RÉVISIONS

B. Landelle

Table des matières

Les modules scientifique et graphique seront importés avec leurs alias habituels :

```
import numpy as np, matplotlib.pyplot as plt
```

On notera log la fonction logarithme népérien et $\log_2$ le logarithme en base 2 défini par

$$\forall x > 0 \qquad \log_2(x) = \frac{\log x}{\log 2}$$

I Types simples, variables et types composés

1 Les types simples

• Les entiers

```
>>> type(1)
<class 'int'>
>>> 1+2
3
>>> 2**3
8
>>> 7%2
1
>>> 7//3
2
>>>
```

• Les booléens

```
>>> type(True)
<class 'bool'>
>>> not True
False
>>> True or False
True
>>> 1>2
False
```

Un comportement intéressant : l'évaluation paresseuse :

```
>>> True or 1/0==1
True
```

Python confond `1` et `True`, `0` et `False` :

```
>>> 1==True
True
>>> id(1)
1451996848
>>> id(True)
1451639424
```

Exercice : Soit `L` une liste de nombres. Écrire une instruction qui détermine la proportion de ceux inférieurs ou égaux à 1. On autorise l'utilisation de la fonction `np.mean` qui effectue un calcul de moyenne.

Corrigé : On saisit :

```
>>> np.mean([x<=1 for x in L])
```

• **Les flottants**

```
>>> type(1.5)
<class 'float'>
>>> 4e-3
0.004
```

Avec le module `numpy` :

```
>>> np.floor(2.7)
2.0
>>> np.sqrt(2)
1.4142135623730951
>>> np.sin(np.pi)
1.2246467991473532e-16
```

• **Les complexes**

```
>>> type(1j)
<class 'complex'>
>>> 1j**2
(-1+0j)
>>> np.sqrt(-1)     # dans R
nan
>>> np.sqrt(-1+0j)  # dans C
1j
>>> abs(1+1j)
1.4142135623730951
```

Exercice : Que renvoient les instructions suivantes :

```
>>> type(0)
>>> type(0.)
>>> 0.==0
>>> 1.+True
>>> int(-.5)==np.floor(-.5)
```

Corrigé :

```
>>> type(0)
<class 'int'>
>>> type(0.)
<class 'float'>
>>> 0.==0
True
>>> 1.+True
2.0
>>> int(-.5)==np.floor(-.5)
False
```

2 Les variables

Les variables créées sous python sont des étiquettes qui désignent des *objets* sur lesquels on peut agir avec des fonctions ou des *méthodes*. Les méthodes d'un objet désigné par une variable `var` s'utilisent selon la syntaxe

```
var.methode(arguments)
```

Une méthode peut modifier le contenu de la variable.

Création :

```
>>> a=1
>>> type(a)
<class 'int'>
```

Création simultanée :

```
>>> a,b=1,2
>>> a
1
>>> b
2
>>> a,b
(1, 2)
```

Échange de variables :

```
>>> a,b=b,a
>>> a,b
(2, 1)
```

Opérations/affectations simultanées :

```
>>> a=10
>>> a+=1
>>> a
```

```
11
>>> a//=2
>>> a
5
```

Exercice : Que renvoient les instructions :

```
>>> a,b=1,2
>>> a,b=(a+b)/2,np.sqrt(a*b)
>>> a,b
```

Corrigé :

```
>>> a,b
(1.5, 1.4142135623730951)
```

Quelques méthodes sur les complexes :

```
>>> a=1+2j
>>> a.real
1.0
>>> a.imag
2.0
>>> a.conjugate()
(1-2j)
```

3 Les types composés

• Les chaînes de caractères (non mutables)

```
>>> a="bonjour"
>>> type(a)
<class 'str'>
>>> len(a)
7
>>> a[0]="1"
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    a[0]="1"
TypeError: 'str' object does not support item assignment
>>> b=" bonsoir"
>>> a+b
'bonjour bonsoir'
>>> a[0]
'b'
>>> a[-1]
'r'
>>> "bon" in "bonjour"
True
```

Du slicing :

```
>>> a[1:4]
'onj'
>>> a[:4]
'bonj'
>>> a[4:]
'our'
>>> a[::-1]
'ruojnob'
```

Exercice : Soit `mot` une chaîne de caractères. Écrire une instruction dont le résultat est `True` si `mot` est un palindrome et `False` sinon.

Corrigé :

```
>>> mot==mot[::-1]
```

- **Les listes**

```
>>> a=[1,"bonjour",True]
>>> type(a)
<class 'list'>
>>> id(a)
84754504
```

Les listes sont mutables :

```
>>> a[1]=3.
>>> a
[1, 3.0, True]
>>> id(a)
84754504
>>> a.append(2)          # méthode d'ajout en fin de liste
>>> a
[1, 3.0, True, 2]
>>> a.pop()              #  méthode de suppression en fin de liste
2
>>> a
[1, 3.0, True]
>>> id(a)
84754504
```

Ordonnancement d'une liste :

```
>>> b=[3,2,1,5]
>>> sorted(b)
[1, 2, 3, 5]
```

```
>>> b
[3, 2, 1, 5]
>>> b.sort()           # la méthode sort modifie b
>>> b
[1, 2, 3, 5]
```

Conséquences du caractère mutable pas anodines :

```
>>> a=[1,2,4]
>>> b=a                # b et a sont "colocataires"
>>> b[2]=3
>>> b
[1, 2, 3]
>>> a
[1, 2, 3]
>>> id(a)
67457160
>>> id(b)
67457160
```

Pour des copies indépendantes :

```
>>> b=list(a)
>>> c=a[:]
>>> id(a)
67457160
>>> id(b)
67459336
>>> id(c)
67537160
```

Pour copier des listes contenant des sous-listes en évitant le problème précédemment rencontré, il faut une copie en profondeur :

```
>>> from copy import deepcopy
>>> a=[[1],2]
>>> b=deepcopy(a)
>>> id(a)
90034056
>>> id(b)
90033864
>>> id(a[0])
89938056
>>> id(b[0])
89970056
```

Exercice : L'instruction `list()` convertit une chaîne de caractères en liste de caractères. Écrire des instructions qui permettent de tester si deux chaînes de caractères `mot1` et `mot2` sont anagrammes l'une de l'autre.

Corrigé :

```
>>> L1=list(mot1)
>>> L2=list(mot2)
>>> sorted(L1)==sorted(L2)
```

• **Les ranges (non mutables)**

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2,20,2))
[2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> list(range(10,0,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

• **Les tuples (non mutables)**

```
>>> a=1,2
>>> type(a)
<class 'tuple'>
```

• **Les dictionnaires**

```
>>> dico={}
>>> type(dico)
<class 'dict'>
>>> dico["chat"]="cat"
>>> dico[0]=False
>>> dico["SPE"]=["MP","PC"]
>>> dico
{'chat': 'cat', 0: False, 'SPE': ['MP', 'PC']}
>>> dico["SPE"].append("PSI")
>>> dico
{'chat': 'cat', 0: False, 'SPE': ['MP', 'PC', 'PSI']}
>>> "SUP" in dico
False
```

Les dictionnaires sont mutables :

```
>>> a=dico
>>> a[1]=True
>>> a
{'chat': 'cat', 0: False, 'SPE': ['MP', 'PC', 'PSI'], 1: True}
>>> dico
{'chat': 'cat', 0: False, 'SPE': ['MP', 'PC', 'PSI'], 1: True}
```

II Représentation des nombres

1 Les entiers

Un entier n codé sur p bits a pour écriture binaire $n = \sum_{k=0}^{p-1} d_k 2^k$ avec $d_k \in \{0,1\}$. On note

$$n = \langle d_{p-1}, \dots, d_1, d_0 \rangle_2$$

Un entier n non nul possède une unique écriture binaire de la forme $n = \sum_{k=0}^{p-1} d_k 2^k$ avec $d_k \in \{0,1\}$ pour $k < p-1$ et $d_{p-1} = 1$. Le bit d_{p-1} est dit *bit de poids fort* et d_0 *bit de poids faible*. On a

$$2^{p-1} \leqslant n \leqslant \sum_{k=0}^{p-1} 2^k = 2^p - 1 \quad \Longrightarrow \quad p = \lfloor \log_2(n) \rfloor + 1$$

• Méthode de complément à deux

Les entiers relatifs codés sur p bits le sont avec la méthode de *complément à deux* :

$$n = -d_{p-1} 2^{p-1} + \sum_{k=0}^{p-2} d_k 2^k$$

Ce codage est compatible avec l'algorithme usuel d'addition. Un entier négatif sera codé par $\langle 1, d_{p-2}, \dots, d_0 \rangle_2$.

En python, depuis la version 3, les subtilités des formats d'entiers sont invisibles à l'utilisateur : les entiers positifs, négatifs, grands ou petits en valeur absolue sont tous codés par le type `int` :

```
>>> type(1)
<class 'int'>
>>> type(2**10000)
<class 'int'>
>>> type(-2**10000)
<class 'int'>
```

• Algorithme de Horner

Soit x un réel et $\mathrm{P} = \sum_{k=0}^{n-1} a_k \mathrm{X}^k \in \mathbb{R}[\mathrm{X}]$. Pour calculer efficacement $\mathrm{P}(x)$, on peut observer l'écriture suivante :

$$\mathrm{P}(x) = \sum_{k=0}^{n-1} a_k x^k = (\dots((a_{n-1}x + a_{n-2})\,x + a_{n-3})\,x + \dots)\,x + a_0$$

Application : Calcul d'un nombre à partir de son écriture binaire
Connaissant l'écriture binaire d'un nombre $\langle d_{p-1}, \dots, d_0 \rangle$, on obtient sa valeur en évaluant le polynôme $\sum_{k=0}^{p-1} d_k \mathrm{X}^k$ en 2.

2 Les flottants

Les flottant sont codés selon la norme IEEE 754 et sont de la forme

$$(-1)^s \times \mathrm{M} \times 2^{\mathrm{E}-1023} \quad \text{avec} \quad s \in \{0,1\}, \qquad \mathrm{M} = 1 + \sum_{i=1}^{52} \frac{m_i}{2^i} \qquad \mathrm{E} = \sum_{i=0}^{10} e_i 2^i$$

L'intérêt de cette norme est de pouvoir coder une grande amplitude de nombres. Le bit s sert au codage du signe, le nombre M codé sur 52 bits est appelé *mantisse* et l'entier E est appelé

exposant codé sur 11 bits.

Il y a plusieurs limitations au format flottant : pour la plupart, les nombres ne sont pas codés fidèlement, les limites de précision peuvent donner lieu à des phénomènes d'absorption, le format est borné, etc....

```
>>> .1+.2
0.30000000000000004
>>> 1+2**(-53)-1==0
True
>>> float(2**1023)
8.98846567431158e+307
>>> float(2**1024)
Traceback (most recent call last):
  File "<pyshell#123>", line 1, in <module>
    float(2**1024)
OverflowError: int too large to convert to float
```

 Tester la nullité de `x` nombre flottant par `x==0` n'est pas pertinent : on utilisera un test de la forme `abs(x)<eps` avec `eps` une seuil de précision choisi par l'utilisateur.

III Fonctions et programmes

1 Fonctions

Les arguments et variables créées dans une fonction sont locales (l'instruction `locals()` fournit la liste) et ne vivent que dans le corps de la fonction. On recommande, sauf dans les fonctions conçues *en place*, de ne pas modifier les arguments d'une fonction.

Il existe deux syntaxes en python :

Syntaxe standard

```
def nom_fonction(arguments):
    Instructions
    return résultat
```

On notera la présence de l'indentation pour le *corps* de la fonction. Cette indentation est indispensable. Le retour à l'indentation de `def` détermine la fin du corps de la fonction. L'instruction `return` provoque la sortie de la fonction et renvoie le résultat.

Syntaxe courte

```
nom_fonction=lambda arguments : résultat
```

Différentes manières de coder $x \mapsto x^2$:

```
def sq(x):
    return x*x
```

```
sq=lambda x:x*x
```

Exercice : Que fait la fonction

```
def test(n):
    return n%2==0
```

Corrigé : La fonction `test()` répond à la question de la parité de l'argument.

2 Boucles inconditionnelles, conditionnelles

• Boucle inconditionnelle

```
for Enumeration:
    Instructions
```

Somme des éléments d'une liste de nombres :

```
def somme(L):
    res=0
    for x in L:
        res+=x
    return res
```

On observe qu'on peut directement parcourir la liste.

Remarque : L'instruction native `sum()` fait de même ...

Implémentation de l'algorithme de Horner :

```
def poly(x,P):
    """Calcul de P(x) suivant l'algorithme de Horner
    x : flottant
    P : [a_0, ..., a_{n-1}] liste de flottants"""
    res=0
    n=len(P)
    for k in range(n-1,-1,-1):
        res=x*res+P[k]
    return res
```

On peut casser une boucle `for` avec un `return`. Pour tester la présence d'un élément dans une liste, on peut coder :

```
def detect(L,elt):
    for x in L:
        if x==elt:
            return True
    return False
```

Le fait de casser la boucle `for` rend l'étude de complexité plus opaque mais c'est tellement simple de coder ainsi qu'il serait absurde de s'en priver.

Remarque : L'instruction `x in L` fournit le même résultat.

 **Quelques interdits :** On s'interdira totalement de modifier un ensemble que l'on parcourt dans une boucle `for` : pas de `append`, ni `pop`, ni `remove`, ni `del` ... Les exemples qui suivent sont à proscrire absolument

```
L=[]
for x in L:
    L.append(0) # Boucle infinie!
```

ou

```
L=list(range(10))
res=[]
for x in L:
    res.append(x)
    L.pop()
```

avec en la variable `res` qui contient en sortie de boucle `[0, 1, 2, 3, 4]`.

Un classique : Calcul d'un coefficient binomial
En observant la décomposition

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \begin{cases} \dfrac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1} & \text{si } k > 0 \\ 1 & \text{si } k = 0 \end{cases}$$

on en déduit une manière efficace pour coder $\binom{n}{k}$:

```
def binom(n,k):
    res=1
    for i in range(k):
        res=(n-i)*res//(i+1)
    return res
```

Exercice : Peut-on améliorer ce code ?

Corrigé : Avec l'observation $\binom{n}{k} = \binom{n}{n-k}$, on obtient

```
def binom(n,k):
    p=min(k,n-k)
    res=1
    for i in range(p):
        res=(n-i)*res//(i+1)
    return res
```

Pour coder une suite $(u_n)_n$ définie par la relation de récurrence $u_{n+1} = f(u_n)$, on saisit :

```
def suite(f,u0,n):
    u=u0
    for k in range(n):
        u=f(u)
    return u
```

• **Boucle conditionnelle**

```
while Condition:
    Instructions
```

```
def pgcd(a,b):
    u,v=a,b
    while v!=0:
        u,v=v,u%v
    return u
```

Exercice : On observe le comportement suivant :

```
>>> a,b=1,1
>>> id(a)==id(b)
True
>>> a*=2
>>> b*=2
>>> id(a)==id(b)
True
>>> a*=1000
>>> b*=1000
>>> id(a)==id(b)
False
```

Écrire un programme qui détermine la plus grande puissance de 2 pour laquelle deux entiers ont toujours même identifiant.

Corrigé : On saisit :

```
def plage_int():
    a,b=1,1
    while id(a)==id(b):
        a*=2
        b*=2
    return a//2
```

On obtient :

```
>>> plage_int()
256
```

3 Les tests

• Test simple

```
if Condition:
    Instructions
```

Fonction qui renvoie le nombre d'occurrences d'un élément dans une liste :

```
def nb_occ(L,elt):
    res=0
    for x in L:
        if x==elt:
            res+=1
    return res
```

Rermarque : L'instruction `L.count(elt)` fournit le même résultat.

• Test avec alternative

```
if Condition:
    Instructions
else:
    Instructions
```

•Test à alternatives multiples

```
if Condition 1:
    Instructions
elif Condition 2:
    Instructions
...
elif Condition n:
    Instructions
else:
    Instructions
```

Exercice : Écrire une fonction qui détermine les indices de toutes les occurrences d'un élément dans une liste.

Corrigé : On saisit :

```
def rech_occ(L,elt):
    res=[]
    for k in range(len(L)):
        if L[k]==elt:
            res.append(k)
    return res
```

Exercice : Écrire une fonction `double(a,b,c)` qui renvoie `True` si $aX^2 + bX + c$ admet une racine double et `False` sinon.

Corrigé : On saisit :

```
def double(a,b,c):
    eps=1e-10
    return abs(b**2-4*a*c)<eps
```

IV Manipulation de fichiers

1 Les instructions

Pour manipuler les fichiers, on utilise :
- l'instruction `open` pour l'ouverture ou la création ;
- la méthode `close` pour la fermeture ;
- la méthode `read` pour la lecture intégrale ;
- la méthode `readline` pour lire la ligne courante ;
- la méthode `write` pour l'écriture.

Pour la lecture et l'écriture dans un fichier, on recommande de travailler sur des fichiers `*.csv` avec le caractère `';'` comme séparateur. Pour lire dans un fichier `file.csv`, on utilise :

```
data=open('file.csv')   # ouverture du fichier dans la variable data
contenu=data.read()     # contenu reçoit l'intégralité de data, format str
ligne=data.readline()   # ligne reçoit la ligne courante, format str
data.close()            # fermeture du fichier
```

La lecture de l'intégralité du fichier positionne le curseur de lecture en fin de fichier. La lecture d'une ligne du fichier positionne le curseur sur la ligne suivante ou la fin du ficher le cas échéant. Pour écrire dans un fichier, on l'ouvre avec l'option d'écriture `'w'` et on utilise :

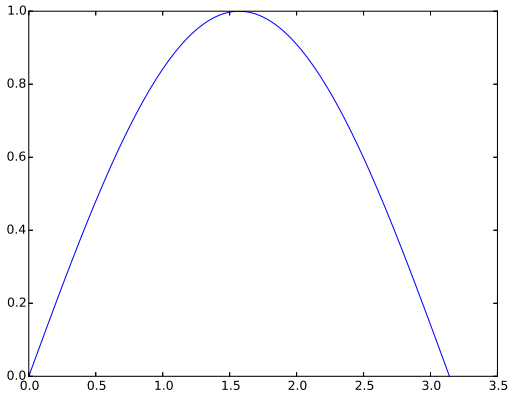
```
data=open('file.csv','w') # option 'w'=write pour écriture
data.write('blabla\n')    # écriture d'une ligne 'blabla' dans data
data.close()
```

Le caractère spécial `'\n'` correspond à un saut de ligne.

2 Lecture dans un fichier

Le fichier `pts.csv` présent dans le répertoire de travail contient une suite de coordonnées de points que l'on souhaite tracer. Les points sont stockés ligne par ligne et le caractère ';' sépare abscisse et ordonnée.

0.0	0.0
0.0317332591272	0.0317279334981
0.0634665182543	0.0634239196566
0.0951997773815	0.0950560433042
0.126933036509	0.126592453574
0.158666295636	0.158001395973
...	...



On peut parcourir directement le fichier ligne par ligne avec une simple boucle `for`. Chaque ligne est lue comme chaîne de caractères.

```
points=open('pts.csv')
tx,ty=[],[]
for coords in points:            # lecture ligne par ligne
    xy=coords.split(";")         # coupe la chaine "x;y" en ("x","y")
    tx.append(float(xy[0]))
    ty.append(float(xy[1]))
points.close()
plt.plot(tx,ty);plt.show()
```

3 Écriture dans un fichier

Pour sauvegarder des données calculées sous python dans un fichier, on ouvre celui-ci avec l'option `'w'` pour autoriser l'écriture. Ainsi, le fichier `pts.csv` considéré précédemment a été généré par le code suivant :

```
fichier=open('pts.csv','w')    # option 'w'=write pour écriture
tt=np.linspace(0,np.pi,100)
ts=np.sin(tt)
for k in range(len(tt)):
    fichier.write(str(tt[k])+";"+str(ts[k])+"\n")   # ';' comme séparateur
fichier.close()
```

V Complexité

1 Définitions

La *complexité temporelle* d'un algorithme désigne le nombre d'opérations élémentaires réalisées par l'algorithme. La *complexité spatiale* d'un algorithme désigne l'espace mémoire occupé lors de l'exécution de l'algorithme.

La complexité temporelle *dans le pire cas* est le temps d'exécution maximum. La complexité temporelle *dans le meilleur cas* est le temps d'exécution minimum.

La complexité spatiale *dans le pire cas* est l'espace mémoire occupé maximum. La complexité spatiale *dans le meilleur cas* est l'espace mémoire occupé minimum.

On utilise dans le cadre du programme la relation de *domination* $u_n = \mathrm{O}(v_n)$, la suite $(u_n)_n$ est dominée par la suite $(v_n)_n$, qui signifie :

$$\exists n_0 \in \mathbb{N} \quad \exists \mathrm{M} > 0 \qquad \forall n \geqslant n_0 \qquad |u_n| \leqslant \mathrm{M}\,|v_n|$$

Théorème 1. *Soit $(u_n)_n$ une suite de réels strictement positifs. On a*

$$\sum_{k=1}^{n} O(u_k) = O\left(\sum_{k=1}^{n} u_k\right)$$

Remarque : La bonne notion pour l'étude de complexité n'est pas en réalité la relation de domination. Effet, si un algorithme a une complexité temporelle en $\mathrm{O}(n)$, on peut tout aussi bien annoncer qu'elle est en $\mathrm{O}(n^2)$. C'est correct même si c'est nettement moins pertinent. L'usage consiste donc à déterminer la domination la plus fine qui soit ce qui équivaut à utiliser la relation hors-programme *de l'ordre de* notée Θ. On note $u_n = \Theta(v_n)$ ce qui signifie $u_n = \mathrm{O}(v_n)$ et $v_n = \mathrm{O}(u_n)$.

2 Classes de complexité

L'entier n désigne en général la taille de l'argument. Dans le cas d'un algorithme portant sur un calcul arithmétique, il peut aussi désigner l'argument lui-même.

On distingue les principales classes de complexité suivantes :
- $\mathrm{O}(1)$: complexité constante ;
- $\mathrm{O}(\log n)$: complexité logarithmique ;
- $\mathrm{O}(\sqrt{n})$: complexité racinaire ;
- $\mathrm{O}(n)$: complexité linéaire ;
- $\mathrm{O}(n \log n)$: complexité quasi-linéaire ;
- $\mathrm{O}(n^2)$: complexité quadratique ;
- $\mathrm{O}(n^p)$: complexité polynomiale ;
- $\mathrm{O}(a^n)$ avec $a > 1$: complexité exponentielle.

Les complexités au plus quasi-linéaires sont raisonnables. Les complexités polynomiales, au moins quadratiques, sont acceptables mais les complexités exponentielles sont à proscrire.

⚠ Les temps d'accès à un élément d'une liste en lecture/écriture sont en $\mathrm{O}(1)$. Cette caractéristique combinée à une structure dynamique est un des atouts majeurs des listes en python.

Complexité temporelle des instructions ou méthodes sur les listes :

Opération	Complexité
`append`	$O(1)^*$
`pop`	$O(1)^*$
`==, !=`	$O(n)$
`in`	$O(n)$
`remove`	$O(n)$
`delete`	$O(n)$
`count`	$O(n)$
`max, min`	$O(n)$
`reverse`	$O(n)$
`sort`	$O(n \log n)$

(*) : il s'agit de *complexité amortie* (complexité moyenne en utilisation).

⚠ Les tests d'appartenance et les temps d'accès en lecture/écriture à une couple (clé,valeur) d'un dictionnaire sont en $O(1)$.

La complexité spatiale de l'instruction `range(n)` est en $O(1)$ tandis que celle de `list(range(n))` est en $O(n)$. Pour s'en convaincre, on exécute :

```
from sys import *

tn=range(100)
tR=[getsizeof(range(n)) for n in tn]
tL=[getsizeof(list(range(n))) for n in tn]
plt.plot(tn,tR,tn,tL)
plt.title("Complexité spatiale de range(n) et list(range(n))")
plt.grid();plt.show()
```

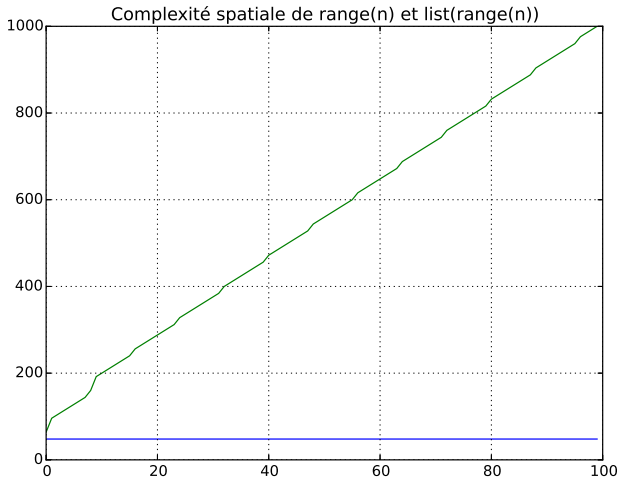


FIGURE 1 – Comparaison des complexités spatiales

3 Exemples

- **Recherche d'un doublon dans une liste**

```
def doublon(L):
    n=len(L)
    for i in range(n-1):
        for j in range(i+1,n):
            if L[i]==L[j]:
                return True
    return False
```

La complexité est en O(1) dans le meilleur des cas (deux doublons en première et deuxième position). Dans le pire des cas, s'il n'y a pas de doublons, on effectue $\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \mathrm{O}(n^2)$ répétitions.

Exercice : Discuter de la complexité de la recherche de doublon dans une liste triée.

Corrigé :

```
def doublon_tri(L):
    for k in range(len(L)-1):
        if L[k]==L[k+1]:
            return True
    return False
```

La complexité est en O(1) dans le meilleur des cas et en $\mathrm{O}(n)$ dans le pire des cas. On verra que le coût d'un tri est en $\mathrm{O}(n\log n)$ ce qui fait dont un coût total en $\mathrm{O}(n)+\mathrm{O}(n\log n) = \mathrm{O}(n\log n)$ ce qui est meilleur que la complexité précédente.

• Recherche d'un élément dans une liste

```
def detect(L,elt):
    for x in L:
        if elt==x:
            return True
    return False
```

La complexité en O(1) dans le meilleur des cas (élément présent en première position) et en $\mathrm{O}(n)$ dans le pire des cas (où n est la taille de `L`).

Remarque : Dans les deux exemples précédents, il s'agit en fait d'une boucle `while` déguisée puisque le `return True` peut casser la boucle `for` avant la fin.

• **Recherche d'un élément dans une liste triée**

```
def rech_dicho(elt,L):
    deb,fin,trouve=0,len(L)-1,False
    while not trouve and deb<=fin:
        milieu=(deb+fin)//2
        if L[milieu]==elt:
            trouve=True
        elif L[milieu]>elt:
            fin=milieu-1
        else:
            deb=milieu+1
    return L[milieu]==elt,milieu
```

La complexité est en $O(1)$ dans le meilleur des cas (élément présent au milieu de la liste) et en $O(\log n)$ dans le pire des cas : la longueur de la zone de recherche est divisée par 2 à chaque étape donc pour $2^{p-1} \leqslant n < 2^p$, le programme s'arrête après p itérations.

• **Exponentiation rapide**

Soit x un réel et n un entier non nul d'écriture binaire $n = \langle d_{p-1}, \dots, d_0 \rangle_2$. On a

$$x^n = x^{\left(\sum_{i=0}^{p-1} d_i 2^i\right)} = \prod_{i=0}^{p-1} \left(x^{d_i 2^i}\right) = \prod_{i=0}^{p-1} \left(x^{2^i}\right)^{d_i}$$

Cette écriture permet d'envisager un algorithme performant pour le calcul de x^n, algorithme dit d'*exponentiation rapide* :

```
def expo(x,n):
    a,r,k=x,x**0,n
    while k>0:
        if k%2==1:
            r*=a
        a*=a
        k//=2
    return r
```

Il y a p produits à effectuer d'où une complexité en $O(p) = O(\log n)$. On fait l'hypothèse très simplificatrice que les multiplications qui interviennent ci-dessus sont à coût constant, ce qui est clairement faux pour de grands nombres.

VI Contrôle et preuve d'un programme

1 Jeux de tests

On prévoit un jeu de tests pour vérifier le bon comportement d'une fonctions. Ceci ne constitue pas une preuve de la correction du programme mais plutôt un indice de confiance.

```
def test_rech_dicho():
    L=list(range(1,20,2))
    assert rech_dicho(1,L)==True,0
    assert rech_dicho(19,L)==True,9
    assert rech_dicho(9,L)==True,4
    assert rech_dicho(10,L)[0]==False
    assert rech_dicho(18,L)[0]==False
    assert rech_dicho(20,L)[0]==False
    assert rech_dicho(2,L)[0]==False
    assert rech_dicho(0,L)[0]==False
```

La fonction est enregistrée dans le fichier `test.py` puis on exécute `pytest test.py` depuis l'invite de commande :

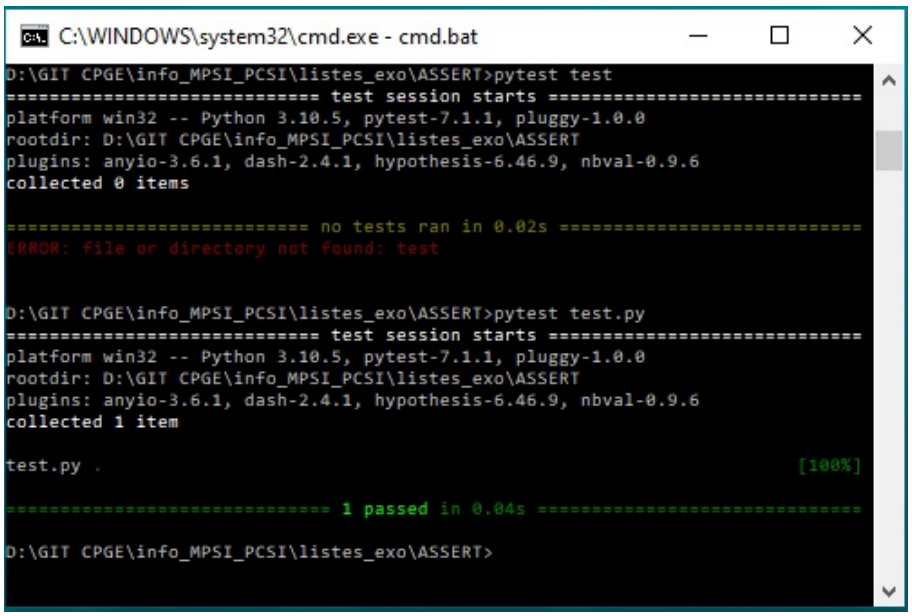


FIGURE 2 – Jeu de tests validé

2 Variant et invariant de boucle

Un *variant* de boucle est une variable à valeur dans $\mathbb{N}$ et qui décroît strictement à chaque passage dans la boucle.

Un *invariant* de boucle est un *prédicat* qui ne change pas au cours de la boucle. On démontre cette propriété par récurrence.

3 Exemples

• Coefficient binomial

```
def binom(n,k):
    res=1
    for i in range(k):
        res=(n-i)*res//(i+1)
    return res
```

La variable `k-i` est un variant de boucle et une invariant est donné par

$$\mathscr{P}(i) : \quad res = \binom{n}{i}$$

• Exponentiation rapide

```
def expo(x,n):
    a,r,k=x,x**0,n
    while k>0:
        if k%2==1:
            r*=a
        a*=a
        k//=2
    return r
```

La variable `k` est une variant de boucle et notant $n = \langle d_{p-1}, \dots, d_0 \rangle$ son écriture binaire, un invariant est donné par

$$\mathscr{P}(j) : \quad k = \langle d_{p-1}, \dots, d_j \rangle_2 \qquad r = x^{\langle d_{j-1}, \dots, d_0 \rangle} \qquad a = x^{2^j}$$

avec j le nombre de passage dans la boucle `while`.

VII Piles et Files

1 Piles

Une *pile* (*stack* en anglais) est une structure de données correspondant à un empilement d'éléments régi par la règle *« dernier arrivé, premier sorti »* (*LIFO* en anglais, *Last In First Out*).

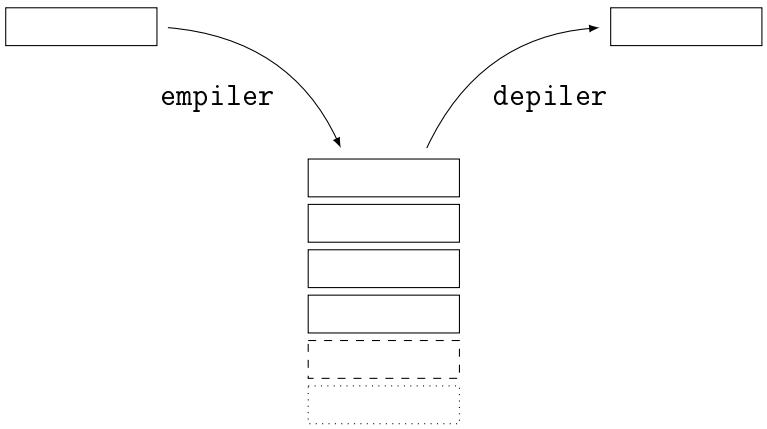


FIGURE 3 – Schéma d'une pile

Pour manipuler des piles, on utilise les *opération primitives* suivantes :
- la fonction `Pile` : crée une pile vide ;
- la méthode `empiler` : ajoute un élément au sommet de la pile ;
- la méthode `depiler` : dépile l'élément au sommet de la pile et le renvoie ;
- la méthode `vide` : indique si la pile est vide.

Exercice : Écrire, uniquement à l'aide des opérations primitives la fonction `duplique(P)` qui, pour une pile `P` non vide, duplique l'élément au sommet de la pile.
Corrigé : On saisit :

```
def duplique(P):
    S=P.depiler()
    P.empiler(S)
    P.empiler(S)
```

2 Files

Une *file* aussi appelée *file d'attente* (*queue* en anglais) est une structure de données basée sur le principe « premier entré, premier sorti » (*FIFO* en anglais, *First In First Out*).

FIGURE 4 – File

En python, on dispose du module `deque` de la bibliothèque `collections` qui propose une implémentation de cette structure

```
from collections import deque
```

Sur une variable de type `deque`, on utilise les méthodes :
- `append` : ajoute un élément à droite ;
- `appendleft` : ajoute un élément à gauche ;
- `pop` : renvoie et supprime l'élément de droite ;
- `popleft` : renvoie et supprime l'élément de gauche.

```
>>> D=deque()
>>> D.append(1)
>>> D.appendleft(2)
>>> D.appendleft(3)
>>> D
deque([3, 2, 1])
>>> D.pop()
1
>>> D
deque([3, 2])
>>> D.popleft()
3
>>> D
deque([2])
```

L'implémentation proposée est optimisée : les opérations `append`, `appendleft`, `pop`, `popleft` sont de complexité temporelle en O(1) (complexité amortie).

VIII Récursivité

1 Propagation et cas de base

Une fonction *récursive* est une fonction qui fait appel à elle-même. Cet appel est dénommé *appel récursif* ou *récursion*.

Une fonction récursive est constituée de deux alternatives fondamentales :
- un (ou plusieurs) *cas de base* sans appel récursif ;
- le *cas de propagation* avec un (ou plusieurs) appel récursif.

• **Factorielle**

```
def fact(n):
    if n==0:
        return 1                # cas de base
    else:
        return n*fact(n-1)      # cas de propagation avec appel récursif
```

Exercice : Écrire une version récursive du calcul de coefficient binomial `binom(n,k)`.

Corrigé : On saisit

```
def binom(n,k):
    if  k>n:
        return 0
    elif k==0:
        return 1
    else:
        return n*binom(n-1,k-1)//k
```

2 Empilement des récursions

Lors de l'appel d'une fonction récursive, une pile est utilisée pour l'empilement des récursions jusqu'au cas de base puis celles-ci sont dépilées.

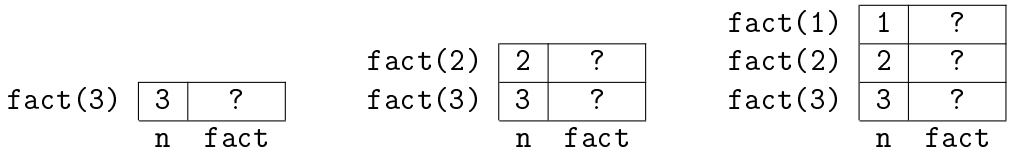


Schéma 1 - Empilement des appels récursifs

Une fois le cas de base atteint, il ne reste plus qu'à dépiler les appels et à effectuer les calculs successifs.

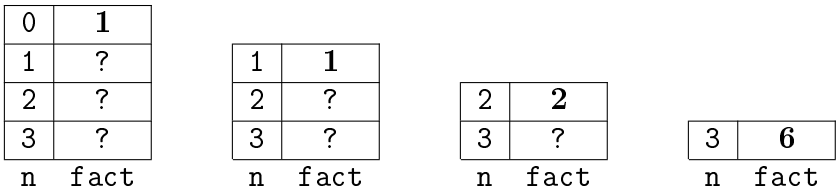


SCHÉMA 2 - Dépilement des appels récursifs

⚠ **Avertissement :** La pile utilisée pour les empilements d'appels est une pile finie.

3 Complexité et récursivité

• Récursions multiples

Les récursions multiples par propagation sont très coûteuses : avec p récursions à chaque propagation pour un argument décroissant linéairement et dont les autres opérations sont de complexité temporelle et spatiale en $O(1)$, la complexité temporelle en $O(\rho^n)$ avec $\rho > 1$.

La suite de Fibonacci $(u_n)_n$ est définie par $u_0 = u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$. On peut la coder naïvement récursivement avec :

```
def fibo(n):
    if n<=1:                        # cas de base pour n=0 et n=1
        return 1
    else:
        return fibo(n-1)+fibo(n-2)  # propagation avec récursion double
```

La complexité temporelle de `fibo(n)` est en $O(\varphi^n)$ avec $\varphi = \dfrac{1+\sqrt{5}}{2}$.

• Exponentiation rapide

Pour $(x, n) \in \mathbb{R} \times \mathbb{N}$, on a

$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2 & \text{si } n \text{ pair} \\ x \times \left(x^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ impair} \end{cases}$$

On en déduit l'implémentation récursive simple :

```
def expo_rec(x,n):
    if n==0:
        return x**0             # cas de base avec exposant nul
    else:
        r=expo_rec(x,n//2)**2   # propagation avec le quotient de n par 2
        if n%2==0:              # distinction selon la parité
            return r
        else:
            return x*r
```

La complexité temporelle vérifie

$$\mathrm{T}(n) = \mathrm{T}\left(n//2\right) + \mathrm{O}(1)$$

Ainsi, notant $n = \sum_{i=0}^{p-1} d_i 2^i$ son écriture binaire avec $p = \lfloor \log_2(n) \rfloor + 1$, il vient

$$\mathrm{T}(n) = \mathrm{T}(0) + \sum_{k=0}^{p-1} \left[\mathrm{T}\left(n//2^k\right) - \mathrm{T}\left(n//2^{k+1}\right)\right] = \mathrm{R}(0) + \mathrm{O}(p) = \mathrm{O}(\log n)$$

• **Diviser pour régner**

Il s'agit d'un paradigme fondamental de l'algorithmique qui confère à la programmation récursive ses lettres de noblesse : pour résoudre un problème, on le divise en sous-problème indépendants que l'on résout récursivement.

L'exemple le plus célèbre est sans doute celui de la FFT (Fast Fourier Transform) : l'implémentation naïve de la transformée de Fourier est de complexité temporelle en $\mathrm{O}(n^2)$ tandis que l'implémentation récursive « diviser pour régner » est en $\mathrm{O}(n \log n)$.

IX Tris

Un tri est dit *en place* s'il modifie directement la structure qu'il est en train de trier sans créer de variable de taille de même ordre que la liste à trier.

1 Tri par insertion

Le *tri par insertion* consiste à itérer le procédé suivant : si les k premiers éléments sont triés, on vient insérer le $k+1$-ème à sa place avec les k premiers.

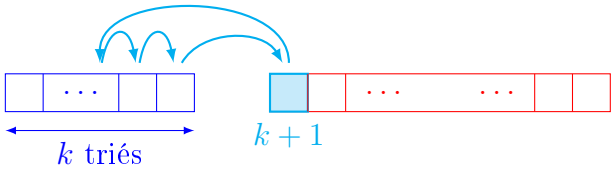


FIGURE 5 – Tri par insertion

Ainsi, pour une liste `T`, on examine les deux premiers éléments et on les échange éventuellement pour qu'ils soient ordonnés, puis on considère le troisième élément et on vient, par remontées successives, l'insérer vis-à-vis des deux premiers déjà triés, etc. ...

```
def tri_ins(T):
    n=len(T)
    for i in range(1,n):          # on parcourt la liste
        c=T[i]                    # c est l'élément courant
        j=i-1                     # j balaie les indices des éléments précédents
        while j>=0 and T[j]>c:    # tant que c n'est pas bien placé
            T[j+1]=T[j]           # on décale les éléments précédents
            j-=1
        T[j+1]=c
```

C'est un tri en place donc la complexité temporelle est :
- dans le meilleur des cas en $O(n)$;
- dans le pire des cas en $O(n^2)$.

2 Tri rapide

Le *tri rapide* repose sur le paradigme « diviser pour régner ». Il consiste en le procédé suivant : choix d'un pivot, placement des éléments de la liste vis-à-vis du pivot puis tri récursif à droite et à gauche du pivot.

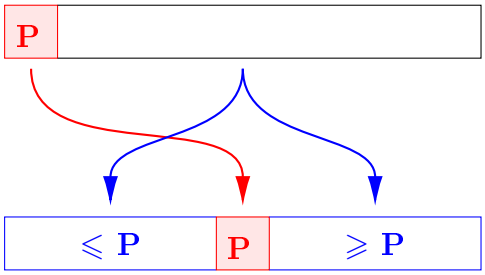


FIGURE 6 – Placement vis-à-vis du pivot

```
def tri_rapide(T):
    if T==[]:
        return []
    else:
        pivot=T[0]              # choix du pivot à gauche
        T1,T2=[],[]
        for x in T[1:]:         # placement des éléments vis-à-vis du pivot
            if x<pivot:
                T1.append(x)
            else:
                T2.append(x)
                                # après placement, tri récursif
        return tri_rapide(T1)+[pivot]+tri_rapide(T2)
```

C'est une version simple à implémenter mais pas en place (défaut mineur en python...).

La complexité temporelle du tri rapide est :
- dans le meilleur des cas en $O(n \log n)$;
- dans le pire des cas en $O(n^2)$.

Il existe une version *en place* du tri rapide (modification directe de l'argument, sans création d'une variable de taille du même ordre que l'argument).

Exercice : Expliquer pourquoi la version proposée du tri rapide n'est pas en place.

Corrigé : On construit au cours des récursions des listes `T1` et `T2` dont les tailles sont de l'ordre de n la taille de l'argument `T`, de l'ordre de $n/2$ si le pivot est à peu près au milieu après placement et au pire de l'ordre de n si le pivot est complètement à droite ou à gauche après placement.

3 Tri fusion

Le *tri fusion* (*merge sort* en anglais) est également basé sur le paradigme « diviser pour régner ». Il consiste en le procédé suivant : coupure de la liste de taille n en deux listes de tailles $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$, tri récursif de ces listes puis fusion de celles-ci.

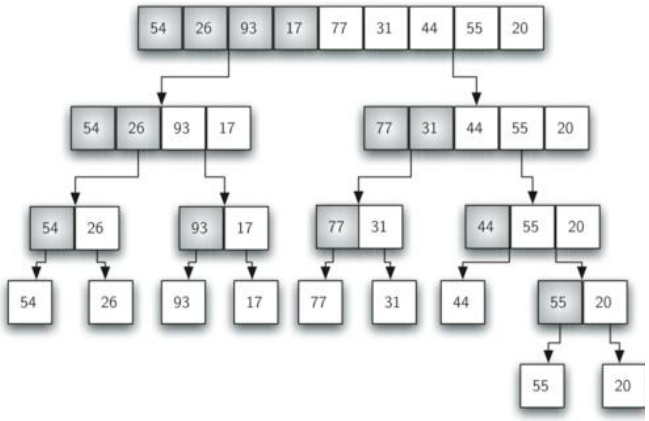


FIGURE 7 – Diviser pour régner

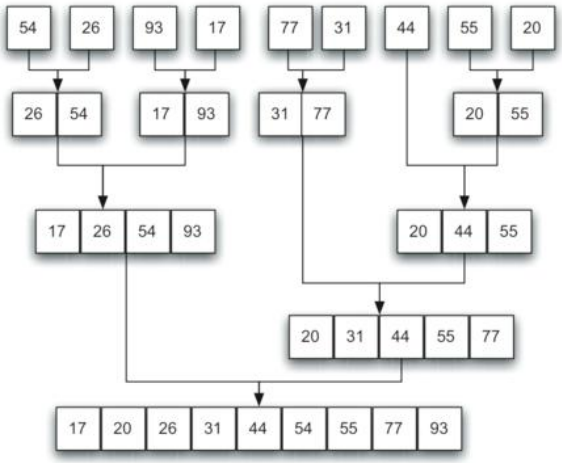


FIGURE 8 – Fusion des listes ordonnées

```
def fusion(T,g,m,d):        # fusionne les listes triées T[g:m] et T[m:d]
    tab=T[g:m]+T[d-1:m-1:-1]
                            # construit une liste "dos-à-dos"
                            # [T[g],...,T[m-1],T[d-1],...T[m]]
    i=0                     # l'indice i parcourt tab en montant (donc T[g:m])
    j=-1                    # l'indice j parcourt tab en descendant (donc T[m:d])
    for k in range(g,d):    # on remplit chaque case
        if tab[i]<tab[j]:   # choix de la liste pour l'élémet courant
            T[k]=tab[i]     # si dans première moitié
            i+=1            # on incrémente i
        else:
            T[k]=tab[j]     #sinon on décrémente j
            j-=1

def tri_fusion_rec(T,g,d):
    if d-g>1:
        m=(g+d)//2
        tri_fusion_rec(T,g,m)   # on trie la moitié gauche
        tri_fusion_rec(T,m,d)   # on trie la moitié droite
        fusion(T,g,m,d)         # on fusionne chaque moitié triée


def tri_fusion(T):
    tri_fusion_rec(T,0,len(T))  # amorce le tri fusion
```

Le tri fusion n'est pas en place et sa complexité temporelle est en $\mathrm{O}(n \log n)$.

L'instruction `sorted` et la méthode `sort` sont une implémentation d'un tri hybride construit à partir du tri par insertion et du tri fusion.

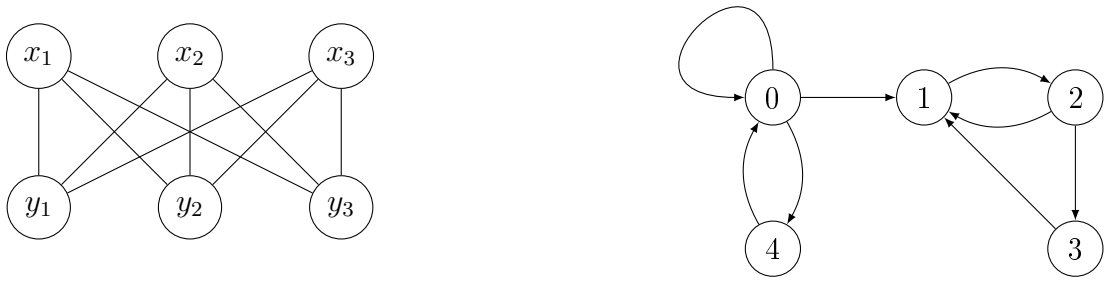
X Graphes

1 Représentation

Pour X un ensemble, on note $\mathcal{P}_2(X)$ l'ensemble des parties de X à un ou deux éléments.

Un *graphe non orienté* G est un couple (S, A) avec S un ensemble fini de *sommets* ou *nœuds* et A une partie de $\mathcal{P}_2(S)$ dont les éléments sont appelés *arêtes*.

Un *graphe orienté* G est un couple (S, A) avec S un ensemble fini de *sommets* ou *nœuds* et A une partie de S^2 dont les éléments sont appelés *arcs*.



Un *graphe valué* ou *pondéré* G est un triplet (S, A, ν) avec (S, A) un graphe orienté ou non muni d'une application $\nu : A \to \mathbb{R}$ appelée *valuation* ou *pondération* du graphe.

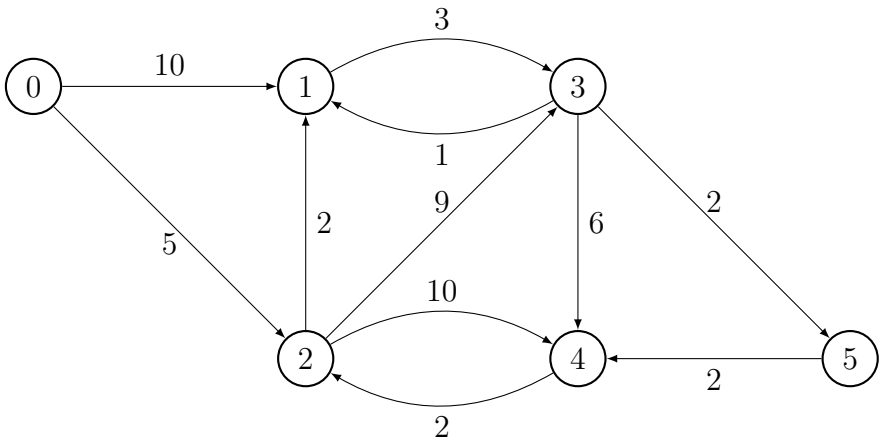
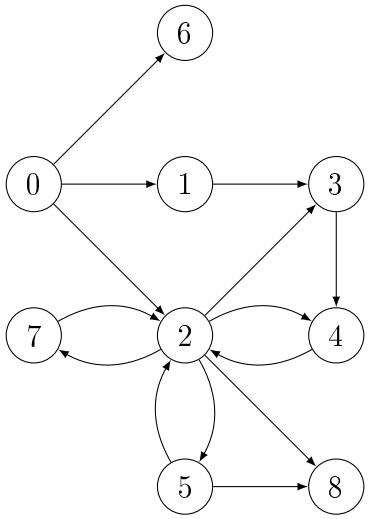


FIGURE 9 – Graphe valué

Deux sommets sont dits *adjacents* s'il existe un arête ou un arc reliant l'un à l'autre.

La représentation d'un graphe $\mathrm{G} = (\mathrm{S}, \mathrm{A})$ avec $\mathrm{S} = [s_0, \ldots, s_{n-1}]$, orienté ou non orienté par *listes d'adjacences* consiste en une liste de listes $\mathrm{LA} = [\mathrm{LA}[i], i \in [\![0\,;\, n-1]\!]]$ telle que pour $i \in [\![0\,;\, n-1]\!]$, la liste $\mathrm{LA}[i]$ contient les sommets adjacents à s_i.



L'ensemble des sommets et la liste d'adjacence sont donnés par

$$\mathrm{S} = [0, 1, \ldots, 8] \qquad \mathrm{LA} = [[1, 2, 6], [3], [3, 4, 5, 7, 8], [4], [2], [2, 8], [], [2], []]$$

qu'on peut implémenter par

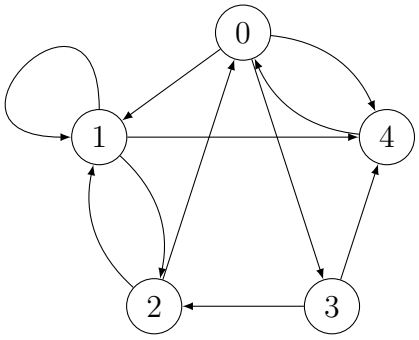
```
S=list(range(9))
A=[[1,2,6],[3],[3,4,5,7,8],[4],[2],[2,8],[],[2],[]]
```

ou avec un dictionnaire :

```
S=list(range(9))
A={0:[1,2,6],1:[3],2:[3,4,5,7,8],3:[4],4:[2],5:[2,8],6:[],7:[2],8:[]}
```

On peut aussi utiliser représenter le graphe par une *matrice d'adjacence* $\mathrm{M} = \left(m_{i,j}\right)_{0 \leqslant i,j \leqslant n-1} \in \mathscr{M}_n(\mathbb{R})$ définie par

$$\forall (i,j) \in [\![0\,;\, n-1]\!] \qquad m_{i,j} = \begin{cases} 1 & \text{si } (s_i, s_j) \in \mathrm{A} \text{ ou } \{s_i, s_j\} \in \mathrm{A} \\ 0 & \text{sinon} \end{cases}$$



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

FIGURE 10 – Graphe orienté et sa matrice d'adjacence

2 Parcours d'un graphe

• Parcours en profondeur

On propose une version récursive du *parcours en profondeur* (DFS en anglais, Depth First Search). Celui-ci s'applique indifféremment aux graphes orientés ou non orientés. Le principe est le suivant :

— on part d'un sommet donné que l'on considère comme « en cours de visite » ;
— s'il admet des sommets adjacents « non visités » (des successeurs), on appelle récursivement le parcours en profondeur depuis chacun d'eux ;
— s'il n'y a pas ou plus de successeur, le sommet est considéré comme « visité ».

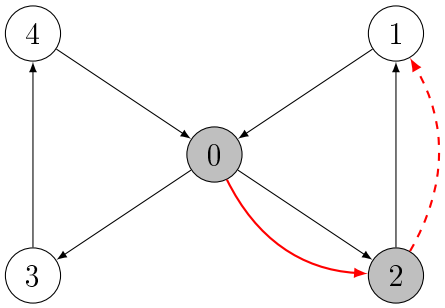


FIGURE 11 – Début du parcours en profondeur depuis le sommet 0

• Parcours en largeur

Le *parcours en largeur* (BFS en anglais, Breadth First Search) s'applique indifféremment aux graphes orientés ou non orientés. Son principe est le suivant :

— on part d'un sommet qu'on place dans une file et que l'on considère comme « en cours de visite » ;
— tant que la file n'est pas vide, on sort le sommet premier entré, on fait entrer dans la file tous ses sommets adjacents « non visités » puis on considère le sommet d'origine comme « visité ».

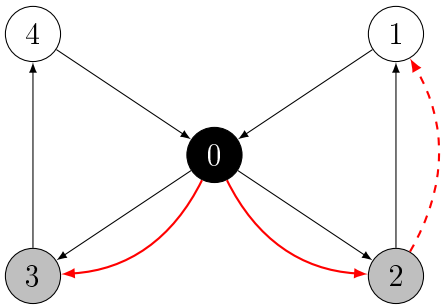


FIGURE 12 – Début du parcours en largeur depuis le sommet 0

3 Plus court chemin

Soit $G = (S, A, \nu)$ un graphe orienté valué. Un *chemin* reliant un sommet x à un sommet y est une suite finie d'arcs consécutifs reliant x à y. Pour un chemin $c = (x_0, \ldots, x_k)$, on définit son *poids* ou *coût* par

$$\mathrm{L}(c) = \sum_{i=1}^{k} \nu(x_{i-1}, x_i)$$

On définit le *coût d'un plus court chemin* entre les sommets x et y noté $\delta(x, y)$ par

$$\delta(x,y) = \begin{cases} \inf\{\mathrm{L}(c), c : \text{chemin de } x \text{ à } y\} & \text{s'il existe un chemin de } x \text{ à } y \\ +\infty & \text{sinon} \end{cases}$$

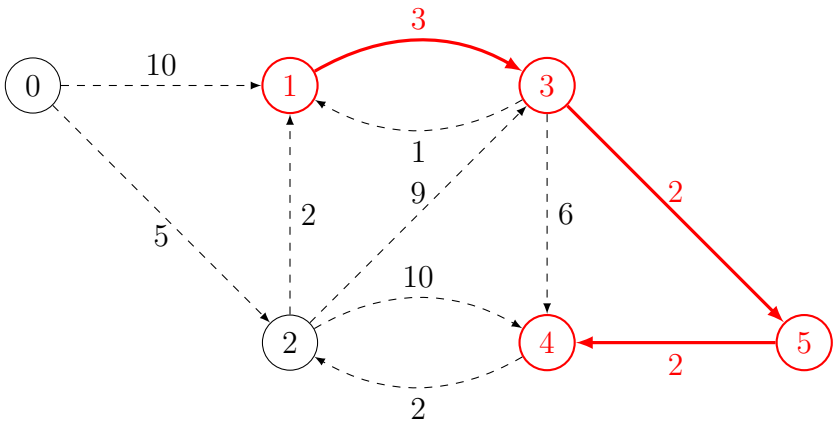


FIGURE 13 – Plus court chemin du sommet 1 au sommet 4

Un chemin est *simple* s'il ne contient pas deux fois le même arc. Un chemin simple dont les extrémités coïncident est appelé *circuit*. Un circuit de coût strictement négatif est dit *absorbant*.

Étant donnés deux sommets x et y reliés par un chemin, il existe un plus court chemin entre x et y si et seulement aucun chemin de x à y ne contient de circuit absorbant.

• Algorithme de Dijkstra

Soit $\mathrm{G} = (\mathrm{S}, \mathrm{A}, \nu)$ un graphe orienté valué positivement. On note s_0 le sommet source et $\mathrm{d}[u]$ la longueur d'un plus court chemin de s_0 à un sommet u découvert à un certain stade de l'algorithme.

Le principe de l'algorithme est le suivant :

- on initialise $\mathrm{d}[s_0] \leftarrow 0$ et $\mathrm{d}[u] \leftarrow \infty$ pour tout $u \in \mathrm{S} \setminus \{s_0\}$;
- tant qu'il y a des sommets « non visités » :
  - on détermine un sommet u non visité tel que $\mathrm{d}[u]$ est minimal ;
  - le sommet u est considéré comme « visité » ;
  - pour chaque sommet s non visité et successeur de u, on effectue l'opération de *relâchement* :

$$\text{si } \mathrm{d}[s] > \mathrm{d}[u] + \nu(u,s), \text{ alors } \mathrm{d}[s] \leftarrow \mathrm{d}[u] + \nu(u,s)$$

    autrement dit : si pour aller à `s` depuis `s0`, il est plus court de passer par `u`, alors on le fait.

On utilise les dictionnaires suivants :

- `cout` : pour `s` un sommet, `cout[s]` est la valeur de $\mathrm{d}[s]$;
- `predec` : pour `s` un sommet, `predec[s]` est son prédécesseur dans le plus court chemin depuis s_0 ;

— `djvu` : pour `s` un sommet, `djvu[s]` est un booléen qui dit si le sommet a été déjà vu ou pas.

```
def dijkstra(S,A,s0):              # recherche de plus courts chemins depuis s0
    cout,predec,djvu={},{},{}
    for s in S:                    # initialisation :
        cout[s]=np.inf             # les sommets à cout infini de s0
        djvu[s]=False              # et non visités
    cout[s0]=0
    for _ in S:
        smin,cmin=s0,np.inf        # recherche d'un sommet pas déjà vu
                                   # et à cout minimal
        for s in S:
            if not djvu[s] and cout[s]<cmin:
                smin,cmin=s,cout[s]
        djvu[smin]=True            # le sommet est considéré visité
        for s,nu in A[smin]:       # pour les successeurs non visités
                                   # on relache
            if not djvu[s] and cout[s]>cout[smin]+nu:
                cout[s]=cout[smin]+nu
                predec[s]=smin
    return cout,predec
```

Remarque : L'implémentation ci-avant détermine la totalité des plus courts chemins existants depuis `s0`. On peut aussi effectuer une recherche plus ciblée d'un plus court chemin entre un sommet de départ et un sommet d'arrivée (à fournir en arguments).
Exercice : Déterminer la complexité temporelle de `dijkstra`. On note $n = \text{Card S}$ et $m = \text{Card A}$. Les accès en lecture en écriture dans une liste ou un dictionnaire sont supposés être en $\text{O}(1)$ (complexité amortie pour une liste et complexité moyenne pour un dictionnaire).

Corrigé : On effectue une première boucle de parcours de `S` puis, à l'intérieur de celle-ci, on parcourt tous les sommets de `S` avec un test et éventuellement une affectation puis on parcourt tous les sommets adjacents au sommet `smin` avec un test et éventuellement deux affectations. Le nombre de sommets adjacents à `smin` est majoré par n et on a donc un coût temporel en

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\text{O}(1) = \text{O}(n^2)$$

Les dictionnaires `cout` et `djvu` sont initialisés avec autant de clés que de sommets. Le dictionnaire `predec` contient les sommets pour lesquels un relâchement a été effectué donc au plus n. Les autres variables sont en nombre fixe et de taille fixée. Par conséquent, la complexité spatiale est en $\text{O}(n)$.

• **Algorithme** A$\star$

L'idée mise en œuvre dans l'algorithme A$\star$ est de privilégier la recherche de chemin d'un sommet source s à un sommet cible c en s'efforçant d'aller dans la direction de c et donc de réduire, par exemple, la distance euclidienne à c.

Soit $G = (S, A, \nu)$ un graphe orienté valué positivement et $c \in S$. Une *heuristique* h est une fonction $h : S \to \mathbb{R}_+$. L'heuristique h est dite *admissible* pour la recherche de plus court chemin au sommet c (la cible) si

$$\forall s \in S \qquad h(s) \leqslant \delta(s, c)$$

L'heuristique h est dite *consistante* si

$$\forall (s, u) \in A^2 \qquad h(s) \leqslant \nu(s, u) + h(u)$$

Avec une heuristique admissible, l'algorithme A$\star$ résout le problème de plus court chemin et qu'avec une heuristique consistante, l'algorithme A$\star$ résout le problème de plus court chemin en complexité linéaire.

```
def d2(A,B):
    xA,yA=A
    xB,yB=B
    return np.sqrt((xA-xB)**2+(yA-yB)**2)

def Astar_deb_fin(S,A,deb,fin):
    nb=0
    def h(s):
        return d2(s,fin)
    cout,dist,predec,djvu={},{},{},{}
    for s in S:
        cout[s]=np.inf
        dist[s]=np.inf
        djvu[s]=False
    cout[deb]=h(deb)
    dist[deb]=0
    while not djvu[fin]:
        nb+=1
        # recherche elt à cout minimal
        smin,cmin=deb,np.inf
        for s in S:
            if not djvu[s] and cout[s]<cmin:
                smin,cmin=s,cout[s]
        djvu[smin]=True
        # relachement chez les successeurs
        for s,nu in A[smin]:
            if not djvu[s] and dist[s]>dist[smin]+nu:
                dist[s]=dist[smin]+nu
                cout[s]=dist[s]+h(s)
                predec[s]=smin
    return cout,predec,nb
```

Remarque : Le programme est correct si les sommets `deb` et `fin` sont reliés. Dans le cas contraire, il faut prévoir une sortie de boucle.

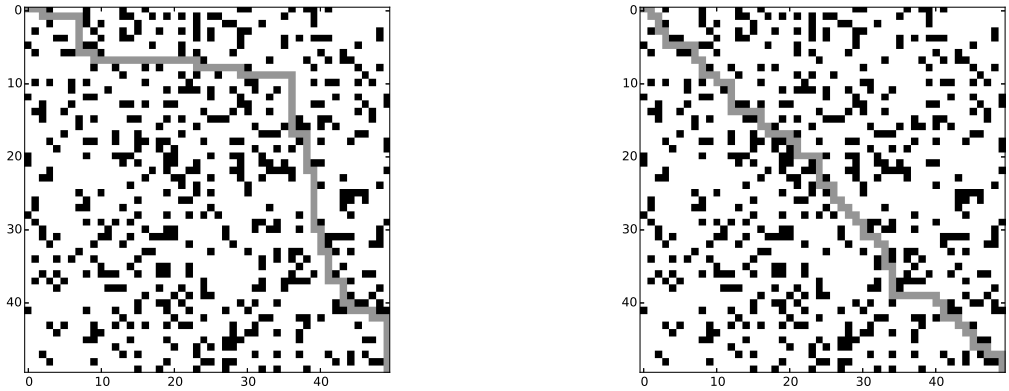


FIGURE 14 – Plus court chemin avec Dijkstra et A$\star$