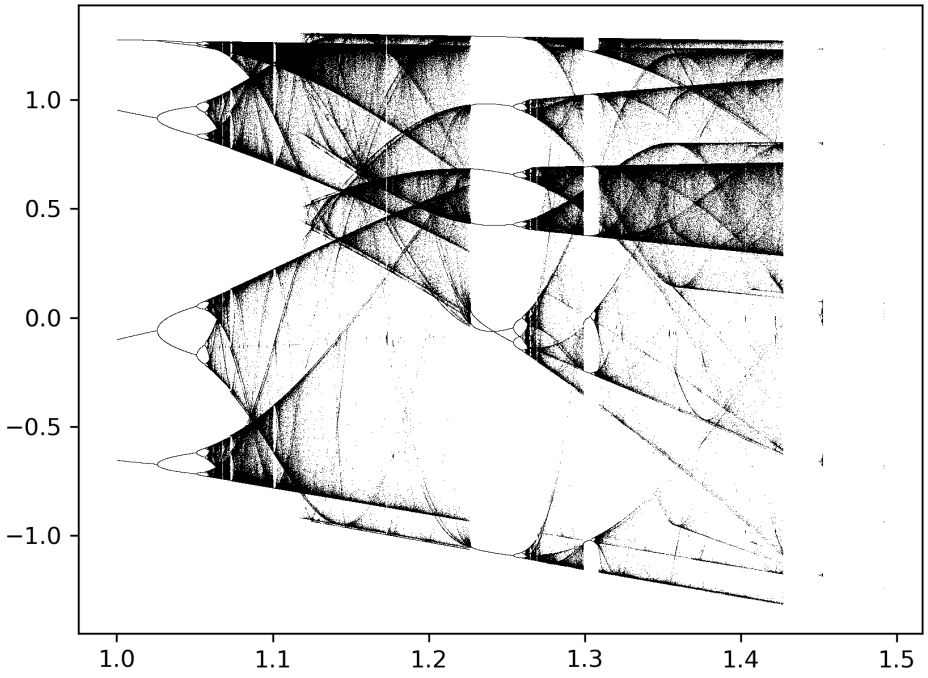


FIGURE A.2 – Valeurs d'adhérence de la suite de Hénon pour $a \in [1, \frac{3}{2}]$ et $b = \frac{3}{10}$.

A.2 Second semestre

Méthodes de programmation

9. *Écrire une fonction renvoyant la liste des facteurs premiers d'un entier donné.*

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

11. *Programmer le jeu de Nim. Voir* `https://en.wikipedia.org/wiki/Nim`. *On pourra utiliser les fonctions* « `read_int` » *et* « `random__int` ».

12. *On définit en Caml un type pour une classe d'expressions mathématiques :*

```
type Expression =
| Constante of float
| Variable of string
| Somme of Expression*Expression
| Différence 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 traduisant une chaîne de caractères du type « `(x+1)/2` » *en une expression comme ci-dessus. On pourra utiliser les fonctions Caml de conversion telles* « `int_of_char` », « `float_of_int` », *etc.*

13. *On considère un labyrinthe donné sous la forme :*

```
let m = [|
  "   XX";
  " XXX ";
  " X   ";
  "   X "
  |] ;;
```

L'objectif est de trouver une suite de cases adjacentes non marquées « `X` » *et reliant les cases* $(0,0)$ *et* $(n-1,n-1)$.

Utiliser la méthode du parcours en profondeur afin de déterminer si un tel chemin existe.

Utiliser la méthode du parcours en largeur afin de déterminer le plus court tel chemin.

Structures de données et algorithmes I

14. *On se propose d'énumérer toutes les permutations* σ *de* $\{1,\dots,n\}$ *en représentant chacune par le vecteur* $[|\sigma(1),\sigma(2),\dots,\sigma(n)|]$*; 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.*

Initiation aux bases de données (SQL)

A.3 Troisième semestre

Algorithmique et programmation II (Python)

15. *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 ?

16 (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 redéplacer les $n-1$ premiers entiers sur la troisième pile. Implanter cette stratégie sous forme de programe récursif. Quelle est le nombre d'étapes utilisées ?

17. *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.*

18 (jeu de la vie). *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 voisinnes étaient vivantes.*
- *Au temps t elle était vivante et deux ou trois de ses voisinnes aussi.*

Programmer l'évolution de ce système pour différents états initiaux.

Structures de données et algorithmes II

19. *On modélise par un arbre de décision un algorithme triant une liste de n éléments distincts $(x_1, \dots, x_n)$. Donner un minorant de sa hauteur. On considère maintenant le cas $n = 4$.*

Supposant qu'à l'exception d'un unique élément la liste soit déjà triée, on souhaite que l'algorithme 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 ?

20. *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.*

21 (arbre AVL). *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.3, celle de la figure A.4 ou l'une de leurs symétriques.*

Implanter ces opérations d'insertion, recherche et suppression puis en borner la complexité.

22 (tri par tas). *Écrire une fonction triant un vecteur 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 vecteur donné et un nombre fini de variables, à l'exclusion de tout autre structure de données.*

23. *La boulangerie de l'exercice 15 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'établissemt 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

24. *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.*

25 (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.*

26 (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.*

27. *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 évenements* $(x, y) \in A$ *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* $\geqslant n/2$ *?*

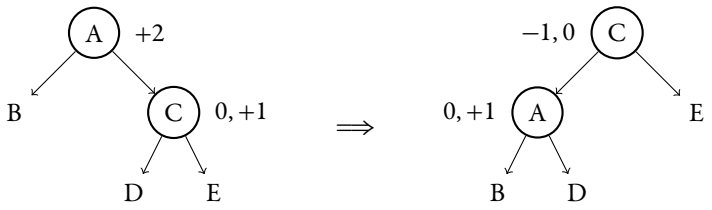


FIGURE A.3 – Ré-équilibrage d'un nœud par rotation simple.

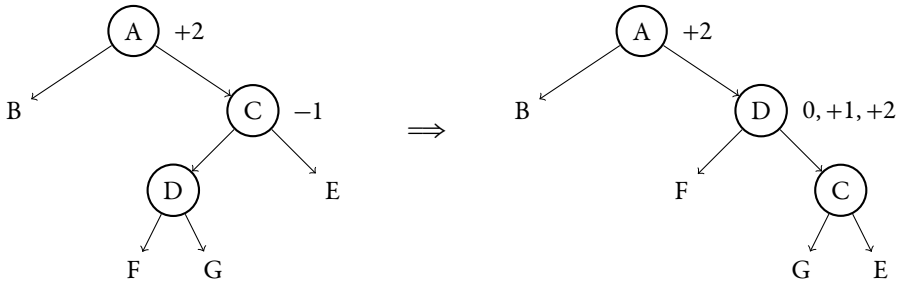


FIGURE A.4 – Ré-équilibrage d'un nœud par rotation double. Ramène au cas de la figure A.3.

A.4 Quatrième semestre

Motifs, automates et expressions

28 (arbre des suffixes). *FIXME*

29 (plus court surmot commun). *Voir [1, §1.4].*

Bibliographie

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

[2] Richard Peirce Brent et Paul Zimmermann. *Modern Computer Arithmetic*.
Monographs on Applied and Computational Mathematics.
Cambridge University Press, 2010. ISBN : 0521194695.

[3] *Caml Light*. A lightweight, portable implementation of the core Caml language.
INRIA. 2002. URL : https://caml.inria.fr/caml-light/.

[4] Edgar Frank Codd. "A relational model of data for large shared data banks".
In : *Communications of the ACM* 13.6 (juin 1970), pages 377-387.
DOI : 10.1145/362384.362685.

[5] *Enseignement de l'option informatique en classes préparatoires scientifiques*.
Note de service numéro 2013-0020 du 4 novembre 2013. NOR : ESRS1327059N.
Ministère de l'enseignement supérieur, 4 nov. 2013. URL : http://www.education.gouv.fr/pid25535/bulletin_officiel.html?cid_bo=75016.

[6] Donald Knuth, James Morris et Vaughan Pratt.
"Fast Pattern Matching in Strings".
In : *SIAM Journal on Computing* 6.2 (1977), pages 323-350.
DOI : 10.1137/0206024.

[7] *Portable Operating System Interfaces (POSIX). Part 2 : Shell and Utilities*.
Standard for Information Technology. IEEE, 1993. ISBN : 0-7381-1376-X.
DOI : 10.1109/IEEESTD.1993.6880751.

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

[9] Kenneth L. Thompson. "Regular expression search algorithm".
In : *Communications of the ACM* 11.6 (1968), pages 419-422.
DOI : 10.1145/363347.363387.

[10] SQLite Tutorial. *Chinook SQLite sample database*. 2017.
URL : http://www.sqlitetutorial.net/sqlite-sample-database/.