PC*

Lycée Marcelin Berthelot

Informatique

Chapitre I
Le langage Python

Le programme d'informatique limite les éléments du langage Python dont la connaissance est exigible des étudiants, et toute utilisation d'autres éléments du langage doit être accompagnée d'une documentation. Dans ce premier chapitre, nous allons passer en revue ces différents éléments ; les titres des sections ainsi que les passages en italique sont extraits du programme officiel.

1. Traits généraux

1.1 Typage dynamique

L'interpréteur détermine le type à la volée lors de l'exécution du code.

Sur un ordinateur, toutes les données manipulées sont représentées par une suite de bits (une quantité qui vaut 0 ou 1) regroupées en octets (= 8 bits). Cette suite de bits est appelée la *valeur* de la donnée. Mais connaître la valeur de la donnée n'est pas suffisant pour déterminer de quel objet il s'agit, car des objets de natures différentes peuvent posséder la même valeur.

Exemple. L'entier `88` et le caractère `'X'` possèdent tous deux la valeur `01011000` dans un codage usuel sur un octet.
C'est pourquoi une donnée est représentée par sa valeur *et par un type* qui décrit la façon dont doit être interprétée cette suite de bits.

```
In [1]: type(88)
Out[1]: <class 'int'>

In [2]: type('X')
Out[2]: <class 'str'>
```

lorsqu'on définit une donnée par l'intermédiaire du clavier, une analyse syntaxique est réalisée lors de l'exécution du code pour déterminer le type de l'objet souhaité, puis sa valeur est calculée.

$$\texttt{2} \xrightarrow{\text{analyse syntaxique}} \texttt{int} \xrightarrow{\text{représentation}} (\texttt{int, 00000010})$$

$$\texttt{'2'} \xrightarrow{\text{analyse syntaxique}} \texttt{str} \xrightarrow{\text{représentation}} (\texttt{str, 00110010})$$

$$\texttt{2.} \xrightarrow{\text{analyse syntaxique}} \texttt{float} \xrightarrow{\text{représentation}} (\texttt{float, 00010000})$$

À l'inverse, lorsqu'un objet doit être affiché sur l'écran, son type permet de déterminer quelle forme lui donner.

$$(\texttt{int, 01011000}) \xrightarrow{\text{affichage à l'écran}} \texttt{88}$$

$$(\texttt{str, 01011000}) \xrightarrow{\text{affichage à l'écran}} \texttt{'X'}$$

$$(\texttt{float, 01011000}) \xrightarrow{\text{affichage à l'écran}} \texttt{1024.}$$

Remarque. Cette détermination « à la volée » permet de ne pas avoir à déclarer le type des paramètres utilisés par une fonction. Néanmoins, pour des raisons aussi bien pédagogiques que de sûreté du code, il est possible de préciser le type des arguments est du résultat des fonctions (c'est ce que fait l'école Centrale dans ses sujets d'informatique). Ainsi,

```
def maFonction(n:int, X:[float], c:str, u) -> (int, numpy.ndarray):
```

signifie que la fonction `maFonction` prend quatre arguments : le premier `n` est un entier, le second `X` est une liste de flottants, le troisième `c` une chaîne de caractères, le type du dernier `u` n'étant pas précisé. Cette fonction renvoie un couple dont le premier élément est un entier et le second un tableau numpy. Néanmoins il ne vous est pas demandé d'utiliser cette syntaxe et vous pouvez utiliser la syntaxe classique :

```
def maFonction(n, X, c, u):
```

1.2 Principe d'indentation

Les impératifs de la programmation structurée nécessitent la définition de blocs d'instructions au sein des structures de contrôles (`def`, `for`, `while`, `if`, ...). Certains langages utilisent des délimiteurs pour encadrer ces blocs d'instructions (des parenthèses en C, des mots-clés en Fortran, etc), mais le langage Python se distingue en utilisant l'*indentation*, qui favorise la lisibilité du code.
Le début d'un bloc d'instructions est défini par un double-point (`:`), la première ligne pouvant être considérée comme un en-tête. Le corps du bloc est alors indenté d'un nombre d'espaces fixes (quatre par défaut), et le retour à l'indentation de l'en-tête marque la fin du bloc.

```
en-tête:
    bloc ........................
    ............................
    d'instructions ..............
```

Il est possible d'imbriquer des blocs d'instructions les uns dans les autres :

```
en-tête 1:
    ............................
    ............................
    en-tête 2:
        bloc ....................
        ........................
        d'instructions ..........
    ............................
    ............................
```

Cette structuration sert entre autre à définir de nouvelles fonctions, à réaliser des tests ou à effectuer des instructions répétitives.

1.3 Portée lexicale

Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du module.

```
In [1]: L = [1, 2, 3]

In [2]: def f():
....        L[0] = 0
....        return L

In [3]: f(), L
Out[3]: [0, 2, 3], [0, 2, 3]
```

```
In [1]: L = [1, 2, 3]

In [2]: def f():
....        L = [1, 2, 3]
....        L[0] = 0
....        return L

In [3]: f(), L
Out[3]: [0, 2, 3], [1, 2, 3]
```

```
In [1]: def f():
....        L[0] = 0
....        return L

In [2]: f()
NameError: name 'L' is not defined
```

Figure 1 – Illustration de la portée lexicale.

Considérons les trois exemples de la figure 1. Dans le premier cas, la liste `L` n'est pas définie au sein de la fonction `f`, donc c'est la liste définie dans l'espace global ligne 1 qui est modifiée. Dans le deuxième cas, une liste `L` est définie au sein de la fonction `f` donc c'est elle qui est modifiée par la fonction. Dans le dernier cas, la liste `L` n'est définie ni dans l'espace de la fonction, ni dans l'espace global donc l'interpréteur renvoie une erreur.

1.4 Appel de fonction par valeur

L'exécution de $f(x)$ évalue d'abord x puis exécute f avec la valeur calculée.

Observons le code suivant :

```
In [1]: a = 2

In [2]: def f(x):
....        x = 3
....        return x

In [3]: f(a), a
Out[3]: 3, 2
```

Dans un langage de programmation qui réaliserait un appel de fonction par *variable* plutôt que par *valeur*, le résultat retourné ligne 3 serait `3, 3` (la variable globale `a` aurait été modifiée).

Il convient néanmoins de noter que si l'argument de la fonction est un objet *mutable* (typiquement une liste) alors l'objet en question est effectivement modifié, car la valeur d'un objet mutable est l'adresse où cet objet est stocké en mémoire.

```
In [1]: L = [1, 2, 3]

In [2]: def f(X):
....        X[0] = 0
....        return X

In [3]: f(L), L
Out[3]: [0, 2, 3], [0, 2, 3]
```

2. Types de base

2.1 Opérations sur les entiers (`int`)

les opérations suivantes sont à connaître :

`+` l'addition de deux entiers ;

`-` la soustraction de deux entiers ;

`*` la multiplication de deux entiers ;

`**` l'élévation à la puissance (ne pas confondre avec `^` qui est une opération hors programme sur les entiers) ;

`//` le quotient de la division euclidienne ;

`%` *(avec des opérandes positifs)* le reste de la division euclidienne.

Ces opérations renvoient toujours un résultat de type `int`.

2.2 Opérations sur les flottants (`float`)

les opérations suivantes sont à connaître :

`+` l'addition de deux flottants ;

`-` la soustraction de deux flottants ;

`*` la multiplication de deux flottants ;

`/` la division de deux flottants ;

`**` l'élévation à la puissance.

Ces opérations renvoient toujours un résultat de type `float`.

2.3 Opérations sur les booléens (`bool`)

`not`, `or`, `and` *et leur caractère paresseux.*

Il convient d'expliquer la signification du caractère paresseux des opérateurs `or` et `and` : lors de l'évaluation d'une expression logique de type `(A and B)`, l'expression `A` est d'abord évaluée, *mais l'expression* `B` *n'est évaluée que si le résultat de l'évaluation de* `A` *est égal à* `True`. En effet, si `A` a été évalué à `False`, l'expression `(A and B)` sera elle aussi évaluée à `False`, quelle que soit l'évaluation de `B`.
Pour des raisons analogues, lors de l'évaluation de l'expression `(A or B)` l'expression `A` est d'abord évaluée, puis l'expression `B` *mais uniquement dans le cas où* `A` *aura été évaluée à* `False`.

Pour comprendre l'intérêt de cette évaluation paresseuse, considérons une fonction qui recherche un élément dans une liste et renvoie l'indice correspondant à sa première apparition :

```
def recherche(x, L):
    i = 0
    while i < len(L) and L[i] != x:
        i = i + 1
    return i
```

Lorsque `x` n'est pas présent dans la liste `L`, cette fonction renvoie la longueur de la liste :

```
In [1]: recherche(2, [1, 1, 1, 1])
Out[1]: 4
```

Inversons maintenant les deux composantes de l'expression booléenne et recommençons l'expérience :

```
def recherche(x, L):
    i = 0
    while L[i] != x and i < len(L):
        i = i + 1
    return i
```

```
In [2]: recherche(2, [1, 1, 1, 1])
IndexError: list index out of range
```

Dans ce deuxième cas de figure, lorsque l'entier `i` prend la valeur 4, la comparaison `L[i] != x` est réalisée en premier et conduit à une erreur puisqu'il n'y a pas de valeur indexée par 4 dans la liste `L`, alors que dans le premier cas de figure, c'est la comparaison `i < len(L)` qui est d'abord évaluée à `False` et permet, grâce au principe de l'évaluation paresseuse, de ne pas évaluer l'expression `L[i] != x`.

2.4 Comparaisons

Sont à connaître les opérations de comparaison suivantes :

`==` l'égalité ;

`!=` la différence ;

`<` strictement inférieur ;

`>` strictement supérieur ;

`<=` inférieur ou égal ;

`>=` supérieur ou égal.

Ces opérations de comparaison renvoient une valeur booléenne.

3. Types structurés

3.1 Structures indicées immuables (chaînes, tuples)

Une structure de donnée est *indicée* lorsqu'on peut accéder à ses éléments individuels par l'intermédiaire de leur indice ; une structure de donnée est *immuable* lorsque ces éléments individuels ne peuvent être modifiés. Deux structures de ce type sont à connaître :

– les chaînes de caractère (le type `str`) qui sont des suites de caractères alphanumériques délimités par des guillemets simples ou doubles. Par exemple `'Le nœud de vipères'` est une chaîne de caractère.

– les tuples (le type `tuple`) ou n-uplets qui sont des suites finies de valeurs séparées par des virgules (si besoin enclos par des parenthèses). Par exemple, `(2, 3, 5, 7, 11)` est un tuple.

Ces structures partagent les opérations suivantes :

– la fonction `len` permet de calculer leur longueur ;

```
In [1]: len('Le nœud de vipères')
Out[1]: 18

In [2]: len((2, 3, 5, 7, 11))
Out[2]: 5
```

– on accède aux éléments individuels avec la syntaxe `[k]` où k est un indice positif valide ;

```
In [3]: 'Le nœud de vipères'[4]
Out[3]: 'œ'

In [4]: (2, 3, 5, 7, 11)[3]
Out[4]: 7
```

– on peut en calculer une tranche avec la syntaxe `[i: j]` qui extrait tous les éléments dont les indices sont compris entre i inclus et j exclus ;

```
In [5]: 'Le nœud de vipères'[3: 7]
Out[5]: 'nœud'
```

– ces structures peuvent êtres concaténées avec l'opérateur `+` et répétées avec l'opérateur `*`.

```
In [6]: 'François' + ' Mauriac'
Out[6]: 'François Mauriac'

In [7]: (1,) * 10
Out[7]: (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

(Notez au passage comment se définit un tuple de longueur 1.)

3.2 Listes

Trois modes de création sont à connaître :

– par *compréhension* en suivant la syntaxe `[expr fox x in s]` ;

– par *duplication* en suivant la syntaxe `[expr] * n` ;

– par `.append` successifs.

La création par duplication présente deux inconvénients : elle ne permet que de créer des listes de valeurs égales et surtout, *elle se révèle incorrecte lorsque cette valeur est de type mutable*. Elle est donc à mes yeux à déconseiller. Pour créer la liste des dix premiers carrés on écrira donc l'une ou l'autre des versions suivantes :

```
carres = [x**2 for x in range(1, 11)]
```

```
carres = []
for x in range(1, 11):
    carres.append(x**2)
```

Tout comme pour les tuples et les chaînes de caractères, la fonction `len` renvoie la longueur d'une liste, on accède aux éléments par indice positif valide, on peut en calculer une tranche et réaliser une concaténation de deux listes par l'opérateur `+`.
Cependant, il faut prendre garde au fait que les deux syntaxes `L.append(x)` et `L = L + [x]`, même si elles conduisent au même résultat apparent, *ne sont pas équivalentes*. En effet, la seconde version *recrée* une nouvelle liste augmentée d'un élément là où la première se contente de rajouter un élément à une liste existante. On s'en doute, la deuxième version peut se révéler beaucoup plus coûteuse pour une liste de grande taille. Pour cette raison, on utilisera avec la plus grande parcimonie l'opérateur de concaténation `+` pour les listes.

La principale différence avec les structures de données précédentes est que les listes sont des structures de données *mutables*, dans le sens où il est possible de modifier un élément individuel d'une liste sans avoir à recréer la liste dans son entièreté.
La copie d'une liste `L` se réalise par la méthode `L.copy()` ou par le calcul d'une tranche comprenant l'entièreté de la liste `L[:]`.
Enfin, la méthode `L.pop()` supprime de la liste `L` son dernier élément et le renvoie en valeur de retour. Cette méthode modifie la liste en temps constant, contrairement à une syntaxe de type `L = L[0:len(L)-1]` qui recalculerait l'entièreté de la liste moins son dernier élément (et serait donc bien moins efficiente).

3.3 Dictionnaires (hors programme en 2021-22)

Un dictionnaire présente de nombreuses similitudes avec les listes, si ce n'est qu'au lieu d'accéder aux éléments individuels par le biais d'un indice, on y accède par le biais d'une *clef*.
La création d'un dictionnaire se réalise en suivant la syntaxe `{c1: v1, ... , cn: vn}` où $c_1,\dots,c_n$ sont des clefs (nécessairement deux-à-deux distinctes) et $v_1,\dots,v_n$ les valeurs qui leur sont associées. Ainsi, `{ }` crée un dictionnaire vide.

Si `D` est un dictionnaire et `c` une clef,

– l'expression `c in D` renvoie un booléen indiquant si la clef est présente ou non dans le dictionnaire ;

– `D[c]` renvoie la valeur associée à la clef si celle-ci est présente dans le dictionnaire (et provoque une erreur sinon) ;

– `D[c] = v` crée une nouvelle association si la clef n'est pas présente dans le distionnaire, et modifie l'association précédente sinon.

4. Structures de contrôle

4.1 Instruction d'affectation

Elle se réalise avec `=`. Notons qu'il est possible de *dépaqueter* des tuples, autrement dit utiliser la syntaxe :

```
x, y, z = a, b, c
```

pour affecter simultanément les valeurs a,b,c aux variables x,y,z. Ceci peut s'avérer utile pour permuter le contenu de deux variables en écrivant `x, y = y, x`.

4.2 Instruction conditionnelle

La syntaxe est la suivante :

```
if condition_1:
    bloc d'instructions_1
elif condition_2:
    bloc d'instructions_2
elif condition_3:
    bloc d'instructions_3
...
else:
    bloc d'instructions
```

Seul l'un de ces quatre (ou plus) blocs d'instructions sera réalisé :

- le premier si la condition 1 est réalisée ;
- le deuxième si la condition 1 n'est pas réalisée et la condition 2 réalisée ;
- le troisième si les conditions 1 et 2 ne sont pas réalisées et la condition 3 réalisée ;
- ...
- le dernier si aucune des conditions n'est réalisée.

Notez que les mots-clefs `elif` et `else` sont optionnels : s'ils ne sont pas présents et que la condition 1 n'est pas vérifiée, il ne se passe rien.

4.3 Boucle conditionnelle

Elles suivent la syntaxe :

```
while condition:
    bloc..............
    d'instructions......
```

Le bloc d'instructions est réalisé tant que la condition est vérifiée. Il est néanmoins possible de forcer la sortie d'une boucle à l'aide d'un `return` (exclusivement dans le cadre de la définition d'une fonction) ou plus généralement à l'aide de `break`.

4.4 Énumération

L'instruction

```
for x in X:
    bloc..............
    d'instructions......
```

exécute le bloc d'instructions pour chacun des objets présents dans `X`, ce dernier pouvant être une chaîne de caractère, un tuple, une liste, un dictionnaire ou une itération de type `range(a, b)`.

4.5 Définition d'une fonction

Elle suit la syntaxe :

```
def f(x1, ... , xn):
    bloc..............
    d'instructions......
    return ...
```

En l'absence de `return`, la valeur renvoyée est égale à `None`
Un `return` peut se trouver au sein du bloc d'instructions, auquel cas son exécution interrompt immédiatement l'exécution du code au sein de la fonction.

5. Divers

5.1 Commentaires

Ils sont précédés du caractère `#`.

5.2 Utilisation simple de `print`

Envoie une chaîne de caractère en direction de la sortie standard (par défaut la console dans laquelle s'exécute votre code). À ne surtout pas confondre avec le résultat d'une fonction renvoyé par `return`.

5.3 Importation de modules

On importe l'entièreté d'un module par la syntaxe `import module` ou `import module as alias`. par exemple :

```
import numpy
```

```
import numpy as np
```

Dans le premier cas, chaque fonction du module `numpy` devra être précédée du préfixe `numpy`, dans le second cas du préfixe `np` :

```
In [1]: numpy.cos(numpy.pi)
Out[1]: -1.0
```

```
In [1]: np.cos(np.pi)
Out[1]: -1.0
```

Il est aussi possible de n'importer qu'une seule fonction d'un module donné à l'aide de la syntaxe

`from module import fonction`.

5.4 Manipulation de fichiers texte

La documentation utile de ces fonctions doit être rappelée ; tout problème relatif aux encodages est éludé.
Pour illustrer les différentes manipulations permises sur un fichier texte, nous allons prendre l'exemple d'un fichier CSV (pour *Comma-Separated Values*). Il s'agit d'un fichier texte représentant des données tabulaires sous formes de valeurs séparées par des virgules. Pour notre exemple, nous allons imaginer un fichier intitulé `naissances.csv` dont le contenu est le suivant :

```
Sexe, Prénom, Année de naissance
M, Alphonse, 1932
F, Béatrice, 1964
F, Charlotte, 1988
```

Avant de pouvoir être lu, un fichier texte doit être ouvert à l'aide de l'instruction `open` :

```
fichier = open('naissances.csv', 'r')
```

Le paramètre `'r'` indique que nous voulons accéder à ce fichier en mode lecture (*read*).
La méthode `.read` lit tout le contenu d'un fichier et renvoie une chaîne de caractère unique :

```
In [1]: fichier.read()
Out[1]: Sexe, Prénom, Année de naissance\nM, Alphonse, 1932\nF, Béatrice, 1964\nF,
....    Charlotte, 1988
```

Le caractère `\n` qui apparaît dans la réponse correspond à l'encodage d'un passage à la ligne.

la méthode `.readline()` lit une ligne du fichier et la renvoie sous forme d'une chaîne de caractères. À chaque nouvel appel de la méthode la ligne suivante est renvoyée.

```
In [2]: fichier.readline()
Out[2]: 'Sexe, Prénom, Année de naissance\n'
```

À la fin du fichier, cette méthode renvoie la chaîne de caractères vide `''`, ce qui permet de réaliser une itération conditionnelle d'un fichier. Cependant, il est plus simple dans ce cas d'utiliser une énumération du fichier :

```
In [3]: for ligne in fichier:
....         print(ligne)
M, Alphonse, 1932

F, Béatrice, 1964

F, Charlotte, 1988
```

Notez que la première ligne n'apparaît pas car elle a déjà été lue par l'instruction 2. Notez aussi que le caractère `\n` a bien été interprété comme un passage à la ligne par l'instruction `print`.

La méthode `.readlines()` lit l'entièreté du fichier et renvoie une liste dont les éléments sont les lignes de ce fichier :

```
In [1]: L = fichier.readlines()

In [2]: L
Out[2]: ['Sexe, Prénom, Année de naissance\n', 'M, Alphonse, 1932\n',
         'F, Béatrice, 1964\n', 'F, Charlotte, 1988\n']
```

Si on veut modifier le contenu d'un fichier, il faut l'ouvrir avec l'instruction open mais avec en option `'w'` (pour *write*) pour créer un nouveau fichier (dans ce cas, un fichier existant du même nom serait détruit) ou `'a'` (pour *append*) pour ajouter du texte supplémentaire à un fichier existant, puis utiliser la méthode `.write`. Par exemple, pour ajouter une ligne au fichier CSV donné en exemple il faudrait écrire :

```
In [1]: fichier = open('naissances.csv', 'a')

In [2]: fichier.write('M, Dereck, 2002\n')
```

Dans tous les cas de figure (lecture ou écriture) il est de bon ton de fermer un fichier ouvert une fois le travail terminé :

```
In [3]: fichier.close()
```

Remarque. Pour utiliser les données d'un fichier CSV une fois lues, il faut être capable de séparer ces données textuelles qui pour l'instant sont fournies sous la forme d'une chaîne de caractères. La méthode `.split()` permet de séparer les différents éléments d'une chaîne de caractères :

```
In [1]: L[3]
Out[1]: 'F, Charlotte, 1988\n'

In [2]: L[3].split(',') # l'argument sert à préciser le séparateur utilisé
Out[2]: ['F', ' Charlotte', ' 1988\n']
```

Notez qu'il resterait à convertir le dernier argument en un objet de type entier et à enlever un espace devant le prénom.

5.5 Assertion

un `assert` est une aide au débogage qui, à l'entrée d'une fonction, vérifie si certaines conditions sont vérifiées. Pour qu'une fonction ne puisse s'appliquer qu'à un entier positif (par exemple pour définir la fonction factorielle), on écrira :

```python
def fact(n):
    assert isinstance(n, int)
    assert n >= 0
    f = 1
    for k in range(2, n+1):
        f = f * k
    return f
```

```
In [1]: fact(5)
Out[1]: 120

In [2]: fact(-5)  # l'argument n'est pas positif
AssertionError

In [3]: fact(5.)  # l'argument n'est pas entier
AssertionError
```

6. Exercices

Exercice 1

On appelle *espace binaire* d'un entier naturel n toute séquence consécutive de `0` délimités par deux `1` dans la décomposition en base 2 de n. Par exemple, le nombre 529 possède deux espaces binaires de longueurs respectives 3 et 4 car $529 = (1000010001)_2$. En revanche, 32 ne possède pas d'espace binaire puisque $32 = (100000)_2$.
Rédiger une fonction `espacemax` qui prend pour argument un entier naturel et renvoie la longueur du plus grand espace binaire présent dans n s'il en existe, et la valeur 0 sinon.

Exercice 2

On appelle *rotation* d'un tableau t le fait de décaler tous les éléments d'une place vers la droite, à l'exception du dernier qui est placé en première place. Par exemple, la rotation du tableau $[1,2,3,4]$ est le tableau $[4,1,2,3]$.

a. Rédiger une fonction `rotation(t)` qui renvoie un *nouveau* tableau égal à la rotation du tableau initial.

b. Rédiger une fonction `rotation2(t)` qui *modifie* le tableau t pour le remplacer par sa rotation.

c. Rédiger une fonction `rotation3(k, t)` qui renvoie un nouveau tableau égal à k rotations du tableau t.

Exercice 3

À partir d'un tableau t d'entiers naturels, rédiger une fonction `entierManquant(t)` qui renvoie le plus petit entier naturel absent du tableau. Par exemple, pour $t = [1,3,7,6,4,1,2,0]$ cette fonction devra renvoyer l'entier 5.

Exercice 4

Un tableau non vide t de n entiers relatifs étant donné, on cherche la valeur maximale de la quantité $\Delta_k = (t[0]+\cdots+t[k]) - (t[k+1]+\cdots+t[n-1])$ lorsqu'on fait varier k dans $[\![0, n-2]\!]$.

a. Rédiger une fonction `delta(k, t)` qui calcule la quantité Δ_k et en déduire une fonction `equilibre(t)` qui résout le problème posé. Évaluer la complexité temporelle de cette dernière fonction.

b. Trouver une fonction `equilibre2(t)` qui résout ce problème en temps linéaire.

Exercice 5

Une petite grenouille se trouve face à une rivière. Initialement située sur une des deux rives (position 0), elle veut se rendre sur la rive opposée (position $x+1$) mais ne peut réaliser que des sauts d'une unité. Heureusement pour elle, des feuilles tombent sur la surface de la rivière et peuvent lui permettre de passer de feuille en feuille.

On donne un tableau t composé de n entiers représentant les feuilles qui tombent : $t[k]$ représente la position où une feuille tombe à l'instant k. L'objectif est de trouver le moment le plus précoce où la grenouille pourra passer d'une rive à l'autre, c'est-à-dire la date où toutes les positions de 1 à x seront couvertes par une feuille.
Rédiger une fonction `grenouille(x, t)` qui résout ce problème. par exemple, pour $x = 5$ et $t = [1, 3, 1, 4, 5, 3, 2, 4]$ cette fonction devra renvoyer l'entier 6.
Évaluer la complexité temporelle et spatiale de votre fonction.

Exercice 6

Un tableau contient un nombre impair d'entiers positifs. Chacun de ces entiers est présent un nombre pair de fois, à l'exception d'un seul.
Rédiger une fonction `impair(t)` qui prend pour argument un tel tableau et renvoie cet unique entier présent un nombre impair de fois. Analysez la complexité de votre algorithme.

Chapitre II
Piles

Les piles sont des structures de données linéaires dynamiques qui se distinguent par les conditions d'ajout et d'accès aux éléments : elles sont fondées sur le principe du « dernier arrivé, premier sorti » (principe LIFO, pour *last in, first out*). C'est le principe même de la pile d'assiette : c'est la dernière assiette posée sur la pile d'assiettes sales qui sera la première lavée.

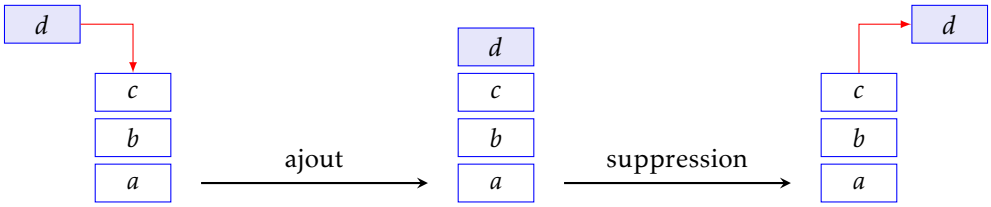


FIGURE 1 – Ajout et suppression dans une pile.

Une réalisation concrète de cette structure de données fournit, outre la structure, les fonctions suivantes :

- une fonction de création d'une pile vide ;
- deux fonctions d'ajout et de suppression *à coût constant* (nommées respectivement `push` et `pop`) ;
- une fonction vérifiant si une pile est vide.

Cette structure de données est assez pauvre, et pour cette raison n'est pas présente en tant que telle dans les langages de programmation de haut niveau comme Python. En revanche, la plupart des microprocesseurs gèrent nativement une pile, qui s'avère être une structure de donnée indispensable dans les langages machine. Nous aurons d'ailleurs l'occasion d'y revenir dans le chapitre consacré à la récursivité.

Nous allons commencer par discuter différentes manières d'implémenter en Python cette structure de données.

1. Implémentation pratique d'une pile

Pour simuler l'utilisation d'une pile en Python, nous allons avoir besoin de définir quatre fonctions :

- une fonction `pile()` qui crée une nouvelle pile vide ;
- une fonction `estVide(p)` qui prend pour argument une pile p et renvoie la valeur `True` si cette dernière est vide, et `False` sinon ;
- une procédure `push(x, p)` qui prend pour arguments un élément x et une pile p et qui *modifie* p en empilant l'élément x à son sommet ;
- une fonction `pop(p)` qui prend pour argument une pile non vide p et renvoie son sommet tout en *modifiant* p en conséquence.

On notera que les deux dernières fonctions doivent réaliser une *mutation* de la pile passée en paramètre (et renvoyer une valeur dans le cas de la seconde) et non pas créer une nouvelle pile.

Implémentation à l'aide d'une liste

Le plus simple est d'utiliser la classe `list` pour simuler une pile, car les éléments de cette classe possèdent des méthodes de mutation bien adaptées à la simulation que l'on souhaite réaliser : les méthodes append et pop.

```
def pile():
    return []

def estVide(p):
    return p == []

def push(x, p):
    p.append(x)

def pop(p):
    return p.pop()
```

On l'aura sans doute compris : avec cette simulation le sommet de la pile est situé en queue de liste ; la méthode append ajoute un élément à cette extrémité, la méthode pop supprime et renvoie la queue de la liste.

Implémentation à l'aide d'un tableau

Dans la pratique, les piles ne sont pas de hauteur infinie, et une erreur se déclenche (*Stack Overflow*) lorsque la capacité totale allouée à la pile est dépassée. Pour modéliser plus précisément ce mécanisme, on peut choisir d'implémenter une pile à l'aide d'un tableau. Puisque ceux-ci sont des structures de données statiques, nous allons devoir ajouter à la structure de données une taille maximale admissible, par exemple déclarée au moment de la création.

Avec cette simulation un indicateur est nécessaire pour connaître la case où se situe le sommet de la pile (illustration figure 2) : la première case du tableau (la case d'indice 0) désignera l'indice q de la première case disponible pour accueillir un nouvel élément dans la pile.

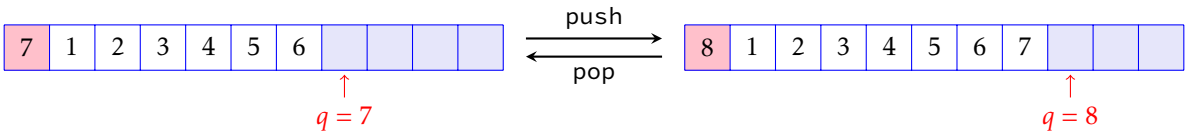


Figure 2 – Une pile implémentée à l'aide d'un tableau, l'ajout et le retrait d'un élément.

Avec cette implémentation une erreur doit se produire lorsqu'on cherche à empiler un nouvel élément sur une pile pleine (ce qu'on appelle un débordement de pile, *Stack Overflow* en anglais).

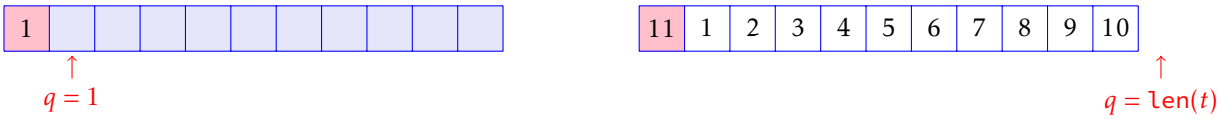


Figure 3 – Une pile vide et une pile pleine

```
def pile(n=256):        # n désigne la taille maximale allouée à la pile
    p = [None] * (n+1)
    p[0] = 1
    return p

def estVide(p):
    return p[0] == 1

def push(x, p):
    if p[0] == len(p): raise RuntimeError('Stack Overflow')
    p[p[0]] = x
    p[0] += 1

def pop(p):
    p[0] -= 1
    return p[p[0]]
```

Il faut bien distinguer la définition d'une structure de donnée abstraite (les piles) de son implémentation concrète, qui peut prendre différentes formes et qui reste en général cachée à l'utilisateur, qui ne peut interagir avec la structure de donnée que par l'intermédiaire des méthodes ou fonctions qui lui sont associées (push et pop ici).

2. Évaluation d'une expression postfixe

La notation *postfixe* d'une expression algébrique consiste à placer les opérateurs après son ou ses opérandes. Par exemple, l'addition de a et de b sera écrite « $a\ b\ +$ » en notation postfixe, la racine carrée de a sera écrite « $a\ \sqrt{}$ ». L'intérêt majeur de cette notation est qu'une expression postfixe n'est jamais ambiguë : alors que expression infixe « $1\ +\ 2\ \times\ 3$ » peut avoir deux significations : « $(1\ +\ 2)\ \times\ 3$ » ou « $1\ +\ (2\ \times\ 3)$ », ce n'est jamais le cas d'une expression postfixe, ce qui rend l'usage des parenthèses superflu : « $1\ 2\ +\ 3\ \times$ » ne peut être compris que de cette façon : « $(1\ 2\ +)\ 3\ \times$ » et « $1\ 2\ 3\ \times\ +$ » de cette façon : « $1\ (2\ 3\ \times)\ +$ ».

Nous allons montrer comment, à l'aide d'une pile, on peut évaluer très simplement une expression algébrique postfixe.

Les expressions algébriques seront ici représentée par les listes qui pourront contenir des nombres (de type `int` ou `float`) ou des fonctions à un ou deux arguments.

Ainsi, l'expression $\dfrac{1+2\sqrt{3}}{4}$ sera représentée par la liste `[1, 2, 3, sqrt, multiply, add, 4, divide]`.

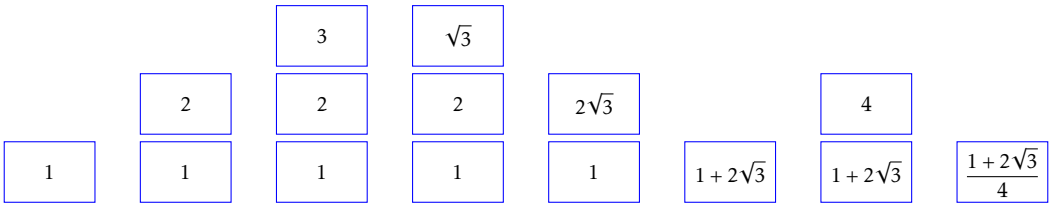


FIGURE 4 – Évolution de la pile associée à l'expression postfixe $1\ 2\ 3\ \sqrt{}\ \times\ +\ 4\ \div$

On suppose données deux listes répertoriant pour l'une les opérateurs unaires autorisés, pour l'autre les opérateurs binaires. Ces deux listes peuvent par exemple être crées en suivant le modèle :

```
from numpy import sqrt, exp, log, add, subtract, multiply, divide

op_uni = [sqrt, exp, log]
op_bin = [add, subtract, multiply, divide]
```

L'évaluation d'une expression postfixe consiste à utiliser une pile initialement vide et à parcourir les éléments de la liste représentant l'expression à évaluer en appliquant les règles suivantes :

- si l'élément est un nombre, il est empilé ;
- si l'élément est un opérateur unaire, le sommet de la pile est dépilé, l'opérateur lui est appliqué et le résultat ré-empilé ;
- si l'élément est un opérateur binaire, deux éléments de la pile sont dépilés, l'opérateur leur est appliqué et le résultat ré-empilé.

Si l'expression postfixe est correcte sur le plan syntaxique (et mathématique), à la fin du traitement de la liste la pile ne contient plus qu'un seul élément égal au résultat de l'évaluation de l'expression.

EXERCICE 1

Rédiger une fonction qui évalue une expression postfixe donnée sous forme de liste. On supposera l'expression passée en argument syntaxiquement correcte.

Un exemple d'utilisation de cette fonction :

```
In [1]: evalue([1, 2, 3, sqrt, multiply, add, 4, divide])
```

```
Out[1]: 1.1160254037844386
```

L'évolution de la pile associée à cet exemple est illustrée figure 4.

3. Parcours d'un graphe

Un *graphe orienté* est constitué d'un ensemble S de *sommets* et d'un ensemble A d'*arcs*, où A est une partie de $S \times S$.

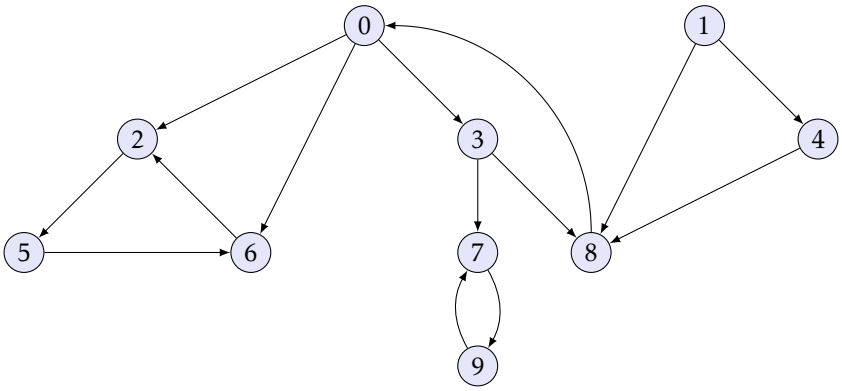


FIGURE 5 – Représentation graphique d'un graphe orienté.

Pour représenter un graphe G = (S, A), on peut utiliser une *matrice d'adjacences* (mais ce n'est pas la seule solution) : on suppose les sommets numérotés arbitrairement : $S = \{0, 1, \dots, n-1\}$ et on représente G par une matrice [1] $M = (a_{ij})$ de taille $n \times n$ pour laquelle

$$a_{ij} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases}$$

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

FIGURE 6 – Représentations du graphe de la figure 5 par matrice d'adjacences.

Bien évidemment, le graphe sera dit *non orienté* lorsque sa matrice d'adjacences est symétrique.

Parcours en profondeur

Parcourir un graphe, c'est explorer les sommets de ce dernier de proche en proche (c'est-à-dire en suivant les arcs) à partir d'un sommet initial. Il existe plusieurs manières de parcourir un graphe ; comme son nom l'indique, le *parcours en profondeur* consiste à descendre le plus « profondément » possible dans le graphe chaque

1. Attention : contrairement aux matrices mathématiques, les indices de lignes et de colonnes sont indexées entre 0 et $n-1$ pour respecter les conventions PYTHON concernant l'indexation des tableaux.

fois que c'est possible. Lors d'un parcours en profondeur, les arcs sont donc explorés à partir du sommet v découvert le plus récemment et dont on a pas encore exploré tous les arcs qui en partent. Lorsque tous les arcs issus de v ont été explorés, on « revient en arrière » pour explorer les arcs qui partent du sommet à partir duquel v a été découvert. Ce processus se répète jusqu'à ce que tous les sommets accessibles à partir du sommet origine initial aient été découverts.

■ Description sur un exemple

Considérons le parcours en profondeur du graphe représenté figure 5 à partir du sommet 0. À partir de ce sommet, trois autres sommets sont accessibles : les sommets 2, 3 et 6.
L'exploration commence par le sommet 2 ; à partir de ce sommet seul le sommet 5 est accessible donc le parcours se poursuit par ce sommet. Pour les mêmes raisons c'est le sommet 6 qui est exploré ensuite.
À partir du sommet 6 le sommet 2 est accessible, mais ce dernier a déjà été exploré. On revient donc en arrière vers le sommet 5, qui n'a plus de sommet accessible à découvrir. On recule donc encore vers le sommet 2, puis pour les mêmes raisons vers le sommet 0.
À cette étape de l'exploration, sur les trois sommets accessibles à partir du sommet 0, deux ont étés explorés : les sommets 2 et 6. L'exploration ne peut donc se poursuivre que vers le sommet 3, qui conduira dans l'ordre, et pour les mêmes raisons que précédemment, à l'exploration des sommets 7, puis 9 et enfin 8.

On peut le deviner à la lecture de la description ci-dessus, l'exploration en profondeur utilise une pile pour y entasser les sommets en cours d'exploration : un sommet fraichement découvert est empilé, et il ne sera dépilé qu'au moment où tous ses voisins auront été complètement explorés. En outre, il sera nécessaire d'utiliser une structure de données pour y ranger les sommets découverts.

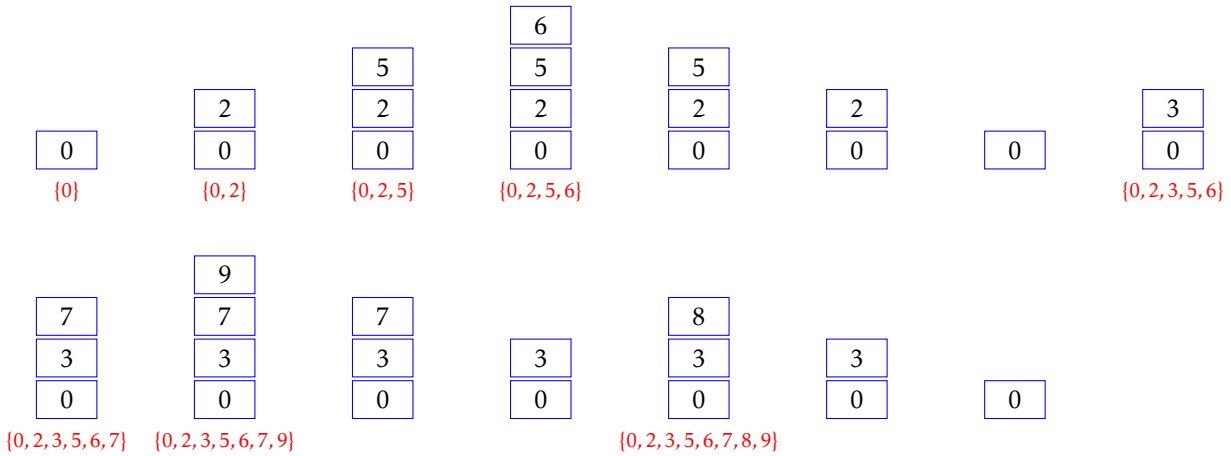


FIGURE 7 – L'évolution de la pile lors d'un parcours en profondeur. Sous la pile, l'évolution de la liste des sommets découverts.

■ Implémentation du parcours en profondeur

EXERCICE 2

Rédiger une fonction python `parcoursEnProfondeur(M, i)` qui prend pour argument un graphe représenté par sa matrice d'adjacences M et un entier i, et qui réalise un parcours en profondeur à partir du sommet i. On affichera chaque sommet au moment de sa découverte (lorsqu'il entre dans la pile) puis à la fin de son traitement (lorsqu'il en sort).

4. Exercices

Dans tous ces exercices, on supposera définies les quatre fonctions `pile`, `estVide`, `push` et `pop` décrites page 1 de ce document.

Exercice 3

Rédiger une fonction `trier(p)` qui prend en argument une pile `p` contenant des nombres entiers et qui modifie l'ordre de ses éléments de sorte qu'en fin de traitement les nombres pairs soient situés sous les nombres impairs.
Indication. Créer et utiliser une ou plusieurs piles auxiliaires.

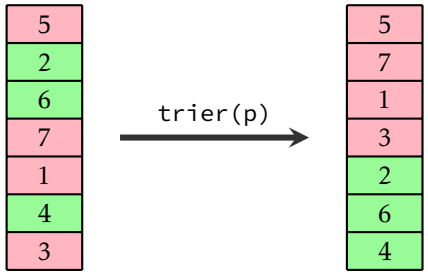


Figure 8 – Un exemple d'utilisation de la fonction `trier`.

Exercice 4

a. À l'aide d'une pile auxiliaire rédiger une fonction `insere(x, p)` qui prend pour arguments un entier x et une pile p formée d'entiers triés par ordre croissant, et qui insère x au sein de p en préservant l'ordre relatif des éléments.

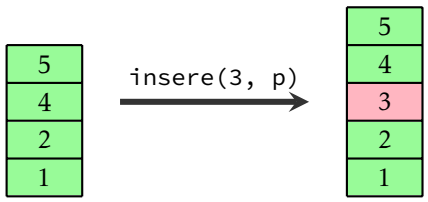


Figure 9 – Un exemple d'utilisation de la fonction `insere`.

b. En déduire une fonction `tri(p)` qui prend pour arguments une pile d'entiers et qui renvoie une nouvelle pile contenant les mêmes éléments mais triés par ordre croissant. Quelle est la complexité temporelle de cette fonction ?

Exercice 5

Une chaîne de caractères S est dite *bien parenthésée* lorsque l'une de ces conditions est vérifiée :

- S est vide ;
- S est de la forme (U) ou [U] où U est une chaîne de caractères bien parenthésée ;
- S est de la forme UV où U et V sont des chaînes de caractères bien parenthésées.

Par exemple, `'[()[]]'` est une expression bien parenthésée tandis que `'([)()]'` ne l'est pas.
Rédiger une fonction `parenthese(S)` qui prend pour argument une chaîne de caractères et renvoie `True` lorsque celle-ci est bien parenthésée, et `False` sinon.

Exercice 6

On dit qu'une permutation $(a_1 \quad a_2 \quad \cdots \quad a_n)$ de $(1 \quad 2 \quad \cdots \quad n)$ peut être engendrée par une pile lorsque il est possible, à partir de la séquence d'entrée $(1 \quad 2 \quad \cdots \quad n)$ et d'une pile (initialement vide), de produire la séquence de sortie $(a_1 \quad a_2 \quad \cdots \quad a_n)$ en utilisant les opérations suivantes :

- empiler l'élément suivant de la séquence d'entrée ;
- ou dépiler le sommet de la pile et l'imprimer à l'écran.

Par exemple, si E et D désignent respectivement chacune des deux opérations permises, la permutation $(2 \quad 3 \quad 1)$ est engendrée par la suite d'opérations : EEDEDD.

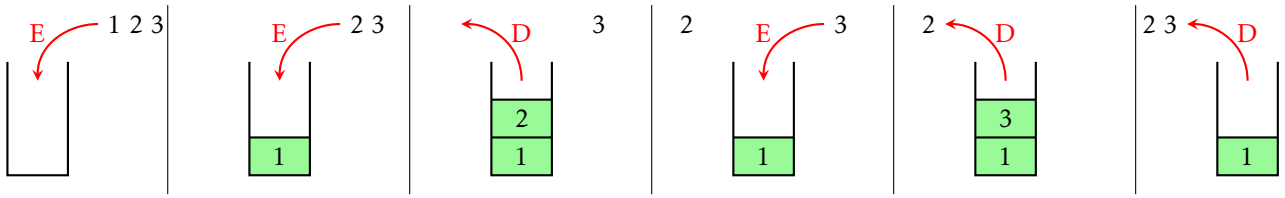


Figure 10 – La permutation $(2 \quad 3 \quad 1)$ est engendrable par une pile.

a. Parmi les deux permutations suivantes, laquelle peut être engendrée par une pile ?

$(3 \quad 5 \quad 7 \quad 6 \quad 8 \quad 4 \quad 9 \quad 2 \quad 10 \quad 1)$ $\qquad$ $(4 \quad 6 \quad 5 \quad 3 \quad 8 \quad 7 \quad 1 \quad 9 \quad 10 \quad 2)$

b. Rédiger une fonction `genere(s)` qui prend pour argument une permutation s représentée par une liste en renvoie un booléen `True` ou `False` suivant que s est engendrable par une pile ou non.

Chapitre III
Récursivité

To understand what recursion is,
you must first understand recursion.

1. Algorithmes récursifs

1.1 Un premier exemple

Considérons le problème suivant : rédiger une fonction `star1(n)` qui prend pour argument un entier naturel $n \in \mathbb{N}$ et affiche dans la console n étoiles sur la première ligne, $n-1$ sur la seconde ligne, …, et enfin une étoile sur la n^{e} ligne.

```
In [1]: star1(5)
*****
****
***
**
*
```

```
In [2]: star2(5)
*
**
***
****
*****
```

```
In [3]: star3(4)
****
***
**
*
**
***
****
```

FIGURE 1 – Un exemple d'utilisation des fonctions `star1`, `star2` et `star3`.

Il est bien entendu très simple de rédiger une version itérative de cette fonction, mais on peut aussi observer qu'il ne s'agit finalement, lorsque n est strictement positif, que de tracer n étoiles sur la première ligne puis d'appliquer `star1(n-1)`.Traduit en Python, cela donne :

```
def star1(n):
    if n > 0:
        print(n * '*')
        star1(n - 1)
```

Une telle fonction est dite *récursive* car elle est utilisée *au sein même de sa définition*. Pour qu'une telle définition soit valide, il est nécessaire que deux conditions soient vérifiées :

- il doit y avoir une *condition d'arrêt*, c'est-à-dire au moins une valeur de n pour laquelle aucun appel récursif n'est effectué ;
- il ne doit pas y avoir de suite *infinie* d'appels récursifs.

Dans l'exemple de la fonction `star1`, la condition d'arrêt correspond aux entiers négatifs ou nuls (il ne se passe rien), et pour chaque entier $n \geqslant 1$, n appels récursifs sont réalisés (un nombre fini, donc).

EXERCICE 1

a. Rédiger une fonction récursive `star2(n)` qui prend pour argument un entier n et qui affiche une étoile sur la première ligne, deux sur la seconde, … et enfin n sur la n^{e} ligne (voir un exemple figure 1).

b. Rédiger une fonction récursive `star3(n)` sur le modèle de la fonction illustrée figure 1.

1.2 L'algorithme d'exponentiation rapide

Mais au fond, à quoi bon écrire un algorithme récursif ? Somme toute, les trois fonctions `star1`, `star2` et `star3` sont très simples à écrire de manière itérative. Il existe cependant des algorithmes *qui s'expriment naturellement de manière récursive*, et ce sont avant tout ces algorithmes qui motivent l'utilisation de la programmation récursive.

C'est le cas par exemple de l'*algorithme d'exponentiation rapide*. Cet algorithme calcule x^n en exploitant les formules :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x & \text{si } n = 1 \\ (x^2)^p & \text{si } n = 2p \text{ est pair} \\ x(x^2)^p & \text{si } n = 2p+1 \text{ est impair} \end{cases}$$

Compte tenu de sa formulation, il est naturel d'utiliser la programmation récursive pour le traduire :

```
def power(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    y = power(x * x, n // 2)
    if n % 2 == 0:
        return y
    else:
        return x * y
```

Il y a ici deux conditions d'arrêts : $n = 0$ et $n = 1$. Montrons maintenant par récurrence sur n que pour tout $n \in \mathbb{N}$, seul un nombre fini d'appels récursif est réalisé.

- Si $n = 0$ ou $n = 1$ il n'y a pas d'appel récursif.
- Si $n \geqslant 2$, supposons le résultat acquis jusqu'au rang $n-1$. Sachant que $p = \lfloor n/2 \rfloor < n$, on peut applique l'hypothèse de récurrence : le calcul de $(x^2)^p$ avec $p = \lfloor n/2 \rfloor$ n'utilise qu'un nombre fini d'appels récursifs ; il en est donc de même du calcul de x^n. La récurrence se propage.

Remarque. Si $C(n)$ désigne le nombre d'appels récursifs nécessaires pour calculer x^n, nous venons de montrer que :

$$C(0) = C(1) = 0 \qquad \text{et} \qquad C(n) = 1 + C(\lfloor n/2 \rfloor) \quad \text{pour } n \geqslant 2$$

En utilisant la décomposition de n en base 2 : $n = (b_p \cdots b_1 b_0)_2$ on trouve que $C((b_p \cdots b_1 b_0)_2) = 1 + C((b_p \cdots b_1)_2)$ puis aisément : $C((b_p \cdots b_1 b_0)_2) = p + C(1) = p$ donc $C(n) = \lfloor \log_2 n \rfloor$. Sachant qu'à part les appels récursifs, les autres opérations (des produits et des divisions) sont de complexité constante, on en déduit que cet algorithme a une complexité temporelle en $O(\log n)$, bien meilleure donc qu'une complexité linéaire que produirait l'algorithme itératif naïf.

1.3 Les tours de Hanoï

Il n'existe pas de réponse définitive à la question de savoir si un algorithme récursif est préférable à un algorithme itératif. Il a été prouvé que ces deux paradigmes de programmation sont équivalents ; autrement dit, tout algorithme itératif possède une version récursive, et réciproquement. Un algorithme récursif est aussi performant qu'un algorithme itératif pour peu que le programmeur ait évité les quelques écueils qui seront étudiés plus loin dans ce cours et que le compilateur ou l'interprète de commande gère convenablement la récursivité. Le choix du langage peut aussi avoir son importance : un langage fonctionnel tel OCaml est conçu pour exploiter la récursivité et le programmeur est naturellement amené à choisir la version récursive de l'algorithme qu'il souhaite écrire. À l'inverse, Python, même s'il l'autorise, ne favorise pas l'écriture récursive (limitation basse par défaut du nombre d'appels récursifs, pas ou peu d'optimisation des algorithmes récursifs).

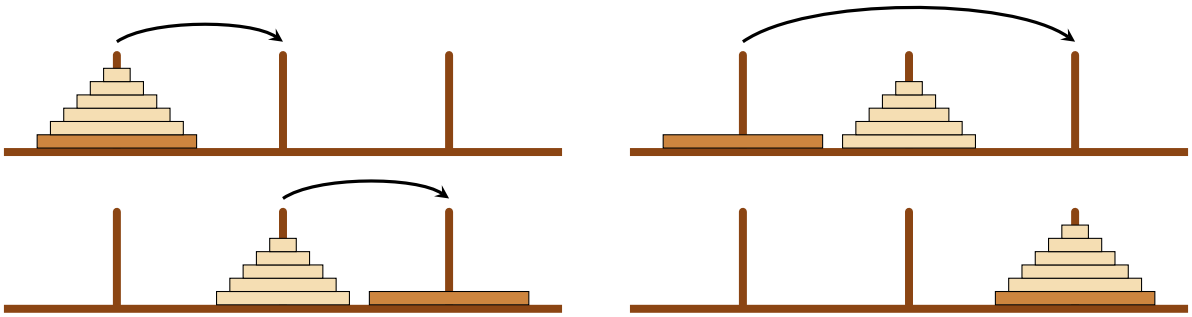
Figure 2 – Le puzzle dans sa configuration initiale.

Parmi les problèmes dont la résolution récursive est beaucoup plus simple que la résolution itérative, l'un des plus emblématiques est sans conteste le problème des tours de Hanoï inventé par le mathématicien français

Édouard Lucas. Ce jeu mathématique est constitué de trois tiges sur lesquelles sont enfilés n disques de diamètres différents. Au début du jeu, ces disques sont tous positionnés sur la première tige (du plus grand au plus petit) et l'objectif est de déplacer tous ces disques sur la troisième tige, en respectant les règles suivantes :

- un seul disque peut être déplacé à la fois ;
- on ne peut jamais poser un disque sur un disque de diamètre inférieur.

Maintenant, raisonnons par récurrence : pour pouvoir déplacer le dernier disque, il est nécessaire de déplacer les $n-1$ disques qui le couvrent sur la tige centrale. Une fois ces déplacements effectués, nous pouvons déplacer le dernier disque sur la troisième tige. Il reste alors à déplacer les $n-1$ autres disques vers la troisième tige.



Tout est dit : pour pouvoir déplacer n disques de la tige 1 vers la tige 3 il suffit de savoir déplacer $n-1$ disques de la tige 1 vers la tige 2 puis de la tige 2 vers la tige 3. Autrement dit, il suffit de généraliser le problème de manière à décrire le déplacement de n disques de la tige i à la tige k en utilisant la tige j comme pivot. Ceci conduit à la définition suivante :

```
def hanoi(n, i=1, j=2, k=3):
    if n == 0:
        return None
    hanoi(n-1, i, k, j)
    print("Déplacer le disque {} de la tige {} vers la tige {}.".format(n, i, k))
    hanoi(n-1, j, i, k)
```

On trouvera figure 3 un exemple de résolution pour $n = 4$.

```
In [1]: hanoi(4)
Déplacer le disque 1 de la tige 1 vers la tige 2.
Déplacer le disque 2 de la tige 1 vers la tige 3.
Déplacer le disque 1 de la tige 2 vers la tige 3.
Déplacer le disque 3 de la tige 1 vers la tige 2.
Déplacer le disque 1 de la tige 3 vers la tige 1.
Déplacer le disque 2 de la tige 3 vers la tige 2.
Déplacer le disque 1 de la tige 1 vers la tige 2.
Déplacer le disque 4 de la tige 1 vers la tige 3.
Déplacer le disque 1 de la tige 2 vers la tige 3.
Déplacer le disque 2 de la tige 2 vers la tige 1.
Déplacer le disque 1 de la tige 3 vers la tige 1.
Déplacer le disque 3 de la tige 2 vers la tige 3.
Déplacer le disque 1 de la tige 1 vers la tige 2.
Déplacer le disque 2 de la tige 1 vers la tige 3.
Déplacer le disque 1 de la tige 2 vers la tige 3.
```

Figure 3 – Une solution du problème des tours de Hanoï avec quatre disques.

Bien qu'il soit tentant de se demander suivant quelle logique les disques de tailles inférieures se déplacent (autrement dit, que se passe-t'il lorsqu'on déroule les différents appels récursifs), on notera bien que ce n'est pas nécessaire. Notre unique tâche est de réduire le problème à un ou plusieurs sous-problèmes identiques et à veiller à respecter les deux conditions pour qu'un algorithme récursif soit valide : existence d'une condition

d'arrêt (ici le cas $n = 0$) et nombre fini d'appels récursifs (il est ici facile de constater que le nombre d'appels récursifs est égal à $2^{n+1} - 2$).

Exercice 2

Résoudre le problème des tours de Hanoï en s'imposant une contrainte supplémentaire : tout mouvement entre les tiges 1 et 3 est interdit (tous les mouvements doivent donc aboutir ou débuter par la tige 2).
Déterminer le nombre de mouvements nécessaires dans ce cas.

1.4 Le tri fusion

Le tri fusion (*merge sort*) est un des premiers algorithmes inventés pour trier un tableau car (selon Donald Knuth) il aurait été proposé par John von Neuman dès 1945 ; il constitue un parfait exemple d'algorithme naturellement récursif.
Son fonctionnement est le suivant :

- diviser le tableau à trier en deux parties sensiblement égales ;
- trier récursivement chacune de ces deux parties ;
- fusionner les deux parties triées dans un seul tableau trié.

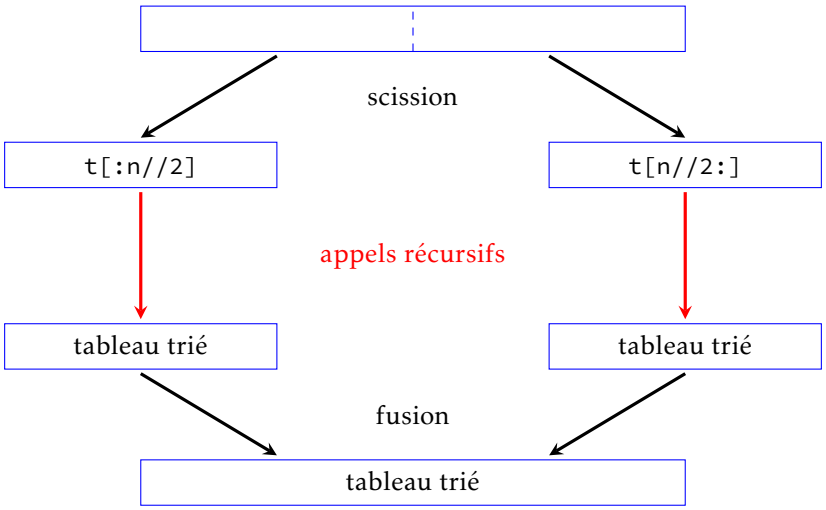


Figure 4 – Une illustration du tri fusion.

Dans la mise en œuvre figure 5, la fonction `merge` prend pour arguments deux tableaux supposés triés par ordre croissant, et renvoie un nouveau tableau, résultat de la fusion des deux tableaux passés en argument. Cette fonction est séparée du corps principal de l'algorithme pour accroitre la lisibilité de la structure récursive.

Quelle est la complexité temporelle de cette fonction ? La fonction `merge` est à l'évidence de coût linéaire vis-à-vis de la somme des longueurs des deux tableaux passés en paramètre. Si $C(n)$ désigne la complexité temporelle du tri d'un tableau de longueur n, on dispose donc de la relation :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + O(n).$$

Ce type de relation de récurrence est typique des méthodes dites « diviser pour régner » ; dans le cas présent on prouve que cette relation implique que $C(n) = O(n \log n)$. Nous apprendrons dans le chapitre suivant qu'il n'est pas possible de faire mieux dans le cadre des algorithmes de tri par comparaison.

1.5 Récursivité et pile d'exécution d'un programme

Un programme n'étant qu'un flux d'instructions exécutées séquentiellement, son exécution peut être représentée par le parcours d'un chemin ayant une origine et une extrémité. Lors de l'appel d'une fonction, ce flux est

```python
def merge(a, b):
    p, q = len(a), len(b)
    c = [None] * (p + q)
    i = j = 0
    for k in range(p+q):
        if j >= q:
            c[k] = a[i]
            i += 1
        elif i >= p:
            c[k] = b[j]
            j += 1
        elif a[i] < b[j]:
            c[k] = a[i]
            i += 1
        else:
            c[k] = b[j]
            j += 1
    return c
```

```python
def mergesort(t):
    n = len(t)
    if n < 2:
        return t
    a = mergesort(t[:n//2])
    b = mergesort(t[n//2:])
    return merge(a, b)
```

FIGURE 5 – Implémentation du tri fusion.

interrompu le temps de l'exécution de cette fonction, avant de reprendre à l'endroit où le programme s'est arrêté (voir figure 6).
Au moment où débute cette bifurcation, il est nécessaire que le processeur sauvegarde un certain nombre d'informations : adresse de retour, état des paramètres et des variables, etc. Toutes ces données forment ce qu'on appelle le *contexte* du programme, et elles sont stockées dans une pile qu'on appelle la *pile d'exécution* [2]. À la fin de l'exécution de la fonction, le contexte est sorti de la pile pour être rétabli et permettre la poursuite de l'exécution du programme.
L'utilisation d"une pile se comprend : lorsque plusieurs appels à des fonctions différentes sont emboités, c'est le contexte de la dernière fonction à avoir été appelée qui doit être rétabli en premier ; c'est bien là le principe des piles LIFO (*Last In, First Out*).

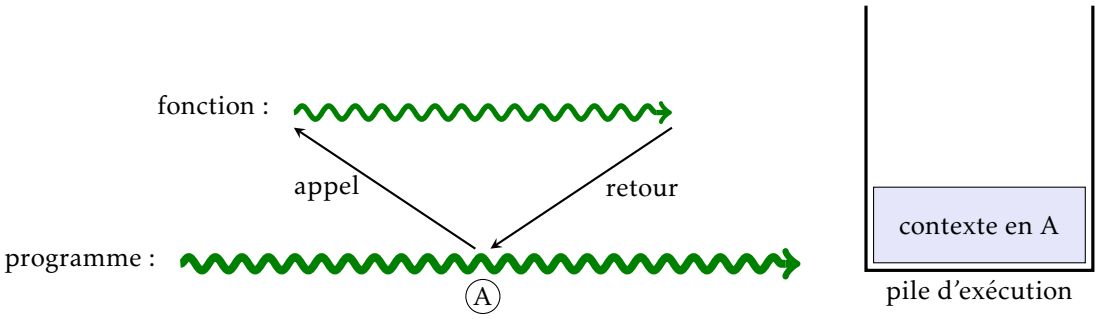


FIGURE 6 – L'exécution d'une fonction au sein d'un programme.

Lors de l'exécution d'une fonction récursive, chaque appel récursif conduit au moment où il se produit à un empilement du contexte dans la pile d'exécution. Lorsqu'au bout de n appels se produit la condition d'arrêt, les différents contextes sont progressivement dépilés pour poursuivre l'exécution de la fonction. La figure 7 illustre les trois premiers appels récursifs de la fonction récursive `power` avec pour paramètres $x = 3$ et $n = 41$ (pour des raisons de lisibilité, seules les valeurs de ces deux paramètres sont présentées dans le contexte).
Il est donc important de prendre conscience qu'une fonction récursive va s'accompagner d'un coût spatial qui va croître avec le nombre d'appels récursifs (en général linéairement, mais ce n'est pas une règle générale, tout dépend du contenu du contexte) ; ce coût ne doit pas être oublié lorsqu'on fait le bilan de la complexité spatiale d'une fonction récursive.

2. Suivant les langages et leurs implémentations, il peut y avoir une pile d'exécution par fonction ou une seule pile globale pour tout le programme.

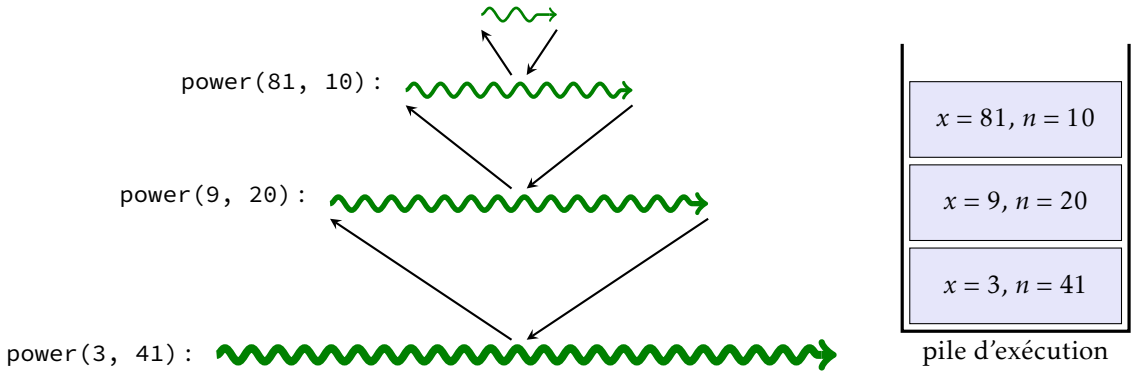


FIGURE 7 – Calcul récursif de 3^{41} par exponentiation rapide.

Remarque. Que ce passe-t-il lorsqu'un algorithme ne se termine pas, soit parce qu'on a oublié la condition d'arrêt, soit parce qu'une entrée génère un nombre infini d'appels récursifs ? La pile d'exécution croît indéfiniment, jusqu'à dépasser la taille limite que peut fournir la mémoire. Ceci pouvant conduire à un comportement erratique de l'ordinateur, Python possède un garde-fou pour empêcher cela : une variable système qui limite le nombre d'appels récursifs (sa valeur dépend du système). C'est pourquoi un algorithme correct peut parfois conduire à une erreur si le nombre maximal d'appels récursifs est dépassé.
Pour illustrer ceci, j'ai délibérément fortement baissé la valeur de cette limite dans l'exemple qui suit :

```
In [1]: star1(10)
**********
*********
********
*******
******
*****
****
RecursionError: maximum recursion depth exceeded while calling a Python object
```

1.6 Trace d'une fonction

Dans les langages de programmation de haut niveau, les spécificités de la pile d'exécution sont cachées au programmeur. Ce dernier a uniquement accès aux appels de fonctions et aux paramètres associés, et non au contenu de la pile elle-même. On peut cependant obtenir une représentation de la pile d'exécution appelée la *trace d'appels* en faisant apparaître les paramètres d'entrées et les valeurs de retour de chaque appel, ce qui peut faciliter entre autre le débogage d'une fonction.

On trouvera figure 8 la trace des appels récursifs de la fonction `power` pour le couple $(x = 3, n = 41)$ et de la fonction `mergesort` pour le tableau $[6, 2, 4, 3, 5, 1]$. Une flèche orientée vers la gauche dénote un appel de la fonction (donc un empilement dans la pile d'exécution) ; une flèche orientée vers la droite un dépilement de celle-ci.

2. Les écueils de la programmation récursive

Nous allons maintenant aborder les principaux pièges qui guettent le programmeur qui souhaite écrire une fonction récursive, non pas pour vous dissuader d'en écrire, mais pour vous permettre de les éviter.

```
power <- (3, 41)
 power <- (9, 20)
  power <- (81, 10)
   power <- (6561, 5)
    power <- (43046721, 2)
     power <- (1853020188851841, 1)
     power -> 1853020188851841
    power -> 1853020188851841
   power -> 12157665459056928801
  power -> 12157665459056928801
 power -> 12157665459056928801
power -> 36472996377170786403
```

```
mergesort <- [6, 2, 4, 3, 5, 1]
 mergesort <- [6, 2, 4]
  mergesort <- [6]
  mergesort -> [6]
  mergesort <- [2, 4]
   mergesort <- [2]
   mergesort -> [2]
   mergesort <- [4]
   mergesort -> [4]
  mergesort -> [2, 4]
 mergesort -> [2, 4, 6]
 mergesort <- [3, 5, 1]
  mergesort <- [3]
  mergesort -> [3]
  mergesort <- [5, 1]
   mergesort <- [5]
   mergesort -> [5]
   mergesort <- [1]
   mergesort -> [1]
  mergesort -> [1, 5]
 mergesort -> [1, 3, 5]
mergesort -> [1, 2, 3, 4, 5, 6]
```

FIGURE 8 – Un exemple de trace des fonctions `power` et `mergesort`.

2.1 Attention aux coûts cachés

Les principales difficultés sont liées à de mauvaises évaluations des coûts tant temporels que spatiaux. Partons d'un exemple connu, l'algorithme de recherche dichotomique. Étant donné un tableau t de taille n contenant une liste *triée* par ordre croissant d'éléments et un élément x, on cherche à savoir si x se trouve dans t. La démarche est connue, et vous l'avez étudié en première année : il s'agit de comparer x à t_k avec $k = \lfloor n/2 \rfloor$ puis :

- si $x = t_k$, la recherche est terminée ;
- si $x < t_k$ la recherche se poursuit dans $t[0 \dots k-1]$;
- si $x > t_k$ la recherche se poursuit dans $t[k+1 \dots n-1]$.

Cette description se prête à merveille à une programmation de nature récursive, et il est tentant d'écrire le code que l'on trouve figure 9.

```
def dicho(x, t):
    if len(t) == 0:
        return False
    k = len(t) // 2
    if x == t[k]:
        return True
    elif x < t[k]:
        return dicho(x, t[:k])
    else:
        return dicho(x, t[k+1:])
```

FIGURE 9 – Une première version de la recherche dichotomique dans un tableau.

Bien que particulièrement limpide, ce code se révèle mauvais car le calcul de `t[:k]` et de `t[k+1:]` est de coût linéaire, tant temporel que spatial (car on procède à une recopie de la moitié de tableau dans un autre espace mémoire). La relation de récurrence qui régit la complexité est donc de la forme : $C(n) = C(n/2) + O(n)$, ce qui conduit à $C(n) = O(n)$. Cet algorithme est de complexité linéaire, donc du même ordre que l'algorithme de recherche dans un tableau non trié, et bien loin du coût logarithmique de l'algorithme itératif étudié en première année.

Il faut donc écrire une version récursive de l'algorithme qui ne procède à aucune recopie de tableau, et pour ce faire on doit généraliser le problème en écrivant une fonction qui recherche x dans la partie du tableau $t[i \dots j-1]$ en comparant x à t_k avec $k = \lfloor (i+j)/2 \rfloor$:

- si $x = t_k$, la recherche est terminée ;
- si $x < t_k$ la recherche se poursuit dans $t[i \dots k-1]$;
- si $x > t_k$ la recherche se poursuit dans $t[k+1 \dots j-1]$.

Le code de la version récursive correcte se trouve figure 10.

```
def dicho_aux(x, t, i, j):
    if j <= i:
        return False
    k = (i + j) // 2
    if x == t[k]:
        return True
    elif x < t[k]:
        return dicho_aux(x, t, i, k)
    else:
        return dicho_aux(x, t, k+1, j)

def dicho(x, t):
    return dicho_aux(x, t, 0, len(t))
```

Figure 10 – Une version récursive correcte de la recherche dichotomique.

Cette fois toutes les opérations qui précèdent les appels récursifs sont de coût constant donc la complexité vérifie une relation du type $C(n) = C(n/2) + O(1)$ qui donne $C(n) = O(\log n)$ (coût tant temporel que spatial).

Exercice 3

Un *palindrome* est un mot ou une phrase qui peut se lire indifféremment de la gauche vers la droite, tels *kayak* ou *ressasser*, ou encore, si on ne tiens pas compte des espaces et de la ponctuation, *Tu l'as trop écrasé, César, ce Port-Salut !*.
Rédiger une fonction récursive `palindrome(m)` qui prend en argument un mot m et renvoie `True` si ce mot est un palindrome, `False` sinon.

2.2 Appels récursifs multiples

Une difficulté plus sérieuse se présente lors d'appels récursifs multiples qui sont en interaction. L'exemple emblématique de ce problème concerne le calcul du n^e terme f_n de la suite de Fibonacci. La définition récursive naturelle est la suivante :

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

Malheureusement, les performances de cette fonction se dégradent extrêmement rapidement (voir figure 11) et au lieu d'un coût linéaire attendu on obtient un coût temporel d'apparence exponentiel.

Un début d'explication est donnée lorsqu'on trace cette fonction pour calculer f_6 (voir figure 12).
On constate que f_0 est calculé cinq fois, f_1 huit fois, f_2 cinq fois, f_3 trois fois, f_4 deux fois, f_5 une fois et f_6 une fois, ce qui fait (en tout) 25 appels à la fonction `fib` au lieu des 7 appels attendus.
Par exemple, f_4 est calculé une première fois pour calculer $f_5 = f_4 + f_3$ et une deuxième fois pour calculer $f_6 = f_5 + f_4$, f_3 va être calculé une fois pour chacun des deux calculs de f_4 et une fois pour calculer f_5 dont trois fois en tout, etc.

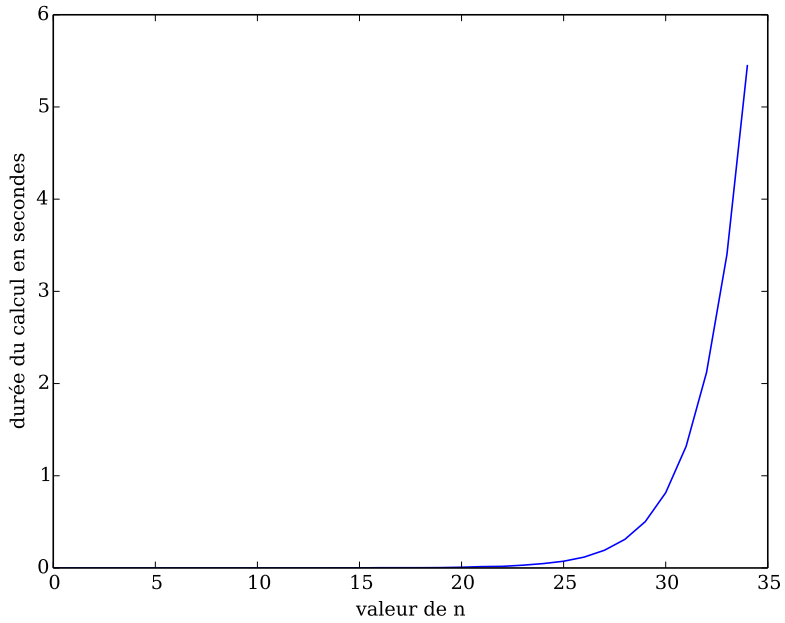


FIGURE 11 – Temps d'exécution en secondes pour calculer f_n par l'algorithme récursif.

Précisons les choses en calculant le nombre a_n d'appels à la fonction `fib` pour calculer f_n. On dispose des relations :

$$a_0 = a_1 = 1 \qquad \text{et} \qquad \forall n \geqslant 2,\ a_n = a_{n-1} + a_{n-2} + 1.$$

Cette suite se résout en $a_n = 2f_{n+1} - 1$. Sachant que $f_n \sim \frac{1}{\sqrt{5}}\phi^n$ (où ϕ est le nombre d'or) on obtient $a_n = O(\phi^n)$; le coût de cette fonction, tant temporel que spatial, est exponentiel !

Une solution : la mémoïsation

Il existe plusieurs solutions pour obtenir un coût plus raisonnable. Celle que nous allons adopter consiste à mémoriser le résultat des calculs une fois qu'ils auront été calculés, de manière à ne pas refaire deux fois le même calcul. La structure de données qui s'impose ici est la structure de *dictionnaire*, qui est constituée de paires associant une clef à une valeur. Dans le cas qui nous intéresse, la clef est l'entier n et la valeur, l'entier f_n.

Nous ne détaillerons pas l'implémentation des dictionnaires qui est complexe, et nous admettrons que le coût de l'ajout comme de la lecture dans un dictionnaire est en moyenne constant.

Nous pouvons maintenant réécrire la fonction `fib`, en la faisant précéder de la création d'un dictionnaire. Ensuite, le calcul de f_n se déroulera de la façon suivante : on commence par regarder si n est déjà présent dans le dictionnaire, et si ce n'est pas le cas (et uniquement dans ce cas) on calcule f_n à l'aide de la formule récursive. Une fois calculée, l'association (n, f_n) sera intégrée au dictionnaire :

```
d_fib = {0: 0, 1: 1}

def fib(n):
    if n not in d_fib:
        d_fib[n] = fib(n-1) + fib(n-2)
    return d_fib[n]
```

Cette solution permet de concilier la simplicité de la solution récursive avec l'efficacité temporelle, au prix d'un coût spatial linéaire (constitué du dictionnaire et de la pile d'exécution) :

```
fib <- 6
 fib <- 5
  fib <- 4
   fib <- 3
    fib <- 2
     fib <- 1
     fib -> 1
     fib <- 0
     fib -> 0
    fib -> 1
    fib <- 1
    fib -> 1
   fib -> 2
   fib <- 2
    fib <- 1
    fib -> 1
    fib <- 0
```

```
     fib -> 0
    fib -> 1
   fib -> 3
   fib <- 3
    fib <- 2
     fib <- 1
     fib -> 1
     fib <- 0
     fib -> 0
    fib -> 1
    fib <- 1
    fib -> 1
   fib -> 2
  fib -> 5
  fib <- 4
   fib <- 3
    fib <- 2
```

```
     fib <- 1
     fib -> 1
     fib <- 0
     fib -> 0
    fib -> 1
    fib <- 1
    fib -> 1
   fib -> 2
   fib <- 2
    fib <- 1
    fib -> 1
    fib <- 0
    fib -> 0
   fib -> 1
  fib -> 3
 fib -> 8
```

FIGURE 12 – La trace de la fonction récursive `fib`.

- `d = {c1: v1, c2: v2, c3: v3}` crée un dictionnaire `d` comportant pour l'instant trois paires d'association ;
- `d[c2]` renvoie la valeur (ici v_2) associée à la clef c_2 ou déclenche l'exception `KeyError` si l'association n'existe pas ;
- `d[c4] = v4` permet d'ajouter une nouvelle paire d'association (ou de remplacer la précédente association si c_4 est déjà dans le dictionnaire) ;
- `(c in d)` renvoie le booléen `True` si la clef c est dans d, et False sinon.

Seules restrictions : il n'est pas permis d'avoir plus d'une entrée par clef, et ces dernières ne doivent être des objets immuables.

FIGURE 13 – Un récapitulatif des principales fonctions des dictionnaires.

```
In [1]: fib(10)
Out[1]: 55

In [2]: d_fib
Out[2]: {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55}
```

Autre exemple classique, le calcul des coefficients binomiaux à l'aide de la formule de PASCAL doit impérativement être mémoïsée sous peine d'obtenir un coût temporel exponentiel.

```
d_binom = {}

def binom(n, p):
    if (n, p) not in d_binom:
        if p == 0 or n == p:
            d_binom[(n, p)] = 1
        else:
            d_binom[(n, p)] = binom(n-1, p-1) + binom(n-1, p)
    return d_binom[(n, p)]
```

Sans mémoïsation, il faut près de 2 minutes pour calculer $\binom{30}{15}$ avec un processeur Intel Core 2 Duo à 2,13 GHz alors que le calcul demande moins d'une milli-seconde avec mémoïsation.

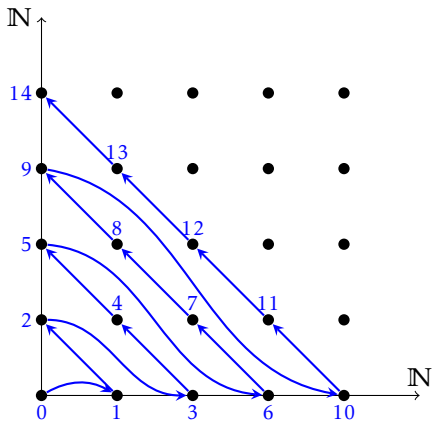
3. Exercices

Exercice 4

Écrire une fonction récursive qui calcule x^n en exploitant la relation : $x^n = x^{\lfloor n/2 \rfloor} \times x^{\lceil n/2 \rceil}$. Combien de multiplications effectue-t-elle ?

Exercice 5

On démontre que l'ensemble $\mathbb{N} \times \mathbb{N}$ est dénombrable en numérotant chaque couple $(x, y) \in \mathbb{N}^2$ suivant le procédé suggéré par la figure ci-dessous :



Rédiger une fonction *récursive* qui retourne le numéro du point de coordonnées (x, y).
Rédiger la fonction réciproque, là encore de façon récursive.

Exercice 6

On suppose donné un tableau $t[0 \cdots n-1]$ (contenant au moins trois éléments) qui possède la propriété suivante : $t_0 \geqslant t_1$ et $t_{n-2} \leqslant t_{n-1}$. Soit $k \in [\![1, n-2]\!]$; on dit que t_k est un *minimum local* lorsque $t_k \leqslant t_{k-1}$ et $t_k \leqslant t_{k+1}$.

a) Justifier l'existence d'un minimum local dans t.

b) Il est facile de déterminer un minimum local en coût linéaire : il suffit de procéder à un parcours du tableau. Mais pourriez-vous trouver un algorithme récursif qui en trouve un en coût logarithmique ?

Exercice 7

On suppose les ensembles représentés par des listes non triées d'éléments deux-à-deux distincts. Rédiger une fonction `subset` qui prend pour argument un ensemble et renvoie l'ensemble de ses sous-ensembles. Par exemple :

```
In [1]: subset([1,2,3])
Out[1]: [[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
```

Exercice 8

Écrire une fonction `somme(M)` qui prend en paramètre une séquence imbriquée, de profondeur et de structure quelconques, dont tous les composants élémentaires sont des nombres, et calcule la somme de tous ces éléments. Par exemple, `somme([[[1, 2], [3, 4, 5]], 6, [7, 8], 9])` devra renvoyer `45`.
Indication. L'expression booléenne `isinstance(x, numbers.Real)` *permet de tester si x est un nombre.*

```
In [1]: isinstance(1, numbers.Real)
Out[1]: True

In [1]: isinstance([1], numbers.Real)
Out[1]: False
```

Exercice 9

On suppose disposer d'une fonction `circle((x, y), r)` qui trace à l'écran un cercle de centre (x,y) de rayon r. Définir deux fonctions récursives permettant de tracer les dessins présentés figure 14 (chaque cercle est de rayon moitié moindre qu'à la génération précédente).

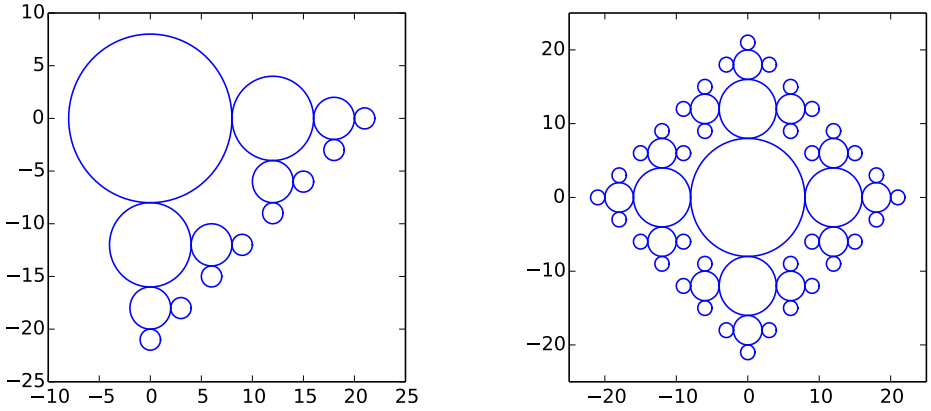


Figure 14 – Le résultat des fonctions `bubble1(4)` et de `bubble2(4)`.

Exercice 10

On suppose disposer d'une fonction `polygon((xa, ya), (xb, yb), (xc, yc))` qui trace le triangle plein dont les sommets ont pour coordonnées (x_a,y_a), (x_b,y_b), (x_c,y_c). Définir une fonction récursive permettant le tracé présenté figure 15 (tous les triangles sont équilatéraux).

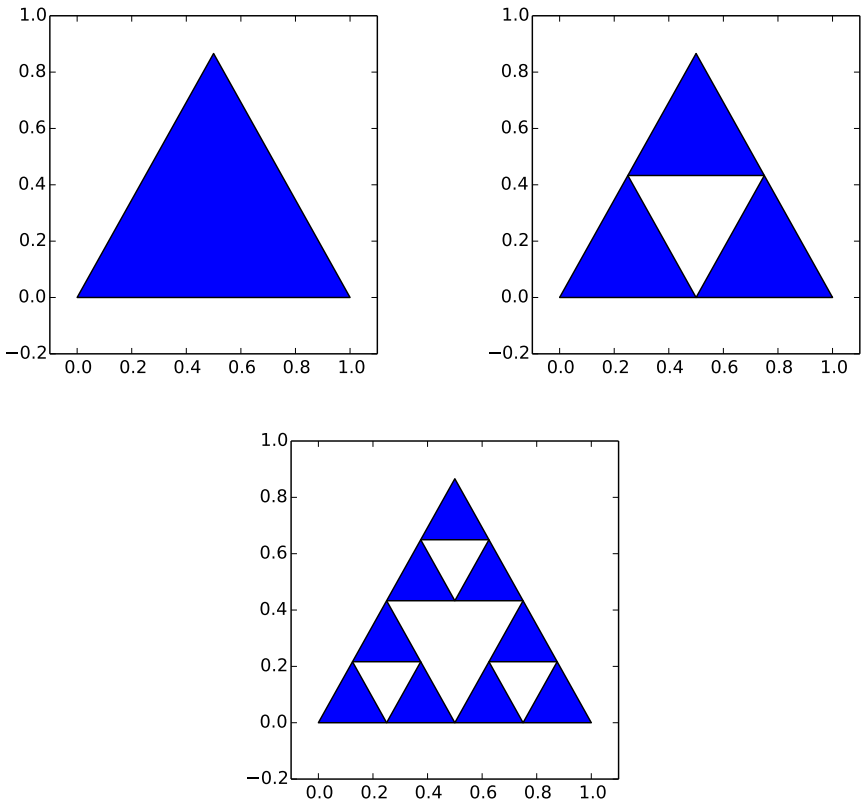


Figure 15 – Le résultat des fonctions `sierpinski(n)` pour $n = 1, 2, 3$.

Exercice 11

Le *problème des n reines* consiste à placer n reines sur un échiquier de taille $n \times n$ de sorte que deux reines quelconques ne puissent jamais s'attaquer (pour ceux d'entre vous qui ne sont pas familiers avec le jeu d'échecs, ceci signifie que deux reines quelconques ne partagent jamais la même ligne, la même colonne ou la même diagonale). De telles positions seront par la suite qualifiées de *valides* (illustration figure 16).
Il est évident que dans chaque colonne doit se trouver exactement une reine ; ainsi il est possible de représenter ce problème par un tableau de n cases $q = [q_0, \ldots, q_{n-1}]$ dans lequel q_j désigne la ligne où est placée la reine de la colonne j.

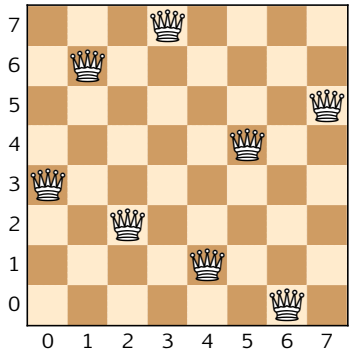


Figure 16 – Une solution, représentée par le tableau $[3, 6, 2, 7, 1, 4, 0, 5]$.

On appelera *solution partielle de rang j* un tableau q de longueur n dont les j premières cases sont remplies par des positions valides pour les reines, les $n - j$ autres cases restant à remplir.

Rédiger une fonction récursive `reine(q, j)` qui prend pour arguments un entier j et une solution partielle q et qui réalise les opérations suivantes :

- si $j = n$ cette fonction se contente d'afficher le tableau q (le problème est résolu) ;
- si $j < n$, cette fonction recherche parmi les n valeurs possibles pour q_j celles qui correspondent à des positions valides et pour chacune d'elles poursuit la recherche au rang $j + 1$.

Chapitre IV
Algorithmes de tri

1. Introduction

Outre l'intérêt intrinsèque que peut représenter le tri des éléments d'un ensemble, il peut être utile, en préalable à un traitement de données, de commencer par trier celles-ci. Considérons par exemple le problème de la recherche d'un élément dans un tableau. Ce problème a un coût linéaire lorsque le tableau n'est pas trié, et un coût logarithmique si ce dernier est trié. Ainsi, si on prévoit de faire de nombreuses recherches, il peut devenir intéressant de commencer par trier ces données, même si le coût du tri s'ajoute au coût des différentes recherches.
À titre d'exemple, notons n le nombre d'éléments du tableau, et p le nombre de recherches à effectuer. Si on ne trie pas le tableau la complexité totale est un $p \times O(n) = O(np)$; si on trie le tableau au préalable, la complexité totale est égale à $C(n) + p \times O(\log n) = C(n) + O(p \log n)$, où $C(n)$ est la complexité de l'algorithme de tri choisi. Nous verrons dans ce chapitre que les algorithmes de tris efficients ont une complexité $C(n) = O(n \log n)$; la complexité totale de la deuxième méthode est alors égale à un $O((n+p)\log n)$.
On a $(n+p)\log n \leqslant np \iff p \geqslant \dfrac{n \log n}{n - \log n} \sim \log n$ donc lorsque $p \geqslant \log n$, il devient préférable de trier la liste au préalable.

1.1 Présentation du problème

Avant toute chose, il importe de prendre conscience que la performance d'un tri va être directement liée à la structure de données utilisée et en particulier du temps d'accès à un élément : nous savons que l'accès à un élément d'un tableau est de coût constant, mais d'autres structures de données peuvent avoir des temps d'accès de complexité non constante. Certains algorithmes peuvent donc se révéler plus adaptés à une structure de donnée plutôt qu'à une autre. Les méthodes de tris peuvent aussi différer suivant que la structure de donnée à trier soit mutable ou pas. Dans le cas d'une structure mutable (un tableau par exemple), on distinguera deux type de tris :

- les tris *en place*, pour lesquels on procède par permutation des éléments au sein de la structure de données ;
- les autres tris, pour lesquels les éléments sont recopiés et triés, dans une nouvelle structure de données.

L'intérêt des tris en place est de posséder une complexité spatiale moindre (en général constante) que les autres tris, qui ont une complexité spatiale au moins égale à la taille de la structure de donnée à trier.

Par exemple, en Python, la fonction `sorted` ne réalise pas un tri en place : dans l'exemple qui suit le tableau `t` n'est pas modifié.

```
In [1]: t = [5, 3, 6, 9, 1, 2, 4, 7, 8]

In [2]: sorted(t)
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: t
Out[3]: [5, 3, 6, 9, 1, 2, 4, 7, 8]
```

En revanche, la méthode `sort` réalise un tri en place :

```
In [4]: t.sort()

In [5]: t
Out[5]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

L'autre élément pertinent de mesure de la complexité est la manière dont on détermine l'ordre relatif des éléments à trier. Le programme se limite aux tris qui procèdent par *comparaison entre les éléments* de la structure

à trier. Nous allons donc considérer que ces éléments appartiennent à un ensemble E muni d'une relation d'ordre $\leqslant$ et nous autoriser uniquement à comparer *entre eux* deux éléments de la structure. Nous supposerons de plus que cette comparaison est de complexité constante.

Dans la pratique, les algorithmes que nous allons étudier seront illustrés en Python par le tri d'un tableau (de type `list` ou `numpy.array`) à valeurs numériques. Ainsi, l'affectation et la comparaison de deux éléments entre eux se réaliserons à coût constant et constituerons la mesure de complexité de ces algorithmes.

Sous ces hypothèses nous admettrons le théorème suivant :

Théorème 1.1 — *Tout algorithme de tri par comparaison d'un tableau de n cases possède une complexité dans le pire des cas au moins égale à un* $O(n\log n)$.

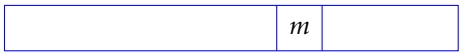
Remarque. Il existe des algorithmes qui n'utilisent pas de comparaison entre éléments mais tirent profit d'une information supplémentaire dont on dispose sur les éléments à trier. Le tri par base (*radix sort* en anglais) utilise la décomposition dans une base donnée des nombres entiers pour les trier. D'autres algorithmes tirent partie de la représentation en mémoire des objets à trier.

Nous allons maintenant étudier quelques algorithmes de tri par comparaison élémentaires, avant de nous intéresser aux algorithmes de tri les plus performants.

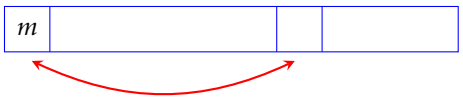
1.2 Le tri par sélection

Appelé *selection sort* en anglais, c'est l'algorithme le plus simple qui soit : on cherche d'abord le plus petit élément du tableau, que l'on échange avec le premier. On applique alors cette méthode au sous-tableau restant.

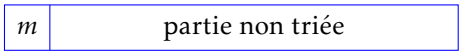
– recherche de l'élément minimal :



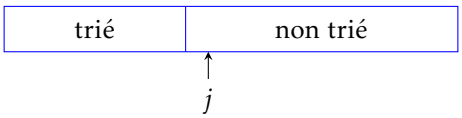
– permutation avec le premier :



– tri du tableau restant :



Considérons le tableau en cours de tri. La partie gauche $t[:j]$ du tableau est triée, les éléments de cette partie sont à leurs places définitives, et il reste à trier la partie droite $t[j:]$.



Nous allons donc rédiger une fonction `minimum(t, j)` qui prend pour arguments un tableau t et un entier j et qui renvoie l'indice du minimum de la coupe $t[j:]$, pour en déduire une fonction `selectionSort(t)` qui prend pour argument un tableau t et réalise le tri en place de ce dernier en suivant la méthode du tri par sélection.

```
def minimum(t, j):
    i, m = j, t[j]
    for k in range(j+1, len(t)):
        if t[k] < m:
            i, m = k, t[k]
    return i
```

```
def selectionSort(t):
    for j in range(len(t)-1):
        i = minimum(t, j)
        if i != j:
            t[i], t[j] = t[j], t[i]
```

Étude de la complexité

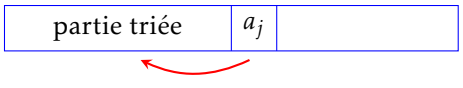
Le nombre de comparaisons effectuées par la fonction `minimum(t, j)` est égal à $n-1-j$ donc le nombre total de comparaisons est égal à $\sum_{j=0}^{n-2}(n-1-j) = \frac{n(n-1)}{2} \sim \frac{n^2}{2}$ (quelle que soit la configuration du tableau) et le nombre d'échanges inférieur ou égal à $n-1$. Il s'agit donc d'un algorithme de complexité quadratique $O(n^2)$. Une de

ses particularités est le nombre réduit (linéaire) d'échanges effectués, mais à part cela c'est le plus mauvais (en termes de comparaisons) que nous étudierons.

1.3 Le tri par insertion

Appelé *insertion sort* en anglais, il consiste à parcourir le tableau en insérant à chaque étape l'élément d'indice j dans la partie du tableau (déjà triée) à sa gauche, un peu à la manière d'un joueur de cartes insérant dans son jeu trié les cartes au fur et à mesure qu'il les reçoit.

– insertion dans un tableau trié :



– tri du tableau restant :

Nous allons donc rédiger une fonction `insere(t, j)` qui prend pour arguments un tableau t et un entier j en supposant la coupe $t[:j]$ triée et qui réalise l'insertion de l'élément $t[j]$ dans cette partie triée, puis une fonction `insertionSort(t)` qui prend pour argument un tableau t et réalise le tri en place de ce dernier en suivant la méthode du tri par insertion.

```
def insere(t, j):
    k, a = j, t[j]
    while k > 0 and a < t[k-1]:
        t[k] = t[k-1]
        k -= 1
    t[k] = a
```

```
def insertionSort(t):
    for j in range(1, len(t)):
        insere(t, j)
```

Étude de la complexité
Insérer un élément dans un tableau trié de longueur j nécessite au plus j comparaisons, ce qui conduit à un nombre maximal de comparaisons égal à $\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} \sim \frac{n^2}{2}$. Cette situation a effectivement lieu dans le cas où la liste initiale est triée par ordre décroissant, ce qui nous permet d'énoncer le :

Théorème 1.2 — *Dans le pire des cas, le nombre de comparaisons du tri par insertion est équivalent à* $\frac{n^2}{2}$.

Il s'agit donc là encore d'un algorithme de complexité quadratique $O(n^2)$. De plus, le nombre de comparaisons dans le pire des cas est identique au nombre de comparaisons utilisé par l'algorithme de tri par sélection. Cependant, lorsque le tableau est déjà trié l'algorithme ne réalise que n comparaisons, ce qui est un des points forts du tri par insertion (il est très efficace pour trier des tableaux quasi-triés), et il s'avère aussi plus performant en moyenne puisqu'il est possible de prouver le résultat suivant, que nous admettrons :

Théorème 1.3 — *Le tri par insertion effectue en moyenne un nombre de comparaisons équivalent à* $\frac{n^2}{4}$.

2. Algorithmes de tris efficients

Nous venons d'étudier deux algorithmes de tri par comparaison, tous deux de complexité quadratique, aussi bien dans le pire des cas qu'en moyenne. Nous allons maintenant nous intéresser à deux algorithmes plus efficients : le tri fusion déjà étudié au chapitre précédent, et le tri rapide.

2.1 Le tri fusion

Appelée *merge sort* en anglais, nous l'avons déjà rencontré comme exemple d'algorithme récursif. Cette méthode adopte une approche « diviser pour régner » : on partage le tableau en deux parties de tailles respectives $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$ que l'on trie par un appel récursif, puis on fusionne les deux parties triées (illustration figure 1).

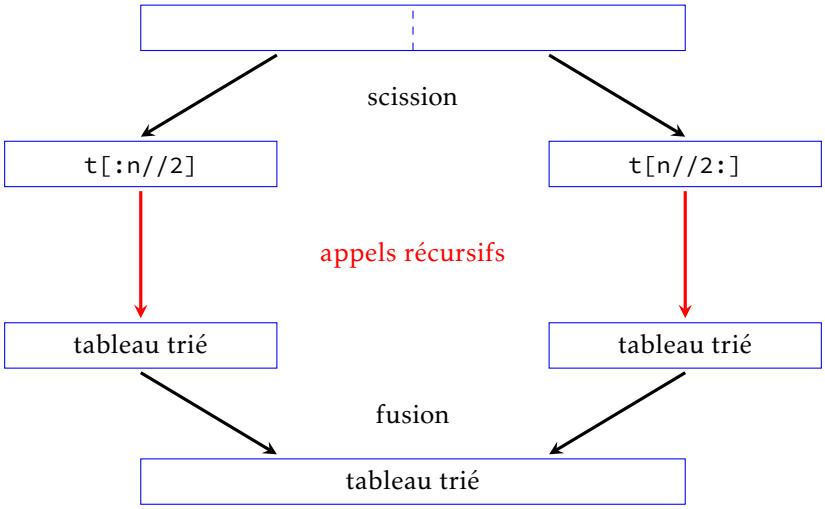


FIGURE 1 – Une illustration du tri fusion.

Cependant, cette méthode de tri possède un inconvénient : il est très difficile (pour ne pas dire impossible) de fusionner deux demi-tableaux en place, c'est à dire en s'autorisant uniquement la permutation de deux éléments dans le tableau sans espace mémoire supplémentaire. Pour cette raison, les implémentations du tri fusion ne sont pas des tris en place, autrement dit calculent un *nouveau* tableau trié.

Le code du tri fusion a déjà été donné au chapitre précédent.

Étude de la complexité temporelle

Notons $T(n)$ la complexité temporelle du tri fusion. La fusion de deux tableaux triés se réalise en temps linéaire vis-à-vis de la somme des tailles des deux tableaux à fusionner donc $T(n)$ vérifie une relation de récurrence de la forme :

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n).$$

Autrement dit, il existe une constante K telle que pour tout $n \geqslant 2$, $T(n) \leqslant T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + Kn$.
Lorsque $n = 2^p$ la suite $u_p = T(2^p)$ vérifie la relation $u_p \leqslant 2u_{p-1} + K2^p$ que l'on peut encore écrire : $\dfrac{u_p}{2^p} - \dfrac{u_{p-1}}{2^{p-1}} \leqslant K$.
Par télescopage on en déduit $\dfrac{u_p}{2^p} - u_0 \leqslant Kp$ puis $u_p \leqslant (Kp + u_0)2^p$, soit $T(n) = O(n \log n)$. Nous admettrons que cette formule reste vraie pour un entier n quelconque. Ainsi, *le coût temporel du tri fusion est semi-logarithmique*.

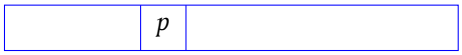
Étude de la complexité spatiale

Notons $S(n)$ la complexité spatiale du tri fusion. Telle que nous l'avons écrit au chapitre précédent, la fusion a un coût linéaire vis-à-vis de la somme des tailles de ses arguments, ce qui conduit à une relation de la forme : $S(n) = S(\lfloor n/2 \rfloor) + S(\lceil n/2 \rceil) + O(n)$, soit $S(n) = O(n \log n)$. Cependant il est possible assez facilement de ramener ce coût à un $O(n)$.

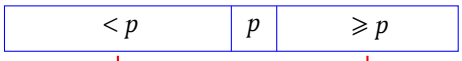
2.2 Le tri rapide

Appelé *quick sort* en anglais, ce tri adopte lui aussi une démarche de type « diviser pour régner » : on commence par segmenter le tableau autour d'un pivot choisi parmi les éléments du tableau en plaçant les éléments qui lui sont inférieurs à sa gauche, et les éléments qui lui sont supérieurs, à sa droite. À l'issue de cette étape, le pivot se trouve à sa place définitive, et les parties gauche et droite sont triées par l'intermédiaire d'un appel récursif.

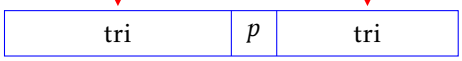
– choix d'un pivot :



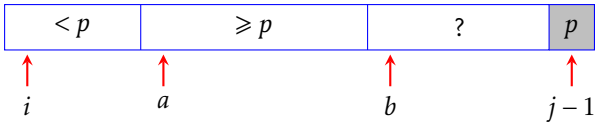
– segmentation :



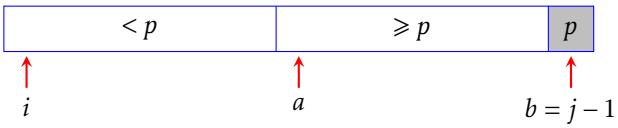
– appels récursifs :



Il existe plusieurs façons de segmenter une portion de tableau $t[i:j]$. la méthode que nous allons suivre consiste à choisir pour pivot l'élément $p = t[j-1]$ et à maintenir l'invariant suivant :



Les valeurs initiales sont $a = b = i$ et la condition d'arrêt $b = j-1$. Lorsque le processus se termine, la situation est la suivante :



et il suffit alors de permuter les cases d'indices a et $j-1$ pour achever la segmentation du tableau $t[i:j]$.

On commence donc par rédiger une fonction `segmente(t, i, j)` qui prend pour arguments un tableau t et deux indices i et j, qui segmente la coupe $t[i:j]$ en utilisant l'élément $p = t[j-1]$ comme pivot et qui renvoie l'indice a du pivot en position finale. Ensuite, il nous reste à écrire la fonction `quickSort(t, i, j)` qui prend pour argument un tableau t et deux indices i et j et réalise le tri en place de la coupe $t[i:j]$ en suivant la méthode du tri rapide.

```
def segmente(t, i, j):
    p = t[j-1]
    a = i
    for b in range(i, j-1):
        if t[b] < p:
            t[a], t[b] = t[b], t[a]
            a += 1
    t[a], t[j-1] = t[j-1], t[a]
    return a
```

```
def quickSort(t, i, j):
    if i + 1 < j:
        a = segmente(t, i, j)
        quickSort(t, i, a)
        quickSort(t, a + 1, j)
```

Pour avoir une fonction de tri en place d'un tableau par la méthode du tri rapide il reste à définir la fonction :

```
def triRapide(t):
    quickSort(t, 0, len(t))
```

2.3 Étude de la complexité du tri rapide

Il apparaît clairement que la complexité de la segmentation est linéaire. Cette dernière ayant pour objet de partager le tableau en deux sous-ensembles de tailles n_1 et n_2 avec $n_1 + n_2 = n-1$ (le pivot n'appartient à aucun de ces deux sous-ensembles), la complexité de l'algorithme de tri rapide vérifie la relation : $C(n) = C(n_1) + C(n_2) + O(n)$.

L'intuition nous pousse à penser que cette méthode est d'autant meilleure que n_1 et n_2 sont proches et qu'à l'inverse un partitionnement très déséquilibré risque d'avoir une influence négative sur le coût total. Avant de justifier rigoureusement ces assertions, intéressons-nous à ces deux situations extrêmes.

Partitionnement dans le cas le plus défavorable
Supposons qu'à chaque partitionnement l'une des deux parties du tableau segmenté soit vide : nous avons alors $n_1 = n-1$ et $n_2 = 0$ et la relation de récurrence devient : $C(n) = C(n-1) + O(n)$, ce qui conduit immédiatement à $C(n) = O(n^2)$. Cette situation a lieu par exemple lorsque le tableau est déjà trié (dans un sens comme dans l'autre) puisque le pivot choisi est à chaque étape un élément extrémal de la partie à segmenter.

Partitionnement dans le cas le plus favorable
À l'inverse, supposons qu'à chaque étape du partitionnement les deux parties du tableau segmenté soient exactement de même taille ; nous avons alors $n = 2^p - 1$ et la relation de récurrence devient $C(n) = 2C(\lfloor n/2 \rfloor) + O(n)$. Posons $u_p = C(2^p - 1)$; alors $u_p = 2u_{p-1} + O(2^p)$. Nous avons déjà effectué ce calcul pour le tri fusion et obtenu $u_p = O(p2^p)$ soit $C(n) = O(n \log n)$.

Nous allons maintenant prouver le :

Théorème 2.1 — *Lorsque le choix du pivot est arbitraire, le coût de l'algorithme de tri rapide dans le pire des cas est quadratique.*

Partitionnement équilibré
Pourquoi le qualifier de tri rapide alors que le cas défavorable est quadratique ? Parce qu'en moyenne la complexité est quasi-linéaire (c'est-à-dire en $O(n \log n)$), résultat que nous admettrons :

Théorème 2.2 — *En moyenne, la complexité du tri rapide est un* $O(n \log n)$.

Remarque. Plus précisément on peut démontrer que $C(n) \sim 2n \ln n \approx 1,4\, n \log n$. Or on peut montrer que tout algorithme de tri par comparaison réalise dans le pire des cas au moins $n \log n$ comparaisons (c'est un raffinement du théorème 1.1). La constante $1,4$ qui apparaît dans la complexité moyenne du tri rapide montre que l'on est pas loin de la valeur optimale. Ainsi, ce tri se montre dans la pratique plus rapide que la plus-part des autres algorithmes de tri de complexité $O(n \log n)$, ce qui justifie son nom.

2.4 Choix du pivot

Nous l'avons constaté, le choix du dernier élément comme pivot pour segmenter $t[i : j]$ donne en moyenne une complexité quasi-linéaire, mais pour des tableaux triés ou presque triés la complexité est quadratique.
Dans le cas de figure où l'on prévoit d'appliquer l'algorithme de tri rapide à des données peu mélangées, on peut avoir intérêt à choisir au hasard un pivot dans $t[i : j]$. La modification de la fonction de segmentation est mineure : il suffit de tirer au hasard un entier α dans l'intervalle $[\![i, j-1]\!]$ puis à échanger les éléments $t[\alpha]$ et $t[j-1]$ avant de procéder à la segmentation. Le pivot étant maintenant choisi au hasard on peut s'attendre à ce que le partitionnement du tableau d'entrée soit en moyenne relativement équilibré.
Paradoxalement, une autre solution pour trier un tableau presque trié à l'aide du tri rapide consiste … à commencer par mélanger ce dernier ! Néanmoins, cette solution s'avère en pratique moins intéressante que le fait de choisir au hasard le pivot.

3. Exercices

Tris par comparaison

Exercice 1

On désire un algorithme qui détermine si un tableau présente des doublons en son sein.

a. Rédiger un algorithme naïf qui résout le problème. Quelle est sa complexité ?

b. Rédiger maintenant un second algorithme en supposant cette fois le tableau trié. Quelle est sa complexité ? A-t-on intérêt à trier un tableau pour résoudre ce problème ?

Exercice 2

Un singe trie un paquet de cartes de la façon suivante : il prend les cartes, les jette en l'air, les ramasse puis regarde si elles sont triées. Si ce n'est pas le cas il relance les cartes.
On suppose donnée une fonction `shuffle(t)` qui prend une séquence t pour argument et en mélange son contenu en place[a]. Rédiger une fonction `monkeySort(t)` qui réalise le tri en place suivant cette méthode.

a. Cette fonction est présente dans la bibliothèque `numpy.random`.

Exercice 3

Montrer que $n-1$ comparaisons et échanges *entre éléments consécutifs* permettent de placer l'élément maximal en queue d'un tableau à trier. En déduire une nouvel algorithme de tri en place (le tri bulle, ou *bubble sort*), que vous rédigerez en Python.
Faire une analyse de sa complexité temporelle.

Exercice 4

Imaginé par Donald Shell en 1959, *shellsort* est une optimisation du tri par insertion.
Dans le tri par insertion, un élément se rapproche de sa place finale en progressant lentement, case par case. L'accélération consiste à le déplacer en commençant par faire des grands pas, puis des pas de plus en plus petits, jusqu'à, évidemment, des pas de 1 pour que le tableau soit trié.
On considère une suite d'entiers $(h_p)_{p\in\mathbb{N}^*}$ strictement croissante, débutant par $h_1 = 1$. Pour tout $n \in \mathbb{N}^*$, il existe donc un unique entier $p \in \mathbb{N}^*$ tel que $h_p \leqslant n < h_{p+1}$.
L'étape de base de l'algorithme consiste à trier à l'aide du tri par insertion chacun des sous-tableaux débutant respectivement par $a_0, \dots, a_{h_p-1}$, et dont les éléments sont séparés de h_p cases. Une fois cette étape achevée, on dit que le tableau initial est h_p-trié. On répète alors cette opération de base avec la valeur h_{p-1}, et ce jusqu'à la valeur finale 1.

a. Rédiger une fonction `insert(t, h)` qui réalise le h-tri en place du tableau t.

b. Hibbard a démontré en 1963 que la suite $h_p = 2^p - 1$ conduit à un coût en $O(n^{3/2})$. Rédiger en Python le tri de Shell pour cette suite de valeurs.

Remarque. Connaître la suite (h_p) conduisant à la meilleure complexité pour ce tri est à l'heure actuelle un problème ouvert, ainsi que la valeur de cette complexité optimale. La meilleure complexité obtenue à ce jour est en $O(n(\log n)^2)$.

Exercice 5

On considère l'algorithme suivant :

```
def stoogeSort(t, i, j):
    if t[i] > t[j-1]:
        t[i], t[j-1] = t[j-1], t[i]
    if j - i > 2:
        k = (j - i) // 3
        tri(t, i, j-k)
        tri(t, i+k, j)
        tri(t, i, j-k)
```

a) Montrer que si t est un tableau de taille n, l'exécution de `stoogeSort(t, 0, n)` trie correctement le tableau t.

b) Donner une relation de récurrence vérifiée par la complexité temporelle $C(n)$ de cet algorithme, puis, en admettant qu'il existe un réel α tel que $C(n) = O(n^\alpha)$, déterminer cette valeur α.

Remarque. Les trois Stooges (Moe, Larry et Curly) étaient une troupe de comiques américains actifs entre 1930 et 1960 ; ils sont connus en France sous le nom des trois corniauds.

Segmentation et tri rapide

Exercice 6

(Algorithme du *drapeau tricolore*)
On considère un tableau contenant des objets possédant une couleur qui ne peut être que rouge, blanc ou bleu. On souhaite trier ce tableau de sorte que les éléments rouges soient placés en tête, ensuite les éléments blancs, puis enfin les éléments bleus [a].
Pour modéliser cette situation en Python, on suppose que les objets à trier possèdent un attribut `couleur` ne pouvant prendre que les valeurs `'rouge'`, `'blanc'` ou `'bleu'`.
Rédiger en Python une fonction effectuant ce tri en procédant par segmentation ; l'algorithme ne devra effectuer qu'un seul parcours du tableau, et ne déterminer la couleur qu'une fois par élément.

a. ce problème est attribué à Edsger Dijkstra, il suit donc l'ordre des couleurs du drapeau néerlandais.

Exercice 7

Déduire du principe de segmentation un algorithme, qu'on rédigera en Python, pour déterminer le k^{e} plus petit élément d'un vecteur, puis pour déterminer l'élément médian d'un tableau. Que penser du coût de ce dernier algorithme ?

Chapitre V
Bases de données

1. Description d'une base de données

1.1 Introduction

Chacun d'entre nous utilise quotidiennement, mais sans souvent en avoir conscience, des bases de données. Ces dernières regroupent des informations relatives à un domaine précis, souvent ordonnées sous une forme très structurée. Les exemples sont innombrables : gestion des stocks (magasins en ligne, bibliothèques, ...), gestion des réservations (trains, avions, spectacles ...), gestion du personnel d'une entreprise, création de listes diverses (vos playlists musicales préférées par exemple), etc.

Prenons l'exemple du site de réservation de la SNCF. Les clients peuvent réaliser un certain nombre de tâches, parmi lesquelles :

- la consultation des trains répondants à certains critères (date, trajet, places disponibles, etc) ;
- la réservation d'une place dans le train choisi ;
- l'annulation d'une réservation.

Le personnel de la SNCF, outre les opérations précédentes, peut en outre :

- modifier la composition ou les horaires des trains ;
- ajouter des trains supplémentaires ;
- supprimer des trains.

À l'exception de la consultation, les autres opérations sont complexes car elles doivent gérer le principe de simultanéité qui régit les bases de données : plusieurs clients peuvent chercher à réserver en même temps des places dans le même train, et il ne s'agit pas d'attribuer la même place à plusieurs personnes différentes. Il en va de même de la suppression d'un train, qui doit tenir compte des personnes ayant déjà réservé dans celui-ci. C'est pourquoi nous nous limiterons, dans ce cours d'introduction aux bases de données, à la simple consultation des données.

1.2 Structure d'une base de données

L'accès à une base de données se fait usuellement par l'intermédiaire d'une application qui traduit les demandes de l'utilisateur (en général via une interface graphique) dans un langage dédié à la communication avec une base de données. Ce langage, le SQL (*Structured Query Language*) est devenu un standard disponible sur presque tous les systèmes de gestion des bases de données (SGBD).

Figure 1 – Communication avec une base de données.

Le SQL comporte des instructions relatives :

- à l'interrogation de bases de données ;
- à la création et la modification des données ;
- au contrôle d'accès des données.

Comme indiqué dans le préambule, nous nous intéresserons uniquement aux instructions de la première catégorie.

Dans la suite de ce cours j'utiliserai MySQL comme système de gestion pour interroger une base de données intitulée `world.sql`. Cette base de donnée contient un certain nombre d'informations géographiques et politiques mondiales (en 2001).
Commençons par observer son contenu :

```
mysql> show tables;
+-----------------+
| Tables_in_world |
+-----------------+
| city            |
| country         |
| countrylanguage |
+-----------------+
```

Une base de données est un ensemble de *tables* (ou de *relations*, les deux termes étant synonymes dans ce contexte) que l'on peut représenter sous forme de tableaux bi-dimensionnels. Dans l'exemple qui nous intéresse, notre base de données est composée de trois tables :

- `city`, qui contient des informations relatives à certaines villes du monde ;
- `country`, qui contient des informations relatives aux pays ;
- `countrylanguage`, qui contient des informations relatives aux langues parlées dans le monde.

Examinons le contenu de la première, ou plutôt un extrait de son contenu qui est très important :

```
mysql> select * from city limit 10;
+----+----------------+-------------+---------------+------------+
| ID | Name           | CountryCode | District      | Population |
+----+----------------+-------------+---------------+------------+
|  1 | Kabul          | AFG         | Kabol         |    1780000 |
|  2 | Qandahar       | AFG         | Qandahar      |     237500 |
|  3 | Herat          | AFG         | Herat         |     186800 |
|  4 | Mazar-e-Sharif | AFG         | Balkh         |     127800 |
|  5 | Amsterdam      | NLD         | Noord-Holland |     731200 |
|  6 | Rotterdam      | NLD         | Zuid-Holland  |     593321 |
|  7 | Haag           | NLD         | Zuid-Holland  |     440900 |
|  8 | Utrecht        | NLD         | Utrecht       |     234323 |
|  9 | Eindhoven      | NLD         | Noord-Brabant |     201843 |
| 10 | Tilburg        | NLD         | Noord-Brabant |     193238 |
+----+----------------+-------------+---------------+------------+
```

Nous pouvons constater qu'une table est un tableau bi-dimensionnel : les en-têtes des différentes colonnes sont appelés des *attributs*, les lignes des *enregistrements*. Ainsi, la table `city` possède 5 attributs : `ID`, `Name` (le nom d'une ville), `CountryCode` (le code du pays dans lequel se trouve cette ville), `District` (sa région d'appartenance) et `Population` (son nombre d'habitants).

Précisons maintenant les caractéristiques de chacun de ces attributs :

```
mysql> describe city;
+-------------+----------+------+-----+
| Field       | Type     | Null | Key |
+-------------+----------+------+-----+
| ID          | int      | NO   | PRI |
| Name        | char(35) | NO   |     |
| CountryCode | char(3)  | NO   |     |
| District    | char(20) | NO   |     |
| Population  | int      | NO   |     |
+-------------+----------+------+-----+
```

Nous constatons qu'un attribut est un objet *typé* (dans ce contexte, le type d'un attribut est son *domaine*). `ID` et `Population` sont des entiers, `Name` une chaîne d'au plus 35 caractères, `CountryCode` une chaîne d'au plus 3 caractères, et `District` une chaîne d'au plus 20 caractères. Ces caractéristiques sont fixées lors de la création d'une table et influent sur les opérations que nous serons susceptibles de réaliser sur les valeurs de ces attributs :

+, -, *, * pour les entiers et les nombres flottants, =, <>, <, <=, >, >= pour les comparaisons (pour les chaînes de caractères, l'ordre lexicographique est utilisé).
La colonne suivante indique que tous les attributs doivent posséder une valeur ; enfin, la troisième colonne apporte une information importante : chaque enregistrement d'une table doit pouvoir être identifié de manière unique ; c'est le rôle de la *clef primaire*, constitué d'un ou plusieurs attributs. Ici, la clef primaire est constituée de l'unique attribut `ID`. Il ne peut donc y avoir deux enregistrements ayant le même `ID`.

Regardons maintenant le contenu de la table `country` :

```
mysql> describe country;
+----------------+---------------------------------------------------+------+-----+
| Field          | Type                                              | Null | Key |
+----------------+---------------------------------------------------+------+-----+
| Code           | char(3)                                           | NO   | PRI |
| Name           | char(52)                                          | NO   |     |
| Continent      | enum('Asia','Europe','North America',             | NO   |     |
|                | 'Africa','Oceania','Antarctica','South America')  |      |     |
| Region         | char(26)                                          | NO   |     |
| SurfaceArea    | decimal(10,2)                                     | NO   |     |
| IndepYear      | smallint                                          | YES  |     |
| Population     | int                                               | NO   |     |
| LifeExpectancy | decimal(3,1)                                      | YES  |     |
| GNP            | decimal(10,2)                                     | YES  |     |
| GovernmentForm | char(45)                                          | NO   |     |
| HeadOfState    | char(60)                                          | YES  |     |
| Capital        | int                                               | YES  |     |
+----------------+---------------------------------------------------+------+-----+
```

Ici, la clef primaire est l'attribut `Code`, qui est une chaîne de trois caractères. On constate en outre que certaines données ne sont pas obligatoires : les attributs `IndepYear`, `LifeExpectancy`, `GNP`, `HeadOfState` et `Capital` peuvent être absents, auquel cas la valeur qui leur est attribuée est `NULL`.
Voyons un extrait de cette table, par exemple l'enregistrement qui concerne la France :

```
mysql> select * from country where Name = 'France';
+------+--------+-----------+----------------+-------------+-----------+------------+
| Code | Name   | Continent | Region         | SurfaceArea | IndepYear | Population |
+------+--------+-----------+----------------+-------------+-----------+------------+
| FRA  | France | Europe    | Western Europe |   551500.00 |       843 |   59225700 |
+------+--------+-----------+----------------+-------------+-----------+------------+

+----------------+------------+----------------+----------------+---------+
| LifeExpectancy | GNP        | GovernmentForm | HeadOfState    | Capital |
+----------------+------------+----------------+----------------+---------+
|           78.8 | 1424285.00 | Republic       | Jacques Chirac |    2974 |
+----------------+------------+----------------+----------------+---------+
```

Deux attributs sont remarquables : l'attribut `Code` (clef primaire de la table) correspond à l'attribut `CountryCode` de la table `city`. Cette remarque est importante car c'est elle qui nous permettra de chercher des informations croisées dans ces deux tables. L'attribut `Capital` est plus intriguant : pourquoi la capitale de la France est-elle désignée par un entier ? Il se trouve que cet attribut correspond à l'attribut `ID` de la table `city`, comme on peut s'en convaincre en consultant l'enregistrement dont l'attribut `ID` vaut `2974` :

```
mysql> select * from city where ID = 2974;
+------+-------+-------------+----------------+------------+
| ID   | Name  | CountryCode | District       | Population |
+------+-------+-------------+----------------+------------+
| 2974 | Paris | FRA         | Île-de-France  |    2125246 |
+------+-------+-------------+----------------+------------+
```

Regardons enfin le contenu de la troisième et dernière table :

```
mysql> describe countryLanguage;
+-------------+---------------+------+-----+
| Field       | Type          | Null | Key |
+-------------+---------------+------+-----+
| CountryCode | char(3)       | NO   | PRI |
| Language    | char(30)      | NO   | PRI |
| IsOfficial  | enum('T','F') | NO   |     |
| Percentage  | decimal(4,1)  | NO   |     |
+-------------+---------------+------+-----+
```

Cette table décrit les langues utilisées dans les différents pays en précisant le pourcentage de locuteurs et si cette langue est officielle ou non. Une même langue peut être parlée dans différents pays, et dans un même pays plusieurs langues peuvent être pratiquées. Ainsi, aucun des attributs ne peut prétendre être une clef primaire, c'est pourquoi nous avons ici un couple d'attributs en guise de clef primaire : le couple `(CountryCode, Language)`.

Ainsi, pour pouvoir efficacement interroger une base de données, il importe de connaître avec précision le contenu de chacune des tables, et les attributs qui vont nous permettre de les relier entre elles. Cet ensemble d'informations est appelé le *schéma de la base de donnée* ; celui de notre exemple est représenté figure 2.

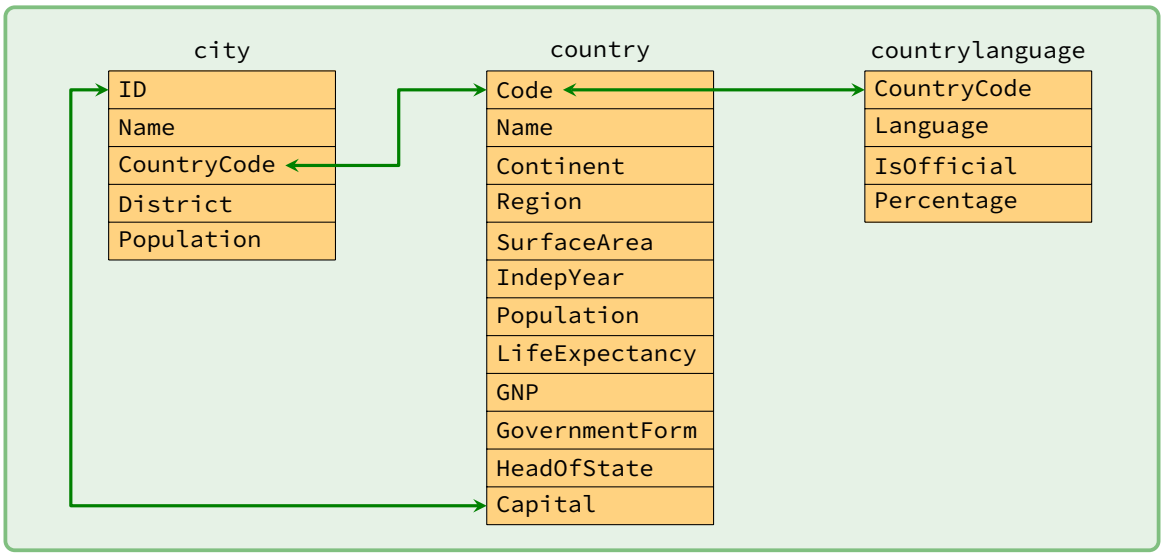


FIGURE 2 – le schéma de la base de données `world`.

2. Requêtes SQL

Nous allons maintenant nous intéresser à la syntaxe des requêtes qui vont nous permettre d'interroger la base de données. Cette syntaxe est proche de la langue anglaise ; en général, la lecture d'une requête permet d'en comprendre le sens. Elle n'en reste pas moins assez rigide dans sa structure, et les mots clefs que nous allons utiliser doivent être rangés dans un ordre bien précis :

SELECT ... **FROM** ... **WHERE** ... **GROUP BY** ... **HAVING** ... **ORDER BY** ... **LIMIT** ... **OFFSET** ...

Seuls les deux premiers (**SELECT** et **FROM**) sont indispensables, les autres sont optionnels, mais s'ils sont présents ils doivent être placés dans cet ordre.
Notons enfin que ces noms de commandes sont insensibles à la casse. Autrement dit, vous pouvez les écrire indifféremment en majuscules ou en minuscule.

2.1 Sélection des attributs et des enregistrements

- On sélectionne un ou plusieurs attributs d'une table par la syntaxe :

```
SELECT a1, a2, ..., an FROM table
SELECT DISTINCT a1, a2, ..., an FROM table
```

Le mot-clef **DISTINCT** permet d'éviter l'apparition de doublons parmi les résultats obtenus.
En algèbre relationnelle, cette opération s'appelle une *projection* et se note $\boxed{\pi_{(A_1,\dots,A_n)}(R)}$ où $A_1,\dots,A_n$ sont les attributs sélectionnés dans la relation R.

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_1	b_1	c_3

$\pi_{(A,B)}(R)$

A	B
a_1	b_1
a_2	b_2

Exemple. La liste des différentes langues présentes dans la base de donnée `world` s'obtient en écrivant :

```
mysql> SELECT DISTINCT Language FROM countrylanguage;
+---------------------------+
| Language                  |
+---------------------------+
| Dutch                     |
| English                   |
| Papiamento                |
| Spanish                   |
.............................
| Bemba                     |
| Chewa                     |
| Lozi                      |
| Nsenga                    |
+---------------------------+
457 rows in set (0,00 sec)
```

457 langues sont présentes dans la base de données.

- On sélectionne les enregistrements d'une table qui satisfont une expression logique par la syntaxe :

```
SELECT * FROM table WHERE expression_logique
```

En algèbre relationnelle cette opération s'appelle une *sélection* et se note $\boxed{\sigma_E(R)}$ où E est l'expression logique et R une relation.

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3
a_4	b_4	c_4

$\sigma_E(R)$

A	B	C
a_2	b_2	c_2
a_4	b_4	c_4

Ces deux opérations peuvent bien entendu être combinées pour extraire de la table une sélection d'attributs et d'enregistrements :

```
SELECT a1, ..., an FROM table WHERE expression_logique
```

ce qui se note en algèbre relationnelle : $\pi_{(A_1,\dots,A_n)}(\sigma_E(R))$.
Par exemple, nous pouvons visualiser les villes françaises et leurs populations respectives en écrivant :

```
mysql> SELECT Name, Population FROM city WHERE CountryCode = 'FRA';
+----------------------+------------+
| Name                 | Population |
+----------------------+------------+
| Paris                |    2125246 |
| Marseille            |     798430 |
| Lyon                 |     445452 |
| Toulouse             |     390350 |
....................................
| Argenteuil           |      93961 |
| Tourcoing            |      93540 |
| Montreuil            |      90674 |
+----------------------+------------+
40 rows in set (0,00 sec)
```

- Le *renommage* permet la modification du nom d'un attribut d'une relation. Renommer l'attribut a en l'attribut b dans la relation R s'écrit $\boxed{\rho_{a\leftarrow b}(\mathrm{R})}$ en algèbre relationnelle et à l'aide du mot-clef **AS** en SQL :

```
SELECT a AS b FROM table
```

  La nécessité du renommage apparaîtra lorsque nous aborderons les sous-requêtes.

Filtrage des résultats

À la toute fin d'une requête il est possible de trier les résultats et de n'en renvoyer qu'une partie. Ces opérations se notent :

- **ORDER BY** *a* **ASC** / **DESC** pour trier suivant l'attribut a par ordre croissant/décroissant ;
- **LIMIT** n pour limiter la sortie à n enregistrements ;
- **OFFSET** n pour débuter à partir du n^{e} enregistrement.

Par exemple, les cinq villes mondiales les plus peuplées sont :

```
mysql> SELECT Name FROM city ORDER BY Population DESC LIMIT 5;
+-----------------+
| Name            |
+-----------------+
| Mumbai (Bombay) |
| Seoul           |
| São Paulo       |
| Shanghai        |
| Jakarta         |
+-----------------+
```

et les cinq suivantes sont :

```
mysql> SELECT Name FROM city ORDER BY Population DESC LIMIT 5 OFFSET 5;
+-------------------+
| Name              |
+-------------------+
| Karachi           |
| Istanbul          |
| Ciudad de México  |
| Moscow            |
| New York          |
+-------------------+
```

Exercice 1

a. Rédiger une requête SQL donnant le nom des dix pays asiatiques les plus grands.

b. Rédiger une requête SQL donnant le nom et la date d'indépendance des dix pays les plus anciens.

c. Rédiger une requête SQL donnant la liste des langues non officielles pratiquées en France, par ordre décroissant d'importance.

d. Rédiger une requête SQL donnant le nom des cinq pays européens les moins densément peuplés.

2.2 Jointures et sous-requêtes

La *jointure* est une opération qui porte sur deux relations R_1 et R_2 et retourne une relation qui comporte les enregistrements combinés de R_1 et de R_2 qui satisfont une contrainte logique E. Cette nouvelle relation se note $\boxed{R_1 \bowtie_E R_2}$.
En SQL on réalise une jointure par la requête :

```
SELECT * FROM table1 JOIN table2 ON expression_logique
```

R_1

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3

R_2

D	E
a_1	e_1
a_2	e_2
a_2	e_3

$R_1 \bowtie_{A=D} R_2$

A	B	C	D	E
a_1	b_1	c_1	a_1	e_1
a_2	b_2	c_2	a_2	e_2
a_2	b_2	c_2	a_2	e_3

Seules les conditions d'égalité entre deux attributs figurent au programme.

Remarque. Deux tables reliées par une jointure peuvent posséder des attributs de même nom mais n'ayant rien à voir. C'est le cas par exemple de l'attribut `Name` présent dans les deux tables `city` et `country`. Pour les distinguer lors d'une jointure il est nécessaire de faire précéder le nom de l'attribut par le nom de la table, séparé par un point. Par exemple, lors d'une jointure entre les deux tables de notre base de données exemple, le nom des villes est désigné par `city.name` et celui des pays par `country.Name`.
Notons que le mot-clef **AS** permet de renommer les tables ainsi que les attributs afin d'éviter d'écrire des noms à rallonge.

Exemple. Les deux tables `city` et `country` possèdent deux jointures naturelles :

- on peut identifier les attributs `city.countryCode` et `country.Code`. Dans ce cas, la table résultante de la jointure possédera autant d'entrées que la table `city` et permettra de relier à chaque ville de la base les informations du pays auquel elle appartient ;
- on peut identifier les attributs `city.ID` et `country.Capital`. Dans ce cas, la table résultante de la jointure possédera autant d'entrées que la table `country` et à chaque pays sera attaché les informations relatives à sa capitale. En revanche, les villes qui ne sont pas des capitales ne seront pas présentes dans cette jointure.

Par exemple, si on souhaite connaître la capitale de l'Ouzbekistan et son nombre d'habitants on écrira :

```
mysql> SELECT city.Name, city.Population
    -> FROM city JOIN country ON city.ID = country.Capital
    -> WHERE country.Name = 'Uzbekistan';
+---------+------------+
| Name    | Population |
+---------+------------+
| Toskent |    2117500 |
+---------+------------+
```

Exercice 2

a. Rédiger une requête SQL donnant la liste des pays ayant adopté le Français parmi leurs langues officielles.

b. Rédiger une requête SQL donnant les cinq villes les plus peuplées d'Europe.

c. Rédiger une requête SQL donnant les cinq villes les plus peuplées d'Europe parmi celles qui ne sont pas des capitales.

d. Rédiger une requête SQL donnant les capitales des pays dans lesquels l'allemand est langue officielle.

■ Sous-requêtes

Revenons au problème consistant à déterminer la capitale de L'Ouzbekistan. Nous l'avons résolu à l'aide d'une jointure, mais il aurait aussi été possible de procéder à deux requêtes successives : une première requête pour déterminer l'identifiant de la capitale de l'Ouzbekistan, une seconde pour connaître le nom de cette ville.
Il est possible d'imbriquer la première au sein de la seconde pour n'en faire qu'une seule ; on parle alors de sous-requête. Celle-ci doit impérativement être délimitée par des parenthèses, et peut être située :

- en lieu et place d'une table après le **FROM** ;
- ou au sein d'une expression logique après le **WHERE**.

Par exemple, la détermination de la capitale de l'Ouzbekistan peut être obtenue par la requête :

```
mysql> SELECT city.Name FROM city
    -> WHERE ID = (SELECT Capital FROM country WHERE name = 'Uzbekistan');
+---------+
| Name    |
+---------+
| Toskent |
+---------+
```

Exercice 3

Rédiger une requête SQL donnant la liste des pays dont l'espérance de vie est supérieure ou égale à celle de la France.

Notons enfin que lorsqu'une sous-requête prend la place d'une table, il est impératif de renommer les attributs de la sous-requête pour qu'ils soient utilisés dans la requête principale.

2.3 Fonctions d'agrégation

SQL possède un certain nombre de fonctions statistiques (voir la liste figure 3) qui par défaut s'appliquent à l'ensemble des enregistrements sélectionnés par la clause du **WHERE**.

COUNT( )	nombre d'enregistrements
MAX( )	valeur maximale d'un attribut
MIN( )	valeur minimale d'un attribut
SUM( )	somme d'un attribut
AVG( )	moyenne d'un attribut

Figure 3 – Fonctions statistiques.

Par exemple, pour obtenir la population de l'Europe on pourrait écrire :

```
mysql> SELECT SUM(Population) FROM country WHERE Continent = 'Europe';
+-----------------+
| SUM(Population) |
+-----------------+
|       730074600 |
+-----------------+
```

Exercice 4

Rédiger une requête SQL déterminant la ville la moins peuplée de la base de données.

Mais il est aussi possible de regrouper les enregistrements d'une table par *agrégation* à l'aide du mot-clef **GROUP BY**. Ce regroupement permet d'appliquer la fonction statistique à chacun des groupes : le résultat de la requête est l'ensemble des valeurs prises par la fonction statistique sur chacun des regroupements.

R

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_3	c_3
a_3	b_4	c_4
a_3	b_5	c_5

group by A ⟶

R

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_3	c_3
a_3	b_4	c_4
a_3	b_5	c_5

On utilise la syntaxe suivante (dans laquelle a_1 et a_2 sont des attributs et f une fonction d'agrégation) :

```
SELECT f(a1) FROM table WHERE expression_logique GROUP BY a2
```

Cette requête sélectionne les enregistrements de la table qui vérifient l'expression logique, les regroupe selon la valeur de l'attribut a_2, puis applique la fonction f sur l'attribut a_1 à chacun des groupes obtenus.

Notons enfin que le mot-clef **HAVING** permet d'imposer des conditions sur les groupes à qui on applique la fonction d'agrégation. La syntaxe générale de la requête prend alors la forme suivante :

```
SELECT f(a1) FROM table WHERE e1 GROUP BY a2 HAVING e2
```

Cette requête sélectionne les enregistrements de la table qui vérifient l'expression logique e_1, les regroupe selon la valeur de l'attribut a_2, puis applique la fonction f sur l'attribut a_1 à chacun des groupes vérifiant l'expression logique e_2.
Par exemple, pour obtenir les populations de chacun des continents on écrira :

```
mysql> SELECT Continent, SUM(Population) FROM country
    -> GROUP BY Continent;
+---------------+------------------+
| Continent     | SUM(Population) |
+---------------+------------------+
| North America |       482993000 |
| Asia          |      3705025700 |
| Africa        |       784475000 |
| Europe        |       730074600 |
| South America |       345780000 |
| Oceania       |        30401150 |
| Antarctica    |               0 |
+---------------+------------------+
```

Si on veut ordonner cette liste, il est nécessaire de renommer l'attribut calculant la population totale de chaque continent :

```
mysql> SELECT Continent, SUM(Population) AS Pop FROM country
    -> GROUP BY Continent ORDER BY Pop DESC;
+---------------+------------+
| Continent     | Pop        |
+---------------+------------+
| Asia          | 3705025700 |
| Africa        |  784475000 |
| Europe        |  730074600 |
| North America |  482993000 |
| South America |  345780000 |
| Oceania       |   30401150 |
| Antarctica    |          0 |
+---------------+------------+
```

Enfin, si on veut se restreindre aux continents qui comportent plus de 40 pays, on écrira :

```
mysql> SELECT Continent, SUM(Population), COUNT(*) FROM country
    -> GROUP BY Continent HAVING COUNT(*) > 40;
+-----------+-----------------+----------+
| Continent | SUM(Population) | COUNT(*) |
+-----------+-----------------+----------+
| Asia      |      3705025700 |       51 |
| Africa    |       784475000 |       58 |
| Europe    |       730074600 |       46 |
+-----------+-----------------+----------+
```

Exercice 5

a. Rédiger une requête SQL donnant la liste des districts des États-Unis dont la population totale est supérieure à 3 000 000.

b. Rédiger une requête SQL déterminant les cinq langues les plus parlées au monde.

c. Rédiger une requête SQL déterminant, pour chaque continent, le pays le plus peuplé.

2.4 Opérateurs ensemblistes

On peut procéder à des opérations ensemblistes entre deux tables (en général le résultat de requêtes) à condition que celles-ci aient même structure. Ces opérations sont :

- **UNION** pour calculer la réunion $A \cup B$ de deux requêtes ;
- **INTERSECT** pour calculer l'intersection $A \cap B$ de deux requêtes ;
- **EXCEPT** pour calculer la différence $A \setminus (A \cap B)$ de deux requêtes.

Ces opérations présentent surtout un intérêt lorsque les deux requêtes sont issues de deux tables différentes. MySQL (ainsi que d'autres SGBD) n'implémente d'ailleurs que l'union, les deux autres opérations ensemblistes pouvant être aisément remplacées par des requêtes équivalentes :

- **SELECT** a **FROM** table1 **INTERSECT SELECT** b **FROM** table2 est équivalent à

```
SELECT a FROM table1 WHERE a IN (SELECT b FROM table2)
```

- **SELECT** a **FROM** table1 **EXCEPT SELECT** b **FROM** table2 est équivalent à

```
SELECT a FROM table1 WHERE a NOT IN (SELECT b FROM table2)
```

Autrement dit, seule l'union présente un réel intérêt.

Il est enfin possible de réaliser le produit cartésien de deux tables (ou plus) en suivant la syntaxe

```
SELECT A1, A2 FROM table1, table2
```

mais la création d'un produit cartésien doit être réalisé avec prudence, car le cardinal du résultat est égal au produit des cardinaux des tables qui le composent :

```
mysql> select count(*) from city
    -> union select count(*) from country
    -> union select count(*) from countrylanguage
    -> union select count(*) from city, country, countrylanguage;
+-----------+
| count(*)  |
+-----------+
|      4079 |
|       239 |
|       984 |
| 959282904 |
+-----------+
```

3. Exercices

On souhaite utiliser une base de données pour stocker les résultats obtenus par une communauté de joueurs en ligne. On suppose qu'on dispose d'une base de données comportant deux tables : `Joueurs(id_j, nom, pays)` et `Parties(id_p, date, duree, score, id_joueur)` où :

- `id_j` de type entier, est la clef primaire de la table `Joueurs` ;
- `nom` est une chaîne de caractères donnant le nom du joueur ;
- `pays` est une chaîne de caractères donnant le pays du joueur ;
- `id_p` de type entier, est la clef primaire de la table `Parties` ;
- `date` est la date (AAAAMMJJ) de la partie ;
- `duree` de type entier, est la durée en secondes de la partie ;
- `score` de type entier, est le nombre de points marqués au cours de la partie ;
- `id_joueur` est un entier qui identifie le joueur de la partie.

Exercice 6

Rédiger une requête SQL qui renvoie la date, la durée et le score de toutes les parties jouées par Alice, listées par ordre chronologique.

Exercice 7

Alice vient de réaliser un score de n points. Rédiger une requête SQL qui renvoie la position qu'aura le score n dans le classement des parties par ordre de score (on suppose que la partie que vient de jouer Alice n'est pas encore insérée dans la base de données). En cas d'ex aequo pour le score n le rang sera le même pour tous les joueurs ayant ce score.

Exercice 8

Rédiger une requête SQL qui renvoie le record de France pour ce jeu.

Exercice 9

Rédiger une requête SQL qui renvoie le rang d'Alice, c'est-à-dire sa position dans le classement des joueurs par ordre de leur meilleur score.