ILTNSI

Remerciements

Les auteurs et l'équipe de Prepamath Édition tiennent à remercier Jean-Côme Charpentier et David Guénée pour l'aide précieuse qu'ils ont apportée à cet ouvrage.

Illustrations : Prisca Baverey, Stéphane Pasquet

Coordination éditoriale : François Déliac

Conception couverture : Simon Geliot

Votre avis nous intéresse : `contact@prepamath.fr`

Retrouvez l'actualité des Interros des Lycées sur Facebook `facebook.com/interrosdeslycees/` et Twitter `@prepamath`.

Avant-propos

En classe de terminale générale, l'enseignement de spécialité de numérique et sciences informatiques (NSI) qui entre en vigueur à la rentrée 2020 est entièrement nouveau.

Le programme est conçu autour de quatre thèmes principaux :

- les données et leurs représentations,
- les bases de données,
- la programmation, les langages et les algorithmes,
- les architectures matérielles et réseaux.

À ces thèmes s'ajoutent des notions transverses telles que l'histoire de l'informatique ou les interfaces qui permettent la commande des systèmes.

Comme pour notre ouvrage de première NSI, nous avons mis l'accent sur la programmation et pour cela, proposé de télécharger la plupart des programmes mentionnés dans ce livre.

À partir de votre smartphone ou de votre tablette, l'application *Nathan Live* vous permettra très facilement de télécharger les fichiers présentés dans l'ouvrage et de les envoyer par mail vers un ordinateur personnel. Gagnant ainsi un temps précieux, vous pourrez tester ces programmes chez vous et/ou les modifier pour exercer votre créativité et assimiler parfaitement leur fonctionnement.

Dans ce livre, les méthodes les plus utiles de Python 3 vous sont présentées. La liste exhaustive se trouve sur le site suivant :

https ://docs.python.org/fr/3/

Accessibles aussi via Nathan Live, des vidéos vous proposent des explications détaillées sur le fonctionnement de certains algorithmes (voir la liste complète en fin d'ouvrage).

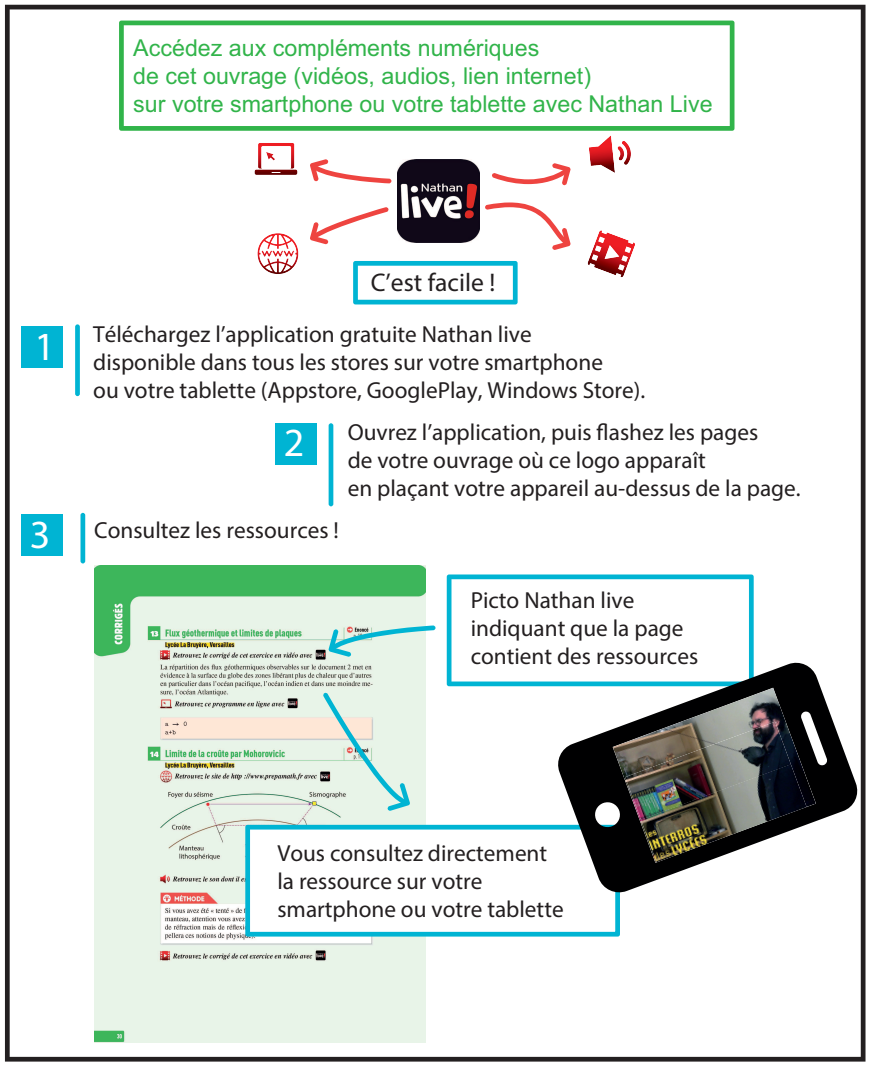
Même si l'informatique s'appuie sur une base théorique, nous insistons sur l'importance fondamentale de la pratique : chaque élève doit être capable de prendre des initiatives pour construire ses propres programmes, se poser des questions et ainsi les faire évoluer à sa guise.

Nombreux et variés, les exercices proposés dans cet ouvrage formeront un bon tremplin pour atteindre cet objectif.

Nous vous souhaitons à présent une bonne lecture et un bon apprentissage de cette matière passionnante, en constante évolution et pleine d'avenir.

Les auteurs

Table des matières



Pour télécharger les programmes contenus dans cet ouvrage :

`http://www.prepamath.fr/ILTNSI`

Méthodes de travail

Les conseils qui suivent ont été mis en pratique par un grand nombre de majors (sortis premiers) de Polytechnique ou de l'ÉNA, par des professionnels de l'organisation et sont également recommandés par de nombreux professeurs. Pour approfondir votre méthode de travail et obtenir les meilleurs résultats pendant vos études, nous vous recommandons la lecture du livre « Comment travailler plus efficacement », par F. Déliac, U. Hadrien, E. Matrullo et E. Maurette.

Faire des « feed-back »

Le « ***feed-back*** » est le conseil le plus important et le plus utilisé par ceux qui réussissent brillamment leurs études. Il consiste à contrôler systématiquement, sans s'aider de notes, ce que l'on vient d'apprendre (exercices et cours). Ce contrôle peut se faire mentalement, oralement ou par écrit.

- Dans les transports, essayez de vous rappeler mentalement, et sans vous aider de vos notes, le cours et les exercices vus le matin en classe (feed-back mental).
- Après avoir relu votre cours le soir, essayez de retrouver par écrit les principaux paragraphes et démonstrations sans regarder votre leçon (feed-back écrit).
- Après avoir résolu un problème, prenez 5 minutes pour contrôler par écrit que vous vous rappelez clairement l'énoncé ainsi que la démarche de résolution (feed-back écrit).
- Expliquez à des amis la leçon que vous venez d'apprendre ou l'exercice que vous venez de résoudre : c'est un excellent feed-back oral. Choisissez le type de « feed-back » qui vous convient le mieux et faites-en le plus régulièrement possible (après chaque cours et chaque série d'exercices). Pour être efficace, un « feed-back » doit se faire sans l'aide de vos notes. Ainsi, faire des fiches de résumés de cours à partir de vos cahiers ouverts ne constitue nullement un « feed-back ».

Miser sur la qualité

De nombreux témoignages démontrent que pour obtenir de bons résultats, il est préférable de faire un nombre limité d'exercices de manière approfondie plutôt que d'en survoler un grand nombre. Une tendance très répandue consiste à abattre une grande quantité d'exercices, à la chaîne, mais superficiellement, en espérant que le jour du contrôle, on aura déjà vu ce type de problème et que l'on saura s'en souvenir. Cette méthode est absolument inefficace car la seule manière de se souvenir d'un exercice de mathématiques ou de physique, c'est de l'avoir parfaitement compris et assimilé. Ainsi :

- À la fin d'un problème, prenez 5 à 10 minutes pour essayer de trouver un moyen de le généraliser ou de le compliquer (c'est ce que font souvent les professeurs pour concevoir leurs contrôles écrits) ; trouvez ce que cela pourrait changer dans la solution.

- Prenez également l'habitude, après chaque exercice, de faire un « feed-back » en faisant ressortir la démarche générale et en tissant des liens avec le cours. Bref, il ne faut pas vous contenter de résoudre l'exercice, mais il vous faut lui apporter de la valeur ajoutée et vous interroger sur son contenu.
- *Idem* pour le cours. Ne vous contentez pas de le parcourir de manière passive. Il vous faut avoir la rigueur d'effacer toutes les zones d'ombre. Pour chaque théorème, il faut vous demander quels types d'exercices son utilisation permettra de résoudre.

Travailler par « couches successives »

Cette méthode, très utile pour les étudiants préparant des examens ou des révisions, peut également être utilisée dès le lycée.

On observe que pour apprendre un gros volume de cours, rien n'est plus inefficace que de l'attaquer de front, de manière linéaire. La bonne manière consiste à d'abord survoler l'ensemble, en ne retenant que la structure, c'est-à-dire les grands titres, ainsi que les noms des paragraphes (première couche, étape devant durer 5 minutes). Dans l'étape suivante (deuxième couche, d'une durée de 10 minutes), on reprend son cours du début en retenant cette fois également les théorèmes et résultats importants. Après cette deuxième couche, on a déjà une idée claire de la structure de l'ensemble du cours. On peut alors aborder la dernière étape (troisième couche) : on reprend son cours au début pour, cette fois-ci, l'étudier en profondeur en apprenant le détail des démonstrations.

Il est à noter que cette méthode peut être également appliquée avec succès à des matières littéraires, ainsi qu'aux révisions du bac de français. Par exemple :

- Pour la préparation d'un contrôle, on commencera par passer en revue rapidement l'ensemble du cours et des exercices du chapitre précédemment étudié, avant de les réviser en détail. Ainsi aura-t-on développé une compréhension synthétique et claire.
- De même, avant d'aborder un problème volumineux (tel qu'un contrôle écrit), il est préférable d'en survoler l'ensemble avant de l'attaquer.

Travailler sa rapidité

Pour acquérir de la rapidité, trois voies sont possibles :

- Prenez l'habitude, en travaillant chez vous, de vous concentrer sur une seule chose à la fois, c'est-à-dire ne pas attaquer un problème ou une dissertation, en rêvassant à ce que vous pourriez trouver à manger dans le réfrigérateur ou en écoutant de la musique.
- Prenez l'habitude de travailler chez vous dans les mêmes conditions qu'en devoirs surveillés. Le minutage de chacun des exercices de ce livre est fait en ce sens. Cependant, cela ne devrait pas, une fois la résolution faite, vous empêcher d'y réfléchir plus calmement afin de vérifier la bonne assimilation du problème. Si les seuls moments où vous vous pressez sont les contrôles écrits, vous ne deviendrez jamais rapide.

- Essayez de contenir tout votre travail à la maison dans une plage horaire serrée. Engagez-vous, par exemple, à travailler chez vous tous les jours entre 18 h et 20 h et efforcez-vous de ne jamais déborder (quelle que soit votre charge de travail). En effet, si l'on ne se donne pas de limite de temps pour accomplir un travail, on a naturellement tendance à le laisser traîner en longueur et à rêvasser. L'étroitesse de la plage horaire vous obligera à ne pas vous endormir et à devenir efficace.

Spécial Baccalauréat

Le sujet de l'épreuve écrite comporte cinq exercices, parmi lesquels le ou la candidat.e choisit les trois qu'il ou elle traitera en 3h30.

- Le sujet comprend obligatoirement au moins un exercice relatif à chacune des trois rubriques suivantes :
  - → traitement de données en tables et bases de données ;
  - → architectures matérielles, systèmes d'exploitation et réseaux ;
  - → algorithmique, langages et programmation.

  Lisez donc dans les grandes lignes les énoncés des cinq exercices afin de voir ceux qui traitent de vos points forts.
- Pour chaque exercice, faites attention au temps : chaque exercice devrait être traité en 60 minutes approximativement. Laissez-vous 10 minutes pour vous relire à la fin de chaque exercice.
- Si vous êtes habitués à utiliser la méthode du feed-back exposée dans les premières pages de cet ouvrage, il est bon d'envisager un court feed-back mental sur les parties du cours concernées par l'épreuve du jour. Si vous êtes nerveux, cela vous aidera à vous mettre en confiance et à « démystifier » l'épreuve, que vous pourrez aborder avec la même sérénité qu'un devoir surveillé classique.
- Il faut absolument soigner la présentation. Au cours de l'année scolaire, votre professeur s'est habitué à votre style de rédaction, à votre écriture. Il sait ce que vous valez. Ce n'est pas le cas du correcteur qui lira votre copie. Il n'aura ni l'indulgence que pourrait avoir votre professeur, ni la patience d'essayer de vous déchiffrer si votre copie est présentée comme un brouillon. Par conséquent, il vous faut faire un réel effort de présentation le jour du bac, plus encore que pendant l'année scolaire. Vous devez notamment veiller à indiquer clairement le numéro des questions auxquelles vous répondez de façon à faciliter la correction de votre copie.
- Enfin, relisez-vous. Réservez-vous pour cela le temps nécessaire. Si vous vous laissez 60 minutes pour la résolution d'un exercice, il vous restera 10 minutes pour sa relecture. Celle-ci doit être très attentive (ce n'est pas toujours facile à l'issue de plus de trois heures d'épreuve). Enfin, souvenez-vous, si vous manquez de temps pour tout relire, qu'il vaut mieux une relecture partielle très attentive qu'une relecture « express » en diagonale, totalement inefficace pour détecter les erreurs résiduelles.

Chapitre 1

Structures indexées et dynamiques

Plan du chapitre

Exercice type

1 En Python, on considère une pile `P`.

(a) Quelle instruction permet d'empiler la valeur « 7 » dans `P` ?

(b) Quelle instruction permet de dépiler `P` ?

(c) Quelle instruction permet d'afficher les deux éléments au sommet de la pile sans les défiler ?

2 En Python, on considère une file `F`.

(a) Quelle(s) instruction(s) permet(tent) d'enfiler successivement les valeurs « 2 », « 7 » et « 3 » dans `F` ?

(b) Quelle instruction permet de défiler `F` ?

(c) Quelle instruction permet d'afficher l'élément en tête de file sans le dépiler ?

Voir corrigé page 10

1 Structures indexées

1.1 Notion de structure

À l'heure du *big data*, les quantités de données à traiter en informatique sont de plus en plus importantes. Le traitement de ces données (*data* en anglais) nécessite donc une organisation structurée et efficace. Plus les données seront structurées, plus leur traitement sera simple, et plus le risque d'erreur sera limité.

Exemple : un agenda contenant des personnes identifiées par leurs noms, prénoms, date de naissance adresse et numéro de téléphone est une structure de données.

COURS

Définition 1

Une *structure de données* est une manière d'organiser des données afin de pouvoir les traiter de façon simplifiée.

1.2 Structures indexées

1.2.1 - Tableaux

Définition 2

Un *tableau* est une structure de données représentant une séquence finie d'éléments auxquels on peut accéder par leur position dans la séquence, appelée *indice*.

L'indexation d'un tableau de n éléments commence à 0 (premier élément) et se termine à $n-1$ (dernier élément).

Exemple :
En Python, on peut définir un tableau de 5 entiers de la manière suivante :

```
tab = [ 7 , 12 , 45 , 0 , 12 ]
```

Ceci est un tableau unidimensionnel. Il existe aussi des tableaux multidimensionnels **dont chaque élément est lui-même un tableau**.

Exemple : la matrice :

$$\begin{pmatrix} 1 & 0 & 3 \\ 5 & -2 & 8 \end{pmatrix}$$

peut-être représentée en Python par le tableau suivant :

```
tab = [
        [ 1 , 0 , 3 ]
        [ 5 , -2 , 8 ]
      ]
```

Ici, `tab` est un tableau dans lequel chaque élément est un tableau de trois entiers.

Un tableau se caractérise par :

- le type des éléments qu'il contient (entiers, caractères, tableaux,...) ;
- le nombre d'éléments qu'il contient (sa *taille*).

Exemple : dans cet exemple écrit en C, impossible de définir `tab[3]` car la taille du tableau a été fixée à 3 éléments uniquement.

```
double[] tab = new double[3];
tab[0] = 8.7;
tab[1] = 2.3;
tab[2] = 1.5;
```

1.2.2 - Les dictionnaires

Définition 3

Un *dictionnaire*, aussi appelé *tableau associatif*, est une structure de données dans laquelle des *clés* sont associées à des *valeurs*.

Remarque : en mathématiques, l'équivalent du dictionnaire est une *application*, qui associe à chaque valeur de la variable une autre valeur.

Exemple :

En Python

```
capitale = { "la France" : "Paris",
             "l'Angleterre" : "Londres",
             "l'Allemagne" : "Berlin"}

for cle in capitale:
    print("La capitale de {} est {}.".format(cle,
  capitale[cle]))
```

Ce programme affiche :

```
La capitale de la France est Paris.
La capitale de l'Angleterre est Londres.
La capitale de l'Allemagne est Berlin.
```

2 Structures dynamiques

2.1 Listes

Définition 4

Une *liste* est une structure de données regroupant des éléments ordonnés qu'il est possible de manipuler individuellement (accès, ajout, suppression et modification).

Exemples de définitions et de manipulations de listes :

En C

```
List<double> maliste = new List<double>();

maliste.Add(7.2);
maliste.Add(9.7);
```

COURS

En Python

```
maliste = []
maliste.append(7.2)
maliste.append(9.7)
```

Remarques

- La taille d'un tableau est statique, contrairement à celle d'une liste.
- Le langage Python permet de confondre les deux notions car la notion de tableau n'est pas à proprement parler implémentée dans ce langage. Ainsi, quand on programme en Python, on parlera de tableaux ou de listes indifféremment.

2.2 Les piles

Retrouvez une explication des piles et des files en vidéo avec Nathan Live.

2.2.1 - Définition

Définition 5

Une *pile* est une liste remplie sur le principe du « dernier arrivé, premier sorti », abrégé en *LIFO*, de l'anglais « Last In, First Out ».

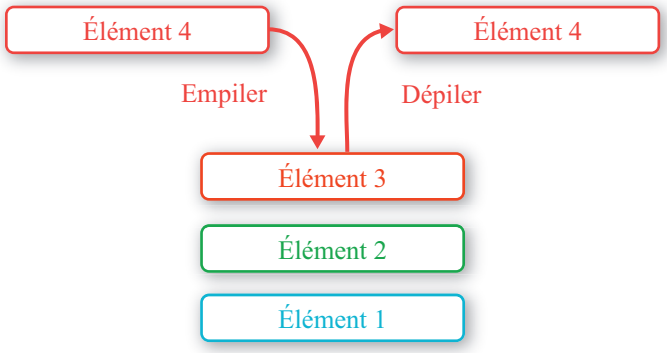


Figure 1.1 – Illustration d'une pile

Quand on ajoute un élément à une pile, on dit que l'on *empile* ; quand on retire un élément d'une liste, on dit que l'on *dépile*.

Exemples

- Dans un navigateur web, une pile sert à mémoriser les dernières pages visitées : la première à apparaître est la dernière à avoir été visitée (accessible en cliquant sur l'icône « Page précédente » par exemple).
- Dans un traitement de texte, la combinaison de touches « [Ctrl] + [Z] » annule la dernière action : cela repose sur le fait que les actions sont empilées.

2.2.2 - Les piles en Python

On peut implémenter une pile à l'aide d'une liste.

- `p.append(v)` ajoute `v` au sommet de la pile (à la fin de la liste) ;
- `p[-1]` est l'élément au sommet de la pile (c'est le dernier élément de la liste) ;
- `p.pop()` retire l'élément au sommet de la pile (le dernier élément de la liste). À noter que `p.pop(n)` supprime l'élément d'indice `n`.

Exemple :

```
pile = [] # pile vide
pile.append(3) # empile la valeur '3'
pile.append(7) # empile la valeur '7'
print(pile[-1]) # affiche : 7, le sommet de la pile
pile.pop()
print(pile) # affiche : [ 3 ]
```

2.3 Les files

2.3.1 - Définition

Définition 6

Une *file* est une liste remplie sur le principe du « premier arrivé, premier sorti », abrégé en *FIFO*, de l'anglais « First In, First Out ».
Quand on ajoute un élément à une file, on dit que l'on *enfile* ; quand on retire un élément d'une file, on dit que l'on *défile*.

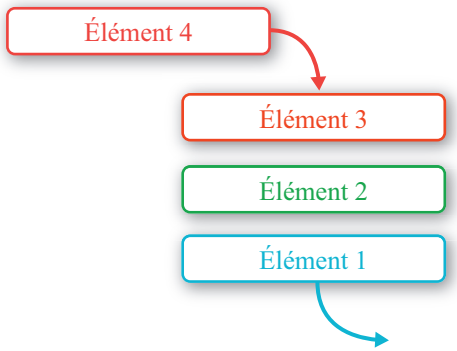


Figure 1.2 – Illustration d'une file

Exemple : lorsque l'on imprime plusieurs documents, les requêtes d'impression sont traitées selon une file.

COURS

2.3.2 - Les files en Python

Les listes sont mal adaptées pour représenter en Python des files car l'ajout d'élément se fait toujours en fin de liste, et non en début.

C'est la raison pour laquelle on pourra utiliser la fonction `deque` du module `collections`. La syntaxe de cette fonction est :

```
file = deque([ liste de nombres séparés par des virgules ])
```
ou : `file = deque((liste de nombres séparés par des virgules))`

Ainsi,

```
file = deque( (3,2,1) )
```

représente la file dont le premier élément (la tête de file) est « 1 » et dont le dernier élément enfilé est « 3 ».

On utilisera alors :

- `file.appendleft(élément)` pour enfiler un élément ;
- `file.pop()` pour défiler la tête de file. À noter toutefois qu'en mode console, écrire « `f.pop()` » défile l'élément défilé, ce qui n'est pas le cas dans un programme.

Exemple :

```
from collections import deque
f = deque( (3,2,1) ) # initialise la file
f.appendleft(4) # ajoute la valeur 4
print(f) # affiche la file
print(f.pop()) # affiche la tête de file et défile
print(f)
```

À RETENIR

Tableaux	Séquence finie et de taille fixe d'éléments auxquels on peut accéder par leur indice.
Dictionnaires	Association dynamique de paires clé/valeur.
Liste	Groupe d'éléments ordonnés que l'on peut manipuler individuellement.
Pile	Liste remplie sur le principe « dernier arrivé, premier sorti » (LIFO).
File	Liste remplie sur le principe « premier arrivé, premier sorti » (FIFO).

Voir exercices 1 à 12.

3 Packing et unpacking explicite en Python

3.1 Introduction

Nous avons vu en classe de 1re que la syntaxe suivante :

```
a , b = 3 , -2
```

permet de définir un tuple. C'est ce que l'on appelle l'*unpacking* implicite.
De même, quand on écrit :

```
L = [15 , 48]
```

on définit une liste ; c'est ce que l'on appelle le *packing implicite*.

Dans certaines situations, des fonctions qui admettent par exemple deux paramètres doivent être appelées à partir d'une liste.
Dans d'autres cas, il est utile de créer une fonction dont le nombre de paramètres peut être variable.
Fort heureusement, il existe des solutions simples pour cela.

3.2 L'unpacking explicite

Considérons l'exemple basique d'une fonction permettant de calculer la longueur entre deux points $A(x_A ; y_A)$ et $B(x_B ; y_B)$ dans un repère.

```
def longueur(xA,yA,xB,yB):
    return ( (xB-xA)**2 + (yB-yA)**2 )**0.5

A = (-2,5)
B = (7,3)
print( longueur(*A,*B) )
```

Cette syntaxe est assez pratique car elle est lisible et donc compréhensible assez facilement :

- on commence par définir deux points A et B à l'aide de leurs coordonnées, donc à l'aide de deux tuples ;
- ensuite, on fait appel à la fonction `longueur`, qui admet quatre paramètres. Mais comme `A` et `B` ne sont que deux variables, on transforme chacune d'elles en deux paramètres à l'aide de l'opérateur *splat* (« * »).
  C'est l'*unpacking* (en français, le « déballage ») explicite.

COURS

L'unpacking explicite peut s'appliquer aux tuples comme aux listes. Il peut aussi être utilisé sur les dictionnaires, mais avec une syntaxe légèrement différente :

```
def longueur(xA,yA,xB,yB):
    return ( (xB-xA)**2 + (yB-yA)**2 )**0.5

points = { 'xA' : -2 , 'yA' : 5 , 'xB' : 7 , 'yB' : 3 }

print( longueur(**points) )
```

On transforme ici les valeurs successives en paramètres.

À noter toutefois que les clés du dictionnaire doivent avoir le même nom que les paramètres de la fonction.

Ainsi,

```
points = { 'a' : -2 , 'b' : 5 , 'c' : 7 , 'd' : 3 }
```

ne fonctionne pas.

3.3 Le packing explicite

Nous pouvons nous en douter, le *packing* (en français, l'« emballage ») explicite est l'inverse de l'unpacking explicite.

Il est très utile dès lors que l'on écrit une fonction dont le nombre d'arguments est inconnu.

Par exemple, pour afficher un polynôme en fonction de ses coefficients, du terme de degré 0 vers le terme de plus haut degré, on pourra utiliser le programme suivant :

```
def polynome(*coef):
    r = ''
    for e in range(len(coef)):
        if e == 0 and coef[e] != 0:
            r += str( coef[e] )
        elif e == 1:
            if coef[e] == 1:
                r += '+x'
            elif coef[e] == -1:
                r += '-x'
            elif coef[e] < 0:
                r += str( coef[e] ) + 'x'
            elif coef[e] > 0:
                r += '+' + str( coef[e] ) + 'x'
```

(suite du programme page ci-contre)

```
        else:
            if coef[e] == 1:
                r += '+x^' + str(e)
            elif coef[e] == -1:
                r += '-x^' + str(e)
            elif coef[e] < 0:
                r += str( coef[e] ) + 'x^' + str(e)
            elif coef[e] > 0:
                r += '+' + str( coef[e] ) + 'x^' + str(e)

    return r

print( polynome(-1,0,-2) )
```

Ce programme affiche ici :

```
-1-2x^2
```

Si nous souhaitons manipuler non pas une liste mais un dictionnaire dans la fonction, nous devrons alors utiliser le double *splat*.

```
def polynome(**coef):
    e = 0
    for key,value in coef.items():
        print('Le coefficient du terme de degré {} est : {}
    '.format(e,value))
        e += 1

polynome(coef0 = -9, coef1 = 2, coef2 = 5)
```

Ce programme affiche :

```
Le coefficient du terme de degré 0 est : -9
Le coefficient du terme de degré 1 est : 2
Le coefficient du terme de degré 2 est : 5
```

Remarque : il est commun de voir `*args` et `**kwargs` dans de nombreux programmes. Ce n'est pas une obligation (comme nous venons de le voir) mais plutôt une habitude.

Voir exercices 13 à 17.

COURS

Solution de l'exercice type

1 En Python une pile est représentée par une liste.

(a) Empiler une valeur revient à ajouter cette valeur à la fin de la liste.
L'instruction qui permet d'empiler la valeur « 7 » dans `P` est alors :

```
>>> P.append(7)
```

(b) Dépiler `P` revient à enlever la dernière valeur de la liste.
Ainsi, l'instruction qui permet de dépiler `P` est :

```
>>> P.pop()
```

(c) Les deux éléments au sommet de la pile sont les deux derniers éléments de la liste.
Ainsi, l'instruction qui permet d'afficher d'abord le 2e puis le 1er est :

```
>>> P[-2] , P[-1]
```

Remarque : en ligne de commande, inutile de mettre « `print` ».

2 En Python, une file `F` est aussi représentée par une liste.

(a) Pour enfiler une valeur dans `F`, on peut utiliser `deque` du module `collections`.
Dans ce cas, on pourra utiliser les instructions suivantes :

```
>>> F = deque()
>>> for i in [2,7,3]:
>>> ... F.appendleft(i)
```

À ce stade, `F = deque([3, 7, 2])`.

(b) Défiler `F` signifie enlever l'élément « le plus ancien » de la file, à savoir ici « 2 ».
Il s'agit du dernier élément de l'objet `F` (on ne peut pas réellement dire ici que `F` est une liste car quand on affiche `F`, ce n'est pas exactement une liste que l'on voit).
Ainsi, pour défiler, on utilise l'instruction :

```
>>> F.pop()
```

(c) Comme pour la pile, l'instruction qui permet de connaître ou de retrouver l'élément en tête de la file `F` est :

```
>>> F[-1]
```

Voir énoncé page 1

1 QCM Vérification de connaissances

5 min

Corrigé p. 20

Pour chacune des questions suivantes, plusieurs propositions de réponses sont faites dont une seule est correcte. Laquelle ?

1 On souhaite insérer dans une structure de données les différentes hauteurs d'un arbre au fil des années sans y insérer les années. La structure de données la plus adaptée en Python est :

- **a** une liste ;
- **b** un dictionnaire ;
- **c** une pile ;
- **d** une file.

2 Dans une administration, on souhaite insérer dans une structure de données les doléances reçues afin de les traiter selon leur ordre d'arrivée. On aura alors plutôt intérêt à les enregistrer dans :

- **a** une liste ;
- **b** un dictionnaire ;
- **c** une pile ;
- **d** une file.

3 Dans une association, on souhaite enregistrer une liste d'informations pour chaque adhérent : taille, poids, etc. La meilleure structure de données pour cela est :

- **a** une liste ;
- **b** un dictionnaire ;
- **c** une pile ;
- **d** une file.

4 Pour un logiciel, on doit enregistrer la liste des actions dans une structure de données de sorte à voir la dernière action en priorité. On enregistrera donc les actions dans :

- **a** une liste ;
- **b** un dictionnaire ;
- **c** une pile ;
- **d** une file.

2 Fabricant de tee-shirts ★

5 min

Corrigé p. 20

Un fabricant décide de créer des tee-shirts dont la taille peut-être : XS, S, M, L, XL, XXL. À chaque taille son prix ; il adopte le principe suivant : 8 € pour la taille XS et il ajoute 2 € en passant à la taille supérieure, jusqu'au XXL.

1 Implémenter en Python ces informations dans la structure de données la mieux adaptée.

INTERROS

2 Ce même fabricant décide de changer sa façon de fixer le prix de revente des tee-shirts. Ceux dont la taille est XS sont toujours à 8 €, mais cette fois-ci, pour passer d'une taille à la suivante, il ajoute au prix de la taille inférieure la moitié de sa racine carrée.

Par exemple, pour obtenir le prix des tailles S, il fait :

$$8 + \sqrt{8} \div 2 \approx 9{,}41.$$

Proposer un programme Python qui automatise ces calculs et les enregistre dans une structure de données adaptée.

3 Séparation opérations / nombres

★ — 5 min — Corrigé p. 21

On considère une expression arithmétique `E` contenant des nombres et les opérations élémentaires (+, -, *, et /).

Écrire une fonction Python `separe(E)` qui renvoie deux listes : la première, nommée `nombres`, contenant les nombres de `E` et la seconde, nommée `operations`, contenant les opérations de `E`.

Par exemple, si `E = '3 + 5 * 7 - 2'` alors `nombres = [3,5,7,2]` et `operations = ['+','*','-']`

4 Contrôle de parenthèses

★ — 10 min — Corrigé p. 22

On considère une expression arithmétique qui peut contenir des parenthèses.

1 Écrire un algorithme permettant de vérifier que l'expression ne comporte pas d'erreur de parenthésage (s'il y a n parenthèses ouvrantes, il faut qu'il y ait aussi n parenthèses fermantes).

2 Implémenter en Python une fonction `controle(E)` qui renvoie « True » si l'expression `E` est correctement parenthésée, et « False » sinon.

5 Bon parenthésage ?

★ — 10 min — Corrigé p. 22

Une expression (algébrique ou arithmétique) est correctement parenthésée si d'une part, le nombre de parenthèses (et crochets) ouvrant.e.s et fermant.e.s est le même et si d'autre part les correspondances ne se croisent pas.

Par exemple, l'expression « $\big[3 + (5 - 7] * 3\big)$ » n'est pas correctement parenthésée car la parenthèse fermante ne peut pas venir après le crochet fermant.

1 Écrire un algorithme qui s'aide d'une pile pour contrôler le bon parenthésage d'une expression.

2 Écrire en Python une fonction `is_goodpar(E)` qui renvoie « True » si l'expression `E` est correctement parenthésée, et « False » dans le cas contraire.

6 Implémentation d'une file

★ 15 min Corrigé p. 24

Le module `collections`, avec `deque`, permet d'implémenter une file. Ainsi, les instructions :

```
>>> f = deque()
>>> f.appendleft(1)
>>> f.appendleft(2)
>>> print(f)
```

enfile la valeur « 1 » puis la valeur « 2 », et affiche : « `deque([2,1]` », qui représente la file.

Écrire une fonction `enfile(liste,tuple)` qui admet pour arguments une liste `liste` (qui nous sert de file) et un tuple regroupant les éléments à enfiler dans l'ordre que l'on veut.

Par exemple,

```
>>> f = []
>>> f = enfile( f , (1,2) )
>>> f = enfile( f , 3)
```

retourne la liste `[3,2,1]` (on a enfilé « 1 », puis « 2 », et enfin « 3 »).

7 D'une liste à un dictionnaire

★ 15 min Corrigé p. 24

Écrire en Python une fonction `list_to_dico(liste)` qui admet en argument une liste de la forme `[clé1,valeur1,clé2,valeur2,...]` et qui retourne le dictionnaire correspondant de la forme :

`{ clé 1 : valeur 1, clé 2 : valeur 2, ... }`.

8 Chiffrement

★★ 15 min Corrigé p. 25

La fonction `ord(<caractère>)` renvoie le point de code Unicode du caractère spécifié. Par exemple, « `ord('f')` » renvoie la valeur 102.

La fonction inverse est `chr(<entier>)`. Par exemple, « `chr(102)` » renvoie « f ».

Pour calculer a modulo n, on utilise la syntaxe « a % n ».

Pour chiffrer un mot M, on décide de procéder ainsi : pour chaque caractère c de M,

- on calcule $3 \times \text{ord}(c) - 61$ modulo 91, puis on ajoute 33. Le résultat est noté r ;

INTERROS

- on trouve ensuite le caractère correspondant à r avec `chr(r)` : on note ce caractère c_1.
- si r est pair, on calcule $2 \times \text{ord}(c) - 31$ modulo 91 puis on ajoute 33, et on trouve le caractère correspondant à ce résultat, noté c_2. Dans ce cas, le caractère c est chiffré en c_1c_2 ;
- si r est impair, on calcule $2 \times \text{ord}(c) - 32$ modulo 91 puis on ajoute 33, que l'on transforme en son caractère c_2, puis on calcule $3 \times \text{ord}(c) - 60$ modulo 91 puis on ajoute 33, que l'on transforme en son caractère c_3. Le caractère c est alors chiffré en $c_1c_2c_3$.

Par exemple, pour chiffrer « P » :

- `ord('P') = 80`, donc on calcule :

$$3 \times 80 - 61 \mod 89 = 179 \mod 91 = 88,$$

  puis on ajoute 33 :

$$r = 88 + 33 = 121,$$

  qui est impair et qui correspond à `chr(121)` = « y ». Donc c_1 correspond au caractère : « y » ;
- r est impair donc on calcule :

$$2 \times 80 - 32 \mod 91 = 37,$$

  puis on ajoute 33 :

$$37 + 33 = 70,$$

  qui correspond à $c_2 = \text{chr}(70) =$ « F » ;
- ensuite, on calcule :

$$3 \times 80 - 60 \mod 91 = 89$$

  puis on ajoute 33 :

$$89 + 33 = 122$$

  qui correspond à $c_3 = \text{chr}(122) =$ « z » ;
- « P » est donc chiffré en « yFz ».

Afin d'optimiser le temps de cryptage, on souhaite implémenter une structure de données nous permettant d'obtenir le cryptage de tous les caractères dont le point de code Unicode est compris entre 32 et 122 compris.

1. Implémenter la structure de données adéquate.
2. Chiffrer le message : « informatique ».

9 Labyrinthe

★★ 15 min Corrigé p. 26

Une manière d'implémenter un labyrinthe rectangulaire est de découper le rectangle en cases, puis de définir pour chaque case s'il y a un mur en haut, en bas, à gauche et à droite avec des booléens.

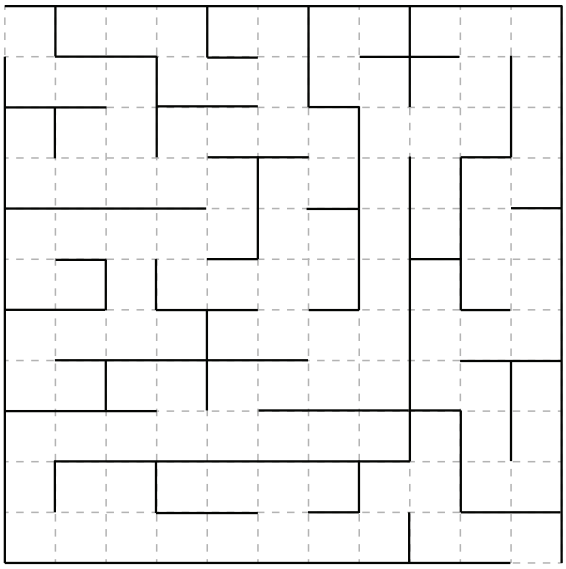


Figure 1.3 – Labyrinthe à implémenter en Python

Utiliser une structure de données adaptée afin d'implémenter ce labyrinthe selon cette façon de concevoir les choses.

On ne demande pas d'implémenter tout le labyrinthe, mais d'expliquer comment on ferait, en donnant l'exemple de la première case (en haut à gauche).

10 Suites imbriquées

★★ 20 min Corrigé p. 27

On considère deux suites numériques (u_n) et (v_n) définies par :

$$\forall n \in \mathbb{N}, \quad \begin{cases} u_0 = 5,\ u_1 = 3 \\ u_{n+2} = 2u_n + 3u_{n+1} \end{cases} \qquad \text{et} \qquad \begin{cases} v_0 = 2,\ v_1 = 4 \\ v_{n+2} = \dfrac{v_n}{v_{n+1}} + \dfrac{u_{n+2}}{u_{n+1}} \end{cases}$$

1 Vérifier que $u_2 = 19$ et $v_2 = \dfrac{41}{6}$.

2 Écrire un premier programme en Python qui permet d'afficher tous les termes des deux suites jusqu'à $n = 20$ à l'aide d'une boucle.

Quelle est sa complexité ?

INTERROS

3 Écrire un second programme qui fait la même chose que le premier, mais à l'aide d'une structure de données adaptée.
Quelle est sa complexité ?

11 Expressions postfixées

★★★ 20 min Corrigé p. 28

On ne considère dans cet exercice que les expressions numériques contenant les quatre opérations élémentaires (addition, soustraction, multiplication et division).

La notation postfixée consiste à mettre les opérations arithmétiques après leurs deux arguments. Ainsi, l'opération $a \bigstar b$, où $\bigstar \in \{+; -; \times; \div\}$, a pour notation postfixée : $ab\bigstar$.

1er exemple : l'expression 3 * ((4 + 5) + 6) peut s'écrire « 3 4 5 + 6 + * » (mais aussi : « 4 5 + 6 + 3 * » ou encore « 3 6 4 5 + + * »).

2e exemple : (4 + (5 + 6)) * 3 peut s'écrire « 4 5 6 + + 3 * ».

1 Écrire un algorithme utilisant une pile pour effectuer une opération en notation postfixée.

2 Implémenter l'algorithme en Python en définissant une fonction `calcul(E)`, où `E` est l'expression en notation postfixée dans laquelle chaque nombre ou symbole est séparé par une espace.

12 Permutations

★★★ 30 min Corrigé p. 30

On considère un ensemble $E = \{e_1; e_2; e_3; \ldots; e_8; e_9\}$ de 9 éléments.
On définit alors une application σ qui transforme E en l'ensemble :

$$E_1 = \{e_5; e_9; e_3; e_6; e_1; e_4; e_2; e_7; e_8\}.$$

σ est appelée une *permutation*, et on peut noter :

$$E_1 = \sigma(E).$$

Une notation matricielle de σ est alors :

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 5 & 9 & 3 & 6 & 1 & 4 & 2 & 7 & 8 \end{pmatrix}$$

car e_1 se transforme en e_5, e_2 en e_9, etc.

Pour tout entier naturel n, on note E_n l'ensemble :

$$E_n = \sigma^n(E) = \sigma(\sigma(\ldots\sigma(E)\ldots)),$$

où σ a été appliquée n fois à E.
Par exemple, $E_2 = \sigma\big(\sigma(E)\big)$.

1 Écrire une fonction `sigma(E,n)` retournant une liste des éléments de E_n. Par exemple, si :

```
E = [1,2,3,4,5,6,7,8,9]
```

alors :

```
sigma(E,1) = [5,9,3,6,1,4,2,7,8]
sigma(E,0) = [1,2,3,4,5,6,7,8,9].
```

2 Un théorème mathématique nous permet de dire qu'il existe un entier n tel quel $E_n = E_0$.
Écrire un programme en Python permettant de trouver le plus petit entier n tel que $E_n = E_0$.

3 Un *cycle* est un objet mathématique qui s'écrit par exemple (5,2,3) et qui signifie que l'élément 5 d'un ensemble devient l'élément 2 de cet ensemble, puis que l'élément 2 devient l'élément 3, et enfin que l'élément 3 devient l'élément 5, tout en conservant les autres éléments. Ainsi, sur un ensemble à 5 éléments,

$$(5,2,3) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 5 & 4 & 2 \end{pmatrix}.$$

Si σ et τ sont deux cycles, on note $\sigma \circ \tau$ le cycle correspondant au fait que l'on ait appliqué le cycle τ puis le cycle σ à un ensemble.

(a) Vérifier que sur un ensemble à 8 éléments :

$$(4,8) \circ (8,5) \circ (5,1) = (1,4,8,5).$$

(b) Écrire en Python une fonction `image(sigma,tau,n)` qui admet deux cycles en arguments et un entier (le nombre d'éléments d'un ensemble) et qui renvoie sous forme de liste la permutation correspondant à $\sigma \circ \tau$.
Vérifier que :

```
image( (1,3,4,5,8,6) , (2,7) , 8 )
```

retourne la liste :

```
[3,7,4,5,8,1,2,6].
```

Packing et unpacking

13 Longueur d'une ligne brisée

★ — 5 min — Corrigé p. 33

Nous disposons du dictionnaire suivant :

```
points = { 'A':(-2,4), 'B':(1,-2), 'C':(3,7), 'D':(5,-3) }
```

Compléter le programme page suivante afin qu'il retourne la longueur $AB + BC + CD$.

```
def longueur(xA,yA,xB,yB):
    return ( (xB-xA)**2 + (yB-yA)**2 )**0.5

def longTotale(points):
    somme = ...
    prem = ...
    for key,value in points.items():
        if prem:
            previous = ...
            prem = ...
        else:
            somme += ...
            previous = ...

    return somme

points = {
    'A' : (-2,4),
    'B' : (1,-2),
    'C' : (3,7),
    'D' : (5,-3)
}

print( longTotale(points) )
```

14 Somme de nombres

★ — 5 min — Corrigé p. 34

Écrire en Python une fonction `somme`, admettant un nombre indéfini d'arguments numériques, qui calcule la somme de tous les nombres mis en argument.

Par exemple, `somme(1,2,3,4,5,6,7,8,9)` doit retourner « 45 ».

15 Convertir une liste en dictionnaire

★ — 10 min — Corrigé p. 34

Écrire une fonction `convertToDico` qui admet un nombre indéfini de mots en argument et qui retourne un dictionnaire dont les clés sont nommées « clé 1 », « clé 2 », ... et dont les valeurs sont les mots passés en arguments.

Par exemple, `convertToDico("un","cheval","blanc")` retournera un dictionnaire de la forme :

```
{ "Clé 1" : "un",
"Clé 2" : "cheval",
"Clé 3" : "blanc" }
```

16 Calculs mathématiques

★ 10 min Corrigé p. 35

On souhaite que la ligne :

```
discr(a = 1, b = 2, c = -3)
```

appelle une fonction qui affiche la valeur $b^2 - 4ac$.

Proposer une telle fonction en utilisant la méthode du packing, c'est-à-dire une fonction définie par :

```
def discr(**coefs):
    ...
```

17 Remplacements

★ 10 min Corrigé p. 35

Nous disposons de 50 fichiers :

fichier1.txt, fichier2.txt, ... , fichier50.txt

dans lesquels nous souhaitons remplacer le mot « dinosaure » par « brachiosaure ».

On suppose que l'on a déjà à notre disposition la liste :

```
L = [
    "fichier1.txt" ,
    "fichier2.txt",
    ... ,
    "fichier50.txt"
]
```

contenant les noms de tous les fichiers.

Nous souhaitons alors créer une fonction `remplace` qui permet d'effectuer cette tâche en écrivant :

```
remplace("dinosaure","brachiosaure",*L)
```

Proposer une fonction.

CORRIGÉS

1 QCM Vérification de connaissances

Enoncé p. 11

1 Réponse **a**. Quand on observe l'évolution d'une quantité numérique, ce n'est en général pas pour sortir le premier ou le dernier élément en premier, donc pile et file sont à exclure.

De plus, il n'y a pas d'association donc un dictionnaire est impossible. Seule la liste convient pour ce genre de problème.

2 Réponse **d**. On enregistre les doléances selon le principe : « première arrivée, première traitée » (« first in, first out », donc FIFO) ce qui correspond à une file.

3 Réponse **b**. Il s'agit ici d'enregistrer des associations, c'est-à-dire des éléments qui sont rattachés à d'autres (adhérents → informations) donc le dictionnaire est le plus adapté. Ici, les clés seront les adhérents et les valeurs seront les informations.

Remarque : dans ce cas de figure, les informations pourront aussi être implémentées sous forme d'ensemble (*set*) car il y en a plusieurs. On peut alors imaginer une entrée du dictionnaire principal sous la forme :

```
{ 'Cécile' : { 157 , 48 , 17 } }
```

Ici, « Cécile » est une clé dont la valeur est l'ensemble `{157,48,17}`.

4 Réponse **c**. Il s'agit en effet d'enregistrer les actions selon le principe : « dernière action faite, première action affichée » (« last in, first out », donc LIFO), ce qui correspond à une pile.

À RETENIR

- Liste : succession de données accessibles sans contrainte.
- Pile (LIFO) : on imagine une pile de pièces où la dernière mise est la première à pouvoir être prise.
- **F**ile (**F**IFO) : on peut penser aux files d'attente, où la première personne arrivée est la première à être servie.

2 Fabricant de tee-shirts

Enoncé p. 11

Comme toujours, quand il s'agit d'associer deux entités (ici, taille et prix), on utilisera un dictionnaire.

1 On implémente le dictionnaire ainsi :

```
prix = {
    'XS': 8, 'S': 10, 'M': 12, 'L': 14, 'XL': 16,
    'XXL': 18
}
```

2 Dans cette question, il faut imaginer un programme qui effectue les calculs au fur et à mesure. On peut donc par exemple stocker les tailles dans une liste, puis créer le dictionnaire élément par élément à partir de cette liste comme dans la proposition suivante :

```
dico = {}
tailles = ['XS' , 'S' , 'M' , 'L' , 'XL' , 'XXL']

for i in range( len(tailles) ):
    if i == 0:
        dico[ tailles[i] ] = 8
    else:
        dico[ tailles[i] ] = round( dico.get(
    tailles[i-1] ) + dico.get( tailles[i-1] ) ** 0.5
     / 2 , 2)

print(dico)
```

Ce qui donne :

```
{
  'XS': 8, 'S': 9.41, 'M': 10.94, 'L': 12.59,
  'XL': 14.36, 'XXL': 16.25
}
```

Remarque : la racine carrée d'un nombre x peut s'écrire `x ** 0.5` car $\sqrt{x} = x^{1/2}$.

3 Séparation opérations / nombres

Enoncé p. 12

Il suffit ici de parcourir l'expression `E` est d'effectuer des tests pour savoir si le caractère rencontré est un nombre (avec la méthode `isdigit` par exemple), une opération, ou autre chose (en l'occurrence, une espace).

```
def separe(E):
    nombres , operations = [] , []
    for i in E:
        if i.isdigit():
            nombres.append(i)
        if i in '+-*/':
            operations.append(i)
    return nombres , operations

print( separe('3+5*9-6') )
```

CORRIGÉS

4 Contrôle de parenthèses

Enoncé p. 12

1 Il s'agit ici de compter le nombre de parenthèses ouvrantes et fermantes. Rien de bien compliqué.

```
E est une expression arithmétique
f ← 0 (nombre de parenthèses fermantes)
o ← 0 (nombre de parenthèses ouvrantes)
Pour chaque élément i de E:
   Si i = "(":
     o ← o + 1
   Sinon:
     Si i = ")":
       f ← f + 1
   Fin du Si
   Si f = o:
     Renvoyer True
   Sinon:
     Renvoyer False
```

2 Une implémentation possible est alors :

```
def controle(E):
    o , f = 0 , 0
    for i in E:
        if i == '(':
            o += 1
        if i == ')':
            f += 1
    return o == f
```

5 Bon parenthésage ?

Enoncé p. 12

1 Il s'agit ici d'empiler les crochets et parenthèses ouvrant.e.s et de dépiler quand on rencontre son équivalent fermant.e.

```
P est une pile vide
E est une expression
Pour chaque élément i de E:
   Si i = "(" ou i = "[":
     i est empilé
```

...

```
   Si i = ")":
     Si la tête de P est "(":
       On dépile "("
     Sinon:
       Renvoyer False
     Fin du Si (tête de P)
   Fin du Si i = ")"
   Si i = "]":
     Si la tête de P est "]":
       On dépile "["
     Sinon:
       Renvoyer False
     Fin du Si (tête de P)
   Fin du Si i = "]"
Si P est vide:
   Renvoyer True
```

2 Une implémentation de cet algorithme en Python est :

```
def is_goodpar(E):
    pile = []
    for i in E:
        if i == '(' or i == '[':
            pile.append(i)
        elif i == ')':
            if pile[-1] == '(':
                del pile[-1]
            else:
                return False
        elif i == ']':
            if pile[-1] == '[':
                del pile[-1]
            else:
                return False
    if len(pile) == 0:
        return True
```

Remarque : on aurait pu aussi écrire `pile.pop()` au lieu de `del pile[-1]`.

CORRIGÉS

6 Implémentation d'une file

Enoncé p. 13

Une implémentation possible est la suivante :

```
def enfile(liste,t):
    file = [i for i in liste]
    if isinstance(t,tuple):
        for i in t:
            file = [i] + file
    else:
        file = [t] + liste

    return file

f = [2,1]

f = enfile(f,(3,4))
f = enfile(f,5)
print(f)
```

Ainsi, on aura par exemple :

```
>>> f = [2,1] # On initialise la file comme une liste
>>> f = enfile(f,(3,4)) # On enfile 3, puis 4
>>> f = enfile(f,5) # puis 5
>>> f
[5, 4, 3, 2, 1]
```

7 D'une liste à un dictionnaire

Enoncé p. 13

Il s'agit ici de parcourir la liste donnée en argument en sautant un élément sur deux afin de mettre l'élément sauté en valeur.

Une fonction possible est alors la suivante :

```
def liste_to_dico(liste):
    d = {}
    for i in range(0,len(liste),2):
        d[ liste[i] ] = liste[i+1]

    return d
```

Ce qui donne par exemple :

```
>>> L = [ 'France' , 'Paris' ,
... 'Angleterre' , 'Londres' ,
... 'Espagne' , 'Madrid' ,
... 'Allemagne' , 'Berlin' ]
>>> print( liste_to_dico(L) )
{'France': 'Paris',
'Angleterre': 'Londres',
'Espagne': 'Madrid',
'Allemagne': 'Berlin'}
```

8 Chiffrement

Enoncé p. 13

1 **MÉTHODE**

On peut ici créer un dictionnaire où les clés seront les caractères dont le point de code Unicode est compris entre 32 et 122 et où les valeurs seront les résultats des chiffrements.

On obtient une implémentation comme celle-ci :

```
crypt = {}

for i in range(32,123):
    r = (3 * i - 61 ) % 91 + 33
    c1 = chr(r)
    if r % 2 == 0:
        c2 = chr( (2 * i - 31) % 91 + 33)
        crypt[ chr(i) ] = c1 + c2
    else:
        c2 = chr( ( 2 * i - 32) % 91 + 33)
        c3 = chr( ( 3 * i - 60) % 91 + 33)
        crypt[ chr(i) ] = c1 + c2 + c3
```

Ici, `crypt` est un dictionnaire auquel on pourra faire appel lors de tout chiffrement.

Suite de l'exercice page suivante.

CORRIGÉS

2 On peut utiliser le programme suivant :

```
def cryptage(M):
    r = ''
    for c in M:
        r = r + crypt[c]
    return r

M = "informatique"

print( cryptage(M) )
```

On obtient alors le résultat suivant :

```
ixjx(`s{)!)/*u%vQhR/30ixj&.26]p^
```

Téléchargez le programme complet.

9 Labyrinthe

Enoncé p. 15

MÉTHODE

On peut représenter la grille par une matrice `L`, où `L[i][j]` représente la case de la $(i+1)$-ième ligne en partant du haut (par exemple) et $(j+1)$-ième colonne en partant de la gauche (par exemple).
Ensuite, la valeur de chaque cellule est un dictionnaire dont les clés seront par exemple quatre booléens `top`, `bottom`, `left` et `right` représentant la présence ou l'absence de murs respectivement en haut, en bas, à gauche et à droite.

On aura ainsi :

```
M[0][0] = { 'top' : True , 'bottom' : False ,
            'left' : False , 'right' : True }
```

Cette implémentation a le mérite d'être claire, mais peu pratique car assez longue à faire. Ainsi, pour alléger, on peut opter pour une liste de booléens dans chaque cellule, en convenant de garder le même ordre (mais l'ordre peut différer selon la logique de la personne qui programme). On aura alors :

```
M[0][0] = [ True , False , False , True]
```

ce qui est bien plus simple et rapide à écrire, mais cette redondance introduit un risque de discordance des informations, souvent à l'origine de bugs dans la pratique.

Remarque : nous verrons plus tard qu'il existe d'autres façons, peut-être plus pratiques, de représenter un labyrinthe, notamment à l'aide d'une *classe* (voir le chapitre sur la Programmation Orientée Objet).

10 Suites imbriquées

Enoncé p. 15

1 Cette question est uniquement posée afin de voir si vous avez compris le mode de calcul des deux suites.

- $u_2 = u_{0+2} = 2u_0 + 3u_1 = 2 \times 5 + 3 \times 3 = 19$.
- $v_2 = v_{0+2} = \dfrac{v_0}{v_1} + \dfrac{u_2}{u_1} = \dfrac{2}{4} + \dfrac{19}{3} = \dfrac{41}{6}$.

2 Le premier programme peut ressembler à celui-ci :

```
u_penult , u_preced , v_penult , v_preced = 5 , 3 ,
    2 , 4

for i in range(2,21):
    u = 2*u_penult + 3*u_preced
    v = v_penult / v_preced + u / u_preced
    print('u(',i,') = ',u,' ; v(',i,') = ',v)
    u_penult = u_preced
    u_preced = u
    v_penult = v_preced
    v_preced = v
```

Dans ce genre de cas, où les termes d'une suite se calculent non pas en fonction du seul précédent mais du précédent et du pénultième, il faut nécessairement faire intervenir une variable transitoire.

C'est la raison pour laquelle nous avons ici choisi de calculer d'abord la valeur de deux variables transitoires `u` et `v`, qui jouent le rôle de u_n et v_n, puis de faire les affectations nécessaires pour la boucle suivante, à savoir dans un premier temps :

- `u_penult`, qui joue le rôle de u_{n-2}, devient `u_preced`, qui joue le rôle de u_{n-1},
- et enfin remplacer `u_preced` par la valeur du terme fraîchement calculé `u`

car en effet, d'une boucle à l'autre, u_{n-2} devient u_{n-1} et u_{n-1} devient u_n.

Le raisonnement est analogue pour les termes de la suite (v_n).

On voit ainsi s'afficher :

```
u( 2 ) = 19 ; v( 2 ) : 6.833333333333333
u( 3 ) = 63 ; v( 3 ) : 3.9011553273427473
u( 4 ) = 227 ; v( 4 ) : 5.3547924599257275
⋮
```

CORRIGÉS

```
u( 18 ) = 11975495755 ; v( 18 ) : 4.561936982933018
u( 19 ) = 42651360591 ; v( 19 ) : 4.561326557332556
u( 20 ) = 151905073283 ; v( 20 ) : 4.56168663914045
```

La complexité de ce programme est linéaire (13 affectations et opérations élémentaires par boucle, donc une complexité égale à $13n$). On dit que la complexité est en $O(n)$, si n représente l'indice du dernier terme affiché.

3 Dans cette question, on souhaite utiliser une structure de données. Il s'agit donc ici d'enregistrer la valeur des termes successifs dans une liste car cette dernière semble être la plus adaptée à nos exigences.

Un programme possible est alors :

```
u , v = [5,3] , [2,4]

for i in range(2,21):
    u.append( 2 * u[i-2] + 3 * u[i-1] )
    v.append( v[i-2] / v[i-1] + u[i] / u[i-1] )
    print('u(',i,') =',u[i],' ; v(',i,') =',v[i])
```

Ce programme est plus intuitif et plus court que le premier.

De plus, sa complexité, bien que linéaire aussi, est égale à $9n$: pour chaque terme, il y a 3 opérations plus une affectation dans la liste, soit 4 opérations, et une affectation pour la variable `i`.

Ce dernier programme est donc meilleur sous tous les angles : plus rapide, plus court et plus explicite que le premier.

11 Expressions postfixées

Enoncé p. 16

1 Un algorithme possible est le suivant :

```
P est une pile
E est une expression en notation postfixée
Pour chaque élément i de E:
   Si i est un nombre:
     Empiler i dans P
   Sinon:
     Dépiler les deux premières valeurs v1 et v2 de P
     Effectuer l'opération i sur v1 et v2
     Empiler le résultat dans P
   Fin du Si
```

2 Une implémentation possible de cette fonction est la suivante :

```
def calcul(E):
    expr = E.split(' ')
    liste_nombres = []

    for i in expr:
        if i not in '+-*/':
            liste_nombres.append(i)
        else:
            if i == '+':
                r = float(liste_nombres[-2]) + float
    (liste_nombres[-1])
            elif i == '-':
                r = float(liste_nombres[-2]) - float
    (liste_nombres[-1])
            elif i == '*':
                r = float(liste_nombres[-2]) * float
    (liste_nombres[-1])
            else:
                r = float(liste_nombres[-2]) / float
    (liste_nombres[-1])
            del liste_nombres[-2]
            del liste_nombres[-1]
            liste_nombres.append(str(r))

    return liste_nombres[0]
```

- On commence par découper la chaîne de caractères représentant l'expression à l'aide de la méthode `split(' ')`, qui va donc créer une liste dont les éléments sont ceux séparés par une espace dans E. Il s'agit donc ici des nombres et des opérations.
- On définit ensuite une liste `liste_nombres` qui va nous servir de pile.
- On parcourt ensuite la liste `expr` : si l'élément n'est pas une opération alors on enregistre cet élément dans la liste `liste_nombres` ; sinon, on effectue l'opération rencontrée entre les deux éléments de la liste `liste_nombres`.
- Une fois l'opération effectuée, on efface les deux premiers éléments de la pile, donc les deux derniers éléments de la liste `liste_nombres` (avec `del`) puis on ajoute à cette liste le résultat de l'opération trouvé.
- À la fin, la pile est composée d'un élément qui est le résultat de l'opération.

CORRIGÉS

12 Permutations

Enoncé p. 16

1 **MÉTHODE**

Il faut d'abord définir une fonction `permut(liste)` qui renvoie une liste dont les éléments sont ceux de l'argument `liste` permutés selon la permutation σ.
Ensuite, on définit la fonction `sigma(E,n)` qui retourne `E` si $n = 0$ et sinon, qui calcule `permut(E)` n fois (à l'aide d'une boucle).

Cela donne par exemple le programme suivant :

```
def permut(liste):
    P = [ 5, 9, 3, 6, 1, 4, 2, 7, 8]
    result = [ ]
    for i in P:
        result.append( liste[ i-1 ] )

    return result

def sigma(E, n):
    result = E
    for i in range(n):
        result = permut(result)

    return result
```

2 L'idée ici est de définir deux listes identiques `E` et `F` : la première va servir de référence (c'est l'ensemble initial) et la seconde sera transformée par σ tant que le résultat ne sera pas identique à `E`.
Cela donne :

```
E = [1,2,3,4,5,6,7,8,9]
F = [1,2,3,4,5,6,7,8,9]

id , n = False , 0

while id == False:
    n += 1
    F = sigma( F , 1 )
    if F == E:
        id = True

print(n)
```

On trouve $n = 4$.

3 **(a)** On doit partir du dernier cycle : (5,1) signifie que l'on permute les éléments 1 et 5 :

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 2 & 3 & 4 & 1 & 6 & 7 & 8 \end{pmatrix}$$

(8,5) signifie que l'on permute les éléments 8 et 5 du nouvel ensemble obtenu après la première permutation :

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 2 & 3 & 4 & 1 & 6 & 7 & 8 \\ 5 & 2 & 3 & 4 & 8 & 6 & 7 & 1 \end{pmatrix}$$

Et enfin, (4,8) signifie que l'on permute les éléments 4 et 8, ce qui donne :

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 2 & 3 & 4 & 1 & 6 & 7 & 8 \\ 5 & 2 & 3 & 4 & 8 & 6 & 7 & 1 \\ 5 & 2 & 3 & 1 & 8 & 6 & 7 & 4 \end{pmatrix}$$

Au final, on obtient la permutation $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 2 & 3 & 1 & 8 & 6 & 7 & 4 \end{pmatrix}$, ce qui donne : (1,5,8,4) (car les autres éléments ne sont pas permutés).

(b) Un programme possible est le suivant :

```
def cycle_to_permut(cycle,n):
    p = [0] * n
    for i in range(n):
        if i+1 in cycle:
            if cycle.index(i+1) == len(cycle) -
    1:
                p[i] = cycle[0]
            else:
                p[i] = cycle[ cycle.index(i+1) +
     1 ]
        else:
            p[i] = i+1
    return p

def permut(liste,p):
    L = [0] * len(liste)
    for i in range(len(liste)):
        L[i] = liste[ p[i] - 1 ]
    return L
```

```
def image(sigma,tau,n):
    E = range(1,n+1)
    t = cycle_to_permut(tau,n)
    s = cycle_to_permut(sigma,n)
    R = permut( permut(E,t) , s)
    return R
```

Nous avons opté ici pour créer une fonction `cycle_to_permut` qui transforme un cycle en une permutation (définie par une liste) afin de nous ramener à ce qui a été fait précédemment.

Puis nous nous sommes inspirés de la fonction `permut` de la question précédente pour en créer une autre avec un argument en plus (la permutation).

Enfin, nous avons créé la fonction principale `image` qui permet d'obtenir la permutation finale.

Pour vérifier l'exactitude du programme, on essaie avec les cycles donnés dans l'énoncé :

```
>>> sigma , tau = (1,3,4,5,8,6) , (2,7)
>>> image(sigma,tau,8)
[3, 7, 4, 5, 8, 1, 2, 6]
```

Cela signifie que :

$$(1,3,4,5,8,6) \circ (2,7) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 3 & 7 & 4 & 5 & 8 & 1 & 2 & 6 \end{pmatrix}.$$

POUR ALLER PLUS LOIN

Les permutations sont utiles dans la théorie du Rubik's Cube.
C'est notamment à l'aide de permutations que l'on peut démontrer que la répétition de certains mouvements conduit à l'état initial.

Par exemple, si le mouvement consiste à tourner la face droite de 90° vers le haut, puis de tourner la face du haut de 90° vers la droite, si on le répète 63 fois, on obtient la configuration initiale.

Téléchargez le programme complet.

13 Longueur d'une ligne brisée

Enoncé p. 17

Le programme complété est le suivant :

```
def longueur(xA,yA,xB,yB):
    return ( (xB-xA)**2 + (yB-yA)**2 )**0.5

def longTotale(points):
    somme = 0
    prem = True
    for key,value in points.items():
        if prem:
            previous = value
            prem = False
        else:
            somme += longueur(*previous,*value)
            previous = value

    return somme

points = {
    'A' : (-2,4),
    'B' : (1,-2),
    'C' : (3,7),
    'D' : (5,-3)
}

print( longTotale(points) )
```

Explications :

- dans la fonction `longTotale`, on commence par initialiser la variable `somme` à 0 : elle représentera la somme totale demandée ;
- ensuite, on initialise un booléen `prem` à `True` : il ne sert ici qu'à indiquer le premier point ;
- on parcourt ensuite le dictionnaire passé en arguments de la fonction avec la ligne :
  `for key,value in points.items()`
- si nous sommes sur le premier point, indiqué à l'aide du booléen, nous définissons la variable `previous` comme étant les coordonnées du point sur lequel nous sommes, et nous passons le booléen à `False` pour indiquer que les points suivants ne sont pas le premier point ;
- si nous ne sommes pas sur le premier point alors nous incrémentons la variable `somme` de la longueur entre le point précédent (dont les coordonnées sont stockées dans la variable `previous`) et le point actuel.

Nous utilisons ici l'unpacking lors de l'appel de la fonction `longueur` car les variables sont des tuples ;

- ensuite, nous définissons `previous` comme étant les coordonnées du point actuel (qui représenteront ainsi les coordonnées du point précédent quand nous serons sur le point suivant) ;
- quand le parcours du dictionnaire est fini, nous retournons la valeur contenue dans la variable `somme`.

14 Somme de nombres

Enoncé p. 18

Le packing explicite va nous servir pour écrire la fonction demandée :

```
def somme(*nombres):
    result = 0
    for n in nombres:
        result += n
    return result

print( somme(1,2,3,4,5,6,7,8,9) )
```

Dans notre fonction, `nombres` représente une liste grâce à l'opérateur *splat* (l'étoile).

Ainsi, il ne reste plus qu'à parcourir cette liste et à ajouter chaque élément.

15 Convertir une liste en dictionnaire

Enoncé p. 18

Voici une fonction possible :

```
def convertToDico(*mots):
    result = dict()
    for m in mots:
        num = mots.index(m) + 1
        key = "Clé " + str(num)
        result[key] = m

    return result

print( convertToDico("un","cheval","blanc") )
```

16 Calculs mathématiques

Enoncé p. 19

Une fonction possible est la suivante :

```
def discr(**coefs):
    a , b , c = coefs['a'] , coefs['b'] , coefs['c']
    delta = b**2 - 4*a*c
    print(delta)

discr(a = 1, b = 2, c = -3)
```

Les arguments de la fonction `discr` étant nommés respectivement `a`, `b` et `c`, en passant `**coefs` en argument de la fonction `discr`, on crée un dictionnaire dont les entrées sont nommées `a`, `b` et `c`, et dont les valeurs sont celles des variables `a`, `b` et `c`. C'est pour cela que l'on écrit « `a = coef['a']` » dans la fonction (et de même pour les autres variables).

17 Remplacements

Enoncé p. 19

Voici une proposition :

```
def remplace(original, remplacant, *args):
    for fichier in args:
        content = ""
        with open(fichier, 'r') as f:
            for line in f.readlines():
                line = line.replace(original,
remplacant)
                content += line

        with open(fichier, 'w') as f:
            f.write(content)
```

Dans cette fonction, on parcourt la liste `args` avec la boucle : `for fichier in args`. Dans cette boucle, on ouvre le fichier courant en mode lecture (`'r'` pour *read*) et on crée un alias `f` pour ce fichier. Pour chaque ligne parcourue, on remplace le mot contenu dans la variable `original` par celui contenu dans `remplacant`. La variable `content` contient ainsi les lignes modifiées.

À la sortie de la boucle, il ne reste plus qu'à enregistrer le fichier (`'w'` pour *write*) dont le contenu est celui de la variable `content`.

Chapitre 2

Programmation : généralités

Plan du chapitre

Exercice type 1

Le *problème de l'arrêt* est une démonstration par l'absurde. proposée par Alan Turing.
Supposons qu'il existe un algorithme H qui décide le problème de l'arrêt, c'est-à-dire que, quel que soit l'algorithme A qu'on lui passe en paramètre,

$$\begin{cases} H(A) \to 1 & \text{si } A \text{ s'arrête;} \\ H(A) \to 0 & \text{si } A \text{ ne s'arrête pas.} \end{cases}$$

Posons alors $\overline{H}$ l'algorithme admettant l'algorithme A en paramètre qui exécute $H(A)$ tel que :

$$\begin{cases} \text{si } H(A) = 1 \text{ alors une boucle infinie est créée;} \\ \text{si } H(A) = 0 \text{ alors il s'arrête.} \end{cases}$$

Montrer que le problème de l'arrêt est indécidable.

Voir corrigé page 44

1 Introduction

En informatique, la programmation est l'art de créer des programmes, qui sont des suites d'instructions ordonnées destinées à être exécutées par une machine numérique. Pour cela, il est indispensable d'utiliser des langages, qui permettent à la machine de comprendre l'humain. Ces trois notions (machines, programmation et langages) ont évolué de manière parallèle. Commençons par un peu d'histoire...

COURS

2 Historique

1842	La comtesse Ada Lovelace crée des diagrammes pour la machine analytique de Charles Babbage, mathématicien anglais. Ils sont aujourd'hui considérés comme les premiers programmes informatiques.
1936	Le concept de programmation et de programme enregistré est émis par Alan Turing, mathématicien anglais, dans un article intitulé « *On Computable Numbers, with an Application to the Entscheidungsproblem : Proceedings of the London Mathematical Society* ». Lors de la seconde guerre mondiale, Turing contribue à déchiffrer le code secret utilisé par les Allemands pour chiffrer leurs messages avec la machine *Enigma*.
Années 1940	Les premiers ordinateurs sont créés. Ils fonctionnent à l'aide de cartes perforées en carton.
1954	Développement du premier système d'exploitation au MIT (Massachusetts Institute of Technology). Apparition des premiers assembleurs et du premier compilateur pour le langage Fortran (Formula Translator), premier langage de programmation de haut niveau.
1964	le langage BASIC rend l'informatique accessible aux étudiants non scientifiques
1970	Arrivée de la programmation structurée, permettant aux programmeurs de concevoir des programmes plus poussés.
1972	Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, créent le langage C, permettant de développer le système d'exploitation multitâche multi-utilisateur UNIX.
1974	Vince Cerf et Bob Kahn inventent le protocole TCP/IP, qui servira de base à Internet
1981	arrivée de l'IBM-PC sous MS-DOS et développement de la micro-informatique en entreprise.
1983	Création du langage C++ et arrivée de la programmation orientée objet (POO).
1989	Le programmeur néerlandais Guido van Rossum crée un langage de script multi-plateforme en s'inspirant des langages ABC, C, et d'UNIX. Fan des Monty Python, il nomme son invention *Python*. La première version publique sort en 1991 sous licence libre.
1994	Apparition du langage HTML (Hypertext Markup Language). Associé au navigateur web Netscape et au protocole www (World Wide Web), HTML rend le réseau Internet accessible à tous.
2014	Trois milliards d'internautes utilisent quotidiennement l'informatique sur un milliard de sites en ligne.

3 Différents types de langages

Au plus bas niveau, le processeur ne comprend que le langage machine, qui est une suite d'opérations codées en binaire, incompréhensibles pour un être humain normalement constitué. C'est pourquoi des langages de plus haut niveau ont été inventés, mais il est nécessaire de les traduire en binaire. Il existe pour cela trois méthodes, qui correspondent à trois types de langages de programmation.

- les langages *compilés*, dans lesquels le code source écrit par le programmeur est transformé en code binaire par un logiciel appelé *compilateur*.
  Exemples : langages C, C++, Pascal et OCaml ;

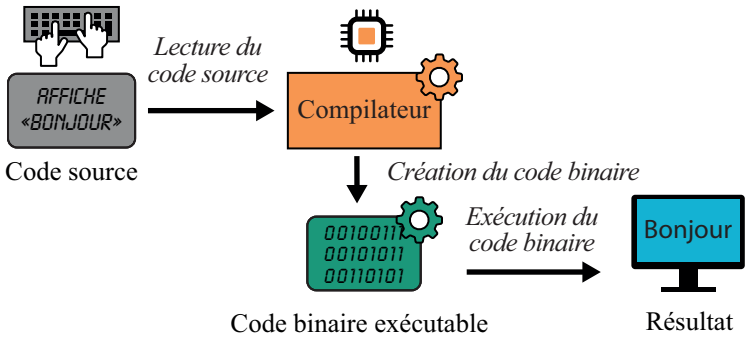


Figure 2.1 – Langage compilé

- les langages *interprétés*, dont le code source est utilisé en entrée d'un logiciel appelé *interpréteur*, qui l'exécute ligne par ligne en décidant à chaque étape ce qu'il va faire ensuite.
  Exemples : Python, JavaScript, Node JS, PHP

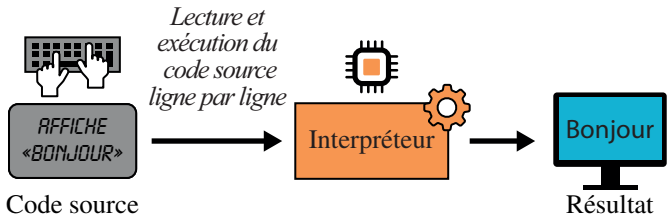


Figure 2.2 – Langage interprété

- les langages *semi-interprétés*, qui requièrent une compilation différente de celle vue précédemment, car elle ne transforme pas le code source en instructions binaires.
  Elle se contente de préparer le travail de l'interpréteur en réécrivant les instructions du code source dans un langage intermédiaire (bytecode), qui sera plus facile à traiter par l'interpréteur lors de son exécution.
  Exemple : Java, C#.

COURS

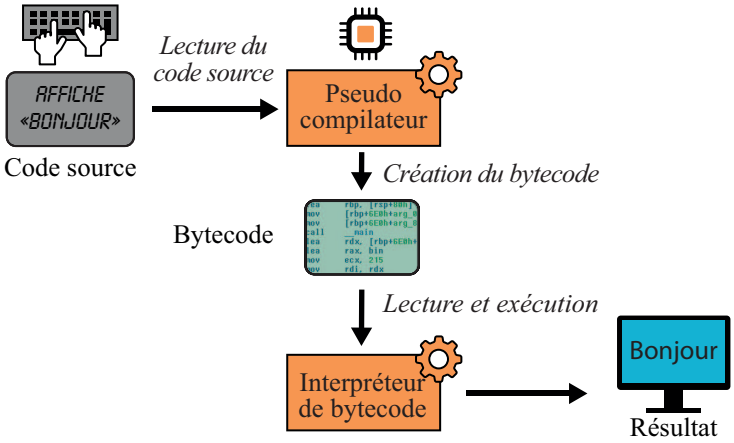


Figure 2.3 – Langage semi-interprété

4 Calculabilité et décidabilité

4.1 Calculabilité

La notion de calculabilité est apparue avec Turing en 1936. Pour l'appréhender, il est nécessaire d'en savoir un peu plus sur la machine de Turing.

4.1.1 - La machine de Turing

Retrouvez en vidéo une explication de la machine de Turing illustrée d'un exemple.

C'est un *modèle* abstrait de machine à calculer composé :

- d'un *ruban* composé de plusieurs cases dans lesquelles se trouve un élément de l'ensemble {0; 1} ; par exemple :

0	0	1	0	1	···

- d'une *tête de lecture* qui se trouve à chaque instant devant une des cases du ruban et qui peut être dans deux états possibles : 0 ou 1 ; par exemple :

0	0	1	0	1	···
		↑			
		0			

- d'une *table de transitions*, tableau contenant, comme son nom l'indique, des transitions (des transformations) de la forme :

$$(t, v) \to (t', v')$$

où t et t' sont deux états de la tête de lecture, et où v et v' sont deux valeurs contenues dans une case ;
par exemple, la table :

(0,0)	→	(0,1)
(0,1)	→	(1,1)

signifie que :

→ si la tête de lecture est dans l'état « 0 » et pointe sur une case contenant la valeur « 0 » alors la tête de lecture reste sur l'état « 0 » mais la valeur passe à « 1 » et la tête de lecture avance d'une case ;

→ si la tête de lecture est dans l'état « 0 » et pointe sur une case contenant la valeur « 1 » alors la tête de lecture passe à l'état « 1 » et la valeur reste à « 1 » et la tête de lecture avance d'une case ;

→ pour les autres cas, rien ne change : l'état de la tête de lecture reste à sa valeur et il en est de même pour la valeur de la case vers laquelle pointe la tête de lecture. La tête de lecture reste où elle est.

Exemple (incrémentation) : Voyons comment modéliser l'opération $n \to n+1$ pour un entier n à l'aide d'une machine de Turing.

L'entier n est représenté par un ruban contenant n « 1 » suivis de plusieurs « 0 » ; prenons l'exemple de $n = 2$. Initialement, la tête de lecture pointe sur la première case et est dans l'état « 0 ».

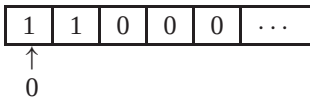


Figure 2.4 – Représentation du nombre « 2 » dans la machine de Turing

La table de transitions est la suivante :

(0,1)	→	(0,1)
(0,0)	→	(1,1)

Alors, comme la tête de lecture est initialement sur l'état « 0 » et que la valeur de la première case est « 1 », d'après la première transition, la tête de lecture restera à « 0 » et la valeur restera à « 1 ».

Une fois la transition prise en compte, la tête de lecture avance d'une case.

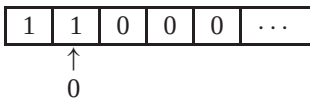


Figure 2.5 – Après la 1re transition

COURS

Là encore, la tête de lecture est à « 0 » et le nombre de la case vers laquelle elle pointe vaut « 1 », donc aucun changement n'est fait et la tête de lecture passe à la case suivante :

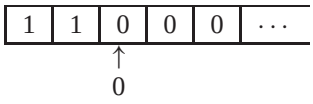


Figure 2.6 – Après la 2e transition

D'après la 2e ligne de la table de transitions, la tête de lecture passe à l'état « 1 » et la valeur de la case aussi, puis la tête de lecture avance d'une case :

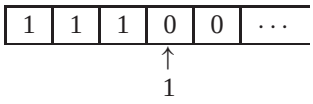


Figure 2.7 – Après la 3e transition

Comme aucune transition ne porte sur l'état (1,0), il n'y a plus de changement et la tête de lecture n'avance plus.

On aboutit ainsi à la valeur « 3 » (car il y a 3 « 1 »), ce qui correspond bien à l'incrémentation de 2.

4.1.2 - Thèse de Church-Turing et calculabilité

Cette thèse, qui n'est pas un théorème puisqu'elle n'a pas été démontrée, stipule que :

Si un problème est tel qu'il existe un algorithme le résolvant, alors il existe une machine de Turing qui résout le problème.

Cette thèse mène alors à la notion de calculabilité.

Définition 1

On dira qu'un problème est *calculable* s'il existe une machine de Turing le résolvant.

On peut ainsi définir un algorithme :

Définition 2

Un *algorithme* est une table de transitions d'une machine de Turing.

4.1.3 - La machine universelle

Généralisons la notion de machine de Turing et, pour bien comprendre, reprenons l'exemple de l'incrémentation.

Nous avons vu précédemment que nous avons passé en argument à la machine de Turing le nombre *n* afin d'obtenir son suivant. Appelons `Inc(n)` cette machine de Turing. `Inc` est définie par sa table de transitions, que l'on peut représenter simplement par la succession de binaires : « 01010011 ».

(0,1)	→	(0,1)
(0,0)	→	(1,1)

devient « 01010011 »

L'idée permettant de construire la machine *U* (la machine universelle) est de passer d'abord en argument la table de transitions sous forme binaire, puis le nombre *n* (pour notre exemple).

La machine universelle constitue les principes fondamentaux des ordinateurs d'aujourd'hui : on peut considérer un programme comme étant une table de transitions que l'on passe en paramètre.

Remarque : tous les problèmes ne sont pas calculables.

Voir exercice 12 page 56.

4.2 Décidabilité

Définition 3

Un problème est *décidable* si et seulement si il existe un algorithme qui le résout.

Cela sous-entend que l'algorithme se termine en un nombre fini d'opérations.

Exemple : un entier donné est-il un carré parfait ?

On peut aisément répondre à ce problème à l'aide de l'algorithme suivant :

```
n ← 1
x : nombre entier
Tant que (n² != x) et (n < √x):
   n ← n + 1
Retourner (n² == x)
```

COURS

Le code Python correspondant s'écrit :

```
def is_square(x):
    n = 1
    while (n**2 != x) and (n < x**0.5):
        n += 1
    return (n**2 == x)

print(is_square(64))
```

Le problème est donc décidable.

Alan Turing a démontré en 1936 que tous les problèmes n'étaient pas décidables à l'aide d'un contre-exemple : le *problème de l'arrêt*. C'est l'objet de l'exercice type proposé au début de ce chapitre.

Solution de l'exercice type 1

Nous avons posé $\overline{H}$ l'algorithme admettant l'algorithme A en paramètre qui exécute $H(A)$ tel que :

$$\begin{cases} \text{si } H(A) = 1 \text{ alors une boucle infinie est créée ;} \\ \text{si } H(A) = 0 \text{ alors il s'arrête.} \end{cases}$$

$\overline{H}$ existe si H existe.

Considérons alors $\overline{H}\left(\overline{H}\right)$.

- Si $\overline{H}$ s'arrête alors $H\left(\overline{H}\right) = 1$ et donc $\overline{H}\left(\overline{H}\right)$ crée une boucle infinie, donc ne s'arrête pas.
- Si $\overline{H}$ ne s'arrête pas alors $H\left(\overline{H}\right) = 0$ et donc $\overline{H}\left(\overline{H}\right)$ s'arrête.

Dans les deux cas, il y a contradiction. Ainsi, $\overline{H}$ n'existe pas, et donc H non plus.

Le problème de l'arrêt est donc indécidable.

Voir énoncé page 37

Voir exercice 12 page 56.

5 Récursivité

Exercice type 2

Implémenter en Python une fonction récursive `somme(n)` qui renvoie la valeur de $S_n = 1+2+3+\cdots+(n-1)+n$, en remarquant que $S_n = S_{n-1}+n$.

Voir corrigé page 49

Retrouvez une vidéo sur la récursivité avec Nathan Live.

5.1 Définition

Définition 4

Une fonction *récursive* est une fonction qui fait appel à elle-même.

Remarque : on parle aussi d'*algorithmes récursifs*.

Exemple (calcul d'une factorielle) : en mathématiques, on définit la factorielle d'un nombre entier n par :

$$n! = 1 \times 2 \times 3 \times 4 \times \cdots \times n.$$

Ainsi, si $f(n) = n!$ alors $f(n+1) = (n+1) \times f(n)$. On peut alors définir la fonction `factorielle` de façon récursive.

En Python, cela donne par exemple le code suivant.

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

En tapant :

```
print(factorielle(5))
```

le résultat « 120 » s'affiche, qui est bien égal à $5! = 1 \times 2 \times 3 \times 4 \times 5$.

Un autre exemple est celui du calcul du pgcd de deux nombres (voir exercice 4 page 51).

Remarque : le langage Python limite le nombre d'appels récursifs. Pour savoir le nombre maximal autorisé, on pourra taper :

```
import sys
print(sys.getrecursionlimit())
```

COURS

Par exemple, `factorielle(1000)` va afficher l'erreur :

```
RecursionError: maximum recursion depth exceeded in
comparison
```

Pour augmenter ce nombre limite, on pourra utiliser la syntaxe suivante :

```
sys.setrecursionlimit(1500)
```

On peut alors calculer `factorielle(1000)`, mais dans la mesure où le résultat comporte 2 568 (= `len(str(factorielle(1000)))`) chiffres, il ne figurera pas dans ce livre...

5.2 Complexité d'un programme récursif

Propriété 1

Un programme récursif a une complexité linéaire ou exponentielle.

Exemple : reprenons l'exemple précédent (algorithme récursif de la factorielle). Notons $C(n)$ la complexité de la fonction pour la variable entière n.

- Si $n = 0$, seul le test `n == 0` est effectué donc $C(0) = 1$;
- si $n > 0$, le test et le produit sont pris en compte en plus des opérations élémentaires de la fonction pour $n - 1$, donc $C(n) = 2 + C(n - 1)$.

On peut donc dire que la complexité de cette fonction est la suite (c_n) définie par $c_0 = 1$ et $c_{n+1} = c_n + 2$.

Dans ce cas, la complexité est *linéaire*, et on notera

$$C(n) = O(n).$$

C'est la *notation de Landau*.

5.3 Correction d'un algorithme récursif

Pour qu'un algorithme récursif fonctionne correctement, il est nécessaire de vérifier deux propriétés :

- l'algorithme doit se terminer (c'est ce que l'on appelle le *problème de la terminaison*) ;
- si l'algorithme se termine, il faut qu'il donne ce que l'on souhaite (c'est ce que l'on nomme la *correction partielle*).

5.3.1 - Terminaison

Regardons sur l'exemple de la fonction récursive `factorielle` vue précédemment comment vérifier sa terminaison.

La fonction admet un argument entier positif.

De plus,

- `factorielle(n)` fait appel à `factorielle(n-1)` ;
- `factorielle(n-1)` fait appel à `factorielle(n-2)` ;
- etc.
- `factorielle(1)` fait appel à `factorielle(0)` ;
- `factorielle(0)` renvoie la valeur « 1 ».

On est donc assurés que cette fonction se termine.

5.3.2 - Correction partielle

Définition 5

Dans une fonction récursive, un *appel interne* est un appel de la fonction déclenchée par elle-même (un appel *récursif*).

Pour prouver la correction partielle d'un algorithme (d'une fonction) récursif(ve), il faut montrer que si les appels internes à l'algorithme (ou la fonction) font ce qu'on attend d'eux, alors l'algorithme (la fonction) fait ce qu'on attend de lui (elle).

Par exemple, pour la fonction `factorielle`, si on part du principe que `factorielle(n-1)` représente $(n-1)!$ alors `n * factorielle(n-1)` représente bien $n \times (n-1)! = n!$.

Cela ressemble au *raisonnement par récurrence* vu en mathématiques.

Voir exercices 2 à 11 pages 51 à 54.

6 Modularité

6.1 Principe

Quand on doit élaborer un programme complexe, il est primordial de le penser en terme de modularité. Cela signifie qu'il est nécessaire de le découper en un maximum de blocs afin de le rendre plus lisible et fonctionnel.

Concevoir un programme avec plusieurs fonctions permet de faire appel à ces fonctions à plusieurs endroits dans le programme.

Les fonctions les plus utilisées pourront être placées dans des fichiers auxiliaires, autrement appelés *modules* ou *bibliothèques* (en Python par exemple).

Exemple : imaginons un module nommé `bienpratique` renfermant plusieurs fonctions (voir page ci-contre). On peut alors écrire un programme principal faisant appel à ce module, c'est-à-dire le programme qui contient l'ensemble des instructions qui seront directement exécutées à son lancement.

COURS

Remarque : quand on écrit un module, il est nécessaire de le documenter au mieux afin que tout.e. lecteur.trice. puisse s'y retrouver et comprendre les blocs de code.

Module : bienpratique.py

```
"""
Fonction : is_square(entier n)
But : teste si la variable est un carré parfait
"""

def is_square(x):
    n = 1
    while (n ** 2 != x) and (n < x ** 0.5):
        n += 1
    return (n ** 2 == x)

"""
Fonction : is_odd(entier n)
But : teste si la variable est impaire
"""

def is_odd(n):
    if (n % 2 == 0):
        return False
    else:
        return True
```

Programme principal

```
from bienpratique import *

n = int(input("Entrez un nombre entier : "))

if is_square(n):
    print("C'est un carré parfait !")

if is_odd(n):
    print("C'est un nombre impair.")
else:
    print("C'est un nombre pair.")
```

6.2 Ne pas réinventer la roue

Il existe déjà de nombreux modules libres et gratuits, fournis par Python, des entreprises et des particuliers. Il n'est donc pas nécessaire de créer des fonctions qui existent déjà.

Par exemple,

- le module `random` contient des fonctions qui ont un rapport avec la génération de nombre pseudo-aléatoires.
  Documentation : `https://docs.python.org/fr/3/library/random.html`.
- le module `numpy` concerne les calculs scientifiques.
  Documentation : `https://www.numpy.org`.
- le module `sympy` concerne les mathématiques symboliques (le calcul algébrique par exemple).
  Documentation : `https://www.sympy.org/en/index.html`.
- le module `pandas` concerne les statistiques et les manipulations de données.
  Documentation : `https://pandas.pydata.org`.
- le module `os` fournit des fonctionnalités dépendantes du système d'exploitation.
  Documentation : `https://docs.python.org/fr/3/library/os.html`.

Vous trouverez sur la page suivante un grand nombre de modules de Python 3.8 classés par catégories :

https ://docs.python.org/fr/3.8/library/index.html

qui en donne un certain nombre (mais pas tous).

Solution de l'exercice type 2

Comme dans toute fonction récursive, il faut penser à la terminaison, sans quoi le programme risque de ne jamais s'arrêter.
Ici, le premier terme de la somme est 1 donc il nous faut renvoyer « 1 » quand $n = 1$.
De plus, si on note $S_n = 1 + 2 + \cdots + n$, on voit que $S_n = S_{n-1} + n$, ce qui nous permet de dire que si $n \neq 1$ alors on doit retourner la somme de `somme(n-1)` et `n`.
On en déduit la fonction suivante :

```
def somme(n):
    if n == 1:
        return 1
    return n + somme(n-1)

print(somme(100))
```

Voir énoncé page 45

INTERROS

1 QCM Vérification de connaissances

5 min

Corrigé p. 58

1 Un ordinateur comprend mieux :

- **a** des instructions en anglais ;
- **b** des instructions en langage binaire (0 et 1) ;
- **c** des instructions exprimées en langage d'ordinateur (Fortran, Python,...)

2 Lorsque l'on rédige un programme pour un ordinateur, on utilise :

- **a** un langage informatique comme C, Basic, PHP,... ;
- **b** le langage binaire (0 et 1) obligatoirement ;
- **c** de préférence l'anglais.

3 Si on veut pouvoir modifier aisément un programme, il vaut mieux disposer :

- **a** du programme binaire ;
- **b** du programme source ;
- **c** du programme rédigé en anglais ou en français ;
- **d** la question n'a pas de sens : c'est impossible de modifier un programme.

4 La compilation, c'est :

- **a** la transformation du texte du programme binaire vers un langage informatique ;
- **b** la transformation du programme rédigé en anglais vers le français ;
- **c** l'ajout bout à bout de différents programmes ;
- **d** la transformation d'un programme source, écrit dans un langage informatique, en langage binaire ou en langage intermédiaire.

5 La compilation d'un programme doit se faire :

- **a** avant chaque utilisation du programme ;
- **b** régulièrement pendant le fonctionnement du programme ;
- **c** à chaque modification du code source ;
- **d** jamais.

6 Si je veux diffuser un programme, mais que je ne souhaite pas que les utilisateurs sachent comment il fonctionne,

- **a** je dois distribuer le code binaire uniquement ;
- **b** je dois distribuer le code source uniquement ;
- **c** il n'y a aucun moyen d'empêcher les utilisateurs de comprendre comment le programme fonctionne.

7 Si je veux diffuser un programme et que je souhaite que les utilisateurs puissent l'améliorer...

- **a** je dois distribuer le code binaire uniquement ;
- **b** je dois diffuser le code source du programme ;
- **c** il n'y a aucun moyen de permettre aux utilisateurs d'améliorer un programme.

8 Un code source en langage interprété :

a peut être directement exécuté par le noyau et le processeur ;

b ne peut pas être directement exécuté par le noyau et le processeur.

2 Fonction mystère

★ — 5 min — Corrigé p. 58

Que fait la fonction suivante ?

```
def mystere(L , M = [ ]):
    # L est une liste
    if not( L ):
        return M
    a = L.pop(0)
    if a not in M:
        M.append( a )
    return mystere(L , M)
```

3 Le retour de la fonction mystère

★ — 5 min — Corrigé p. 58

Que fait la fonction suivante :

```
def mystere(L):
    if len(L) == 1 :
        return L[0]
    if L[0] < L[1]:
        L.pop(1)
    else:
        L.pop(0)

    return mystere(L)
```

4 Calcul du pgcd de deux nombres

★ — 10 min — Corrigé p. 59

Écrire en Python une fonction récursive `pgcd(a,b)` renvoyant le plus grand diviseur commun de deux nombres a et b.

Pour cela, on utilisera le résultat mathématique suivant :

« $\text{pgcd}(a,b) = \text{pgcd}(b,r)$, où $a = bq + r$. »

5 Nombre d'adhérents

★ — 10 min — Corrigé p. 59

Une association a remarqué que d'une année à l'autre,

- elle perd 5 % de ses adhérents ;
- elle gagne 200 adhérents.

INTERROS

1. En admettant que le nombre d'adhérents de cette association était égal à 2 000 au 1er janvier 2019, écrire en Python une fonction récursive nommée `nombre(n)` affichant le nombre théorique d'adhérents après n années, $n \geqslant 1$.
2. Dans ce même programme, afficher le nombre théorique d'adhérents au cours des 20 prochaines années.

6 Suite de Fibonacci

★ — 10 min — Corrigé p. 60

La suite de Fibonacci est la suite numérique définie par :

$$F_0 = F_1 = 1 \quad , \quad \forall n \geqslant 0,\ F_{n+2} = F_{n+1} + F_n.$$

Écrire en Python une fonction récursive `Fibo(n)` permettant d'afficher le terme F_n, où $n \geqslant 0$, puis afficher les termes de F_0 à F_{20}.

7 La fonction compteur

★★ — 10 min — Corrigé p. 60

Écrire une fonction récursive `compteur(chaine,caractere)` retournant le nombre de fois que l'on peut compter le caractère dans la chaîne.

Par exemple, `compteur('Bonjour Lucie','u')` renvoie le nombre 2 car il y a deux « u » dans la chaîne « Bonjour Lucie ».

Remarque : la méthode `count` permet d'obtenir ce résultat, mais le but de cet exercice n'est pas de se servir de cette méthode. On peut en effet avoir le nombre d'occurrences du caractère « p » dans une chaîne en tapant `chaine.count('p')`.

8 La vengeance de la fonction mystère

★★ — 10 min — Corrigé p. 61

On considère le programme suivant :

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

print( f(3,10) )
```

1. Quel résultat retourne `f(7,9)` ? (exécuter le programme à la main)
2. En déduire la signification de la valeur retournée par cette fonction pour deux entiers naturels non nuls a et b quelconques.

3 Pour une fonction récursive, un *cas de base* est un cas qui ne nécessite pas d'appel récursif à la fonction.
Quel est le cas de base de cette fonction ?

4 Qu'est-ce qui garantit que le programme s'arrête ?

5 La complexité de cette fonction est-elle linéaire ? Quadratique ? Exponentielle ?

9 Diagonale de Cantor

★★ — 15 min — Corrigé p. 62

Dans un repère, les points à coordonnées entières sont numérotés selon le principe suivant :

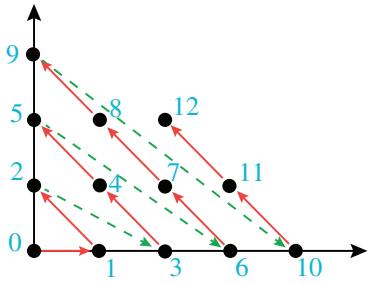


Figure 2.8 – Diagonale de Cantor

Écrire une fonction récursive `numero(x,y)` qui retourne le numéro du point de coordonnés $(x ; y)$.

Par exemple, `numero(3,2)` doit retourner le nombre « 17 ».

10 Les tours de Hanoï

★★★ — 20 min — Corrigé p. 62

Selon Wikipedia :

> Les tours de Hanoï (originellement, la tour d'Hanoï) sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles page suivante.

> - on ne peut déplacer plus d'un disque à la fois ;
> - on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.
>
> On suppose que cette dernière règle est également respectée dans la configuration de départ.

Ainsi, en notant n le nombre de disques, la configuration de départ pour $n = 3$ est la suivante :

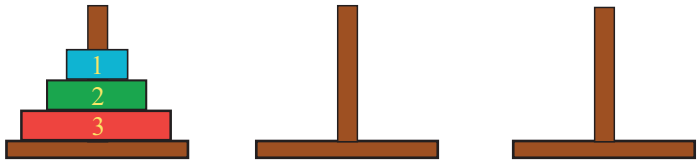


Figure 2.9 – Configuration initiale des tours de Hanoï

Pour résoudre le jeu, il faut déplacer tous les disques sur la Tour 3 en se servant de la tour 2 comme intermédiaire, ceci en commençant par déplacer le disque bleu sur la tour 3, puis le disque vert sur la tour 2, puis on peut mettre le disque bleu sur le disque vert, etc.

1. Pour $n = 3$, résoudre « à la main » le jeu. On notera (comme sur la figure 2.9) les disques avec les chiffres 1, 2 et 3 et on décrira les déplacements (par exemple, 1 va sur T3, 2 va sur T2, etc.)
2. Pour n quelconque, ramener le problème initial à un même problème en faisant intervenir les $n - 1$ premiers disques d'une part et le n-ième disque d'autre part.
3. En déduire une fonction récursive `hanoi(n,start=1,end=3,aux=2)` permettant d'afficher tous les mouvements jusqu'à résolution complète du jeu, où `start` désigne le numéro de la tour d'où provient un disque, `end` celui de la tour vers laquelle on déplace le disque et `aux` celui de la tour auxiliaire.

11 Grille de Sudoku

★★★ 50 min Corrigé p. 64

Un *Sudoku* classique est une grille de 9 lignes et 9 colonnes, donc composée de 9 blocs distincts de 3 lignes et 3 colonnes.

Remplir une grille de Sudoku consiste à utiliser tous les chiffres de 1 à 9 pour chacun des 9 blocs de sorte que chaque ligne et chaque colonne de la grille totale ne comporte aucun chiffre en double.

Dans cet exercice, on décidera de représenter une telle grille par une liste `S` de neuf listes à neuf éléments, chacune des listes représentant une ligne.

L'objectif de cet exercice est d'obtenir le remplissage de la grille page suivante (figure 2.10).

1. Compléter la variable :`S = [ [0,1,0,0,7,8,0,0,0] , ... ]`
2. Écrire une fonction `ligne(S,i)` qui retourne la liste des chiffres de 1 à 9 qui apparaissent sur la ligne i. On devra avoir pour notre grille :

```
>>> ligne(S,0)
[1,7,8]
```

	1			7	8			
	8			4		9		
		5	6				1	
1				6				5
	4		9	1	5		7	2
	6	7		8		4		
			3			1		
	7		8	9			2	3
					4			

Figure 2.10 – Grille de Sudoku à implémenter

3 Écrire une fonction `colonne(S,j)` qui retourne la liste des chiffres de 1 à 9 qui apparaissent dans la colonne j. On devra avoir pour notre grille :

```
>>> colonnes(S,0)
[1]
```

4 Écrire une fonction `bloc(S,i,j)` qui retourne la liste des chiffres de 1 à 9 qui apparaissent sur le bloc 3×3 auquel appartient la case de la ligne i et de la colonne j. On devra avoir pour notre grille :

```
>>> bloc(S,3,4)
[6,9,1,5,8]
```

5 Écrire alors une fonction `possibles(S,i,j)` qui retourne la liste des chiffres de 1 à 9 que l'on peut écrire dans la case de la ligne `i` et de la colonne `j` (en tenant compte des règles du jeu). On devra avoir pour notre grille :

```
>>> possibles(S,0,0)
[2,3,4,6,9]
```

6 On complète la grille de la ligne 0 à la ligne 8, de la colonne 0 à la colonne 8.
Écrire une fonction `suivante(i,j)` qui reçoit en paramètre la ligne `i` et la colonne `j` d'une case, et qui renvoie le tuple (ligne,colonne) de la case suivante. On devra avoir :

```
>>> suivante(0,0) , suivante(0,8) , suivante(8,8)
(0,1) , (1,0) , (9,0)
```

7 Compléter la fonction principale de résolution suivante :

```
def solve(S,i,j):
    if i == 9:
        return ...
    elif S[i][j] > 0:
        ...
        return ...
    for k in possibles(S,i,j):
        S[i][j] = k
        ...
        if solve(S,a,b):
            return ...
    S[i][j] = 0
    return False
```

12 Preuve : incalculabilité

★★★ 30 min Corrigé p. 66

Cet exercice a pour but de démontrer que tous les problèmes ne sont pas calculables.

Pour cela, nous aurons besoin des définitions suivantes :

- une *bijection* de $\mathbb{N}$ vers l'ensemble E est une fonction f telle que :
  - → quel que soit l'entier naturel n, il existe une unique image de n par f ;
  - → quel que soit l'élément e de E, il existe un unique antécédent dans $\mathbb{N}$ à e.

  La fonction $n \mapsto 2n$ est par exemple une bijection de $\mathbb{N}$ vers l'ensemble des entiers positifs pairs ;
- un ensemble E est *dénombrable* s'il existe une bijection de $\mathbb{N}$ vers E.

1 Un algorithme est une succession de caractères pris dans un alphabet fini, noté A. Montrer que l'ensemble $\mathcal{A}$ des algorithmes est dénombrable.

Aide : on pourra se servir de l'ensemble C des combinaisons des caractères de $\mathcal{A}$.

2 Un problème de décision peut être représenté par une fonction booléenne $f : n \to \{0; 1\}$ (« 0 » pour le cas où la propriété est fausse pour la variable et « 1 » dans le cas contraire).

Par exemple, le problème « Est-ce que l'entier est pair ? » se représente par la fonction $f : 2p \mapsto 1$ et $f : 2p + 1 \mapsto 0$, pour tout entier naturel p.

On note $\mathcal{B}$ l'ensemble des fonctions booléennes.

On suppose que $\mathcal{B}$ est dénombrable, c'est-à-dire qu'il existe des fonctions f_1, f_2, ... qui peuvent être représentées par exemple comme page ci-contre.

Valeurs de n	0	1	2	3	$\cdots$
f_0	1	0	0	1	$\cdots$
f_1	1	1	0	0	$\cdots$
f_2	0	1	0	0	$\cdots$
$\vdots$					

On pose alors g telle que :

$$\begin{array}{rccl} g & : & \mathbb{N} & \to & \{0;1\} \\ & & k & \mapsto & 1 - f_k(k) \end{array}$$

(a) Calculer $g(0)$, $g(1)$ et $g(2)$ pour les fonctions f_0, f_1 et f_2 données en exemple dans l'énoncé.

(b) Dans un cas général, montrer que, quel que soit l'entier naturel k, $g \neq f_k$.

(c) Justifier que $g \in \mathcal{B}$ et conclure à une contradiction.
Que cela permet-il de conclure sur $\mathcal{B}$?

3 Justifier alors que tous les problèmes ne sont pas dénombrables.

CORRIGÉS

1 QCM Vérification de connaissances

Enoncé p. 50

1. Réponse **b**. Un ordinateur ne sait interpréter que le langage binaire.
2. Réponse **a**. Les langages informatiques permettent d'obtenir (compilation) des programmes binaires (ou intermédiaires) qui sont compris par les ordinateurs (ou par un programme interpréteur).
3. Réponse **b**. Il est beaucoup plus facile pour un humain de comprendre et de modifier un code source qu'un programme binaire.
4. Réponse **d**. La compilation est précisément la traduction du programme source en programme binaire (ou intermédiaire).
5. Réponse **c**. Quand on modifie le code source d'un programme, il faut le compiler à nouveau pour remplacer le programme binaire déjà existant afin de prendre en compte les dernières modifications.
6. Réponse **a**. En ne disposant que du code binaire d'un programme, il est humainement difficile de le comprendre et de le modifier.
7. Réponse **b**. Un programme peut être modifié en ayant son code source.
8. Réponse **b**. Un code source en langage interprété ne peut pas être directement exécuté par le noyau et le processeur. C'est le cas de Python (contrairement au C).

2 Fonction mystère

Enoncé p. 51

La première instruction de la fonction consiste à tester si la liste `L` est absente, auquel cas la fonction retourne `M`, c'est-à-dire une liste vide.

Ensuite, on définit `a` comme étant le premier élément de `L` tout en enlevant cet élément de `L` (la méthode `pop(i)` supprime l'élément d'indice *i*).

Si `a` n'est pas dans la liste `M` alors on l'y ajoute.

En écrivant `return mystere(L,M)`, on exécute à nouveau la fonction `mystere` avec pour arguments la nouvelle liste `L` (donc sans l'élément `a`) et `M` (qui contient alors `a`).

Au final, on obtient une liste `M` des éléments distincts de `L`.

La fonction retourne donc la liste des éléments distincts d'une autre liste.

3 Le retour de la fonction mystère

Enoncé p. 51

La fonction commence par tester si la liste a un seul élément, auquel cas elle retourne cet élément. Cela nous donne donc un indice sur ce qu'elle fait : le retour est un nombre.

Ensuite, on compare les deux premiers éléments de la liste : si le premier est plus petit que le deuxième alors on supprime le deuxième sinon on supprime le premier ; on ne garde donc que le plus petit des deux premiers éléments et on écourte la liste d'un élément (le plus grand des deux premiers).

On termine par faire appel à la fonction elle-même.

Il s'agit donc ici de diminuer la liste initiale en ne conservant que le plus petit des deux premiers éléments à chaque fois. L'élément restant au final est donc le plus petit élément de la liste initiale.

4 Calcul du pgcd de deux nombres

Enoncé p. 51

Une fonction récursive permettant de calculer et d'afficher le pgcd de deux nombres a et b est la suivante :

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

Rappelons que l'opérateur « % » permet d'obtenir le reste d'une division euclidienne. Ainsi, « `a % b` » représente le reste de la division euclidienne de `a` par `b`.

5 Nombre d'adhérents

Enoncé p. 51

1 La fonction récursive calculant et donnant le nombre théorique d'adhérents est :

```
def nombre(n):
    if n == 0:
        u = 2000
    else:
        u = nombre(n-1)*0.95+200
    return u
```

En effet, on peut noter u_n le nombre d'adhérents lors de l'année $2019+n$. Ainsi, $u_0 = 2\,000$ et $u_{n+1} = 0{,}95u_n + 200$ (une perte de 5 % revient à conserver 95 % de la valeur précédente). Cette dernière égalité nous donne la formule récursive à mettre dans la fonction.

2 Pour afficher les 20 valeurs qui suivent la première, on utilise une boucle itérative :

```
for i in range(1,21):
    print(nombre(i))
```

6 Suite de Fibonacci

Enoncé p. 52

Voici une suggestion de fonction :

```
def fibo(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)

for i in range(21):
    print(fibo(i))
```

En effet, la relation $\forall n \geqslant 0,\ F_{n+2} = F_{n+1} + F_n$ peut aussi s'écrire :

$$\forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}\ ,$$

d'où la formule : « `f = fibo(n-1)+fibo(n-2)` ».

Cependant, ce programme (naïf) n'est pas optimal. En effet, des mêmes termes sont calculés plusieurs fois, ce qui rend le programme quelque peu inefficace.

Nous verrons dans le chapitre suivant une autre solution utilisant la *programmation dynamique* (voir exercice 15 page 113).

7 La fonction compteur

Enoncé p. 52

Une première possibilité est la suivante :

```
def compteur(chaine , caractere):
    if chaine == '':
        return 0
    if chaine[0] ==  caractere:
        n = 1
    else:
        n = 0
    return n + compteur(chaine[1:] , caractere)
```

L'idée consiste ici à comparer le premier caractère de la chaîne au caractère recherché ; s'ils coïncident alors on ajoute 1 au nombre d'occurrences du caractère dans la chaîne obtenue à partir de la chaîne principale en ayant enlevé le premier caractère. Sinon, on n'ajoute rien.

Une autre possibilité, bien moins naturelle, est la proposition suivante :

```
def compteur(chaine , caractere):
    L = list(chaine)
    if caractere in L:
        L.pop( L.index(caractere) )
        return 1 + compteur( str(L) , caractere)
    return 0
```

MÉTHODE

On transforme d'abord la chaîne de caractères en une liste.

Si le caractère est trouvé dans les éléments de cette nouvelle liste alors on l'enlève (avec la méthode `pop`) et on ajoute 1 au compteur qui correspond à la chaîne initiale privée du caractère enlevé (la liste a été transformée en chaîne avec la méthode `str`).

Enfin, il ne faut pas oublier de retourner « 0 » si le caractère n'est pas présent... sinon, un magnifique message d'erreur apparaît !

Mais cette façon de voir est légèrement tirée par les cheveux...

8 La vengeance de la fonction mystère

Enoncé p. 52

1 La fonction calcule :

$$\begin{aligned} 7 + f(7,8) &= 7 + 7 + f(7,7) \\ &= 2 \times 7 + 7 + f(7,6) \\ &= 3 \times 7 + 7 + f(7,5) \\ &= 4 \times 7 + 7 + f(7,4) \\ &= 5 \times 7 + 7 + f(7,3) \\ &= 6 \times 7 + 7 + f(7,2) \\ &= 7 \times 7 + 7 + f(7,1) \\ &= 8 \times 7 + 7 = 9 \times 7 = 63. \end{aligned}$$

2 De l'exemple précédent, on peut remarquer que la fonction retourne $a \times b$.

3 Le cas de base de cette fonction est « $b = 1$ » car dans ce cas, on retourne une valeur (ici, la valeur de a).

4 Le fait d'appeler `f(a,b-1)` et le fait que b soit un entier naturel non nul nous assure que la valeur de b diminue de 1 en 1 et qu'elle va atteindre la valeur « 1 », auquel cas le programme s'arrête.

5 Dans la mesure où il n'y a pas de multiplication, la complexité est ici linéaire : elle est en $O(a)$.

CORRIGÉS

9 Diagonale de Cantor

Enoncé p. 53

Une fonction possible est la suivante :

```
def numero(x,y):
    if (x,y) == (0,0):
        return 0
    elif y == 0:
        return 1 + numero(0,x-1)
    else:
        return 1 + numero(x+1,y-1)
```

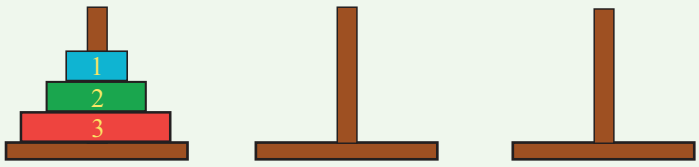
L'idée ici est de partir du point de coordonnées (x,y), qui initialise notre compteur à 1, et d'ajouter `numero(x+1,y-1)` tant que l'on ne se trouve pas sur l'axe des abscisses. Si l'ordonnée est nulle, il faut en effet revenir à une abscisse nulle et à une ordonnée qui est égale à l'abscisse du point duquel nous sommes partis diminuée de 1 : `numero(0,x-1)`.

On s'arrête pour le point de cordonnées (0,0). À chaque fois que l'on fait appel à la fonction récursive, on ajoute 1 (car un point est compté en plus).

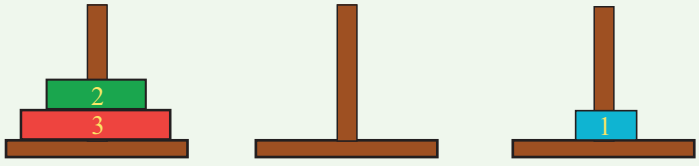
10 Les tours de Hanoï

Enoncé p. 53

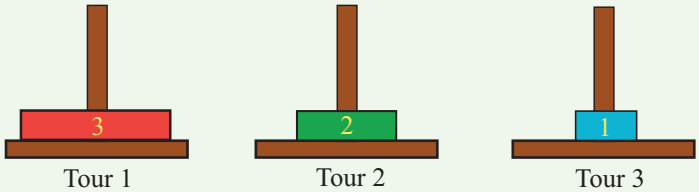
1 Déplacement 1 :



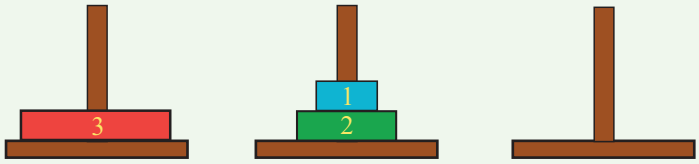
Déplacement 2 :



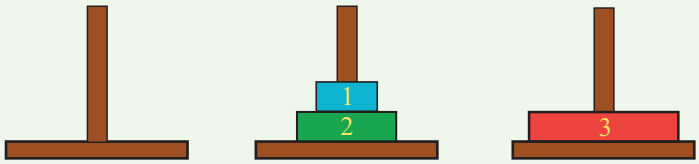
Déplacement 3 :



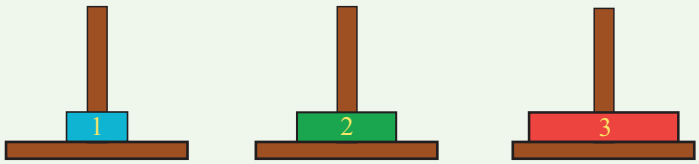
Déplacement 4 :



Déplacement 5 :



Déplacement 6 :



2 Afin de résoudre le jeu, il faut qu'à l'avant-dernier déplacement, tous les $n-1$ plus petits disques soient empilés sur la tour 2 afin de pouvoir déplacer le disque n vers la tour 3.

Déplacer n disques de T1 vers T3 (en utilisant T2 comme intermédiaire) revient à :

- déplacer $n-1$ disques de T1 à T2 (en utilisant T3 comme intermédiaire) ;
- déplacer le n-ième disque de T1 vers T3 ;
- déplacer à nouveau les $n-1$ disques de T2 vers T3 (en utilisant T1 comme intermédiaire).

3 Dans la fonction récursive, il faut une condition d'arrêt (le cas de base) : ce sera le cas où $n = 1$, auquel cas on affichera : « Déplacement du disque de la tour X vers la tour 3 », qui marquera presque la fin de la récursivité (il faudra encore déplacer à nouveau les $n-1$ disques situés sur T2).

Si $n \neq 1$ alors le problème revient à déplacer $n-1$ disques vers la tour auxiliaire, donc on devra faire appel à `hanoi(n-1,start,aux,end)`, puis on devra déplacer les $n-1$ disques de l'auxiliaire vers la dernière tour, donc faire appel à `hanoi(n-1,aux,end,start)`.

```
def hanoi(n , start = 1 , end = 3 , aux = 2):
    if n == 1:
        print('T',start,' --> T',end)
    else:
        hanoi(n-1 , start , aux , end)
        print('T',start,' --> T',end)
        hanoi(n-1 , aux , end , start)
```

```
>>> hanoi(3)
T 1 -> T 3
T 1 -> T 2
T 3 -> T 2
T 1 -> T 3
T 2 -> T 1
T 2 -> T 3
T 1 -> T 3
```

11 Grille de Sudoku

Enoncé p. 54

1

```
S = [
        [0,1,0,0,7,8,0,0,0] ,
        [0,8,0,0,4,0,9,0,0] ,
        [0,0,5,6,0,0,0,1,0] ,
        [1,0,0,0,6,0,0,0,5] ,
        [0,4,0,9,1,5,0,7,2] ,
        [0,6,7,0,8,0,4,0,0] ,
        [0,0,0,3,0,0,1,0,0] ,
        [0,7,0,8,9,0,0,2,3] ,
        [0,0,0,0,0,4,0,0,0]
]
```

2

```
def ligne(S,i):
    return [S[i][j] for j in range(0,9) if S[i][j]
    != 0 ]
```

3

```
def colonne(S,j):
    return [S[i][j] for i in range(0,9) if S[i][j]
    != 0 ]
```

4

```
def bloc(S,i,j):
    r = []
    ligne = (i // 3) * 3
    colonne = (j // 3) * 3
    for l in  [ligne,ligne+1,ligne+2]:
        for c in [colonne,colonne+1,colonne+2]:
            if S[l][c] != 0:
                r.append(S[l][c])
    return r
```

Ici, la difficulté réside dans le choix de la méthode que l'on souhaite adopter. Dans la proposition faite, nous avons fait le choix de calculer les indices de la case haut-gauche du bloc auquel appartient la case (i, j). Pour se faire, on calcule le quotient de i par 3 (si $i = 4$ alors le quotient est 1) et on le multiplie par 3 car les lignes des cases haut-gauche sont toutes 0, 3 ou 6.
Il en est de même pour les colonnes.

5

```
def possibles(S,i,j):
    return [k for k in range(1,10)
            if k not in bloc(S,i,j) and
            k not in ligne(S,i) and
            k not in colonne(S,j)]
```

Il s'agit ici de trouver tous les nombres k compris entre 1 et 9 (`for k in range(1,10)`) qui ne sont pas encore dans le bloc auquel appartient la case (i, j) (`if k not in bloc(S,i,j)`), qui ne sont pas non plus sur la ligne i (`if k not in ligne(S,i)`) ni dans la colonne j (`for k not in colonne(S,j)`).

6

```
def suivante(i,j):
    if j == 8:
        i += 1
        j = 0
    else:
        j += 1

    return i,j
```

7

```
def solve(S,i,j):
    if i == 9:
        return True
    elif S[i][j] > 0:
        i,j = suivante(i,j)
        return solve(S,i,j)
```

CORRIGÉS

```
    for k in possibles(S,i,j):
        S[i][j] = k
        a,b = suivante(i,j)
        if solve(S,a,b):
            return True
    S[i][j] = 0
    return False
```

Explications :

- on commence par le plus évident : si $i = 9$, c'est que l'on est arrivé à remplir la grille donc on retourne `True` ;
- sinon, si la case (i, j) est déjà remplie, on passe à la case suivante et on résout la grille ;
- le cas le plus difficile : si la case n'est pas remplie alors on teste chaque chiffre possible et on regarde si la résolution de la grille est possible en passant à la case suivante. Si tel est le cas, on renvoie `True` ;
- si on sort de la boucle, c'est que la grille n'est pas solvable.
  On remplace alors le chiffre de la case (i, j) par 0 (afin de revenir en arrière d'une case) et on retourne `False` pour signifier la non-résolubilité de la grille actuelle.

Téléchargez le programme complet avec Nathan Live.

ANECDOTE

La méthode consistant à tester tous les nombres et à revenir en arrière si la solution n'est pas satisfaisante est appelée *backtracking*.

12 Preuve : incalculabilité

Enoncé p. 56

1 Notons $A = \{a_0, a_1, \cdots, a_{n-1}\}$ l'ensemble des caractères nécessaires à la rédaction des algorithmes.

Considérons alors la fonction f définie comme suit :

$$\begin{array}{ccc}
0 & \mapsto & a_0 \\
\vdots & \mapsto & \vdots \\
n-1 & \mapsto & a_{n-1} \\
n & \mapsto & a_0a_0 \\
n+1 & \mapsto & a_0a_1 \\
\vdots & \mapsto & \vdots \\
2n-1 & \mapsto & a_0a_{n-1} \\
2n & \mapsto & a_0a_0a_0 \\
\vdots & \mapsto & \vdots
\end{array}$$

f est manifestement une bijection de $\mathbb{N}$ vers l'ensemble C constitué des combinaisons des n caractères de A. Or, $\mathcal{A}$ (l'ensemble des algorithmes) est inclus dans C (car tout algorithme est une combinaison des caractères de A).

Il existe donc bien une bijection de $\mathbb{N}$ vers $\mathcal{A}$, donc $\mathcal{A}$ est dénombrable.

2 **(a)** $g(0) = 1 - f_0(0) = 1 - 1 = 0$.
$g(1) = 1 - f_1(1) = 1 - 1 = 0$.
$g(2) = 1 - f_2(2) = 1 - 0 = 1$.

(b) Dans un cas général,

- $g(0) = 1 - f_0(0)$ donc $g(0) \neq f_0(0)$, ce qui montre que $g \neq f_0$;
- $g(1) = 1 - f_1(1)$ donc $g(1) \neq f_1(1)$, ce qui montre que $g \neq f_1$;
- quel que soit l'entier k, $g(k) = 1 - f_k(k) \neq f_k(k)$, donc $g \neq f_k$.

(c) Quel que soit l'entier naturel k, $f_k \in \mathcal{B}$ donc $g : 1 - f_k \in \mathcal{B}$.

Or, nous venons de montrer à la question précédente que g n'était pas une fonction f_k, donc que $g \notin \mathcal{B}$. Il y a donc bien une contradiction.

Cela suppose donc que notre hypothèse de départ est fausse, à savoir que $\mathcal{B}$ est dénombrable.

3 D'une part, nous avons montré que l'ensemble des algorithmes était dénombrable, d'autre part que l'ensemble des fonctions représentant les problèmes de décisions ne l'était pas. Cela signifie que ce dernier ensemble a un cardinal supérieur à celui de l'ensemble $\mathcal{A}$, et donc qu'il existe plus de problèmes de décision que d'algorithmes.

Chapitre 3

Méthodes de programmation

Plan du chapitre

Exercice type 1

1 Qu'affiche le programme suivant ?

```
global g
def calculSomme(nb):
  global g
  print("g+{} = {}".format(nb,g+nb))

def calculDiff(nb):
  global g
  print("g-{} = {}".format(nb,g-nb))

g = 200
print("g={}".format(g))
calculSomme(50)
calculDiff(5)
```

2 On modifie la première fonction de la manière suivante :

```
def calculSomme(nb):
  global g
  g = g + nb
  print("g + {} = {}".format(nb,g))
```

Qu'affiche alors le programme principal précédent ?
Qu'est-ce que cette modification a permis de mettre en relief ?

Voir corrigé page 71

COURS

1 Les différents types de variables

1.1 Les variables locales

Définition 1

Une *variable locale* est une variable qui ne peut être utilisée que dans une fonction ou une classe.

Exemple : Considérons le programme suivant :

```
def maFonction(variable):
    try:
        print("'valeur' vaut {}.".format(valeur))
    except NameError:
        print("La variable 'valeur' n'existe pas encore.")
    valeur = variable
    print("'valeur' vaut {}.".format(valeur))

maFonction(10)
print(valeur)
```

Quand on l'exécute, il s'affiche le message suivant :

```
La variable 'valeur' n'existe pas encore.
'valeur' vaut 10.
NameError: name 'valeur' is not defined
```

Dans la fonction `maFonction`, on *essaie* (« `try` ») d'afficher la valeur contenue dans la variable `valeur`. Si cette valeur n'existe pas (« `except NameError` »), on affiche un message le spécifiant.
Ensuite, on affecte la valeur passée en paramètre à la variable `valeur`, puis on affiche cette valeur.
On appelle alors la fonction avec le paramètre « 10 ».

Pour finir, on demande d'afficher la valeur de la variable `valeur` à l'extérieur de la fonction, mais cette variable n'est pas reconnue (car nous sommes justement à l'extérieur de la fonction).
La variable `valeur` est *locale* : elle n'existe que dans la fonction définie, et pas ailleurs.

1.2 Les variables globales

Définition 2

Une *variable globale* est une variable déclarée à l'extérieur d'une fonction ou d'une classe pouvant être utilisée n'importe où dans le programme.

En Python, une variable sera désignée (référencée) comme globale à l'aide du mot-clé `global`, suivi du nom de la variable.

À noter toutefois que Python a un comportement particulier concernant les variables extérieures dans un espace local : il peut les lire, mais pas les modifier.

Exemples

Partie 1

```
def afficheA():
    print("a = {}".format(a))

a = 10
afficheA()
```

Partie 2

```
def modifieA():
    global a
    a+=1

a = 10
afficheA()
modifieA()
afficheA()
```

Dans la partie 1, la variable `a` est déclarée à l'extérieur de la fonction `afficheA()` ; pourtant, la fonction affiche bien sa valeur.

Dans la partie 2, `a` est toujours déclarée à l'extérieur des fonctions, mais il est nécessaire de la référencer avec le mot-clé `global` pour pouvoir modifier sa valeur.

2 Programmation impérative

Définition 3

La *programmation impérative* décrit les différentes opérations sous forme de suites d'instructions exécutées par la machine sur laquelle est lancé le programme pour modifier l'état du programme.

C'est le type de programmation (ou *paradigme* de programmation) le plus utilisé. La plupart des processeurs sont construits de manière à exécuter des suites d'instructions.

Définition 4 : effet de bord

Un *effet de bord* (*side effect*) est une modification provoquée par une fonction sur une dépendance partagée (variable ou périphérique par exemple) autre que celle explicitement spécifiée lors de l'appel de cette fonction.

Généralement, un effet de bord aboutit à des valeurs ou des comportements non prévus.

Solution de l'exercice type 1

1 Le programme affiche successivement : « 250 » (résultat obtenu avec l'instruction `calculSomme(50)`) et « 195 » (résultat obtenu avec l'instruction `calculDiff(5)`).

COURS

Solution de l'exercice type 1 (suite)

2 Le même programme principal donnera après modification : « 250 » (résultat obtenu avec l'instruction `calculSomme(50)`) et « 245 » (résultat obtenu avec l'instruction `calculDiff(5)`).

On constate que la fonction `calculSomme()` continue de fonctionner et retourne le même résultat. En revanche, la fonction `calculDiff()`, qui n'a pourtant pas été modifiée, retourne une erreur de calcul !

Cela met en relief le fait que `calculDiff()` est *victime* d'un effet de bord suite à la modification de `calculSomme()`.

Voir énoncé page 69

3 Programmation fonctionnelle

3.1 Principe de base

Définition 5

La *programmation fonctionnelle* est un concept de programmation ayant pour but principal d'éviter les effets de bord.
Pour cela, elle se concentre essentiellement sur les fonctions, d'où son appellation : elle s'interdit toute variable globale pour ne faire appel qu'à des fonctions, qui peuvent être imbriquées.

Exemple : voici deux programmes écrits en Python de deux manières différentes, mais dont le résultat est identique : incrémenter de 1 une variable. À gauche, la méthode de programmation est impérative, tandis qu'à droite, elle est fonctionnelle.

Impérative

```
variable = 0
def plus():
    global variable
    variable += 1
plus()
print( variable )
```

Fonctionnelle

```
def plus(variable):
    return variable + 1

print( plus(0) )
```

3.2 Manipulation sur les listes

3.2.1 - Introduction

En programmation fonctionnelle, l'utilisation de `pop()` ou `append()` sur une liste ne serait pas tolérée car ces fonctions modifient l'état de la liste courante.

À RETENIR

- `liste = liste.append(element)` est remplacé par :

```
liste + [element]
```

- `element = liste.pop()` est remplacé par :

```
liste, element = liste[:-1], liste[-1]
```

3.2.2 - La fonction « map »

En Python, la fonction `map` est un outil pour programmer de façon fonctionnelle. Cette fonction prend en argument une fonction `f` et une liste de données `L`.

Elle applique successivement la fonction `f` à chacune des données de la liste et stocke les résultats obtenus dans une nouvelle liste `G`.

Ainsi, `G = [ f(L[0]), f(L[1]), ..., f(L[-1]) ]`.

Exemples

- On souhaite connaître la longueur de plusieurs chaînes de caractères.

```
def longueur(liste):
    return list(map(len, liste))

print(longueur(["Bordeaux", "Paris", "Lyon"]))
```

Ce programme affiche : `[8, 5, 4]`.

- On souhaite calculer le carré des longueurs calculées précédemment.

```
def longueur(liste):
    return list(map(len, liste))

def carre(liste):
    return list(map(lambda x: x**2, longueur(liste)))

print(carre(["Bordeaux", "Paris", "Lyon"]))
```

Ce programme affiche : `[64, 25, 16]`.

3.2.3 - La fonction « reduce »

La fonction `reduce` (du module `functools`) est un autre outil de la programmation fonctionnelle. Elle fonctionne de la même façon que `map`, mais à partir d'une fonction `lambda` à deux paramètres : l'élément de la liste à traiter (`x`) et le résultat du traitement précédent (`a`). Elle ne retourne qu'un seul résultat, qui correspond au dernier calcul effectué.

COURS

ATTENTION

Il est à noter que la fonction `lambda` n'est pas exécutée sur le premier élément de la liste (en tant que `x`), pour lequel il n'existe pas encore d'antécédent ! C'est le premier élément de la liste qui sert d'antécédent lors de la première exécution de la fonction `lambda`. Cette toute première exécution de `lambda` par `reduce()` porte donc sur le deuxième élément de la liste (en tant que `x`) avec le premier élément de la liste comme antécédent (devenant `a`).

Exemple : on souhaite calculer le résultat de l'opération :

$$2^2 + 3^2 + 4^2 + 5^2.$$

On utilise alors le programme suivant :

```
from functools import reduce

def somme(liste):
    return reduce(lambda a, x: a + x**2, liste)

print( somme([2,3,4,5]) )
```

Ce programme affiche « 52 », et non « 54 » (qui correspondrait au résultat de $2^2 + 3^2 + 4^2 + 5^2$). En effet, ce programme calcule le résultat de l'opération : $2 + 3^2 + 4^2 + 5^2$.

À RETENIR

- La fonction `reduce()`, tout comme la fonction `map()`, exécute de multiples fois la fonction `lambda` : `len(liste)-1` itérations.
- Contrairement à `map()`, les résultats des exécutions successives (hormis le dernier) ne sont pas conservés. Le résultat de l'itération $n - 1$ (de lambda) est utilisé en tant qu'antécédent (`a`) de l'exécution n (de lambda).

C'est cette « mécanique » qui permet d'agréger (de consolider) le résultat final, donc de *réduire* les itérations successives à un unique résultat.

Voir exercices 1 à 6 pages 81 à 82.

4 Diviser pour régner

Exercice type 2

Écrire un algorithme de *tri fusion* permettant de trier une liste de chiffres en la scindant en deux parties à peu près égales, puis en triant récursivement chacune de ces sous-listes.
Écrire le code Python correspondant.

Voir corrigé page 75

Pour trier une liste d'entiers, il vient naturellement à l'esprit de la scinder en deux sous-listes de même taille, qu'on triera séparément, puis de combiner le résultat de chacun de ces tris.

Si par exemple on veut trier la liste `[ 5 2 3 8 4 1 6 9 ]`, on peut diviser cette liste en :

`[ 5 2 3 8 ]` et `[ 4 1 6 9 ]`,

puis les trier séparément :

`[ 2 3 5 8 ]` et `[ 1 4 6 9 ]`,

et ensuite combiner ces résultats pour obtenir la liste complète triée :

`[ 1 2 3 4 5 6 8 9 ]`.

On aurait pu aussi scinder en deux chacune des sous-listes `[ 5 2 3 8 ]` et `[ 4 1 6 9 ]` pour parvenir de manière récursive au même résultat.

Cette méthode est appelée « Diviser pour régner ».

Définition 6 : diviser pour régner

Technique algorithmique consistant à diviser un problème initial en sous-problèmes que l'on résout récursivement ou directement, puis à combiner les résultats.

Elle comporte trois phases :

- *diviser* : on divise le problème initial en plusieurs sous-problèmes,
- *régner* : on résout récursivement ou directement chacun des sous-problèmes,
- *combiner* : on combine les différents résultats obtenus pour parvenir à la solution finale.

Voir exercices 7 à 9 pages 83 et 84.

Solution de l'exercice type 2

Un algorithme possible est le suivant :

```
fonction tri(L):
   L : liste à trier
   N ← longueur de L
   Si N = 0 ou N = 1:
     Retourner L
   Sinon:
     M ← E(N/2)
     L1 ← liste des M premiers éléments de L
     L2 ← liste L privée de L1
     Retourner la fusion de tri(L1) et tri(L2)
```

COURS

Solution de l'exercice type 2 (suite)

En Python, cela donne :

Tri fusion en Python

```
def fusion(L1,L2):
    if L1 == []:
        return L2
    elif L2 == []:
        return L1
    elif L1[0] < L2[0]:
        return [L1[0]] + fusion(L1[1:],L2)
    else:
        return [L2[0]] + fusion(L1,L2[1:])

def tri(L):
    n = len(L)
    if n < 2:
        return L
    else:
        m = n // 2
        return fusion(tri(L[:m]),tri(L[m:]))
```

ATTENTION

Il ne faut jamais oublier d'écrire une condition d'arrêt à une fonction récursive, sous peine d'aboutir à une boucle infinie...

Voir énoncé page 74

5 Recherche textuelle

PROBLÉMATIQUE

Rechercher une chaîne de caractères dans une autre est un problème récurrent, que ce soit en informatique ou dans d'autres sciences comme par exemple en génétique, dès lors qu'il s'agit de localiser un motif dans une séquence d'ADN. Une approche naïve mènerait à un algorithme de complexité conséquente. Comment y remédier ?

5.1 Approche naïve

Nous disposons de la séquence d'ADN suivante :

ATAACAGAGAATAAGGCTAGGAAATACTGAA

Nous souhaitons savoir si le motif « AGGCTA » apparaît dans cette séquence.

L'approche naïve consiste à effectuer une comparaison du motif avec la séquence en partant de la droite et en décalant le motif vers la droite jusqu'à trouver une coïncidence.

Comparaison	Coïncidence
ATAACAGAGAATAAGGCTAGGAAATACTGAA AGGCTA	Non
ATAACAGAGAATAAGGCTAGGAAATACTGAA •AGGCTA	Non
ATAACAGAGAATAAGGCTAGGAAATACTGAA ••AGGCTA	Non
⋮	
ATAACAGAGAATAAGGCTAGGAAATACTGAA ••••••••••••AGGCTA	Oui

Dans cette approche, on aligne donc le motif avec la séquence sur la gauche et on compare les caractères de la gauche vers la droite.

Ainsi, pour la première comparaison, les premiers caractères (A) coïncident, mais pas les deuxièmes ; on décale donc le motif d'un caractère vers la droite.

Le « A » se retrouve donc sous un « T » : les deux caractères ne coïncident pas, donc on décale à nouveau.

On fait cela jusqu'à obtenir une parfaite coïncidence.

En Python, cela donne :

Approche naïve

```
def comparaison(sequence , motif):
    for i in range(len(sequence) - len(motif) + 1 ):
        if sequence[i:i+len(motif)] == motif:
            return sequence[:i]+'['+motif+']'+sequence[i+
    len(motif)+1:]

    return False

sequence = "ATAACAGAGAATAAGGCTAGGAAATACTGAA"
motif = "AGGCTA"

print(comparaison(sequence,motif))
```

qui retourne : `ATAACAGAGAATA[AGGCTA]GAAATACTGAA`.

La complexité de cet algorithme est $O\big(m(s-m)\big)$, où m et s représentent respectivement le nombre de lettres du motif et de la séquence.

COURS

5.2 L'algorithme de Boyer-Moore

5.2.1 - Approche à travers un exemple

Nous allons expliquer cet algorithme à travers l'exemple de la séquence et du motif précédent.

- On commence par aligner la séquence et le motif à gauche, comme précédemment, mais la comparaison se fait sur le dernier caractère du motif :

  ATAACAGAGAATAAGGCTAGGAAATACTGAA
  AGGCTA

  Ce dernier caractère coïncide avec le 6e de la séquence. On compare alors les deux caractères précédents :

  ATAACAGAGAATAAGGCTAGGAAATACTGAA
  AGGCTA

  Ils ne coïncident pas.
- Le caractère rencontré dans la séquence (C) apparaît tout de même dans le motif : nous allons donc décaler le motif de sorte que ce caractère (C) coïncide avec le même caractère du motif :

  ATAACAGAGAATAAGGCTAGGAAATACTGAA
  AGGCTA
- On recommence alors la comparaison en partant de la droite du motif :

  ATAACAGAGAATAAGGCTAGGAAATACTGAA
  AGGCTA

  Ils ne coïncident pas.
- Mais « G » apparaît dans le motif donc on décale ce dernier de sorte à faire coïncider le « G » de la séquence avec le premier « G » du motif rencontré en partant de la droite :

  ATAACAGAGAATAAGGCTAGGAAATACTGAA
  AGGCTA
- On recommence alors la comparaison en partant de la droite du motif :

  ATAACAGAGAATAAGGCTAGGAAATACTGAA
  AGGCTA

  Les caractères de droite coïncident, mais pas leurs précédents.
- On décale donc le motif de sorte que le « G » de la séquence coïncide avec le premier « G » du motif en partant de droite :

  ATAACAGAGAATAAGGCTAGGAAATACTGAA
  AGGCTA

  Les derniers caractères ne coïncident pas.

- On décale donc le motif de sorte que le « T » de la séquence coïncide avec le « T » du motif qui est juste avant :

ATAACAGAGAATAAGGCTAGGAAATACTGAA
AGGCTA

Les deux derniers caractères coïncident mais pas les troisièmes.

- On décale donc le motif de sorte à faire coïncider le « A » de la séquence avec le « A » du motif qui est à gauche des caractères qui coïncidaient (donc avec le « A » de gauche du motif) :

ATAACAGAGAATAAGGCTAGGAAATACTGAA
AGGCTA

Les derniers caractères ne coïncident pas.

- On décale donc le motif de sorte à aligner le « G » de la séquence avec celui du motif qui est le plus à droite :

ATAACAGAGAATAAGGCTAGGAAATACTGAA
AGGCTA

Il y a alors correspondance entre le motif et le bout de séquence. L'algorithme s'arrête donc.

5.2.2 - Table de sauts

À travers l'exemple vu précédemment, on peut facilement comprendre l'utilité de construire une première table qui indique la distance maximale de chaque caractère distinct au dernier dans le motif : on appelle cette table la « première table de sauts », ne dépendant que du motif et pas de la séquence.

On commence par l'avant-dernier caractère (en effet, le dernier caractère du motif n'est pas pris en compte pour cette table) : le motif étant « AGGCTA », le caractère « T » est à une distance de 1 du dernier « A », etc.

On obtient la table suivante :

Caractère	Distance
T	1
C	2
G	4
A	5
Autres	6

Tableau 3.1 – 1re table de sauts du motif « AGGCTA »

Remarque : pour un motif où les caractères sont tous distincts, on aura par exemple :

Caractère	Distance
C	1
B	2
A	3
Autres	4

Tableau 3.2 – 1re table de sauts du motif « ABCD »

COURS

Cette première table concerne la *séquence*, bien qu'elle soit construite sur le *motif*, car elle donne le décalage du motif à faire sur la séquence (on doit décaler le motif de 1, 2, 3,... sur la séquence).

5.2.3 - Programme en Python

La table de sauts précédemment vue nous donne l'idée de construire un dictionnaire contenant toutes les lettres de l'alphabet considéré ainsi que leur premier rang dans le motif à trouver, puis de s'appuyer sur ces valeurs pour décaler le motif. C'est le but de la fonction `table_sauts` du programme à télécharger ci-dessous.

Téléchargez le programme Python correspondant à l'algorithme de Boyer-Moore.

Programmation fonctionnelle

1 Réécriture

★ — 5 min — Corrigé p. 85

Réécrire de façon fonctionnelle le programme suivant :

```
L = [ 'Jean' , 'Luc' , 'Lucie' , 'Marie' ]

for i in range( len(L) ):
    L[i] = hash(L[i])
```

Remarque : la fonction `hash` renvoie la valeur de hachage d'un objet. Une fonction de hashage produit un condensé des données de l'objet. Ce condensé est de taille fixe, dont la valeur diffère suivant la fonction utilisée.

2 Fonction carré

★ — 5 min — Corrigé p. 85

Écrire en Python de manière fonctionnelle une fonction `carre(n)` qui affiche sous forme de liste le carré des entiers de 0 à n (compris).

3 Compteur d'occurrences

★★ — 10 min — Corrigé p. 85

À l'aide de la fonction `reduce` du module `functools`, écrire en Python une fonction `compteur(liste,mot)` qui retourne le nombre d'occurrences de la chaîne `mot` dans la liste `liste`.

4 Taille d'une liste

★★ — 10 min — Corrigé p. 85

Supposons que la fonction `len` n'existe pas.

Écrire, en Python et en utilisant le paradigme de la programmation fonctionnelle, une fonction récursive `taille(liste)` admettant une liste comme argument et retournant sa taille.

5 Calcul d'une taille moyenne

★★ — 15 min — Corrigé p. 86

On dispose d'une liste de dictionnaires `gens` :

```
gens = [
       'nom': 'Pierre' , 'taille': 178,
       'name': 'Marie' , 'taille': 181,
       'name': 'Paul' , 'taille': 157
]
```

INTERROS

Écrire en Python de manière fonctionnelle une fonction `average(gens)` admettant une liste de dictionnaires de la forme décrite précédemment et retournant la taille moyenne des personnes figurant dans cette liste.

Aide : on utilisera les fonctions :

`reduce` et `filter(fonction,dictionnaire)`,

cette dernière fonction renvoyant un dictionnaire contenant tous les éléments du dictionnaire d'origine pour lesquelles `fonction = True`.

On pourra aussi importer la fonction `add` du module `operator` de sorte que `reduce(add,liste)` retourne la somme des éléments de `liste`.

6 Une fonction « pipeline_each »

★★★ 10 min Corrigé p. 87

On souhaite écrire en Python une fonction `pipeline_each(liste,fonctions)` comme suit :

- son premier paramètre (`liste`) reçoit une liste de chaînes de caractères ;
- son second paramètre (`fonctions`) reçoit une liste d'une ou plusieurs fonctions qui, chacune, reçoit une liste de chaînes en argument et y effectue un traitement.

La fonction `pipeline_each(liste,fonctions)` doit appliquer, tour à tour, chacune des fonctions de la liste `fonctions` à la liste de chaînes de caractères `liste`. Puis elle doit renvoyer cette liste à l'issue des traitements successifs.

Compléter le programme suivant :

```
from functools import reduce

def pipeline_each(liste , fonctions):
    ...

def format1(chaine):
    return chaine.title()

def format2(chaine):
    return chaine.replace('.' , ';')

print( list( pipeline_each(
        ['Le monde est à nous.',
         ' Il ne faut pas le détériorer.'] ,
        [format2,format1]
    ) ) )
```

afin d'avoir en affichage :

```
['Le Monde Est À Nous;', ' Il Ne Faut Pas Le Détériorer;']
```

Diviser pour régner

7 Fonction « maximum » d'une liste

★ — 15 min — Corrigé p. 87

On dispose d'une liste `L` de nombres.

En utilisant la méthode du « diviser pour régner », implémenter une fonction `maximum(L,debut,fin)` retournant le maximum de `L`.

8 Algorithme de Karatsuba

★★★ — 30 min — Corrigé p. 87

L'objectif de cet exercice est d'utiliser l'algorithme de Karatsuba afin de trouver le produit de deux polynômes P et Q.

On rappelle qu'un *polynôme en X de degré n* est une expression de la forme :

$$P(X) = \sum_{k=0}^{n} a_k X^k.$$

On représente alors en Python le polynôme P par la liste `[a0,a1,..,an]` des coefficients a_0, a_1, ..., a_n.

1 Écrire une fonction `standardise(P,Q)` qui retourne un couple de deux listes de dimension $\max\big(\deg(P);\deg(Q)\big)$, où $\deg(P)$ représente le degré de P.

Par exemple, si $P(X) = 3 + 2X$ et $Q(X) = 5 - 7X + 4X^2$ (soit en Python : `P = [3,2]` et `Q = [5,-7,4]`), la fonction devra retourner : `([3,2,0] , [5,-7,4])`.

2 Écrire une fonction `add(P,Q)` qui retourne une liste dont les éléments sont les coefficients de la somme des deux polynômes P et Q, puis une fonction `sous(P,Q)` qui retourne sous forme de liste le résultat de $P(X) - Q(X)$.

Par exemple, si $P(X) = 3 + 2X$ et $Q(X) = 5 - 7X + 4X^2$ alors :

- `add(P,Q) = [8,-5,4]` ;
- `sous(P,Q) = [-2,9,-4]`.

3 L'algorithme de Karatsuba consiste à décomposer P et Q de la manière suivante :

$$\begin{cases} P(X) &= X^k P_1(X) + P_0(X), \quad \deg P_0 \leqslant k \\ Q(X) &= X^k Q_1(X) + Q_0(X), \quad \deg Q_0 \leqslant k \end{cases}$$

où $k = E\left(\frac{n}{2}\right)$ (partie entière de la moitié de `n`).

(a) On suppose que :

$$(PQ)(X) = (P_0Q_0)(X) + \big((P_0+P_1)(Q_0+Q_1) - P_0Q_0 - P_1Q_1\big)(X)X^k + (P_1Q_1)(X)X^{2k}.$$

Écrire une fonction `prodmonome(P,k)` qui renvoie sous forme de liste le produit du polynôme P par X^k.

(b) En déduire une fonction `prod(P,Q)` qui renvoie sous forme de liste le produit de deux polynômes P et Q.

(c) Tester cette fonction sur les polynômes :
$P(X) = 5-2X^4+3X^3-X^2+2X-1$ et $Q(X) = -3+5X+2X^2$.

9 Le problème du sac à dos

★★★ 30 min Corrigé p. 89

On dispose d'un sac à dos ne pouvant supporter qu'un poids P. On dispose de plus d'une collection de n objets, chaque objet ayant un poids p_i et une valeur v_i, $1 \leqslant i \leqslant n$, avec $p_1 + p_2 + \cdots + p_n > P$.

On souhaite mettre dans le sac le plus d'objets possible tout en maximisant la valeur totale des objets.

En utilisant une approche récursive « diviser pour régner », donner une solution en Python qui affiche la valeur maximale.

Appliquer le programme au cas où l'ensemble des objets (poids,valeur) est :

$$E = \{(12,14);\ (1,2);\ (4,10);\ (1,1);\ (2,2)\}$$

et où $P = 15$.

1 Réécriture

Enoncé p. 81

Voici une possibilité :

```
L = [ 'Jean' , 'Luc' , 'Lucie' , 'Marie' ]

print( list( map(hash , L) ) )
```

2 Fonction carré

Enoncé p. 81

Une possibilité est la suivante :

```
def carre(n):
    return list( map(lambda x : x*x , range(n + 1) ) )

print( carre(5) )
```

3 Compteur d'occurrences

Enoncé p. 81

Une possibilité est la suivante :

```
from functools import reduce

def compteur(liste,mot):
    return reduce( lambda a , x : a + x.count(mot) ,
    liste , 0 )

print( compteur( ['Le coq Hardi est le plus beau du
    poulailler ' , 'Je pense que le voisin est un geek '
    ] , 'le' ) )
```

Dans cet exemple, le programme retourne « 3 » ; ce n'est pas un bug : le premier « Le » n'est pas compté (car il a une majuscule) mais dans le mot « poulailler », la chaîne « le » apparaît une fois. Il y a donc bien trois occurrences de « le » dans la liste.

4 Taille d'une liste

Enoncé p. 81

Une première possibilité est la suivante :

Version itérative

```
def taille(liste):
    s = 0
    for x in liste: s += 1
    return s
```

Une deuxième possibilité est, en utilisant la récursivité :

Version fonctionnelle récursive

```
def taille(liste):
    if not liste:
        return 0
    return 1 + taille( liste[1:] )
```

5 Calcul d'une taille moyenne

Enoncé p. 81

Voici une possibilité :

```
from functools import reduce
from operator import add

def average(gens):
    tailles = list(
                map( lambda x: x['taille'] ,
                    filter(
                        lambda x: 'taille' in x , gens
                    )
                )
            )

    if len(tailles) > 0:
        return reduce(add , tailles) / len(tailles)

gens = [
        {'nom': 'Pierre' , 'taille': 178},
        {'name': 'Marie' , 'taille': 181},
        {'name': 'Paul'  , 'taille': 157}
       ]

print( average(gens) )
```

Ce programme affiche « 172 », qui correspond à la taille moyenne des trois personnes du dictionnaire.

6 Une fonction « pipeline_each »

Enoncé p. 82

Une solution envisageable est la suivante.

```
from functools import reduce

def pipeline_each(liste , fonctions):
    return reduce(
        lambda a , x: map(x , a) , fonctions ,liste )

def format1(chaine):
    return chaine.title()

def format2(chaine):
    return chaine.replace('.' , ';')

print( list( pipeline_each(
        [ 'Le monde est à nous.' ,
          ' Il ne faut pas le détériorer.' ] ,
        [format2,format1] ) ) )
```

7 Fonction « maximum » d'une liste

Enoncé p. 83

L'idée consiste à diviser en deux la liste L en deux parties et à trouver le maximum des deux parties. Le maximum global sera la plus grande des valeurs ainsi trouvées. On doit partager en deux jusqu'à obtenir des listes élémentaires (à 1 élément). On doit alors utiliser la récursivité.

Une fonction possible est alors la suivante :

```
def maximum(L,debut,fin):
    if debut == fin:
        return L[debut]
    milieu = (debut + fin) // 2
    x = maximum(L , debut , milieu)
    y = maximum(L , milieu + 1 , fin)
    return x if x > y else y
```

8 Algorithme de Karatsuba

Enoncé p. 83

1 Une fonction possible est la suivante :

```
def standardise(P, Q):
    n, m = len(P), len(Q)
    if n < m: P = P + [0] * (m - n)
    else: Q = Q + [0] * (n - m)
    return P, Q
```

CORRIGÉS

2 Une fonction possible est la suivante :

```
def add(P, Q):
    P, Q = standardise(P, Q)
    return [ P[i] + Q[i] for i in range(len(P)) ]

def sous(P, Q):
    P, Q = standardise(P, Q)
    return [ P[i] - Q[i] for i in range(len(P)) ]
```

3 **(a)** Une fonction possible est la suivante :

```
def prodmonome(P, k):
    return [0] * k + P
```

En effet,

$$P(X)X^k = X^k \sum_{i=0}^{n} a_i X^i = \sum_{i=0}^{n} a_i X^{i+k} = \sum_{i=k+1}^{n+k} a_{i-k-1} X^i.$$

Ainsi, a_0 n'est plus le coefficient de X^0 mais de X^{0+k}, a_1 est celui de X^{1+k}, etc.

(b)

```
def prod(P, Q):
    n = max(len(P), len(Q))
    if n == 1:
        return [P[0]*Q[0]]
    P , Q = standardise(P, Q)
    k = n // 2
    P0 , P1 = P[0:k] , P[k:n]
    Q0 , Q1 = Q[0:k] , Q[k:n]
    R0 = prod(P0 , Q0)
    R1 = prod(P1 , Q1)
    R2 = prod( add(P0 , P1) , add(Q0 , Q1) )
    R2 = sous( sous(R2 , R1) , R0)
    R1 = prodmonome(R1 , 2 * k)
    R2 = prodmonome(R2 , k)
    return add( add(R0 , R1) , R2)
```

(c) On définit dans cette question P et Q ainsi :

```
P , Q = [5,-2,3,-1,2,-1] , [-3,5,2]
```

En appelant la fonction `prod`, on obtient la liste :

```
[-15, 31, -9, 14, -5, 11, -1, -2, 0, 0, 0]
```

ce qui signifie que :

$$PQ(X) = -15 + 31X - 9X^2 + 14X^3 - 5X^4 + 11X^5 - X^6 - 2X^7.$$

Télécharger le programme complet.

9 Le problème du sac à dos

Enoncé p. 84

Pour résoudre un tel problème, il faut avant tout passer par une formalisation mathématique : une traduction algébrique.

- Notons $E = \{(p_1, v_1); \cdots; (p_n, v_n)\}$ l'ensemble des n objets.
- La solution attendue sera un n-uplet $(x_1; x_2; \ldots; x_n)$ où $x_i = 0$ si l'on ne prend pas l'objet i et $x_i = 1$ dans le cas contraire.
- Il faut alors que $\sum_{i=1}^{n} x_i p_i \leqslant P$ et $\sum_{i=1}^{n} x_i v_i$ maximale.

Nous avons vu dans le livre de Première NSI qu'une approche gloutonne était peu recommandée car dans certains cas, la solution donnée n'est pas optimale.

On nous demande ici une approche récursive de type « diviser pour régner ». Il est alors important dans ce type d'exercices d'établir une formule de récurrence. Le raisonnement prend nécessairement en compte le poids et la valeur d'un objet.

Notons $V(i, p)$ la valeur maximale des objets que l'on peut mettre dans un sac de capacité p en ne prenant que les i premiers objets. Ce qui nous intéresse est alors $V(n, P)$. On a alors :

- $V(0, p) = 0$ car s'il n'y a pas d'objet, la valeur maximale est nécessairement 0 ;
- Si $p_i > p$ alors $V(i, p) = V(i-1, p)$ car si le poids de l'objet i est supérieure au poids maximal, on ne l'ajoute pas et la valeur maximale des objets reste inchangée par rapport à celle de l'étape précédente ;
- Enfin, si $p_i \leqslant p$ alors $V(i, p) = \max\left(V(i-1, p); V(i-1, p-p_i) + v_i\right)$ car il faut regarder la valeur obtenue en ajoutant la valeur de l'objet i (v_i) à la valeur maximale correspondant à $i-1$ objets dans un sac de capacité $p - p_i$.

Ces relations de récurrence nous mènent alors à l'implémentation suivante :

```
def sac(E,P,i):
    if i == 0: return 0
    if E[i-1][0] > P: return sac(E , P , i-1)
    else: return max( sac(E , P , i-1) , E[i-1][1] +
                      sac(E , P-E[i-1][0] , i-1) )
def initSac(E,P):
    return sac(E , P , len(E))
```

Cela donne avec notre exemple :

```
>>> E = [ (12,14) , (1,2) , (4,10) , (1,1) , (2,2) ]
>>> P = 15
>>> initSac(E,P)
18
```

« 18 » est en effet la valeur maximale que l'on obtient en faisant $14 + 2 + 2$ (objets 1, 2 et 5).

Chapitre 4

Programmation orientée objet, programmation dynamique

Plan du chapitre

1 Programmation orientée objet (POO)

Exercice type 1

On considère une classe `Personnage` représentant un personnage de jeu.
Le plateau du jeu est représenté par un repère orthonormé à trois axes. La position du joueur dans le plateau est repérée par ses attributs x, y et z.
Écrire les méthodes `avance`, `droite` et `saute` permettant respectivement de faire avancer, aller à droite et sauter le personnage, c'est-à-dire d'augmenter de 1 respectivement x, y et z.
Implémenter de plus une autre méthode `coord` renvoyant les coordonnées sous forme d'un triplet.

Voir corrigé page 96

PROBLÉMATIQUE

Comment modéliser des éléments du monde réel sous forme d'objets informatiques afin de les manipuler de manière simple ?

1.1 Historique et définitions

La *programmation orientée objet* (en abrégé : POO) est un paradigme de programmation qui consiste à considérer un programme comme un ensemble d'objets en interaction.

COURS

Les *objets* peuvent représenter un concept du monde réel (comme un meuble, une personne ou encore un livre par exemple).

Pour faire de la POO, on peut utiliser un *langage à classes*, comme Python.

La POO trouve son origine en Norvège au début des années 1960. Elle fut élaborée par les informaticiens Ole-Johan Dahl et Kristen Nygaard.

Dans les années 1970, elle fut reprise par l'informaticien américain Alan Kay qui contribua à son essor. Cependant, il fallu attendre les années 1990 pour que ses principes soient clairement formalisés.

Retrouvez une présentation originale de la programmation orientée objet en vidéo avec Nathan Live.

1.2 Les classes sous Python

1.2.1 - Introduction

Nous allons considérer un objet `Individu` qui représente une personne.

Cette personne a plusieurs caractéristiques. Par exemple :

- sa date de naissance ;
- son genre ;
- sa taille ;
- sa masse ;
- etc.

qui vont représenter les *variables* de la personne.

De plus, cette personne peut faire plusieurs choses :

- parler ;
- manger ;
- marcher ;
- etc.

qui vont correspondre à des *fonctions* propres à elle-même.

Pour définir cette personne ainsi que les fonctions qui lui sont associées, on peut définir une *classe*.

1.2.2 - Définitions et vocabulaire

Définition 1

Une *classe* est une entité informatique qui définit en son sein :

- des variables membres (appelées *attributs*) ;
- des fonctions membres (appelées *méthodes*).

Exemple : voici un exemple en Python où l'on définit une classe `Personne` à l'aide de trois caractéristiques (année de naissance, taille en cm, masse en kg) et où l'on définit une fonction propre à la classe (grandir) qui admet un paramètre numérique (exprimé en cm).

```
class Personne:
    def __init__(self , birthyear , tall , weight):
        self.birthyear = birthyear
        self.tall = tall
        self.weight = weight

    def grandir(self,h):
        self.tall += h


luc = Personne(2002,178,69)
init_tall = luc.tall
luc.grandir(8)
new_tall = luc.tall
print("Luc pèse" , luc.weight , "kg.")
print("Luc mesure à présent" , new_tall , "cm après avoir
    grandi de 8 cm en 1 an. Avant ça, il mesurait:" ,
    init_tall , "cm.")
```

Ce programme affiche :

```
Luc pèse 69 kg.
Luc mesure à présent 186 cm après avoir grandi de 8 cm en
1 an. Avant ça, il mesurait: 178 cm.
```

Définitions 2

Dans l'exemple précédent,

- `birthyear`, `tall` et `weight` sont appelées des *attributs* de la classe `Personne` ;
- `grandir` est appelée une *méthode* de la classe `Personne`.

On accède aux attributs d'une classe à l'aide de la syntaxe :

```
<nom de la classe>.<nom de l'attribut>
```

On accède aux méthodes d'une classe à l'aide de la syntaxe :

```
<nom classe>.<nom méthode>(<arguments potentiels>)
```

COURS

1.2.3 - Constructeur d'une classe

Définition 3

Le *constructeur* d'une classe est une fonction :

- dont le nom est nécessairement `__init__` ;
- admettant au moins un paramètre nommé `self` placé obligatoirement en premier s'il y en a plusieurs.

Le paramètre `self` représente l'objet cible : c'est une variable qui contient une référence vers l'objet qui est en cours de création. Grâce à ce dernier, on va pouvoir accéder aux attributs et fonctionnalités de l'objet cible.

1.2.4 - Instanciation et variables d'instance

Une classe est une sorte de moule qui sert à fabriquer des objets.

L'*instanciation* est l'opération qui consiste à créer un objet. L'objet ainsi créé est alors appelé une *instance* de la classe.

Exemple : si on reprend la classe `Personne` de l'exemple précédent, en définissant :

```
luc = Personne(2002,178,69)
```

nous avons défini une instance de la classe `Personne`.

On peut ainsi définir plusieurs instances d'une même classe, chacune ayant des attributs qui lui sont propres.

Si on demande d'afficher la variable `luc`, on voit apparaître quelque chose de la forme :

```
<__main__.Personne object at 0x02CDDC50>
```

Cela signifie que `luc` est une référence vers l'objet créé, c'est-à-dire une indication sur son emplacement en mémoire. Ici, par exemple, l'objet `Personne` se trouve en mémoire à la position 0x02CDDC50.

Un attribut de la classe est représenté par une variable appelée *variable d'instance*, accessible à l'aide du paramètre `self`. L'idée, dans notre exemple, est que le constructeur initialise ces variables d'instances, qui seront alors stockées dans l'objet et en mémoire pour toute la durée de vie de l'objet.

Il est important de faire la différence entre les deux types de variables qui se trouvent dans le code de ce constructeur :

- la variable `self.tall` (par exemple) représente la variable d'instance, c'est-à-dire celle associée à l'objet, qui existe à partir de la création de l'objet jusque sa destruction ;
- la variable `tall` représente le paramètre reçu par le constructeur et n'existe que dans le corps de ce dernier.

Le paramètre `self` permet donc d'accéder aux variables d'instance, c'est-à-dire aux attributs de l'objet cible, depuis le constructeur.

1.2.5 - Méthodes d'une classe

Définition 4

Une *méthode* d'une classe est une fonction définie au sein de la classe et admettant au moins le paramètre `self` (obligatoirement en premier s'il y a plusieurs paramètres).

Le paramètre `self` représente l'objet cible sur lequel la méthode est appelée. Il permet notamment d'avoir accès aux variables d'instance de l'objet.

Dans l'exemple précédent, `grandir(self,h)` est une méthode admettant deux paramètres, dont le second est un nombre. Cette méthode redéfinit la variable d'instance `self.tall` en lui ajoutant la valeur passée en second paramètre.

Une méthode est appelée en spécifiant l'objet cible (ici, `luc`) suivi de la méthode en utilisant l'opérateur d'appel : le point. Cela donne ici :

luc.grandir(8)

1.2.6 - Redéfinition des opérations

Maintenant que l'on sait définir un objet à l'aide d'une classe, on peut s'intéresser aux opérations que l'on peut faire sur ces objets.

Par exemple, si on souhaite définir un objet `vector` représentant un vecteur du plan, il serait utile de pouvoir définir la somme de deux de ces vecteurs, qui est aussi un objet de classe `vector` (car la somme de deux vecteurs du plan est aussi un vecteur du plan).

Python permet de définir une telle opération de la manière suivante :

```
class vector:
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __add__(self,other):
        return vector(self.x + other.x , self.y + other.y)
```

En mode console, cela donne par exemple :

```
>>> u , v = vector(-2,8) , vector(3,-5)
>>> w = u + v
>>> print("w({},{})".format(w.x,w.y))
w(1,3)
```

Cette capacité du langage Python s'appelle la *surcharge d'opérateur*.

COURS

1.2.7 - Encapsulation

La classe `vector` définie précédemment aurait pu être définie autrement :

```
class vector:
    def __init__(self,x,y):
        self.coord = (x , y)

    def __add__(self,other):
        return vector(self.coord[0] + other.coord[0] , self.
    coord[1] + other.coord[1])
```

On est passé de deux variables d'instance à une seule, mais cette différence ne doit pas être perçue par l'utilisateur.trice : la manière dont on gère les valeurs passées en paramètres au sein de la classe ne doit pas être vue « de l'extérieur ». Ce principe fondamental de la POO est appelé *encapsulation*.

Voir QCM 1 page 103 et exercices 3 à 13 pages 105 à 111.

Solution de l'exercice type 1

Une définition possible de la classe `Personnage` est la suivante :

```
class Personnnage:
    def __init__(self,x,y,z):
       self.x, self.y, self.z = x, y, z
    def avance(self):
        self.x += 1
    def droite(self):
        self.y += 1
    def saute(self):
        self.z += 1
    def coord(self):
        return (self.x , self.y , self.z)
```

En utilisant le code, voici ce que donne un exemple d'utilisation :

```
>>> Lara = Personnnage(0,0,0)
>>> Lara.avance()
>>> Lara.saute()
>>> Lara.coord()
(1,0,1)
```

Voir énoncé page 91

2 Programmation dynamique

Exercice type 2

Étant donné un système de monnaie S comportant des pièces de valeurs différentes, écrire un programme qui affiche le nombre *minimal* de pièces à utiliser pour rendre une somme donnée, ainsi que leur énumération.

Voir corrigé page 101

2.1 Introduction

Il s'agit d'un problème d'optimisation, puisqu'on veut utiliser un nombre de pièces et de billets minimal. A travers l'étude de ce problème d'algorithmique classique, nous allons voir qu'il existe de multiples possibilités d'y répondre. La programmation dynamique est la plus efficace.

2.2 Approche gloutonne

C'est l'approche que font bon nombre de commerçants quand il rendent la monnaie à leurs clients.

Elle consiste à prendre le billet ou la pièce de valeur maximale en dessous de la somme à restituer, puis à recommencer ce geste par rapport à la différence. Cela donne l'algorithme suivant :

```
V ← valeur à atteindre
Tant que V n'est pas nulle:
   Choisir le billet ou la pièce de plus grande valeur v_i
   telle que v_i ⩽ V
   V ← V - v_i
Fin du Tant que
```

Exemple : comment rendre la somme de 14 € dans un système monétaire européen simplifié constitué uniquement de pièces de 1, 2, 5 et 10 € ?

On a alors $S = \{1; 2; 5; 10\}$.

Pour rendre 14 €, la plus grande valeur possible à déduire est 10 €, ce qui donne $14 - 10 = 4$.

La plus grande valeur possible à déduire de 4 € est 2×2 €, ce qui donne $4 - 2 \times 2 = 0$.

COURS

La solution renvoyée par notre algorithme est donc le quadruplet (0; 2; 0; 1), soit un total de 3 pièces. Les autres possibilités sont :

- (0; 2; 2; 0), soit 4 pièces,
- (0; 7; 0; 0), soit 7 pièces,
- (14; 0; 0; 0), soit 14 pièces.

L'algorithme glouton a ici donné la solution optimale. Mais cela ne serait pas forcément le cas pour d'autres systèmes.

Imaginons en effet un système composé uniquement de pièces de 1, 7 et 9 €, c'est-à-dire que $S = \{1; 7; 9\}$.

L'algorithme glouton nous amène à la solution (5; 0; 1), soit un rendu de 6 pièces. Alors que la solution optimale est (0; 2; 0), soit seulement 2 pièces.

Conclusion : l'algorithme glouton a le mérite d'être simple, mais il ne rend pas toujours la solution optimale. Il ne convient donc pas à la résolution de notre problème.

2.3 Approche récursive « diviser pour régner »

Notons à présent $S = \{v_1; v_2; \ldots; v_n\}$, v_i étant les valeurs des n pièces ou billets composant notre système monétaire S, et V la somme à rendre.

Nous allons calculer le nombre minimal de pièces à rendre que nous noterons `minimum`. Il servira ensuite à déterminer le n-uplet optimal recherché. La méthode précédente suggère de prendre une approche récursive.

- Si $V = 0$, on retourne « 0 »... Logique !
- Sinon, on fixe la valeur de la variable `minimum` (qui représentera au final le plus petit nombre de pièces) délibérément à une valeur assez grande (de toute évidence, le nombre de pièces est inférieur à la somme totale, donc fixer `minimum` à `V+1` nous assure que les valeurs de `minimum` qui seront calculées par la suite seront inférieures).
- On parcourt le système monétaire `S` et pour chaque valeur `S[i]`, on teste si elle est inférieure ou égale à `V` ; si tel est le cas, on prend en compte la valeur `S[i]` et on ajoute le nombre minimum de pièces ou billets nécessaires pour atteindre la valeur `V - S[i]`. Si la valeur obtenue est plus petite que `minimum` alors cela signifie que l'on a trouvé une combinaison plus optimale que les précédentes et on change la valeur de la variable `minimum` pour la valeur de `n`.
- Une fois que l'on a parcouru le système monétaire, on retourne la valeur de `minimum`, qui correspond au plus petit nombre possible de pièces et de billets.

le script page ci-contre est de type « diviser pour régner », puisque nous avons ramené le calcul du nombre recherché pour le montant V au même calcul pour des montants inférieurs (V − piece1, V − piece2, etc.). Nous avons divisé le problème initial, puis appliqué un traitement récursif. Notons cependant que pour cela, de nombreux calculs redondants ont été effectués. Cet algorithme n'est donc pas optimisé.

```
def Nombre(V , S = [1,7,9]):
    if V == 0:
        return 0
    else:
        minimum = V + 1
        for piece in S:
            if piece <= V:
                n = 1 + Nombre( V - piece )
                if n < minimum:
                    minimum = n
        return minimum

print( Nombre(14) )
```

2.4 Approche dynamique Top Down

Nous allons améliorer l'algorithme précédent en mémorisant dans une liste nommée `L` les calculs effectués à chaque étape. `L` est une liste unidimensionnelle de $(V+1)$ éléments qui va contenir le nombre de pièces minimal `L[v]` que l'on doit utiliser pour rendre la monnaie, pour chaque montant `v` compris dans $[0;\ V]$. Cette liste est construite de manière récursive à partir du montant initial V. De cette manière, les redondances de l'algorithme précédent disparaissent et l'algorithme est optimisé.

C'est ce qu'on appelle l'approche dynamique *Top Down* (de haut en bas).

Définition 5 : Top Down

La forme récursive *Top Down* est une technique d'optimisation ayant pour but de diminuer le temps d'exécution d'une fonction récursive en mémorisant (dans une liste par exemple) les valeurs retournées par cette fonction après avoir vérifié qu'elles n'existaient pas encore dans la liste.

Cette technique est aussi appelée *mémoïsation*.

```
def ConstructListe(V):
    L = [0]*(V+1)
    return NombreTopDown(V , L)

def NombreTopDown(V , L , S = [1,7,9]):
    if V == 0:
        return 0
    elif L[V] > 0:
        return L[V]
```

(Suite du programme page suivante)

COURS

```
    else:
        minimum = V + 1
        for piece in S:
            if piece <= V:
                n = 1 + NombreTopDown(V - piece , L)
                if n < minimum:
                    minimum = n
                    L[V] = minimum
        return minimum

print( ConstructListe(14) )
```

2.5 Approche dynamique Bottom Up

Dans cette approche, nous conservons notre liste `L` qui va être, cette fois, construite de manière itérative, en partant de 0 pour aller jusqu'à V. Il n'y a plus d'appel récursif comme c'était le cas pour la fonction `NombreTopDown`.

Cette technique de programmation dynamique est appelée *Bottom Up* (de bas en haut).

Définition 6 : Bottom Up

La forme itérative *Bottom Up* est une technique consistant à résoudre des problèmes en partant de celui qui a la taille la plus petite et en allant vers celui qui a la taille la plus grande, tout en stockant les résultats de chacun d'eux dans une liste (par exemple).

Les techniques d'optimisation « Bottom Up » et « Top Down » sont à la base de la programmation dynamique, ce qui nous amène à la définition suivante :

Définition 7 : programmation dynamique

La *programmation dynamique* est une méthode de résolution de problèmes d'optimisation en scindant le problème en plusieurs sous-problèmes et en utilisant l'une des méthodes « Top Down » ou « Bottom Up ».

Ainsi, la programmation dynamique est un paradigme de programmation complémentaire à « diviser pour régner », ce dernier se révélant inefficace dans certains cas.

Voici le fonctionnement du programme final.

- On commence par initialiser deux listes `L` et `valeurs`, toutes les deux étant de dimension `V+1`, `V` étant la valeur à décomposer, les indices variant de 0 à `V`.
  `L` est une liste où `L[x]` contiendra le nombre minimum de pièces et billets qui compose la valeur `x` (comprise entre 0 et `V`) tandis que `valeurs[x]` contiendra l'indice `i` pour lequel `S[i]` $\leqslant$ `x` et pour lequel `1+L[x-S[i]]` est plus petit ou égal au nombre minimum de pièces et billets nécessaires pour la décomposition de la somme souhaitée.

- On remplit donc `L` avec la boucle : `for x in range(1,V+1)`.
- Dans cette boucle, on commence par initialiser le minimum à `V+1` ; nous sommes ainsi assuré.e.s que le minimum est plus petit que cette valeur.
- On parcourt ensuite tous les indices `i` de 0 à `len(S)`, et on effectue le test `if (S[i]<=x) and (1+L[x-S[i]]<minimum)` pour déterminer la valeur du nouveau minimum car si ces conditions sont remplies, le minimum devient `1+L[x-S[i]]`. Dans ce cas, on précise que l'indice à garder est celui sur lequel on est : on le stocke dans la variable `v` pour plus tard.
- Une fois à l'extérieur du test conditionnel, pour la valeur d'argent `x` sur laquelle on est, nous stockons le minimum obtenu dans `L[x]` ainsi que l'indice `v` obtenu dans `valeurs[x]`.
- À la sortie de la boucle itérative sur `x`, les listes `L` et `valeurs` sont ainsi remplies respectivement du minimum de pièces et billets qui composent chaque indice jusqu'à `V`.
  On initialise alors la variable `x` à `V` et une liste vide `R`, qui contiendra le rendu de monnaie, donc la décomposition.
- Pour remplir la liste `R`, il suffit de soustraire à `x` la valeur que l'on y insère, et cette valeur correspond à `S[valeurs[x]]`.
- On finit par retourner `L[V]`, à savoir le nombre minimum de pièces et billets nécessaires pour décomposer `V`, ainsi que la liste `R` nouvellement obtenue.

Le programme qui suit affiche `(2, [7, 7])`, qui représente bien la solution à notre problème :

- la solution optimale est composée de 2 pièces ;
- l'énumération des pièces à utiliser pour un montant de 14 € est : (7 €, 7 €).

Voir exercice 2 page 104 et exercices 14 à 18 pages 113 à 114.

Solution de l'exercice type 2

```
def NombreBottomUp(V):
    S = [1,7,9]
    L = [0]*(V+1)
    valeurs = [0]*(V+1)
    for x in range(1,V+1):
        minimum = V + 1
        for i in range(len(S)):
            if (S[i] <= x) and (1+L[x-S[i]] < minimum):
                minimum = 1 + L[ x - S[i] ]
                v = i
```

COURS

Solution de l'exercice type 2 (suite)

```
        L[x] = minimum
        valeurs[x] = v

    while x > 0:
        R.append( S[ valeurs[x] ] )
        x -= S[ valeurs[x] ]

    return L[V] , R

print( NombreBottomUp(14) )
```

Voir énoncé page 97

À RETENIR : principaux types de programmation

Désignation	Description	Compléments
Programmation impérative	Suites d'instructions exécutées par la machine.	Programmation intuitive.
Programmation fonctionnelle	Évite les effets de bords.	Pas de variables globales.
Diviser pour régner	Découpe un problème en plusieurs sous-problèmes.	Utilisation de la récursivité.
Programmation Orientée Objet (POO)	Considère un programme comme une collection d'objets en interaction.	Utilisation de classes.
Programmation dynamique	Optimise un programme en temps et en mémoire.	Utilisation de la forme récursive *Top Down* ou de la forme itérative *Bottom Up*.

1 QCM Vérification de connaissances

10 min Corrigé p. 115

Pour chacune des questions suivantes, plusieurs propositions de réponses sont faites dont une seule est correcte. Laquelle ?

1 On définit la classe `identite` de la manière suivante :

```
class identite:
    def __init__(self , nom , prenom , an):
        self.nom = nom
        self.prenom = prenom
        self.an = an

    def age(self, a):
       return a - self.an
```

Si on tape :

```
>>> hugo = identite('Hugo' , 'Dupont' , 1999)
>>> print( hugo.age(2020) )
```

que s'affiche-t-il ?

a 'Hugo' b 'Dupont' c 1999 d 21

2 On définit la classe `voiture` de la manière suivante :

```
class voiture:
    def __init__(self , nom , couleur , puissance):
        self.nom = nom
        self.couleur = couleur
        self.puissance = puissance

    def nc(self):
        return nom + couleur
```

Si on tape :

```
>>> F430 = voiture('Ferrari' , 'Rouge' , 43)
>>> print(F430.nc())
```

que s'affiche-t-il ?

a 'Ferrari Rouge' b Une erreur

INTERROS

3 Qu'affiche le programme suivant ?

```
class toto:
    def __init__(self , a , b , c):
        self.a = a
        self.b = b
        self.c = c

    def is_square(self):
        if self.a**2 + self.b**2 == self.c**2:
            return True
        else:
            return False

t = toto(3,4,5)

print(t.is_square())
```

[a] Une erreur [b] True [c] False

2 V/F Vérification des connaissances

10 min Corrigé p. 115

Les affirmations suivantes sont-elles vraies ou fausses ? Justifier la réponse.

1 *La programmation dynamique.*

(a) La mémoïsation est une forme itérative.

(b) La programmation dynamique consiste à résoudre des problèmes, de celui qui a la taille la plus petite vers celui qui a la taille la plus grande.

(c) Un problème d'optimisation où la variable peut être réelle peut être résolu en utilisant la programmation dynamique.

(d) Le problème du rendu de monnaie est un problème classique qui ne peut être résolu que par programmation dynamique.

2 *La Programmation Orientée Objet (POO).*

(a) Les variables d'une classe sont appelées des *méthodes*.

(b) Dans une classe, la définition d'une méthode `__init__` n'est pas obligatoire.

Classes

3 La classe « Piece »

★ 5 min Corrigé p. 116

Voici un programme en Python :

```
import random

class Piece :
    def alea(self) :
        return random.randint(0,1)

    def moyenne(self, n):
        tirage = [ ]
        for i in range (n) :
            tirage.append( self.alea() )

        return sum(tirage) / n

p = Piece()
print( p.moyenne(100) )
```

Expliquez en détail ce qu'il permet d'afficher.

4 Écrire une classe

★ 10 min Corrigé p. 116

Pour chacune des questions suivantes, un constructeur initialisant les mesures devra être présent dans chacune des classes.

1. Écrire une classe nommée `Carre` admettant la mesure des côtés d'un carré en attribut, avec deux méthodes :
   - une méthode `perimetre` permettant de retourner le périmètre du carré ;
   - une méthode `aire` permettant de retourner son aire.
2. Écrire une classe `Triangle` représentant un triangle et admettant la mesure des trois côtés comme attributs, avec deux méthodes :
   - une méthode `aire` renvoyant l'aire du triangle à l'aide de la formule de Héron :
   $$\mathcal{A} = \sqrt{p(p-a)(p-b)(p-c)},$$
   où a, b et c sont les longueurs des trois côtés et où $p = \dfrac{a+b+c}{2}$;
   - une méthode `rect` qui renvoie `True` si le triangle est rectangle, et `False` sinon.
3. Définir alors un carré de côté 5, puis afficher son aire et son périmètre. Définir ensuite un triangle de côtés 5.4, 9 et 7.2 pour afficher son aire et regarder s'il est rectangle.

INTERROS

5 Devinette et remplissage

★ 10 min Corrigé p. 118

On considère la classe `mystere` définie de la manière suivante :

```
class mystere:
    def __init__(self):
        self.Liste = [ ]

    def add(self, x):
        if len(self.Liste) == 0
        or x >= self.Liste[ len(self.Liste)-1 ]:
            self.Liste.append(x)
        elif x < self.Liste[0]:
            self.Liste = [x] + self.Liste
        else:
            for i in range( len(self.Liste) ):
                if x > self.Liste[i]
                and x <= self.Liste[i+1]:
                    self.Liste = self.Liste[:i+1] + [x]
                                 + self.Liste[i+1:]
                    break
```

1 Que fait le programme suivant ?

```
L = mystere()

for i in [78,89,10,50,7]:
    L.add(i)

print(L.Liste)
```

2 Ajoutez à cette classe une méthode `ecc` qui crée et renvoie une liste dont les éléments sont les cumuls de ceux de l'attribut `Liste` de la classe.
Par exemple, si `Liste = [1,2,4,5]`, `ecc` doit retourner :

`[1, 3 ,7, 12]`.

6 La classe « Complexe »

★ 20 min Corrigé p. 119

En mathématiques, dans un plan rapporté à un repère orthonormé $(O;\vec{u},\vec{v})$, tout point M de coordonnées $(x;y)$ peut être représenté par ce que l'on nomme un *nombre complexe*, qui peut s'écrire sous la forme :

$$z = x + iy.$$

On dit que x est la *partie réelle* de z et y, sa *partie imaginaire*.

En posant $z = x + \mathrm{i}y$ et $z' = x' + \mathrm{i}y$, on définit alors les opérations suivantes :

- $z + z' = (x + x') + \mathrm{i}(y + y')$
- $z - z' = (x - x') + \mathrm{i}(y - y')$
- $z \times z' = (xx' - yy') + \mathrm{i}(xy' + x'y)$

De plus, on dit que $z = z'$ si $x = x'$ et $y = y'$.

Écrire en Python une classe `complexe` :

- qui définit un nombre complexe (le constructeur devra initialiser un tuple de deux nombres : la partie réelle et la partie imaginaire ;
- ayant une méthode permettant d'afficher le nombre complexe sous forme d'un tuple de deux éléments ;
- permettant d'ajouter, soustraire, multiplier et comparer (en terme d'égalité) deux nombres complexes ;
- permettant de donner la distance de l'origine du repère au point représenté par le nombre complexe (on appelle cette distance le *module*, qui est égal à $\sqrt{x^2 + y^2}$.

Tester cette classe avec les nombres :

$$z = -3 + 5\mathrm{i} \qquad \text{et} \qquad z' = 7 + \mathrm{i}.$$

7 Une classe « Polynôme »

★★ — 20 min — Corrigé p. 120

En s'inspirant du corrigé de l'exercice 8 page 83 sur l'algorithme de Karatsuba, écrire en Python une classe `Polynome` permettant de faire une addition, une soustraction, une division de deux polynômes.

On écrira de plus une méthode `affiche` permettant d'afficher un polynôme.

8 La classe « Temps »

★★ — 25 min — Corrigé p. 122

En Python, écrire une classe `Temps` qui permet de définir un horaire au format hh : mm : ss et qui admet les méthodes suivantes :

- `affiche`, qui affiche l'horaire au format « 12 h 37 min 45 s » ;
- `__add__`, qui ajoute deux horaires de la classe `Temps` ;
- `__sub__`, qui calcule la différence entre deux horaires de la classe `Temps`.

9 La classe « Fraction »

★★ — 25 min — Corrigé p. 123

En Python, écrire une classe `Fraction` qui vérifie les instructions page suivante.

INTERROS

```
>>> x = Fraction(12,9)
>>> x.num
12
>>> x.denom
9
>>> x = Fraction(12,9).simplify()
>>> x.num , x.denom
4,3
>>> y = Fraction(48,36)
>>> x == y
True
```

Aides :

- pour la simplification de fraction, on s'appuiera sur le résultat mathématique suivant :

$$\text{fraction : } \frac{a}{b} \longrightarrow \text{ fraction simplifiée : } \frac{a \div \text{pgcd}(a\,;\,b)}{b \div \text{pgcd}(a\,;\,b)}$$

- on pourra s'aider de l'exercice 4 page 51 pour le calcul du PGCD ;
- pour redéfinir le test d'égalité, on pourra redéfinir de façon interne à la classe la méthode `__eq__`.

10 La classe « Vecteur »

★★ 30 min Corrigé p. 124

En Python, écrire une classe `Vecteur` admettant trois attributs (ses coordonnées dans l'espace) comportant :

- une méthode permettant d'afficher les coordonnées du vecteur sous la forme $(x\,;\,y\,;\,z)$;
- une méthode renvoyant le vecteur somme de deux vecteurs (en redéfinissant `__add__`) ;
- une méthode renvoyant la norme du vecteur ;
- une méthode renvoyant le produit scalaire de deux vecteurs ;
- une méthode permettant de vérifier si un vecteur est colinéaire à un autre (passé en attribut) ;
- une méthode permettant de vérifier si un vecteur est orthogonal à un autre (passé en attribut).

11 La classe « SerieStat »

★★ 30 min Corrigé p. 124

Écrire en Python une classe `SerieStat` remplissant le cahier des charges suivant :

- un objet de cette classe est un tableau statistique comportant les caractères étudiés ainsi que leur effectif. Il sera implémenté à travers un dictionnaire ;
- une méthode `affiche` qui affiche toutes les valeurs de la série ;
- une méthode `moyenne` permettra de calculer la moyenne de cette série ;
- une méthode `var` permettra de calculer la variance de la série ;
- une méthode `ecart` permettra de calculer son écart-type.

12 Classes « Mot » et « Phrase »

★★ 30 min Corrigé p. 126

On considère la classe `Mot` définie ainsi :

```
from random import shuffle

class Mot:
  def __init__(self,m):
    self.m = m

  def doReverse(self):
    self.m = self.m[::-1]

  def doShuffle(self):
    L = list(self.m)
    shuffle(L)
    self.m = ''.join(L)

  def value(self):
    return self.m
```

1 Indiquer ce que fait la fonction `fctA()` définie par :

```
def fctA():
  m = Mot("Socrate")
  m.doReverse()
  print(m.value())
```

2 Indiquer ce que fait la fonction `fctB()` définie par :

```
def fctB():
  m = Mot("Socrate")
  m.doShuffle()
  print(m.value())
```

INTERROS

On souhaite écrire une classe `Phrase`. Toutes les questions suivantes porteront sur cette classe.

3 Écrire un constructeur qui définit une liste `self.mots` remplie de tous les mots de la phrase passée en paramètre, chacun des mots devant être de classe `Mot`.

4 Écrire deux méthodes `doReverse` et `value` telles que la fonction `fctC()` suivante :

```
def fctC():
  p = Phrase("Tous les hommes sont mortels")
  p.doReverse()
  print(p.value())
```

affiche :

« mortels sont hommes les Tous »

5 Écrire une méthode `doShuffle` afin que la fonction suivante :

```
def fctD():
  p = Phrase("Tous les hommes sont mortels")
  p.doShuffle()
  print(p.value())
```

affiche les mots de la phrase « Tous les hommes sont mortels » dans un ordre aléatoire.

6 On définit la méthode `motAt` de la manière suivante :

```
def motAt(self,pos):
  return self.mots[pos]
```

Que fait la fonction `fctE()` suivante ?

```
def fctE():
  p = Phrase("Tous les hommes sont mortels")
  m = p.motAt(3)
  m.doReverse()
  print(p.value())
```

7 On définit la méthode `insert` de la manière suivante :

```
def insert(self,pos,chaine):
  self.mots.insert(pos,Mot(chaine))
```

Que fait la fonction `fctF()` suivante ?

```
def fctF():
  p = Phrase("Tous les hommes sont mortels")
  p.insert(3,"ne")
  p.insert(5,"pas")
  print(p.value())
```

8 On définit la méthode `remove` de la manière suivante :

```
def remove(self,pos):
    self.mots.pop(pos)
```

Que fait la fonction `fctG()` suivante ?

```
def fctG():
  p = Phrase("Tous les hommes sont mortels")
  m = p.motAt(4)
  m.doShuffle()
  p.remove(2)
  p.insert(2,m.value())
  print(p.value())
```

13 Une autre classe « polynôme »

★★★ 30 min Corrigé p. 128

Dans cet exercice, on se propose de construire une classe nommée `polynome` représentant un polynôme, c'est-à-dire une expression de la forme :

$$a_0 + a_1X + a_2X^2 + \cdots + a_nX^n.$$

Dans cet exemple, n est appelé le *degré* du polynôme, et $(a_0, a_1, \ldots, a_n)$ sont ses coefficients.

Les termes a_kX^k (pour $0 \leqslant k \leqslant n$) de l'expression sont appelés des *monômes*.

On souhaite pouvoir ajouter, soustraire et multiplier deux polynômes à l'aide de la classe `polynome`.

Pour faciliter le travail, nous allons devoir créer une deuxième classe nommée `monome` permettant de définir un monôme avec son coefficient et son degré.

1 *La classe `monome`.*

- Écrire le constructeur de la classe `monome` qui permet d'initialiser les variables d'instance `coef` et `deg`.
  Par exemple, `monome(5,2)` devra représenter le monôme $5X^2$.

INTERROS

- Écrire une méthode `affiche` qui retourne un affichage correct du monôme. On devra notamment éviter les affichages :

  → « 1X^1 », lui préférant « X »,

  → « 3X^0 », lui préférant « 3 »,

  → ou encore « 0X^2 », auquel cas on ne doit rien afficher.

- Écrire ensuite trois méthodes `__add__`, `__sub__` et `__mul__` permettant respectivement d'ajouter deux monômes de même degré, de soustraire deux monômes de même degré et de multiplier deux monômes entre eux.
  Pour rappel,

  → $5X^2 + 3X^2 = (5+3)X^2$,

  → $(5X^2) \times (3X^4) = (5 \times 3)X^{2+4}$.

2 *La classe `polynome`.*

- Écrire le constructeur de la classe `polynome` qui permet d'initialiser une variable d'instance `ListeMonomes`, liste de tous les monômes composant le polynôme, du monôme de plus bas degré à celui de plus haut degré.
  Par exemple, `polynome(3,-2,0,1)` devra représenter le polynôme $3 - 2X + 0X^2 + X^3$.
  Aide : le constructeur devra être de la forme `__init__(self,*args)`, dans lequel `args` sera la liste de tous les coefficients.
- Écrire une méthode `affiche` retournant une chaîne de caractères correspondant à l'affichage du polynôme. On veillera à éviter l'affichage « $+1 + X + \cdots$ » (par exemple), lui préférant « $1 + X + \cdots$ » (car le premier signe n'est pas nécessaire à l'affichage si le premier coefficient est positif).
- Écrire une méthode `standardise` qui retourne deux polynômes ayant le même nombre de monômes.
  Par exemple, `P.standardise(Q)` devra retourner `P` et `Q` de sorte que `P.ListeMonomes` et `Q.ListeMonomes` sont deux listes de même longueur.
- Écrire alors trois méthodes `__add__`, `__sub__` et `__mul__` permettant respectivement d'ajouter, de soustraire et de multiplier deux polynômes entre eux.
  Aide : pour le produit de deux polynômes, il faudra penser à la distributivité (produit de monômes) et à la réduction (somme de monômes).

3 *Test de la classe.*

Tester la classe `polynome` avec les deux polynômes suivants :

$$P(X) = 1 + 2X + 3X^2 \qquad \text{et} \qquad Q(X) = -2 + 5X + 7X^3$$

en les ajoutant, soustrayant et multipliant.

Programmation dynamique

14 Coefficients binomiaux

★ 15 min Corrigé p. 132

Pour tout entier naturel n et pour tout entier naturel $k \leqslant n$, on note $\binom{n}{k}$ le nombre d'ensembles de k éléments pris parmi n. Une propriété mathématique affirme l'égalité suivante :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

De plus, pour tout entier naturel n, $\binom{n}{0} = \binom{n}{n} = 1$.

En utilisant une programmation dynamique, écrire en Python un programme affichant le coefficient binomial $\binom{n}{k}$ pour un n et un k donné.

Pour cela, on pourra s'aider du *triangle de Pascal* suivant, permettant de visualiser tous les coefficients binomiaux.

n \ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Début du triangle de Pascal

15 Suite de Fibonacci : le retour

★★ 15 min Corrigé p. 133

Nous avons vu dans la correction de l'exercice 6 page 60 que le programme sur la page suivante permettait de calculer tous les termes de la suite de Fibonacci (F_n) de F_0 à F_{20}.

```
def fibo(n):
    if n == 0 or n == 1: return 1
    else: return fibo(n-1) + fibo(n-2)
for i in range(21):
    print(fibo(i))
```

Adaptez ce programme à l'aide de la programmation dynamique et de la mémoïsation afin de le rendre plus efficace.

INTERROS

16 Modélisation d'une suite numérique ★★ 20 min Corrigé p. 134

On considère la suite numérique (u_n) définie par ses deux premiers termes $u_0 = 1$, $u_1 = 2$ et par la relation de récurrence suivante :

$$\forall\, n \in \mathbb{N}, \quad u_{n+2} = \frac{u_n}{u_{n+1}}.$$

Écrire un programme Python permettant de calculer u_n pour un entier n donné en utilisant la programmation dynamique (version Bottom Up et Top Down).

17 Somme dans une suite numérique ★★★ 30 min Corrigé p. 135

On définit une suite (u_n) par son premier terme $u_0 = 1$ et par la relation de récurrence :

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \frac{1}{n+1}\sum_{k=0}^{n}(k+1)u_k.$$

1. Calculer u_1, u_2 et u_3.
2. Proposer deux programmes permettant de calculer u_n pour tout entier naturel n en utilisant le paradigme de la programmation dynamique (version Bottom Up et Top Down).

18 Le retour du problème du sac à dos ★★★ 30 min Corrigé p. 136

Nous avons vu dans l'exercice 9 page 84 ce en quoi consiste le problème du sac à dos. Nous en avons d'ailleurs donné une solution récursive de type « diviser pour régner ».

1. Proposer une approche dynamique de type Bottom Up et Top Down de ce problème qui affiche uniquement la valeur maximale des objets.
   Vérifier les programmes avec l'exemple où le poids maximal est $P = 10$ et où la collection d'objets est $E = \{(5,7); (3,1); (3,4); (3,2)\}$.
2. Compléter l'approche Bottom Up (par exemple) afin d'afficher la combinaison des objets retenus.

1 QCM Vérification de connaissances

Enoncé p. 103

1 Réponse **d**. En effet, la méthode calcule la différence entre la valeur de l'argument `a` et la valeur de `self.an`, qui correspond à l'année (sans doute de naissance) de Hugo. Il s'agit donc ici de $2020 - 1999 = 21$.

2 Réponse **b**. En effet, dans la déclaration de la méthode `nc`, les variables `nom` et `couleur` ne sont pas définies. Il aurait fallu écrire `self.nom + self.couleur` pour que cela soit correct.

Rappelons que la déclaration d'une méthode peut faire appel :

- à des variables locales (qui doivent être initialisées avant utilisation et qui sont « détruites » à la fin de la méthode) ;
- à des attributs de l'objet lui-même qui « appartiennent » à l'objet, qui sont partagés entre toutes les méthodes, et qui subsistent en mémoire jusqu'à ce que l'objet lui-même soit détruit. Dans ce cas, il faut y faire référence en utilisant le paramètre `self`, qui est le premier argument de toute méthode.

3 Réponse **b**. Ici, tout est correctement défini. Ainsi, la méthode `is_square` regarde si $a^2 + b^2$ est bien égal à c^2, ce qui est le cas pour notre exemple. « True » est donc affiché.

2 V/F Vérification des connaissances

Enoncé p. 104

1 *La programmation dynamique.*

(a) *Faux*. La mémoïsation, ou Top Down, est une forme récursive.

(b) *Faux*. La technique décrite dans la question est celle du Bottom Up. La programmation dynamique, quant à elle, consiste à découper un problème en plusieurs sous-problèmes de tailles inférieures et en utilisant l'une des techniques Bottom Up ou Top Down pour les résoudre.

(c) *Faux*. La programmation dynamique ne sert que lorsque la variable est entière.

(d) *Faux*. Nous avons vu dans le cours qu'il existait d'autres façons que la programmation dynamique pour résoudre le problème du rendu de monnaie, comme l'approche gloutonne ou diviser pour régner. Cependant, la programmation dynamique a pour objectif d'optimiser le code. C'est donc cette technique qui est préférable aux autres car elle nécessite moins d'opérations de calcul et de traitement que les autres.

2 *La Programmation Orientée Objet (POO).*

(a) *Faux*. Une *méthode* est une *fonction* d'une classe. Les *attributs* sont les variables de la classe.

(b) *Vrai*. On peut tout à fait définir une classe sans la méthode `__init__`. En voici un exemple :

```
class toto:
    b = 5
    def fct(self,a):
        return a * self.b

d = toto()
print( d.fct(10) )
```

qui renvoie « 50 ».
Le constructeur `__init__` est rendu obligatoire dès lors qu'un paramètre est passé lors de l'appel de la classe (par exemple, `d = toto(7)`) ou qu'il est nécessaire d'initialiser certains attributs dès la création de l'objet.

3 La classe « Piece »

Enoncé p. 105

La classe `Piece` est composée de deux méthodes.

- La méthode `alea` retourne simplement un nombre aléatoire de l'ensemble {0; 1}.
- La méthode `moyenne`, qui admet un attribut `n`, commence par définir une liste vide nommée `tirage`. Ensuite, une boucle itérative est créée, pour 100 itérations. Au cours de chacune d'elles, on insère dans la liste `tirage` le retour de la méthode `alea` (on insère donc « 0 » ou « 1 » *n* fois).
  Une fois la boucle finie, on retourne la valeur de « `sum(tirage) / n` », c'est-à-dire la somme des nombres contenus dans la liste `tirage` divisée par *n* : il s'agit ici du calcul du nombre moyen de « 1 » obtenus lors des *n* choix aléatoires.

Ainsi, dans ce contexte, si on décide d'associer « Pile » à « 1 » lors du lancer d'une pièce, le programme calcule le nombre moyen de « Pile » sur 100 lancers.

4 Écrire une classe

Enoncé p. 105

live 1

```
class Carre:
    def __init__(self,a):
        self.cote = a
    def perimetre(self):
        return self.cote*4
    def aire(self):
        return self.cote**2
```

2 Une classe possible est la suivante :

```
class Triangle:
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a, b, c

    def aire(self):
        p = (self.a + self.b + self.c)/2
        return (p*(p-self.a)*(p-self.b)*(p-self.c))
    **0.5

    def rect(self):
        L = [self.a , self.b , self.c]
        L.sort()

        return L[2]**2 == L[0]**2 + L[1]**2
```

3 Définissons dans une console le carré et le triangle demandés.

```
>>> C = Carre(5)
>>> C.aire() , C.perimetre()
(25, 20)
>>> T = Triangle(5.4 , 9 , 7.2)
>>> (T.aire() , T.rect())
(19.44000000000001, True)
```

On voit ainsi que le triangle défini est rectangle et que son aire vaut 19,44.

Remarque : on pourrait améliorer légèrement l'affichage dans le cas où le triangle est rectangle.

En effet, dans ce cas précis, pour calculer l'aire du triangle, on peut utiliser la formule $\frac{b \times h}{2}$, où b et h sont respectivement une base du triangle et la hauteur relative à cette base.
Ainsi, dans la méthode `aire`, on pourrait écrire le code page ci-contre.

À RETENIR

Quel que soit le contexte d'un programme, il ne faut pas s'éloigner de ses règles. Si le programme concerne des mathématiques alors il faut utiliser les propriétés mathématiques pour optimiser le code.

```
class Triangle:
    def __init__(self,a,b,c):
        self.cotes = [a , b, c]
        self.cotes.sort()

    def rect(self):
        return self.cotes[2]**2 == self.cotes[0]**2
    + self.cotes[1]**2

    def aire(self):
        if self.rect():
            return self.cotes[0] * self.cotes[1] / 2
        else:
            p = (self.cotes[0] + self.cotes[1] +
    self.cotes[2])/2
            return (p*(p-self.cotes[0])*(p-self.
    cotes[1])*(p-self.cotes[2]))**0.5
```

5 Devinette et remplissage

Enoncé p. 106

1 `L.Liste` est *a priori* une liste (d'après la ligne : `self.Liste = [ ]`). Ainsi, l'affichage sera une liste. Regardons maintenant de plus près la définition de la fonction `add` :

- avant tout, si l'argument `x` est supérieur ou égal au dernier élément de la liste ou si la liste est vide, on ajoute la valeur de `x` à la liste ;
- sinon, si `x` contient une valeur strictement plus petite que le premier élément de la liste, on place la valeur de `x` en tête de la liste ;
- dans les autres cas, on parcourt la liste et on teste si la valeur contenue dans `x` est comprise entre deux valeurs de la liste. Si tel est le cas, on insère cette valeur entre les deux autres valeurs.

On peut alors dire que la fonction `add` ajoute la valeur de l'argument tout en faisant en sorte que la liste soit rangée dans l'ordre croissant.

Ainsi, le programme proposé affichera :

```
[7, 10, 50, 78, 89]
```

2 Voici une proposition pour la méthode `ecc` :

```
def ecc(self):
  E, prev = [], 0
  for nb in self.Liste:
    prev = prev + nb
    E.append(prev)
  return E
```

Ici, on commence par créer une liste `E` dont le seul élément est le premier de la liste `self.Liste`, puis on parcourt `self.Liste` à partir du deuxième élément : pour chaque nombre d'index `i`, on lui ajoute `E[i-1]` et on stocke le résultat à la fin de la liste `E`, donc dans `E[i]`.
La boucle terminée, on retourne `E`, qui est la liste de tous les cumuls.

Remarque : en statistiques, on appelle ces nombres les *effectifs cumulés croissants*.

Téléchargez le programme complet.

6 La classe « Complexe »

Enoncé p. 106

Voici une classe possible :

```
class complexe:
    def __init__(self,re,im):
        self.z = (re , im)
    def nombre(self):
        return (self.z[0] , self.z[1])
    def __add__(self , other):
        return complexe( self.z[0] + other.z[0] , self.
    z[1] + other.z[1] )
    def __sub__(self , other):
        return complexe( self.z[0] - other.z[0] , self.
    z[1] - other.z[1] )
    def __mul__(self , other):
        return complexe( self.z[0] * other.z[0] - self.
    z[1] * other.z[1] , self.z[0] * other.z[1] + other.z
    [0] * self.z[1] )
    def __eq__(self , other):
        return self.z[0] == other.z[0] and self.z[1] ==
     other.z[1]
    def module(self):
        return (self.z[0] ** 2 + self.z[1] ** 2) ** 0.5
```

En testant avec les nombres $z = -3 + 5\mathrm{i}$ et $z' = 7 + \mathrm{i}$, cela donne :

```
>>> z , v = complexe(-3,5) , complexe(7,1)
>>> print(" +  = ".format(z.nombre() , v.nombre() ,
    (z+v).nombre() ) )
(-3, 5) + (7, 1) = (4, 6)
```

```
>>> print(" -  = ".format(z.nombre() , v.nombre() ,
    (z-v).nombre() ) )
(-3, 5) - (7, 1) = (-10, 4)
>>> print(" *  = ".format(z.nombre() , v.nombre() ,
    (z*v).nombre() ) )
(-3, 5) * (7, 1) = (-26, 32)
>>> print("Le module de  est ".format( z.nombre() ,
    z.module() ) )
Le module de (-3, 5) est 5.830951894845301
>>> print(" est-il égal à  ? ".format(z.nombre() ,
    v.nombre() , z == v))
(-3, 5) est-il égal à (7, 1) ? False
```

---

7 Une classe « Polynôme »

Enoncé p. 107

Voyons étape par étape la construction d'une telle classe :

- Le constructeur.

```
def __init__(self,*args):
    self.coef = [i for i in args]
```

On initialise ici une liste : celle des coefficients informés en paramètre.

- La méthode `affiche`.

```
def affiche(self):
    r = str(self.coef[0]) + ' '
    for i in range( 1 , len(self.coef) ):
        if i != 1:
            end = '^' + str(i) + ' '
        else:
            end = ' '

        if self.coef[i] < 0:
            r = r + str(self.coef[i]) + 'X' + end
        elif self.coef[i] > 0:
            r = r + '+ ' + str(self.coef[i]) + 'X'
            + end

    print(r)
```

On commence par initialiser une chaîne de caractères `r` par le premier coefficient (transformé pour l'occasion en chaîne de caractères avec la fonction `str`) et en lui adjoignant un espace à la fin.
On parcourt ensuite tous les degrés possibles pour chaque terme du polynôme, donc de 1 jusqu'au degré maximal. Pour chaque degré différent de 1, on définit l'affichage de l'exposant par « ^ » suivi du degré. Pour le cas où l'exposant est 1, on n'affiche rien (car $X^1 = X$).
Si le coefficient est négatif, on ajoute à `r` ce coefficient (avec le signe « − ») suivi de « X^ degré » ; si le coefficient est positif, il faut penser à ajouter un « + » avant le « X ».
Nous avons choisi de clore cette méthode par un affichage de `r`, mais on aurait pu simplement retourner sa valeur.

- La méthode `standardise`.

```
def standardise(self , Q):
    n , m = len(self.coef) , len(Q.coef)
    if n < m:
        self.coef = self.coef + [0] * (m - n)
    else:
         Q.coef = Q.coef + [0] * (n - m)

    return self.coef , Q.coef
```

Dans cette méthode, on complète la liste des coefficients des deux polynômes la plus courte pour qu'elle ait une longueur égale à la plus grande : on ajoute donc à la liste la plus courte autant de « 0 » que nécessaire (la différence entre les deux degrés des polynômes).

- Somme et différence de deux polynômes.

```
def __add__(self , Q):
    P , Q = self.standardise(Q)
    return Polynome( *[ P[i] + Q[i] for i in range(
    len(P) ) ] )

def __sub__(self , Q):
    P , Q = self.standardise(Q)
    return Polynome( *[ P[i] - Q[i] for i in range(
    len(self.coef) ) ] )
```

Dans les deux méthodes, on standardise les deux polynômes afin de pouvoir ajouter (ou soustraire) les éléments d'indice `i` des deux listes. Il est à noter ici la présence de l'astérisque devant les listes ainsi obtenues : cela permet de transformer les listes en énumérations convenables.

En effet, `Polynome([1,2,3])` n'est pas la bonne syntaxe pour définir un polynôme ; il faut donc transformer `[1,2,3]` en tuple `1,2,3`. On appelle cela l'*unpacking*.

- Produit de deux polynômes.

```
def prodmonome(self, k):
    L = [0] * k + self.coef
    return Polynome(*L)

def __mul__(self , Q):
    n = max( len(self.coef) , len(Q.coef) )
    if n == 1:
        return Polynome(*[self.coef[0] * Q.coef[0]])

    P , Q = self.standardise(Q)
    k = n // 2
    P0 , P1 = Polynome(*P[0:k]) , Polynome(*P[k:n])
    Q0 , Q1 = Polynome(*Q[0:k]) , Polynome(*Q[k:n])

    return P0*Q0 + ((P0+P1)*(Q0+Q1)-P0*Q0-P1*Q1).
    prodmonome(k) + (P1*Q1).prodmonome(2*k)
```

Afin de pouvoir utiliser l'algorithme de Karatsuba, nous devons définir une méthode `prodmonome` qui se charge de multiplier un polynôme par X^k, puis nous pouvons définir le produit en utilisant les formules de l'algorithme.

Ainsi, avec $P(X) = 1 + 2X + 3X^2$ et $Q(X) = 4 + 5X + 6X^2$, on obtient :

```
>>> P , Q = Polynome(1,2,3) , Polynome(4,5,6)
>>> (P*Q).affiche()
4 + 13X + 28X^2 + 27X^3 + 18X^4
```

Télécharger la classe complète

8 La classe « Temps »

Enoncé p. 107

Une définition de la classe `Temps` est donnée page suivante.

Avec cette définition, voici ce que l'on peut obtenir :

```
>>> t1, t2 = Temps(8,45,37), Temps(15,31,18)
>>> (t1+t2).affiche(), (t1-t2).affiche()
'24 h 16 min 55 s' , '6 h 45 min 41 s'
```

```
class Temps:
    def __init__(self,h,m,s):
        self.h, self.m, self.s = h, m, s

    def affiche(self):
        return '{} h {} min {} s'.format(str(self.h),
    str(self.m), str(self.s))

    def __add__(self,t):
        sec = self.s + t.s
        rs = sec % 60
        minute = self.m + t.m + sec//60
        rm = minute % 60
        rh = self.h + t.h + minute//60
        return Temps(rh,rm,rs)

    def __sub__(self,t):
        a = self.h*3600 + self.m*60 + self.s
        b = t.h*3600 + t.m*60 + t.s
        diff = abs(b-a)
        h = diff // 3600
        m = (diff-3600*h)//60
        s = diff-3600*h-60*m
        return Temps(h,m,s)
```

9 La classe « Fraction »

Enoncé p. 107

```
class Fraction:
    def __init__(self,a,b):
        self.num, self.denom = a , b
    def pgcd(self,a,b):
        if b == 0: return a
        else:
            r = a % b
            return self.pgcd( b , r)
    def simplify(self):
        a = self.num // self.pgcd(self.num , self.denom)

        b = self.denom // self.pgcd(self.num , self.
    denom)
        return Fraction(a,b)
    def __eq__(self,y):
        return self.num * y.denom == self.denom * y.num

x , y = Fraction(12,9) , Fraction(4,3)
print(x == y)
```

CORRIGÉS

10 La classe « Vecteur »

Enoncé p. 108

Une définition de la classe `Vecteur` est la suivante.

```
class Vecteur:
    def __init__(self,x,y,z):
        self.x, self.y, self.z = x, y, z
    def affiche(self):
        return '({};{};{})'.format(str(self.x),str(self.
    y),str(self.z))
    def __add__(self,v):
        return Vecteur( self.x + v.x , self.y + v.y ,
    self.z + v.z )
    def __mul__(self,v):
        return self.x*v.x + self.y*v.y + self.z*v.z
    def norme(self):
        return (self.x**2 + self.y**2 + self.z**2)**0.5
    def is_colin(self,v):
        return self.x/v.x == self.y/v.y == self.z/v.z
    def is_ortho(self,v):
        return self*v == 0
```

Avec cette classe, on peut par exemple obtenir :

```
>>> u, v = Vecteur(-1,2,1) , Vecteur(3,5,-7)
>>> u.affiche(), (u+v).affiche()
'(-1;2;1)' , '(2;7;-6)'
>>> u.norme()
2.449489742783178
>>> u.is_ortho(v)
True
>>> u.is_colin(v)
False
```

11 La classe « SerieStat »

Enoncé p. 109

Pour les calculs, il est nécessaire de faire quelques rappels mathématiques. $(x_i ; n_i)$ désigne la série statistique à N éléments.

- $\bar{x} = \dfrac{\sum_{i=1}^{N} x_i \times n_i}{\sum_{i=1}^{N} n_i}$ (moyenne)

- $V = \dfrac{\sum_{i=1}^{N}(x_i - \bar{x})^2 \times n_i}{\sum_{i=1}^{N} n_i}$ (variance)
- $\sigma = \sqrt{V}$ (écart-type)

Remarque : la variance est donc la moyenne de la série initiale dans laquelle nous avons remplacé x_i par $(x_i - \bar{x})^2$... Cela va nous intéresser pour l'implémentation.

```
class SerieStat:
    def __init__(self,C,E):
        # C et E sont deux listes de mêmes dimensions
        if len(C) != len(E): print('Error : lists must
    have same dimensions')
        else: self.tab = dict(zip(C, E))
    def affiche(self):
        for key,value in self.tab.items():
            print(key,":",value)
    def moyenne(self):
        m,s = 0,0
        for key,value in self.tab.items():
            m += key * value
            s += value
        return m / s
    def var(self):
        R , F = [] , []
        m = self.moyenne()
        for key,value in self.tab.items():
            R.append( (key - m)**2 )
            F.append( value )
        return SerieStat(R,F).moyenne()
    def ecart(self):
        return self.var()**0.5
```

MÉTHODE

`zip(C,E)` permet de créer une association entre les éléments `i` des listes `C` et `E`. Pour mettre ces associations sous forme de dictionnaire, nous avons utilisé la fonction `dict`.

Nous pouvons regarder ce que cela donne à travers un exemple.

```
>>> from random import randint
>>> C = range(21)
>>> E = [randint(0,20) for i in C]
>>> serie = SerieStat(C,E)
>>> serie.affiche()
S'affiche ici le dictionnaire
>>> print('Moyenne : ',serie.moyenne())
Moyenne : 8.401041666666666
>>> print('Variance : ',serie.var())
Variance : 33.21937391493056
>>> print('Ecart-type : ',serie.ecart())
Ecart-type : 5.76362506717175
```

12 Classes « Mot » et « Phrase »

Enoncé p. 109

1 L'instruction « `m = Mot("Socrate")` » définit dans un premier temps un objet `Mot` comme étant la chaîne de caractères « Socrate ».
Ensuite, « `m.doReverse()` » fait appel à la méthode `doReverse` de la classe `Mot` qui, comme son nom l'indique, transforme la chaîne de caractères en inversant les lettres.
Enfin, « `print(m.value())` » affiche ce qui est contenu dans `self.m` de la classe `Mot`, donc affiche ici : « etarcoS » (le mot écrit à l'envers).

2 Comme dans la fonction précédente, on commence par définir un objet de classe « Mot », puis on fait appel à la méthode `doShuffle` qui retourne une anagramme du mot.
La fonction `fctB()` affiche donc une anagramme de « Socrate ».

3 Un constructeur possible est le suivant :

```
class Phrase:
  def __init__(self,phrase):
    L = phrase.split()
    self.mots = []
    for m in L:
      self.mots.append(Mot(m))
```

On commence par définir une liste `L` composée de tous les mots de la phrase passée en paramètre avec la méthode `split` : à ce stade, les mots se sont que des chaînes de caractères. Or, nous souhaitons que chaque mot soit de classe `Mot`. On parcourt alors cette liste et, à partir de chaque item, on définit un objet de classe `Mot` que l'on insère dans une liste `self.mots` (« self » est indispensable pour lier cette nouvelle liste à l'objet de classe `Phrase` ainsi défini).

4 Une proposition de méthodes est la suivante :

```
def doReverse(self):
  self.mots = self.mots[::-1]

def value(self):
  result  = ''
  for m in self.mots:
    result += m.value() + ' '
  return result
```

Dans la méthode `doReverse`, nous n'avons fait qu'inverser l'ordre des items de la liste `self.mots` avec l'opération `[::-1]`, et dans la méthode `value`, nous avons initialisé une chaîne de caractères `result` comme étant vide, puis parcouru chaque item de la liste `self.mots` pour concaténer à `result` la chaîne de caractères correspondant à l'item (`m.value()`) en y ajoutant un espace afin qu'au final, `result` représente la phrase.

5 Ici, nous pouvons utiliser à nouveau la fonction `shuffle` du module `random`, ce qui donne :

```
def doShuffle(self):
  shuffle(self.mots)
```

6 Dans la fonction `fctE()`, on commence par définir une phrase, puis un mot `m` comme étant l'objet `Mot` en position 3 dans la phrase définie. Ici, il s'agit donc de « sont » (attention : le premier mot est à la position 0, comme dans toute liste). Ensuite, on applique la méthode `doReverse` de la classe `Mot` au mot ainsi trouvé, ce qui a pour effet de mettre ses lettres dans le sens inverse.
On affiche ensuite la phrase ainsi obtenue, ce qui donne :

Tous les hommes tnos mortels

7 Dans la fonction `fctE()`, on commence par définir une phrase, puis on fait appel à la méthode `insert` qui insère le mot « ne » dans la phrase en position 3, c'est-à-dire après le 3e mot de la phrase. On fait de même en position 5, c'est-à-dire après le 5e mot : on y insère le mot « pas ». Et enfin, on affiche la phrase ainsi obtenue, ce qui donne :

Tous les hommes ne sont pas mortels

8 On commence par prendre le mot en position 4 dans la phrase, donc « mortels ». On mélange ensuite les lettres dans ce mot en faisant appel à la méthode `doShuffle` de la classe `Mot`. Ensuite, on fait appel à la

méthode `remove` de la classe `Phrase` en passant en paramètre le nombre 2 : on supprime alors le mot « hommes ». Enfin, on insère en position 2 le contenu de la variable `m`, c'est-à-dire l'anagramme de « mortels » obtenu précédemment. On affiche alors le résultat, ce qui donne par exemple :

Tous les elmsort sont elmsort

Remarque : le dernier mot n'est plus « mortels » ; en effet, nous avons définit `m` comme l'objet en position 4 dans l'objet `p`. Dès lors que l'on écrit `m.doShuffle`, cet objet est modifié dans l'objet `p`. Ainsi, à ce stade, l'objet « mortels » n'existe plus mais il est remplacé par une anagramme de la chaîne de caractères qu'il représente.

Télécharger le programme complet.

13 Une autre classe « polynôme »

Enoncé p. 111

1 *La classe `monome`.*

- Le constructeur.

```
def __init__(self,coef,deg):
    self.coef = coef
    self.deg = deg
```

On initialise les deux variables d'instance demandées : `coef` (le coefficient du monôme) et `deg` (pour le degré du monôme).

- La méthode `affiche`.

```
def affiche(self):
    aff = ''
    if self.deg == 0 and abs(self.coef) != 1:
        X = ''
  elif self.deg == 0:
    X = '1'
    elif self.deg == 1:
        X = 'X'
    else:
        X = 'X^' + str(self.deg)
    if self.coef < 0 and self.coef != -1:
        aff = str(self.coef) + X
  elif self.coef == -1:
    aff = '-' + X
    elif self.coef > 0 and self.coef != 1:
        aff = '+' + str(self.coef) + X
    elif self.coef == 1:
        aff = '+' + X
    return aff
```

On commence par initialiser une chaîne de caractères vide `aff` (qui contiendra l'affichage souhaité). Pour que l'affichage soit correct, on fait d'abord la différence entre un monôme de degré 0, 1 et supérieur. Ensuite, on regarde le coefficient : s'il est strictement négatif, on l'affiche avec son signe ; s'il est positif mais différent de 1, on ajoute le signe « + » avant le coefficient et s'il est égal à 1, on affiche un « + » sans le coefficient.

Remarquez qu'il n'y a pas la condition « `self.coef == 0` » car dans ce cas, il ne faut rien afficher, donc on laisse la variable `aff` vide.

- Les opérations (addition, soustraction, multiplication).

Pour l'addition et la soustraction, on commence par vérifier que les degrés des deux monômes sont égaux car on souhaite que ces méthodes portent sur deux monômes de mêmes degrés.
On définit alors le résultat comme le monôme dont le coefficient est la somme ou la différence des coefficients des deux monômes ajoutés ou soustraits, et dont le degré est le degré des monômes (on rappelle que l'on a : $aX^p + bX^p = (a+b)X^p$).
Quant au produit, il est définit par le monôme dont le coefficient est le produit des coefficients des monômes et dont le degré est la somme des degrés des monômes ($aX^p \times bX^q = (a \times b)X^{p+q}$).

```
def __add__(self , other):
    if self.deg != other.deg: return False
    else: return monome(self.coef + other.coef ,
    self.deg)

def __sub__(self , other):
    if self.deg != other.deg: return False
    else: return monome(self.coef - other.coef ,
    self.deg)

def __mul__(self , other):
    return monome(self.coef * other.coef , self.
    deg + other.deg)
```

2 *La classe `polynome`.*

- Le constructeur.

```
def __init__(self,*args):
    self.ListeMonomes = [ monome( args[d] , d )
    for d in range(len(args)) ]
```

Nous avons utilisé ici une syntaxe concise pour construire la liste demandée : cette liste est composée de tous les monômes (`monome( args[d] , d )`) pour tout degré `d` compris entre 0 et la longueur de la liste `args` (longueur qui correspond au degré du polynôme).

- La méthode `affiche`.

```
def affiche(self):
    r , prems = '' , True
    for m in self.ListeMonomes:
        r += m.affiche()
        if prems == True and m.coef > 0:
            r = r[1:]
        prems = False
    return r
```

La variable `r` est une chaîne de caractère correspondant à l'affichage désiré. Quant à `prems`, c'est un booléen qui ne sert qu'à supprimer le signe « + » devant le premier coefficient s'il est positif.

- La méthode `standardise`.

```
def standardise(self , other):
    if len(self.ListeMonomes) !=
            len(other.ListeMonomes):
        if len(self.ListeMonomes) == max( len(self.
    ListeMonomes) , len(other.ListeMonomes) ):
            for i in range(len(self.ListeMonomes)-
    len(other.ListeMonomes)):
                other.ListeMonomes += [ monome(0 ,
     len(other.ListeMonomes) + i) ]
        else:
            for i in range(len(other.ListeMonomes)-
    len(self.ListeMonomes)):
                self.ListeMonomes += [ monome(0 ,
    len(self.ListeMonomes) + i) ]

    P = [m.coef for m in self.ListeMonomes]
    Q = [m.coef for m in other.ListeMonomes]
    return polynome(*P) , polynome(*Q)
```

L'idée consiste ici à comparer la longueur des listes `self.ListeMonomes` et `other.ListeMonomes`, prendre le maximum de ces deux longueurs et de compléter la liste la plus courte par des monômes de coefficients 0.

Ensuite, on crée deux listes de coefficients `P` et `Q` pour pouvoir définir deux polynômes à partir de ces deux listes et les retourner.

- La somme de deux polynômes.

```
def __add__(self , other):
    P , Q = self.standardise(other)
    R = []
    for d in range( len(P.ListeMonomes) ):
        R += [ P.ListeMonomes[d]
               + Q.ListeMonomes[d] ]

    return polynome(*[ m.coef for m in R ])
```

On commence pas standardiser les polynômes afin de pouvoir effectuer les sommes des différents monômes et mettre les résultats dans une liste `R`.
On retourne ensuite un polynôme construit à partir des coefficients des monômes de la liste `R`.

- La différence de deux polynômes.

```
def __sub__(self , other):
    P , Q = self.standardise(other)
    R = []
    for d in range( len(P.ListeMonomes) ):
        R += [ P.ListeMonomes[d]
               - Q.ListeMonomes[d] ]

    return polynome(*[ m.coef for m in R ])
```

Le principe est le même que pour la somme.

- Le produit de deux polynômes.
On commence par parcourir tous les monômes du premier polynôme et pour chacun d'eux, on parcourt tous ceux du second polynôme afin de les multiplier ; on construit ainsi une liste `R` composée de tous les produits des monômes (résultat du développement en utilisant la distributivité).

```
def __mul__(self , other):
    R , R_final = [] , []
    for m in self.ListeMonomes:
        for mm in other.ListeMonomes:
            R += [ m * mm ]
```

(Suite du script page suivante)

CORRIGÉS

```
        for degre in range(len(self.ListeMonomes) +
        len(other.ListeMonomes)):
            result = monome(0,degre)
            for m in [ monome for monome in R if
        monome.deg == degre ]:
                result += m
            R_final.append(result)

        return polynome(*[ m.coef for m in R_final])
```

On crée ensuite une boucle de variable `degre` (qui varie de 0 à la somme des degrés des polynômes, car le produit de deux polynômes de degré n et m est un polynôme de degré $n+m$). Pour chacun des degrés, on initialise la variable `result` comme étant un monôme de coefficient 0 et de degré `degre`, puis on lui ajoute tous les monômes de même degré ; la liste :

`[ monome for monome in R if monome.deg == degre ]`

est composée uniquement des monômes dont le degré est égal à la valeur de la variable `degre` courante.
Une fois la boucle en `m` finie, nous avons ajouté tous les monômes de degré `degre` ; nous pouvons alors ajouter ce résultat à la liste `R_final`, qui contiendra (au final) tous les monômes composant le produit des polynômes.
On retourne alors le polynôme construit à partir de cette liste (en ne prenant que les coefficients des monômes).

3 Avec $P(X) = 1 + 2X + 3X^2$ et $Q(X) = -2 + 5X + 7X^2$, on obtient :

```
>>> P, Q = polynome(1, 2, 3), polynome(-2, 5, 7)
>>> A, S, M = P + Q, P - Q, P * Q
>>> print( '(P+Q)(X)=, (P-Q)(X)=, (P*Q)(X)='.format(
     A.affiche(),S.affiche(),M.affiche() ) )
(P+Q)(X)=-1+7X+3X^2+7X^3,
(P-Q)(X)=3-3X+3X^2-7X^3,
(P*Q)(X)=-2+X+4X^2+22X^3+14X^4+21X^5
```

Téléchargez le programme complet.

14 Coefficients binomiaux

Enoncé p. 113

Pour tout entier naturel n, $\binom{n}{0} = \binom{n}{n} = 1$.

Donc $\binom{0}{0} = 1$, $\binom{1}{0} = 1$ et $\binom{1}{1} = 1$.

Pour l'implémentation, on commence par initialiser un tableau rempli de 0.

Ensuite, on insère les coefficients binomiaux ligne par ligne à l'aide d'une boucle itérative (en i) :

- si $i = 0$, c'est-à-dire si nous sommes sur la 1re ligne, on fixe le coefficient à 1 ;
- sinon, on parcourt toutes les colonnes de la ligne i : si c'est la 1re ($p = 0$), on fixe le coefficient à 1 et sinon, on calcule le coefficient à l'aide de la formule de récurrence.

```
def coefbin(n,k):
    T = [[0] * (n+1) for p in range(n+1)]
    for i in range(n+1):
        if i == 0:
            T[0][0] = 1
        else:
            for p in range(i+1):
                if p == 0:
                    T[i][p] = 1
                else:
                    T[i][p] = T[i-1][p-1] + T[i-1][p]
    return T[n][k]
```

15 Suite de Fibonacci : le retour

Enoncé p. 113

MÉTHODE

La mémoïsation consiste à optimiser un code en mémorisant les différents termes calculés.

Nous allons donc placer chaque terme calculé dans un tableau afin de pouvoir y faire appel quand on en a besoin.

En effet, dans le programme donné, pour calculer F_{20} (par exemple), il faut calculer F_{19} et F_{18}. Or, pour calculer F_{19}, il faut calculer F_{18} et F_{17}. Ainsi, à ce stade, F_{18} est calculé deux fois. Et ce n'est rien comparé aux termes d'indices petits, qui sont appelés bien plus souvent pour les termes d'indices supérieurs.

De plus, la complexité de la fonction `fibo(n)` est ici exponentielle ; en effet, elle est égale à $O(\varphi^n)$, où $\varphi = \dfrac{1+\sqrt{5}}{2}$ est le *nombre d'or*.

Cela donne le programme page suivante.

```
F = [1,1]
def fibo(n):
    for i in range(len(F),n+1):
        F.append(F[i-2] + F[i-1])

    return F[n]

for i in range(21):
    print( fibo(i) )
```

La fonction `fibo` ne permet pas de trouver directement le terme de rang n. Elle doit être appelée sur chaque terme, depuis le premier jusqu'à n, comme ce qui est fait dans l'énoncé, pour compléter progressivement `F`.

On complète donc `F` à la demande, seulement si nécessaire, depuis son dernier terme jusqu'au terme de rang n souhaité.

Dans ce dernier programme, la complexité de la fonction `fibo(n)` est rendue linéaire.

16 Modélisation d'une suite numérique

Enoncé p. 114

Une version Bottom Up du programme dynamique est la suivante :

Bottom Up

```
def u(n):
    U = [ 0 ] * (n+1)
    for i in range(n+1):
        if i == 0: U[i] = 1
        elif i == 1: U[i] = 2
        else:
            U[i] = U[i-2] / U[i-1]
    return U[n]
```

Une version optimisée (en espace mémoire) est la suivante :

Bottom Up (optimisée en espace mémoire)

```
def u(n):
    u,v = 1,2
    for i in range(2,n+1):
        if n == 0: return u
        elif n == 1: return v
        else:
            w = u / v
            u = v
            v = w
    return w
```

Une version Top Down est la suivante :

Top Down

```
def u(n , U = None):
    if n == 0: return 1
    elif n == 1: return 2
    else:
        U = [None] * (n+1)
        U[0] , U[1] = 1 , 2
        return u_cache(U,n)

def u_cache(U,n):
    if U[n] == None:
        U[n] = u(n-2 , U) / u(n-1 , U)
    return U[n]
```

Une manière plus simple peut aussi être envisagée :

Top Down

```
def u(n , U = None):
  if not (U) :
    U = [1,2] + [None] * (n-1)
  if U[n] == None:
    U[n] = u(n-2 , U) / u(n-1 , U)
  return U[n]
```

17 Somme dans une suite numérique

Enoncé p. 114

1 Bien que mathématique, et non informatique, cette question permet de comprendre le mode de génération des termes de la suite définie.

- $u_1 = u_{0+1} = \dfrac{1}{0+1}\displaystyle\sum_{k=0}^{0}(k+1)u_k = u_0 = 1$;
- $u_2 = u_{1+1} = \dfrac{1}{1+1}\displaystyle\sum_{k=0}^{1}(k+1)u_k = \dfrac{1}{2}(1u_0 + 2u_1) = \dfrac{3}{2}$;
- $u_3 = u_{2+1}$

$$= \frac{1}{2+1}\sum_{k=0}^{2}(k+1)u_k$$
$$= \frac{1}{3}(1u_0 + 2u_1 + 3u_2)$$
$$= \frac{1}{3} \times \left(3 + \frac{9}{2}\right) = \frac{5}{2}.$$

2 La difficulté de ce genre de calcul réside dans la sommation. Il faut penser à la stocker au même titre que les termes de la suite.

Bottom Up

```
def u(n):
    U = [None] * (n+1)
    U[0] = (1,0)
    for k in range(1,n+1):
        S = U[k-1][1] + k * U[k-1][0]
        U[k] = (S/k , S)
    return U[n][0]

for n in range(20):
    print ( u(n) )
```

L'idée ici est donc de stocker non pas un terme mais un couple constitué du terme u_k et de la somme $u_0 + 2u_1 + \cdots + (k+1)u_k$ pour pouvoir calculer par la suite u_{k+1}.

Top Down

```
def u(n , U = None):
    if not U:
        U = [(1,0)] + [(None,None)] * n
    if U[n][0] == None:
        val = u(n-1 , U)[1] + n * u(n-1 , U)[0]
        U[n] = (val / n , val)
    return U[n]

for n in range(20):
    print( u(n)[0] )
```

18 Le retour du problème du sac à dos

Enoncé p. 114

1 Rappelons avant tout la fonction récursive initiale du problème :

```
def sac(E,P,i):
    if i == 0: return 0
    if E[i-1][0] > P: return sac(E , P , i-1)
    else: return max( sac(E , P , i-1) , E[i-1][1] +
                      sac(E , P-E[i-1][0] , i-1) )

def initSac(E,P):
    return sac(E , P , len(E))
```

- *Approche dynamique de type « Top Down ».*

  Pour améliorer cette fonction récursive, il nous faut mémoriser les solutions aux sous-problèmes (mémoïsation). Nous allons donc stocker les valeurs de `V[i][p]`, avec $0 \leqslant i \leqslant n$ et $0 \leqslant p \leqslant P$, où P est le poids maximal autorisé, dans une liste nommée par exemple `temp`.
  La solution est très proche de l'initiale.

```
def sac(E,P,i,temp):
    if temp[i][P]>0: return temp[i][P]
    if i == 0: return 0
    if E[i-1][0] > P:
        temp[i][P] = sac(E , P , i-1 ,temp)
        return temp[i][P]
    else:
        temp[i][P] = max( sac(E , P , i-1 , temp) ,
     E[i-1][1] + sac(E , P - E[i-1][0] , i-1 , temp
    ) )
        return temp[i][P]

def initSac(E,P):
    temp = [ [0] * (P+1) for i in range( len(E)+1 )
     ]
    return sac(E , P , len(E) , temp)
```

  Ce qui donne pour notre exemple :

```
>>> E, P = [ (5,7) , (3,1) , (3,4) , (3,2) ] , 10
>>> initSac(E,P)
11
```

  Cela signifie que la valeur maximale des objets que l'on peut mettre dans le sac est égale à 11 ($7 + 4$, avec un poids de $5 + 3 = 8$).

- *Approche dynamique de type « Bottom Up ».*

```
def sac(E,P):
    temp = [ [0] * (P+1) for i in range( len(E)+1 )
    ]
    for i in range(1,len(E)+1):
        for p in range(P+1):
            if E[i-1][0] > p:
                temp[i][p] = temp[i-1][p]
            else:
                temp[i][p] = max( temp[i-1][p] , E
    [i-1][1] + temp[i-1][ p - E[i-1][0] ] )
    return temp[len(E)][P]
```

Avec cette approche, il n'y a plus de récursivité mais une itération.
On garde à peu près la même structure pour la fonction `sac`.

On obtient bien entendu le même résultat, à savoir « 11 ».

2 Comme nous pouvons le voir ici, ces solutions n'affichent que la valeur optimale. Le problème du sac à dos étant d'obtenir la combinaison des objets retenus, il nous faut compléter ces solutions.
Si on complète l'approche Bottom up, cela donne le programme suivant :

```
def sac(E,P):
    temp = [ [0] * (P+1) for i in range( len(E)+1 )]
    garde  = [ [False] * (P+1) for i in range( len(E
    )+1) ]
    for i in range(1,len(E)+1):
        for p in range(P+1):
            if E[i-1][0] > p:
                temp[i][p] = temp[i-1][p]
            else:
                if E[i-1][1] + temp[i-1][p-E[i
    -1][0]] > temp[i-1][p]:
                    temp[i][p] =  E[i-1][1] + temp[i
    -1][ p - E[i-1][0] ]
                    garde[i][p] = True
                else:
                    temp[i][p] = temp[i-1][p]
    p,L = P,[]
    for i in range(len(E),0,-1):
        if garde[i][p]:
            L.append(i)
            p -= E[i-1][0]

    return temp[len(E)][P] , L
```

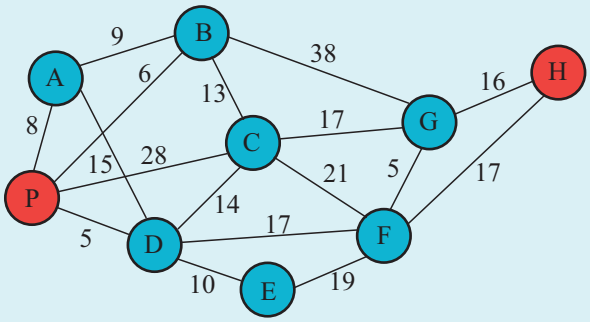
Chapitre 5

Les graphes : structures relationnelles

Plan du chapitre

Exercice type

On considère le graphe pondéré suivant :



Implémenter ce graphe en Python, puis trouver le chemin le plus court de P à H.

Voir corrigé page 158

PROBLÉMATIQUE

La gestion des réseaux, qu'ils soient routiers, électriques ou sociaux, est un enjeu majeur de l'informatique.
Comment représenter de telles structures ?

COURS

1 Notion de graphes

Définition 1

Un *graphe* $G = (V, E)$ est la donnée :

- d'un ensemble V d'éléments, appelés *sommets* (appelés aussi *nœuds*) ;
- d'un ensemble E dont les éléments sont des parties à un ou deux éléments de V, nommés *arêtes* ou *arcs*.

Remarque : nous avons utilisé les lettres V et E pour respectivement *Vertex* (« sommet ») et *Edge* (« fil », sous-entendu : le fil reliant deux sommets).

Un graphe est utilisé pour représenter le lien entre divers objets ou entités.

Exemple : $V = \{A; B; C\}$ et $E = \big\{(A,B); (A,C); (C)\big\}$.
Dans ce cas, $G = (V, E)$ est le graphe de sommets A, B et C, où A et B sont reliés, A et C sont reliés et où C est connecté à lui-même.

Définition 2

La *représentation sagittale* d'un graphe $G = (V, E)$ est un schéma où les éléments de V sont représentés par des disques et où les éléments de E sont représentés par des traits (rectilignes ou courbés).

Exemple : une représentation sagittale du graphe défini dans l'exemple précédent est la suivante :

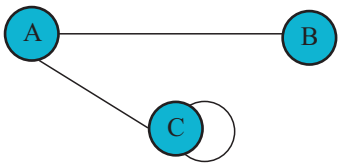


Figure 5.1 – Représentation sagittale du graphe $G = (V, E)$ avec $V = \{A; B; C\}$ et $E = \big\{(A,B); (A,C); (C)\big\}$

Par la suite, par souci de simplification, on confondra le graphe et sa représentation sagittale.

Définitions 3 : vocabulaire

- L'*ordre* d'un graphe est le nombre de ses sommets.
- Un graphe est dit *cyclique* s'il existe un chemin reliant n'importe quel sommet à lui-même.
- Un graphe est dit *connexe* si ses sommets peuvent être reliés deux à deux par une arête ou un arc.

2 Les différents types de graphes

Il existe deux types de graphes : *orientés* et *non orientés*.

2.1 Graphes non orientés

Définition 4

Un graphe *non orienté* est un graphe dont les *arêtes* n'ont pas de sens.

Cela signifie que si $G = (V,E)$ est un graphe avec $V = \{A; B; C\}$ alors les éléments (A,B) et (B,A) de E sont identiques.

Exemple : trois villes A, B et C sont reliées par un réseau routier :

- A et B sont reliées par une route ;
- A et C sont reliées par une route ;
- B et C sont reliées par une route.

On peut alors représenter ce réseau routier par le graphe suivant $G = (V,E)$ où :

$$V = \{A; B; C\} \text{ et } E = \big\{(A,B); (A,C); (B,C)\big\}$$

dont une représentation sagittale est :

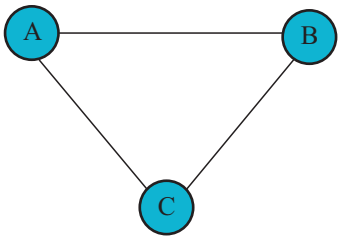


Figure 5.2 – Représentation sagittale d'un graphe non orienté

Ce graphe est d'ordre 3 (car il y a 3 sommets) ; il est de plus connexe et cyclique.

Remarque : la représentation sagittale d'un graphe n'est pas unique car on peut, par exemple, placer les sommets n'importe où.

Définitions 5

Dans un graphe non orienté,

- deux sommets reliés par une arête sont dits *adjacents* ;
- une arête ne possédant qu'un élément est une *boucle* ;
- un graphe qui ne comporte pas de boucle est qualifié de *simple*.

COURS

2.2 Graphes orientés

Définition 6

Un graphe *orienté* est un graphe dont les *arcs* ont un sens.

Remarque : cela signifie que dans le graphe $G = (V, E)$ où $V = \{A; B; C\}$, les éléments (A, B) et (B, A) sont distincts.

Exemple : imaginons un site internet composé de trois pages, nommées A, B et C.

- Sur A sont présents des liens vers B et C.
- Sur B figure un lien vers C.
- Sur C figure un lien vers A.

On peut représenter ceci à l'aide d'un graphe où les sommets sont A, B et C et où les arcs représentent les liens.

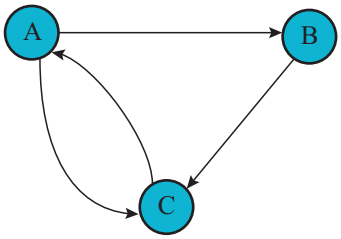


Figure 5.3 – Représentation sagittale d'un graphe orienté

À RETENIR

Un graphe non orienté a des *arêtes*.
Un graphe orienté a des *arcs*.

Définitions 7

Soit $G = (V, E)$ un graphe orienté. Si (A, B) est un arc de G alors :

- A est appelé le *prédécesseur* de B ;
- B est appelé le *successeur* de A.

Remarque : on dit aussi que pour un arc (A, B), A est son *origine* et B son *extrémité*.

2.3 Graphes pondérés

Définitions 8

On dit d'un graphe qu'il est *pondéré* (ou *valué*) quand ses arêtes ou ses arcs sont couplés avec des nombres. Ces nombres sont appelés des *poids*.

Exemple (graphe pondéré) : imaginons le cas d'un livreur dont le parcours peut être représenté par le graphe suivant :

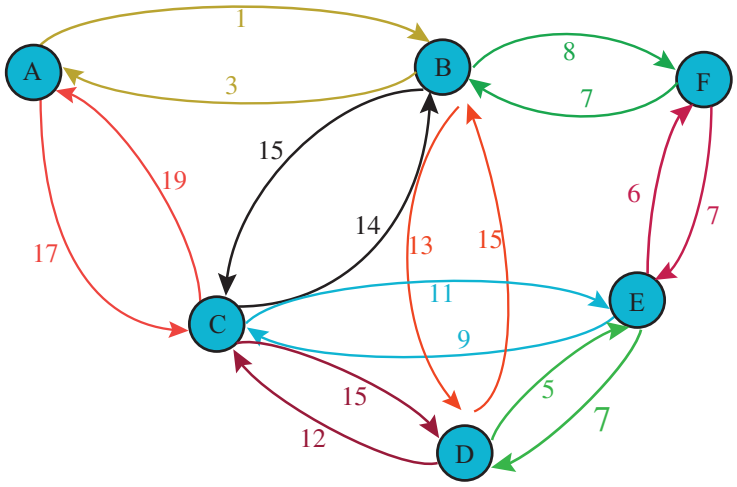


Figure 5.4 – Graphe du livreur

Les nombres attribués à chaque arc représentent ici le temps (en minutes) qu'il met pour aller d'un sommet à un autre.

Définition 9 : plus court chemin

Dans le cas d'un graphe pondéré, on appelle *plus court chemin* le chemin dont la somme des pondérations est minimale.

Exemple : dans l'exemple précédent, le plus court chemin de A à E est :

$$A - B - F - E$$

dont le poids total est : $1 + 8 + 7 = 16$.

3 Matrices de représentation

3.1 Matrice d'adjacence (graphe non pondéré)

Définition 10

Soit G un graphe d'ordre n. On appelle *matrice d'adjacence de G* la matrice carrée $M = (m_{ij})$ d'ordre n telle que :

- si G est non orienté,

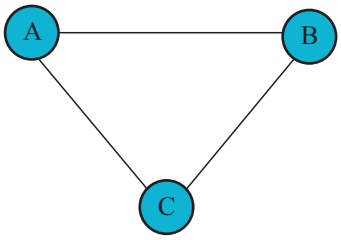
$$m_{ij} = \begin{cases} 1 & \text{si les sommets } i \text{ et } j \text{ sont adjacents} \\ 0 & \text{sinon} \end{cases}$$

COURS

- si G est orienté,

$$m_{ij} = \begin{cases} 1 & \text{si le sommet } i \text{ est l'origine d'un arc} \\ & \text{dont le sommet } j \text{ en est une extrémité} \\ 0 & \text{sinon} \end{cases}$$

Exemple :



Ce graphe admet pour matrice d'adjacence :

$$M = \begin{array}{c} \begin{matrix} A & B & C \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} A \\ B \\ C \end{matrix}$$

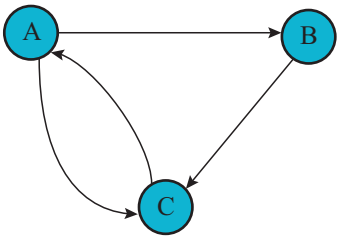
En effet,

- le sommet A n'est pas relié à lui-même (donc il y a un « 0 » à l'intersection de la 1re ligne et de la 1re colonne) ;
- les sommets A et B sont reliés (donc il y a un « 1 » à l'intersection de la 1re ligne et de la 2e colonne, ainsi qu'à l'intersection de la 1re colonne et de la 2e ligne ;
- etc.

Propriété 1

La matrice d'adjacence d'un graphe non orienté est nécessairement symétrique par rapport à sa diagonale principale (diagonale qui va du haut-gauche vers le bas-droit de la matrice).

Exemple (graphe orienté) :



Ce graphe admet pour matrice d'adjacence :

$$M = \begin{array}{c} \begin{matrix} A & B & C \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} A \\ B \\ C \end{matrix}$$

En effet,

- le sommet A n'est pas relié à lui-même (d'où le « 0 » à l'intersection de la 1re ligne et 1re colonne ;
- il y a un arc de A vers B, d'où le « 1 » à l'intersection de la 1re ligne et 2e colonne mais pas d'arc de B vers A, d'où le « 0 » à l'intersection de la 2e ligne et 1re colonne ;
- etc.

3.2 Matrice de valuation (graphe pondéré)

Définition 11

La *matrice de valuation* d'un graphe pondéré d'ordre n est la matrice $(m_{i,j})$ telle que :

- $m_{i,j}$ est le poids de l'arête (ou arc) du sommet i au sommet j ;
- $m_{i,j} = 0$ si aucune arête (ou arc) ne relie les sommets i et j.

Exemple : reprenons le graphe de l'exemple du livreur (figure 5.4 page 143).
Sa matrice de valuation est :

$$M = \begin{array}{c} \begin{matrix} A & B & C & D & E & F \end{matrix} \\ \begin{pmatrix} 0 & 1 & 17 & 0 & 0 & 0 \\ 3 & 0 & 15 & 13 & 0 & 8 \\ 19 & 14 & 0 & 15 & 11 & 0 \\ 0 & 15 & 12 & 0 & 5 & 0 \\ 0 & 0 & 9 & 7 & 0 & 6 \\ 0 & 7 & 0 & 0 & 7 & 0 \end{pmatrix} \end{array} \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix}$$

Voir exercices 1 à 3 pages 160 et 161.

4 Implémentation d'un graphe

4.1 Densité d'un graphe

Définition 12

La densité d'un graphe est égale au rapport du nombre d'arêtes sur le nombre total d'arêtes possibles.
La densité est donc un nombre compris entre 0 et 1.

COURS

Remarques

- Une densité nulle correspond à un graphe sans arêtes : tous les sommets sont isolés.
- Une densité égale à 1 correspond à un graphe complet : chaque sommet est relié à tous les autres.

Exemple : pour un graphe simple non orienté à n sommets, chaque sommet ne peut être relié qu'à $n-1$ sommets au maximum (car le graphe est simple, donc il n'y a pas de boucle). Il y a donc $n(n-1)$ arêtes au maximum.
La densité du graphe est alors :

$$d = \frac{2 \times \text{nombre d'arêtes}}{n(n-1)}.$$

En effet, si une arête est entre les sommets A et B, alors on compte deux fois cette arête dans le calcul de la densité (une pour A, et une pour B) car le graphe est non orienté.

Définition 13 : graphe creux, graphe denses

- Un graphe à n sommets est *creux* si son nombre d'arêtes « est proche » de n.
- Un graphe à n sommets est *dense* si son nombre d'arêtes « est proche » de n^2.

Propriété 2

Un graphe est dense si sa densité est « assez proche » de 1.
Un graphe est creux si sa densité est « assez éloignée » de 1.

Remarque : la notion de *graphe dense* n'est pas une notion précise. En effet, « assez proche » peut différer d'un individu à l'autre.
Aussi, dans la pratique, on pourra considérer qu'un graphe est dense si sa densité est au moins égale à 0,75 (tout en ayant à l'esprit que cette valeur est arbitraire).

Exemple : reprenons le graphe orienté 5.4 de la page 143.
Le nombre d'arêtes est égal à 18 et le nombre maximal d'arêtes est :

$$n(n-1) = 6 \times 5 = 30.$$

La densité de ce graphe est donc :

$$d = \frac{18}{30} = 0{,}6.$$

On peut alors estimer que le graphe est creux.

4.2 Graphes non pondérés

4.2.1 - Graphes denses : par matrice d'adjacence

Dans le cas des graphes denses, on attribue un booléen à chaque coefficient de la matrice.

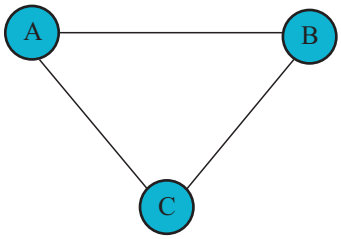
```
Pour i variant de 0 à n-1:
  Pour j variant de 0 à n-1:
    m[i][j] ← 1 si i et j sont reliés, 0 sinon
  Fin du Pour j
Fin du Pour i
```

L'un des avantages de cette représentation est sa simplicité : étant donnés deux sommets de rangs i et j, le booléen à la i-ième ligne et j-ième colonne représente l'existence ou l'absence d'une arrête les reliant.

Exemple : si nous reprenons le graphe de la figure 5.2 de la page 141, dont la matrice d'adjacence est $M = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$ comme vu précédemment, nous pouvons implémenter ce graphe en Python comme page ci-contre.

En Python

```
G = [
    [0,1,1] ,
    [1,0,1] ,
    [1,1,0]
]
```



La complexité est ici en $O(|S|^2)$, où $|S|$ est le nombre de sommets du graphe.

En effet, il y a deux boucles de n itérations, donc $n \times n = n^2$ affectations pour les coefficients de la matrice, et n^2 affectations pour i et j, soit en tout $2n^2$ affectations élémentaires.

Or, $n = |S|$; la complexité est donc bien en $O(n^2)$.

4.2.2 - Graphes creux : par liste d'adjacence

Définition 14

Soit $G = (V, E)$ un graphe tel que $V = \{A_1; A_2; \ldots; A_n\}$.
L'*ensemble d'adjacence* d'un sommet A_k, $1 \leqslant k \leqslant n$, est l'ensemble des A_p tels que A_k et A_p sont reliés par une arête (graphe non orienté) ou un arc (graphe orienté).

COURS

Dans le cas des graphes creux, on utilise l'*ensemble d'adjacence* pour l'implémentation. L'idée consiste alors, pour chaque sommet du graphe, à construire cet ensemble.

```
G ← liste vide
Pour k variant de 0 à n-1:
   L[k] est une liste vide
   Pour p variant de 0 à n-1:
     Si A_k est relié à A_p:
       On ajoute A_p dans la liste L[k]
     Fin du Si
   Fin du Pour p
   G ← G + (A_k , L[k])
Fin du Pour k
```

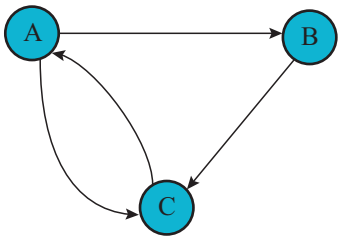
La complexité est ici en $O(|S|+|A|)$, où $|S|$ et $|A|$ sont respectivement le nombre de sommets et le nombre d'arêtes du graphe.

En Python, on pourra par exemple utiliser un dictionnaire.

Exemple : reprenons le graphe représenté sur la figure 5.3 page 142 en y ajoutant un sommet D, origine d'aucun arc. Nous pouvons l'implémenter comme suit :

En Python

```
G = {
    'A' : ['B','C'] ,
    'B' : ['C','D'] ,
    'C' : ['A','D'],
    'D' : [None]
}
```



Remarque : il existe plusieurs manières d'implémenter le sommet D n'est pas unique. Nous aurions aussi pu écrire par exemple :

```
'D' : [ ]
```

4.2.3 - À l'aide d'une classe

Un graphe peut être implémenté à l'aide d'une classe. Cela permet de définir par là même des méthodes de parcours.

Voyons page suivante ce que cela peut donner en définissant par liste d'adjacence le graphe non orienté de la figure 5.5 page 152.

Implémentation d'un graphe par une classe

```
class Graphe:
    def __init__(self):
        self.Liste = {}

    def addArete(self, u, v):
        if v not in self.Liste:
            self.Liste[v] = []
        if u not in self.Liste:
            self.Liste[u] = []
        self.Liste[u].append(v)
        self.Liste[v].append(u)

G = Graphe()
G.addArete('A', 'B')
G.addArete('A', 'D')
...
G.addArete('G', 'H')
for key,value in G.Liste.items():
    print(key,":",value)
```

Ce programme affiche :

```
B : ['A', 'C']                C : ['B', 'D']
A : ['B', 'D', 'E']           F : ['E', 'G']
D : ['A', 'C', 'E']           G : ['E', 'F', 'H']
E : ['A', 'D', 'F', 'G']      H : ['G']
```

On peut ensuite utiliser cette classe afin d'y implémenter des parcours.

Téléchargez la classe complète.

4.3 Graphes pondérés

4.3.1 - Par liste d'adjacence

On peut implémenter un graphe pondéré à l'aide d'un dictionnaire dont les valeurs sont aussi des dictionnaires.

Ainsi, pour le graphe de la figure 5.4 page 143, une implémentation possible est celle présentée page suivante (on ne met ici que le début du dictionnaire).

COURS

Implémentation d'un graphe pondéré en Python

```
G = {
    'A' : {
        'B' : 1,
        'C' : 17
    },
    'B' : {
        'A' : 3,
        'C' : 15,
        'D' : 13,
        'F' : 8
    },
    ...
}
```

Les clés du dictionnaire principal sont les sommets du graphe et les valeurs sont des dictionnaires où chaque clé représente un sommet connecté à la clé du dictionnaire principal et où chaque valeur est la pondération de l'arc.

Remarque : un sommet qui n'est l'origine d'aucun arc peut être représenté par un dictionnaire vide par exemple.

Implémenter un graphe pondéré peut alors se faire en créant une classe comme ceci :

Implémentation d'un graphe pondéré par liste d'adjacence

```
class GraphePond:
    def __init__(self):
        self.Liste = {}

    def addArc(self,u,v,p):
        if u not in self.Liste:
            self.Liste[u] = { v : p }
        else:
            self.Liste[u][v] = p
```

Cela donne en mode console :

```
>>> G = GraphePond()
>>> G.addArc('K','M',4)
>>> G.addArc('K','E',1)
>>> G.addArc('A','B',5)
>>> G.Liste
{'K': {'M': 4, 'E': 1}, 'A': {'B': 5}}
```

4.3.2 - Par matrice d'adjacence

Sur le même modèle que précédemment, on peut créer une classe qui définit un graphe quand on connaît sa matrice d'adjacence :

Implémentation d'un graphe pondéré par matrice d'adjacence

```
class GraphePondM:
    def __init__(self):
        self.Liste = []

    def add(self,L):
        self.Liste.append(L)
```

En mode console, on aura par exemple :

```
>>> G = GraphePondM()
>>> G.add([0,1,2])
>>> G.add([2,0,5])
>>> G.add([5,3,0])
>>> G.Liste
[[0, 1, 2], [2, 0, 5], [5, 3, 0]]
```

5 Algorithmique

5.1 Parcours d'un graphe

Parcourir un graphe consiste à lister tous ses sommets une seule fois. Il peut y avoir deux raisons principales au parcours d'un graphe : lister simplement tous les sommets ou trouver un chemin pour relier un sommet à un autre.

5.2 Parcours en largeur

L'idée est d'utiliser une file de la manière suivante :

1. on part d'un sommet ;
2. on visite ses voisins directs et on les enfile s'ils ne sont pas déjà présents dans la file (en les numérotant) ;
3. on défile (on enlève la tête de la file) ;
4. on recommence à partir du 2 tant que la file n'est pas vide.

COURS

Exemple : regardons par rapport au graphe suivant :

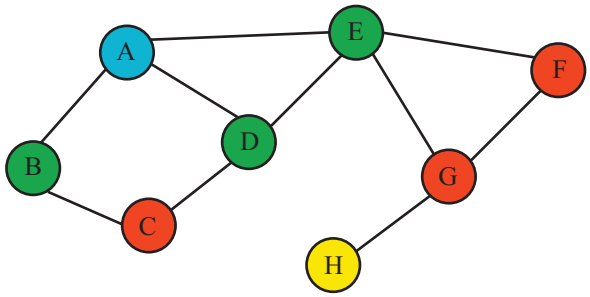


Figure 5.5 – Parcours en largeur d'un graphe

Nous souhaitons parcourir ce graphe en partant du sommet A.

- Les sommets B, D et E sont à une distance 1 de A ; ils sont donc enfilés en premier (peu importe l'ordre). On défile alors A.
- On enfile C, voisin de B et D, puis on défile B et D ; on enfile ensuite les sommets F et G, voisins de E (remarquez que les sommets C, G et F sont à une distance de 2 du sommet A), puis on défile E.
- F n'ayant pas de voisin dans la liste des sommets non traités, on le défile et on passe à G. Il a un voisin, H, que l'on enfile. On défile alors G, puis H (qui n'a pas de voisin dans la liste des sommets non traités). Remarquez que le sommet H est à une distance de 3 du sommet A.

Un algorithme possible de parcours en largeur est donc le suivant :

Parcours en largeur d'un graphe

```
F est une file vide
On enfile un sommet
Tant que F n'est pas vide:
   S ← tête de F
   On enfile les voisins de S
   On défile F
Fin du Tant que
```

Exemple : reprenons le graphe de la figure 5.5 (exemple précédent).

Parcours en largeur écrit en Python

```
from collections import deque
G = {
    'A' : ['B','D','E'] ,
    'B' : ['A','C'] ,
```

(suite du programme page suivante)

```
    'C' : ['B','D'] ,
    'D' : ['A','C','E'] ,
    'E' : ['A','D','G','F'] ,
    'F' : ['E','G'] ,
    'G' : ['E','F','H'] ,
    'H' : ['G']
}

sortie = []
file = deque()
file.append('A')

while file:
    S = file.popleft()
    if S not in sortie:
        sortie.append(S)
        unvisited = [n for n in G[S] if n not in sortie]
        file.extend(unvisited)

print(sortie)
```

Ce programme affiche :

```
['A', 'B', 'D', 'E', 'C', 'G', 'F', 'H']
```

Pour le cas de graphes implémentés par matrice d'adjacence, vous trouverez un exercice à ce sujet (exercice 9 page 164).

L'algorithme de parcours en largeur permet de dresser la liste des sommets d'un graphe par distance croissante au sommet d'origine. On peut ainsi s'en servir pour trouver un chemin de distance minimale entre deux sommets.

5.3 Parcours en profondeur

L'idée est d'utiliser une pile de la manière suivante :

1. on part d'un sommet que l'on empile ;
2. si le sommet de la pile a des voisins qui ne sont pas dans la pile et qui ne sont pas déjà passés dans la pile, on choisit l'un des ces voisins et on l'empile. Dans le cas contraire, on dépile (on supprime l'élément du haut de la pile) ;
3. on revient au point 2 tant que la pile n'est pas vide.

COURS

Exemple : le programme suivant est basé sur le même graphe que précédemment.

Parcours en profondeur écrit en Python

```
from collections import deque

G = {
    'A' : ['B','D','E'] ,
    'B' : ['A','C'] ,
    'C' : ['B','D'] ,
    'D' : ['A','C','E'] ,
    'E' : ['A','D','G','F'] ,
    'F' : ['E','G'] ,
    'G' : ['E','F','H'] ,
    'H' : ['G']
}

sortie = []
pile = deque()
pile.append('A')

while pile:
        S = pile.pop()
        if S not in sortie:
            sortie.append(S)
            non_visites = [n for n in G[S] if n not in
    sortie]
            pile.extend(non_visites)

print(sortie)
```

Il affiche : `['A', 'E', 'F', 'G', 'H', 'D', 'C', 'B']`

5.4 Recherche d'un chemin

On peut s'inspirer de l'un des parcours (en largeur ou en profondeur) pour élaborer un algorithme permettant de trouver un chemin d'un sommet à un autre dans un graphe.

Algorithme de recherche d'un chemin

```
G est un graphe
start ← sommet de départ
end ← sommet d'arrivée
P est une pile
```

Algorithme de recherche d'un chemin (suite)

```
Empiler le couple (start,[start])
Tant que P n'est pas vide:
   (S,path) ← tête de P
   list_nodes ← liste des sommets adjacents à S
                qui ne sont pas dans path
   Pour i dans list_nodes:
     Si i = end:
       Afficher path + [i]
     Sinon:
       Empiler (i,path+[i])
     Fin du Si
   Fin du Pour
Fin du Tant que
```

Le programme suivant donne par exemple tous les chemins qui mènent de A à H sans passer deux fois par le même sommet :

Chemins d'un sommet à un autre en Python

```
G = {
    'A' : ['B','D','E'] ,
    'B' : ['A','C'] ,
    'C' : ['B','D'] ,
    'D' : ['A','C','E'] ,
    'E' : ['A','D','G','F'] ,
    'F' : ['E','G'] ,
    'G' : ['E','F','H'] ,
    'H' : ['G']
}

start, end = 'A','H'

pile = []
pile.append((start, [start]))

while pile:
    (S, path) = pile.pop()
    list_nodes = [n for n in G[S] if n not in path]
    for i in list_nodes:
        if i == end:
            print(path + [i])
        else:
            pile.append((i, path + [i]))
```

COURS

Expliquons ce programme :

- on commence par définir le sommet d'origine et le sommet de destination ;
- on initialise ensuite une liste qui contiendra des couples de la forme :
  `(sommet de départ , [chemin à partir du sommet de départ])`
- tant que la pile n'est pas vide, on définit le couple `S,path` comme étant la tête de la pile, et on dépile ;
- on construit alors la liste `list_nodes` comme étant tous les sommets reliés à `S` qui ne sont pas déjà dans la liste `path` (représentant le chemin) ;
- on parcourt alors la liste `list_nodes` : si l'élément `i` sur lequel nous sommes est le sommet d'arrivée alors on affiche le chemin complet en y ajoutant `i`, qui est le sommet de destination ; sinon, on ajoute à la pile le couple composé de `i` accompagné du chemin auquel nous avons ajouté le sommet `i`.

Ce programme affiche alors :

```
['A', 'E', 'F', 'G', 'H']
['A', 'E', 'G', 'H']
['A', 'D', 'E', 'F', 'G', 'H']
['A', 'D', 'E', 'G', 'H']
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
['A', 'B', 'C', 'D', 'E', 'G', 'H']
```

5.5 Recherche d'un plus court chemin

Retrouvez en vidéo le principe de l'algorithme de Dijkstra.

Pour la recherche de plus courts chemins, l'algorithme le plus connu est l'algorithme de Dijkstra.

Il recherche autant de « plus courts chemins » qu'il existe de sommets accessibles depuis le sommet de départ. C'est pour cela qu'il ne reçoit pas, en entrée, le sommet de destination souhaité.

Dans cet algorithme,

- $G = (V, E)$ est un graphe pondéré ;
- `L` est une liste contenant les distances du sommet de départ vers les autres sommets ;
- `P` est une liste contenant le prédécesseur dans le chemin le plus court à partir du sommet de départ ;
- `M` est une liste de tous les sommets déjà traités ;
- `poids(A,B)` désigne le poids de l'arc ou arête qui va de A à B.

Algorithme de Dijkstra

```
*- initialisation -*
D ← sommet de départ
L[D] ← 0
P[D] ← D
Pour tout S (sommet) ≠ D:
   Si S est un successeur de D:
     L[S] ← poids(D,S)
     P[S] ← D
   Sinon:
     L[S] ← "0"
     P[S] ← ""
   Fin du Si
Fin du Pour
M ← S
*- Traitement -*
Tant que M ≠ V:
   Choisir S ∈ V \ M tel que ∀ J ∈ V \ M, L[S] ⩽ L[J]
   Pour J ∈ V \ M tel que J successeur de S:
     Si L[J] > L[S] + poids(S,J):
       L[J] ← L[S] + poids(S,J)
       P[J] ← S
     Fin du Si
   Fin du Pour
Ajouter à M le sommet S
Fin du Tant que
```

Téléchargez l'implémentation de Dijkstra en Python.

5.6 Recherche d'un cycle

Définition 15

Un *cycle* est un chemin dont l'origine et l'extrémité sont identiques.

Exemple : dans le graphe de la figure 5.5 page 152, le chemin :

A – B – C – D – A

est un cycle.

Pour obtenir tous les cycles d'un graphe, on peut s'inspirer de l'algorithme de recherche d'un chemin.

COURS

Voici ce que cela donne en Python :

Recherche d'un cycle en Python

```
from collections import deque

G = {
    'A' : ['B','D','E'] ,
    'B' : ['A','C'] ,
    'C' : ['B','D'] ,
    'D' : ['A','C','E'] ,
    'E' : ['A','D','G','F'] ,
    'F' : ['E','G'] ,
    'G' : ['E','F','H'] ,
    'H' : ['G']
}

for key in G:
    start = key
    pile = deque()
    pile.append((start, []))

    while pile:
        (S, path) = pile.pop()
        list_nodes = [n for n in G[S] if n not in path]
        for i in list_nodes:
            if i == start:
                print([start] + path + [i])
            else:
                pile.append((i, path + [i]))
```

Voir exercices 4 à 12 pages 162 à 166.

Solution de l'exercice type

Nous pouvons avant tout construire la matrice d'adjacence de ce graphe :

$$M = \begin{array}{c} \begin{matrix} P & A & B & C & D & E & F & G & H \end{matrix} \\ \begin{pmatrix} 0 & 8 & 6 & 28 & 5 & 0 & 0 & 0 & 0 \\ 8 & 0 & 9 & 0 & 15 & 0 & 0 & 0 & 0 \\ 6 & 9 & 0 & 13 & 0 & 0 & 0 & 38 & 0 \\ 28 & 0 & 13 & 0 & 14 & 0 & 21 & 17 & 0 \\ 5 & 15 & 0 & 14 & 0 & 10 & 17 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 & 0 & 19 & 0 & 0 \\ 0 & 0 & 0 & 21 & 17 & 19 & 0 & 5 & 17 \\ 0 & 0 & 38 & 17 & 0 & 0 & 5 & 0 & 16 \\ 0 & 0 & 0 & 0 & 0 & 0 & 17 & 16 & 0 \end{pmatrix} \end{array} \begin{matrix} P \\ A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{matrix}$$

Solution de l'exercice type (suite)

À l'aide de la classe page 151, on peut alors implémenter le graphe. Mais on peut aussi utiliser le programme « `dijkstra` » fourni page 157.

Téléchargez le programme.

Les sommets sont représentés par des nombres (ici, de 0 à 8) et on cherche le plus court chemin de 0 à 8.
Ce programme affiche :

```
Plus courts chemins de...
Longueur 5 : 0 -> 4
Longueur 6 : 0 -> 2
Longueur 8 : 0 -> 1
Longueur 15 : 0 -> 4 -> 5
Longueur 19 : 0 -> 4 -> 3
Longueur 22 : 0 -> 4 -> 6
Longueur 27 : 0 -> 4 -> 6 -> 7
Longueur 39 : 0 -> 4 -> 6 -> 8
```

Ainsi, le chemin le plus court est : P – D – F – H.

Voir énoncé page 139

INTERROS

1 QCM Vérification de connaissances

5 min

Corrigé p. 167

On considère le graphe $\mathcal{G}$ dont la représentation sagittale est la suivante :

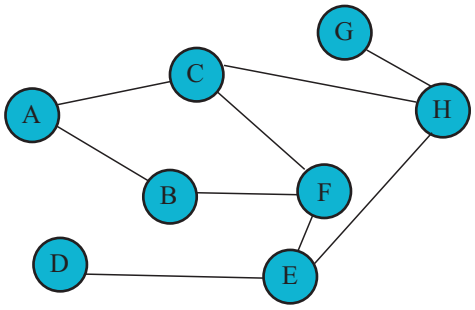


Figure 5.6 – Représentation sagittale du graphe $\mathcal{G}$

Pour chacune des questions suivantes, plusieurs propositions de réponses sont faites dont une seule est correcte. Laquelle ?

1 L'ordre de $\mathcal{G}$ est égal à :

a 9 **b** 8

2 Le graphe $\mathcal{G}$ est :

a orienté **b** non orienté

3 Le graphe $\mathcal{G}$ est :

a cyclique **b** non cyclique

4 Le graphe $\mathcal{G}$ est :

a connexe **b** non connexe

5 La matrice d'adjacence de $\mathcal{G}$ est :

a

	A	B	C	D	E	F	G	H
A	1	1	1	0	0	0	0	0
B	1	1	0	0	0	1	0	0
C	1	0	1	0	0	1	0	1
D	0	0	0	1	1	0	0	0
E	0	0	0	1	1	1	0	1
F	0	1	1	0	1	1	0	0
G	0	0	0	0	0	0	1	1
H	0	0	1	0	1	0	1	1

b

	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	0	0
B	1	0	0	0	0	1	0	0
C	1	0	0	0	0	1	0	1
D	0	0	0	0	1	0	0	0
E	0	0	0	1	0	1	0	1
F	0	1	1	0	1	0	0	0
G	0	0	0	0	0	0	0	1
H	0	0	1	0	1	0	1	0

Théorie

2 Le plan de ville

★ 15 min Corrigé p. 167

Voici le plan simplifié du centre ville d'Aix-en-Provence :

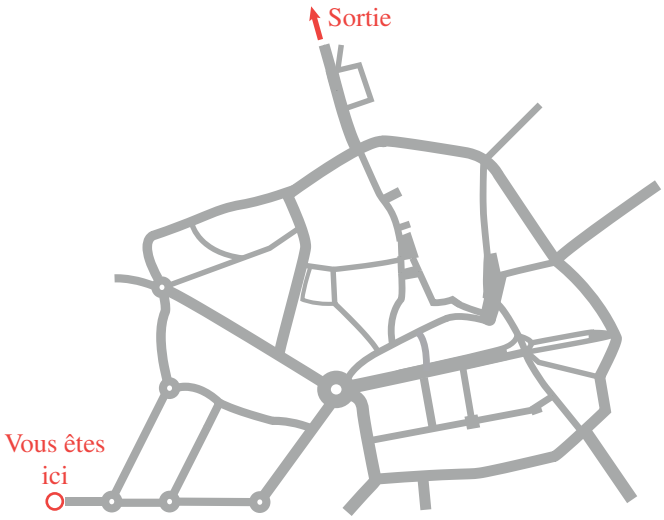


Figure 5.7 – Plan du centre ville d'Aix-en-Provence

Représenter ce centre ville sous forme de graphe en considérant que chaque sens giratoire et chaque intersection de rues est un sommet, et que chaque route est une arête.

On supposera qu'il n'y a pas de rue à sens unique et que le graphe peut être par conséquent non orienté.

Quelles caractéristiques a ce graphe ?

Donner un chemin pour aller du point marqué « Vous êtes ici » à la sortie indiquée par la flèche afin de passer par le moins d'intersections possible ?

3 La maison piégée

★★ 20 min Corrigé p. 168

Une maison comporte quatre entrées et est composée de plusieurs pièces, certaines sans danger et d'autres qui sont piégées. Dans l'une des pièces sans danger se trouve une récompense (voir schéma page ci-contre).

1. Représenter cette maison à l'aide d'un graphe le plus simple possible, permettant de trouver l'entrée la plus optimisée et le chemin le plus court pour aller à la pièce « récompense » en évitant tous les pièges.
2. Trouver ce chemin optimal.

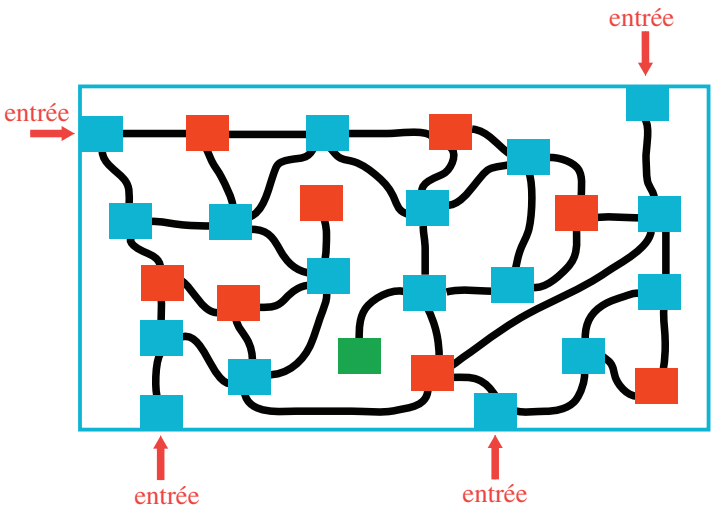


Figure 5.8 – Carte de la maison piégée

Plus court chemin

4 Le loup et compagnie...

★ 15 min Corrigé p. 170

Sur la rive d'un fleuve se trouvent un loup, une chèvre, un chou et un passeur.

Le problème consiste à faire tous les faire passer sur l'autre rive à l'aide d'une barque, menée par le passeur, en respectant les règles suivantes :

- la chèvre et le chou ne peuvent pas rester sur la même rive sans le passeur,
- de même pour la chèvre et le loup,
- le passeur ne peut mettre qu'un seul « passager » avec lui.

On décide de représenter le passeur par la lettre P, la chèvre par C, le loup par L et le chou par X.

1. Représenter ce problème à l'aide d'un graphe où les sommets sont tous les états possibles sur la rive de départ (par exemple, « $PLCX$ » est un sommet représentant le fait que tous sont sur la rive de départ).
2. Trouver alors une solution au problème en indiquant chacun des déplacements.

5 Algorithme de Dijkstra

★★ 10 min Corrigé p. 171

On considère le graphe suivant :

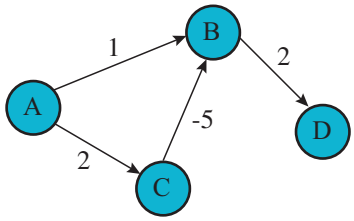


Figure 5.9 – Graphe orienté pondéré

1. Appliquer l'algorithme de Dijkstra à ce graphe en partant du sommet A.
2. Montrer que la distance entre A et D obtenue n'est pas minimale.

6 Plus court chemin

★★ 15 min Corrigé p. 171

On considère le graphe pondéré suivant :

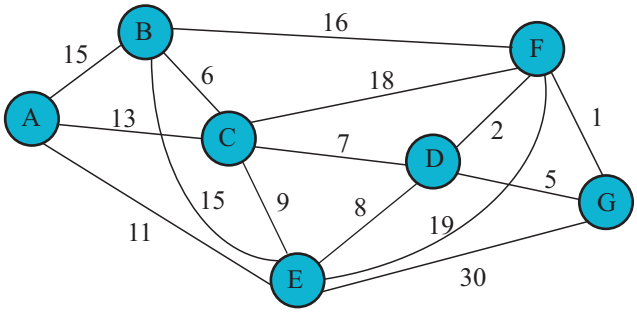


Figure 5.10 – Graphe non orienté pondéré

À l'aide de l'algorithme de Dijkstra, déterminer le plus court chemin pour aller de A à G.

7 Le politicien

★★ 15 min Corrigé p. 175

Un politicien se trouve à sa permanence (en P) et doit se rendre à l'hôpital (en H) le plus rapidement possible.

Le graphe ci-dessous représente le temps de trajet pour aller d'un point à un autre.

INTERROS

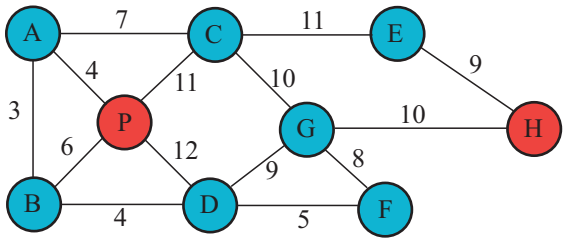


Figure 5.11 – Graphe pondéré par les temps de trajet

Quel chemin doit-il emprunter pour y arriver au plus vite ?

Implémentation

8 Matrice d'adjacence aléatoire

★ — 5 min — Corrigé p. 176

Écrire une fonction en Python admettant un paramètre entier n et permettant de créer une matrice d'adjacence aléatoire de dimensions $n \times n$.

9 Par matrice d'adjacence

★★ — 15 min — Corrigé p. 177

On considère le graphe de sommets A, B, C, D, E et F dont la matrice d'adjacence est la suivante :

$$M = \begin{array}{c} \begin{matrix} A & B & C & D & E & F \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix}$$

À l'aide de M, implémenter ce graphe par liste d'adjacence.

10 Flux migratoires

★★ — 20 min — Corrigé p. 178

Une région est composée de trois départements que l'on notera X, Y et Z.

Chaque année,

- 5 % des habitants du département X déménagent pour le département Y ;
- 1 % des habitants du département X déménagent pour le département Z ;
- 7 % des habitants du département Y déménagent pour le département X ;
- 3 % des habitants du département Y déménagent pour le département Z ;

- 2 % des habitants du département Z déménagent pour le département X ;
- 4 % des habitants du département Z déménagent pour le département Y ;
- 1 % des habitants du département X déménagent pour une autre région ;
- 3 % des habitants du département Y déménagent pour une autre région ;
- 2 % des habitants du département Z déménagent pour une autre région.
- On suppose que le pourcentage d'habitants des autres régions qui aménagent dans un des départements X, Y ou Z est assez faible pour l'assimiler à 0.

1. Représenter ces flux migratoires à l'aide d'un graphe (on n'oubliera pas les personnes restant dans leur département).
2. Donner la matrice de transition M de ce graphe.
3. À l'aide de `numpy`, implémenter M.
4. À l'aide du module `numpy.linalg` et de `matrix_power`, déterminer M^{10}.
5. En 2019, un recensement a établi qu'il y avait :
   - 1,081 million d'habitants dans le département X,
   - 1,076 million d'habitants dans le département Y,
   - 0,327 million d'habitants dans le département Z,
   - 67 millions d'habitants dans le pays.

   On note $P_0 = \begin{pmatrix}1{,}081 & 1{,}076 & 0{,}327 & 64{,}516\end{pmatrix}$ la matrice ligne représentant l'état initial.
   On suppose que le nombre d'habitants dans chaque département au bout de n années est donné par la matrice $P_n = P_0 \times M^n = \begin{pmatrix}x_n & y_n & z_n & a_n\end{pmatrix}$, où x_n, y_n et z_n représentent respectivement le nombre d'habitants des départements X, Y et Z (et où a_n représente le nombre d'habitants dans les autres régions).
   À l'aide de la fonction `dot` de `numpy`, qui permet de calculer le produit de deux matrices, déterminer x_{10}, y_{10} et z_{10}.
6. Écrire un programme permettant de voir l'évolution du nombre d'habitants dans chaque département au cours des 200 années à venir. Que constate-t-on ? Quelles conclusions peut-on faire ?

11 Une simple histoire de temps

★★ 20 min Corrigé p. 181

Dans un pays imaginaire, si le temps d'un jour est pluvieux ou neigeux, alors il reste inchangé dans 50 % des cas le lendemain et ne devient beau que dans 25 % des cas. Lorsque le temps est beau, le lendemain il est pluvieux ou neigeux de manière équiprobable mais il ne restera pas ensoleillé deux jours à la suite.

Les habitants affirment qu'il ne fait beau qu'un jour sur cinq. Ont-ils raison ?

Pour répondre à cette question, on pourra :

- établir le graphe représentant les changements climatiques ;
- déterminer sa matrice de transition M ;
- s'aider de Python, et plus particulièrement de `numpy`, pour déterminer la probabilité qu'il fasse beau un jour donné en calculant une puissance de M assez grande.

12 Le labyrinthe

★★★ 30 min Corrigé p. 182

Le « labyrinthe de Pan » est une attraction d'une fête foraine : c'est un labyrinthe dont les murs intérieurs sont des miroirs. Le plan de ce labyrinthe est donné ci-dessous sur une grille :

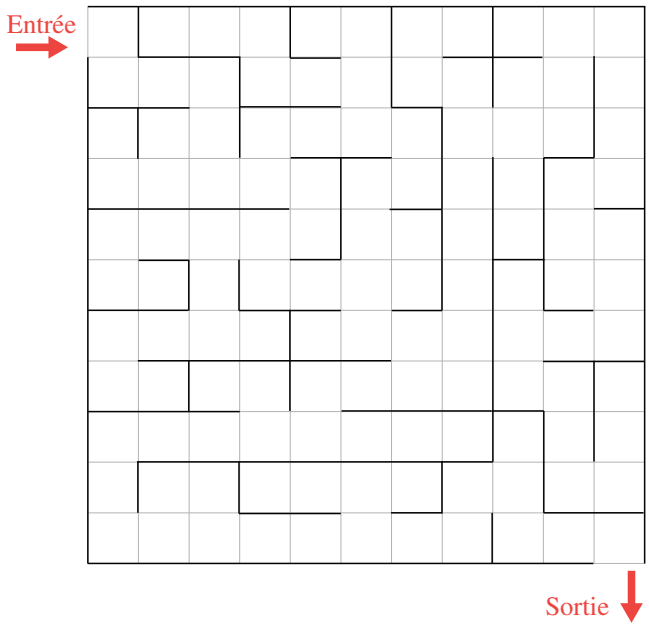


Figure 5.12 – Plan du « Labyrinthe de Pan »

En considérant chaque case comme un sommet de graphe, implémenter ce labyrinthe en Python, puis trouver le chemin le plus court de l'entrée à la sortie.

On pourra pour cela s'aider de la classe `Graphe` complète téléchargeable à la page 149.

1 QCM Vérification de connaissances

Enoncé p. 160

1 Réponse **b**. L'ordre d'un graphe est le nombre de sommets qu'il comporte. Donc ici, c'est 8.

2 Réponse **b**. Le graphe est en effet non orienté car aucun sens n'est spécifié pour passer d'un sommet à un autre.

3 Réponse **b**. Le graphe n'est en effet pas cyclique car il n'existe aucun chemin reliant le sommet D, par exemple, à lui-même. Il en est de même pour le sommet G.

4 Réponse **a**. Le graphe est en effet connexe car il n'existe pas de sommets isolés (non reliés à au moins l'un des autres sommets).

5 Réponse **b**. La seule chose qui change entre les deux matrices proposées est la diagonale principale, qui ne comporte que des « 1 » pour la 1re et que des « 0 » pour la 2nde. Dans la mesure où le graphe ne comporte pas de boucle, il ne doit pas y avoir de « 1 » sur cette diagonale. La première matrice ne convient donc pas.

2 Le plan de ville

Enoncé p. 161

Il n'y a rien de bien compliqué dans ce que l'on nous demande ici. Le travail est juste long...

D'abord, on doit repérer tous les sommets :

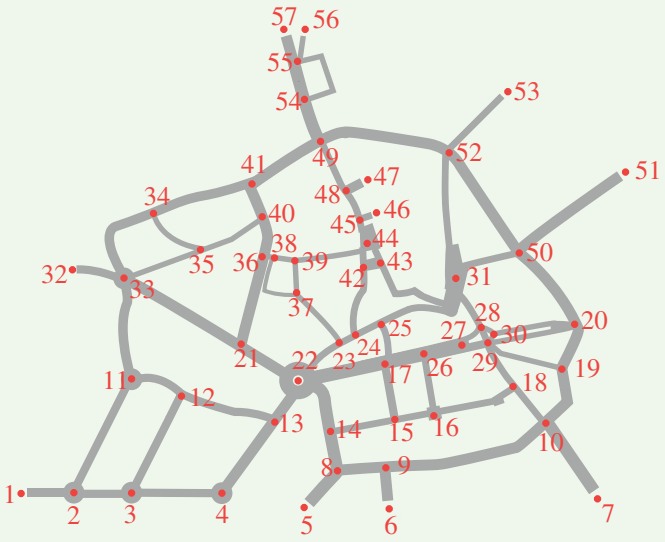


Figure 5.13 – Marquage des intersections

Nous avons ici nommé les sommets par des nombres car il y en a plus de 26, mais nous aurions pu les nommer S_1, S_2, etc.

Ensuite, nous simplifions le plan en mettant les arêtes :

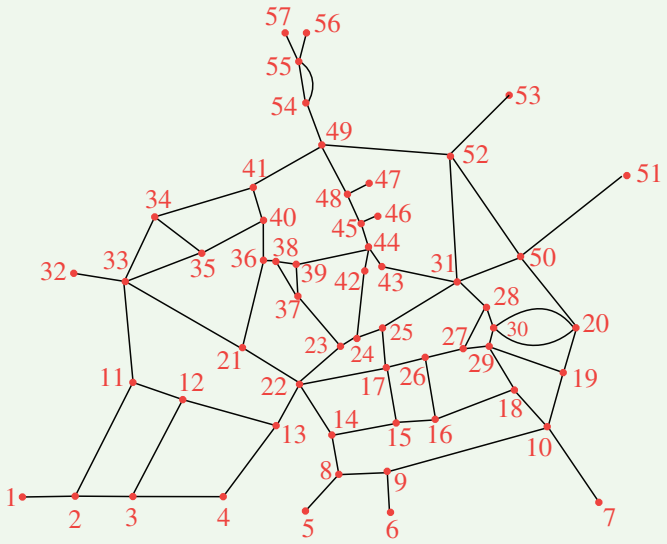


Figure 5.14 – Ajout des arêtes

On peut dire de ce graphe qu'il est :

- connexe (car aucun sommet n'est isolé) ;
- acyclique (il n'y a pas de cycle pour le sommet 1 par exemple) ;
- les sommets 30 et 20, ainsi que les sommets 54 et 55, sont reliés par deux arêtes et ce sont les seuls sommets dans ce cas.

Pour ce qui est d'un chemin pour rejoindre la sortie, nullement besoin d'un programme : on peut voir que le chemin :

```
1 - 2 - 11 - 33 - 34 - 41 - 49 - 54 - 55 - 57
```

convient, avec seulement 8 intersections entre les sommets 1 et 57.

Ce n'est pas le seul car on peut remplacer 34 par 35, mais ce sont les deux seuls chemins qui répondent à notre exigence.

3 La maison piégée

Enoncé p. 161

1 Il semble assez évident de représenter ici les pièces par des sommets et les couloirs par des arêtes. La seule question éventuellement épineuse est de savoir comment représenter les pièces piégées. Dans la mesure où l'on souhaite un graphe le plus simple possible, et comme il ne faut pas aller dans ces pièces, il suffit de les ignorer : le graphe ne va donc pas contenir de sommets représentant les pièces piégées.

De plus, le graphe est non orienté car nullement question de sens dans les déplacements à l'intérieur de cette maison.

Avant de construire le graphe, enlevons donc les pièces piégées.

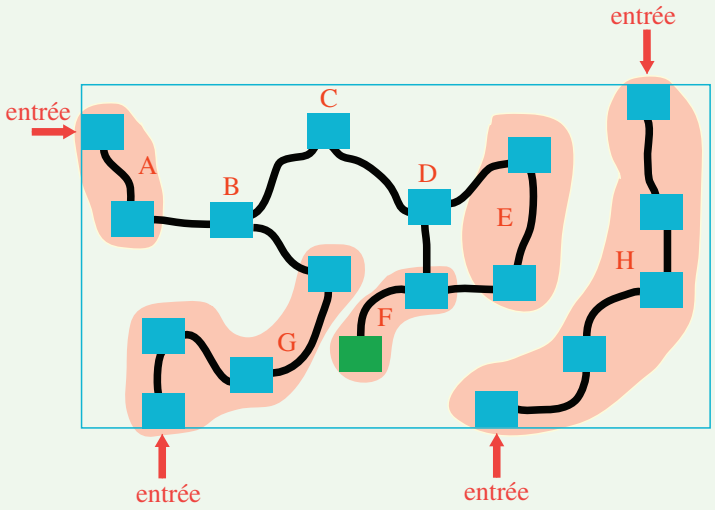


Figure 5.15 – Carte simplifiée de la maison

Nous voyons alors que l'on peut encore simplifier le problème car plusieurs pièces se trouvent sur un même chemin non ramifié, pièces que l'on a réunies sur la figure 5.17 en orange en « groupes » : A, E, G, F et H.

L'idée ici est de regrouper les pièces n'ayant qu'une entrée et une sortie : par exemple, la pièce B comporte trois connexions, donc elle ne fait pas partie d'un groupe. La pièce C comporte bien deux connexions mais aucune de ses deux pièces adjacentes (B et D) n'est, comme elle, limitée à deux connexions. C ne peut donc pas faire partie d'un groupe car elle ne peut pas en constituer un avec ses voisins.

Le problème peut donc se représenter par le graphe suivant :

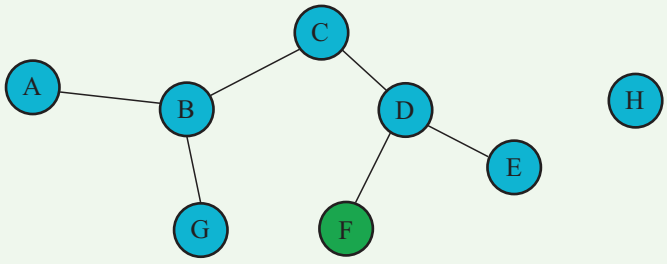


Figure 5.16 – Graphe représentant la maison

2 On déduit du graphe précédent que le chemin le plus court pourrait être :

A – B – C – D – F ou G – B – C – D – F

mais en regardant sur le schéma original de la maison, seul le chemin A – B – C – D – F convient, ce qui correspond au chemin page ci-contre dans la maison.

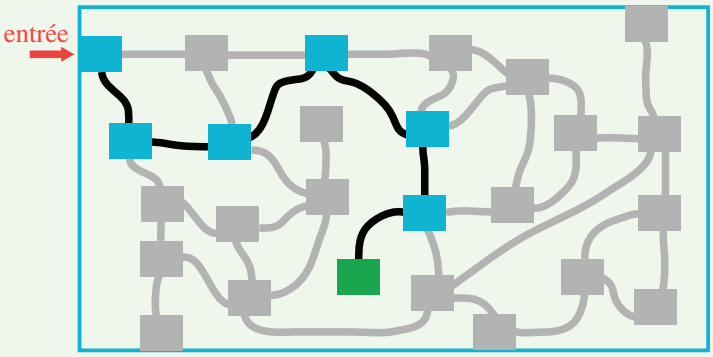


Figure 5.17 – Chemin le plus court

4 Le loup et compagnie...

Enoncé p. 162

1 **MÉTHODE**

Dans ce genre de problème, il est bon d'énumérer tous les états qui font intervenir « P » :

« PLCX » , « PLC » , « PLX » , « PCX » et « PC ».

et sans « P » (en tenant compte des contraintes) :

« LX » , « L » , « C » , « X » et « vide ».

Ce dernier état signifie que la rive est vide, et donc que le problème est résolu.

Le graphe doit traduire les différents déplacements possibles entre chaque état. On peut alors avoir une représentation sagittale qui ressemble à la suivante :

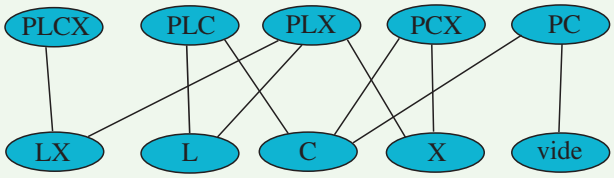


Figure 5.18 – Représentation du problème

2 Résoudre le problème revient alors à trouver un chemin du sommet « PLCX » au sommet « vide ». On obtient alors, par exemple :

PLCX → LX → PLX → X → PCX → C → PC → vide.

Cela signifie que :

- au départ, le passeur amène la chèvre sur la rive de destination, laissant le loup et le chou sur la rive de départ ;
- le passeur revient et se retrouve avec le loup et le chou ;

- il amène le loup sur la rive de destination, laissant le chou ;
- il ramène la chèvre sur la rive de départ car elle ne peut rester avec le loup sur la rive de destination ;
- il amène le chou sur la rive de destination, laissant la chèvre seule ;
- il revient seul car le loup et le chou peuvent rester ensemble sur la rive de destination ;
- il se retrouve donc avec la chèvre sur la rive de départ, et l'amène sur l'autre rive.

5 Algorithme de Dijkstra

Enoncé p. 163

1 On a le tableau suivant :

M	L[A]	L[B]	L[C]	L[D]	P[A]	P[B]	P[C]	P[D]
A	0	1	2	∞	-	A	A	-
A,B	0	1	2	3	-	A	B	B
A,B,C,D	0	-3	2	3	-	C	A	B

2 La distance obtenue de A à D est égale à 3 avec l'algorithme de Dijkstra. Or, nous voyons bien que le chemin A – C – B – D a un poids total de $2 - 5 + 2 = -1$ et que ce poids est minimal.

La raison de ce « bug » est que le sommet C a été traité après le sommet B ; or, le poids de C à B étant négatif, L[B] a été mis à jour. L'algorithme de Dijkstra ne traitant qu'une seule fois un sommet, cela n'a pas pu entraîner la mise à jour de L[D].

C'est le problème de cet algorithme quand il y a des poids négatifs.

6 Plus court chemin

Enoncé p. 163

Nous allons voir pas à pas comment utiliser l'algorithme de Dijkstra.

- Avant tout, construisons une grille carrée à 7 lignes (plus une ligne en haut pour mettre les sommets dans un ordre qui suit à peu près celui du graphe pour mieux y voir) et 7 colonnes car il y a 7 sommets.

A	B	C	D	E	F	G

- Nous partons du sommet A ; par conséquent, nous allons mettre des croix dans la colonne sous « A ». De A, nous pouvons aller jusqu'à B, C et E avec des distances respectives de 15, 13 et 11.

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×						
×						
×						
×						
×						
×						

Nous mettons entre parenthèses le sommet d'où on provient (donc ici : A).

- On prend le nombre le plus petit : 11 (A). On l'abaisse à la ligne suivante et on le marque (ici, nous avons colorié la cellule, mais on peut l'entourer), tout en mettant des croix dans la colonne correspondante :

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×				11 (A)		
×				×		
×				×		
×				×		
×				×		
×				×		

- On part de E. Sur la même ligne, on met tous les sommets qui peuvent être atteints : C (avec une distance totale de $11 + 9 = 20$), D (avec une distance totale de $11 + 8 = 19$), etc.

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×	26 (E)	20 (E)	19 (E)	11 (A)	30 (E)	41 (E)
×				×		
×				×		
×				×		
×				×		
×				×		

- Dans les colonnes où il n'y a pas encore de croix, on regarde quel est le nombre le plus petit qui n'a pas encore été marqué : il s'agit ici de « 13 (A) » : on l'abaisse donc à la ligne suivante, on met des croix dans la colonne et on le marque.

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×	26 (E)	20 (E)	19 (E)	11 (A)	30 (E)	41 (E)
×		13 (A)		×		
×		×		×		
×		×		×		
×		×		×		
×		×		×		

- En partant de C, on peut aller jusqu'à B, F, D et E (mais « E » étant déja marqué, on l'oublie !). Donc sur cette même ligne où l'on vient de marqué le « 13 (A) », on reporte dans les colonnes B, F et D les distances totales : $13 + 6 = 19$ pour B, $13 + 18 = 31$ pour F et $13 + 7 = 20$ pour D.

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×	26 (E)	20 (E)	19 (E)	11 (A)	30 (E)	41 (E)
×	19 (C)	13 (A)	20 (C)	×	31 (C)	
×		×		×		
×		×		×		
×		×		×		
×		×		×		

- Dans les colonnes où il n'y a pas de croix, parmi les nombres non marqués, le plus petit est « 15 (A) ». On l'abaisse donc à la ligne suivante, on le marque et on met des croix dans cette colonne.

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×	26 (E)	20 (E)	19 (E)	11 (A)	30 (E)	41 (E)
×	19 (C)	13 (A)	20 (C)	×	31 (C)	
×	15 (A)	×		×		
×	×	×		×		
×	×	×		×		
×	×	×		×		

- En partant de B, on peut aller jusqu'à F (le sommet C étant déjà marqué, on l'oublie !). On marque donc sur la même ligne et dans la colonne F la distance totale : $15 + 16 = 31$.

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×	26 (E)	20 (E)	19 (E)	11 (A)	30 (E)	41 (E)
×	19 (C)	13 (A)	20 (C)	×	31 (C)	
×	15 (A)	×		×	31 (B)	
×	×	×		×		
×	×	×		×		
×	×	×		×		

- Dans les colonnes non encore complétées d'une croix, on regarde quel est le nombre le plus petit : c'est ici « 19 (E) », que l'on s'empresse donc d'abaisser à la ligne suivante tout en complétant la colonne de croix.

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×	26 (E)	20 (E)	19 (E)	11 (A)	30 (E)	41 (E)
×	19 (C)	13 (A)	20 (C)	×	31 (C)	
×	15 (A)	×		×	31 (B)	
×	×	×	19 (E)	×		
×	×	×	×	×		
×	×	×	×	×		

- En partant de D, on peut aller jusquà F (avec une distance totale de $19 + 2 = 21$) et G (avec une distance totale de $19 + 5 = 24$), que l'on met sur la même ligne :

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×	26 (E)	20 (E)	19 (E)	11 (A)	30 (E)	41 (E)
×	19 (C)	13 (A)	20 (C)	×	31 (C)	
×	15 (A)	×		×	31 (B)	
×	×	×	19 (E)	×	21 (D)	24 (D)
×	×	×	×	×		
×	×	×	×	×		

- Dans les colonnes non encore complétées de croix, le plus petit nombre est « 21 (D) » ; on l'abaisse donc à la ligne suivante, on le marque et on complète la colonne d'une croix.

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×	26 (E)	20 (E)	19 (E)	11 (A)	30 (E)	41 (E)
×	19 (C)	13 (A)	20 (C)	×	31 (C)	
×	15 (A)	×		×	31 (B)	
×	×	×	19 (E)	×	21 (D)	24 (D)
×	×	×	×	×	21 (D)	
×	×	×	×	×	×	

- Du sommet F, on peut aller jusqu'à G avec une distance totale de $21 + 1 = 22$. Comme 22 est le plus petit nombre de la colonne G, on le marque.

A	B	C	D	E	F	G
×	15 (A)	13 (A)		11 (A)		
×	26 (E)	20 (E)	19 (E)	11 (A)	30 (E)	41 (E)
×	19 (C)	13 (A)	20 (C)	×	31 (C)	
×	15 (A)	×		×	31 (B)	
×	×	×	19 (E)	×	21 (D)	24 (D)
×	×	×	×	×	21 (D)	
×	×	×	×	×	×	22 (F)

- « 22 » est donc le poids minimal et la chaîne correspondante se trouve à l'aide des sommets entre parenthèses lus à l'envers en partant de G ; on lit alors :
« G → F → (colonne F) → D → (colonne D) → E → (colonne E)→ A »
On obtient donc (à l'envers) :

$$A \to E \to D \to F \to G$$

7 Le politicien

Enoncé p. 163

On utilise l'algorithme de Dijkstra en indiquant chaque étape dans une ligne du tableau page suivante.

Dans la colonne « H », on voit qu'il y a deux chemins qui mènent à ce sommet ; on prend celui qui a la plus petite pondération.

La durée minimale est donc de 29 minutes avec le trajet :

$$P \to B \to D \to G \to H.$$

Remarque : notons que le « 11 (P) » a été reporté et non le « 11 (A) » car il a été considéré avant. C'est une règle générale : on reporte toujours ce qui a été considéré en premier.

CORRIGÉS

P	A	B	C	D	E	F	G	H
×	4 (P)	6 (P)	11 (P)	12 (P)				
×	4 (P)	7 (A)	11 (A)					
×	×	6 (P)		10 (B)				
×	×	×		10 (B)		15 (D)	19 (D)	
×	×	×	11 (P)	×	22 (C)		21 (C)	
×	×	×	×	×		15 (D)	23 (F)	
×	×	×	×	×		×	19 (D)	
×	×	×	×	×		×	×	29 (G)
×	×	×	×	×	22 (C)	×	×	31 (E)

8 Matrice d'adjacence aléatoire

Enoncé p. 164

MÉTHODE

« aléatoire » implique souvent « appel au module random ». De plus, on exploite le fait qu'une matrice d'adjacence est carrée, ne comporte que des « 0 » et des « 1 », que sa diagonale principale est remplie de 0 et qu'elle est symétrique par rapport à la diagonale principale.

Une fonction possible est alors la suivante :

```
from random import randint

def matadj(n):
    M = [ [0 for j in range(n)] for i in range(n) ]
    for ligne in range(n):
        for colonne in range(n):
            if colonne > ligne:
                M[ligne][colonne] = randint(0,1)
            elif colonne < ligne:
                M[ligne][colonne] = M[colonne][ligne]

    return M
```

L'idée est donc ici d'initialiser une matrice de dimensions $n \times n$ remplie par exemple de « 0 ».

Ensuite, pour chaque ligne (`for ligne in range(n)`), on parcourt chaque colonne : si on est au-dessus de la diagonale principale (`if colonne > ligne`) alors on choisit un coefficient au hasard entre 0 et 1. Si on est au contraire en dessous, le coefficient doit être égal à son symétrique par rapport à la diagonale principale (`M[ligne][colonne] = M[colonne][ligne]`).

Inutile de faire le cas où `ligne == colonne` car dans ce cas, le coefficient doit être nul, ce qui est déjà le cas par construction initiale de la matrice.

9 Par matrice d'adjacence

Enoncé p. 164

On pourrait implémenter ce graphe « à la main » à l'aide de la classe `Graphe` donnée page 149, en énumérant toutes les arêtes grâce à la matrice d'adjacence donnée, mais si on pouvait implémenter une façon de passer de la matrice d'adjacence à une liste d'adjacence, ce serait mieux.

Le programme page suivante répond à cette requête.

Il affiche :

```
B : ['C', 'D', 'F']
A : ['B', 'C', 'E']
C : ['D', 'E', 'F']
E : ['F']
D : ['E']
F : []
```

En effet, une arête est créée entre les sommets i et j si le coefficient m_{ij} est égal à 1. Il suffit donc de faire deux boucles, sur i et j (où i représente la ligne et j la colonne de la matrice) et d'effectuer un test de valeur pour savoir si on doit créer une arête ou non.

De plus, on sait que la matrice d'adjacence est toujours symétrique par rapport à sa diagonale principale. Il est donc inutile d'effectuer ces tests pour les coefficients m_{ij} avec $j < i$:

$$M = \begin{array}{c} \begin{matrix} A & B & C & D & E & F \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix}$$

CORRIGÉS

```
class Graphe:
    def __init__(self):
        self.Liste = {}

    def addArete(self, u, v):
        if v not in self.Liste:
            self.Liste[v] = []

        if u not in self.Liste:
            self.Liste[u] = []

        self.Liste[u].append(v)

    def affiche(self,M, sommets):
        for i in range( len(M) ):
            for j in range(i , len(M) ):
                if M[i][j] == 1:
                    self.addArete( sommets[i] , sommets
    [j] )

        for key,value in self.Liste.items():
            print(key , ":" , value)

M = [
        [ 0 , 1 , 1 , 0 , 1 , 0 ] ,
        [ 1 , 0 , 1 , 1 , 0 , 1 ] ,
        [ 1 , 0 , 0 , 1 , 1 , 1 ] ,
        [ 0 , 1 , 1 , 0 , 1 , 0 ] ,
        [ 1 , 0 , 1 , 1 , 0 , 1 ] ,
        [ 0 , 1 , 1 , 0 , 1 , 0 ]
    ]

G = Graphe()
sommets = [ 'A' , 'B' , 'C' , 'D' , 'E' , 'F' ]

G.affiche(M , sommets)
```

10 Flux migratoires

Enoncé p. 164

1 Dans la mesure où certains habitants déménagent vers une autre région, nous allons représenter les flux migratoires par un graphe à quatre sommets : X, Y, Z et A (où A représentera tout ce qui n'est pas dans la région).

De plus, le graphe doit être orienté (car le mouvement X → Y n'est pas similaire au mouvement Y → X) et pondéré (par les pourcentages, mis en décimaux par exemple). Il ne faut pas oublier les pourcentages non indiqués dans l'énoncé, à savoir ceux correspondants aux habitants qui restent dans leur département.

On a alors la représentation sagittale suivante :

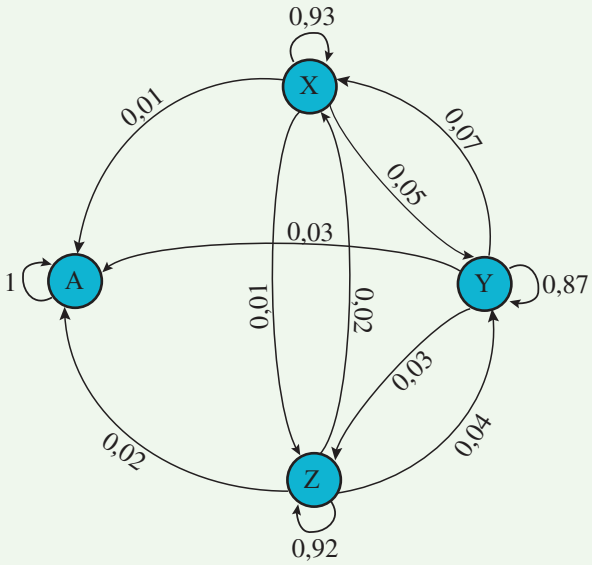


Figure 5.19 – Représentation des flux migratoires

2 La matrice de transition de ce graphe est :

$$M = \begin{array}{c} \begin{matrix} X & Y & Z & A \end{matrix} \\ \begin{pmatrix} 0{,}93 & 0{,}05 & 0{,}01 & 0{,}01 \\ 0{,}07 & 0{,}87 & 0{,}03 & 0{,}03 \\ 0{,}02 & 0{,}04 & 0{,}92 & 0{,}02 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} X \\ Y \\ Z \\ A \end{matrix}$$

3 On implémente la matrice de la manière suivante :

```
>>> from numpy import *
>>> M = array( [
  [0.93,0.05,0.01,0.01] ,
  [0.07,0.87,0.03,0.03] ,
  [0.02,0.04,0.92,0.02] ,
  [0,0,0,1]
  ] )
```

4 On obtient M^{10} en tapant :

```
>>> from numpy.linalg import matrix_power
>>> matrix_power(M,10)
array([[0.56932088, 0.21942888, 0.08467249,
0.12657775],
  [0.30805016, 0.33744598, 0.13505622, 0.21944764],
  [0.16509636, 0.18149117, 0.46765393, 0.18575854],
  [0. , 0. , 0. , 1. ]])
```

5 On peut écrire :

```
>>> P = array( [ 1.081 , 1.076 , 0.327 , 64.516] )
>>> dot( P , matrix_power(M,10) )
array([ 1.00088435, 0.65964211, 0.38977429,
64.94969926])
```

On obtient ainsi :

$$x_{10} \approx 1{,}000\,884 \quad , \quad y_{10} \approx 0{,}659\,642 \quad , \quad z_{10} \approx 0{,}389\,774.$$

6 Une simple boucle suffit :

```
from numpy import array, dot
from numpy.linalg import matrix_power

P = array( [ 1.081 , 1.076 , 0.327 , 64.516] )

M = array( [
    [0.93,0.05,0.01,0.01] ,
    [0.07,0.87,0.03,0.03] ,
    [0.02,0.04,0.92,0.02] ,
    [0,0,0,1]
    ] )

for n in range(1,201):
    print( dot( P , matrix_power(M,n) ) )
```

On s'aperçoit alors que les nombres d'habitants des départements X, Y et Z ne font que décroitre, ce qui nous laisse à penser que ces départements se dépeuplent et qu'à long terme, ils seront désertés si aucun plan n'est adopté pour remédier à cette migration.

Voir encadré « À retenir » page ci-contre.

À RETENIR

Les graphes sont très souvent utilisés pour traduire des changements d'états :

- mouvement d'un solide vers plusieurs endroits,
- changement d'état d'un objet (marche, arrêt, veille, etc.),
- changement climatique,
- etc.

Le graphe ainsi obtenu est appelé *graphe probabiliste* et la matrice de transition permet d'obtenir, lorsque n tend vers $+\infty$, une matrice M^n dont chaque colonne représente la probabilité des états. Dans la pratique, on prendra n assez grand. Nous en parlons dans l'exercice suivant.

11 Une simple histoire de temps

Enoncé p. 165

- Commençons par représenter les changements climatiques à l'aide d'un graphe :

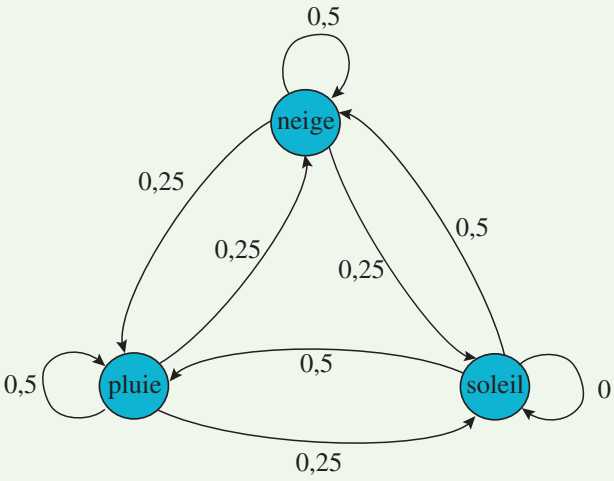


Figure 5.20 – Représentation des changements climatiques

- La matrice de transition de ce graphe est :

$$M = \begin{array}{c} \\ N \\ P \\ S \end{array} \begin{array}{c} \begin{array}{ccc} N & P & S \end{array} \\ \begin{pmatrix} 0,5 & 0,25 & 0,25 \\ 0,25 & 0,5 & 0,25 \\ 0,5 & 0,5 & 0 \end{pmatrix} \end{array}$$

- On implémente alors la matrice de la manière suivante :

```
>>> from numpy import *
>>> M = array( [
  [0.5,0.25,0.25] ,
  [0.25,0.5,0.25] ,
  [0.5,0.5,0] ,   ] )
```

- On calcule par exemple M^{100} :

```
>>> from numpy.linalg import matrix_power
>>> matrix_power(M,100)
array([[0.4, 0.4, 0.2],
  [0.4, 0.4, 0.2],
  [0.4, 0.4, 0.2]])
```

Cette dernière matrice nous dit alors que la probabilité qu'il fasse beau à long terme est égale à 0,2 (par rapport à l'ordre que nous avons choisi, on regarde la dernière colonne), soit un jour sur cinq.

12 Le labyrinthe

Enoncé p. 166

Il existe des méthodes plus économes que celle souhaitée par l'énoncé, comme par exemple la fusion de cases pour réduire le nombre de sommets du graphe représentant le labyrinthe, mais contentons-nous pour l'instant de la méthode la plus évidente : à chaque case son sommet. Notons $S_{i,j}$ le sommet de la case sur la i-ième ligne et j-ième colonne. Si on fait un zoom sur l'entrée, cela donne :

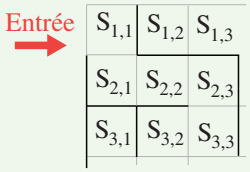


Figure 5.21 – Zoom sur l'entrée

On marque par une arête la présence d'un passage entre 2 cases adjacentes. Par exemple : il n'y a pas d'arête entre $S_{1,1}$ et $S_{1,2}$ mais il y en a une entre $S_{1,1}$ et $S_{2,1}$.

En tout, il aura $11 \times 11 = 121$ sommets. Le graphe est creux car il y aura très peu d'arêtes par rapport au nombre de sommets. Ceci nous pousse à implémenter le graphe par liste d'adjacence.

Téléchargez l'implémentation complète.

À l'aide de ce programme, on voit que le plus court chemin est le suivant, de longueur 36 (nombre d'arêtes) :

```
'S1-1', 'S2-1', 'S2-2', 'S2-3', 'S3-3', 'S4-3', 'S4-4',
'S4-5', 'S5-5', 'S5-4', 'S6-4', 'S6-5', 'S6-6', 'S7-6',
'S7-7', 'S8-7', 'S8-6', 'S8-5', 'S9-5', 'S9-4', 'S9-3',
  'S9-2', 'S9-1', 'S10-1', 'S11-1', 'S11-2', 'S11-3',
 'S11-4', 'S11-5', 'S11-6', 'S11-7', 'S11-8', 'S10-8',
        'S10-9', 'S11-9', 'S11-10', 'S11-11'
```

Cela correspond au chemin suivant :

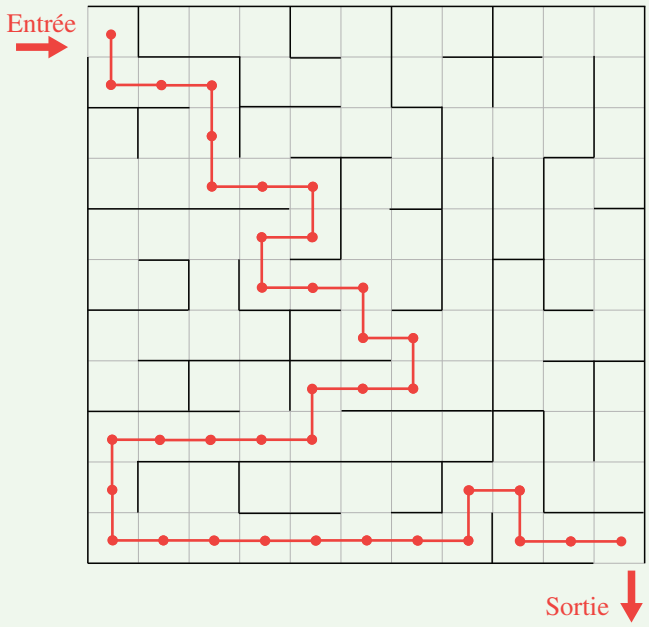


Figure 5.22 – Chemin le plus court

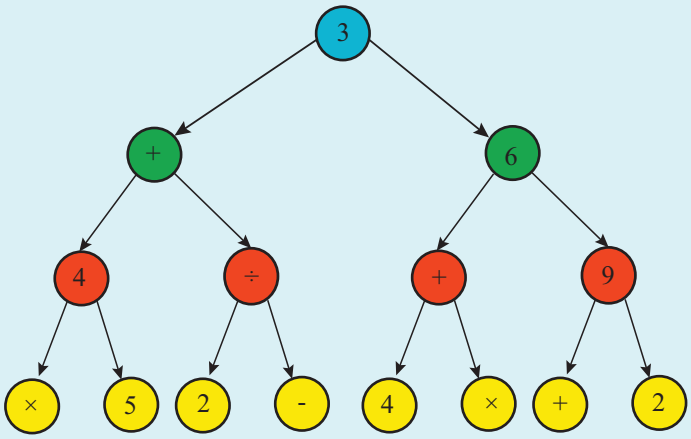
Chapitre 6

Les arbres : structures hiérarchiques

Plan du chapitre

Exercice type

On considère l'arbre suivant :



On parcourt cet arbre en profondeur avec un ordre préfixe.

1 Quel est le résultat de l'opération obtenue si l'on tient compte des priorités opératoires, c'est-à-dire du fait que la multiplication et la division sont prioritaires sur l'addition et la soustraction ?

2 Implémenter cet arbre et retrouver le résultat de la question précédente à l'aide d'un programme écrit en Python.

Voir corrigé page 198

COURS

Les listes et tableaux sont des structures de données peu pratiques à manipuler dès lors que l'on souhaite effectuer des recherches sur leurs valeurs, ou même ajouter et supprimer des valeurs.

PROBLÉMATIQUE

Comment représenter un ensemble ordonné de clés de sorte à effectuer simplement des opérations élémentaires ?

1 Notion d'arbres

Définition 1

Un *arbre* est une structure de données utilisée pour représenter un graphe acyclique orienté connexe.

Définitions 2 : vocabulaire

- Le premier sommet d'un arbre est la *racine*.
- Les arêtes d'un arbre sont aussi appelées *branches*.
- Les sommets autres que le premier sont appelés les *nœuds*.
- Les nœuds issus d'un nœud N sont appelés les *fils* de N. N est alors appelé le *père*.
- Les nœuds d'un même niveau sont appelés les *nœuds frères*.
- Les nœuds n'ayant pas de fils sont aussi appelés des *feuilles*.
- La *distance* d'un nœud à un autre est le nombre de branches reliant l'un à l'autre.
- La *profondeur* d'un nœud est la distance de la racine à ce nœud.
- La *hauteur* de l'arbre est la plus grande distance de la racine à une feuille.
- La *taille* de l'arbre est le nombre de nœuds.
- Une *clé* est la valeur d'un nœud.

Remarque : par convention, un arbre vide a une hauteur égale à -1.

Exemple : on a représenté ci-dessous un arbre de hauteur 2 et de taille 9.

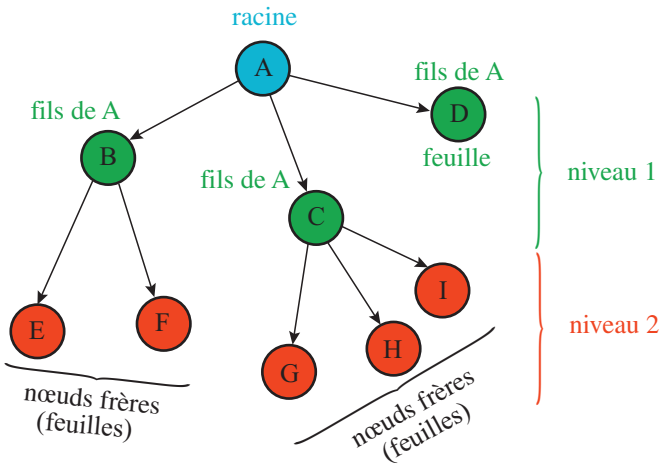


Figure 6.1 – Représentation sagittale d'un arbre

On peut représenter un arbre par une liste commençant par la racine et suivie d'une suite de listes représentant les différents chemins de l'arbre.

Exemple : reprenons l'arbre de l'exemple précédent en ajoutant un autre niveau :

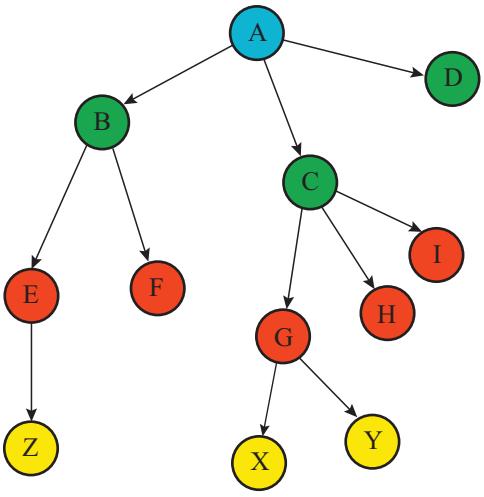


Figure 6.2 – Arbre à trois niveaux

Cet arbre pourra alors être implémenté par la liste :

```
[ 'A' ,
  ['B',
    ['E' ,
      ['Z']
    ] ,
    ['F']
  ],
  ['C' ,
    ['G' ,
      ['X'] ,
      ['Y']
    ] ,
    ['H'] ,
    ['I']
  ],
['D']
]
```

COURS

2 Arbre binaire

2.1 Définitions

Définitions 3

Un arbre est dit *binaire* si chaque nœud admet au plus deux fils, que l'on appelle *fils gauche* et *fils droit*.
Si, pour tout nœud de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1, l'arbre est un *arbre AVL*, du nom de Georgy Adelson-Velsky et Evgenii Landis, qui l'ont inventé en 1962.

Définition 4 : sous-arbres gauche et droit

Soit T un arbre binaire. On appelle *sous-arbre gauche* (resp. *droit*) d'un nœud l'arbre issu du fils gauche (resp. droit) de ce nœud.

Exemple : on a représenté ci-dessous un arbre binaire de hauteur 3 et de taille 9.

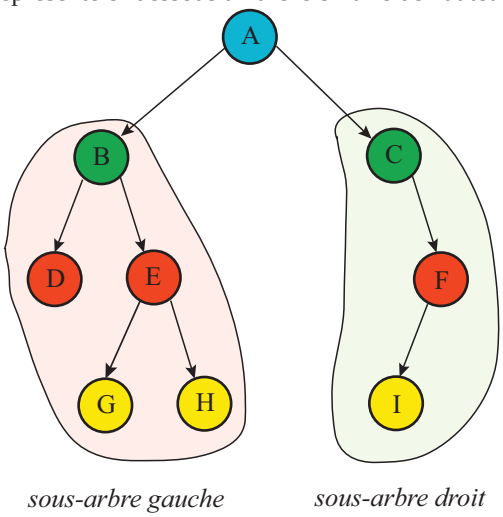


Figure 6.3 – Représentation des sous-arbres gauche et droit de la racine

2.2 Propriétés sur les arbres binaires

Propriété 1

Un arbre binaire de hauteur h admet au plus $1 + 2 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$ nœuds.

Propriétés 2

- Un arbre binaire ayant n nœuds a une hauteur h telle que :
$$\log_2(n+1) - 1 \leqslant h \leqslant n$$
c'est-à-dire :
$$\frac{\ln(n+1)}{\ln 2} - 1 \leqslant h \leqslant n.$$
- Tout arbre binaire ayant f feuilles a une hauteur h telle que :
$$h \geqslant \log_2(f).$$
- Soit un arbre binaire ayant f feuilles et n nœuds. Alors,
$$\log_2(f) \leqslant \log_2\left(\frac{n+1}{2}\right).$$

2.3 Arbre Binaire de Recherche (ABR)

Retrouvez en vidéo la notion d'arbre binaire de recherche.

Définition 5

Un *arbre binaire de recherche*, en abrégé : ABR, est un arbre vide ou un arbre binaire dont :

- tous les nœuds sont étiquetés par des nombres ;
- la racine est supérieure ou égale à tout nœud du sous-arbre gauche ;
- la racine est inférieure à tout nœud du sous-arbre droit ;
- les deux sous-arbres de la racine sont eux-mêmes des ABR.

Exemple : l'abre binaire ci-dessous est un ABR.

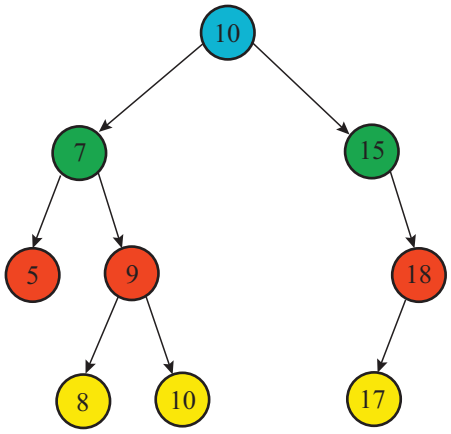


Figure 6.4 – Représentation d'un ABR

COURS

3 Représentations des arbres binaires

3.1 Représentation contigüe

On peut représenter un arbre binaire à l'aide d'un tableau (une liste) à une dimension.

Si L est cette liste :

- L[0] est la racine ;
- L[1] est le fils gauche de la racine (notons-le R.g) ;
- L[2] est le fils droit de la racine (notons-le R.d) ;
- L[3] est le fils gauche de R.g ;
- L[4] est le fils droit de R.g ;
- L[5] est le fils gauche de R.d ;
- L[6] est le fils droit de R.d ;
- etc.

Exemple : la représentation de l'arbre de la figure 2.1 page 188 est donnée ci-dessous.

Rang	0	1	2	3	4	5	6	7	8	9	10	11	12
Valeur	A	B	C	D	E		F			G	H	I	

```
L = [A , B , C , D , E , , F , , , G , H , I , ]
```

3.2 Représentation avec une classe

3.2.1 - Première possibilité

Nous pouvons représenter un arbre binaire à l'aide d'une classe `node` (par exemple) contenant trois champs :

- la valeur du nœud ;
- le parent du nœud ;
- la position du nœud par rapport à son parent (gauche ou droit)

puis d'une classe `tree` contenant une liste des nœuds, dont le premier élément est la racine.

Si nous implémentons en Python l'arbre de la figure 2.1, cela donne :

```
class node:
    def __init__(self , data , parent , position):
        self.data = data
    self.parent = parent
        self.position = position
```

```
class tree:
    def __init__(self , liste):
        self.racine = liste[0].data
        self.noeud = liste[1:]

A = node("A" , None , None)
B = node("B" , "A" , "Gauche")
C = node("C" , "A" , "Droit")
D = node("D" , "B" , "Gauche")
E = node("E" , "B" , "Droit")
F = node("F" , "C" , "Droit")
G = node("G" , "E" , "Gauche")
H = node("H" , "E" , "Droit")
I = node("I" , "F" , "Gauche")

arbre = tree([A,B,C,D,E,F,G,H,I])

print( arbre.noeud[2].data ,
        "est le fils" ,
        arbre.noeud[2].position ,
        "de" , arbre.noeud[2].parent)
```

qui affiche : `D est le fils Gauche de B`

3.2.2 - Autre possibilité

Nous pouvons utiliser la récursivité implicitement liée aux arbres binaires. Ainsi, pour l'arbre de la figure 2.1), on peut définir de manière simple une classe `Arbre` comme dans le code donné page suivante.

```
class ArbreBinaire:
    def __init__(self,racine,filsGauche,filsDroit):
        self.racine = racine
        self.filsGauche = filsGauche
        self.filsDroit = filsDroit

    def Affiche(self, space = 0):
        spaces = "  "*space
        print(spaces,self.racine)
        if self.filsGauche:
            self.filsGauche.Affiche(space+1)
        if self.filsDroit:
            self.filsDroit.Affiche(space+1)
```

COURS

```
tree = ArbreBinaire('A', # racine de l'arbre principal
  ArbreBinaire('B', ArbreBinaire('D',None,None),
    ArbreBinaire('E',ArbreBinaire('G',None,None),
    ArbreBinaire('H',None,None)) ),
  # fin du fils gauche
  ArbreBinaire('C', ArbreBinaire('-',None,None),
    ArbreBinaire('F',ArbreBinaire('I',None,None),
    ArbreBinaire('-',None,None)) )
  # fin du fils droit
)
tree.Affiche()
```

Ce programme affiche :

```
A
   B
      D
      E
         G
         H
   C
      -
      F
         I
         -
```

4 Algorithmique

4.1 Calcul de la taille d'un arbre binaire

Ce calcul se fait à l'aide d'un algorithme récursif.

```
fonction taille(arbre):
   Si arbre est vide:
     retourner 0
   Sinon:
     G ← taille(gauche(arbre))
     D ← taille(droit(arbre))
     return 1 + G + D
```

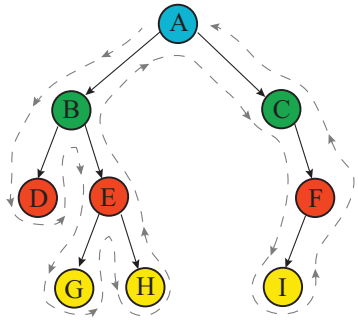
Ici, `gauche(arbre)` représente le sous-arbre gauche du nœud sur lequel nous sommes et `droit(arbre)` son sous-arbre droit.

4.2 Calcul de la hauteur d'un arbre binaire

On utilise ici le même type d'algorithme que pour le calcul de la taille, en convenant de prendre -1 pour hauteur d'un arbre vide (où seule la racine existe).

```
fonction hauteur(arbre):
   Si arbre est vide:
     retourner -1
   Sinon:
     G ← hauteur(gauche(arbre))
     D ← hauteur(droit(arbre))
     return 1 + max(G,D)
```

4.3 Parcours en profondeur d'un arbre binaire



Parcourir un arbre binaire signifie accéder à l'information contenue dans les nœuds.
Nous allons donc imaginer que l'on parcourt l'arbre de la façon indiquée ci-contre.

Figure 6.5 – Parcours d'un arbre binaire

Ce parcours est appelé *parcours en profondeur* de l'arbre.

Il existe trois façons d'énumérer les nœuds.

4.3.1 - Ordre préfixe

On liste les nœuds dans l'ordre dans lequel on les rencontre pour la première fois.

Exemple : l'arbre de la figure 6.5 donne :

$$A - B - D - E - G - H - C - F - I$$

Algorithme (récursif) de parcours préfixe

```
ParcoursPrefixe(arbre):
   Afficher la clé de l'arbre
   ParcoursPrefixe(gauche(arbre))
   ParcoursPrefixe(droit(arbre))
```

COURS

4.3.2 - Ordre postfixe

On liste les nœuds dans l'ordre dans lequel on les rencontre pour la dernière fois.

Exemple : l'arbre de la figure 6.5 donne :

D – G – H – E – B – I – F – C – A

Algorithme (récursif) de parcours postfixe

```
ParcoursPostfixe(arbre):
   ParcoursPostfixe(gauche(arbre))
   ParcoursPostfixe(droit(arbre))
   Afficher la clé de l'arbre
```

4.3.3 - Ordre infixe

On liste les nœuds selon la règle suivante :

- chaque nœud sans fils gauche est répertorié la première fois qu'on le rencontre ;
- chaque nœud comportant un fils gauche est répertorié la seconde fois qu'on le rencontre ;

Exemple : pour l'arbre de la figure 6.5, nous allons avant tout faire un tableau.

Sommets	Fils gauche	Répertorié la...	Parcours
A	Oui	2nde fois	
B	Oui	2nde fois	
D	Non	1re fois	D
B	Oui	2nde fois	D – B
E	Oui	2nde fois	D – B
G	Non	1re fois	D – B – G
E	Oui	2nde fois	D – B – G – E
H	Non	1re fois	D – B – G – E – H
E	Oui	2nde fois	D – B – G – E – H
B	Oui	2nde fois	D – B – G – E – H
A	Oui	2nde fois	D – B – G – E – H – A
C	Non	1re fois	D – B – G – E – H – A – C
F	Oui	2nde fois	D – B – G – E – H – A – C
I	Non	1re fois	D – B – G – E – H – A – C – I
F	Oui	2nde fois	D – B – G – E – H – A – C – I – F
C	Non	1re fois	D – B – G – E – H – A – C – I – F
A	Oui	2nde fois	D – B – G – E – H – A – C – I – F

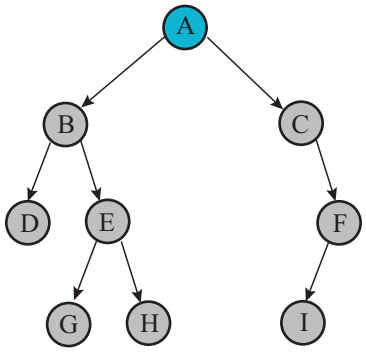
Algorithme (récursif) de parcours infixe

```
ParcoursInfixe(arbre):
   ParcoursInfixe(gauche(arbre))
   Afficher la clé de l'arbre
   ParcoursInfixe(droit(arbre))
```

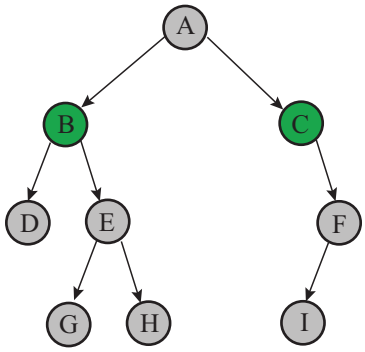
4.4 Parcours en largeur

Reprenons l'arbre de la figure 2.1 page 188.

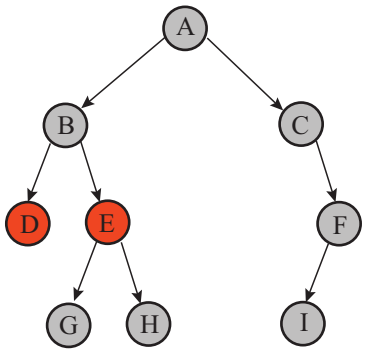
L'idée consiste à lister chaque nœud-fils niveau par niveau.



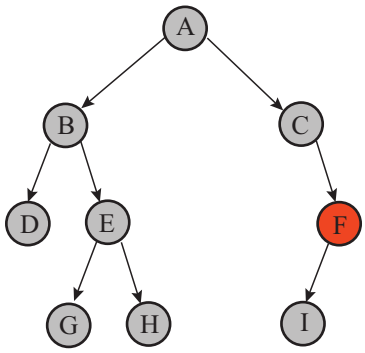
Étape 1



Étape 2

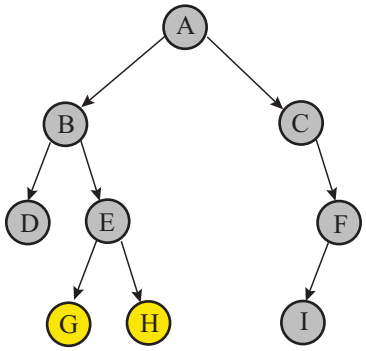


Étape 3

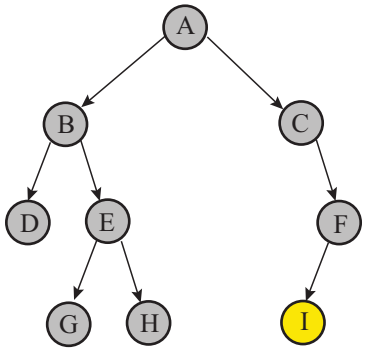


Étape 4

COURS



Étape 5



Étape 6

L'algorithme de parcours en largeur est basé sur une file.

Algorithme de parcours en largeur

```
F est une file vide
P est une liste vide
Enfiler racine à F
Ajouter racine à P
Tant que F non vide:
   U ← tête de F
   Pour V fils de U:
     Enfiler V à F
     Ajouter V à P
   Fin du Pour
   Défiler U de F
Fin du Tant que
```

4.5 Recherche dans un ABR

La structure d'un ABR facilite grandement les recherches, et c'est la raison pour laquelle cette structure de données est intéressante. En effet, on sait par définition que chaque clé de nœuds-fils-gauches est inférieure à celle du nœud courant et que chaque clé de nœuds-fils-droits est supérieure. Cela donne l'algorithme suivant :

Algorithme (récursif) de recherche dans un ABR

```
Recherche(ABR,val):
   Si ABR est vide:
     Retourner Faux
   Sinon:
     r ← racine de ABR
     Si val = r:
       Retourner ABR
```

Algorithme (récursif) de recherche dans un ABR (suite)

```
    Sinon:
      Si val ⩽ r:
        Retourner Recherche(gauche(ABR),val)
      Sinon:
        Retourner Recherche(droite(ABR),val)
      Fin du Si val ⩽ r
    Fin du Si val = r
  Fin du Si ABR est vide
```

Propriété 3

Le coût d'une recherche dans un ABR est en $O(h)$, où h est la hauteur de l'arbre.

4.6 Insertion dans un ABR

Pour insérer une clé dans un ABR, il suffit de faire une recherche et de faire l'insertion à l'endroit où la recherche est infructueuse.

Algorithme d'insertion dans un ABR

```
Insere(ABR,clé):
  Si ABR est vide:
    return CreerABR(clé)
  Sinon:
    Si clé ⩽ racine(ABR):
      G ← Insere(gauche(ABR),clé)
      gauche(ABR) ← G
      Retourner ABR
    Sinon:
      D ← Insere(droit(ABR),clé)
      droit(ABR) ← D
      Retourner ABR
    Fin du Si clé ⩽ racine(ABR)
  Fin du Si ABR est vide
```

Remarque : la fonction `CreerABR` est une fonction qui crée un arbre à partir d'une valeur ; quant à `gauche(ABR)` et `droit(ABR)`, ce sont deux fonctions qui retournent respectivement les fils gauche et droit de ABR.

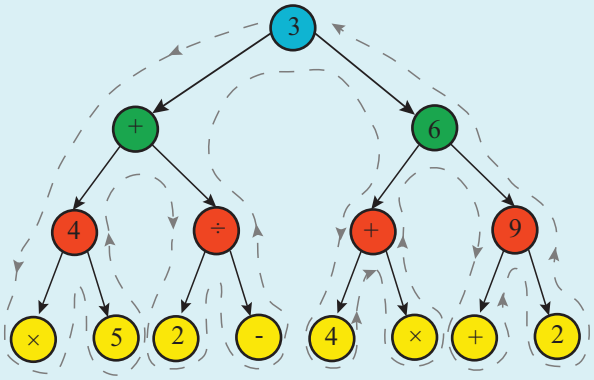
COURS

À RETENIR

Arbre binaire	Chaque nœud admet au plus deux fils : un fils gauche et un fils droit.
Arbre AVL	Pour tout nœud de l'arbre, les hauteurs des sous-arbre gauche et droit diffèrent d'au plus 1.
Arbre Binaire de Recherche (ABR)	Arbre vide ou arbre étiqueté par des nombres où : • la racine est supérieure ou égale à tout nœud du sous-arbre gauche ; • la racine est inférieure à tout nœud du sous-arbre droit ; • les deux sous-arbres de la racine sont eux-mêmes des ABR.

Solution de l'exercice type

1 Le parcours en profondeur est le parcours suivant :



Il donne alors l'opération : $3 + 4 \times 5 \div 2 - 6 + 4 \times 9 + 2 = 45$.

2 Nous pouvons nous inspirer de la classe `Arbre` de la page 191 et de sa méthode `Affiche` (qui permet de parcourir en profondeur l'arbre avec un ordre préfixe). On obtient ainsi le programme suivant :

```
class Arbre:
    def __init__(self,racine,filsGauche,filsDroit):
        self.root = racine
        self.G = filsGauche
        self.D = filsDroit
```

Solution de l'exercice type (suite)

```
    def parcours(self , L = []):
        L.append(self.root)
        if self.G:
            self.G.parcours(L)
        if self.D:
            self.D.parcours(L)
        return L

tree = Arbre(3,
        Arbre(
            '+',
            Arbre( 4 , Arbre('*',None,None) , Arbre
    (5,None,None) ),
            Arbre( '/' , Arbre(2,None,None) , Arbre
    ('-',None,None) )
        ),
        Arbre(
            6,
            Arbre( '+' , Arbre(4,None,None) , Arbre
    ('*',None,None) ),
            Arbre( 9 , Arbre('+',None,None) , Arbre
    (2,None,None) )
        ) )

L = tree.parcours()
calcul = ''.join(str(i) for i in L)
print ( eval(calcul) )
```

Une fois l'arbre défini, on construit une liste `L` à l'aide de la méthode `parcours`, dans laquelle on ajoute tous les nœuds de l'arbre dans l'ordre voulu. La liste `L` est alors transformée en chaîne de caractères (avec la méthode `join`) afin de pouvoir l'interpréter comme un calcul (avec la fonction `eval`).

Voir énoncé page 185

INTERROS

1 V/F Vérification de connaissances

5 min

Corrigé p. 209

Les affirmations suivantes sont-elles vraies ou fausses ? Justifier la réponse.

1. Le nombre maximal de nœuds d'un arbre binaire de hauteur h est $2^h - 1$.
2. L'arbre binaire suivant est un ABR.

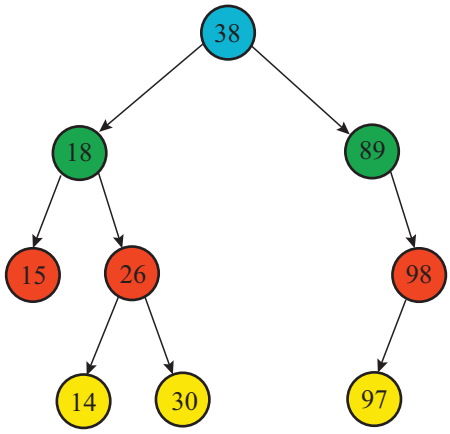


Figure 6.6 – Arbre binaire

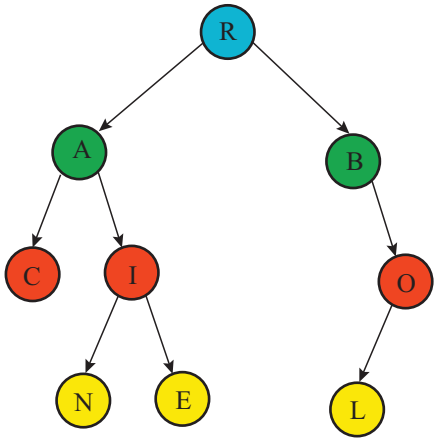


Figure 6.7 – Arbre binaire

3. Le parcours en profondeur par ordre préfixe de l'arbre binaire de la figure 6.7 donne :

R – A – C – I – N – E – B – O – L

4. Le parcours en profondeur par ordre postfixe de l'arbre binaire de la figure 6.7 donne :

C – N – E – I – A – L – O – B – R

5 Le parcours en profondeur par ordre infixe de l'arbre binaire de la figure 6.7 donne :

C – A – N – E – I – B – O – L – R

6 Le parcours en largeur de l'arbre binaire de la figure 6.7 donne :

R – A – B – C – I – O – N – E – L

2 Arbres de Fibonacci

★ 5 min Corrigé p. 210

On définit un *arbre de Fibonacci* comme étant un arbre A_n tel que les sous-arbres gauche et droit de A_n sont respectivement A_{n-1} et A_{n-2}, en admettant que A_0 est un arbre avec une unique racine et A_1 est composé d'une racine ayant un fils gauche et un fils droit.

1 Donner une représentation sagittale de l'arbre A_3.

2 Les arbres de Fibonacci sont-ils des AVL ? Justifier.

3 Écrire une fonction d'affichage

★ 10 min Corrigé p. 211

On considère un arbre implémenté comme ci-dessous :

```
arbre = [
'A' ,
['B', ['E'] , ['F'] ],
['C' , ['G'] , ['H'] , ['I'] ],
['D']
]
```

Écrire une fonction `afficher(arbre,marge)`, où `marge` est un nombre entier, qui affiche l'arbre sous une forme plus facile à lire que cette expression : chaque nœud doit être sur une ligne, avec une marge à gauche qui reflète la position hiérarchique (la *profondeur*) du nœud.

4 Arbre binaire complet

★ 10 min Corrigé p. 211

Un arbre binaire complet est un arbre binaire dans lequel tous les nœuds qui ne sont pas des feuilles ont exactement deux fils.

Dans ce cas, il peut être représenté par une liste de longueur $2^{h+1} - 1$, où h est sa hauteur, où les fils gauche et droit de l'élément d'indice i sont stockés en indices $2i + 1$ et $2i + 2$. Les feuilles manquantes peuvent être représentées par des éléments vides ou des éléments `None` par exemple.

INTERROS

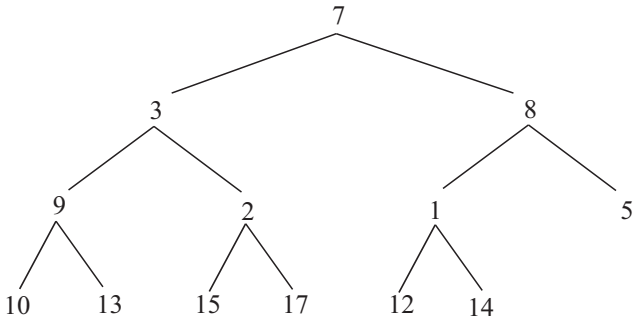


Figure 6.8 – Un exemple d'arbre binaire complet

1. Quelle est, en Python, la liste représentant l'arbre de la figure 6.8 ?
2. Quelle est la formule permettant de calculer la hauteur d'un arbre binaire complet en fonction de la longueur de la liste des nœuds ?
3. Écrire alors une fonction `affiche(arbre)` qui affiche l'arbre en fonction de la liste `arbre` en utilisant un affichage similaire à la figure 6.8 (sans les traits).

5 Labyrinthe et arbre binaire

★★ — 10 min — Corrigé p. 212

On considère le labyrinthe représenté ci-dessous :

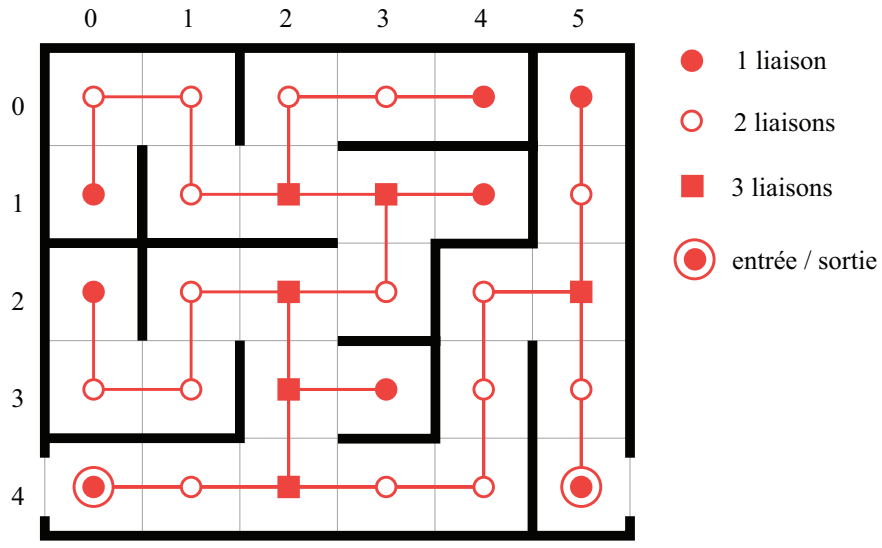


Figure 6.9 – Source : wikipédia

Construire un arbre binaire représentant ce labyrinthe dans lequel chaque case est représentée par un nœud. On partira du nœud noté $(4,0)$ et chaque nœud sera noté (i, j) où i et j représentent respectivement la ligne et la colonne de la case correspondante.

Construction de classes

6 Arborescence d'un répertoire

★★ 20 min Corrigé p. 214

Dans cet exercice, on souhaite créer une classe nous permettant d'afficher l'arborescence d'un répertoire.

On s'aide alors de la classe suivante :

```
class Node:
  def __init__(self,nodeValue):
    self.subNodes = []
    self.value = nodeValue

  def addSubNode(self,node):
    self.subNodes.append(node)

  def view(self,depth=0):
    if self.subNodes:
      print( "{}[{}]".format("  " * depth , self.value)
     )
    else:
      print("  " * depth , self.value)
    depth += 1
    for node in self.subNodes:
            node.view(depth)
```

1. Expliquez la façon dont est défini le constructeur de la classe `Node`.
2. Expliquez les deux méthodes de cette classe.
3. En faisant appel au module `os`, écrire une fonction admettant comme argument le chemin d'un répertoire, et qui retourne un arbre représentant l'architecture du répertoire informé en argument, arbre que l'on pourra afficher à l'aide de la méthode `view` de la classe `Node`. On pourra nommer cette fonction : `getDirectoryNode(path)` .

INTERROS

7 Arbres binaires

★★★ 45 min Corrigé p. 215

Dans cet exercice, on souhaite construire une classe `ArbreBinaire`, initialisée à partir d'une liste.

On se basera sur l'arbre suivant :

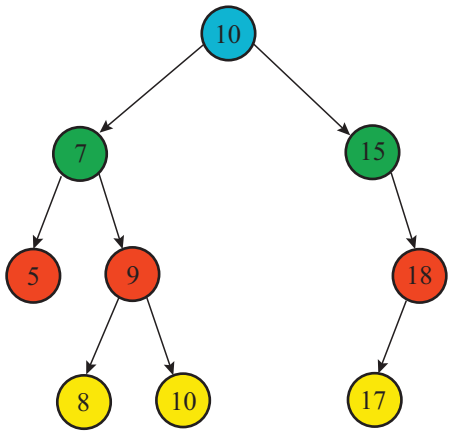


Figure 6.10 – Arbre binaire servant d'exemple

On souhaite que cet arbre soit implémenté par la liste suivante :

```
L =
[ 10 ,
   [ 7 ,
     [5 , [] , []] ,
     [9 , [8 , [] , []] , [10 , [] , []] ]
   ] ,
  [ 15 ,
     [ ],
     [ 18 , [17 , [] , []] , [ ] ]
   ]
]
```

1. Écrire un constructeur de cette classe ainsi qu'une méthode `arbor` de sorte à avoir l'affichage illustré sur la figure 6.11 (voir page suivante).
2. Écrire une méthode `infixe` permettant de parcourir l'arbre avec un ordre infixe et de retourner la liste des nœuds selon cet ordre.
3. Déduire de la question précédente une méthode permettant de vérifier si l'arbre binaire est un ABR.
4. Écrire une méthode `min_max` retournant un couple composé du minimum et du maximum des étiquettes de l'arbre.
5. Écrire une méthode `hauteur` qui retourne la hauteur de l'arbre, puis en déduire une méthode `isAVL` qui permet de vérifier si l'arbre est un arbre AVL.

```
>>> A = ArbreBinaire(L)
>>> A.arbor()
10
   7
     5
       -
       -
     9
       8
         -
         -
       10
         -
         -
   15
     -
     18
       17
         -
         -
       -
```

Figure 6.11 – Illustration de l'affichage souhaité

8 Une classe « Arbre » complète

★★★ 45 min Corrigé p. 217

Dans cet exercice, on se propose de construire une classe `Arbre` relativement complète en se basant sur ce qui a été vu en cours, notamment les algorithmes. Cette classe admet le constructeur suivant :

```
class Arbre:
    def __init__(self,racine,*enfants):
        self.root = racine
        self.enfants = enfants
```

Nous souhaitons implémenter un arbre comme page ci-contre.

1. Implémenter une méthode `Prefixe` qui retourne une liste donnant tous les nœuds de l'arbre dans un ordre préfixe, à l'aide d'un parcours en profondeur.
2. Implémenter une méthode `Postfixe` qui retourne une liste donnant tous les nœuds de l'arbre dans un ordre postfixe, à l'aide d'un parcours en profondeur.
3. Implémenter une méthode `Largeur` qui retourne une liste donnant tous les nœuds de l'arbre en ayant parcouru ce dernier en largeur.

```
tree =
Arbre('A',
    Arbre(
        'B',
        Arbre( 'C' ,
            Arbre('D' , Arbre('Z') ) ,
            Arbre('E' ,
                Arbre('X') , Arbre('Y') , Arbre('P')
            )
        ),
        Arbre( 'F' , Arbre('G') , Arbre('H') )
    ),
    Arbre(
        'I',
        Arbre( 'J' , Arbre('K') , Arbre('L') ),
        Arbre( 'M' , Arbre('N') , Arbre('O') )
    )
)
```

1 Implémenter alors une méthode `AfficheListe(chaine)` de sorte à avoir :

```
>>> tree.AfficheListe('prefixe')
['A', 'B', 'C', 'D', 'Z', 'E', 'X', 'Y', 'P', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']
>>> tree.AfficheListe('postfixe')
['Z', 'D', 'X', 'Y', 'P', 'E', 'C', 'G', 'H', 'F', 'B',
'K','L', 'J', 'N', 'O', 'M', 'I', 'A']
>>> tree.AfficheListe('largeur')
['A', 'B', 'I', 'C', 'F', 'J', 'M', 'D', 'E', 'G', 'H',
'K','L', 'N', 'O', 'Z', 'X', 'Y', 'P']
```

2 Implémenter une méthode `Affiche` qui permet d'afficher l'arbre avec une marge, comme dans le paragraphe 3.2.2 page 191.

9 Une classe pour ABR

★★★ 60 min Corrigé p. 219

Dans cet exercice, on se propose de construire une classe pour représenter des arbres binaires de recherche.

On part alors du constructeur suivant :

```
class ABR:
    def __init__(self , data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None
```

où :

- `data` représente la valeur d'un nœud ;
- `left` et `right` désignent respectivement les fils gauche et droit ;
- `parent` est le parent du nœud.

Ainsi, par défaut, un nœud n'a ni fils ni parent.

On initialiser un ABR en écrivant :

```
A = ABR(150)
```

où « 150 » est la valeur de la racine de l'arbre.

Pour tester les méthodes que nous allons implémenter, nous représenterons l'arbre suivant :

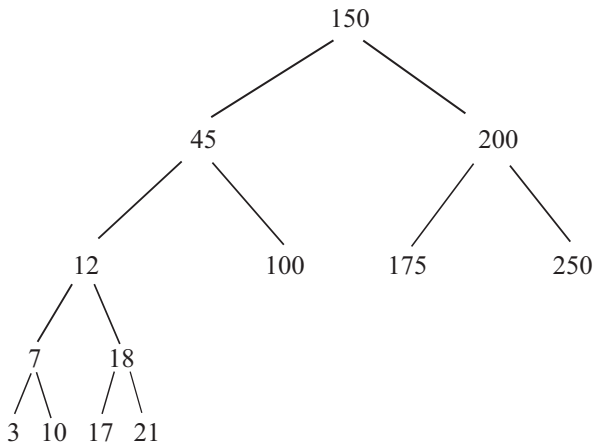


Figure 6.12 – ABR à implémenter

1 Écrire une méthode `insert` qui permet d'insérer une valeur dans l'ABR.

Suite des questions page suivante.

INTERROS

2 Écrire une méthode `affiche` permettant d'afficher l'arbre horizontalement, c'est-à-dire que l'arbre de la figure 6.12 s'affichera sous la forme :

```
                250
        200
                175
150
                100
        45
                                21
                        18
                                17
                12
                                10
                        7
                                3
```

3 Écrire une méthode `hauteur` qui renvoie la hauteur de l'arbre.

4 Écrire une méthode `trouve(data)` qui retourne le nœud qui contient la valeur `data`, et qui retourne « `None` » si la valeur n'est pas trouvée dans l'ABR.

5 Écrire une méthode `liste` qui renvoie une liste ordonnée des valeurs contenues dans l'arbre.

6 Écrire deux méthodes `is_left_child` et `is_right_child` qui vérifient si un nœud est le fils gauche ou droit de son parent, ou pas. Ces méthodes doivent renvoyer un booléen.

7 Écrire une méthode `minimum` qui retourne le nœud ayant la plus petite valeur.

8 Le successeur d'un nœud N est le nœud ayant la plus petite valeur supérieure à N.

Écrire une méthode `successeur(self)` qui retourne le successeur d'un nœud.

9 Écrire enfin une méthode `delete` qui permet de supprimer le nœud dont la valeur est un nombre passé en argument.

On pourra s'aider des méthodes `trouve` et `successeur`.

1 V/F Vérification de connaissances

Enoncé p. 200

1 *Faux*. Il y a en effet au maximum :

- 1 nœud à la racine, niveau 0 ;
- 2 nœuds au niveau suivant, niveau 1 ;
- 2^2 nœuds au niveau 2 ;
- etc. jusqu'à 2^h nœuds au niveau h.

D'après la formule $1 + q + q^2 + \cdots + q^n = \dfrac{q^{n+1} - 1}{q - 1}$, avec $q = 2$ et $n = h$, on obtient au maximum $2^{h+1} - 1$ nœuds.

2 *Faux*. En effet, pour qu'un arbre soit binaire de recherche, il est nécessaire que la clé de tout nœud du sous-arbre droit issu d'un nœud N_k soit supérieure à la clé de N_k. Or, « 14 » est la clé d'un nœud du sous-arbre droit issu du nœud de clé « 18 » et $14 < 18$.

3 *Vrai*. Le parcours en profondeur par ordre préfixe donne les nœuds dans l'ordre dans lesquels on les rencontre pour la première fois :

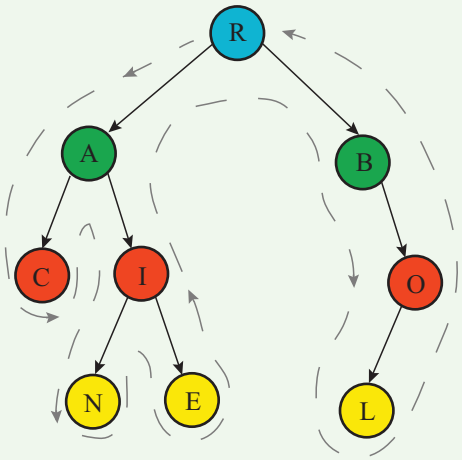


Figure 6.13 – Parcours en profondeur de l'arbre

Ce qui donne :

R – A – C – I – N – E – B – O – L.

À RETENIR

PRÉfixe = PREmière fois

4 *Vrai*. Quand on liste les nœuds dans l'ordre postfixe dans un parcours en profondeur, on les énumère dans l'ordre dans lequel on les rencontre pour la dernière fois. Cela donne bien :

C – N – E – I – A – L – O – B – R.

5 *Faux*. Quand on liste les nœuds dans l'ordre infixe dans un parcours en profondeur, si le nœud a un fils gauche, on le liste lors du deuxième passage sinon, lors du premier. Cela donne alors :

C – A – N – I – E – R – B – L – O.

6 *Vrai*. Dans un parcours en largeur, les nœuds sont listés par hauteur (par niveau) : on commence par la racine, on continue avec les nœuds-fils de hauteur 1, puis on continue avec ceux de hauteur 2, etc.
Cela donne bien :

R – A – B – C – I – O – N – E – L.

2 Arbres de Fibonacci

Enoncé p. 201

1 Pour connaître la représentation sagittale de A_3, il nous faut avant tout obtenir A_2.

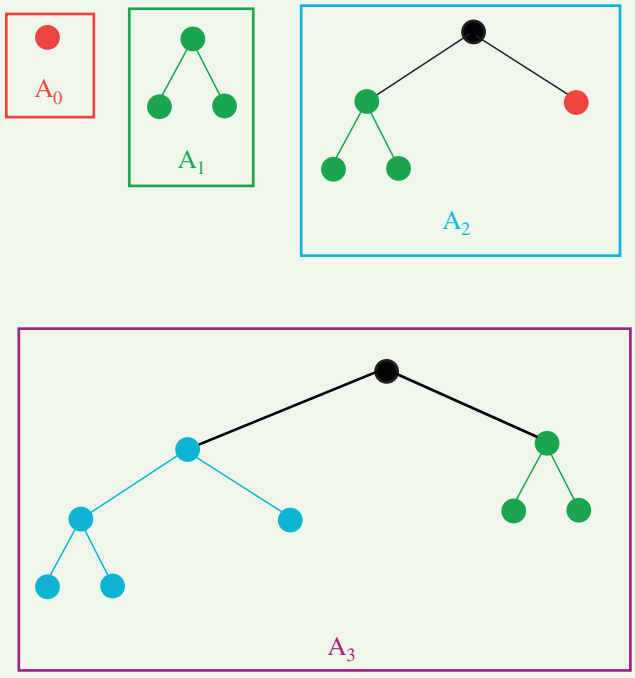


Figure 6.14 – Construction des premiers arbres de Fibonacci

2 Rappelons qu'un arbre AVL est un arbre dans lequel, pour tout nœud, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1. Les arbres de Fibonacci sont donc des AVL. En effet, pour tout entier naturel k, A_k a une hauteur égale à k. Donc la différence de hauteur entre A_k et A_{k+1} est égale à 1. Or, A_n est composé de A_{n-1} (fils gauche) et A_{n-2} (fils droit), de hauteur $n-1$ et $n-2$, hauteurs différant donc de 1.

3 Écrire une fonction d'affichage

Enoncé p. 201

Il s'agit ici de parcourir la liste `arbre` et, pour chaque élément de cette liste, d'afficher son premier élément avec une marge égale à la valeur de la variable `marge`, puis de parcourir les listes qui suivent et de procéder de même en multipliant la marge par *n*, où *n* est la position de la liste.

On voit alors une récursivité dans le traitement des informations.

Un programme possible est donc le suivant :

```
def afficher(arbre,marge=0):
    print(" " * marge + arbre[0])
    for subtree in arbre[1:]:
        afficher(subtree,marge+2)

arbre = [
  'A' ,
  ['B', ['E'] , ['F'] ],
  ['C' , ['G'] , ['H'] , ['I'] ],
  ['D']
]

afficher(arbre)
```

Ce programme affiche :

```
A
  B
    E
    F
  C
    G
    H
    I
  D
```

4 Arbre binaire complet

Enoncé p. 201

1 L'arbre de la figure 6.8 se représente par la liste :

```
[7,3,8,9,2,1,5,10,13,15,17,12,14,None,None]
```

Il suffit de lire les nœuds de gauche à droite, en descendant à partir de la racine.

CORRIGÉS

2 Notons $n = 2^{h+1} - 1$ la longueur de la liste des nœuds.

Dans notre exemple, $15 = 2^4 - 1$ (il y a 15 éléments dans la liste, et la hauteur de l'arbre est égale à 3).

Alors,

$$n = 2^{h+1} - 1 \iff 2^{h+1} = n + 1$$
$$\iff h = \log_2(n+1) - 1.$$

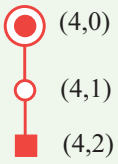
3 Une implémentation possible est la suivante :

```
from math import log

def affiche(arbre):
    hauteur = int( log( len(arbre) + 1 , 2 ) )

    for i in range(hauteur):
        ligne = ''
        # ecartement entre les noeuds sur cette
    ligne
        ecart_ligne = ( 2**(hauteur-i+1) - 3) * ' '
        # écart au début
        ecart_debut= ( 2**(hauteur-i) - 2) * ' '
        for j in range( 2**i - 1 , min( 2**(i+1) - 1
     , len(arbre) ) ):
            ligne += "{0:^3}".format( str( arbre[j]
    ) )
            if j < min( 2**(i+1) - 1 , len(arbre) )
    - 1:
                ligne += ecart_ligne
        print('\n'+ecart_debut+ligne)

arbre = [7,3,8,9,2,1,5,10,13,15,17,12,14,None,None]
affiche(arbre)
```

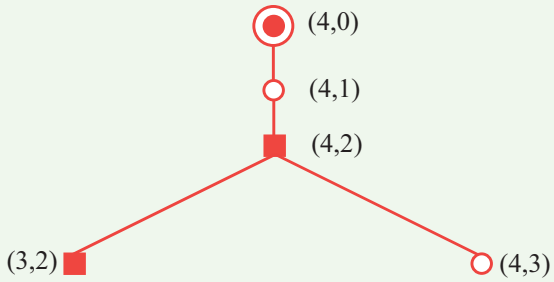
5 Labyrinthe et arbre binaire

Enoncé p. 202

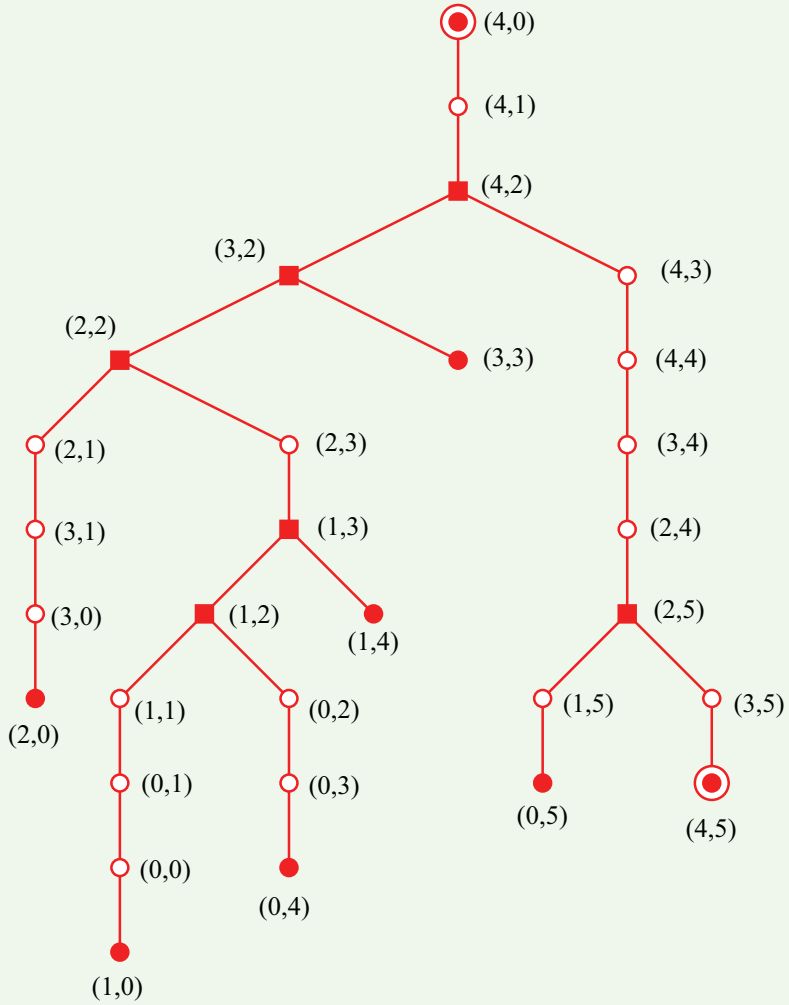
On part de la case (4,0) et nous avons un unique choix : aller à la case (4,1), et ensuite, il n'y a pas d'autre choix que d'aller en (4,2) :



Maintenant, nous avons le choix entre aller en (3,2) ou en (4,3) :



On continue ainsi jusqu'à chaque feuille pour obtenir l'arbre suivant :



CORRIGÉS

6 Arborescence d'un répertoire

Enoncé p. 203

1 Le constructeur de la classe `Node` permet d'initialiser une liste nommée `self.subNodes`, ainsi qu'une variable `self.value` prenant pour valeur celle de l'argument renseigné lors de l'appel à cette classe.

Les noms des variables étant explicites, on peut supposer que la liste contiendra les nœuds fils et que la variable `self.value` représente l'étiquette du nœud.

2 La méthode `addSubNode` admet un argument : `node`. A priori, elle ajoute à la liste `self.subNodes` la valeur de ce dernier argument.

Cette méthode a donc pour but d'ajouter à la liste des nœuds déjà existante un nœud supplémentaire.

Dans la définition de la méthode `view`, on commence par tester si la liste `self.subNodes` n'est pas vide, auquel cas on affiche la valeur du nœud entre crochets, avec une marge dépendant de sa profondeur dans l'arbre. Dans le cas contraire, si la liste est vide, on affiche simplement la valeur du nœud, sans crochet (pour signifier que c'est une feuille).

On augmente ensuite la profondeur et on passe aux éventuels enfants par récursivité : on fait appel à cette méthode en prenant cette fois-ci chaque nœud fils.

3 Voici une proposition de fonction répondant au cahier des charges :

```
def getDirectoryNode(path):
  node = Node(os.path.split(path)[1])
  if os.path.isdir(path):
    for itemName in os.listdir(path):
      fullPath = os.path.join(path,itemName)
      node.addSubNode(getDirectoryNode(fullPath))
  return node
```

Dans cette fonction, on commence par définir la racine de notre arbre : c'est le répertoire dont on veut trouver l'architecture. On prend soin ici de ne garder que le nom du répertoire et non le chemin complet, à l'aide de l'instruction `os.path.split(path)[1]`.

Le test qui suit nous sert à voir si le chemin désigne un répertoire (`if os.path.isdir(path)`), auquel cas on ajoute à l'arbre créé un sous-nœud pour chaque répertoire et fichier qu'il contient.

Ainsi, pour afficher l'arborescence d'un répertoire, on écrira par exemple :

```
>>> tree = getDirectoryNode("C:\\test_directory")
>>> tree.view()
```

```
[test_directory]
    [dir_1]
       [dir_11]
          file_11a.txt
          file_11b.txt
       [dir_12]
          [dir_123]
             fil_123a.txt
    [dir_2]
⋮
```

Téléchargez le programme complet.

7 Arbres binaires

Enoncé p. 204

1 Voici une proposition de script répondant au cahier des charges de cette première question :

```
class ArbreBinaire:
    def __init__(self , L):
        if L:
            self.root = L[0]
            self.filsGauche = ArbreBinaire( L[1] )
            self.filsDroit = ArbreBinaire( L[2] )
        else:
            self.root = None
            self.filsGauche = None
            self.filsDroit = None

    def arbor(self , space = 0):
        if self.root != None:
            print(" "*space,self.root)
            if self.filsGauche.root == None:
                spaces = " " * (space + 2)
                print(spaces,"-")
            else:
                self.filsGauche.arbor(space + 2)
            if self.filsDroit.root == None:
                spaces = " " * (space + 2)
                print(spaces,"-")
            else:
                self.filsDroit.arbor(space + 2)
```

L'idée consiste ici à construire un arbre binaire basé sur une liste de la forme `[racine , filsGauche , filsDroit]`, où `filsGauche` et `filsDroit` peuvent aussi être des arbres binaires.

2 Le parcours infixe de l'arbre peut-être obtenu à l'aide de la méthode suivante (inspirée de l'algorithme du cours page 195).

```
def infixe(self , L = []):
    if self.filsGauche:
        self.filsGauche.infixe(L)
    if self.root != None:
        L.append(self.root)
    if self.filsDroit:
        self.filsDroit.infixe(L)
    return L
```

3 Cette question pourrait être perçue comme très compliquée si nous ne remarquons pas que le parcours infixe d'un ABR donne toujours une liste croissante de nombres. On s'aide alors de la méthode `infixe` implémentée à la question précédente, ce qui donne :

```
def isABR(self):
    L = self.infixe()
    prev = None
    for node in L:
        if prev == None:
            prev = node
        elif node < prev:
            return False
    return True
```

4 **MÉTHODE**

Pour déterminer le minimum et le maximum d'un arbre (dans le cas où les étiquettes sont des nombres), il suffit de se baser sur la liste donnée par son parcours infixe : le minimum est le premier élément et le maximum, son dernier.

Cela donne la méthode suivante :

```
def min_max(self):
    L = self.infixe()
    return L[0] , L[-1]
```

5 Pour calculer la hauteur d'un arbre, on ajoute 1 à la hauteur maximale des sous-arbres, ce qui donne :

```
def hauteur(self):
    return 1 + max(
        self.filsGauche.hauteur() if self.filsGauche
     else -2,
        self.filsDroit.hauteur()  if self.filsDroit
     else -2
    )
```

Pour vérifier si l'arbre est bien un AVL, il faut s'assurer que pour tout nœud, les hauteurs des sous-arbres diffèrent d'au plus 1. On va donc s'aider de la méthode `hauteur` :

```
def isAVL(self):
    hD =  self.filsDroit.hauteur() if self.filsDroit
     != None else 0
    hG =  self.filsGauche.hauteur() if self.
    filsGauche != None else 0
    if abs(hD-hG) > 1:
        return False

    if self.filsDroit != None:
        d = self.filsDroit.isAVL()
    if not d: return d
  if self.filsGauche != None:
    g = self.filsGauche.isAVL()
    if not g: return g
  return True
```

On s'assure ici que l'arbre donné dans l'énoncé n'est pas un AVL.

Téléchargez la classe complète.

8 Une classe « Arbre » complète

Enoncé p. 205

1 Voici une méthode `Prefixe` basée sur l'algorithme de la page 193 :

```
def Prefixe(self , L = []):
    L.append(self.root)
    for child in self.enfants:
        child.Prefixe( L )
    return L
```

CORRIGÉS

2 Voici une méthode `Postfixe` basée sur l'algorithme de la page 194 :

```
def Postfixe(self , L = []):
    for child in self.enfants:
        child.Postfixe( L )
    L.append(self.root)
    return L
```

3 Voici une méthode `Largeur` basée sur l'algorithme de la page 196 :

```
def Largeur(self):
    F , L = deque([]) , []
    L.append(self.root)
    F.append(self)

    while F:
        for child in F[0].enfants:
            L.append(child.root)
            F.append(child)
        F.popleft()

    return L
```

4 Une méthode d'affichage sous forme de liste est alors la suivante :

```
def AfficheListe(self , ordre):
    if ordre == 'prefixe':
        print( self.Prefixe() )
    elif ordre == 'postfixe':
         print( self.Postfixe() )
    elif ordre == 'largeur':
        print( self.Largeur() )
```

5 En s'inspirant du script permettant l'affichage vu en page 191 du cours, on obtient la méthode suivante :

```
def Affiche(self , space = 0):
    spaces = " " * space
    print(spaces , self.root)
    for child in self.enfants:
        child.Affiche(space + 2)
```

Téléchargez la classe complète.

9 Une classe pour ABR

Enoncé p. 206

1 Quand on insère une valeur dans un ABR, il faut voir si celle-ci est inférieure ou égale à sa racine, auquel cas la valeur sera placée dans le sous-arbre gauche, ou au contraire supérieure, auquel cas elle sera placée dans le sous-arbre droit.

Si la valeur est inférieure ou égale à la valeur de la racine, on regarde s'il existe un fils gauche : si tel n'est pas le cas, on définit le fils gauche par la valeur à insérer, et on définit le parent comme étant la racine. Si, au contraire, le fils gauche existe, alors on répète l'opération en prenant pour racine le fils gauche : on fait donc appel à la récursivité.

Pour une valeur strictement supérieure à la valeur de la racine, on utilise un raisonnement analogue. On a alors la méthode écrite page suivante.

```
def insert(self, data):
    if data <= self.data:
        if self.left == None:
            self.left = ABR(data)
            self.left.parent = self
        else:
            self.left.insert(data)
    else:
        if self.right == None:
            self.right = ABR(data)
            self.right.parent = self
        else:
            self.right.insert(data)
```

2 Le nœud le plus à droite de l'arbre sera affiché sur la première ligne avec une marge dépendant de sa hauteur. À la ligne suivante devra s'afficher le parent du nœud précédemment affiché, et ensuite, le fils droit du parent.

Cette façon de raisonner nous pousse à utiliser la récursivité ; en effet, tant qu'il y a un fils droit, on fait appel à la méthode `affiche` en augmentant la hauteur de 1 ; quand il n'y en a plus, on affiche le nœud et on passe au fils gauche. D'où la méthode suivante :

```
def affiche(self, h = 0):
    if self.right:
        self.right.affiche(h + 1)
    spaces = ' ' * 7 * h
    print(spaces,self.data)
    if self.left:
        self.left.affiche(h + 1)
```

3 Pour calculer la hauteur d'un arbre, on ajoute 1 à la plus grande des hauteurs entre celles des sous-arbres gauche et droit, d'où la méthode récursive suivante :

```
def hauteur(self):
    return 1 + max(
        self.left.hauteur() if self.left else -1,
        self.right.hauteur() if self.right else -1
    )
```

4

```
def trouve(self , data):
    if data == self.data:
        return self
    elif data < self.data and self.left != None:
            return self.left.trouve(data)
    elif data > self.data and self.right != None:
        return self.right.trouve(data)
    return None
```

Il n'y a rien de compliqué pour cette méthode : si `data` coïncide avec la valeur du nœud courant, on retourne le nœud ; si elle est inférieure, on va chercher du côté gauche et dans le cas contraire, du côté droit. Si rien n'est trouvé au final, on retourne `None`.

5 Là encore, rien de bien compliqué : on initialise une liste vide en argument de la méthode, on y ajoute la valeur du nœud courant et si ce nœud a des fils, on fait appel à cette méthode en passant la liste en argument pour chacun des fils, ce qui donne :

```
def liste(self , L = []):
    L.append(self.data)
    if self.left != None:
        self.left.liste(L)
    if self.right != None:
        self.right.liste(L)
    return sorted(L)
```

6 Une possibilité est celle donnée page suivante.

L'idée consiste ici à comparer le parent du nœud courant et le fils gauche (ou droit) du parent.

« `self is self.parent.left` » est déjà un booléen : il représente si le nœud courant est bien le fils gauche de son parent.

Ainsi, « `self.parent and self is self.parent.left` » s'assure que le parent existe ET que le nœud courant est son fils gauche.

Il en est bien sûr de même pour le fils droit.

```
# l'enfant est-il le fils gauche de son parent ?
def is_left_child(self):
    return self.parent and self is self.parent.left

# l'enfant est-il le fils droit de son parent ?
def is_right_child(self):
    return self.parent and self is self.parent.right
```

7 Comme nous sommes dans un ABR, le nœud ayant la plus petite valeur est celui qui est situé le plus à gauche, d'où la solution suivante :

```
def minimum(self):
    node = self
    while node.left:
        node = node.left
    return node
```

On parcourt l'arbre tant qu'un fils gauche existe, et si tel n'est pas le cas, c'est que l'on est sur le nœud qui a la plus petite valeur.

8 Tous les nœuds du sous-arbre gauche de N ont une valeur plus petite, donc son successeur se trouve dans son sous-arbre droit. D'après les propriétés de l'ABR, ça sera la plus petite valeur de ce sous-arbre droit : son dernier fils gauche s'il y a des descendants à gauche, sinon son premier nœud.

Si le nœud n'a pas de sous-arbre droit, il faut remonter dans l'arbre pour trouver une valeur plus grande car c'est le seul endroit où on pourra en trouver une. Le successeur sera le premier des parents tel que N apparaît dans son sous-arbre gauche. Cela donne alors la méthode suivante :

```
def successeur(self):
    if self.right:
        return self.right.minimum()
    node = self
    while node.is_right_child():
        node = node.parent
    return node.parent
```

9 C'est sans doute la méthode la plus difficile à écrire ; il faut donc s'armer de patience et de persévérance pour arriver à nos fins.

On commence par rechercher la valeur dans l'arbre avec la méthode `trouve` car si la valeur n'existe pas, inutile de faire quoi que ce soit.

Ensuite, en définissant la méthode par : `def delete(self,data)`, on teste la valeur du nœud : `if self.data == data`.

Le cas le plus simple est quand le nœud n'a pas de fils : dans ce cas, on le supprime tout simplement en spécifiant que le parent n'a plus de fils gauche ou droit selon sa position.
Si le nœud a un fils unique, alors il est remplacé par son fils.
Si le nœud a deux fils, on remplace le nœud par son successeur :

- son successeur sera forcément dans son sous-arbre droit car le nœud a deux enfants, donc un sous-arbre droit ;
- si son successeur a des enfants, ils ne pourront être qu'à sa droite car son successeur est dans notre cas la plus petite valeur du sous-arbre droit ;
- on remplace donc la valeur du nœud courant par celle de son successeur ;
- on corrige les branches de l'arbre pour aller de notre nœud vers le fils de son successeur (et vice-versa) pour rendre le successeur orphelin et le sortir de l'arbre ;
- on supprime son successeur.

Si la valeur `data` est inférieure à celle du nœud courant, on s'oriente vers le sous-arbre gauche et si elle lui est supérieure, vers le sous-arbre droit.
Cela donne la méthode suivante `delete` dans la classe à télécharger :

```
def delete(self , data):
    if not self.trouve(data):
        return
    if data == self.data:
        # si le noeud n'a pas de fils
        if self.left == None and self.right == None:
            if self.parent.left.data == data:
                self.parent.left = None
            else:
                self.parent.right = None
            del self
       # si le noeud a un fils...
        elif self.left == None:
            if self.parent.left != None and self.
    parent.left.data == data:
                self.parent.left = self.right
            else:
                self.parent.right = self.right
            del self
    # (suite dans le programme complet...)
```

Téléchargez la classe complète

Chapitre 7

Bases de données

Plan du chapitre

Exercice type

Une base de données a la structure relationnelle suivante :

```
famille (id_prenom,prenom)
genres (id_genre,designation)
livres (id_livre,titre,auteur,id_genre,id_prenom)
```

- La table « famille » désigne l'ensemble des membres de la famille Numos ;
- La table « genres » représente l'ensemble des genres des livres que possède la famille (policier, science-fiction, etc.) ;
- La table « livres » désigne l'ensemble des livres de la famille Numos. L'attribut « id_prenom » sera vide si le livre n'est pas en la possession d'un membre de la famille.

Les attributs des tables sont explicites.

1 Pour chaque table, quel attribut peut être une clé primaire ?

2 Pour la table « livres », existe-t-il un attribut qui peut être défini comme clé étrangère ?

3 La famille Numos est composée de Mathilde, François, Nathan, Ludovic et Jeanne.
Écrire une requête SQL permettant d'insérer une de ces informations dans la table « famille ».

4 On suppose qu'un membre de la famille peut être en possession de plusieurs livres en même temps.
Écrire une requête SQL permettant de trouver tous les livres en possession de Mathilde.

5 Écrire une requête SQL permettant de calculer le nombre total de livres indisponibles.

Voir corrigé page 246

COURS

PROBLÉMATIQUE

Comment stocker, manipuler et partager un volume d'informations toujours plus important ?

1 Systèmes de Gestion de Bases de Données (SGBD)

1.1 Introduction

Définition 1

Un Système de Gestion de Bases de Données (SGBD) est un outil informatique permettant la sauvegarde, l'interrogation, la recherche et la mise en forme de données.

Un SGBD est donc un ensemble de logiciels systèmes qui permettent aux programmateurs d'insérer, de modifier et de rechercher des données spécifiques dans une grande masse d'informations partagées par plusieurs utilisateurs.

Ces informations sont généralement enregistrées sur des disques magnétiques (par exemple, des serveurs).

Un SGBD donne l'impression à chaque utilisateur qu'il est le seul à travailler avec les données.

Oracle Database, 4D, Microsoft SQL Server, SQLite ou MySQL sont des exemples de SGBD très répandus. SQLite et MySQL sont des logiciels libres (open source), et sont par conséquent très utilisés, aussi bien par le grand public que par les professionnels.

1.2 Fonctions d'un SGBD

En plus des fonctions primaires citées dans la définition précédente, un SGBD :

- assure le partage des données ;
- vérifie qu'une opération s'effectue entièrement ou pas du tout : c'est ce que l'on nomme l'*atomicité* ;
- protège les données contre tout incident (détérioration, suppression accidentelle, etc.) ;
- optimise les performances (temps de recherche minimisé par exemple).

Les SGBD permettent la description des données (définition des types par des noms, formats, ...) de manière séparée de leur utilisation (mise à jour et recherche). Ils permettent aussi de retrouver les caractéristiques d'un type de données à partir de son nom (par exemple, comment est décrit un produit).

1.3 Les différents modèles de SGBD

Il existe cinq modèles de SGBD :

- *le modèle hiérarchique* : les données sont classées hiérarchiquement, selon une arborescence descendante (arbre). Ce modèle utilise des pointeurs entre les différents enregistrements. Il s'agit du premier modèle de SGBD.

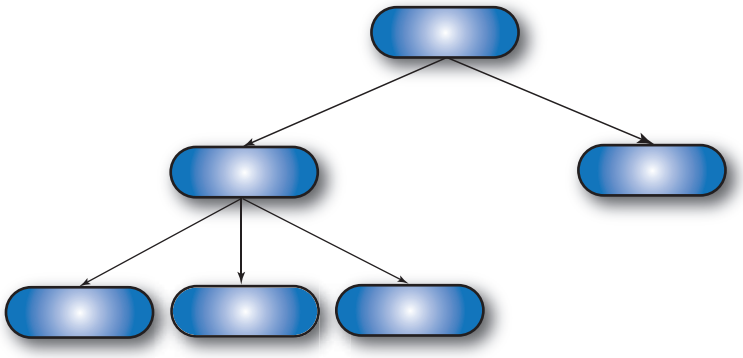


Figure 7.1 – Modèle hiérarchique d'un SGBD

- *le modèle réseau* : ce modèle ressemble au modèle hiérarchique à ceci près qu'il n'est plus nécessairement descendant.

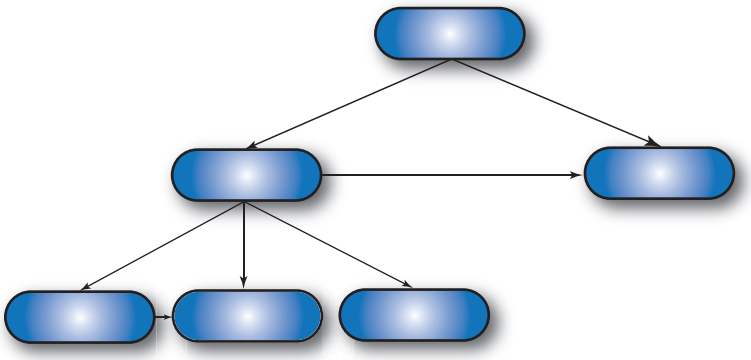


Figure 7.2 – Modèle réseau d'un SGBD

- *le modèle relationnel (SGBDR)* : les données sont enregistrées dans des tableaux à deux dimensions (lignes et colonnes) ;
- *le modèle déductif* : similaire au modèle relationnel, mais la manipulation des tables se fait différemment (nous ne donnerons pas de détails sur les calculs utilisés car hors programme) ;
- *le modèle objet (SGBDO)* : les données sont stockées sous forme de classes.

COURS

Depuis la fin des années 1990, le modèle relationnel est le plus répandu (environ trois quarts des bases de données utilisent ce modèle). Dans la suite de cet ouvrage, nous ne nous intéresserons qu'à ce modèle.

2 Le modèle relationnel

2.1 Historique

Le modèle relationnel a été imaginé dans les années 1970 par l'informaticien britannique Edgar Frank Codd. Avant cela, les modèles de bases de données ne permettaient pas de décrire de façon satisfaisante les relations entre deux données à l'aide de pointeurs logiques. Dix années de recherches ont été nécessaires avant de publier l'article « A Relational Model of Data for Large Shared Data Banks » (*un modèle de données relationnel pour de grandes banques de données partagées*), CACM 13, No. 6, June 1970.

2.2 Représentations

Comme nous l'avons dit dans la section précédente, le modèle relationnel consiste à représenter les données dans des tableaux, que l'on appelle *tables*. Chaque table est une sorte de classe, qui admet donc des attributs.

Prenons l'exemple d'un site internet, qui demande à ses utilisateurs trois données afin de pouvoir leur créer un compte : nom, prénom et email. On peut alors imaginer une classe *user*, représentant un utilisateur ou une utilisatrice, classe comportant les trois attributs : surname, name et email. Cette classe est alors représentée par la table :

user		
surname	name	email

Tableau 7.1 – Structure de la table représentant les utilisateurs

Une fois la structure de la table connue, nous pouvons y insérer les données, par exemple :

user		
surname	name	email
Dupont	Jean	jean.dupont@free.fr
Durand	Anne	anne.dur@orange.fr
⋮	⋮	⋮

Tableau 7.2 – Insertion de données dans la table représentant les utilisateurs

2.3 Définitions

Définitions 2

- Dans une table, chaque ligne constitue un *enregistrement*.
- Les *attributs* d'une table sont les noms de ses colonnes.
- Le *degré* d'une table est le nombre de ses attributs.
- Le *domaine* d'un attribut est son type : entier, flottant, chaîne de caractères, booléen, ...
- La *clé primaire* d'une table est l'attribut qui permet d'identifier de manière unique (sans aucun risque de doublon) un enregistrement de cette table.
- Une *clé étrangère* est un attribut d'une table qui fait référence à la clé primaire d'une autre table. Une clé étrangère permet de mettre en relation un enregistrement d'une table (appelée *table fille*) qui la contient avec un enregistrement de la table référencée (appelée *table parent*).

Exemple : reprenons la table 7.2 précédente et ajoutons-y une clé primaire *id* :

user			
id	surname	name	email
1	Dupont	Jean	jean.dupont@free.fr
2	Durand	Anne	anne.dur@orange.fr
⋮	⋮	⋮	⋮

Tableau 7.3 – Table avec clé primaire

Les attributs de cette table sont donc : *id*, *surname*, *name* et *email*.
Le degré de la table est donc égal à 4 car il y a quatre attributs.
Le domaine de l'attribut *id* est : entier, et celui des autres attributs est : chaîne de caractères.

Imaginons maintenant que ce site internet propose plusieurs abonnements : « Free », « Basic » et « Premium ».
On peut alors créer une seconde table nommée « abonnement » représentant les abonnements :

abonnement	
id_abo	type
1	Free
2	Basic
3	Premium

Tableau 7.4 – Table des abonnements

COURS

On peut alors mettre en relation les tables « user » et « abonnement » en insérant une clé étrangère dans la première table :

user				
id	surname	name	email	id_abo
1	Dupont	Jean	jean.dupont@free.fr	3
2	Durand	Anne	anne.dur@orange.fr	1
⋮	⋮	⋮	⋮	

Tableau 7.5 – Table des utilisateurs avec clé étrangère

On met ici en relation les deux tables en considérant que l'attribut *id_abo* de la table « user » correspond à l'attribut *id_abo* de la table « abonnement ».
Ainsi, si l'on regarde le premier enregistrement de la table « user », on peut voir que Jean Dupont a souscrit à un abonnement Premium, c'est-à-dire à l'enregistrement de la table « abonnement » dont l'attribut *id_abo* est 3.

2.4 Schématisation

Pour schématiser la relation qu'il peut y avoir entre deux tables, on peut utiliser la représentation suivante (en reprenant les deux tables de l'exemple précédent) :

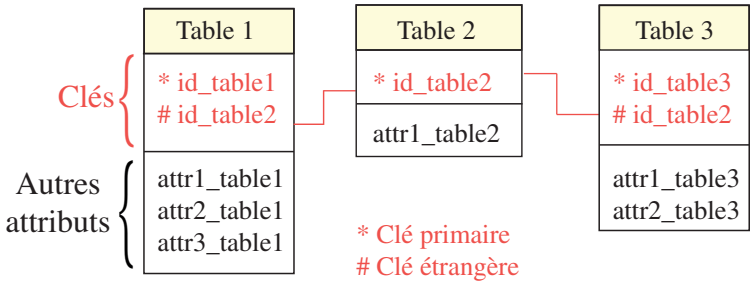


Figure 7.3 – Schématisation de la relation entre deux tables

La relation entre les deux tables est indiquée par un trait qui les lie.

Dans chacune des tables, on peut indiquer la présence d'une clé primaire en précédant l'attribut d'un astérisque (*), et la présence d'une clé étrangère en précédant l'attribut d'un dièse (#).

On peut aussi, pour plus de clarté, séparer les clés des autres attributs.

Définition 3

Une *base de données* est un ensemble de tables, dont certaines peuvent être mises en relation.

3 Algèbre relationnelle

3.1 Opérateurs ensemblistes

L'idée ici est de reprendre ce qui se fait en mathématiques, et plus précisément en *théorie des ensembles*.

3.1.1 - Union

Définition 4

On considère deux tables T_1 et T_2 d'une base de données.
L'*union* des deux tables T_1 et T_2 est l'ensemble des éléments qui sont dans T_1 ou dans T_2 :

$$T_1 \cup T_2 = \{e \in T_1 \text{ ou } e \in T_2\}.$$

Exemple : nous avons représenté ci-dessous deux tables T_1 et T_2 ainsi que leur union.

T_1
nom
Dupont
Durand
Dufon
Dupain

T_2
nom
Dugont
Dufon
Dulong
Dupont

$T_1 \cup T_2$
nom
Dupont
Durand
Dufon
Dupain
Dugont
Dulong

Tableau 7.6 – Deux tables et leur union

3.1.2 - Intersection

Définition 5

On considère deux tables T_1 et T_2 d'une base de données.
L'*intersection* des deux tables T_1 et T_2 est l'ensemble des éléments qui sont dans T_1 et dans T_2 :

$$T_1 \cap T_2 = \{e \in T_1 \text{ et } e \in T_2\}.$$

Exemple : en reprenant les tables T_1 et T_2 de l'exemple précédent (tableau 7.6), l'intersection est la table ci-contre :

$T_1 \cap T_2$
nom
Dupont
Dufon

COURS

3.1.3 - Différence

Définition 6

On considère deux tables T_1 et T_2 d'une base de données.
La *différence* $T_1 - T_2$ est l'ensemble des éléments qui sont dans T_1 mais pas dans T_2 :

$$T_1 - T_2 = \{e \in T_1 \text{ et } e \notin T_2\}.$$

Exemple : en reprenant les tables T_1 et T_2 du tableau 7.6 page 229, la différence est la table suivante :

$T_1 - T_2$
nom
Durand
Dupain

3.1.4 - Produit cartésien

Définition 7

On considère deux tables T_1 et T_2 d'une base de données.
Le *produit cartésien* $T_1 \times T_2$ est l'ensemble :

$$T_1 \times T_2 = \{(e_1, e_2) \mid e_1 \in T_1 \text{ et } e_2 \in T_2\}.$$

Exemple : si l'on considère les deux tables suivantes,

T_1	
id	surname
1	Dupont
2	Durand

Tableau 7.7 – Table 1

T_2	
stockage	price
201	15
41	7

Tableau 7.8 – Table 2

le produit cartésien de ces deux tables est :

$T_1 \times T_2$			
id	surname	stockage	price
1	Dupont	201	15
1	Dupont	41	7
2	Durand	201	15
2	Durand	41	7

Tableau 7.9 – Produit cartésien

3.2 Opérations

3.2.1 - Sélection

Définition 8

On considère une table T.
La *sélection* est l'opération qui consiste à ne retenir que les enregistrements de T qui vérifient une condition donnée.

Exemple : étant donnée la table suivante,

id	surname	name	genre
1	Dupont	Jean	Homme
2	Dupont	Jeanne	Femme
3	Durand	Camille	Femme
4	Dufon	Pierre	Homme

Tableau 7.10 – Table d'utilisateurs

on ne souhaite que les enregistrements dont l'attribut « genre » est « Homme ».
Le résultat est alors :

id	surname	name	genre
1	Dupont	Jean	Homme
4	Dufon	Pierre	Homme

Tableau 7.11 – Résultat de la requête : « genre = Homme »

3.2.2 - Projection

Définition 9

On considère une table T.
La *projection* est l'opération qui consiste à ne retenir que certains attributs de T.

Exemple : reprenons la table 7.10 de l'exemple précédent. La projection de cette table sur les attributs « id » et « name » est :

id	name
1	Jean
2	Jeanne
3	Camille
4	Pierre

Tableau 7.12 – Projection de la table sur « id » et « name »

COURS

3.2.3 - Jointures

Une jointure permet de créer une liaison entre deux ou plusieurs tables pour pouvoir récupérer un résultat bien précis. Le résultat est en quelque sorte une fusion virtuelle de plusieurs tables. Chaque enregistrement de chaque table est retourné dans le résultat en fonction de la jointure utilisée... car il existe plusieurs types de jointures.

Définition 10 : jointure interne

Soient T_1 et T_2 deux tables. La *jointure interne* de ces deux tables sur un attribut donné A est une table formée en ne prenant que les enregistrements de T_1 et T_2 qui ont une valeur de A commune.

On peut représenter cette jointure par le schéma suivant :

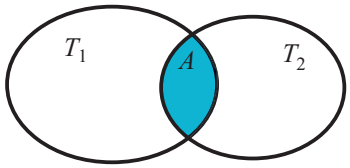


Figure 7.4 – Schématisation d'une jointure interne

Exemple : en considérant les deux tables suivantes,

user			
id	id_town	surname	name
1	1	Dupont	Jean
2	2	Durand	Agathe
3	1	Dufon	Jérôme

Tableau 7.13 – Table T_1

town	
id_town	name_town
1	Bordeaux
3	Paris

Tableau 7.14 – Table T_2

leur jointure sur l'attribut « id_town » est :

id	id_town	surname	name	id_town	name_town
1	1	Dupont	Jean	1	Bordeaux
3	1	Dufon	Jérôme	1	Bordeaux

Tableau 7.15 – Jointure interne de T_1 et T_2

Définition 11 : jointure externe gauche

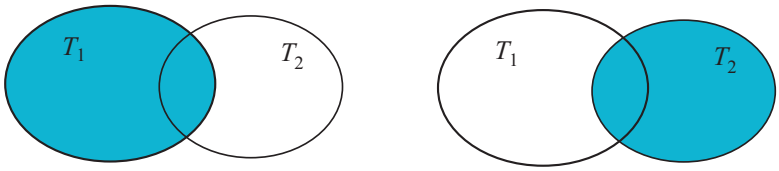
Étant données deux tables T_1 et T_2, la *jointure externe gauche* de T_1 avec T_2 est une table réunissant

- les enregistrements de T_1 et T_2 qui ont une valeur de A commune (comme pour une jointure interne) ;
- les enregistrements de T_1 (table de gauche) pour lesquels l'attribut A de T_1 ne correspond avec aucun attribut A de T_2.

Définition 12 : jointure externe droite

Étant données deux tables T_1 et T_2, la *jointure externe gauche* de T_1 avec T_2 est une table réunissant

- les enregistrements de T_1 et T_2 qui ont une valeur de A commune (comme pour une jointure interne) ;
- les enregistrements de T_2 (table de droite) pour lesquels l'attribut A de T_2 ne correspond avec aucun attribut A de T_1.



Jointure externe gauche — Jointure externe droite

Figure 7.5 – Représentation des jointures externes

Exemple : reprenons les tables 7.13 et 7.14 de l'exemple précédent.
Alors, la jointure externe gauche donnera :

id	id_town	surname	name	id_town	name_town
1	1	Dupont	Jean	1	Bordeaux
3	1	Dufon	Jérôme	1	Bordeaux
2	2	Durand	Agathe	NULL	NULL

Tableau 7.16 – Jointure externe gauche de T_1 et T_2

Dans cette jointure gauche, nous conservons, par définition, tous les enregistrements de la table T_1 (gauche). Or l'enregistrement où apparaît Agathe comporte un *id_town* qui ne correspond à aucun *id_town* dans la table T_2. Le SGBD utilise alors la valeur NULL pour exprimer l'absence de correspondance dans T_2.

COURS

La jointure externe droite, quant à elle, donnera la table suivante :

id	id_town	surname	name	id_town	name_town
1	1	Dupont	Jean	1	Bordeaux
3	1	Dufon	Jérôme	1	Bordeaux
NULL	NULL	NULL	NULL	3	Paris

Tableau 7.17 – Jointure externe droite de T_1 et T_2

Dans la jointure droite, nous conservons tous les enregistrements de la table T_2, et comme aucun enregistrement de la table T_1 ne fait référence à la valeur de *id_town* de « Paris », les valeurs « NULL » sont données aux attributs *id*, *id_town*, *surname* et *name*.

4 Le langage SQL

4.1 Introduction

Développé par IBM dans les années 1970, le langage SQL (pour « Structured Query Language », ou « langage de requête structurée » en français) est rapidement devenu le standard des langages de bases de données relationnelles. Il permet de manipuler la structure d'une base de données et d'y rechercher, insérer, modifier ou supprimer les données à l'aide d'une vingtaine d'instructions dont la syntaxe ressemble à celle de phrases ordinaires en anglais. SQL peut s'utiliser de manière interactive, ou à l'intérieur d'un langage hôte tel que C, Fortran, ou Cobol, par exemple.

4.2 Environnements nécessaire à l'utilisation de SQL

4.2.1 - Généralités

Pour travailler sur des bases de données avec SQL, plusieurs outils sont nécessaires :

- un serveur pour héberger la base,
- un SGBD pour la manipuler.

Si l'on souhaite utiliser SQL de manière interactive, il nous faudra un logiciel d'édition ou une interface graphique pour entrer manuellement les requêtes.

Au Lycée, vous aurez principalement affaire à SQLite et MyPHP, dont nous allons rapidement étudier les environnements.

4.2.2 - Environnement SQLite

SQLite est une solution légère et gratuite, ce qui en fait la plus utilisée au monde. Quelque soit votre système d'exploitation, vous pouvez télécharger les outils nécessaires sur :

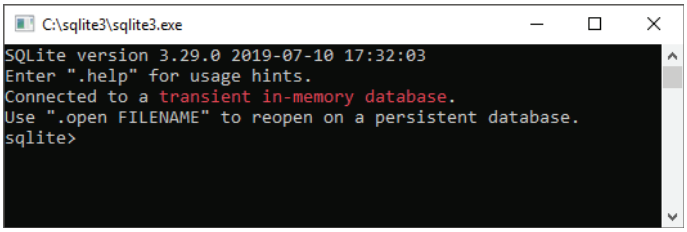
https ://www.sqlite.org/download.html

Par exemple, sous Windows 10, vous pouvez télécharger le fichier :

`sqlite-tools-win32-x86-xxxxxxx.zip`

qui contient trois exécutables : `sqlite3`, `sqldiff` et `sqlite3_analyser`.

Lorsque le programme principal sqlite3 est exécuté, une console permet de lancer les requêtes SQL nécessaires à la manipulation des bases de données.



Cependant, cette console n'étant pas toujours très pratique, on peut aussi utiliser une interface graphique telle que DB Browser ou SQLiteSpy.

4.2.3 - Environnement MySQL

MySQL étant un logiciel libre, il s'intègre parfaitement dans un environnement composé du système d'exploitation Linux, du serveur Apache et du langage PHP. Cet environnement est appelé *LAMP*, acronyme de « Linux - Apache - MySQL - PHP ».

Sous Windows, l'environnement correspondant s'appelle *WAMP* (W pour Windows), et sous Mac *OS MAMP* (M pour Mac).

Notons que le langage PHP est le complément idéal à MySQL pour la création de pages Web dynamiques via un serveur HTTPS.

PhpMyAdmin est l'interface la plus connue pour utiliser MySQL.

Une fois l'environnement installé avec MySQL ou SQLite, nous sommes prêts à manipuler des bases de données avec SQL.

4.3 Les requêtes SQL

Retrouvez une explication des requêtes SQL en vidéo avec Nathan Live.

4.3.1 - Fonctions de base

La syntaxe des fonctions de base en SQL est très simple.

COURS

- Création d'une table nommée « matable » dans la base de données :

```
CREATE TABLE matable (
  attribut1 type,
  attribut2 type,
  ...
)
```

- Suppression de la table :

```
DROP TABLE matable
```

- Insertion d'un enregistrement dans une table :

```
INSERT INTO
    matable
VALUES (
    'valeur 1',
    'valeur 2',
    ...
)
```

- Modification d'une ou plusieurs valeurs d'attributs :

```
UPDATE
    matable
SET
    attribut1 = 'valeur',
    attribut2 = 'valeur'
WHERE
    condition
```

- Suppression d'un enregistrement dans une table :

```
DELETE FROM
    matable
WHERE
    condition
```

- Sélection d'un ou plusieurs attributs dans une table :

```
SELECT
    attribut1,
    attribut2
FROM
    matable
```

- Classification des données selon un ou plusieurs attributs :

```
SELECT
    *
FROM
    matable
ORDER BY
    attribut
```

À noter que « * » désigne la totalité des attributs.

- Suppression des doublons sur un attribut :

```
SELECT DISTINCT
    attribut
FROM
    matable
```

Exemple : à l'aide d'une interface graphique (DB Browser pour SQLite ou PhpMyAdmin pour MySQL), il est facile de créer une base de données nommée « gestion » ainsi que deux tables :

→ « restaurants » d'attributs : id, nom, rue, cp, ville, email, tel et id_gerant ;

→ « gerants » d'attributs : id, nom, prenom, email et tel.

- Pour changer l'adresse email de l'enregistrement dont l'id est 1, on écrira :

```
UPDATE
    restaurants
SET
    email = "chezmomo@free.fr"
WHERE
    id = 1;
```

COURS

- Pour supprimer tous les enregistrements concernant la ville de Nantes dans la table « restaurants », on écrira :

```
DELETE FROM
    restaurants
WHERE
    ville = "Nantes"
```

4.3.2 - Jointures

Il existe trois types de jointures : INNER (jointure interne), OUTER (jointure externe) et CROSS (produit cartésien).

4.3.2.1 - Jointure interne

La syntaxe pour une jointure interne est la suivante :

```
SELECT
    champ1,
    ...
FROM
    table1
INNER JOIN
    table2
ON
    table1.champA = table2.champB
```

Cette requête joint les deux tables `table1` et `table2` et on choisit un critère de sélection avec « ON ». La condition qui vient après peut être une égalité ou autre chose.

4.3.2.2 - Jointures externes

La syntaxe pour une jointure externe est la suivante :

```
SELECT
    champ1,
    ...
FROM
    table1
[LEFT|RIGHT] OUTER JOIN
    table2
ON
    table1.champA = table2.champB
```

Si aucune correspondance n'est trouvée pour un attribut, la valeur « NULL » est donnée pour l'attribut. Ainsi, si l'on dispose de deux tables `clients` (avec les attributs *nom_client* et *ville_client*) et `distributeurs` (avec les attributs *nom_distrib* et *ville_distrib*), la requête :

```
SELECT
    clients.nom_client,
    distributeurs.nom_distrib,
    clients.ville_client
FROM
    clients
LEFT OUTER JOIN
    distributeurs
ON
    clients.ville_client = distributeurs.ville_distrib
```

donne une table comme ci-dessous :

nom_client	nom_distrib	ville_client
Jean Naimart	CARREFOUR	BORDEAUX
Anne Onime	LECLERC	BORDEAUX
Paul Tronc	AUCHAN	PARIS
Chloé Opieu	NULL	MARSEILLE

La dernière ligne signifie que l'enregistrement correspondant à Chloé Opieu n'est mis en relation avec aucun enregistrement de la table `distributeurs`.

Maintenant, si on teste la requête :

```
SELECT
    clients.nom_client ,
    distributeurs.nom_distrib ,
    clients.ville_client
FROM
    clients
RIGHT OUTER JOIN
    distributeurs
ON
    clients.ville_client = distributeurs.ville_distrib
```

s'afficheront uniquement les entrées de la seconde table qui satisfont la condition, avec éventuellement des « NULL » si certains enregistrements de la table `distributeurs` ne sont mis en relation avec aucun enregistrement de la table « clients ».

COURS

Astuce : quand les noms des tables sont « longs », on peut créer des raccourcis. Par exemple, la requête précédente peut s'écrire :

```
SELECT
    C.nom_client , D.nom_client , C.ville_client
FROM
    clients C
RIGHT OUTER JOIN
    distributeurs D
ON
    C.ville_client = D.ville_distrib
```

Remarquez que nous avons mis un raccourci tout de suite après avoir fait appel à une table : « distributeurs D » (donc ici, la lettre `D` est le raccourci pour cette table) et « clients C ».

4.3.2.3 - Produit cartésien

Pour effectuer un produit cartésien, on utilisera une syntaxe telle que :

```
SELECT
    *
FROM
    clients
CROSS JOIN
    distributeurs
```

Le CROSS JOIN met en relation *tous* les enregistrements de la table de gauche avec tous ceux de la table de droite sans rechercher aucune égalité ni logique particulière.

4.3.3 - Opérations d'agrégation

4.3.3.1 - Ajouter

Dans certains cas, il est nécessaire d'ajouter toutes les entrées d'une colonne (d'un attribut). On pourra le faire à l'aide de la fonction d'agrégation SUM.

Exemple : considérons la table « commandes » suivante :

commandes		
id_commande	id_client	total
1	12	125
2	7	35
3	8	50

Tableau 7.18 – Table « commandes »

La requête :

```
SELECT SUM(montant) AS somme
FROM commandes
```

retourne :

somme
210

où « 210 » représente le total des valeurs de l'attribut `montant` des commandes. Il n'est pas indispensable de spécifier d'alias (`AS`), mais c'est assez pratique dans les requêtes plus complexes. Dans notre exemple, nous avons choisi l'alias « somme » pour désigner la somme obtenue.

4.3.3.2 - Compter

Dans certains cas, savoir combien une colonne comporte d'entrées s'avère utile. On le fera avec la fonction d'agrégation COUNT.

Exemple : prenons à nouveau la table « commandes » du tableau 7.18 de l'exemple précédent.

Si on considère la requête suivante :

```
SELECT COUNT(*) AS nb
FROM commandes
WHERE montant < 100
```

alors, elle affiche :

nb
2

car il y a deux enregistrements où la valeur de l'attribut « montant » est strictement inférieure à 100.

4.3.3.3 - Effectuer une moyenne

Cela se fait à l'aide de la fonction d'agrégation AVG (abréviation de *average*, qui signifie « moyenne » en anglais).

Exemple : en reprenant la table représentée dans le tableau 7.18, la requête :

```
SELECT AVG(montant) AS
    moy
FROM commandes
```

affiche :

moy
66.6667

où « 66,6667 » représente donc la moyenne des commandes.

COURS

4.3.3.4 - Maximum et minimum

Connaître le maximum et le minimum des valeurs d'un attribut se fait avec les fonctions d'agrégation MAX et MIN.

Exemple : toujours avec la table représentée par le tableau 7.18, la requête :

```
SELECT
    MIN(C.montant) AS minimum ,
    MAX(C.montant) AS maximum
FROM commandes C
```

affiche :

minimum	maximum
25	125

4.3.4 - Complément : créer une vue

Il sera de temps en temps utile de nommer la table obtenue à l'issue d'une requête ; c'est ce que l'on appelle *créer une vue* de la requête. Prenons par exemple la requête précédente qui permet de calculer le maximum et le minimum, et créons une vue que l'on nomme « mavue »

```
CREATE VIEW mavue AS
    SELECT
        MIN(C.montant) AS minimum ,
        MAX(C.montant) AS maximum
    FROM commandes C
```

On peut ainsi utiliser la table (la vue) dans une autre requête, par exemple pour calculer l'écart entre le maximum et le minimum :

```
SELECT maximum - minimum FROM mavue
```

5 Manipuler les bases de données avec Python

Pour manipuler des bases de données avec Python, on fera appel au module `sqlite3` ou au module `mysql.connector` selon le serveur SQL que l'on aura installé.

5.1 Avec SQLite 3

Retrouvez une mise en pratique de SQLite 3 en Python en vidéo avec Nathan Live.

5.1.1 - Préliminaires

Il faut d'abord faire appel au module `sqlite3` :

```
import sqlite3
```

Ensuite, il faut établir une connexion à une base de données :

```
conn = sqlite3.connect('gestion.db')
```

Ici, on donne le nom « conn » à notre connexion, et on donne le nom « gestion.db » à notre base de données. À ce stade, cette dernière n'existe pas encore. Pour la créer, on écrit :

```
cursor = conn.cursor()
```

On donne en quelque sorte le curseur à notre base.

Cette opération provoque la création d'un fichier `gestion.db` dans le répertoire qui contient le script Python. C'est dans ce fichier que l'on va stocker les tables créées ainsi que leurs enregistrements.

ATTENTION

Le format du fichier est propre à Sqlite (mélange de binaire et de texte) et ne doit pas être manipulé directement via un éditeur de texte. C'est le SGBD qui maintient ce fichier à jour en y exécutant les requêtes qui lui sont transmises (ici par Python).

5.1.2 - Passer une requête SQL

Pour exécuter une requête SQL, on utilisera la commande `cursor.execute` :

```
cursor.execute("""<requête>""")
```

Exemples

- Pour créer une table :

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS restaurants (
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    nom TEXT,
    ville TEXT) """)
```

COURS

- Pour insérer un enregistrement :

```
cursor.execute("""
INSERT INTO restaurants(nom, ville) VALUES(?, ?)""",
("Chez Momo", "Marseille"))
```

5.1.3 - Insérer un enregistrement à partir d'un dictionnaire

On utilise toujours la méthode `execute` mais la syntaxe est légèrement différente, comme le montre l'exemple de la page suivante.
Exemple :

```
dico = {"nom" : "Chez Fanny", "ville" : "Marseille"}
cursor.execute("""
INSERT INTO restaurants(nom, ville)
VALUES(:nom, :ville)""", dico)
```

5.1.4 - Insérer plusieurs enregistrements à partir d'une liste

On utilise ici la méthode `executemany` comme dans l'exemple suivant.

Exemple :

```
liste = []
liste.append(("Le cochon vert","Paris"))
liste.append(("Le gazon maudit","Paris"))
cursor.executemany("""
INSERT INTO restaurants(nom, ville) VALUES(?, ?)""", liste)
```

5.1.5 - Afficher les enregistrements

On pourra utiliser une syntaxe comme celle de l'exemple suivant.

Exemple :

```
print('Tous les enregistrements :')
cursor.execute("""
SELECT id, nom, ville FROM restaurants""")
for row in cursor:
    print('{0} : {1}, {2}'.format(row[0], row[1], row[2]))
```

5.1.6 - Enregistrements avec la clause WHERE

Quand on souhaite afficher les enregistrements en fonction de la valeur d'un attribut avec la clause WHERE, la syntaxe est un peu spéciale, comme le montre l'exemple page ci-contre.

Exemple :

```
print('Les restaurants de la ville de Marseille :')
cursor.execute("""
SELECT id, nom FROM restaurants
WHERE ville = %s""", ("Marseille", ))
for row in cursor:
    print('{0} : {1}'.format(row[0], row[1]))
```

5.1.7 - Valider les modifications

Avant de clore la connexion, il faut s'assurer que les éventuelles modifications ou insertions soient prises en compte. On utilise alors la méthode `commit()` :

```
conn.commit()
```

5.1.8 - Clôture de la connexion

Quand on a fini de se servir de la base de données, il ne faut pas oublier de clore la connexion.

```
conn.close()
```

Télécharger le programme Python complet sur sqlite3.

Pour plus d'informations sur le module `sqlite3`, n'hésitez pas à vous servir de la commande *help* :

```
>>> import sqlite3
>>> help(sqlite3)
```

5.2 Avec mysql.connector

Par défaut, ce module (auquel on doit faire appel si l'on utilise MySQL) n'est pas installé. Il faudra donc se renseigner pour l'installer sur votre distribution Python (avec la commande `pip` ou, sous Anaconda, la commande `conda`).

Si l'on part de la base de données « gestion » définie précédemment, voici un exemple.

```
import mysql.connector
# connexion à la base de données "gestion"
conn = mysql.connector.connect(
    host="localhost", user="root",
    password="", database="gestion")
```

COURS

```
cursor = conn.cursor()
# Afficher selon un critère sur la valeur de "ville"
cursor.execute(
    """SELECT id, nom, email
    FROM restaurants
    WHERE ville = %s""", ("Marseille", ))
rows = cursor.fetchall()
for row in rows:
    print('{0} : {1} - {2}'.format(row[0], row[1], row[2]))
conn.close()
```

Solution de l'exercice type

1 Pour la table « famille », l'attribut « id_prenom » peut être une clé primaire. En effet, une clé primaire est un attribut dont les valeurs sont toutes distinctes. Dans notre situation, pour la table « famille », tous les enregistrements ont un id_prenom différent.

Il en est de même pour la table « genres » : « id_genre » peut être une clé primaire.

Pour la table « livres », l'attribut « id_livres » peut être une clé primaire.

2 Pour la table « livres », l'attribut « id_prenom » est lié à la table « famille » donc il peut être défini comme clé étrangère. Mais « id_genre » est lié à la table « genres » donc peut aussi être une clé étrangère.

3 Une requête SQL permettant d'insérer une des informations dans la table « famille » est :

```
INSERT INTO
    famille (prenom)
VALUES
    ('Mathilde')
```

Pour insérer tous les enregistrements, il suffit de répéter quatre autres fois cette requête en changeant bien entendu le prénom.

On peut aussi, plus simplement, faire la requête suivante :

```
INSERT INTO
    famille (prenom)
VALUES
    ('Mathilde'), ('François'),
    ('Nathan'),   ('Ludovic'),
    ('Jeanne')
```

Solution de l'exercice type

4 Dans cette question, la difficulté réside dans le fait que dans la table « films », les prénoms n'apparaissent pas ; seuls les identifiants sont enregistrés. Il faut donc avant tout rechercher l'identifiant de Mathilde à l'aide de la requête :

```
SELECT
    id_prenom AS prenom
FROM
    famille
WHERE
    prenom = 'Mathilde'
```

Cette dernière requête renvoyant l'identifiant, il faut s'en servir dans la requête SQL permettant de trouver tous les livres en possession de Mathilde. On obtient alors :

```
SELECT *
FROM livres
WHERE
    id_prenom = (
        SELECT
            id_prenom AS prenom
        FROM
            famille
        WHERE
            prenom = 'Mathilde'  )
```

Autre possibilité (avec une jointure) :

```
SELECT *
FROM livres
   INNER JOIN famille
   ON famille.id_prenom = livres.id_prenom
WHERE
   famille.prenom = 'Mathilde'
```

5 Une requête SQL permettant de calculer le nombre total de livres indisponibles est celle proposée page suivante.

Il s'agit ici de compter tous les enregistrements de la table « livres » dont la valeur de l'attribut « id_prenom » n'est pas vide (ce qui signifie qu'un membre de la famille possède le livre correspondant, et donc que ce dernier est indisponible).

...

Solution de l'exercice type (suite)

```
SELECT
    COUNT(*)
FROM
    livres
WHERE
    id_prenom IS NOT NULL
```

L'attribut « id_prenom » étant par définition un entier, sa valeur sera nécessairement « NULL » si aucune valeur ne lui a été attribuée.

Voir énoncé page 223

1 QCM Vérification de connaissances

5 min

Corrigé p. 256

On dispose d'une base de données comportant les tables suivantes :

```
clients
    (idClient,designation,adresse,cp,ville,email,tel)
produits
    (idProduit,designation,prix,idEntrepot)
commandes
    (idCommande,date,idProduit,idClient,quantite)
entrepots
    (idEntrepot,designation,adresse,cp,ville,superficie)
```

Le nom des champs (entre les parenthèses) est choisi de façon implicite pour qu'il n'y ait pas d'ambiguïté.

1 La requête :

```
SELECT * FROM clients ORDER BY ville DESC
```

a affiche le nom de tous les clients par ordre alphabétique de leur ville ;

b affiche le nom de tous les clients selon un ordre alphabétique inverse de leur ville ;

c affiche tous les champs de la table « clients » par ordre alphabétique inverse de leur nom ;

d affiche tous les champs de la table « clients » par ordre alphabétique inverse du nom de leur ville.

2 La requête :

```
SELECT *
FROM (
    clients INNER JOIN entrepots
    ON clients.ville = entrepots.ville)
```

a affiche tous les champs des tables « clients » et « entrepots » qui ont la même ville ;

b affiche tous les champs de la table « clients » où le contenu du champ « ville » est identique à celui d'au moins un champ de la table « entrepots » ;

c affiche tous les champs de la table « entrepots » où le contenu du champ « ville » est identique à celui d'au moins un champ de la table « clients » ;

d affiche les contenus du champ « ville » communs aux tables « entrepots » et « clients ».

INTERROS

3 La requête :

```
SELECT *
FROM (
    clients INNER JOIN entrepots
    ON clients.ville = entrepots.ville)
```

est équivalente à :

a `SELECT * FROM clients WHERE clients.ville = entrepots.ville`
b `SELECT * FROM entrepots WHERE clients.ville = entrepots.ville`
c `SELECT * FROM clients,entrepots WHERE clients.ville = entrepots.ville`
d `SELECT ville FROM clients,entrepots WHERE clients.ville = entrepots.ville`

4 La requête :

```
SELECT *
FROM
    clients LEFT OUTER JOIN entrepots
    ON (clients.ville = entrepots.ville)
```

a affiche uniquement les clients dont la ville coïncide avec celle d'un entrepôt ;
b affiche tous les clients ;
c affiche tous les entrepôts ;
d n'affiche rien.

5 La requête :

```
SELECT
    SUM(commandes.total) AS somme
FROM
    commandes
WHERE
    commandes.date > '2019-09-02'
```

a affiche le total des commandes effectuées après le 2019-09-02 ;
b affiche le total de chaque commande effectuée après le 2019-09-02 ;
c affiche toutes les entrées de la table « commandes » où la date est supérieure à 2019-09-02 en effectuant la somme des totaux petit à petit ;
d n'affiche rien.

2 V/F Vérification de connaissances

10 min — Corrigé p. 256

Les affirmations suivantes sont-elles vraies ou fausses ? Justifier la réponse.

On se place dans une base de données nommée « bddPerso ».

1. Une ligne d'une table de `bddPerso` est appelée une entrée.
2. Une table de `bddPerso` est définie par la structure relationnelle suivante :

```
clients (idClient , nom , prenom)
```

   Alors « nom » est un attribut de la table « clients ».
3. Dans la table « clients » de la question précédente, on peut prendre « nom » comme clé primaire.
4. On définit une autre table de la manière suivante :

```
connexions (idCommande , idClient , date)
```

   où « idClient » fait référence à la clé du même nom de la table « clients ».

   « idClient » est alors une clé étrangère de la table « connexions ».

3 Festival de musique ★

15 min — Corrigé p. 257

On dispose d'une base de données d'un festival de musique, où il y a plusieurs représentations. Un ou plusieurs musiciens peuvent participer à une représentation, mais un musicien ne peut participer qu'à une seule représentation.

La structure de la base de données est la suivante :

```
Representation (idRep , titreRep , lieu)
Musicien (idMus , nom , idRep)
Programmer (Date , idRep, tarif)
```

Écrire la requête SQL permettant d'afficher :

1. La liste des titres des représentations.
2. La liste des titres des représentations ayant lieu sur le lieu nommé « Dionysos ».
3. La liste des noms des musiciens et les titres des représentations auxquelles ils participent.
4. La liste des titres des représentations, les lieux et les tarifs d'une date précisée.
5. Le nombre des musiciens qui participent à la représentation numéro 7.
6. Les représentations et leurs dates dont le tarif ne dépasse pas 30 €.

INTERROS

4 Les employés du département

★ — 15 min — Corrigé p. 258

On dispose d'une base de données dont la structure est la suivante :

```
Departements (DNO, DNOM, DIR, VILLE)
Employes (ENO, ENOM, PROF, DATEEMB, SAL, COMM, DNO)
```

Donner les requêtes SQL qui permettent d'obtenir :

1. la liste des employés ayant une commission (attribut « COMM » déclaré comme booléen) ;
2. les noms, emplois et salaires des employés obtenus en classant les emplois dans l'ordre alphabétique, et pour chaque emploi, en classant les salaires du plus grands au plus petit ;
3. le salaire moyen des employés.
4. le salaire moyen dans le département nommé « Production » (`DNOM = 'Production'`).

5 Obtenir la ligne d'un maximum

★★ — 10 min — Corrigé p. 259

On dispose de la table :

```
commandes (idCommande,date,idClient,montant)
```

Quelle requête permet d'afficher la (les) enregistrements où le prix est le plus grand ?

6 Notes annuelles des étudiants

★★ — 20 min — Corrigé p. 260

On considère le modèle relationnel suivant concernant la gestion des notes annuelles d'une promotion d'étudiants :

```
ETUDIANT
    (NumEtudiant, Nom, Prenom)
MATIERE
    (CodeMat, LibelleMat, CoeffMat)
EVALUER
    (NumEval, #NumEtudiant, #CodeMat, Date, Note)
```

Les clés étrangères sont précédées d'un « # » et les clés primaires sont soulignées.

Exprimez en SQL les requêtes suivantes.

1. Quel est le nombre total d'étudiants ?
2. Quelles sont, parmi l'ensemble des notes, la note la plus haute et la note la plus basse ?

3. Quelles sont les moyennes de chaque étudiant dans chacune des matières ?
4. Quelles sont les moyennes de la classe par matière ?
5. Quelle est la moyenne générale de chaque étudiant ?
6. Quelle est la moyenne générale de la promotion ?
7. Quels sont les étudiants qui ont une moyenne générale supérieure ou égale à la moyenne générale de la promotion ?

Aide : on pourra utiliser « GROUP BY » pour plusieurs questions, qui regroupe les entrées. Par exemple, si l'on considère la table suivante :

clients		
id	nom	montant
1	Marisol Pleureur	125.65
2	Jacques Umul	45.7
3	Marisol Pleureur	100.5
4	Jacques Umul	78.6

alors, la requête :

```
SELECT
    nom ,
    SUM(montant) AS total
FROM
    clients
GROUP BY
    nom
```

calculera le total des valeurs des attributs « montant » en les regroupant par nom, et donnera la table :

nom	total
Jacques Umul	124.3
Marisol Pleureur	226.15

7 Dans un lycée

★★ 15 min Corrigé p. 263

On considère une base de données dont le modèle relationnel est le suivant :

```
eleves (idEleve,nom,prenom,age)
controles (idControle,#idEleve,#idMatiere,date,note)
matieres (idMatiere,nomMatiere,coef)
```

Les champs soulignés sont des clés primaires et les champ marqués d'un « # » désignent des clés étrangères.

1. Indiquer une requête SQL permettant d'afficher le nombre de contrôles passés par chaque élève pour une discipline donnée (par exemple, NSI).
   Aide : pour afficher des résultats par élève, on peut utiliser `GROUP BY`. Par exemple :

   ```
   SELECT * FROM table GROUP BY colonne
   ```

   On souhaite avoir comme résultat une table ayant pour attributs les nom, prénom et nombre de contrôles.
2. Créer une requête qui permet d'afficher la moyenne de chaque élève par matière.
   On souhaite obtenir une table ayant pour attributs les nom et prénom des élèves, les noms des matières, leurs coefficients et la moyenne de la matière pour chaque élève.
3. Créer une requête qui permet d'afficher la moyenne générale de chaque élève en s'aidant de la vue obtenue à l'aide de la requête de la question précédente.

8 Un cas pratique : site web marchand ★★★ 30 min Corrigé p. 265

Un webmaster souhaite créer une base de données nommée « venteAdonf » pour un site internet marchand `vente-a-donf.com` dont la clientèle est uniquement française (pour simplifier les formats des enregistrements).

Dans cette base de données, il devra y avoir les quatre tables suivantes :

- `users (idUser,nom,prenom,email,rue,cp,ville,tel)`
  où « cp » désigne le code postal de sa ville de résidence, les autres attributs étant explicites ;
- `modeles (idModele,nomModele)`
- `articles (idArticle,nom,descriptif,idModele,prix)`
  où « idModele » est une clé étrangère relative à la colonne `idModele` de la table `modeles`.
- `panier (idPanier,idUser,idArticle,idModele,quantite, total)`
  Ici, « total » représente le résultat de « quantite * prix de l'article ».

On décide de mettre en clés primaires les premières colonnes de chaque table.

1. Écrire un fichier SQL nommé « venteAdonf-create.sql » permettant de créer ces quatre tables.
   On rappelle qu'un tel fichier est composé de quatre requêtes séparées par un point-virgule.

2 Voici les premiers articles mis en vente :

Nom	Descriptif	Modèle	Tarif
Mug	Magnifique mug personnalisable avec votre photo.	Unique	7,99 €
T-shirt « I Kiffe Beef »	T-shirt unique en son genre.	S / M / L / XL / XXL	24,99 €

Écrire un fichier SQL nommé « venteAdonf-insertArt.sql » permettant d'insérer ces informations dans la base de données.

3 Le jour de la mise en ligne du site, deux personnes ont passé commande :

Nom	Dupont	Aimaun
Prénom	Jean	Anne
Email	jdupont@free.fr	amz@free.fr
Rue	3 rue des tulipes	1 rue du glas
Cp	75000	33000
Ville	Paris	Bordeaux
Tel	+33601020304	+33799989796

Écrire un fichier SQL nommé « venteAdonf-insertUsers.sql » permettant d'insérer ces informations dans la base de données.

4 Jean Dupont a commandé 2 mugs et 3 tee-shirts (tailles S, M et L). Anne Aimaun a, quant à elle, commandé 1 mug et 1 tee-shirt (taille M).

(a) Écrire un fichier SQL nommé « venteAdonf-insertPanier.sql » permettant d'insérer ces informations dans la base de données.

(b) Écrire une requête permettant d'afficher le montant total du panier de Jean Dupont.

(c) Si la table `panier` n'avait pas de colonne « total », quelle requête permettrait de renvoyer le résultat de la question précédente ?

CORRIGÉS

1 QCM Vérification de connaissances

Enoncé p. 249

1. Réponse **d**. En effet, « SELECT * FROM clients » signifie que l'on veut voir tous les champs de la table « clients » et « ORDER BY ville DESC » signifie que l'on souhaite effectuer un tri sur le contenu de « ville » mais dans un ordre alphabétique inverse (DESC).
2. Réponse **a**. « clients INNER JOIN entrepots » désigne l'intersection des deux tables et « ON clients.ville = entrepots.ville » désigne le critère à prendre en compte pour trouver l'intersection (ici, le nom des villes). « SELECT * » signifie que l'on souhaite afficher tout donc la requête affiche tous les champs des deux tables où le nom des villes coïncide.
3. Réponse **c**. La requête `SELECT * FROM clients,entrepots WHERE clients.ville = entrepots.ville` est explicite : on sélectionne tous les champs des deux tables « clients » et « entrepots » où le contenu des champs « ville » est commun. Elle fait donc exactement la même chose que la requête de la question précédente.
4. Réponse **b**. La requête `SELECT * FROM clients LEFT OUTER JOIN entrepots ON (clients.ville = entrepots.ville)` prend en compte la table de gauche (LEFT), donc « clients », et affiche toutes ses entrées en commençant par celles où les noms de villes coïncident. Elle affiche ensuite les entrées où le contenu du champ « ville » ne coïncide avec aucun contenu du champ « ville » de la table « entrepots ».
5. Réponse **a**. La requête `SELECT SUM(commandes.total)` renvoie un tableau d'une ligne et d'une colonne dont la valeur est la somme des entrées de la colonne « total ». Il suffit ensuite de regarder la condition (ici, « WHERE date > '2019-09-02' » ainsi que l'alias (ici, « AS somme », qui signifie que le nom de la somme sera accessible via l'appellation de `somme`).

2 V/F Vérification de connaissances

Enoncé p. 251

1. *Faux*. Dans une base de données, une ligne d'une table est appelée un *enregistrement*.
2. *Vrai*. Les étiquettes des colonnes d'une table sont appelées les attributs de la table.
3. *Faux*. Une clé primaire a pour but de repérer de manière unique un enregistrement. On ne peut donc pas prendre un attribut (ici « nom ») qui peut prendre plusieurs fois la même valeur. Il serait ici préférable de prendre « idClient » comme clé primaire, si cet attribut a été créé comme étant un entier auto-incrémenté (par exemple).
4. *Vrai*. En effet, toute clé primaire d'une table, recopiée en tant qu'attribut d'une autre table, est appelé *clé étrangère* de cette dernière.

3 Festival de musique

Enoncé p. 251

1 La liste des titres des représentations est donnée par la requête :

```
SELECT
    titreRep
FROM
    Representation
```

2 La liste des titres des représentations ayant lieu sur le lieu nommé « Dionysos » est donnée par la requête :

```
SELECT
    titreRep
FROM
    Representation
WHERE
    lieu = "Dionysos"
```

3 La liste des noms des musiciens et les titres des représentations auxquelles ils participent est donnée par la requête :

```
SELECT
    M.nom , R.titreRep
FROM
    Musicien M
INNER JOIN
    Representation R
ON
    R.idRep = M.idRep
```

4 La liste des titres des représentations, les lieux et les tarifs d'une date précisée (par exemple le 24/12/2020) est donnée par la requête :

```
SELECT
    R.titreRep ,
    R.lieu ,
    P.tarif
FROM
    Programmer P
INNER JOIN
    Representation R
ON
    P.idRep = R.idRep
WHERE
    P.date = "2020-12-24"
```

5 Le nombre des musiciens qui participent à la représentation numéro 7 est donné par la requête :

```
SELECT
    COUNT (*)
FROM
    Musicien
WHERE
    idRep = 7
```

6 Les représentations et leurs dates dont le tarif ne dépasse pas 30 € sont données par la requête :

```
SELECT
    R.idRep ,
    R.titreRep ,
    P.Date
FROM
    Representation R
INNER JOIN
    Programmer P
ON
    R.idRep = P.idRep
WHERE
    P.tarif <= 30
```

4 Les employés du département

Enoncé p. 252

1 La liste des employés ayant une commission :

```
SELECT
    *
FROM
    Employes
WHERE
    COMM = 1
```

2 **MÉTHODE**

Pour obtenir une liste de résultats selon un ordre *croissant* ou *décroissant*, on utilise une requête avec le mot-clé `ORDER BY ... ASC` (pour un ordre croissant) et `ORDER BY ... DESC` (pour un ordre décroissant).

Les noms, emplois et salaires des employés par emploi croissant, et pour chaque emploi, par salaire décroissant :

```
SELECT
    ENOM ,
    PROF ,
    SAL
FROM
    Employes
ORDER BY
    PROF ASC ,
    SAL DESC
```

3 Le salaire moyen des employés :

```
SELECT
    AVG ( SAL )
FROM
    Employes
```

4 Le salaire moyen du département « Production » :

```
SELECT
    AVG ( E.SAL )
FROM
    Employes E
INNER JOIN
    Departements D
ON
    E.DNO = D.DNO
WHERE
    D.DNOM = "Production"
```

5 Obtenir la ligne d'un maximum

Enoncé p. 252

Au prime abord, on pourrait penser que la requête :

```
SELECT
    *, MAX(prix)
FROM
    commandes
```

suffirait.

Mais ce n'est pas le cas car elle affiche uniquement la première ligne de la table, en ajoutant une colonne où est inscrite la valeur maximale du prix, ce qui n'est pas ce que l'on souhaite.

On peut alors penser à la requête :

```
SELECT
    * , MAX(prix)
AS
    m
FROM
    commandes
WHERE
    prix = m
```

mais cela ne fonctionne pas : un message d'erreur apparaît car le `m` n'est qu'un alias et ne se substitue pas ici à la valeur du maximum.

L'idée consiste donc à insérer une requête après le WHERE dans la requête. La requête demandée est alors :

```
SELECT
    *
FROM
    commandes
WHERE
    prix = (
            SELECT
                MAX(total)
            FROM
                commandes
            )
```

Il est important que le nombre après « WHERE prix = » soit retourné par une requête SQL (ou soit spécifié concrètement, mais nous ne le connaissons pas dans ce cas de figure).

6 Notes annuelles des étudiants

Enoncé p. 252

1 Le nombre total d'étudiants est donné par la requête :

```
SELECT
    COUNT(*)
FROM
    ETUDIANT
```

2 Parmi l'ensemble des notes, la note la plus haute et la note la plus basse sont données par la requête :

```
SELECT
    MIN(Note) AS minimum,
    MAX(Note) AS maximum
FROM
    EVALUER
```

3 Les moyennes de chaque étudiant dans chacune des matières sont données par la requête :

```
SELECT
  ETU.NumEtudiant,
  ETU.Nom,
  ETU.Prenom,
  MAT.LibelleMat,
  MAT.CoeffMat,
  AVG(EVA.Note) AS MoyenneMat
FROM
  ETUDIANT AS ETU
  INNER JOIN EVALUER AS EVA
    ON EVA.NumEtudiant = ETU.NumEtudiant
  INNER JOIN MATIERE AS MAT
    ON MAT.CodeMat = EVA.CodeMat
GROUP BY
  ETU.NumEtudiant,
  ETU.Nom,
  ETU.Prenom,
  MAT.LibelleMat,
  MAT.CoeffMat
ORDER BY
  ETU.Nom
```

4 Les moyennes de la classe par matière peuvent être données par la requête :

```
SELECT
  MAT.CodeMat ,
  MAT.LibelleMat ,
  AVG(EVA.Note) AS MoyClasse
FROM
  MATIERE AS MAT
  INNER JOIN EVALUER AS EVA
    ON EVA.CodeMat=MAT.CodeMat
```

(suite du script page suivante)

```
GROUP BY
  MAT.CodeMat,
  MAT.LibelleMat
ORDER BY
  MAT.LibelleMat
```

5 Pour obtenir la moyenne générale de chaque étudiant, on va se servir d'une *vue*, celle obtenue avec la requête de la question 3 :

```
CREATE VIEW MoyGenEtu AS
SELECT
  ETU.NumEtudiant,
  ETU.Nom,
  ETU.Prenom,
  MAT.LibelleMat,
  MAT.CoeffMat,
  AVG(EVA.Note) AS MoyenneMat
FROM
  ETUDIANT AS ETU
  INNER JOIN EVALUER AS EVA
    ON EVA.NumEtudiant = ETU.NumEtudiant
  INNER JOIN MATIERE AS MAT
    ON MAT.CodeMat = EVA.CodeMat
GROUP BY
  ETU.NumEtudiant,
  ETU.Nom,
  ETU.Prenom,
  MAT.LibelleMat,
  MAT.CoeffMat
ORDER BY
  ETU.Nom
```

Une vue est une façon de construire une table qui peut servir dans une autre requête.

À l'aide de cette vue, nous pouvons maintenant afficher les moyennes générales par élève :

```
SELECT
    Nom , Prenom ,
    SUM(CoeffMat * MoyenneMat) / SUM(CoeffMat) AS
    moyenne
FROM MoyGenEtud
GROUP BY Nom , Prenom
```

6 Pour déterminer la moyenne générale de la promotion, on peut utiliser la vue basée sur la réponse à la question précédente :

```
CREATE VIEW SecondeVue AS
SELECT
    Nom ,
    Prenom ,
    SUM(CoeffMat * MoyenneMat) / SUM(CoeffMat) AS
    moyenne
FROM
    MoyGenEtud
GROUP BY
    Nom ,
    Prenom
```

Il ne reste plus qu'à écrire la requête suivante :

```
SELECT
    AVG ( moyenne )
FROM
    SecondeVue
```

7 Pour obtenir les étudiants qui ont une moyenne générale supérieure ou égale à la moyenne générale de la promotion, on peut utiliser une fois de plus la vue `SecondeVue` :

```
SELECT
  Nom , Prenom , moyenne
FROM
  SecondeVue
WHERE
  moyenne >= (
            SELECT
                AVG( moyenne )
            FROM
                SecondeVue
              )
```

7 Dans un lycée

Enoncé p. 253

1 **MÉTHODE**

Une jointure interne permet de lier plusieurs tables (ici, `eleves`, `controle` et `matiere`). On l'utilise pour ce genre de question.

```
SELECT
    E.idEleve,
    E.nom,
    E.prenom,
    COUNT(*) AS nbControle
FROM
    eleves E
    INNER JOIN controle C ON C.idEleve = E.idEleve
    INNER JOIN matiere M ON M.idMatiere = C.
    idMatiere
WHERE
    M.nomMatiere = 'NSI'
GROUP BY
    E.idEleve,
    E.nom,
    E.prenom
```

2 Une requête qui permet d'afficher la moyenne de chaque élève par matière (NomMat, Coef) peut être la suivante :

```
SELECT
    E.nom ,
    E.prenom ,
    M.nom ,
    M.coef ,
    AVG(C.note) AS moyenne
FROM
    controles C
INNER JOIN
    eleves E ON C.idEleve = E.idEleve
INNER JOIN
    matieres M ON C.idMatiere = M.idMatiere
GROUP BY
    E.nom ,
    E.prenom ,
    M.nom ,
    M.coef
```

3 **MÉTHODE**

Lorsqu'une requête semble être compliquée à construire, il est souvent utile de la décomposer en créant une ou plusieurs vues. C'est ici le cas : nous avons créé une première vue dans la question précédente, et nous allons nous en servir pour en créer une autre, qui nous servira dans la requête principale répondant à notre question.

On crée une vue « mavue » à partir de la requête précédente :

```
CREATE VIEW mavue AS
    SELECT
        E.nom ,
        E.prenom ,
        M.nom ,
        M.coef ,
        AVG(C.note) AS moyenne
    FROM
        controles C
    INNER JOIN
        eleves E ON C.idEleve = E.idEleve
    INNER JOIN
        matieres M ON C.idMatiere = M.idMatiere
    GROUP BY
        E.nom ,
        E.prenom ,
        M.nom ,
        M.coef
```

Ensuite, on écrit une requête permettant de calculer comme demandé la moyenne générale de chaque élève :

```
SELECT
    nom ,
    prenom ,
    SUM(coef*moyenne)/(SELECT SUM(coef) FROM
    matieres) AS  MoyGen
FROM
    mavue
GROUP BY
    nom ,
    prenom
```

8 Un cas pratique : site web marchand

Enoncé p. 254

1 Un fichier SQL permettant de créer les quatre tables est le suivant :

```
CREATE TABLE IF NOT EXISTS users (
    idUser BIGINT PRIMARY KEY NOT NULL
    AUTO_INCREMENT ,
    nom VARCHAR (30) ,
    prenom VARCHAR (30) ,
```

(Suite page suivante)

```
    email VARCHAR (50) ,
    rue VARCHAR (100) ,
    cp VARCHAR (10) ,
    ville VARCHAR (50) ,
    tel VARCHAR (12) );

CREATE TABLE IF NOT EXISTS modeles (
    idModele BIGINT PRIMARY KEY NOT NULL
    AUTO_INCREMENT ,
    nomModele VARCHAR (20) );

CREATE TABLE IF NOT EXISTS articles (
    idArticle BIGINT PRIMARY KEY NOT NULL
    AUTO_INCREMENT ,
    nom VARCHAR (50) ,
    descriptif VARCHAR (255) ,
    idModele BIGINT ,
    prix DOUBLE ,
    FOREIGN KEY (idModele) REFERENCES modeles(
    idModele) );

CREATE TABLE IF NOT EXISTS panier (
    idPanier BIGINT PRIMARY KEY NOT NULL
    AUTO_INCREMENT ,
    idUser BIGINT,
    idArticle BIGINT ,
  idModele BIGINT ,
    quantite INT ,
    total DOUBLE ,
    FOREIGN KEY (idUser) REFERENCES users(idUser) ,
  FOREIGN KEY (idModele) REFERENCES modeles(idModele
    ) ,
    FOREIGN KEY (idArticle) REFERENCES articles(
    idArticle) )
```

La taille des colonnes est ici arbitraire : nous les avons choisi en essayant d'être cohérent.

Pour ce qui est de la colonne « tel », nous avons fait le choix de la déclarer comme chaîne de caractères (pour que le format des numéros soient de la forme `+336XXXXXXXX` ou `+337XXXXXXXX`).

Pour les clés primaires, nous avons fait le choix du format « BIGINT » car nous sommes optimistes... et souhaitons beaucoup d'entrées pour cette base de données, qui signifierait un grand succès du site.

À RETENIR

Concernant le format numérique, nous avons le choix entre :

Type de données	Plage	Stockage
BIGINT	-2^{63} à $2^{63}-1$	Huit octets
INT	-2^{31} à $2^{31}-1$	Quatre octets
SMALLINT	-2^{15} à $2^{15}-1$	Deux octets
TINYINT	0 à 255	Un octet

2 Pour cette question, il faut anticiper : il faut avant tout enregistrer les différents modèles cités (Unique, S, M, L, XL et XXL). Ce n'est qu'ensuite que l'on pourra enregistrer les articles. On a alors :

```
INSERT INTO
    modeles (nomModele)
VALUES
    ( 'Unique' ), ( 'S' ),
    ( 'M' ), ( 'L' ),
    ( 'XL' ), ( 'XXL' );

INSERT INTO
    articles (nom,descriptif,idModele,prix)
VALUES
    (   'Mug' ,
        'Magnifique mug personnalisable avec votre
    photo.' ,
        1 ,
        '7.99' ),
    (   'T-Shirt "I Kiffe Beef"' ,
        'T-shirt unique en son genre.' ,
        2 ,
        '24.99' ),
    (   'T-Shirt "I Kiffe Beef"' ,
        'T-shirt unique en son genre.' ,
        3 ,
        '24.99' ),
      (
        'T-Shirt "I Kiffe Beef"' ,
        'T-shirt unique en son genre.' ,
        4 ,
        '24.99'
    ),
```

(suite du fichier page suivante)

```
    (
        'T-Shirt "I Kiffe Beef"' ,
        'T-shirt unique en son genre.' ,
        '5 ,
        '24.99'
    ),
    (
        'T-Shirt "I Kiffe Beef"' ,
        'T-shirt unique en son genre.' ,
        6 ,
        '24.99'
    )
```

live 3 **Insertion des utilisateurs**

```
INSERT INTO
    users (nom,prenom,email,rue,cp,ville,tel)
VALUES
    (
        'Dupont' ,
        'Jean' ,
        'jdupont@free.fr' ,
        '3 rue des tulipes' ,
        '75000' ,
        'Paris' ,
        '+33601020304'
    ) ,
    (
        'Aimaun' ,
        'Anne' ,
        'amz@free.fr' ,
        '1 rue du glas' ,
        '33000' ,
        'Bordeaux' ,
        '+33799989796'
    )
```

live 4 **(a)** **Insertion des commandes dans le panier**

```
INSERT INTO
    panier
    (idUser,idArticle,idModele,quantite,total)
```

(suite page suivante)

```
VALUES
    ( '1' , '1' , '2' , '15.98' ) ,
    ( '1' , '2' , '1' , '24.99' ) ,
    ( '1' , '3' , '1' , '24.99' ) ,
    ( '1' , '4' , '1' , '24.99' ) ,
    ( '2' , '1' , '1' , '24.99' ) ,
    ( '2' , '3' , '1' , '24.99' )
```

(b) La requête permettant d'afficher le total du panier de Jean Dupont est la suivante :

```
SELECT
    SUM(total)
FROM
    panier
WHERE
    idUser = 1
```

(c) Pour cette question, nous allons avant tout créer une vue, basée notamment sur la table « panier », donnant une table avec deux attributs : le « idUser » d'un enregistrement de la table « panier » ainsi que le total de la commande correspondant à l'enregistrement. Pour cela, nous allons prendre la jointure interne de la table « panier » avec la table « articles » sur l'attribut « idArticle », en regroupant selon l'attribut « idUser ».

```
CREATE VIEW V_TOTAL_PANIER AS
    SELECT
        PAN.idUser ,
        SUM(PAN.quantite*ART.prix) AS TOTAL
    FROM
        panier AS PAN
        INNER JOIN
            articles AS ART
        ON ART.idArticle = PAN.idArticle
    GROUP BY
        PAN.idUser;

SELECT TOTAL
FROM V_TOTAL_PANIER
WHERE idUser = 1
```

Une fois la vue créée, il suffit de sélectionner l'attribut « TOTAL » qui correspond à notre client, à savoir celui pour lequel idUser = 1.

Autre solution (sans « GROUP BY ») :

```
CREATE VIEW V AS
    SELECT * FROM panier WHERE idUsers = 1;

SELECT SUM( quantite * (
    SELECT prix
    FROM articles A
    WHERE A.idArticle = V.idArticle )
    ) AS TOTAL
FROM V
```

Chapitre 8

Architecture matérielle et réseaux

Plan du chapitre

1 Architecture matérielle

1.1 Les composants du numérique

Un téléphone portable, une télévision connectée et de nombreux appareils de notre quotidien sont en réalité de petits ordinateurs. Ils sont dotés d'une architecture similaire à celle d'un PC portable ou d'un PC fixe.

1.1.1 - Composants de base indispensables

- *Processeur*.
  On l'appelle aussi CPU (Central Processing Unit). Il est responsable de l'exécution des instructions de tous les microprogrammes.

  → Il fonctionne de manière cadencée, c'est-à-dire qu'il existe une fréquence à laquelle il exécute les instructions. À chaque période, une nouvelle instruction lui est livrée et il procède au calcul attenant. On appelle cette période un *cycle d'exécution*.

  → Il peut être composé de plusieurs cœurs. Chaque cœur est une unité d'exécution qui travaille en parallèle des autres, à la fréquence d'exécution globale. Un processeur 4 cœurs, par exemple, exécutera 4 instructions en parallèle à chaque période.
  L'augmentation du nombre de cœurs permet donc l'exécution d'un plus grand nombre d'opérations par seconde sans avoir besoin d'augmenter la fréquence globale d'exécution. C'est un point très intéressant car l'augmentation de la fréquence provoque celui de la consommation électrique et de la dissipation thermique (le processeur chauffe d'avantage).

COURS

- *Mémoire de travail (RAM).*
  La RAM (*Random Access Memory*) est une zone de stockage des données volatiles. Les programmes en cours d'exécution ainsi que les données qu'ils manipulent ou qui sont nécessaires à leur fonctionnement y sont chargés. La RAM est toujours vide au démarrage de la machine. C'est d'abord le système d'exploitation qui y est chargé. Il charge ensuite les programmes à exécuter, qui y chargent, à leur tour, les données dont ils ont besoin. Quand un programme termine son exécution, il est déchargé de la mémoire ainsi que tous les éléments qu'il y a lui-même inséré.
- *Contrôleurs et interfaces de communication (E/S ou I/O).*
  Pour fonctionner, les programmes ont besoin de communiquer avec des périphériques comme une souris, un clavier, un disque dur, des réseaux ethernet ou wifi, des capteurs (boutons marche/arrêt, capteur de température,...), le Bluetooth, l'écran, etc.
  Chaque périphérique doit être régi par un contrôleur adapté. Il s'agit d'un ensemble de composants électroniques (dont un sous-processeur), dédiés à chaque type de périphérique et plus ou moins complexes suivant le cas. Un réseau ethernet, par exemple, sera géré par un contrôleur qui se chargera d'encoder/-décoder les données envoyées/reçues vers/depuis les signaux électriques qui transitent dans le câble ethernet.

  Le processeur peut donc communiquer avec chacun des contrôleurs et recevoir des données de leur part (même quand il ne s'y attend pas expressément). Ces communications s'appellent des E/S (« Entrées/Sorties ») ou I/O (« Input/Output »), souvent notées simplement IO ou IOs (à ne pas confondre avec iOS, le célèbre système d'exploitation d'Apple !).
- *Bus de données.*
  Il s'agit des interconnexions entre les différents composants cités ci-dessus. Les données envoyées par le processeur vers la mémoire et les contrôleurs transitent à une fréquence fixe et suivant des protocoles de communication (règles très strictes) qui ne seront pas détaillés ici. Il faut imaginer qu'il existe un véritable réseau d'interconnexions des composants entre eux (un peu comme des routes qui relient des quartiers, des villes, des pays,...). Les communications sont alors comparables aux véhicules qui transitent sur ces routes et le protocole de communication s'apparente au code de la route. Il organise la circulation, définit les vitesses règlementées, évite les collisions, etc.

  Ces « routes » et les règles qui régissent la communication s'appellent un bus de données. Un des plus connus est le bus USB, présent sur tous nos ordinateurs et smartphones. Il est décliné en plusieurs versions (comme USB2, USB3, USB-C) suivant le protocole de communication choisi.
- *Carte mère.*
  Tous les éléments ci-dessus ont besoin d'être interconnectés de manière fiable. C'est le rôle de la carte mère. Circuit imprimé multicouche, elle possède non seulement les pistes d'interconnexion mais aussi un grand nombre de

composants électroniques pour l'organisation des communications, l'adaptation des niveaux électriques, des impédances, la dissipation thermique, des sécurités en cas de surtensions ou de sous-tensions, les bus de communication,...

- *Mémoire de stockage (appelée aussi mémoire de masse).*
  C'est le disque dur, la clé USB ou encore la carte SD, supports de stockage permanent des programmes et des données. On y trouve l'ensemble des programmes susceptibles d'être exécutés et les données stockées (images, ressources, documents, fichiers de configuration,...)

1.1.2 - Composants complémentaires très fréquents

- *Contrôleur vidéo et accélérateur graphique.*
  Nommé généralement GPU (Graphics Processing Unit), il s'agit d'un processeur à part entière et de son microenvironnement minimal (mémoire de travail, bus de communication) qui composent le contrôleur graphique. Il est responsable :
  - → de la confection des images affichées à l'écran dans la résolution appropriée ;
  - → des calculs nécessaires aux représentations 2D et 3D ;
  - → de la traduction des images en signaux vidéos compatibles avec la connexion utilisée (VGA, SVGA, HDMI, RCA) ;
  - → de la gestion éventuelle de plusieurs écrans en parallèle.

  Il décharge le processeur (CPU) des tâches routinières et répétitives liées à l'affichage et la composition des images, car elles doivent être exécutées en permanence quel que soit le programme courant (navigation sur le web, vidéo en streaming, défilement des écrans,...).
  Le CPU peut ainsi consacrer ses cycles d'exécution au fonctionnement des programmes. Ainsi, la fluidité d'une vidéo, par exemple, est davantage liée aux performances de la puce graphique qu'à celles du CPU.
- *Contrôleur audio.*
  C'est l'équivalent du contrôleur vidéo pour la gestion du son. Il élabore les signaux analogiques qui seront transmis aux enceintes ou dispositifs de diffusion sonore (ampli, chaîne hifi, casque). Il intercepte les signaux analogiques provenant d'éventuels microphones, les numérise et transmet l'information au CPU.
- *Contrôleurs de communication.*
  Ils sont responsables de la gestion des communications à faible ou grande portée qui peuvent être réalisées directement depuis un équipement numérique. Les supports et protocoles de communication comme le réseau Ethernet, la fibre optique, le Wifi, le Bluetooth ou les communications GSM Data large bande nécessitent, à chaque fois, un contrôleur spécifique.

COURS

- *Autres contrôleurs diversifiés.*
  Il existe beaucoup d'autres types de contrôleurs, autant qu'il existe de types de périphériques et d'interconnexions. Notons par exemple :
  → le système d'alimentation autonome (batterie, chargeur) ;
  → les capteurs en tout genre (radio, GPS, proximité, accéléromètre, boussole, température,...).

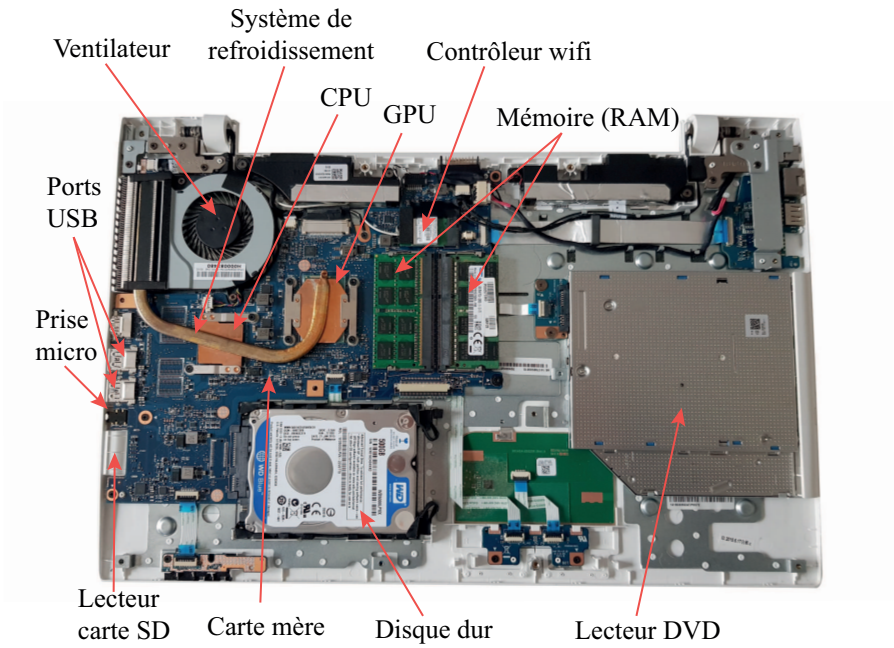


Figure 8.1 – Différents composants d'un ordinateur

1.2 Les SoCs (System on Chip)

Depuis les débuts de l'informatique, les composants électroniques n'ont cessé :

- de se sophistiquer (de plus en plus « intelligents »),
- de gagner en performances/rapidité (augmentation de la puissance de traitement),
- de se miniaturiser,
- de devenir accessibles au plus grand nombre (diminution des coûts de production).

La miniaturisation a atteint de telles proportions qu'il est aujourd'hui possible de réunir au sein d'une seule puce électronique tous les composants vitaux d'un ordinateur (CPU, RAM, mémoire de stockage, bus de communications, contrôleurs, GPU, interconnexions,...). Ces « ordinateurs-puce » s'appellent des SoCs.

Définition 1

Un *SoC* (System On Chip) est un système intégré sur une unique puce électronique qui regroupe les différents composants d'un ordinateur tels que le processeur, la mémoire, les interfaces et bus de communication, etc.

1.2.1 - Avantages

- Le principal avantage des SoCs est leur faible encombrement (juste une puce électronique), qui leur permet d'être implantés dans des appareils de taille réduite (téléphone, montre,...). L'avènement des smartphones est à l'origine de leurs succès.
- Un autre avantage est la faible consommation énergétique d'un SoC due à la miniaturisation des composants. Il y a moins de pistes que sur une carte mère, les distances sont raccourcies, il n'y a pas besoin de connecteur ni de câble et la tension d'alimentation peut être réduite. On obtient ainsi moins de dissipation thermique et de pertes par effet Joule.
- Enfin, les SoCs ont fait baisser les coûts de fabrication et d'industrialisation car ils sont produits en très grande quantité et ne nécessitent pas d'assemblage manuel des éléments qui les composent. Tous les composants électroniques sont gravés en une seule fois au sein d'une unique puce.

1.2.2 - Inconvénients

Les éléments qui composent un SoC étant gravés ensemble, il n'est pas possible de les faire évoluer individuellement. On ne pourra pas changer la « carte réseau » ni la « carte graphique » par exemple.

En conséquence, la durée de vie et la fiabilité d'un SoC sont celles de son maillon le plus faible. Le premier élément d'un SoC à présenter un dysfonctionnement provoquera une altération complète du SoC, qui ne pourra pas être réparé en changeant l'élément défaillant.

À RETENIR

- Un processeur multi-cœur exécute autant d'instructions par cycle qu'il a de cœurs.
- La consommation électrique et l'échauffement augmentent avec la fréquence d'exécution.
- Tous les programmes en cours d'exécution et les données qu'il manipule sont recopiés et « travaillés » dans la RAM.
- Il existe un grand nombre de contrôleurs spécifiques, dédiés aux fonctionnalités assumées par la machine et au pilotage des périphériques. Ils déchargent le processeur (CPU) des opérations routinières.
- Les différents composants d'une machine numérique communiquent via des bus de données.
- Un SoC (System on Chip) est une puce unique qui embarque tous les éléments de base d'un ordinateur.

COURS

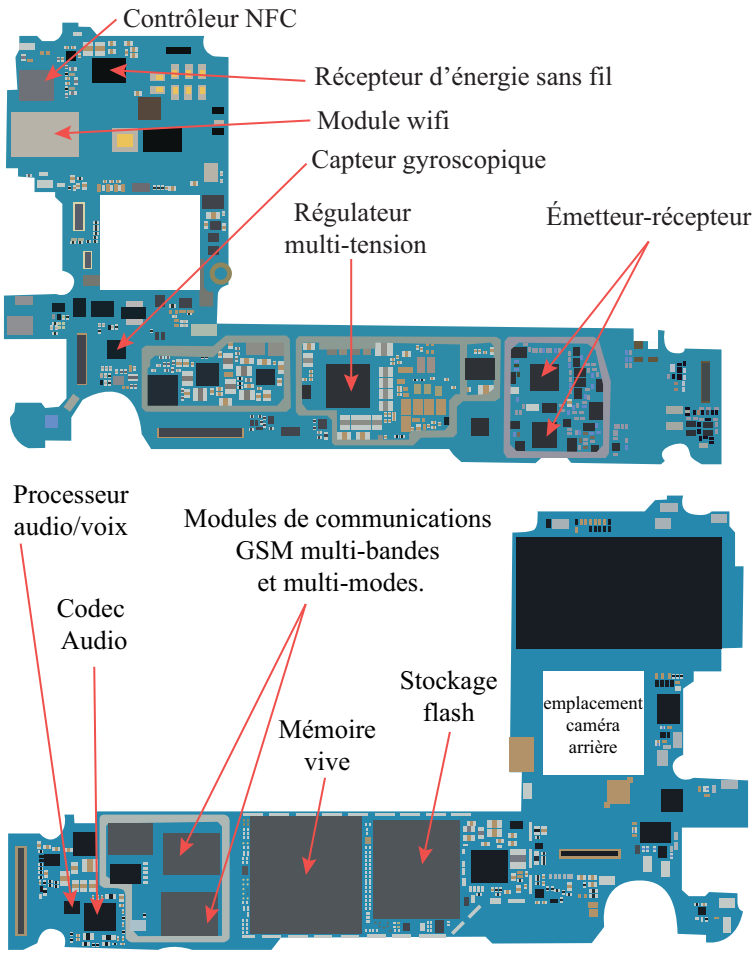


Figure 8.2 – Carte mère d'un smartphone Samsung S7 (recto/verso)

Voir exercices 1 page 300 et 3 à 6 page 301.

2 Processus et ressources partagées

PROBLÉMATIQUE

Comment un système d'exploitation multitâche fait-il pour exécuter de nombreux programmes simultanément alors que la machine ne possède qu'un seul et unique processeur avec un nombre de cœurs très limité ?

2.1 Partage du temps d'exécution

Dans un système d'exploitation multitâche comme Windows, Mac OS, iOS, Androïd ou encore Linux, on a l'impression que plusieurs programmes peuvent s'exécuter en même temps : notre navigateur télécharge un fichier pendant qu'un flux vidéo est en lecture sur VLC (logiciel de lecture vidéo), un traitement de texte est ouvert, une impression est lancée sur l'imprimante et une visio-conférence est en cours, tout ça sur un même PC, en même temps.

Mais cette simultanéité n'est qu'illusion. En réalité, le processeur ne peut pas exécuter plus d'instructions simultanées qu'il n'a de cœurs. Pourtant, notre machine n'a que 2 cœurs et 17 programmes s'exécutent apparemment en même temps, sans compter le système d'exploitation.

Pour réussir ce tour de force, l'OS va exécuter un peu de chaque programme tour à tour.

2.1.1 - Processus

Définition 2

On appelle *processus* une instance d'un programme, chargée en mémoire et qui est exécutée de manière continue ou discontinue par le système d'exploitation.

⚠ ATTENTION

Il y a une différence subtile entre un *programme* et un *processus*.
Un programme (par exemple un éditeur de texte), est un jeu d'instructions susceptibles d'être exécutées. Tant que ce n'est pas le cas, le programme est juste stocké sur la mémoire de masse. Ce n'est alors pas un processus, bien que ce soit un programme.

À son lancement, les instructions qui le composent sont chargées en mémoire de travail (RAM) et exécutées. Le programme a donné lieu à un processus constitué par la recopie de ses instructions en mémoire et l'exécution attenante. On dit que le système d'exploitation exécute une instance du programme, en mémoire, dans un processus dédié.

Supposons maintenant que l'on démarre de nouveau le même programme sans avoir fermé celui en cours d'exécution (par exemple, pour éditer deux documents texte différents). C'est toujours le même programme qui va être, de nouveau, recopié en mémoire, à un autre endroit, puis exécuté (lié au second document). On a alors, en mémoire, deux instances différentes du même programme qui s'exécutent simultanément, chacune dans un processus différent.

COURS

2.1.2 - Contexte d'exécution et switching

Un système d'exploitation multitâche exécute chaque processus dans un environnement mémoire cloisonné et protégé. Un processus ne peut lire et écrire des données en mémoire que dans la zone que le système d'exploitation lui a dédié. L'OS garantit qu'aucun autre processus en cours d'exécution ne pourra lire ni modifier ses données.

En classe de première, on a vu que le processeur possédait plusieurs registres (RI, AC, CC,...), un compteur ordinal et une UAL pour l'exécution des instructions. Lorsqu'un processus est exécuté, ce sont ses instructions qui occupent les registres et la « mémoire interne » du processeur.

Or l'OS doit exécuter les processus tour à tour pour donner l'illusion de leur simultanéité. Il doit donc très souvent interrompre l'exécution du processus en cours (A) pour permettre à un autre processus (B) de s'exécuter. Puis interrompre (B) pour exécuter un troisième processus (C), etc.

Mais lorsqu'il sera question de poursuivre l'exécution du processus (A), il faudra que celui-ci continue à l'endroit précis où il a été interrompu. Pour cela, tous les registres du processeur devront être pré-initialisés dans l'état précis où ils étaient au moment de l'interruption de (A). Pourtant, leur état a fortement été modifié par les exécutions successives de (B) puis de (C).

Lorsqu'il interrompt un processus, l'OS doit donc conserver scrupuleusement en mémoire l'état précis de tous les éléments volatiles au moment de cette interruption pour les restituer exactement dans le même état à la reprise du processus interrompu.

Définition 3 : contexte d'exécution

On appelle *contexte d'exécution* (execution context) l'ensemble de tous les éléments volatiles liés à l'exécution d'un processus. Ils devront être sauvegardés par le système à l'interruption du processus et restitués, en l'état, à sa réactivation.

Définition 4 : changement de contexte

On appelle *changement de contexte* (context switching) l'opération de remplacement d'un contexte d'exécution par un autre et de recopie en mémoire de chaque contexte écrasé.

À titre informatif, voici quelques uns des éléments que l'OS doit conserver à l'interruption d'un processus et restituer à sa réactivation. Ils constituent le *Process Control Block* (PCB) ou *bloc de données contextuelles* de chaque processus.

- Le numéro d'identification du processus (PID pour Process ID).
- L'état du processus, de son compteur ordinal et des autres registres.
- L'emplacement mémoire du code, des données et de la pile d'instructions.

- Les pointeurs vers les ressources utilisées, fichiers, E/S,...
- Le pointeur vers le processus parent, priorité, compteur de threads, durée d'exécution, informations d'attentes,... et bien d'autres informations.

2.2 Le Scheduler (ordonnanceur)

Définition 5 : scheduler

Le *scheduler* est le module du système d'exploitation qui se charge d'organiser l'exécution des processus à tour de rôle. Il a la délicate mission d'octroyer un peu de temps d'exécution processeur (quantum de temps) à tous les processus, en gérant les priorités, les débordements et les changements de contexte.

Définition 6 : quantum de temps

On appelle *quantum de temps* le plus petit temps d'exécution que l'OS peut attribuer à un processus. C'est une valeur qui est peu soumise à modification. Un processus peut se voir allouer 1 à *n* quantum(s) de temps successif(s). Typiquement, un quantum de temps dure 20 à 50 ms suivant les OS et les algorithmes d'ordonnancement utilisés.

2.2.1 - États d'un processus

2.2.1.1 - Définitions

Tous les processus qui cohabitent en mémoire à un instant donné ne sont pas forcément en attente du processeur. Il se peut qu'un processus attende une ressource, un résultat ou un autre processus et ne puisse rien faire tant que la ressource attendue n'est pas disponible.

Prenons, par exemple, le cas (très simplifié) d'un logiciel de lecture vidéo alimenté par un flux en streaming. Les données vidéo arrivent via le réseau. Le logiciel les décode et les transmet au contrôleur vidéo pour affichage. Supposons que le logiciel ait préparé 4 secondes de données vidéo transmises au contrôleur. Il ne peut rien faire tant que la suite des données n'arrive pas sur le réseau. Inutile, dans ces conditions, que le scheduler lui alloue un quantum de temps processeur car le processus n'en ferait rien.

Définition 7 : état « bloqué »

On dit qu'un processus est dans l'état « bloqué » lorsqu'il n'est plus éligible à l'obtention d'un quantum de temps processeur car en attente d'une ressource.

Dans notre exemple, le processus est bloqué jusqu'à ce que de nouvelles données vidéo soient disponibles sur le réseau. Lorsque ces données arrivent, le processus obtient de nouveau du « grain à moudre » mais il n'est pas immédiatement exécuté pour autant. Il repasse d'abord dans l'état « prêt à être exécuté » et c'est le scheduler qui lui attribuera un ou plusieurs quantums de temps.

COURS

Définition 8 : état « prêt »

On dit qu'un processus est dans l'état « prêt » lorsqu'il dispose de toutes les ressources dont il a besoin pour effectuer ses opérations. Il ne lui manque que l'obtention de quantum(s) de temps processeur pour exécuter/poursuivre son travail.

Voici tous les états que peut prendre un processus :

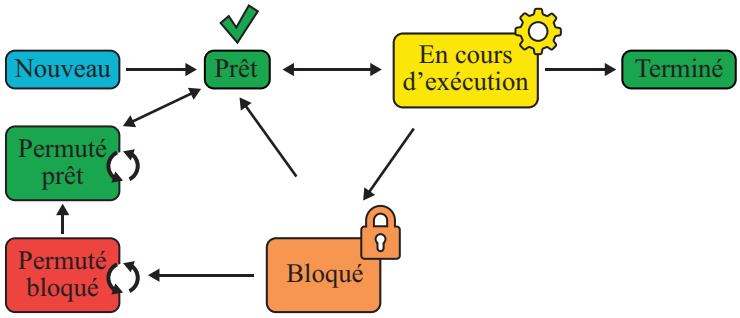


Figure 8.3 – Les différents états d'un processus dans un ordonnancement à court terme

2.2.1.2 - États principaux

- *Nouveau*.
  Un programme a été sélectionné pour démarrage. Ses instructions ont été recopiées en mémoire par l'OS et un nouveau processus y a été attaché mais jamais encore exécuté. Son contexte d'exécution et son PCB doivent être préparés et il doit être inséré dans la liste des processus « prêts ».
- *Prêt*.
  Le processus est en attente de quantum(s) de temps CPU pour exécuter son travail.
- *En cours d'exécution*.
  Un ou plusieurs quantum(s) de temps a (ont) été alloué(s) au processus. Il exécute ses instructions jusqu'à écoulement du temps imparti.
- *Bloqué*.
  Le processus est en attente d'une ou plusieurs ressource(s) (réseau, lecture disque, USB, Bluetooth, action utilisateur comme un clic ou un mot de passe,...) et ne peut rien faire en attendant. Il n'est pas éligible à l'obtention d'un quantum de temps CPU.
- *Terminé*.
  Le processus a achevé son exécution jusqu'à sa dernière instruction. Il doit être déchargé de la mémoire par l'OS. À cette occasion, la mémoire qu'il a utilisée va être libérée ainsi que toutes les ressources détenues (fichiers, connexion réseau,...).

2.2.1.3 - États secondaires

Pour augmenter virtuellement la mémoire dont dispose la machine, certains OS déchargent les processus inactifs de la mémoire (typiquement ceux bloqués pendant de nombreux quantums de temps). Ils stockent alors toutes les informations déchargées sur le disque (ou autre mémoire de masse), y compris le PCB du processus.

Le processus ne pourra alors pas être ré-exécuté lorsqu'il sera « débloqué ». Il devra préalablement être rechargé en mémoire. Ces opérations (typiquement à « long terme ») donnent lieu à deux nouveaux états :

- *Permuté bloqué*.
  Le processus est déchargé de la mémoire et il est, en plus, en attente d'une ressource.
- *Permuté prêt*.
  Le processus a été déchargé de la mémoire mais n'est plus en attente d'une ressource, il doit être rechargé en mémoire pour passer dans l'état « prêt » et être éligible à une exécution.

2.2.2 - Algorithmes de scheduling

Le scheduling (ordonnancement des processus) est un élément crucial d'un OS multitâche. Le partage du processeur doit être fait non seulement entre les processus démarrés par l'utilisateur mais aussi entre les différentes tâches du système d'exploitation, dont le scheduleur lui-même.

Il existe de nombreux algorithmes de scheduling différents qui ont toujours pour but de répartir les quantums de temps de la manière la plus optimisée entre tous les processus. Il serait trop complexe de détailler ici le fonctionnement de chacun d'eux mais voici un aperçu des principales méthodes de scheduling.

- *First-come, first-served (FCFS) : premier arrivé / premier servi*.
  Les processus sont stockés dans une file. Le premier arrivé est admis immédiatement et s'exécute tant qu'il n'est pas bloqué ou terminé. Lorsqu'il se bloque, le processus suivant commence à s'exécuter et le processus bloqué va se mettre au bout de la file d'attente.

  → Avantages : l'algorithme est simple (c'est une simple liste chaînée), l'ordonnancement est équitable.

  → Inconvénient : le processus qui utilise davantage de temps est favorisé par rapport à ceux qui font beaucoup d'appels aux entrées/sorties.
- *Shortest Job First (SJF) : le job le plus court d'abord*.
  Le processus dont le temps de traitement sera supposé le plus court est prioritaire. Si quatre tâches (jobs) ont des durées (en secondes) respectivement égales à a, b, c et d, la première se termine après a secondes, la deuxième après $a + b$ secondes, etc.

COURS

La durée de rotation moyenne est alors $\frac{4a+3b+2c+d}{4}$. Il est donc légitime de traiter en premier la tâche la plus courte car sa durée est multipliée par 4.

ANECDOTE

C'est typiquement l'algorithme qui est fréquemment observé aux caisses de supermarché quand des clients laissent passer ceux qui n'ont que peu d'articles.

Un inconvénient de cet algorithme est le suivant : si de multiples processus courts arrivent sans cesse, les plus longs ne sont jamais exécutés.

- *Shortest Remaining Time (SRT) : l'algorithme du temps restant le plus court.*
  C'est une version plus « agressive » du précédent. On tient compte ici du temps restant, et non du temps total, du processus.
  Imaginez un client n'ayant qu'un article qui arrive à la caisse d'un supermarché : il vient carrément interrompre le client face à la caissière pour faire passer son article s'il reste à ce dernier plus d'un article à faire passer.
- *Round Robin (RR) : l'algorithme du tourniquet.*
  Chaque processus se voit alloué un certain temps, appelé *quantum*, et à tour de rôle, chacun d'eux est traité par le processeur.
  En général, dans les OS multitâches, c'est cet algorithme qui est utilisé, avec un quantum entre 20 ms et 50 ms.
- *L'ordonnancement avec priorité.*
  Une valeur de priorité est assignée à chaque processus, la priorité pouvant varier dynamiquement.
  Par exemple, pour ne pas encombrer la mémoire avec des processus qui passent beaucoup de leur temps à attendre des E/S, on leur accorde une priorité d'autant plus grande qu'ils ne consomment qu'une petite fraction de leur quantum.
  Supposons que la valeur du quantum est fixée à 50 ms. Un processus qui n'utilise que 1 ms avant d'être bloqué aurait droit à une priorité de $\frac{50\text{ ms}}{1\text{ ms}} = 50$ tandis qu'un processus qui se bloque au bout de 25 ms a droit à une priorité de $\frac{50}{25} = 2$ et les processus qui consomment tout le quantum ont une priorité de 1.

2.3 Deadlock (interblocage)

2.3.1 - Principe

Il est possible qu'un processus (A) soit en attente d'un travail que doit fournir un autre processus (B). Dans ce cas, A est bloqué jusqu'à ce que B ait fourni le travail attendu.

Supposons maintenant que pour effectuer ce « travail attendu », le processus B ait besoin, à son tour, d'un travail effectué par le processus A.

Dans ce cas, B est bloqué jusqu'à ce que A ait fourni son travail. Sauf que A est déjà bloqué et ne peut être débloqué tant que B ne termine pas sa mission.

Cette situation provoque un blocage réciproque de A et de B qui s'attendent mutuellement. On appelle cette situation un interblocage (deadlock en anglais).

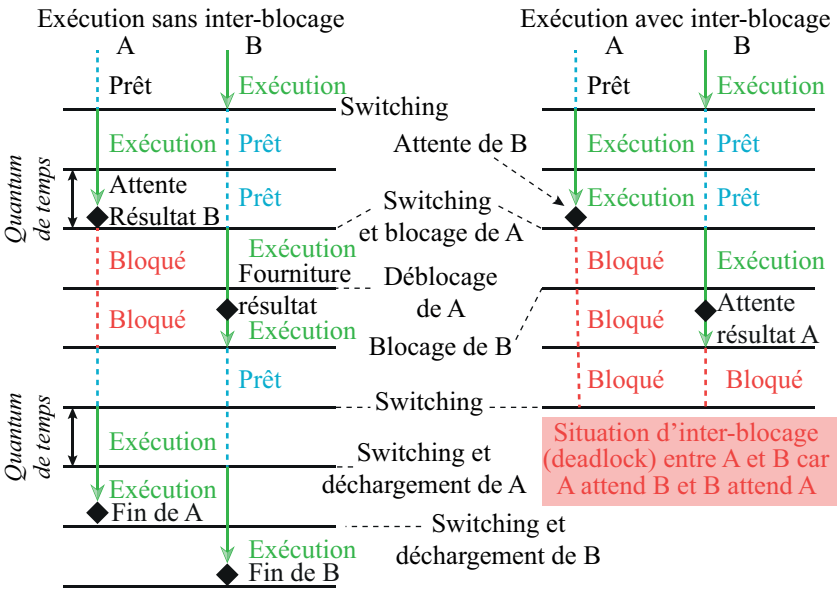


Figure 8.4 – Exécution avec et sans interblocage

2.3.2 - Verrouillage d'une ressource

Un processus peut également requérir l'accès exclusif à une ressource (fichier, port USB, zone mémoire,...). Une fois que la ressource est fournie au processus, par l'OS, de manière exclusive, elle ne peut pas redevenir disponible jusqu'à ce que le processus la libère à l'issue de son travail.

Prenons le cas d'un processus qui copie un fichier source S vers un fichier de destination D.

- Il peut partager le fichier source S avec d'autres processus à condition que ceux-ci ne modifient pas le contenu du fichier S. Les processus ne pourront pas obtenir d'accès en écriture sur le fichier S jusqu'à ce que la copie soit terminée.
- Il doit obtenir un accès exclusif en écriture sur le fichier D. Aucun autre processus ne doit écrire dans le fichier D jusqu'à ce que la copie soit terminée.
- Il est responsable de la libération des fichiers S et D, c'est-à-dire d'informer l'OS qu'il n'a plus besoin d'utiliser ces fichiers lorsqu'il aura terminé sa copie, sans quoi les fichiers restent verrouillés.

COURS

Définition 9 : ressource verrouillée en processus

On dit qu'une ressource est *verrouillée par un processus* (locked en anglais) tant que ce processus est le seul à pouvoir l'utiliser. Tout autre processus qui tente de la verrouiller passe dans l'état « bloqué » jusqu'à libération, par son détenteur, de la ressource convoitée.

Lorsque l'exécution d'un processus est interrompue par le scheduler, il reste en possession de ses verrous. Les ressources sur lesquelles il a obtenu un verrou exclusif ne pourront pas être utilisées par les autres processus exécutés même si le détenteur des verrous est arrêté (« prêt » ou « bloqué »).

Ces situations conduisent souvent à des interblocages sur ressource.

Soient par exemple deux processus A et B exécutés sur un OS multitâche :

- A requiert un verrou exclusif sur une ressource R_1 et l'obtient, puis il est interrompu par l'OS ;
- B requiert un verrou exclusif sur une ressource R_2 et l'obtient ;
- B requiert alors un autre verrou exclusif, sur la ressource R_1. Celle-ci étant détenue par A, B passe dans l'état bloqué (et y restera jusqu'à libération de R_1 par A) ;
- lorsque A poursuit son exécution, il est toujours en possession de R_1. Il requiert à présent un verrou sur R_2. Celle-ci étant détenue par B, A passe dans l'état bloqué (et y restera jusqu'à libération de R_2 par B).

Dans cette situation, A et B sont tous deux inter-bloqués. A est le seul à pouvoir libérer R_1 qui débloquera B, mais B est le seul à pouvoir libérer R_2 qui débloquera A.

Il existe de nombreuses méthodes pour limiter ou empêcher ces interblocages. Les principales sont présentées ci-dessous à titre informatif :

- règles strictes imposant l'ordre d'acquisition des ressources ;
- analyse détaillée des algorithmes lors de la conception des processus pour éliminer les interblocages ;
- système préventif qui détecte un risque d'interblocage avant que celui-ci ne se produise durant l'exécution ;
- système de récupération si un interblocage se produit (le système doit pouvoir repartir dans un état valide).

Exemple (algorithmes « Attente/Mort » et « Blessé/Attente ») : dans ces deux algorithmes, on prend en compte l'âge des processus. On distingue alors le plus âgé (noté *A*) du plus jeune (noté *J*).

- Dans l'algorithme « Attente/Mort » :

  → si *A* requiert une ressource verrouillée par *J*, on attend la libération de cette ressource ;

→ si J requiert une ressource verrouillée par A alors J meurt (il est définitivement arrêté par l'OS avec libération de toutes ses ressources et retour au début de son exécution).

- Dans l'algorithme « Blessé/Attente »,

  → si J requiert une ressource détenue par A, on attend la libération de cette ressource ;

  → si A requiert une ressource détenue par J alors A provoque la mort de J, ce qui libère toutes les ressources en sa possession et J reprend son exécution depuis le début.

Retrouvez une vidéo de présentation des processus avec Nathan Live.

Voir exercices 2 page 300 et 7 page 301.

3 Réseau et routage des données

PROBLÉMATIQUE

Le réseau Internet est étendu à l'échelle planétaire, reliant des milliards de machines les unes aux autres : ordinateurs, routeurs, mais aussi smartphones, tablettes et un nombre en pleine croissance d'objets connectés tels que véhicules, assistants personnels, montres, et bientôt la plupart des équipements de maison.
Comment ces communications voyagent-elles d'un bout à l'autre du globe, parcourant en quelques dizaines de millisecondes des milliers de kilomètres ? Comment les routeurs, aiguilleurs du numérique, font-ils pour organiser tout ce trafic, choisir les routes optimales et garantir l'acheminement des informations de bout en bout, sans embouteillage, collision ni perte de donnée ?

3.1 Le protocole IP

En classe de première, vous avez vu qu'il existait plusieurs architectures réseau, dont la plus répandue est TCP/IP, composée des protocoles TCP (couche 4 du modèle OSI) et IP (couche 3). Dans la suite, nous nous limiterons à cette architecture et nous intéresserons uniquement à des protocoles de routage IP.

Inventé en 1974 par les Américains Vint Cerf et Bob Kahn, les règles du routage IP sont spécifiées dans le document RFC 791 de l'IETF (Internet Engineering Task Force). Ce document laisse le champ libre à différentes stratégies de routage. Nous verrons dans la suite qu'il existe plusieurs types de protocoles IP.

Retrouvez le détail de la RFC 791 définissant le protocole IP avec Nathan Live.

COURS

3.2 Adressage

En informatique, communiquer consiste à envoyer et recevoir des lots de données nommés *paquets*. Chaque paquet est comparable à une lettre dans le système postal. Il faut avant tout disposer d'une adresse pour pouvoir recevoir du courrier. On parle ici d'adresses IP (IPv4 ou IPv6) et d'adresses MAC.

Définition 10 : adresse MAC

L'*adresse MAC* (Media Access Control) d'un équipement est un numéro codé sur 6 octets, lié à une carte réseau (ou autre dispositif physique de communication). C'est l'adresse physique identifiant de manière unique l'équipement.

Définition 11 : adresse IP

Une *adresse IP* (Internet Protocol) est un numéro codé sur 4 octets (IPv4) ou sur 16 octets (IPv6) attribué à un ordinateur ou tout autre dispositif logique de communication.

L'adresse IP est *contextuelle* : elle dépend du réseau sur lequel se connecte la machine et de sa configuration. Elle peut varier dans le temps.

3.3 Topologie réseau

On représente généralement des machines interconnectées sur un réseau IP de la manière suivante :

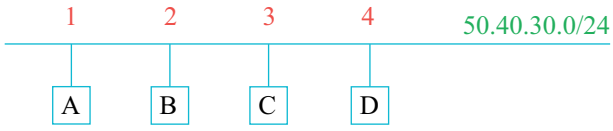


Figure 8.5 – Machines interconnectées sur un réseau IP

Sur cet exemple,

- A, B, C et D sont quatre machines reliées au même réseau IP. Elles ont pour adresses IP :

  A : 50.40.30.1 ; B : 50.40.30.2 ; C : 50.40.30.3 ; D 50.40.30.4

- l'adresse 50.40.30.0/24 n'est pas l'adresse d'une machine en particulier, mais celle du réseau IP lui-même. Elle signifie que toutes les machines raccordées à ce réseau auront une adresse IP de la forme 50.40.30.x avec x compris entre 1 et 254 inclus.

Une adresse IP contient à la fois une partie « réseau » (à gauche) et une partie « machine » (à droite).

Exemple : 50.40.30. 123 / 24
réseau machine

Les trois premiers octets correspondent à l'adresse du réseau et le dernier octet sert à différencier la machine sur ce réseau. Une adresse IPv4 est codée sur quatre octets soit 32 bits. Le complément « /24 », situé derrière l'adresse du réseau, indique le nombre de bits (à gauche de l'adresse) utilisés pour la partie « réseau ». Ici, ce sont les vingt-quatre premiers bits de l'adresse, soit les trois premiers octets. On appelle ce complément le *masque de réseau*.

Définition 12

On appelle *masque de réseau* un moyen d'identifier les bits d'une adresse IP correspondant à sa partie « réseau ».
Un masque peut être donné :

- sous la forme « /N », où N est le nombre de bits utilisés dans l'adresse réseau (comme « /24 ») ;
- sous la forme d'une suite de quatre décimaux, compris entre 0 et 255, représentant chacun un octet formé par les bits servant au réseau.

Exemple : le masque réseau de l'adresse IP 50.40.30.123 est « /24 », que l'on peut aussi écrire : 255.255.255.0.

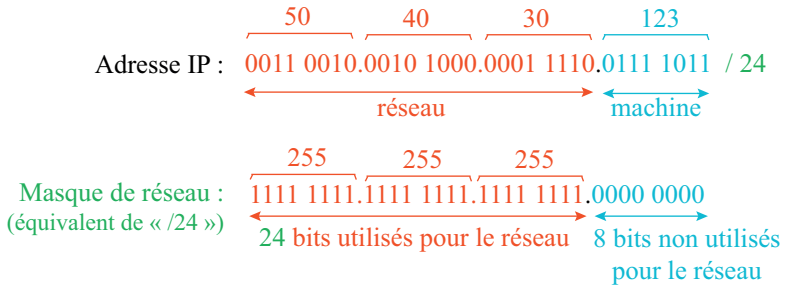


Figure 8.6 – Calcul du masque réseau

Définition 13

Deux machines (A) et (B) sont dites *sur le même réseau local* lorsque qu'elles ont le même masque de réseau et que leur adresse IP ont leur partie « réseau » strictement identiques.

Dans l'exemple de la figure 8.6 précédente, les machines A, B, C et D sont sur le même réseau local.

Les machines d'un même réseau local s'échangent leur adresse MAC par une mécanique qui n'est pas au programme de terminale, mais il faut admettre qu'une machine connait toutes les adresses MAC de ses voisines (sur le même réseau). En revanche, aucune machine ne peut connaitre l'adresse MAC d'une autre machine si elles sont situées sur des réseaux IP différents.

COURS

3.4 Routage IP

Retrouvez une présentation du routage IP avec Nathan Live.

Lorsqu'un ordinateur (A) veut communiquer avec un ordinateur (B), il a besoin :

- de connaitre l'adresse IP de (B) ;
- de disposer lui-même d'une adresse IP pour se « signaler » et recevoir la réponse ;
- de connaitre l'adresse MAC de la machine auprès de laquelle il va déposer son message (qui n'est pas forcément le destinataire final).

Deux cas de figure sont à différencier :

- (A) et (B) sont situées sur le même réseau local (par exemple dans une maison, connectées au même point d'accès Wifi, à la même box internet ou au même switch) et dans ce cas, (A) peut directement entreprendre de déposer son paquet de données sur le réseau (en utilisant l'adresses MAC de son destinataire final) ;
- (A) et (B) sont sur des réseaux IP disjoints. (A) ne connaît alors pas l'adresse MAC de (B) et ne peut pas l'atteindre directement. Dans ce cas, (A) aura besoin de connaitre un intermédiaire (routeur ou passerelle), sur son réseau local, qui se chargera de faire avancer le paquet de données pour qu'il rejoigne (B). Cet intermédiaire est un routeur (comme une box internet par exemple).

Définition 14 : routeur

Un *routeur* est un équipement informatique qui participe à l'acheminement de données entre des réseaux différents.

3.5 Tables de routage

Considérons l'architecture réseau suivante qui présente 4 réseaux locaux interconnectés par le biais de quatre routeurs (R1, R2, R3 et R4) dont un (R4) est connecté à l'Internet (R4 est supposé être une box ADSL ou fibre par exemple).

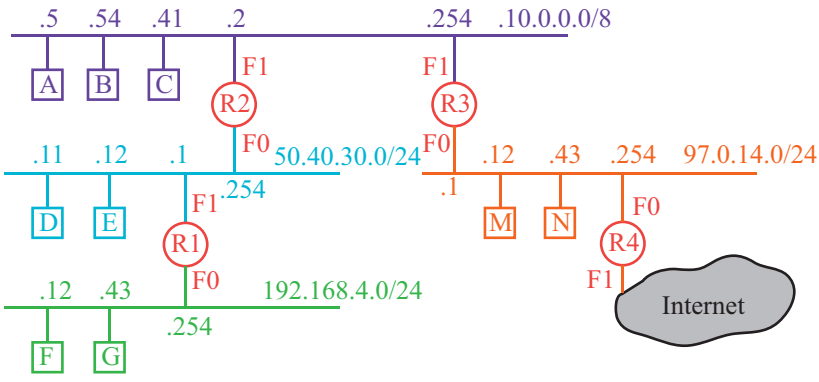


Figure 8.7 – Réseau à quatre routeurs

Une fois démarrés, les routeurs agissent, dans un premier temps, comme des PC pour constituer leur table de routage à partir des réseaux qui leurs sont directement reliés et qu'ils peuvent donc eux-mêmes détecter.

Voici les tables de routage initiales (automatiques) ainsi constituées :

Routeur	Réseau	Masque	Destination
R1	192.168.4.0	255.255.255.0	Interface locale F0
	50.40.30.0	255.255.255.0	Interface locale F1
R2	50.40.30.0	255.255.255.0	Interface locale F0
	10.0.0.0	255.0.0.0	Interface locale F1
R3	97.0.14.0	255.255.255.0	Interface locale F0
	10.0.0.0	255.0.0.0	Interface locale F1
R4	97.0.14.0	255.255.255.0	Interface locale F0

Ces informations ne suffisent pas à router l'ensemble du réseau ci-dessus.

Supposons que la machine (A) ait besoin de communiquer avec la machine (G) :

- (A) prépare un paquet de données TCP/IP vers l'IP 192.168.4.43 ;
- comme (G) n'est pas sur son réseau local, elle envoie ce paquet à (R2) déclaré comme étant sa passerelle par défaut ;
- (R2) ne trouve aucune route dans sa table de routage pour atteindre (G) car le réseau 192.168.4.0/24 ne fait pas partie de ceux qu'il connaît ; (A) et (G) ne peuvent alors pas communiquer entre eux.

Il est toujours possible de compléter manuellement les tables de routage pour indiquer à chaque routeur les réseaux qu'il ne peut pas découvrir seul (non directement reliés).

3.6 Le protocole RIP

Ici, RIP ne signifie pas « Rest In Peace », mais « Routing Information Protocol ». RIP fait partie de la famille des protocoles de routage IP. Il s'agit d'une stratégie de routage, utilisée uniquement par des équipement spécialisés comme les routeurs, et comportant deux règles principales :

1. Chaque routeur doit envoyer périodiquement sa propre table de routage aux routeurs présents sur chacun de ses réseaux locaux (R1 à R2, R2 à R1 et R3, R3 à R2 et R4, R4 à R3). Les routeurs utilisent les informations reçues de leurs voisins pour compléter leur table de routage interne qui sera de nouveau transmise (complétée) à l'itération suivante. L'envoi a lieu, typiquement toutes les 30 secondes (ce délai étant paramétrable).
2. Chaque routeur effectue un décompte du nombre des autres routeurs présents sur chaque route de sa table de routage. Ce décompte aboutit à un entier nommé métrique de la route. Plus la métrique d'une route est élevée, plus un paquet de données, envoyé sur cette route, devra traverser un grand nombre de routeurs (effectuer des sauts) avant de parvenir à son destinataire.

COURS

Voici le résultat de l'application du protocole RIP sur notre architecture (figure 8.7) à l'issue des échanges de tables. La topologie réseau n'étant pas modifiée sur le temps analysé, les tables sont uniquement complétées au fur et à mesure des échanges RIP. Il faut trois échanges successifs pour que chaque routeur ait la vision complète de l'architecture. La colonne « Itération » indique l'échange RIP à l'issue duquel la ligne de routage est insérée dans la table.

Routeur	Itération	Réseau	Masque	Destination	Métrique
R1	0	192.168.4.0	255.255.255.0	F0	0
		50.40.30.0	255.255.255.0	F1	0
	1	10.0.0.8	255.0.0.0	50.40.30.254	1
	2	97.0.14.0	255.255.255.0	50.40.30.254	2
R2	0	50.40.30.0	255.255.255.0	F0	0
		10.0.0.0	255.0.0.0	F1	0
	1	192.168.4.0	255.255.255.0	50.40.30.1	1
		97.0.14.0	255.255.255.0	10.0.0.254	1
R3	0	97.0.14.0	255.255.255.0	F0	0
		10.0.0.0	255.0.0.0	F1	0
	1	50.40.30.0.24	255.255.255.0	10.0.0.2	1
	2	192.168.4.0	255.255.255.0	10.0.0.2	2
R4	0	97.0.14.0	255.255.255.0	F0	0
	1	10.0.0.0	255.0.0.0	97.0.14.1	1
	2	50.40.30.0	255.255.255.0	97.0.14.1	2
	3	192.168.4.0	255.255.255.0	97.0.14.1	3

Remarques

- Ici, métrique et itération sont identiques à cause de la topologie (simple) du réseau mais ce n'est pas le cas pour des réseaux plus complexes.
- Le protocole RIP impose à chaque routeur de supprimer les routes pour lesquelles la métrique est supérieure à 15 afin de limiter l'allongement des tables de routage, notamment lorsque le réseau comporte des boucles ou un très grand nombre de routeurs.

⚠ ATTENTION

Lorsqu'un même réseau est accessible par deux routes différentes, le protocole RIP impose de ne conserver que celle dont la métrique est la plus basse. C'est celle qui est considérée comme étant la plus optimisée (limite du nombre de sauts) en cas de doublons.

3.7 Le Protocole OSPF

Tout comme RIP, OSPF (Open Shortest Path First) est un protocole de routage IP. Il impose également l'échange périodique d'informations entre routeurs voisins. La principale différence réside dans le remplacement de la métrique par une *connaissance partagée du débit* imputable à chaque réseau.

Considérons une topologie dans laquelle sont mis en œuvre des débits différents pour chaque réseau.

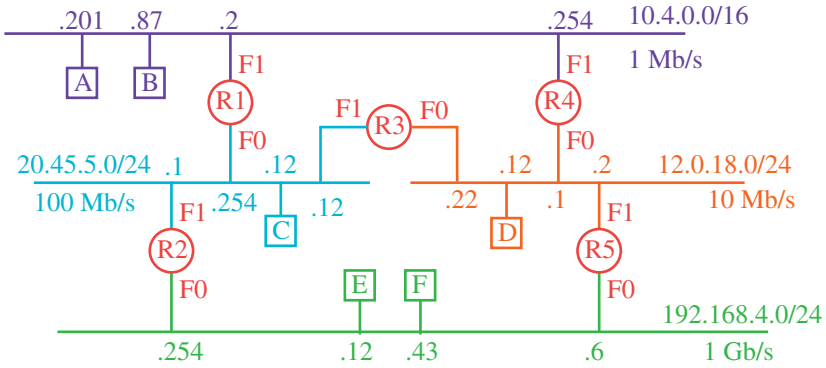


Figure 8.8 – Réseau dont les branches ont des débits différents

Le protocole OSPF permet à chaque routeur de cette topologie de se construire le graphe complet des liaisons inter-routeur pour optimiser le routage en fonction du débit des réseaux.

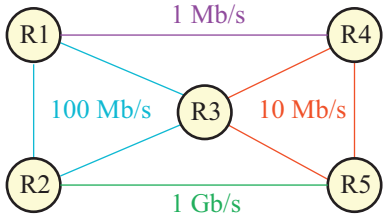


Figure 8.9 – Graphe correspondant au réseau représenté sur la figure 8.8

Supposons que R1 ait besoin de router un grand nombre de paquets TCP/IP vers R4 :

- s'il les lui transmet directement en utilisant son interface locale F1, la communication s'effectuera à 1 Mb/s ;
- si un autre chemin est utilisé (par exemple, « R1, R3, R4 », « R1, R2, R5, R4 » ou « R1, R2, R3, R4 »), l'envoi sera effectué à environ 10 Mb/s, qui est le facteur le plus limitant (réseau jaune) des différents itinéraires possibles.

Avec le protocole RIP, l'itinéraire choisi aurait été (R1, R4) puisqu'il est direct, donc associé à une métrique de 0 dans ce protocole. L'envoi aurait été 10 fois plus lent (à 1 Mb/s).

COURS

Le protocole OSPF n'utilise pas la métrique du RIP mais calcule un coût pour chaque liaison (et même pour chaque arc lorsque les débits sont dissymétriques) dont la valeur est un entier obtenu par arrondi :

$$\text{coût} = \frac{10^8}{\text{bande passante en bit/s}}.$$

ATTENTION

Le coût ne peut pas être nul ; aussi, quand le calcul donne un résultat plus proche de 0 que de 1, on considèrera qu'il vaut tout de même 1.

Chaque routeur de la topologie se représente donc en interne le graphe suivant :

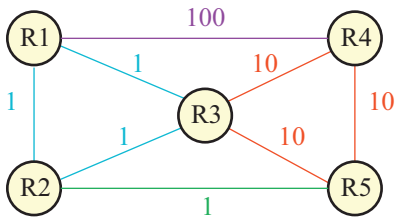


Figure 8.10 – Graphe représenté par chaque routeur

Le protocole OSPF impose à un routeur d'utiliser le chemin dont le coût total est le plus faible pour router les paquets vers son destinataire.

En pratique, l'équipement utilise généralement l'algorithme de Dijkstra pour déterminer ce chemin optimal et en déduire le routeur voisin auquel il doit transmettre un paquet de donnée à router.

OSPF est donc bien plus performant que RIP mais plus complexe d'utilisation et typiquement destiné aux très gros réseaux. Voici quelques-uns des avantages de ce protocole :

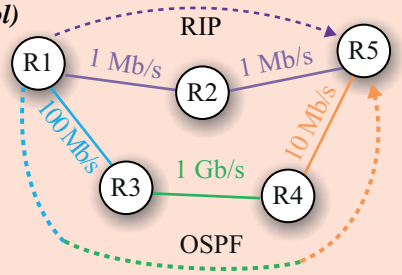
- il n'a pas de limite du nombre de sauts comme le protocole RIP ;
- il permet à chaque routeur une connaissance complète des réseaux au sein d'une zone ;
- il élimine le danger des boucles de routage ;
- les mises à jour sont envoyées uniquement lors d'un changement de topologie, ce qui permet une économie de bande passante ;
- il permet la répartition de charge car le choix du meilleur chemin est basé sur le coût (la bande passante inversée) qui évolue au cours du temps.

D'autres avantages existent mais leur complexité technique les exclut du cadre de ce cours.

À RETENIR

RIP (Routing Information Protocol)
Plus court chemin déterminé
par le nombre de sauts
R1 → R2 → R5

OSPF (Open Short Path First)
Plus court chemin déterminé
par le coût des routes
R1 → R3 → R4 → R5



4 Sécurité des échanges de données

PROBLÉMATIQUE

De plus en plus de données circulent sur les réseaux. Aussi bien locaux (comme wifi ou ethernet) que distants. La sécurité des échanges d'informations constitue un enjeu essentiel dans le développement des réseaux et des applications connectées. Quelles sont les techniques de sécurité pour garantir qu'aucun utilisateur ne peut espionner les échanges entre deux interlocuteurs ?

4.1 Principes de base du chiffrement

Dans un monde parfait où un intrus ne pourrait pas observer les communications entre deux interlocuteurs, le chiffrement serait inutile. Celui-ci n'a pas vocation à empêcher un espion d'observer les échanges d'informations mais à rendre les données transférées incompréhensibles à tout autre individu que leur destinataire légitime.

Définitions 15

On appelle *algorithme de chiffrement* une suite d'opérations destinées à chiffrer un message.
On appelle *algorithme de déchiffrement* la suite d'opérations nécessaires pour déchiffrer le message crypté.
Il arrive que ces deux algorithmes soient identiques.

4.1.1 - Chiffrement à algorithme secret : exemple ROT13 (ROTate by 13 places)

L'algorithme de chiffrement ROT13 consiste à décaler chaque lettre d'un message textuel de 13 positions dans l'alphabet, considéré circulaire (la lettre Z est suivie par A, puis B,...).

COURS

Considérons deux interlocuteurs Alice (A), Bernard (B) et un espion Ève (E) qui peut lire les transmissions entre (A) et (B). Alice souhaite transmettre un message confidentiel (M) à Bernard. Elle utilise l'algorithme ROT13 pour chiffrer son message avant envoi et ne transmet à Bernard que la version cryptée (C) du message :

- Alice encode son message :
  M = « danger imminent » devient C = « qnatre vzzvarag » ;
- Elle envoie (C) à Bernard ;
- Ève observe la communication et voit passer (C) mais ne sait pas comment le déchiffrer ;
- Bernard reçoit (C) et utilise l'algorithme de déchiffrement ROT13 pour reconstruire (M) et le lire.

⚠ ATTENTION

- La sécurité repose ici sur le secret de l'algorithme utilisé. Ève ne peut pas déchiffrer le message car elle ignore qu'Alice et Bernard utilisent l'algorithme ROT13.
- Préalablement à cet échange, Alice et Bernard doivent avoir convenu d'utiliser l'algorithme ROT13 pour communiquer entre eux et Ève ne doit pas avoir eu écho de cette information.

Ce chiffrement extrêmement simpliste est quelquefois utilisé en informatique pour empêcher la lecture directe d'un texte mais ne constitue pas un moyen de sécurité fiable du fait de sa simplicité.

Le décalage de 13 positions dans l'alphabet qui comporte 26 ($= 2 \times 13$) lettres permet d'utiliser le même algorithme pour le chiffrage et le déchiffrage du message car décaler un caractère vers la droite de 13 lettres deux fois consécutives redonne le caractère de départ.

4.1.2 - Chiffrement à clé secrète

Pour sécuriser davantage leurs échanges, Alice et Bernard peuvent convenir d'utiliser une table aléatoire de traduction secrète de l'alphabet, qu'ils seront les seuls à détenir, comme dans l'exemple page ci-contre.

Lettre en clair :	A	B	C	D	E	F	G	H	I	J	K	L	M
Lettre chiffrée :	S	E	T	R	O	G	Q	B	P	I	F	L	W

Lettre en clair :	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Lettre chiffrée :	H	X	C	Y	Z	V	U	J	A	K	N	D	M

La chronologie de l'échange est identique à la précédente :

- Alice encode son message :
  M = « danger imminent » en C = « rshqoz pwwphohu » ;

- Elle envoie (C) à Bernard ;
- Ève observe la communication et voit passer (C) mais ne peut pas le déchiffrer ;
- Bernard reçoit (C) et le déchiffre avec la table de correspondances.

Dans ce cas, la sécurité ne repose plus sur l'algorithme de chiffrement utilisé mais sur la table de correspondances secrète. Alice et Bernard peuvent déclarer ouvertement qu'ils communiquent à l'aide d'un algorithme de substitution et d'une table de correspondances (sans donner cette table évidemment). Ève pourra bâtir un algorithme de substitution mais il sera inefficace tant qu'elle n'utilisera pas la bonne table de correspondances.

Définition 16 : clé de chiffrement

On appelle *clé de chiffrement* un paramètre utilisé dans un algorithme pour effectuer le travail de chiffrement/déchiffrement et qui conditionne le résultat obtenu.

Ici, la clé de chiffrement est la suite des 26 lettres de l'alphabet en version chiffrée :

Clé = SETROGQBPIFLWHXCYZVUJAKNDM

Tout espion disposant de cette clé est en mesure de déchiffrer la communication entre Alice et Bernard ainsi que de chiffrer des messages en se faisant passer pour Alice ou Bernard.

⚠ ATTENTION

- La sécurité repose ici sur le secret de la clé utilisée. Ève ne peut pas déchiffrer le message car elle ignore cette clé, bien qu'elle sache qu'Alice et Bernard utilisent un algorithme de substitution.
- Préalablement à cet échange, Alice et Bernard doivent avoir convenus d'une clé secrète et Ève ne doit pas avoir écho de cette information.

4.1.3 - Attaque et déchiffrement par un espion

Bien que ne possédant pas la clé de chiffrement, Ève peut chercher à décoder le message chiffré.

- La technique la plus rudimentaire consiste à essayer toutes les clés possibles. Sachant qu'Alice et Bernard utilisent un algorithme de substitution, Ève comprend que la clé est une anagramme des 26 lettres de l'alphabet.
  Ses souvenirs de mathématiques lui permettent de dénombrer les clés possibles : la factorielle du nombre de lettres. Or la factorielle de 26 (notée 26!) représente environ 4×10^{26} combinaisons.
  En supposant qu'il faut 1 ms à un ordinateur pour tester une des clés possibles, le déchiffrement ne sera garanti qu'après : $4 \times 10^{26} \times 1 \text{ ms} = 4 \times 10^{23}$ secondes soit environ $1{,}3 \times 10^{16}$ années, c'est-à-dire 13 millions de milliard d'années. De toute évidence, ce n'est pas du tout la bonne technique.

COURS

- Ève utilisera plutôt une autre technique comme l'analyse fréquentielle qui consiste à repérer les répétitions des caractères et à examiner la fréquence d'apparition des lettres chiffrées.
  Dans chaque langue écrite, il existe une fréquence moyenne à laquelle une lettre est susceptible d'apparaître, comme le « e » qui, en français, est la lettre la plus utilisée. On repère ainsi quelle est la lettre cryptée représentatives du « e » et on fait de même avec les autres lettres.
  La performance de cette technique augmente avec le nombre et la taille des messages cryptés analysés. Plus l'échantillon est important, plus l'analyse des répétitions donnera des résultats probants.

Concrètement, l'algorithme de chiffrement par substitution n'est plus utilisé aujourd'hui car les techniques d'analyses fréquentielles sont très affinées et permettent de retrouver rapidement la clé.

4.2 Familles de chiffrements par clés

Il existe beaucoup d'algorithmes de chiffrement et de nouveaux algorithmes sont régulièrement inventés. Les plus pertinents sont ceux basés sur le secret de la clé utilisée (algorithme à clé secrète). Ils se répartissent en deux grandes familles.

4.2.1 - Le chiffrement symétrique

Définition 17

Un chiffrement est dit *symétrique* ou « à clé symétrique » lorsque la clé utilisée pour chiffrer le message sert également à le déchiffrer.

Attention à ne pas confondre *clé symétrique* et *algorithme symétrique*. Il n'est pas nécessaire, ici, d'utiliser le même algorithme pour chiffrer et déchiffrer le message (comme dans le cas du ROT13).

Les algorithmes de chiffrement et déchiffrement peuvent être des programmes différents mais ils nécessitent la même clé secrète. Les algorithmes de substitution, comme celui de l'exemple en amont, sont des algorithmes à clé symétrique.

L'utilisation d'un chiffrement à clé symétrique pose un problème fondamental : celui de l'échange de la clé. Si Alice veut envoyer un message crypté à Bernard, elle doit d'abord lui transmettre la clé qu'elle va utiliser. Mais comment la lui transmettre de manière sécurisée ?

Ce problème est un grand classique : si Alice chiffre la clé, Bernard ne pourra pas la lire puisqu'il lui faut déjà avoir connaissance de la clé pour déchiffrer le message.

À RETENIR

Un système de chiffrement symétrique nécessite un canal de communication déjà sécurisé pour transmettre la clé. Il n'est donc pas possible d'utiliser cette technique pour initialiser une communication entre deux nouveaux interlocuteurs.

4.2.2 - Le chiffrement asymétrique

Inventé vers 1976, le *chiffrement asymétrique* ou « chiffrement à clé publique » repose sur l'utilisation d'une clé composée des deux parties :

- une partie privée (dite *clé privée* et généralement notée s pour « secret »), conservée par le titulaire de la clé et qui n'est jamais communiquée ;
- une partie publique (dite *clé publique* et généralement notée p pour « public ») qui peut être connue de tous, y compris des espions.

Tout message crypté avec une partie de la clé (s ou p) ne peut être décrypté qu'avec l'autre partie de la même clé (respectivement p ou s).

Avec cette technologie, Alice et Bernard communiquent de la manière suivante :

- ils génèrent chacun une clé asymétrique dont ils conservent la clé privée (s) et échangent ouvertement leur clé publique ;
- Ève voit passer les clés publiques et peut les utiliser ;
- Alice écrit un message (M) et utilise la clé publique de Bernard pour le chiffrer en C = p(M) ;
- elle transmet ouvertement (C) ;
- Ève intercepte (C) mais ne peut rien en faire sans la clé privée de Bernard ;
- Bernard reçoit (C) et le déchiffre avec sa clé privée pour restituer M = s(C) ;
- Bernard répond à Alice avec un nouveau message qu'il chiffre en utilisant la clé publique d'Alice ;
- seule Alice, à son tour, peut déchiffrer le message émis par Bernard en utilisant sa clé privée.

Mathématiquement, le chiffrement asymétrique est basé sur l'utilisation de fonctions à sens unique possédant une brèche secrète :

- une fonction à sens unique est une fonction mathématique f dont la réciproque f^{-1} est très difficile à trouver (nécessite des milliers d'années de calcul) ;
- une brèche secrète est une donnée liée à une fonction à sens unique f dont la connaissance permet de trouver très facilement la fonction réciproque f^{-1}.

En cryptographie asymétrique, la fonction à sens unique constitue la clé publique et sa brèche secrète la clé privée. La mise en pratique la plus connue de cette mécanique est le chiffrement RSA.

Le chiffrement asymétrique est hélas plus consommateur de temps de calcul (complexité de mise en œuvre) que le chiffrement symétrique et nécessite des clés de plus en plus longues pour garantir la sécurité.

En pratique, on l'utilise au début d'une communication pour constituer un canal sécurisé servant à échanger une nouvelle clé secrète symétrique. Une fois la clé symétrique en possession des seuls interlocuteurs légitimes, c'est elle qui est utilisée pour le chiffrement de la communication.

Voir exercices 8 et 9 page 302.

COURS

4.3 Utilisation sur le web

Le chiffrement est très largement utilisé dans les communications internet. Les sites dits sécurisés (HTTPS) mettent en œuvre le protocole TLS qui impose :

- le chiffrement asymétrique RSA pour la création et l'échange d'une clé symétrique secrète ;
- le chiffrement symétrique AES, avec la clé échangée, sur l'ensemble de la session de communication.

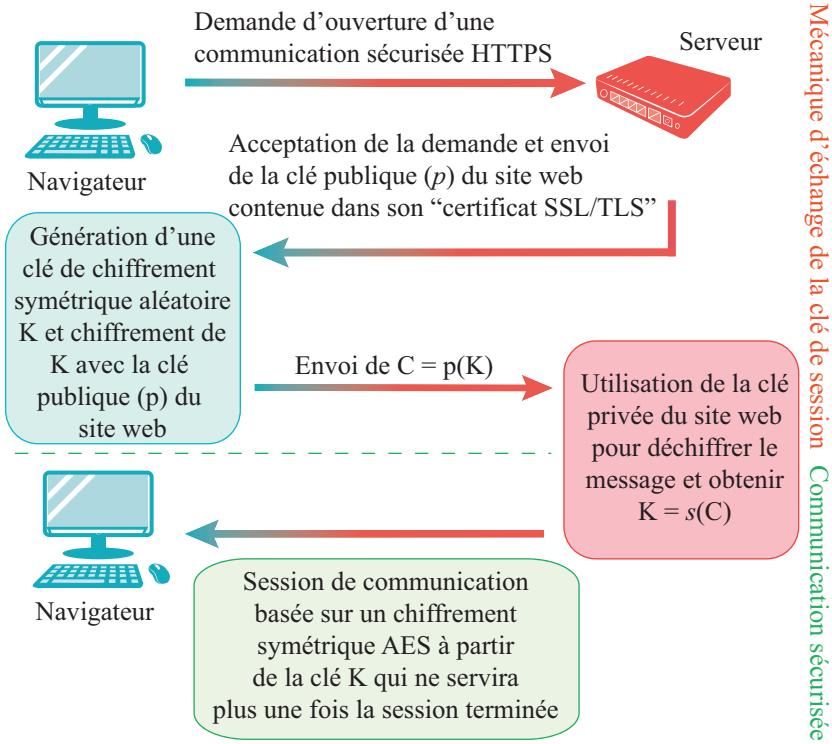


Figure 8.11 – Utilisation des chiffrements symétrique et asymétrique sur le web

4.4 Authentification d'un message

En plus d'être idéale pour l'établissement d'une communication, le chiffrement asymétrique permet aussi de signer et d'authentifier un message. Il apporte alors la garantie que le message a bien été écrit par son auteur et n'a pas été altéré au cours de la communication.

Pour ce faire, on utilise un outil mathématique supplémentaire appelé *fonction de hachage* (aussi nommé CRC pour *Cyclic Redundancy Check*). Il s'agit d'une fonction calculable cycliquement sur un message aussi long que nécessaire pour produire un résultat très court nommé aussi *condensat*.

Le CRC, ou condensat, ne permet pas du tout de reconstruire le message car plusieurs messages très différents pourraient avoir le même CRC.

Soient un message M et son CRC calculé par une fonction de hachage f (le CRC est généralement un entier sur 32 ou 64 bits, indépendamment de la longueur du message). f est conçue de manière à ce qu'une petite altération de M en M' provoque une image par f très différente : f(M') sera très éloigné de f(M).

La transmission d'un message authentifié se fait généralement de la manière suivante :

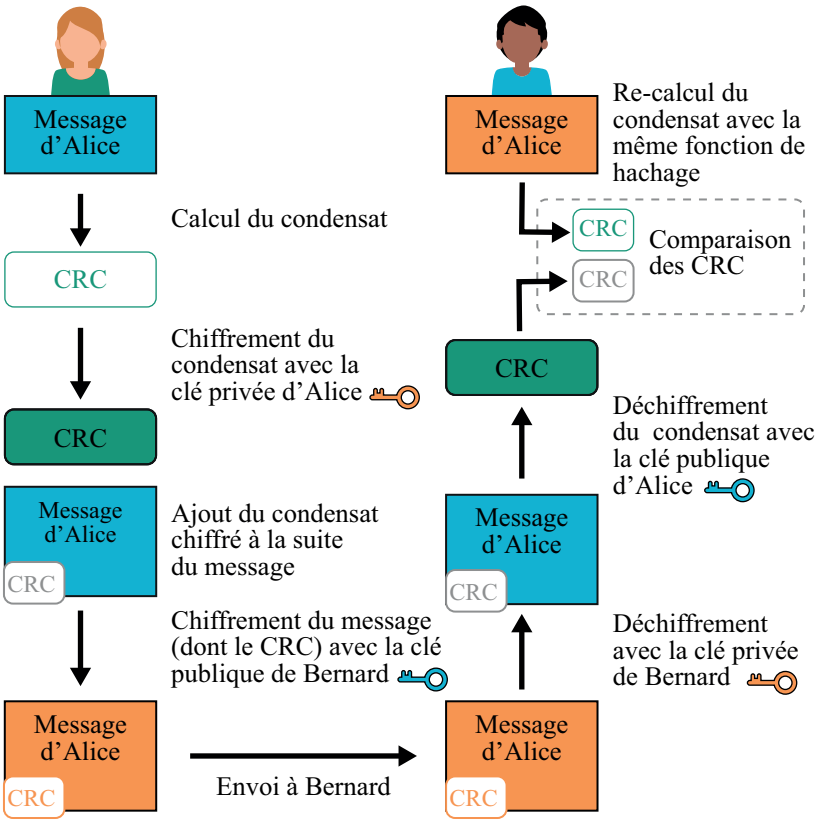


Figure 8.12 – Transmission d'un message authentifié

Si le CRC reçu est identique au CRC calculé alors Bernard a les garanties :

- que c'est bien la clé privée d'Alice qui a chiffré le CRC (puisque la clé publique d'Alice a permis de le déchiffrer) ;
- que le CRC a bien été calculé sur ce message exactement (sinon Bernard aurait obtenu un CRC différent).

Il peut en déduire que le message est authentique, c'est-à-dire qu'il n'a pas été usurpé, ni modifié, ni rédigé à l'insu d'Alice puisqu'elle est la seule à pouvoir utiliser sa clé privée pour chiffrer le condensat.

INTERROS

1 QCM Sur les SoCs

10 min — Corrigé p. 305

Pour chacune des questions suivantes, plusieurs propositions de réponses sont faites dont une seule est réellement pertinente. Laquelle ?

1. Un Soc est :
   - a Une puce électronique ;
   - b Une sorte de micro ordinateur sur une puce électronique ;
   - c Un système d'exploitation sur une puce électronique ;
   - d Une mémoire intégrée à une puce électronique.
2. La fluidité d'une vidéo sur un appareil à SoC dépend avant tout :
   - a de la mémoire embarquée dans le SoC ;
   - b du nombre de cœurs du CPU ;
   - c de la puce graphique (GPU) ;
   - d des ISP (Image Signal Processor).

2 QCM Sur les processus

5 min — Corrigé p. 305

Pour chacune des questions suivantes, plusieurs propositions de réponses sont faites dont une seule est réellement pertinente. Laquelle ?

1. Un processus est :
   - a Un programme ;
   - b Un programme en cours d'exécution ;
   - c Un programme en fin d'exécution ;
   - d Un programme achevé.
2. Un scheduler est :
   - a Un sous-ensemble du système d'exploitation qui organise les tâches à exécuter ;
   - b Un programme antivirus ;
   - c Un processus de gestion de calendrier ;
   - d Un programme qui diagnostique les faiblesses du système d'exploitation.

3 V/F Sur les SoCs

10 min — Corrigé p. 305

Les affirmations suivantes sont-elles vraies ou fausses ? Justifier la réponse.

1. Avec un Soc, le matériel peut être mis à jour.
2. La consommation énergétique d'un SoC est faible.
3. Dans un SoC, tous les éléments sont indépendants.

Architecture matérielle

4 La RAM

★ — 5 min — Corrigé p. 305

1. Donner une définition de la RAM.
2. Quand on allume un ordinateur par exemple, qu'est-ce qui est avant tout stocké dans la RAM ?

5 Les processeurs

★ — 10 min — Corrigé p. 305

1. À quoi correspond la fréquence d'un processeur ?
2. Par quel sigle désigne-t-on un processeur ?
3. Dans un processeur, qu'est-ce qu'un cœur ? Quel est l'avantage principal d'avoir plusieurs cœurs dans un processeur par rapport à la consommation énergétique de ce dernier ?
4. Comment désigne-t-on les données reçues par un processeur par les contrôleurs de périphérique ?
5. Est-il vrai qu'il ne peut y avoir qu'un unique processeur dans une machine ?

6 Les SoCs

★ — 10 min — Corrigé p. 306

1. Définir un SoC.
2. En plus de sa petite taille, quel est l'autre principal avantage d'un SoC concernant l'énergie ? Expliquer pourquoi cet avantage est possible.
3. Quel est l'autre avantage d'un SoC en terme économique ?
4. Citer un inconvénient majeur des SoCs.

Processus et ressources partagées

7 Processus

★ — 15 min — Corrigé p. 306

1. Un processus est-il un programme ?
2. Expliquer dans les grandes lignes le rôle de l'ordonnanceur dans la gestion des processus.
3. Citer tous les états possibles d'un processus en expliquant brièvement chaque état.
4. Expliquer le principe d'interblocage.

INTERROS

Sécurité des échanges de données

8 ROT13

★ 10 min Corrigé p. 307

Écrire une fonction Python `chiffrement_rot13(message)` permettant de chiffrer un message à l'aide de l'algorithme de chiffrement ROT13. Pour cela,

- on s'assurera que toutes les lettres du message sont en minuscule ;
- si le message comporte plusieurs mots, on le découpera ;
- on écrira une fonction `rot13(mot)` chiffrant un mot à l'aide de l'algorithme demandé ;
- dans la fonction `chiffrement_rot13(message)`, on fera appel à la fonction `rot13(mot)`.

Chiffrer alors le message : « cheval indomptable ».

9 Chiffrement à l'aide de XOR

★★ 30 min Corrigé p. 308

1 On donne le programme suivant :

```
def getBytes(msg):
  return [ord(c) for c in msg]

phrase  = "My name is Bond, James Bond..."
liste = getBytes(phrase)
print(phrase)
print(liste)
```

(a) Qu'affiche-t-il ?

(b) Son résultat peut-il être considéré comme une version chiffrée du texte initial ? Expliquez.

2 Écrire la fonction réciproque `getText(liste)` qui, à partir du résultat de `getBytes(msg)`, retourne le texte initial `msg`.

3 On veut maintenant utiliser une clé secrète pour encoder les valeurs de la liste résultat de getBytes() afin qu'il ne soit pas possible à un intervenant de retrouver le texte initial par `getText()` sans connaître la clé.

Pour ce faire, on décide d'utiliser l'opération « OU exclusif » (XOR) entre chaque octet du texte et celui de la clé. XOR est une opération binaire (notée ^ en Python) qui s'effectue sur chacun des bits des deux opérandes :

XOR	0	1
0	0	1
1	1	0

Tableau 8.1 – Table du « OU exclusif »

Par exemple,

56 ^ 83 → 0011 1000 ^ 0101 0011 = 0110 1011 → 107.

Supposons, pour le moment, que la clé est plus longue que le texte :

Clé = « Ceci est une clé de 41 caractères de long »

(a) Écrire une fonction qui reçoit un texte et une clé en arguments et retourne les octets du texte combinés aux octets respectifs de la clé via l'opération XOR. Le résultat sera fourni sous forme d'une liste d'octets.

(b) Appliquer cette fonction au texte « message urgent » avec la clé « cryptographie005 ».

Afficher la liste des octets ainsi cryptés obtenue et comparer avec l'utilisation de la fonction `getBytes()` de la question 1.

(c) Constater qu'une fois l'opération XOR appliquée, il n'est pas possible de récupérer la phrase d'origine avec `getText()`.

(d) Pour décrypter le texte, il faut appliquer l'opération inverse du XOR à la liste des octets chiffrés. Donc, il faut disposer de la clé secrète.

L'opération XOR est involutive, c'est-à-dire que :

si a XOR b = c alors c XOR a = b et c XOR b = a.

Pour l'inverse, il suffit donc de l'appliquer une nouvelle fois, sur les octets chiffrés, avec la même clé.

Constater qu'une fois l'opération XOR appliquée à nouveau, le texte peut être déchiffré.

4 On veut maintenant pouvoir utiliser une clé de longueur quelconque (typiquement plus courte que le message à transmettre). Il n'y aura alors pas assez d'octets dans la clé pour calculer le XOR en une seule fois sur tout le message. On va donc répéter la clé, autant de fois que nécessaire, pour chiffrer le message en entier.

Par exemple,

- `Clé: "Key"`
- `Message: "Ceci est le message"`
  `"KeyKeyKeyKeyKeyKeyK"`

INTERROS

•
```
Message chiffré = [
                        ord("C") ^ ord("K"),
                        ord("e") ^ ord("e"),
                        ord("c") ^ ord("y"),
                        ord("i") ^ ord("K"),
                        ... ]
```

(a) Écrire une fonction `computeXorRepeatingKey(text,key)` qui effectue ce calcul et l'appliquer sur une phrase.

(b) Utiliser la même fonction, avec la même clé pour déchiffrer le résultat précédent et vérifier que la reconstruction de la phase conduit bien à celle de départ.

1 QCM Sur les SoCs

Enoncé p. 300

1. Réponse **b**. Un SoC est un système intégré sur une puce qui regroupe l'essentiel d'un ordinateur : processeurs, mémoire, périphériques d'interface, etc.
2. Réponse **c**. En effet, les performances graphiques dépendent avant tout de la puce graphique. C'est elle qui gère les calculs des images donc il est nécessaire d'avoir un bon GPU pour que les calculs se fassent rapidement.

2 QCM Sur les processus

Enoncé p. 300

1. Réponse **b**. Un processus est un programme en cours d'exécution.
2. Réponse **a**. Un ordonnanceur est en effet un composant du noyau du système d'exploitation qui choisit l'ordre d'exécution des processus.
3. Réponse **c**. Il existe en effet trois niveau d'ordonnancement des processus : l'ordonnancement *à long terme*, l'ordonnancement à *moyen terme* et l'ordonnancement à *court terme*.

3 V/F Sur les SoCs

Enoncé p. 300

1. *Faux*. C'est d'ailleurs l'un des inconvénients du SoC.
2. *Vrai*. C'est l'un des avantages du Soc avec sa petite taille.
3. *Faux*. Au contraire, dans un SoC, tous les éléments sont liés (comme par exemple le processeur dont la fréquence est directement liée à celle de la carte graphique).

4 La RAM

Enoncé p. 301

1. La RAM (Random Access Memory), ou *mémoire vive*, est un espace de stockage des données volatiles, c'est-à-dire une zone où les programmes en cours d'exécution ainsi que les données qu'ils peuvent manipuler ou qui sont nécessaires sont stockés.
2. Au démarrage d'une machine, la RAM est toujours vide. Lorsqu'on l'allume, c'est avant tout le système d'exploitation qui y est chargé, sans quoi la machine ne saurait pas quoi faire.

5 Les processeurs

Enoncé p. 301

1. La fréquence d'un processeur correspond à la fréquence à laquelle il exécute des instructions de tous les microprogrammes.

2 Le sigle désignant un processeur est CPU : *Central Processor Unit*.

3 Dans un processeur, un cœur est une unité qui exécute des instructions. Un cœur travaille en parallèle des autres.

Plus il y a de cœurs dans un processeur et plus le nombre d'opérations exécutées par seconde augmente sans avoir à augmenter la fréquence globale du processeur, ce qui augmenterait la consommation énergétique de ce dernier. De plus le processeur chaufferait davantage.

4 Les données reçues par un processeur par les contrôleurs de périphérique sont désignées par le sigle « I/O » (pour « Input/Output ») ou « E/S » (pour « Entrée/Sortie »).

5 Il n'est pas vrai de dire qu'il ne peut y avoir qu'un unique processeur dans une machine. En effet, le GPU (*Graphics Processing Unit*, contrôleur vidéo), est lui aussi un processeur. Donc un ordinateur portable standard, par exemple, comporte au moins deux processeurs.

6 Les SoCs

Enoncé p. 301

1 Un SoC (ou *System on Chip*) est un système intégré sur une puce qui regroupe les principaux composants d'un ordinateur (comme par exemple la mémoire, le processeur ou encore les bus de communication).

2 La faible consommation d'énergie est un grand avantage des SoCs. Elle est due à sa taille, si petite que très peu d'énergie suffit par rapport à une carte mère, où les distances entre composants sont bien plus grands. On obtient ainsi dans un SoC moins de dissipation thermique et de pertes par effet Joule.

3 Par sa petite taille, un SoC est très économique à fabriquer : les coûts de fabrication sont réduits car ils peuvent être produits en très grande quantité à l'aide de machines (donc pas de main d'œuvre à payer).

4 La petite taille d'un SoC est certes un avantage pour les raisons que nous avons citées précédemment, mais cela constitue aussi un inconvénient majeur : si un composant est défectueux au sein du SoC, c'est le SoC entier qui ne fonctionne pas bien ; il faut donc le remplacer entièrement.

7 Processus

Enoncé p. 301

1 Un programme est stocké sur une mémoire de masse. Tant qu'il n'est pas sollicité, il ne fait rien. Or, un processus est une instance de programme qui est exécutée (de manière continue ou discontinue) par le système d'exploitation.

On ne peut donc pas dire qu'un processus est un programme.

2 L'ordonnanceur, ou *scheduler*, est un module du système d'exploitation qui attribue du temps d'exécution (quantum de temps) à chaque processus en décidant ceux qui seront prioritaires. Il gère aussi les changements de contextes.

3 Les différents états d'un processus sont :

- « *nouveau* » : un programme a été sélectionné pour démarrage et ses instructions ont été recopiées en mémoire ;
- « *prêt* » : le processus est en attente de quantum(s) de temps CPU pour exécuter son travail ;
- « *en cours d'exécution* » : un ou plusieurs quantum(s) de temps a (ont) été alloué(s) au processus. Il exécute ses instructions jusqu'à écoulement du temps imparti ;
- « *bloqué* » : le processus est en attente d'une ou plusieurs ressource(s) et ne peut rien faire en attendant. Il n'est pas éligible à l'obtention d'un quantum de temps CPU.
- « *terminé* » : le processus a achevé son exécution jusqu'à sa dernière instruction et il doit être déchargé de la mémoire.

4 On parle d'interblocage quand un processus (A) est en attente d'un travail que doit fournir un processus (B) qui lui-même est en attente d'un travail que doit fournir (A) (qui ne peut pas le fournir car il est bloqué).

8 ROT13

Enoncé p. 302

Une proposition de programme est donnée page suivante.

Dans la fonction `rot13(mot)`,

- on commence par initialiser une chaîne de caractères `r` vide, qui contiendra le résultat attendu ;
- on parcourt chaque lettre du mot : on détermine son code ASCII avec `ord(lettre)` et on lui ajoute 13. Ensuite, on soustrait le code ASCII du caractère 'a' afin d'associer 'a' à 0, 'b' à 1, etc. Le résultat de ces opérations est pris modulo 26 car l'alphabet est considéré comme un « cycle » de longueur 26. On obtient ainsi le rang de la lettre chiffrée. Il suffit ensuite d'ajouter `ord('a')` au résultat pour obtenir le code ASCII de la lettre chiffrée. Ensuite, on utilise la commande `chr(code ASCII)` pour obtenir le caractère souhaité, que l'on ajoute à la chaîne `r`.

Dans la fonction `chiffrement_rot13(message)`, on transforme le message de sorte qu'il ne contienne que des minuscules, puis on parcourt la liste des mots composant le message (obtenue à l'aide de la méthode `split`) afin d'appliquer à chacun d'eux la fonction `rot13(mot)`.

Le message « cheval indomptable » est alors chiffré en :

« puriny vaqbzcgnoyr »

```
def rot13(mot):
    r = ''
    for lettre in mot:
        r += chr( (ord(lettre) + 13 - ord('a'))%26
        + ord('a') )

    return r

def chiffrement_rot13(message):
    r = ''
    message = message.lower()
    for mot in message.split(' '):
        r += rot13(mot) + ' '

    return r

print(chiffrement_rot13('cheval indomptable'))
```

9 Chiffrement à l'aide de XOR

Enoncé p. 302

1 **(a)** La fonction `getBytes(msg)` retourne la liste des valeurs décimales de chaque octet qui compose le texte reçu `msg`. Par exemple, si le texte reçu est « My » alors la fonction renvoie la liste `[77,121]` car 77 est le code ASCII de « M » et 121 celui de « y ».

Ainsi, le programme proposé affiche la liste de tous les codes ASCII des caractères qui composent le message informé.

(b) Le résultat du programme est effectivement une manière d'encoder (écrire différemment) le texte initial via un encodage réversible. Techniquement, on peut donc dire qu'il s'agit d'un chiffrement dont la sécurité repose sur la connaissance de l'algorithme utilisé (remplacement de chaque lettre par la valeur décimale de l'octet correspondant).

L'algorithme utilisé étant extrêmement simple, le chiffrement serait immédiatement « cassé » par un hacker.

2 Une fonction possible est la suivante :

```
def getText(liste):
  return "" . join([chr(b) for b in liste])
```

Il suffit ici de parcourir la liste informée en argument de la fonction et de trouver, à l'aide de la fonction `chr()`, le caractère qui correspond au code ASCII de l'élément de la liste.

3 **(a)** Voici une proposition de fonction :

```
def computeXor(text,key):
  textBytes   = getBytes(text)
  keyBytes  = getBytes(key)
  return [textBytes[i] ^ keyBytes[i] for i in
    range(len(textBytes))]
```

(b) En utilisant la fonction `computeXor`, la liste suivante est retournée :

```
[14, 23, 10, 3, 21, 8, 2, 82, 20, 2, 15, 12, 11, 68]
```

alors qu'avec la fonction `getBytes`, on obtient la liste suivante :

```
[109, 101, 115, 115, 97, 103, 101, 32, 117, 114, 103,
101, 110, 116]
```

Bien entendu, nous n'obtenons pas du tout la même liste ; la première liste est bien plus « mystérieuse » que la seconde.

(c) Avec `print( getText(xorBytes) )`, où `xorBytes` est la liste retournée par la fonction `computeXor()`, nous avons une belle surprise... Selon l'interpréteur que l'on utilise, l'affichage diffère mais une chose est sûre : ce n'est pas le message d'origine que nous voyons !

(d) Avec le programme suivant :

```
phrase    = "message urgent"
key     = "cryptographie005"
xorBytes  = computeXor(phrase,key)
print(xorBytes)
crypText = getText(xorBytes)
xorBytes2 = computeXor(crypText,key)
print(xorBytes2)
decrypt   = getText(xorBytes2)
print(decrypt)
```

on s'aperçoit en effet que le message initial apparaît en clair.

CORRIGÉS

4 **(a)** Voici une fonction possible :

```
def computeXorRepeatingKey(textBytes,key) :
  keyBytes  = getBytes(key)
  return [textBytes[i] ^ keyBytes[i % len(
    keyBytes)] for i in range(len(textBytes))]

text      = "My name is Bond, James Bond... I'm
    trying to find what is written into this
    message"
key       = "privateKey007"
textBytes     = getBytes(text)
encodedBytes  = computeXorRepeatingKey(textBytes
    ,key)

print("Phrase:\r\n{}".format(text))
print("Octets non encodés:\r\n{}".format(
    textBytes))
print("Octets encodés:\r\n{}".format(
    encodedBytes))
```

(b) On peut utiliser le programme suivant :

```
decodedBytes  = computeXorRepeatingKey(
    encodedBytes,key)
decodedText   = getText(decodedBytes)

print("Phrase:\r\n{}".format(text))
print("Octets encodés:\r\n{}".format(
    encodedBytes))
print("Octets décodés:\r\n{}".format(
    decodedBytes))
print("Restitution de la phrase:\r\n{}".format(
    decodedText))
```

Nous avons ici décidé d'afficher les listes des octets encodés et décodés, mais nous pouvons afficher directement la seule phrase déchiffrée, qui correspond bien à la phrase initiale.

Téléchargez le programme complet

Chapitre 9

Baccalauréat blanc

Voir nos conseils de méthode « Spécial Bac » dans la partie « Méthodes de travail » en début d'ouvrage.

La durée de l'épreuve est de 3 h 30 min.

Le ou la candidat·e devra choisir trois des cinq exercices proposés.

Chaque exercice est noté sur 4 points.

1 Programmation

★★ 70 min > Corrigé p. 324

Trois suites numériques (u_n), (v_n) et (w_n) sont définies par leurs premiers termes $u_0 = 1$, $v_0 = 2$, $w_0 = 3$ et, pour tout entier naturel n, par les égalités suivantes :

$$\begin{cases} u_{n+1} &= u_n + v_n - w_n \\ v_{n+1} &= u_n - v_n + w_n \\ w_{n+1} &= \dfrac{u_n - v_n}{w_n} \end{cases}$$

Nous souhaitons écrire en Python un programme permettant de calculer u_n, v_n et w_n pour un entier n quelconque.

1 *Approche naïve.*

(a) Compléter l'algorithme suivant :

```
u ← 1
v ← 2
w ← 3
n est un entier
Pour k variant de 1 à n:
    ...
    ...
    u ← ...
    v ← ...
    w ← ...
Fin du Pour
Afficher u, v et w
```

(b) Proposer un programme Python correspondant à cet algorithme afin d'obtenir u_7, v_7 et w_7.

(c) Montrer que la complexité de cet algorithme est en $O(n)$.

2 *Approche dynamique.*

(a) Proposer une approche dynamique de type « Bottom Up » de l'algorithme précédent. On donnera directement un programme écrit en Python qui permet d'obtenir u_7, v_7 et w_7.

SUJET

(b) Compléter le programme de type « Top Down » suivant afin qu'il affiche u_7, v_7 et w_7.

```
def construct(n , U = None):
    if not U:
        U = [ (1,2,3) ]
            + [ (None,None,None) ] * n
    if n == 0:
        return ...
    else:
        Uprec = ...
        u, v, w = ...
        u_next = u + v - w
        v_next = u - v + w
        w_next = (u - v) / w
        U += [ (u_next , v_next , w_next) ]
        return ...

print( ... )
```

2 Base de données

★★ 70 min > Corrigé p. 326

L'entreprise MQWZ, dont le début de l'activité est enregistré au 2 janvier 2020, demande à ses salariés une présence du lundi au vendredi, de 8 h 00 à 12 h 00 et de 14 h 00 à 17 h 00. Son PDG, Évariste Galois, demande à son directeur des ressources humaines (DRH), Bart Ernest, de créer une base de données dans laquelle seront enregistrées diverses informations sur le personnel. Pour cela, l'entreprise dispose du système de gestion MySQL. La requête suivante est utilisée pour créer la base de données.

```
CREATE DATABASE MQWZ
```

Partie A

1 Deux requêtes sont écrites page suivante.

(a) Quel rôle joue l'attribut « id » pour chacune des tables `personnel` et `postes` ?

(b) L'attribut « postes_id » de la table `personnel` doit contenir des valeurs de l'attribut « id » de la table `postes`.
Quel nom donne-t-on à cet attribut ?

```
CREATE TABLE personnel (
  id INT NOT NULL AUTO_INCREMENT ,
  nom VARCHAR(30) NOT NULL ,
  prenom VARCHAR(20) NOT NULL ,
  rue VARCHAR(50) NOT NULL ,
  code_postal VARCHAR(5) NOT NULL ,
  ville VARCHAR(50) NOT NULL ,
  telephone VARCHAR(12) NOT NULL ,
  postes_id INT NOT NULL ,
  PRIMARY KEY (id)
);

CREATE TABLE postes (
  id INT NOT NULL AUTO_INCREMENT ,
  designation VARCHAR(50) NOT NULL ,
  PRIMARY KEY (id)
);
```

2 Que fait la requête suivante ?

```
INSERT INTO personnel
  (nom, prenom, rue, code_postal, ville,
   telephone, postes_id)
VALUES
  ('GALOIS', 'Evariste', '15, rue du Moulin',
   '37000', 'TOURS', '+33789562325', '1'),
  ('ERNEST', 'Bart', '5, rue Sésame', '18000',
   'BOURGES', '+33601254212', '2')
```

3 Bart Ernest doit enregistrer les intitulés « PDG » et « DRH » dans la table `postes`.
Écrire la requête SQL de sorte que les enregistrements correspondent à ceux déjà faits dans la table `personnel` ?

4 Une autre table est créée à l'aide de la requête suivante :

```
CREATE TABLE absences (
  id INT NOT NULL AUTO_INCREMENT ,
  id_personnel INT NOT NULL ,
  jour DATE NOT NULL ,
  nb_heures DOUBLE NOT NULL ,
  PRIMARY KEY (id)
);
```

Elle contient toutes les absences du personnel.

SUJET

Le type « DATE » permet d'enregistrer une date au format `YYYY-MM-DD`.

Écrire une requête SQL permettant de compter le nombre d'enregistrements de la table `absences` pendant la période du lundi 1er février 2021 au vendredi 5 février 2021 inclus.

Partie B

Une table `salaires` est créée à l'aide de la requête suivante :

```
CREATE TABLE salaires (
  id INT NOT NULL AUTO_INCREMENT ,
  id_personnel INT NOT NULL ,
  jour_virement DATE NOT NULL ,
  montant DOUBLE NOT NULL ,
  PRIMARY KEY (id)
);
```

où :

- « id_personnel » contient une valeur de l'attribut « id » de la table `personnel` ;
- « jour_versement » contient la date (au format YYYY-MM-DD) où est émis le virement du salaire de la personne correspondant à l'attribut « id_personnel » ;
- « montant » contient le salaire (en euros) de la personne.

1 **(a)** Écrire une requête SQL permettant de retourner le salaire moyen de l'entreprise sur l'ensemble de la table `salaires`.

(b) Écrire une requête SQL permettant de retourner le salaire moyen de l'entreprise en 2020, en supposant que tous les salaires de l'année 2020 ont été enregistrés le dernier jour de chaque mois.

(c) Écrire une requête SQL permettant de retourner le salaire moyen en 2020 de toutes les personnes de l'entreprise sauf le PDG, en supposant que tous les salaires de cette année sont enregistrés.

2 Écrire une requête SQL permettant de calculer le salaire annuel d'Évariste Galois en 2020.

3 Afin de compléter la table `salaires`, une table nommée `base_salaires` est créée à l'aide de la requête suivante :

```
CREATE TABLE base_salaires (
  id INT NOT NULL AUTO_INCREMENT,
  id_personnel INT NOT NULL ,
  salaire_horaire DOUBLE NOT NULL ,
  base_horaire INT NOT NULL ,
  PRIMARY KEY (id)
);
```

où :

- « id_personnel » contient une valeur de l'attribut « id » de la table `personnel` ;
- « salaire_horaire » contient le salaire horaire (en euros) de la personne correspondant à l'attribut « id_personnel » ;
- « base_horaire » contient le nombre d'heures de travail que la personne doit effectuer par jour.

La requête suivante :

```
CREATE VIEW abs_view AS
  SELECT
    id_personnel, SUM(nb_heures) AS total
  FROM
    absences
  WHERE
    jour >= '2021-02-01'
    AND
    jour <= '2021-02-28'
    GROUP BY
      id_personnel
```

crée une vue nommée « abs_view », qui est une table dont les enregistrements sont les « id » des personnes ayant été absentes au moins une fois en février 2021 ainsi que le nombre total d'heures d'absence pour chacune d'elles.

(a) Écrire une requête SQL créant une vue nommée « suppressions » créant une table à deux attributs : « id_personnel » et « total_suppressions » contenant respectivement les « id » des personnes ayant été absentes en février 2021 ainsi que la retenue sur salaire de ces personnes (valeur obtenue en multipliant le nombre d'heures d'absence par le salaire horaire de la personne).

(b) En déduire une requête permettant d'afficher le salaire de février 2021 du PDG de l'entreprise, sachant qu'il y a 20 jours de travail au cours de ce mois.

3 Programmation et base de données ★★

70 min > Corrigé p. 329

On considère la classe `wezz` définie en annexe (voir page 317).

1 **(a)** Voici une suite d'instructions en mode console :

```
>>> obj = wezz(8,3,15)
>>> print( obj.somme() )
 298
>>> print( obj.div() )
 [1, 2, 149, 298]
```

Comment interpréter ces deux résultats ?

SUJET

(b) L'appel à la méthode `somme` dans la méthode `div` pose un problème en terme de complexité. Lequel ?

Proposer alors une amélioration de la méthode `div`.

(c) Compléter le programme suivant afin de définir la méthode `div` de manière encore plus concise.

```
def div(self):
    ...
    L = [ ... ]
    return L
```

2 On s'intéresse à la méthode `fct`.

(a) Expliquer le rôle de l'attribut `*args`.

(b) Expliquer pourquoi cette méthode peut être qualifiée de récursive.

(c) Qu'est-ce qui nous assure l'arrêt de cette méthode ?

(d) Que retourne `wezz(3,7,5).fct()` ?
On donnera uniquement le résultat.

3 On souhaite maintenant effectuer un grand nombre de calculs sur différents « wezz » et stocker les résultats dans une base de données que l'on nomme « wezz.db ». On utilise pour cela le système de gestion de base de données *MySQL*.

On considère alors la fonction `insert_sql` définie en annexe 2 (voir page 318).

(a) Expliquer la requête SQL :

```
INSERT INTO wezz.simulations (a,b,c,s)
VALUES (%s,%s,%s,%s)
```

(b) Quelle requête SQL permet d'obtenir tous les enregistrements où la valeur de « s » est strictement inférieure à 500 ?

(c) Quelle requête SQL permet d'obtenir tous les enregistrements classés selon un ordre croissant sur « a », puis sur « b », puis sur « c » ?

(d) Quelle requête SQL permet d'obtenir la moyenne des valeurs de « s » ?

(e) On suppose que l'on a fait appel à la fonction `insert_sql` à cinq reprises.

Qu'affiche alors la requête :

```
SELECT COUNT(*) FROM simulations
```

Annexe 1 : définition de la classe « wezz »

```
class wezz:
    def __init__(self , a , b, c):
        self.a = abs( int(a) )
        self.b = abs( int(b) )
        self.c = abs( int(c) )

    def somme(self):
        return self.a**2 + self.b**2 + self.c**2

    def div(self):
        L = [ ]
        for n in range(1 , self.somme()+1):
            if self.somme() % n == 0:
                L.append(n)
        return L

    def fct(self , *args):
        if len( args ) == 0:
            a = self.a
            b = self.b
            c = self.c
        else:
            a = args[0]
            b = args[1]
            c = args[2]

        if min(a,b,c) == 0:
            return 0
        else:
            return a * b * c + self.fct(a-1,b-1,c-1)
```

SUJET

Annexe 2 : définition de la fonction « insert_sql »

```
import random
import mysql.connector

def insert_sql():
    conn = mysql.connector.connect(
        host="localhost",
        user="root",
        password="")

    cursor = conn.cursor()
    r = "CREATE DATABASE IF NOT EXISTS wezz"
    cursor.execute( r )
    r = """CREATE TABLE IF NOT EXISTS wezz.simulations
        (id INT PRIMARY KEY AUTO_INCREMENT UNIQUE,
         a INT, b INT, c INT, s INT)"""

    cursor.execute( r )

    for k in range(100):
        a = randint(1,20)
        b = randint(1,20)
        c = randint(1,20)
        w = wezz(a,b,c)
        s = w.fct()
        r = """INSERT INTO wezz.simulations
            (a,b,c,s) VALUES (%s,%s,%s,%s)"""
        val = (a,b,c,s)
        cursor.execute(r , val )

    conn.commit()
    cursor.close()
    conn.close()
```

Téléchargez les annexes 1 et 2 en une fois.

4 Programmation et graphes

★★ 70 min > Corrigé p. 331

On souhaite implémenter en Python un graphe où les sommets sont les régions de la France métropolitaine et où les arêtes représentent la mitoyenneté entre deux régions. On notera $\mathcal{G}$ ce graphe.

Figure 9.1 – Carte de la France métropolitaine et de ses régions

Les sommets du graphes sont notés :

- IDF : Île-De-France ;
- HDF : Hauts-De-France ;
- N : Normandie ;
- B : Bretagne ;
- PDL : Pays-De-La-Loire ;
- NA : Nouvelle-Aquitaine ;
- O : Occitanie ;
- C : Corse ;
- PACA : Provence-Alpes-Côte-D'Azur ;
- ARA : Auvergne-Rhône-Alpes ;
- BFC : Bourgogne-Franche-Comté ;
- GE : Grand-Est ;
- CVL : Centre-Val-De-Loire.

On supposera que la Corse n'est mitoyenne qu'à la région PACA.

1 Donner une représentation sagittale de $\mathcal{G}$.

2 **(a)** Quel est l'ordre de $\mathcal{G}$?

(b) $\mathcal{G}$ est-il connexe ?

(c) $\mathcal{G}$ est-il cyclique ?

3 **(a)** Donner la première ligne de la matrice d'adjacence A de $\mathcal{G}$. Pour cela, on conviendra de noter les sommets dans l'ordre alphabétique, donc dans l'ordre suivant :

ARA, B, BFC, C, CVL, GE, HDF, IDF, N, NA, O, PACA, PDL.

(b) Donner l'ensemble d'adjacence de $\mathcal{G}$ sous la forme « sommet S_i : liste de sommets reliés à S_i ».

(c) Pour implémenter $\mathcal{G}$, que semble-t-il plus cohérent d'utiliser entre la matrice d'adjacence et l'ensemble d'adjacence ? Justifier.

SUJET

4 On utilise la classe suivante pour implémenter $\mathcal{G}$:

```
class Graphe:
    def __init__(self):
        self.sommets = {}

    def edge(self, u, v):
        if v not in self.sommets:
            self.sommets[v] = []
        if u not in self.sommets:
            self.sommets[u] = []
        self.sommets[u].append(v)
        self.sommets[v].append(u)
```

(a) On décide de noter « `G` » le graphe implémenté.
Quelle instruction permet de créer une instance `G` de la classe `Graphe()` ?

(b) Quelles instructions permettent de définir les arrêtes du sommet NA à tous ses sommets mitoyens ?

(c) On suppose à présent que tous les sommets du graphe ont été définis sur l'objet `G` à la manière de la question précédente et on s'intéresse à l'instruction :

```
>>> print(G.sommets())
```

Expliquer ce que fait cette instruction. On écrira, à titre d'exemple, les deux premières lignes du résultat escompté.

5 À la classe « Graphe » précédemment définie, on ajoute la méthode `mystere` suivante :

```
def mystere(self):
    from collections import deque
    sortie = []
    tmp = deque()
    tmp.append('NA')
    while tmp:
        S = tmp.popleft()
        if S not in sortie:
            sortie.append(S)
            unvisited = [n for n in self.sommets[S]
    if n not in sortie]
            tmp.extend(unvisited)
    return sortie
```

Que renvoie-t-elle ?

6 Proposer un algorithme qui permet d'obtenir la matrice d'adjacence du graphe $\mathcal{G}$ obtenue à la question 3.a.

7 Proposer alors une méthode `mat` à ajouter à la classe `Graphe` qui retourne la matrice d'adjacence du graphe.

5 Réseau

★★ 70 min > Corrigé p. 334

Partie A : routage statique

Soit un réseau domestique composé des éléments suivants :

- une box internet (BOX) ;
- une TV connectée à un port ethernet de la box ;
- un routeur/point d'accès wifi (R), connecté à un port Ethernet de la box et qui gère un réseau sans fil différent du réseau filaire (représenté en orange) ;
- un premier PC (PC1) relié à un port Ethernet de la box et qui dispose aussi d'une connexion wifi ;
- un second PC (PC2), une caméra wifi (CAM) et un smartphone (TEL), tous les trois connectés au réseau wifi.

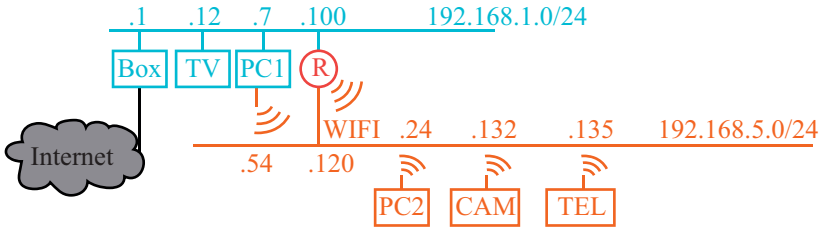


Figure 9.2 – Représentation du réseau domestique

1 Compléter la table de routage (simplifiée) du PC1 :

Destination réseau	Masque réseau	Passerelle
192.168.1.0	? . ? . ? . ?	Interface locale ethernet
? . ? . ? . ?	? . ? . ? . ?	Interface locale wifi
0.0.0.0	0.0.0.0	192.168.1.1

2 Que signifie la dernière ligne du tableau ?

3 L'ordinateur « PC1 » se connecte à la caméra Wifi (CAM).
Quelle ligne de la table de routage ci-dessus correspond à la route utilisée ? Expliquer comment est opéré ce choix et quelle est l'adresse IP source utilisée par PC1.

4 Écrire la table de routage (simplifiée) de l'ordinateur « PC2 » à la manière de celle présentée en question 1.

5 Le smartphone a-t-il connaissance de l'adresse locale de la box internet ? Expliquer.

SUJET

Partie B : routage RIP

On considère la topologie réseau suivante, qui peut être assimilée à un graphe où chaque sommet représente un routeur implémentant le protocole RIP.

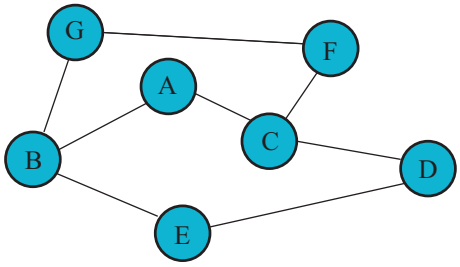


Figure 9.3 – Graphe représentant la topologie réseau

1. Rappeler les grands principes du protocole RIP.
2. On s'intéresse au routeur A.
   - **(a)** Quelles sont les routes qui lui permettent de joindre D et les métriques correspondantes ?
   - **(b)** Quelles seront les routes établies par chacun des routeurs pour rejoindre D après convergence (c'est-à-dire une fois que les routeurs auront échangé suffisamment d'informations RIP pour que les tables de routage de chacun soient complètes). Mentionner les métriques correspondantes.
3. Donner les tables de routage complètes des routeurs A et B après convergence.
4. Le routeur A tombe en panne.
   - **(a)** Expliquer sommairement la gestion de cette panne par le protocole RIP.
   - **(b)** Donner la nouvelle table de routage de B après re-convergence.
   - **(c)** Quelle(s) sera (seront) la (les) route(s) B → C conservée(s) ? Expliquer.

Partie C : routage OSPF

On considère la topologie réseau suivante, qui peut être assimilée à un graphe où chaque sommet représente un routeur implémentant le protocole OSPF. Les chiffres sont les coûts supposés des métriques OSPF assignées à chaque liaison inter-routeur.

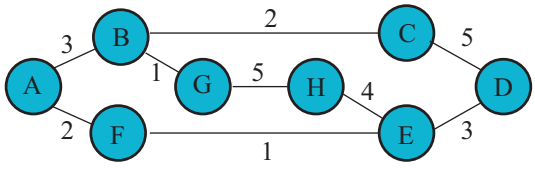


Figure 9.4 – Graphe représentant la topologie réseau

1 Rappeler les grands principes du protocole OSPF (comparé au protocole RIP).

2 Quelle sera la route utilisée entre G et E après convergence (c'est-à-dire lorsque les routeurs auront effectué suffisamment d'échanges de tables de routage pour que toutes les relations d'adjacence soient propagées) ? Quelle aurait été cette route avec le protocole RIP ?

3 On veut déterminer la table de routage de B sous la forme :

Destination	Passerelle à utiliser	Métrique
A		
C		
...	...	...

(a) Quel est l'algorithme utilisé dans la pratique pour déterminer le chemin optimal entre B et chacun des routeurs destinataires envisagés ? Expliquer.

(b) Donner la table de routage complète de B après convergence.

4 *Question bonus.* Le routeur D est à présent connecté à un routeur R supplémentaire qui appartient à un réseau inconnu (réseau mystère).

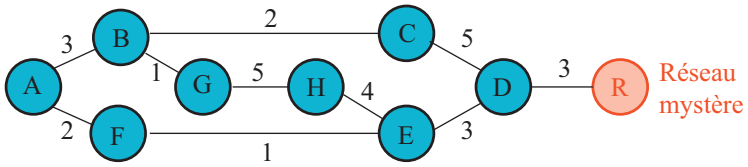


Figure 9.5 – Graphe après connexion à un réseau mystère

Après quelques minutes de fonctionnement du réseau, la convergence est supposée établie et R échange sa table de routage avec chaque routeur A à H. Voici ci-contre un extrait de la table de routage transmise par R.

Destination	Passerelle à utiliser	Métrique
C	D	8
D	D	3
...	...	...
I	I	2
J	J	1
K	I	6
L	J	2
M	J	5

(a) À partir de ces informations, compléter la table de routage de B précédemment renseignée à la question 3.

(b) Dessiner une topologie possible des routeurs I à M du réseau mystère (il y a beaucoup de possibilités !). On impose uniquement qu'elle soit compatible avec les tables de routage de B et de R.

CORRIGÉ

1 Programmation

> Enoncé p. 311

1 *Approche naïve.*

(a) L'algorithme complété est le suivant :

```
u ← 1
v ← 2
w ← 3
n est un entier
Pour k variant de 1 à n:
    utemp ← u
    vtemp ← v
    u ← u + v - w
    v ← utemp - v + w
    w ← (utemp - vtemp) / w
Fin du Pour
Afficher u, v et w
```

Dans la mesure où la variable `u` est affectée d'une nouvelle valeur dans la boucle, et dans la mesure où il est nécessaire d'avoir la valeur précédente de `u` pour calculer la nouvelle valeur de `v`, il est nécessaire de stocker cette valeur précédente de `u` dans une autre variable (que nous avons ici nommée `utemp`).

Il en est de même pour `v` car il est nécessaire d'avoir la valeur précédente de `v` pour calculer la nouvelle valeur de `w`.

(b) Un programme Python correspondant à cet algorithme est le suivant :

```
u, v, w = 1, 2, 3
n = 7

for i in range(1,n+1):
    u_tmp = u
    v_tmp = v
    u = u + v - w
    v = u_tmp - v + w
    w = (u_tmp - v_tmp) / w

print('u =',u,', v =',v,', w =',w)
```

(c) Afin de connaître la complexité de l'algorithme, il faut compter le nombre d'instructions élémentaires (affectations et opérations).

- Il y a avant tout quatre affectations pour `u`, `v`, `w` et `n` ;
- ensuite, pour chaque passage dans la boucle itérative, il y a :

  → 1 affectation pour la variable `i` ;

  → 2 affectations pour `u_tmp` et `v_tmp`

  → 2 opérations et 1 affectation pour `u`, `v` et `w`.

  Il y a donc :

$$4 + n(1 + 2 + 3 \times 3) = 12n + 4$$

  instructions élémentaires.
  La complexité est alors affine, donc en $O(n)$.

Remarque : dans la pratique, on n'effectue pas tous ces calculs. On se contente de dénombrer le nombre de boucles (ici, il y en a n) et de déceler s'il y a des imbrications (ce qui aurait pour conséquence d'élever au carré le n s'il y en a une, élever au cube s'il y en a deux, etc.).
Ainsi, dans cette question, nous aurions pu simplement dire qu'il y a n boucles sans imbrication, ce qui signifie que la complexité est en $O(n)$.

2 **(a)** L'idée est ici de construire une liste pour chaque suite.
Un programme possible est alors le suivant :

```
u, v, w = [1], [2], [3]
n = 7

for k in range(1,n+1):
    u.append( u[k-1] + v[k-1] - w[k-1] )
    v.append( u[k-1] - v[k-1] + w[k-1] )
    w.append( (u[k-1] - v[k-1]) / w[k-1] )

print('u =',u[n],', v =',v[n],', w =',w[n] )
```

(b) Le programme de type « Top Down » complété est le suivant :

```
def construct(n , U = None):
    if not U:
        U = [ (1,2,3) ] + [ (None,None,None) ] *
     n
    if n == 0:
        return (1, 2, 3)
```

(suite du programme page ci-contre)

CORRIGÉ

```
    else:
        Uprec = construct(n-1 , U)
        u, v, w = Uprec[0], Uprec[1], Uprec[2]
        u_next = u + v - w
        v_next = u - v + w
        w_next = (u - v) / w
        U += [ (u_next , v_next , w_next) ]
        return U[-1]

print( construct(7) )
```

2 Base de données

> Enoncé p. 312

Partie A

1 **(a)** Dans chacune des tables `personnel` et `postes`, la présence de « `PRIMARY KEY (id)` » assure le fait que l'attribut « id » joue le rôle de *clé primaire* pour chacune d'elles.

(b) L'attribut « postes_id » met en relation deux tables : il s'agit donc d'une *clé étrangère*.

2 La requête proposée est une requête de type « INSERT INTO » : elle enregistre donc une entrée, en l'occurrence dans la table `personnel`. Nous pouvons constater la présence de deux entrées (avec pour noms « GALOIS » et « ERNEST ».

Elle effectue donc deux enregistrements dans la table `personnel`.

3 La requête SQL permettant d'enregistrer les deux types de postes « PDG » et « DRH » est la suivante :

```
INSERT INTO postes
  (designation)
VALUES
  ('PDG'), ('DRH')
```

Il faut ici faire attention au fait que l'id de chaque enregistrement doit correspondre à ceux déjà informés dans la requête d'insertion des noms dans la table `personnel` ; or, nous voyons que « GALOIS » à un `postes_id` égal à 1 et « ERNEST », un `postes_id` égal à 2, d'où le fait que « PDG » (correspondant au poste de GALOIS) est inséré avant « DRH » (afin d'avoir un `id` égal à 1, tout en attribuant un `id` égal à 2 pour « DRH »).

4 Une requête SQL permettant de compter le nombre d'enregistrements de la table `absences` pendant la période du lundi 1er février 2021 au vendredi 5 février 2021 inclus est celle page ci-contre.

```
SELECT COUNT(*)
FROM absences
WHERE jour >= '2021-02-01' AND jour <= '2021-02-05'
```

Il faut ici faire attention à la présence des apostrophes pour les dates. Sans ces caractères, la requête échoue.

Partie B

1 **(a)** Une requête SQL permettant de retourner le salaire moyen de l'entreprise sur l'ensemble de la table `salaires` est la suivante :

```
SELECT AVG(montant) FROM salaires
```

Par cette requête, on calcule la moyenne sur l'intégralité des enregistrements de la table car aucune condition n'y apparaît.

(b) Une requête SQL permettant de retourner le salaire moyen de l'entreprise en 2020, en supposant que tous les salaires de l'année 2020 ont été enregistrés le dernier jour de chaque mois est :

```
SELECT
  AVG(montant)
FROM
  salaires
WHERE
  jour_virement >= '2020-01-01'
  AND
  jour_virement <= '2020-12-31'
```

(c) Une requête SQL permettant de retourner le salaire moyen en 2020 de toutes les personnes de l'entreprise sauf le PDG, en supposant que tous les salaires de l'année 2020 ont été enregistrés le dernier jour de chaque mois est :

```
SELECT
  AVG(montant)
FROM
  salaires
WHERE
  jour_virement >= '2020-01-01'
  AND
  jour_virement <= '2020-12-31'
  AND
  id_personnel <> 1
```

CORRIGÉ

2 Une requête SQL permettant de calculer le salaire annuel d'Évariste Galois en 2020 est la suivante :

```
SELECT
  SUM(montant)
FROM
  salaires
WHERE
  id_personnel = 1
  AND
  jour_virement >= '2020-01-01'
  AND
  jour_virement <= '2020-12-31'
```

3 **(a)** Une requête possible est la suivante :

```
CREATE VIEW suppressions AS
  SELECT
    abs_view.id_personnel ,
    abs_view.total * base_salaires.base_horaire
    AS total_suppressions
  FROM
    abs_view
      INNER JOIN base_salaires
      ON
      abs_view.id_personnel
      =
      base_salaires.id_personnel
```

(b) Une requête possible est :

```
SELECT
  20 * base_salaires.salaire_horaire
  * base_salaires.salaire_horaire
  - suppressions.total_suppressions
  AS salaire_fevrier
FROM
  base_salaires
  INNER JOIN suppressions
  ON
    base_salaires.id_personnel
    =
    suppressions.id_personnel
WHERE
  base_salaires.id_personnel = 1
```

Pour cette requête, nous avons utilisé le fait que pour calculer le salaire après absences éventuelles, il faut :

- calculer le salaire « normal » :

  20 jours × nombre d'heures travaillées par jour × salaire horaire
- enlever la retenue calculée dans la vue `suppressions`.

3 Programmation et base de données

> Enoncé p. 315

1 **(a)** Procédons par étape.

- La méthode `somme` : elle permet simplement de retourner la somme $a^2 + b^2 + c^2$. Dans notre cas, elle retourne $8^2 + 3^2 + 15^2 = 298$.
- La méthode `div` : on commence ici par initialiser une liste vide `L`, puis on entre dans une boucle itérative en `n`, pour `n` variant de 1 à 298 dans notre cas.
  L'instruction « `if self.somme() % n == 0` » teste si la somme obtenue est divisible par `n`. Si le test est positif, on ajoute dans la liste `L` la valeur de `n`.
  Ainsi, `L` contiendra à la sortie de la boucle itérative tous les diviseurs de 298.

Conclusion : `[1, 2, 149, 298]` est l'ensemble de tous les diviseurs de 298.

(b) Dans la méthode `div`, on calcule la somme $a^2 + b^2 + c^3$ 298 fois dans notre cas, ce qui n'est pas idéal. On peut facilement y remédier en réécrivant la méthode de la manière suivante :

```
def div(self):
    L = [ ]
    somme = self.somme()
    for n in range(1 , somme+1):
        if somme % n == 0:
            L.append(n)
    return L
```

(c) On peut en effet définir la liste `L` de manière plus concise ainsi :

```
def div(self):
    somme = self.somme()
    L = [n for n in range(1 , somme+1) if somme
    % n == 0]
    return L
```

2 **(a)** L'attribut `*args` permet de faire passer des valeurs à la méthode `fct` de manière optionnelle.

CORRIGÉ

(b) Cette méthode est récursive car elle s'appelle au sein même de sa définition : elle fait appel à elle-même.

(c) Par définition, les variables `a`, `b` et `c` sont entières et positives (ou nulles).
De plus, à chaque appel de la méthode `fct`, leurs valeurs diminuent de 1. Il existe donc un appel où une des variables est égale à 0. Or, quand cela se produit, la méthode retourne « 0 », et s'arrête donc.
C'est ce qui nous assure que la méthode s'arrête.

(d) `wezz(3,7,5).fct()` retourne 168. En effet, on calcule successivement : $\min(3,7,5) = 3$, $\min(2,6,4) = 2$, $\min(1,5,3) = 1$, $\min(0,4,2) = 0$: on calcule alors : $1 \times 5 \times 3 + 0 = 15$, puis $2 \times 6 \times 4 + 15 = 63$ et finalement $3 \times 7 \times 5 + 63 = 168$.

3 **(a)**

```
INSERT INTO wezz.simulations (a,b,c,s)
VALUES (%s,%s,%s,%s)
```

Cette requête SQL permet d'insérer un enregistrement dans la table `simulations` de la base de données `wezz`. Cet enregistrement est composé de quatre attributs : les valeurs de *a*, *b*, *c* et *d*.

Remarque : on ne mentionne pas l'attribut « id » car il est auto-incrémenté, donc sa valeur est générée par le SGBD.

(b) Une requête SQL permettant d'obtenir tous les enregistrements où la valeur de « s » est strictement inférieure à 500 est :

```
SELECT * FROM wezz.simulations WHERE s < 500
```

(c) La requête SQL qui permet d'obtenir tous les enregistrements classés selon un ordre croissant sur « *a* », puis sur « *b* », puis sur « *c* » est :

```
SELECT * FROM wezz.simulations ORDER BY a,b,c
```

(d) La requête SQL qui permet d'obtenir la moyenne des valeurs de « *s* » est :

```
SELECT AVG(wezz.simulations.s)
FROM wezz.simulations
```

(e) La requête suivante permet de compter le nombre d'enregistrements dans la table `simulations`.

```
SELECT COUNT(*) FROM wezz.simulations
```

Dans la mesure où la fonction `insert_sql` insère 100 enregistrements dans la table `simulations` et qu'on y a fait appel cinq fois, la requête doit afficher : « 500 » (5×100).

4 Programmation et graphes

> Enoncé p. 319

1 Une représentation sagittale du graphe $\mathcal{G}$ est la suivante :

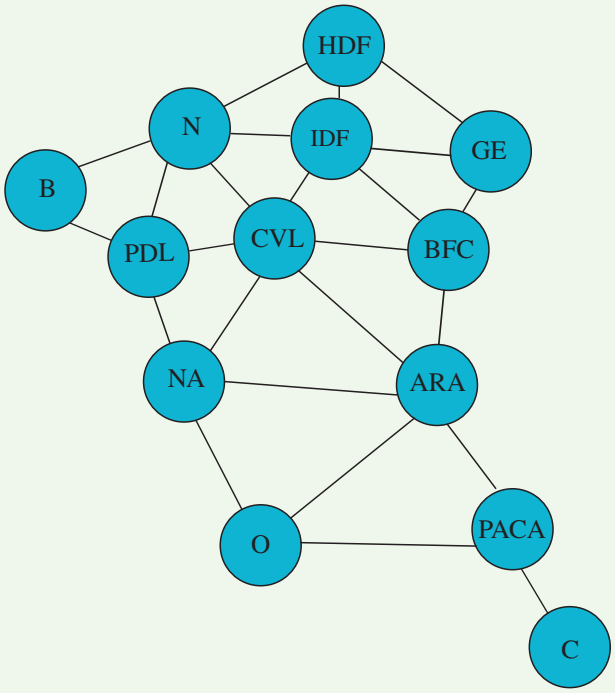


Figure 9.6 – Représentation sagittale du graphe $\mathcal{G}$

2 **(a)** L'ordre d'un graphe est le nombre de sommets. Donc l'ordre de $\mathcal{G}$ est ici égal au nombre de régions représentées, à savoir 13.

(b) Un graphe est connexe s'il ne comporte pas de sommets isolés, ce qui est le cas ici. Donc $\mathcal{G}$ est connexe.

(c) Un graphe est cyclique si, pour n'importe quel sommet, il existe un chemin le reliant à lui-même, ce qui n'est pas le cas pour $\mathcal{G}$. En effet, il n'existe aucun cycle de sommet C.

3 **(a)** Les sommets sont classés par ordre alphabétique, ce qui donne :

ARA, B, BFC, C, CVL, GE, HDF, IDF, N, NA, O, PACA, PDL.

On ne demande ici que la première ligne $A[1]$ de la matrice :

$$A[1] = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Ici, les « 1 » représentent les connexions entre « ARA » et les sommets : BFC, CVL, NA, O et PACA.

CORRIGÉ

(b) L'ensemble d'adjacence de $\mathcal{G}$ est :

- ARA : BFC, CVL, NA, O, PACA
- B : N, PDL
- BFC : ARA, CVL, GE, IDF
- C :PACA
- CVL : ARA, BFC, IDF, N, NA, PDL
- GE : BFC, HDF, IDF
- HDF : GE, IDF, N
- IDF : BFC, CVL, GE, HDF, N
- N : B, CVL, IDF, HDF, PDL
- NA : ARA, CVL, O, PDL
- O : ARA, NA, PACA
- PACA : ARA, C, O
- PDL : B, CVL, N, NA

(c) Afin d'implémenter le graphe $\mathcal{G}$, il semble plus cohérent d'utiliser la liste d'adjacence. En effet, la matrice A est creuse (beaucoup de « 0 »), ce qui occuperait de la mémoire pour rien, sans compter que le nombre de coefficients à saisir étant important, le risque de faire des erreurs dans la saisie n'est pas négligeable.

4 **(a)** L'instruction qui permet de créer une instance `G` de la classe `Graphe` est :

```
G = Graphe()
```

(b) Les instructions permettant de définir les arrêtes du sommet NA à ses voisins sont :

```
liste = [ 'PDL' , 'CVL' , 'ARA' , 'O']
for i in liste:
  G.edge('NA' , i)
```

Remarque : bien entendu, il y a plusieurs possibilités pour cette question. Nous aurions pu éviter la boucle et écrire quatre instructions de la forme « `G.edge('NA','<sommet>')` ».

(c) L'instruction « `print( G.sommets() )` » affiche le dictionnaire qui contient tous les sommets du graphe $\mathcal{G}$ (en tant que clés) et tous les sommets qui sont reliés aux clés (dans une liste). Ainsi, les deux première lignes affichées sont :

```
ARA : ['BFC', 'CVL', 'NA', 'O', 'PACA']
B : ['N', 'PDL']
```

5 La méthode `mystere` ajoutée à la classe `Graphe` est une méthode de parcours (en largeur) à partir du sommet « NA ».

En effet, la liste `tmp` est une file dans laquelle on place en tête le sommet « NA ». Ensuite, tant que cette file n'est pas vide (`while tmp:`), on définit un sommet `S` comme étant le premier élément de la file ; s'il n'est pas dans la liste `sortie` (initialement vide), on l'y ajoute. On définit alors une liste `unvisited` complétée par les sommets qui sont dans la valeur correspondant à la clé du sommet et qui ne sont pas dans la liste `sortie` dans le dictionnaire `sommets` du graphe.

Il s'agit donc bien ici d'un algorithme de parcours de graphe.

6 Voici un algorithme qui permet d'obtenir la matrice A :

```
A ← matrice de dimension 13 remplie de 0
S ← liste des sommets du graphe triée par ordre
                                        alphabétique
Pour chaque élément "e" de S:
   L ← liste des sommets connectés à "e"
   Pour chaque élément "c" de L:
     i ← indice de "e" dans S
     j ← indice de "c"
     A[i][j] ← 1
   Fin Pour
Fin Pour
Afficher A
```

7 Une méthode `mat` possible est la suivante :

```
def mat(self):
    A = [ [0] * len(self.sommets) for i in range(
    len(self.sommets) )]
    S = sorted( list( self.sommets.keys() ) )
    for nom in S:
        for i in self.sommets[nom]:
            A[ S.index(nom) ][ S.index(i) ] = 1
    return A
```

Télécharger la classe complète.

CORRIGÉ

5 Réseau

> Enoncé p. 321

Partie A : routage statique

1 La table complétée est la suivante :

Destination réseau	Masque réseau	Passerelle
192.168.1.0	255.255.255.0	Interface locale ethernet
192.168.5.0	255.255.255.0	Interface locale wifi
0.0.0.0	0.0.0.0	192.168.1.1

2 La dernière ligne de la table signifie qu'il s'agit de la passerelle « par défaut », c'est-à-dire que tous les paquets de données TCP/IP qui ne correspondent pas à un des autres réseaux listés seront envoyés à l'hôte 192.168.1.1 (la box internet).

Cette configuration (usuelle) est indispensable pour permettre le routage des échanges vers les sites internet et tous les services du réseau public. La box sert d'intermédiaire. Elle fait transiter les informations entre le réseau local et le web grâce à des protocoles de translation d'adresses IP qui ne sont pas au programme de terminale.

3 PC1 doit joindre l'hôte : 192.168.5.132 (CAM). L'algorithme de recherche de la passerelle/interface à utiliser est le suivant :

- PC1 examine tour à tour les lignes de sa table de routage, en commençant par celles dont le masque de réseau est le plus long (contient le plus grand nombre de bits à 1).
- Pour chaque ligne examinée, il effectue un « ET » logique (binaire) entre l'adresse de destination et le masque réseau.
- Si le résultat de l'opération correspond à l'adresse du réseau destinataire (colonne de gauche), alors il utilise la passerelle mentionnée dans la ligne (colonne de droite). Sinon, il passe à l'examen de la ligne suivante.

Examen de la ligne 1.

	192.168.5.132	→	1100 0000 . 1010 1000 . 0000 0101 . 1000 0100
ET	255.255.255.0	→	1111 1111 . 1111 1111 . 1111 1111 . 0000 0000
=	192.168.5.0	←	1100 0000 . 1010 1000 . 0000 0101 . 0000 0000

Le résultat (192.168.5.0) ne correspond pas au réseau de destination de la ligne 1 (colonne de gauche). Ce n'est donc pas cette ligne qui est utilisée.

Examen de la ligne 2.

Le calcul est identique car les deux lignes ont le même masque de réseau. Donc on obtient le même résultat : 192.168.5.0.

Cette fois-ci, il correspond au réseau destination de la ligne 2 (colonne de gauche) donc c'est la passerelle située sur cette ligne : « Interface locale wifi » qui est utilisée.

La communication sera donc directe entre les deux équipements et empruntera le réseau Wifi auquel PC1 est directement connecté. Elle ne transitera pas par un routeur. L'adresse source utilisée par PC1 sera celle de son interface wifi : 192.168.5.54.

4 La table de routage de l'ordinateur « PC2 » est la suivante :

Destination réseau	Masque réseau	Passerelle
192.168.5.0	255.255.255.0	Interface locale wifi
0.0.0.0	0.0.0.0	192.168.5.120

- Les communications vers des destinataires situés sur le réseau Wifi (192.168.5.0/24) sont directes (ligne 1).
- Toutes les autres communications, à destination du réseau Internet ou des équipements situés sur le réseau local ethernet, transitent par le routeur/point d'accès R qui est donc la passerelle par défaut.

5 Le smartphone ne peut pas découvrir la topologie du réseau car ce n'est pas un routeur et il n'implémente pas de protocole de routage dynamique comme RIP ou OSPF. Il ne peut donc déduire que la présence des équipements connectés au même réseau que lui c'est-à-dire le réseau Wifi.

Il a la même table de routage que PC2 et envoie toute communication à destination du réseau Internet vers le routeur R qui est sa passerelle par défaut. C'est R qui se charge de faire le relai vers la box internet et qui retransmettra les réponses obtenues au smartphone.

Partie B : routage RIP

1 Les grands principes du protocole RIP sont :

- RIP est un protocole de routage dynamique à vecteur de distance.
- Chaque routeur envoie périodiquement sa table de routage aux routeurs voisins.
- Les routeurs complètent ainsi progressivement leur table de routage et effectuent un décompte du nombre de routeurs présents sur chaque liaison.
- Ils utilisent ce nombre (nombre de sauts) en tant que métrique pour déterminer les itinéraires les plus pertinents.
- Lorsqu'il existe plusieurs liaisons différentes pour rejoindre un même routeur, seule(s) celle(s) de métrique la plus faible est (sont) conservée(s).

CORRIGÉ

2 **(a)** Les routes qui permettent au routeur A de joindre D et les métriques correspondantes sont les suivantes :

Route	Métrique
A → C → D	1
A → B → E → D	2
A → B → G → F → C → D	4
A → C → F → G → B → E → D	5

(b) Les routes établies par chacun des routeurs pour rejoindre D après convergence ainsi que leurs métriques sont :

Routeur	Destination	Passerelle	Métrique
A	D	C	1
B		E	1
C		D	0
E		D	0
F		C	1
G		B	2
		F	2

3 Les tables de routage complètes des routeurs A et B après convergence sont :

Routeur	Destination	Passerelle	Métrique
A	B	B	0
	C	C	0
	D	C	1
	E	B	1
	F	C	1
	G	B	1
B	A	A	0
	C	A	1
	D	E	1
	E	E	0
	F	G	1
	G	G	0

4 **(a)** Le protocole RIP gère la panne du routeur A de la manière suivante :

- les routeurs B et C, voisins de A, ne recevront plus ses données RIP périodiques. Au bout d'un certain temps, ils vont décréter A comme « absent » ;

- B et C vont désactiver de leurs tables de routage toutes les routes faisant intervenir A en tant que destinataire ou passerelle ;
- les nouvelles tables de routage vont se propager de proche en proche jusqu'à une nouvelle convergence.

(b) La nouvelle table de routage de B après re-convergence est la suivante :

Routeur	Destination	Passerelle	Métrique
B	C	E	2
	C	G	2
	D	E	1
	E	E	0
	F	G	1
	G	G	0

Le routeur C ne peut plus être atteint en utilisant A comme passerelle. Les deux autres chemins possibles font respectivement intervenir les routeurs E et G et ont une métrique identique de 2 sauts.

(c) Les métriques des deux routes étant identiques, B conserve ces deux itinéraires. Dans la pratique, il diffusera les paquets à destination de C, alternativement à E et G pour répartir la charge réseau entre les deux chemins jugés équivalents.

Partie C : routage OSPF

1 Les grands principes du protocole OSPF (comparé au protocole RIP) sont :

- OSPF est un protocole de routage dynamique à états de lien ;
- comme dans le cas de RIP, chaque routeur envoie périodiquement sa table de routage aux routeurs voisins et les routeurs complètent ainsi progressivement leur propre table de routage ;
- à la différence de RIP, la métrique est ici calculée en fonction de la performance de débit de chaque liaison. Plus une liaison est rapide plus sa métrique est faible ;
- lorsque qu'un même destinataire peut être joint par plusieurs liaisons différentes, les routeurs ne conservent que celle(s) de métrique la plus faible (donc au débit le plus élevé).

2 • Après convergence, seule la route de métrique OSPF la plus faible sera conservée soit :

$$G \rightarrow B \rightarrow A \rightarrow F \rightarrow E \text{ (métrique OSPF = 7).}$$

Les autres routes ($G \rightarrow H \rightarrow E$ et $G \rightarrow B \rightarrow C \rightarrow D \rightarrow E$) ont des métrique OSPF plus élevées (respectivement 9 et 11).

CORRIGÉ

- Le protocole RIP aurait retenu la route faisant intervenir le moins de routeurs possible (limitation du nombre de sauts) soit G → H → E dont le débit est pourtant inférieur. Le protocole OSPF semble donc plus orienté performances.

3 **(a)** Il s'agit ici de rechercher les plus courts chemins pondérés vers chacun des routeurs destinataires car ce sont uniquement ces chemins qui seront retenus une fois la convergence établie.
Dans la pratique, c'est l'algorithme de Dijkstra qui fait référence et permet de trouver les chemins les plus courts en effectuant un minimum d'opérations.

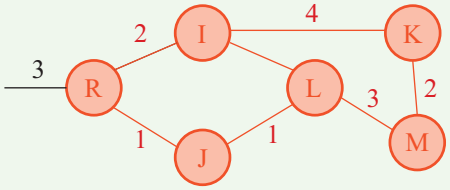
(b) La table de routage complète de B après convergence est la suivante :

Destination	Passerelle à utiliser	Métrique
A	A	3
C	C	2
D	C	7
E	A	6
F	A	5
G	G	1
H	G	6

4 **(a)** La table de routage de B précédemment renseignée complétée est :

Destination	Passerelle à utiliser	Métrique
A	A	3
C	C	2
D	C	7
E	A	6
F	A	5
G	G	1
H	G	6
I	C	12
J	C	11
K	C	16
L	C	12
M	C	15

(b) Voici une topologie possible, compatible avec la table de routage donnée dans l'énoncé et celle de R.



Annexe A

Liste des vidéos

Pour télécharger les programmes contenus dans cet ouvrage :
`http://www.prepamath.fr/ILTNSI`

Annexe B

Index

Liste des pages où figure un renvoi Nathan Live destiné à un usage interne

Liste des Programmes

Liste des renvois Vidéos

Liste des renvois Web