

Informatique MP Arago

Document de compléments de cours et d'exercices avec des corrections¹

Ph. Langlois

version partielle du 23 janvier 2015

1. Excepté le devoir maison, très peu de corrections de questions qui ne demandent pas de programmes python sont proposées dans cette version encore très partielle.

Table des matières

I	Rappels de Python	7
1	Sur les listes python	9
1.1	Définir des listes en compréhension	9
1.2	Principales méthodes sur les listes	9
2	Tracer et mesurer des performances	11
2.1	Tracer avec Turtle	11
2.2	Tracer avec matplotlib	16
2.3	Mesurer le temps d'exécution d'un algorithme	20
II	Exercices	27
3	Remise en jambe et récursivité	29
3.1	Premières récursions	29
3.2	C'est vraiment parti!	32
4	Les piles	35
4.1	Exercices de base	35
4.2	Premières applications	41
4.3	Applications plus conséquentes	46
5	Trier	61
5.1	Les tris vus en cours	61
5.2	Autres algorithmes de tri et compléments	85
6	Couleurs, images et traitements	87
6.1	Principes	87
6.2	On commence en couleurs!	88
6.3	Des images et du traitement	89
III	Devoir maison	91
7	Novembre 2014	93

Bibliographie utilisée et conseillée

- [1] T. Audibert and A. Oussalah. *Informatique. Programmation et calcul scientifique en Python et Scilab*. Ellipses, 2014.
- [2] B. Cordeau and L. Pointal. Une introduction à Python. http://perso.limsi.fr/pointal/_media/python:cours:courspython3.pdf, 2014. version 1.618.
- [3] T. Cormen. *Algorithmes. Notions de base*. Dunod, 2013.
- [4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique*. Dunod, seconde edition, 2002.
- [5] P. Damphousse. *Petite introduction à l'algorithmique. À la découverte des mathématiques du pas à pas*. Ellipses, 2005.
- [6] G. Dowek, J.-P. Archambault, E. Baccelli, C. Cimelli, A. Cohen, C. Eisenbeis, T. Viéville, and B. Wack. *Informatique et Sciences du Numérique - Spécialité ISN en Terminale S*. Eyrolles, 2012.
- [7] E. Le Nagard. *Informatique. Initiation à l'algorithmique en Scilab et Python*. Pearson, 2013.
- [8] B. Wack, S. Conchon, J. Courant, M. de Falco, G. Dowek, J.-C. Filiâtre, and S. Gonnord. *Informatique pour tous en classes préparatoires aux grandes écoles*. Eyrolles, 2014.

Première partie

Rappels de Python

Chapitre 1

Sur les listes python

1.1 Définir des listes en compréhension

Principe : `[fonction(x) for x in uneListe if condition-de-filtrage]`

```
# les carrés des éléments
liste = [1, 2, 3, 4, 5, 6, 7]
les_carres = [x ** 2 for x in liste]

# Afficher les nombres pairs
print [x for x in liste if x % 2 == 0]
# Plus simple que filter, également :)

# Affiche les carrés pairs (combinaison des deux)
print [x ** 2 for x in liste if x ** 2 % 2 == 0]
# ou print [x for x in [a ** 2 for a in liste] if x % 2 == 0]
```

1.2 Principales méthodes sur les listes

- `L.append(x)` : Ajoute l'élément `x` à la fin de la liste `L`
- `L1.extend(L2)` : Étend la liste `L1` en y ajoutant tous les éléments de la liste `L2`
- `L.insert(i, x)` : Insère l'élément `x` dans la liste `L` à la position `i`
- `L.remove(x)` : Supprime de la liste `L` le premier élément dont la valeur est `x`.
- `L.pop([i])` : Enlève de la liste `L` l'élément situé à la position `i`
Si aucune position n'est indiquée, `L.pop()` enlève et retourne le dernier élément de la liste
- `L.index(x)` : Retourne la position du premier élément de la liste `L` ayant la valeur `x`.
- `L.sort()` : Trie les éléments de la liste, en place.
- `L.reverse()` : Inverse l'ordre des éléments de la liste, en place.

Chapitre 2

Tracer et mesurer des performances

2.1 Tracer avec Turtle

`turtle` est un module qui permet de tracer des figures géométriques à l'aide de mouvements élémentaires effectués par une ...tortue. La tortue est représentée par un icône Δ qui fixe sa direction de déplacement. Le tracé est créé par ses déplacements,

- en levant (`up()`) ou non (`down()`) un crayon,
- déplacements contrôlés par des changements de direction : `right(angle)` et `left(angle)`,
- et des distances vers l'avant : `forward(distance)` ou l'arrière : `backward(distance)`.
- `goto(x, y)` permet de se déplacer au point (x, y) sans modifier la direction de la tête.
- `reset()` initialise la fenêtre graphique et `mainloop()` permet de la conserver à l'écran une fois le tracé terminé.

D'autres options (couleur et taille du trait, vitesse de déplacement, ...) se comprennent à l'aide de l'exemple suivant.

```
import turtle as t
"""Turtle : mouvements elementaires"""

def carre(c, colorstring):
    """dessine un carre de cote c et le remplit de color"""
    t.fillcolor(colorstring)
    t.begin_fill()
    for i in range(4):
        t.forward(c)
        t.right(90)
    t.end_fill()

# parametres deplacement
un = 30
deux = 2*un

# init : taille ecran en pixels , clear ecran , reset des variables
t.screensize(100,100)
t.clear()
t.reset()
# localiser (0,0)
t.goto(0,0)
t.dot()
t.up()

# un carre
t.goto(10,10)
t.down()
```

```

carre(deux, "black")

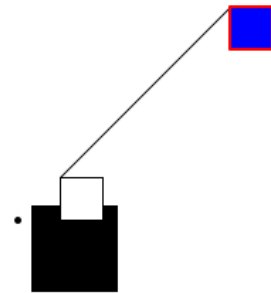
# on se deplace sans laisser de trace
t.up()
t.goto(un, un)
t.down()

# un carre plus petit
carre(un, "white")

# et un autre carre en couleur lie au precedent
t.goto(5*un,5*un)
t.pencolor("red")
t.pensize(2)
t.hideturtle() # pour aller plus vite
carre(un, "blue")

# pour que le trace reste a l ecran
t.mainloop()

```



Plein d'autres détails sont bien sûr accessibles en ligne à : <https://docs.python.org/3/library/>

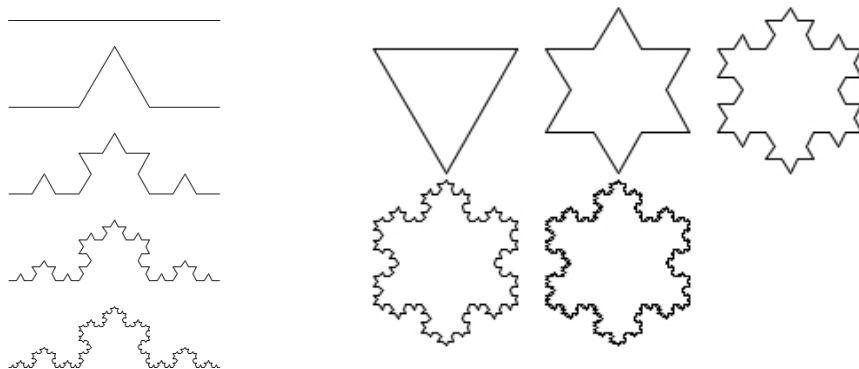
Utiliser `turtle` pour réaliser les tracés suivants.

Exercice 1. (P) Le flocon de Von Koch.

Le flocon de Von Koch (1906) est une courbe fractale (continue partout et dérivable nulle part) obtenue comme la limite de la transformation suivante appliquée récursivement à un segment de départ.

1. le segment est découpée en 3 parties égales,
2. le segment intermédiaire est remplacé par deux segments formant un triangle équilatéral avec le segment supprimé,
3. et ainsi de suite sur les 4 segments ainsi obtenus.

Voici les cinq premières courbes obtenues à partir d'un segment droit (à gauche) ou d'un triangle équilatéral : le flocon de Von Koch (à droite).



1. Un segment.
 - (a) `segment-vk` : écrire la suite de commandes `turtle` qui trace la n -ième transformation d'un segment donné.
 - (b) `segment-vk-n` : compléter `segment-vk` pour obtenir le tracé ci-dessus.
2. Le flocon ! On effectue la même transformation sur chacun des cotés d'un triangle équilatéral.
 - (a) `flocon` : écrire la suite de commandes `turtle` qui trace la n -ième transformation d'un triangle équilatéral donné.

(b) flocon-n : compléter flocon pour obtenir le tracé ci-dessus.

```
import turtle as t
"""Flocon de Von Koch"""

def segment(d, n, color="black"):
    """recursion sur n pour decouper le segment de longueur d et
    tracer d (longueur du segment) pixels si n==0"""
    t.pencolor(color)

    if n==0:
        t.forward(d)
    else:
        dsur3 = d//3
        segment(dsur3, n-1)
        t.left(60)
        segment(dsur3, n-1)
        t.right(120)
        segment(dsur3, n-1)
        t.left(60)
        segment(dsur3, n-1)

def flocon(d, n, color="black"):
    segment(d, n, color)
    t.right(60)
    segment(d, n, color)
    t.right(60)
    segment(d, n, color)
    t.right(60)

# init : taille ecran en pixels , clear ecran , reset des variables
t.screenize(100,100)
t.clear()
t.reset()
t.hideturtle() # pour aller plus vite

# angles en deg
t.degrees()

# taille
d = 400
#nbiter = input("nombre d'iterations = ")
nbiter = 3

#extremite de depart
a = (-200, 0)
t.up()
t.goto(a)
t.down()
# trace
flocon(d, nbiter)

# pour que le trace reste a l ecran
t.mainloop()
```

```
import turtle as t
import cmath as cm
```

```

"""n flocons de Von Koch"""

def segment(d, n, color="black"):
    """recursion sur n pour decouper le segment de longueur d et
    tracer d (longueur du segment) pixels si n==0"""
    t.pencolor(color)

    if n==0:
        t.forward(d)
    else:
        dsur3 = d//3
        segment(dsur3, n-1)
        t.left(60)
        segment(dsur3, n-1)
        t.right(120)
        segment(dsur3, n-1)
        t.left(60)
        segment(dsur3, n-1)

# init : taille ecran en pixels , clear ecran , reset des variables
#t.screenize(1000,500)
t.clear()
t.reset()
t.hideturtle() # pour aller plus vite

# angles en deg
t.degrees()

# taille
d = 243
#nbiter = input("nombre d'iterations = ")
nbiter = 4

for n in range(nbiter):
    #extremities de depart successives
    a = (-d, d - n*d//3)
    t.up()
    t.goto(a)
    t.down()
    # trace
    segment(d, n)

# pour que le trace reste a l ecran
t.mainloop()

```

```

import turtle as t
"""Flocon de Von Koch"""

def segment(d, n, color="black"):
    """recursion sur n pour decouper le segment de longueur d et
    tracer d (longueur du segment) pixels si n==0"""
    t.pencolor(color)

    if n==0:
        t.forward(d)
    else:
        dsur3 = d//3
        segment(dsur3, n-1)

```

```

        t.left(60)
        segment(dsur3, n-1)
        t.right(120)
        segment(dsur3, n-1)
        t.left(60)
        segment(dsur3, n-1)

def flocon(d, n, color="black"):
    segment(d, n, color)
    t.right(120)
    segment(d, n, color)
    t.right(120)
    segment(d, n, color)
    t.right(120)

# init : taille ecran en pixels , clear ecran , reset des variables
t.screenize(100,100)
t.clear()
t.reset()
t.hideturtle() # pour aller plus vite

# angles en deg
t.degrees()

# taille
d = 84
#nbiter = input("nombre d'iterations = ")
nbiter = 2

#extremite de depart
a = (-200, 200)
t.up()
t.goto(a)
t.down()
# trace
flocon(d, nbiter)

# pour que le trace reste a l ecran
t.mainloop()

```

```

import turtle as t
import cmath as cm
"""n flocons de Von Koch"""

def segment(d, n, color="black"):
    """recursion sur n pour decouper le segment de longueur d et
    tracer d (longueur du segment) pixels si n==0"""
    t.pencolor(color)

    if n==0:
        t.forward(d)
    else:
        dsur3 = d//3
        segment(dsur3, n-1)
        t.left(60)
        segment(dsur3, n-1)
        t.right(120)
        segment(dsur3, n-1)

```

```

        t.left(60)
        segment(dsur3, n-1)

def flocon(d, n, color="black"):
    segment(d, n, color)
    t.right(120)
    segment(d, n, color)
    t.right(120)
    segment(d, n, color)
    t.right(120)

# init : taille ecran en pixels , clear ecran , reset des variables
#t.screenize(1000,500)
t.clear()
t.reset()
t.hideturtle() # pour aller plus vite

# angles en deg
t.degrees()

# taille
d = 81
#nbiter = input("nombre d'iterations = ")
nbiter = 6

for n in range(nbiter):
    # n traces cote a cote repartis sur 3 lignes
    ligne, col = divmod(n,3)
    #extremities de depart successives
    decal_x = -250 # pour profiter de la largeur de la fenetre
    a = (decal_x + 1.2*d*col, 2*d - 1.2*d*ligne)
    t.up()
    t.goto(a)
    t.down()
    # trace
    flocon(d, n)

# pour que le trace reste a l ecran
t.mainloop()

```

Le module `turtle` est aussi utilisé pour tracer les labyrinthes parfaits et les chemins de sortie de l'exercice 19.

2.2 Tracer avec `matplotlib`

`matplotlib` est le module adapté à tous les types de tracés scientifiques : courbes, histogrammes, nuages de points, ... Ses options sont très nombreuses, de même que la documentation en ligne ou "livrée avec". L'utilisation basique consiste à tracer des courbes $x - y$ à partir de listes de points, d'un fichier de valeurs ou d'une expression, de choisir les bons axes, les bonnes échelles, d'indiquer les noms des courbes et des axes, de visualiser les tracés à l'écran et de les sauvegarder dans des fichiers (pdf ou png). Il peut être utile de tracer plusieurs "sous-figures" au sein d'une même figure. `numpy` est le compagnon naturel de `matplotlib` — `pylab` est la réunion de deux.

Une première prise en main est très bien décrite par :

<http://www.labri.fr/perso/nrougier/teaching/matplotlib/>

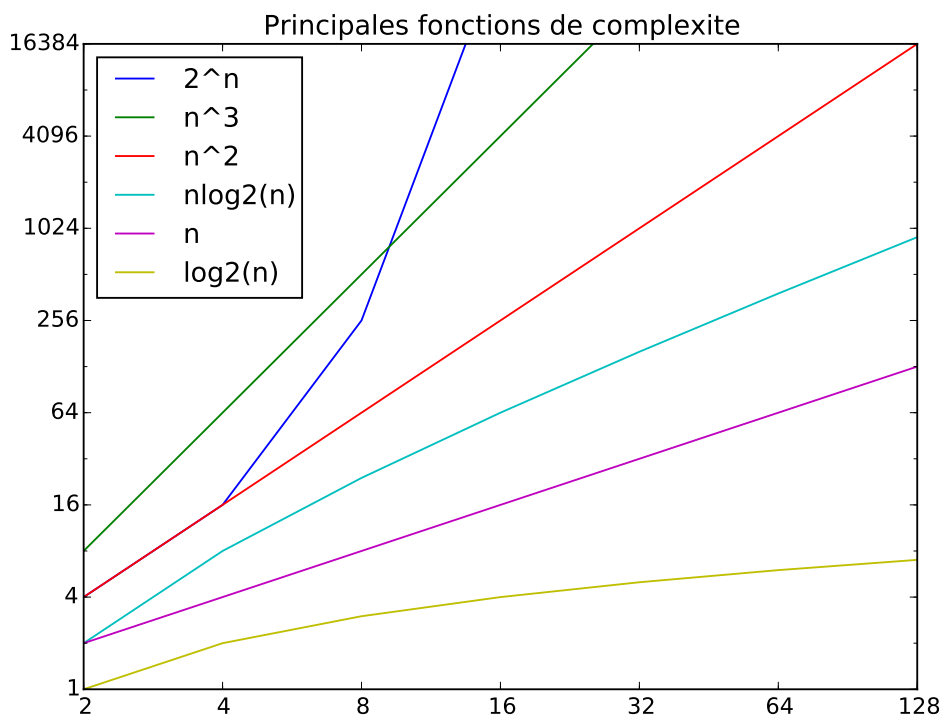
ou le chapitre 3 du *user guide* (qui comporte plus de 2500 pages).

Exercice 2. (P) Principales complexités.

Cet exercice permet d'utiliser les commandes de base pour :

- afficher de courbes : `plot`, `figure`, `title`, `legend`,
- gérer les axes : `xticks`, `xlim`, `loglog`, `semilogx`,
- gérer les affichages : `show`, `savefig`, `close`.

1. Obtenir le tracé suivant des principales classes de complexités : $\log_2 n$, n , $n \log_2 n$, n^2 , n^3 , 2^n pour les premières puissances de 2 (par exemple de 2 à 128). Des échelles logarithmiques sont adaptées.



```
# complexites.py
"""Trace des principales classes de complexites avec matplotlib"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

# premieres puissances de 2
n = [2**i for i in range(1,8)]

# les principales fonctions de complexites
## a l'aide de lambda fonctions
##iden = lambda x:x
##log2 = lambda x: np.log2(x)
##nlog2n = lambda x: x*log2(x)
##n2= lambda x: np.power(x, 2)
##n3 = lambda x: np.power(x, 3)
##fs = [log2, iden, nlog2n, n2, n3 ]
### leurs valeurs en x
```

```

##y = {} # y est un dictionnaire
##for f in fs:
##    y[str(f)] = f(n)

# trace
plt.figure(figsize=(8,6), dpi = 80)
plt.title("Principales fonctions de complexite")
# axes en log2
plt.loglog(basex=2, basey=2)
# valeurs affichees sur l axe des x
label_x = [str(val) for val in n]
plt.xticks(n, label_x)
# valeurs affichees sur l axe des y
y_val = [np.power(2, j) for j in range(0, 15, 2)]
label_y = [str(val) for val in y_val]
plt.yticks(y_val, label_y)
# on limite les valeurs de y trop grandes
plt.ylim(1.0, np.power(2,14))
#
plt.plot(n, np.power(2.0, n), label="2^n")
plt.plot(n, np.power(n, 3), label="n^3")
plt.plot(n, np.power(n, 2), label="n^2")
plt.plot(n, n*np.log2(n), label="nlog2(n)")
plt.plot(n, n, label="n")
plt.plot(n, np.log2(n), label="log2(n)")
#
plt.legend(loc="upper left")

# sortie ecran ou pdf
#plt.show()
pp = PdfPages('./plt-complexites.pdf')
pp.savefig()
pp.close()

```

Exercice 3. (P) Encore le flocon de Von Koch.

Il est plus rapide de calculer les coordonnées des sommets des transformés successifs du segment de départ, puis de les tracer comme une courbe. On obtient ainsi le tracé suivant.

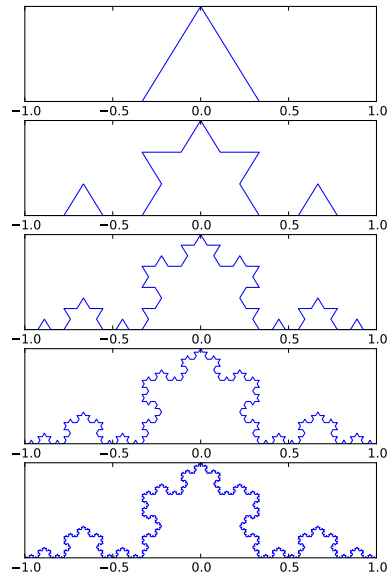
1. Écrire la fonction `segvk` qui à partir des extrémités d'un segment $[a, b]$, par exemple définies par ses affixes complexes, calcule et retourne la liste (des affixes) des points significatifs d'une itération de Von Koch de $[a, b]$.
2. Écrire la fonction `listevk` qui applique une itération de Von Koch à une liste de segments.
3. Appliquer ces fonctions pour obtenir le tracé `matplotlib` des premières itérations du segment $[-1, 1]$. Il est agréable d'obtenir ces tracés sous la forme de sous-figures comme présenté ci-dessus.

```

# plt-segment-vk.py
"""Flocon de Von Koch avec matplotlib"""
import cmath as cm
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

def segvk(za, zb):
    """coupe le segment [za,zb] en 3,
    retourne za et les sommets m1, m2, m3 de VonKoch"""
    d = zb-za #valeur complexe

```



```

dsur3 = d/3# idem
# w = rotation pi sur 3
pisur3 = cm.pi/3
w = complex(np.cos(pisur3), np.sin(pisur3))

zm1 = za + dsur3
zm3 = zm1 + dsur3
zm2 = zm1 + w*dsur3
return [za, zm1, zm2, zm3]

def listevk(L):
    """applique segvk a chaque paire successive de segments decrits dans L
    retourne cette liste Lout"""
    l = len(L)
    Lout = []
    for i in range(l-1):
        za = L[i]
        zb = L[i +1] # sommet suivant za
        Lout.extend(segvk(za, zb))
    Lout.append(zb) #ne pas oublier le dernier
    return Lout

# segment de depart
a = complex(-1,0)
b = complex(1,0)
L0 = [a,b]

# calcul nit niveaux de courbes stockees dans liste de listes
nit = 5
LL = [None]*(nit+1)
LL[0] = L0

```

```

for i in range(1, nit+1):
    print(i)
    LL[i] = listevk(LL[i-1])

# trace
plt.figure(figsize=(6,11)) # 1 figure
plt.suptitle('Flocon de Von Koch — version matplotlib')
plt.axis([-1,1,0, 0.6])
#plt.legend()

# gestion des sous-figures
numfig = 0
for i in range(1, nit+1):
    numfig +=1 # nit lignes, 1 col, i fig
    plt.subplot(nit+1, 1, numfig) #Integer subplot specification must be a
        three digit number
    plt.yticks([]) # rien en y

    # affichage
    plt.plot(np.real(LL[i]), np.imag(LL[i]))

# les sorties : commenter ce qui ne va pas
# on teste d'abord la sortie ecran
##plt.show()
###et quand on est content : la sortie pdf
pp = PdfPages('./plt-segvk.pdf')
pp.savefig()
### pas de sortie jpg dans matplotlib
pp.close()

```

2.3 Mesurer le temps d'exécution d'un algorithme

Mesurer le temps d'exécution d'un programme sur une machine actuelle est une tâche beaucoup plus ardue qu'il n'y paraît. En effet, l'impression que l'exécution de votre programme mobilise *à elle seule* les ressources de votre ordinateur est une illusion. Le temps que vous attendez pour obtenir votre résultat, même si cela vous paraît instantané, est en fait partagé : pendant ce même temps, votre ordinateur effectue certainement de nombreuses autres tâches, par exemple liées au système d'exploitation, aux périphériques, à d'autres utilisateurs si votre machine gère plusieurs sessions, ... Par ailleurs, les unités de calculs et l'organisation (des niveaux) de la mémoire compliquent très fortement la mesure fiable des temps d'exécution et leur analyse.

En pratique, plus ce que vous voulez mesurer est court, plus la mesure sera incertaine et non reproductible. Pour ce qui nous intéresse, l'obtention d'une mesure significative consistera à répéter l'exécution dans une boucle, de mesurer le temps d'exécution de *cette boucle* et d'en retenir la moyenne. Il n'est pas inutile de répéter ce type de mesures et le cas échéant de traiter l'échantillon obtenu.

Les "gros problèmes" sont aussi sujets à remarque. Des tailles de données importantes nécessitent un volume important de transferts entre la mémoire et les unités de calcul. Ces transferts sont complexes et leurs temps est, d'une part non linéaire par rapport à la taille des données, et d'autre part beaucoup plus important que les temps de calcul souvent associés. A titre indicatif, il y a un facteur 10 entre ce temps de transfert et le temps d'un calcul arithmétique élémentaire. Ainsi, les mesures peuvent être surprenantes lorsqu'elles deviennent "dominées" par ces temps de transfert. Le modèle de nos analyses de la complexité des algorithmes ne prend pas en compte cet aspect très difficile.

Pour finir, ceci illustre la différence entre les propriétés d'un algorithme et celles de ses mises en oeuvre et, par la même occasion, confirme l'intérêt des notations asymptotiques \mathcal{O} , θ , Ω dont on a souvent signalé qu'elle masquent pas mal de détails.

Mesurer avec le module `time` Ce module fournit beaucoup de fonctions en rapport avec le temps : gestions de calendriers, des systèmes horaires, d'horloges ... Il permet aussi une estimation des temps d'exécution des programmes python (et autres). La qualité de cette estimation dépend des mises en oeuvre de ce module et des machines visées. Vous pouvez être relativement confiant sur vos PC-linux ou mac.

La fonction `time()` est annoncée avec une précision de la seconde, sans que ce soit garanti pour toutes les machines. Une seconde est suffisamment long pour effectuer 2 giga = $2 \cdot 10^9$ opérations ... Elle peut donc être utile pour mesurer des simulations ou des exécutions conséquentes.

La fonction `perf_counter()` est définie depuis Python 3.3. Lorsqu'elle est disponible, elle permet les mesures les plus fines possibles adaptées à la mesure des temps d'exécution de programmes (sans être pour autant exactes, ni reproductibles). Elle utilise des fonctions spécifiques aux architectures des machines qui exploitent les compteurs matériels.

Ces fonctions sont très simples à utiliser. On procède en deux appels qui encadrent la portion de code à mesurer.

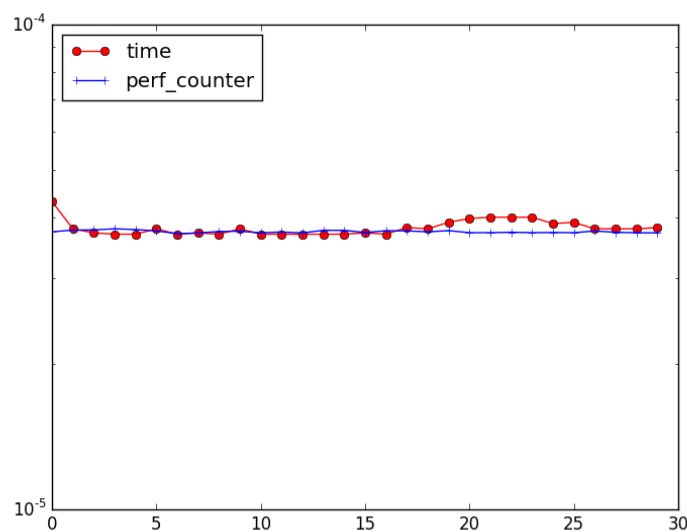
`t0 = time.perf_counter()` : le premier appel initialise une valeur initiale `t0` ;

`t = time.perf_counter()` : le second appel mesure `t` à l'issue de l'exécution de la portion de code,

et `t-t0` est la mesure de ce temps d'exécution (moyennant les réserves précédentes). Ne pas oublier de boucler autour de la portion à analyser et de mesurer à l'extérieur de cette boucle.

Exercice 4. (P) Compteurs de temps ...

Comparer `time` et `perf_counter` pour un algorithme linéaire.



```
"""comparaison time() et perf_counter() sur algo en O(n)"""
```

```

import time
from random import *
import numpy as np
import matplotlib.pyplot as plt

# formats affichage : fx.format(val)
fint = "{:0=5}" #entier sur 7 digits complete avec 0 a gauche
fexp = "{:7.3e}" #float en notation e sur 7 digits dont 3 apres le .
ffl = "{:5.1f}" #float sur 5 digits dont au plus 1 apres le .

# timings operations en O(n)
dim = 100
dimmax = dim

# 1. avec des listes python
a = [random() for i in range(dimmax)]
b = [random() for i in range(dimmax)]

nbrepet = 30
# timings operations en O(n)
t_time = []
for repet in range(nbrepet):
    t0 = time.time()
    c = 0.0
    for i in range(dim):
        c = c + a[i]*b[i]
    t = time.time()
    t_time.append(t-t0)

t_tsc = []
for repet in range(nbrepet):
    t0 = time.perf_counter()
    c = 0.0
    for i in range(dim):
        c = c + a[i]*b[i]
    t = time.perf_counter()
    t_tsc.append(t-t0)

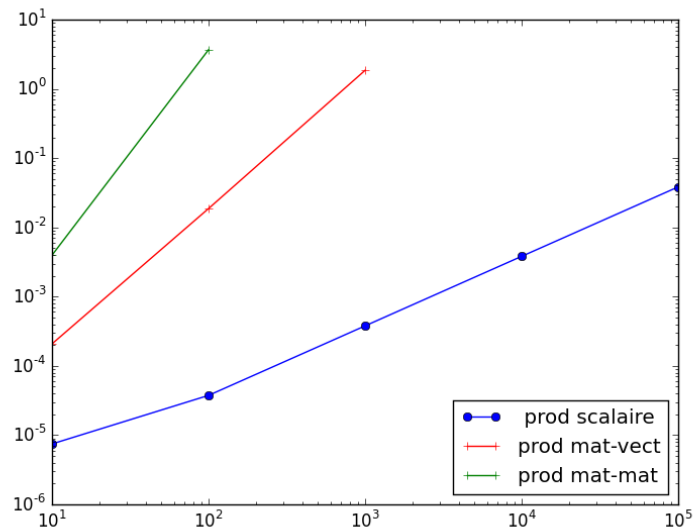
for r in range(nbrepet):
    print(fexp.format(t_time[r]) , fexp.format(t_tsc[r]))

### trace
plt.figure()
plt.semilogy(basey=10)
plt.plot(t_time, 'ro-', label="time")
plt.plot(t_tsc, 'b+- ', label="perf_counter")
#
plt.legend(loc='upper left')
plt.show()
plt.close()

```

Exercice 5. (P) Performance d'algorithmes de complexités diverses ...

Citer un algorithme d'algèbre linéaire pour chacune des complexités suivantes : linéaire, quadratique et cubique. Observer leurs temps d'exécution pour des ensembles de dimension correctement choisis. Proposer un tracé qui exhibe les complexités attendues.



```

""" timings de tableaux sur algos en O(n^1,2,3) avec listes python
    Rmq : les tailles pour n^2 (1000) et n^3 (100) sont les limites de ce
        macbook.
    A verifier car peu ... 10^4x10^4 = 10^8 soit 1/5 du max ci-apres.
    The maximum size of a python list on a 32 bit system is 536,870,912
        elements.
    (meme le cas 10^4 pour n^2, seul, ne passe pas) """

```

```

import time
from random import *
import numpy as np
import matplotlib.pyplot as plt

# formats affichage : fx.format(val)
fint = "{:0=5}" #entier sur 7 digits complete avec 0 a gauche
fexp = "{:7.3e}" #float en notation e sur 7 digits dont 3 apres le .
ffl = "{:5.1f}" #float sur 5 digits dont au plus 1 apres le .

dim = [10**i for i in range(1,2)]
dimmax = dim[ len(dim)-1]

#####
# timings operations en O(n)

# deux listes python
a = [random() for i in range(dimmax)]
b = [random() for i in range(dimmax)]

nbrepet = 10
# timings operations en O(n)
timings = []
for n in dim:
    t0 = time.perf_counter()
    for repet in range(nbrepet):
        # produit scalaire double boucle
        c = 0.0
        for i in range(n):
            c = c + a[i]*b[i]

```

```

t = time.perf_counter()
# timing moyen
tmoy = (t-t0)/nbrepet
timings.append(tmoy)

for n in range(len(dim)):
    print(fint.format(dim[n]), fexp.format(timings[n]))

#####
# timings operations en O(n^2)
dim2 = [10**i for i in range(1,4)]
#dim2 = dim
dimmax2 = dim2[len(dim2)-1]

# deux listes python
a = [random() for i in range(dimmax2)]
b = [random() for i in range(dimmax2)]

# une matrice
aa = [None]*dimmax2
for i in range(dimmax2):
    aa[i] = a

# un vecteur resultat
c = [0.0]*dimmax2

nbrepet2 = 3
# timings operations en O(n2)
timings2 = []
for n in dim2:
    t0 = time.perf_counter()
    for repet in range(nbrepet):
        # produit vect = matrice * vect avec double boucle
        for i in range(n):
            for j in range(n):
                c[j] = c[j] + aa[i][j]*b[j]
    t = time.perf_counter()
    # timing moyen
    tmoy = (t-t0)/nbrepet2
    timings2.append(tmoy)

for n in range(len(dim2)):
    print(fint.format(dim2[n]), fexp.format(timings2[n]))

#####
# timings operations en O(n^3)
dim3 = [10**i for i in range(1,3)]
#dim3 = dim
dimmax3 = dim3[len(dim3)-1]
nbrepet3 = 2
timings3 = []

# deux listes python
a = [random() for i in range(dimmax3)]
b = [random() for i in range(dimmax3)]

# deux autres matrices
bb = [None]*dimmax3
cc = [None]*dimmax3

```



```

for i in range(dimmax3):
    bb[i] = b
    cc[i] = c

# on y va
for n in dim3:
    t0 = time.perf_counter()
    for repet in range(nbrepet):
        # produit vect = matrice * vect avec double boucle
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    cc[i][j] = cc[i][j] + aa[i][j]*bb[j][k]
    t = time.perf_counter()
    # timing moyen
    tmoy = (t-t0)/nbrepet3
    timings3.append(tmoy)

for n in range(len(dim3)):
    print(fint.format(dim3[n]), fexp.format(timings3[n]))

### trace
plt.figure()
plt.loglog(basex=10, basey=10)
plt.plot(dim, timings, 'bo-', label="prod scalaire")
plt.plot(dim2, timings2, 'r+-', label="prod mat-vect")
plt.plot(dim3, timings3, 'g+-', label="prod mat-mat")
plt.legend(loc='lower right')
plt.show()
plt.close()

```


Deuxième partie

Exercices

Chapitre 3

Remise en jambe et récursivité

3.1 Premières récursions

Exercice 6. Exponentiation rapide (introduit en cours).

1. (P) Écrire `expo_rapide` à l'aide de `divmod`.
2. (P) Ajouter un `print` pour tracer les appels successifs.
3. Quels sont appels à `expo(x, n)` évités par `expo_rapide(x, n)`.

Exercice 7. Fibonacci (introduit en cours).

1. (P) Ajouter un `print` pour tracer les appels successifs.

Exercice 8. Écrivons récursif !

1. (P) Écrire une fonction qui calcule 2^n pour n entier.
2. (P) Écrire une fonction qui calcule la somme des n premiers entiers.
3. Quelles sont les complexités de ces fonctions ?

Exercice 9. Euclide récursif et itératif ... surtout.

1. (P) Écrire les versions récursive et itérative de l'algorithme d'Euclide vues en séance de façon à pouvoir comptabiliser (et tracer) les appels ou les itérations.
2. (P) Écrire des versions qui utilisent `divmod`.
3. Pour la version itérative :
 - (a) Montrer sa terminaison.
 - (b) Déterminer et prouver un invariant de boucle.
 - (c) Prouver sa correction.
 - (d) (★) Étudier sa complexité.

Exercice 10. Syracuse ... c'est loin ?

L'exemple classique d'une fonction récursive dont on ne sait pas démontrer la terminaison est la suite de Syracuse. En voici une première version ... économique.

$$\begin{aligned} s(0) &= s(1) = 1 \\ s(n) &= s(n/2) && \text{si } n \text{ est pair,} \\ s(n) &= s(3n + 1) && \text{si } n \text{ est impair.} \end{aligned}$$

1. Premiers contacts.

- (a) (P) Écrire une fonction qui calcule $s(n)$ et obtenir les 30 premiers termes de cette suite.
- (b) (P) Modifier ce programme pour afficher les valeurs intermédiaires nécessaires au calcul de $s(n)$. Observer raisonnablement.
- (c) (P) Modifier ce programme pour compter le nombre $c(n)$ de valeurs intermédiaires calculées dans $s(n)$.
- (d) (P*) Tracer $(n, s(n), c(n))$ pour $n \in \{1, 30\}$. Cette question nécessite d'utiliser *matplotlib*.
- (e) (P) Modifier le premier programme pour calculer et retourner la séquence de valeurs intermédiaires associée à $s(n)$.
2. Pour les curieux. Comme vous vous en doutez, on conjecture que pour tout $N > 0$, $s(N) = 1$. La suite u_n des valeurs intermédiaires calculées à partir de $s(N)$ est la suite de Syracuse associée à N . Elle s'écrit :

$$\begin{aligned}
 u_0 &= N \\
 u_{n+1} &= u_n/2 && \text{si } u_n \text{ est pair,} \\
 u_{n+1} &= 3u_n + 1 && \text{si } u_n \text{ est impair.}
 \end{aligned}$$

- (a) Tracer les valeurs intermédiaires pour un choix arbitraire de N . Observer en particulier les cas $N = 15$ et $N = 127$ — mais pas qu'eux ! Que se passe-t-il si $u_n = 1$?
- (b) Vu la forme précédente, plusieurs notions sont définies autour de cette suite (source Wikipédia).
- Le temps de vol est le plus petit indice n tel que $u_n = 1$.
 - Le temps de vol en altitude est le plus petit indice n tel que $u_n \leq N$.
 - L'altitude maximale est la valeur maximale u_n de la suite.
- Écrire des fonctions qui calculent ces notions pour un N arbitraire, puis jouer avec !

```

def s(n):
    """suite de Syracuse avec arrêts sur 0 et 1"""
    if n==0 or n == 1:
        return 1
    elif n%2 == 0 :
        return s(n//2)
    else :
        return s(3*n+1)

def s2(n):
    """suite de Syracuse avec arrêts sur 0 et 1"""
    """ version avec trace valeurs intermediaires """
    if n==0 or n == 1:
        print(n)
        return 1
    elif n%2 == 0 :
        u = n//2
        print("pair", u)
        return s2(u)
    else :
        u = 3*n+1
        print("impair", u)
        return s2(u)

def s3(n, c):
    """suite de Syracuse avec arrêts sur 0 et 1"""
    """ version avec decompote des valeurs intermediaires """
    if n==0 or n == 1:
        return 1, c+1

```

```

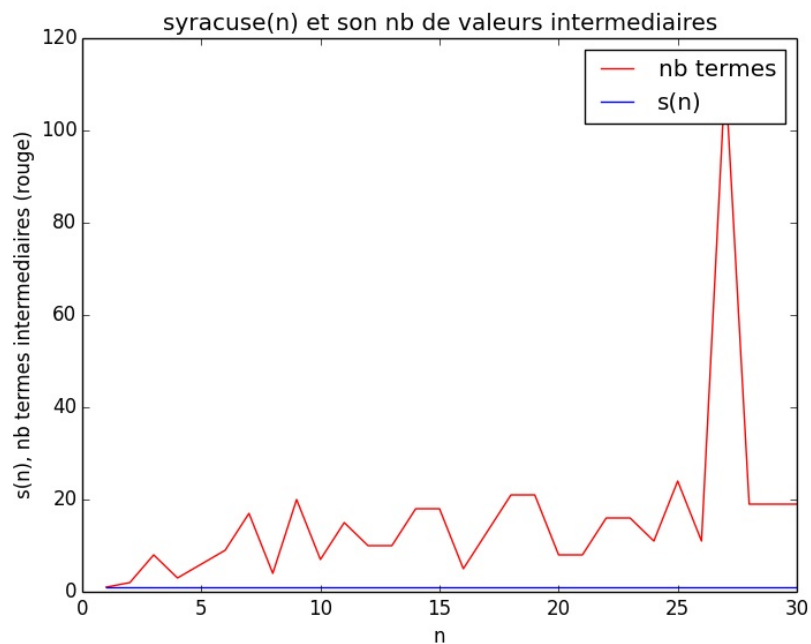
elif n%2 == 0 :
    return s3(n//2, c+1)
else :
    return s3(3*n+1,c+1)

##n = int(input("syracuse : nb de termes ? = "))
##for i in range(0, n):
##    print("-- n = ", i)
##    print("s(", i, ") = ", s(i))

n = int(input("syracuse : nb de termes ? = "))
for i in range(0, n):
    print("-- n = ", i)
    print("s(", i, ") = ", s2(i))

##n = int(input("syracuse(n) : n = "))
##print("--")
##print("s2(", n, ") = ", s2(n))
##res, nb_termes = s3(n, 0)
##print("s3(", n, ") = ", res, " et nb de termes intermediaires = ",
      nb_termes)

```



```

def s(a0):
    u, i = a0, 0
    while u != 1:
        q, r = divmod(u, 2)
        i += 1
        if r == 0:
            u = q
        else:
            u = 3*u+1
        print(u, i)
    return u, i

```

```
n = int(input("syracuse : nb de termes ? = "))
for i in range(0, n):
    print("s(", i, ") = ", s(i))
```

3.2 C'est vraiment parti !

Exercice 11. Décomposition binaire récursive.

La représentation binaire (aussi appelée la valeur binaire) d'un entier positif n est la suite de 0 et 1 (*bits*) notée $(n)_2 = b_p \cdots b_1 b_0 = \sum_0^p b_i 2^i$, où $b_i \in \{0, 1\}$. La suite de bits de la représentation binaire peut être stockée comme un tableau (ou une liste) de bits selon un ordre à préciser. Ainsi $(13)_2 = [1, 0, 0, 1]$ et $(16)_2 = [1, 0, 0, 0, 0]$ pour l'ordre décroissant des puissances de 2.

Bien sûr, la fonction python `bin(n)` calcule cette valeur et la retourne sous la forme d'une chaîne de bits (string en format binaire).

1. Propriétés préliminaires.

- (P) Écrire une fonction qui calcule p tel que 2^p est la plus grande puissance de 2 inférieure ou égale à n .
- Montrer que cette fonction termine.
- Déterminer et prouver un invariant de boucle.
- Quelle est la longueur minimale t du tableau nécessaire au stockage de la représentation binaire de l'entier positif n .
- Borner $n - 2^p$ pour $2^{p-1} \leq n < 2^p$.
- Écrire n en fonction de $n - 2^p$. En déduire une relation entre les tableaux qui stockent les valeurs binaire de n , 2^p et $n - 2^p$.

2. Version récursive.

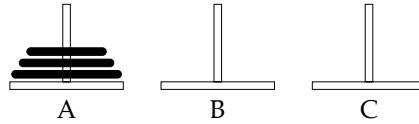
- Décrire les étapes d'un algorithme récursif, d'entrées deux entiers naturels n et t , qui calcule la valeur binaire de n et la retourne dans un tableau de longueur t . Dans un premier temps, on supposera la longueur t suffisante pour stocker des entiers positifs `int` codés en machine sur 32 bits.
- (P) Écrire cet algorithme sous la forme d'une fonction (récursive) python. Vérifier son comportement sur quelques exemples bien choisis.
- Prouver que cet algorithme termine et est correct.
- Borner le nombre d'appels récursif de cet algorithme. Expliciter les valeurs de n correspondantes.
- (P) Modifier le programme précédent pour observer ce nombre d'appels.
- (P) Modifier le programme initial pour que le tableau t soit de longueur arbitraire. Justifier votre choix.

Exercice 12. Les tours de Hanoï : première visite.

Le problème des *tours de Hanoï* (Édouard Lucas, 1883) consiste à trouver une stratégie pour déplacer une pile de n disques (troués en leur milieu) de diamètres tous différents ($n \geq 1$), d'une tour A vers une tour C en passant par une tour intermédiaire B. Au départ, les disques sont tous empilés sur la tour A par diamètre décroissant, comme indiqué sur la figure ci-dessous, pour $n = 3$.

Ensuite, les règles de déplacement sont les suivantes :

- (R1) un seul disque est déplacé à chaque déplacement,



(R2) un disque est empilé uniquement sur un disque de plus grand diamètre ou sur une tour vide.

Ce problème est résolu récursivement grâce à l'algorithme `Hanoi(n, src, dst, int)` qui déplace n disques de la tour `src` vers la tour `dst` en utilisant la tour intermédiaire `int` :

- Si $n = 1$: déplacer ce disque de la tour `src` vers la tour `dst`,
- Si $n \neq 1$:
 - 1) déplacer $(n - 1)$ disques de la tour `src` vers la tour `int` en passant par la tour `dst`,
 - 2) déplacer 1 disque de la tour `src` vers la tour `dst`,
 - 3) et déplacer $(n - 1)$ disques de la tour `int` vers la tour `dst` en passant par la tour `src`.

L'appel principal sera : `Hanoi(n, A, C, B)`.

1. Prise en main.

- (a) Pour $n = 3$, dérouler "à la main" l'algorithme `Hanoi(n, A, C, B)`. Expliciter l'arbre des appels récursifs et les déplacements successifs. Vérifier que le problème est ainsi résolu.
- (b) En déduire l'arbre des appels pour $n = 4$.
- (c) Combien de déplacements sont nécessaires à la résolution du problème pour $n = 3$? Et pour $n = 4$?
- (d) Expliciter des pré-conditions pour que `Hanoi` s'exécute correctement.

2. Codages et expérimentations.

- (a) (P) Coder une première version de l'algorithme récursif qui calcule et affiche la séquence de déplacements qui résolvent le problème des tours de Hanoï (sous la forme `A -> C, ...`).
- (b) (P) Coder une autre version de `Hanoi` afin i) de compter les nombres de déplacements de chaque disque, leur nombre total pour un n arbitraire (attention en pratique !) et d'afficher ces déplacements.
- (c) (P) Proposer et coder une dernière version de l'algorithme initial en remplaçant l'étape 2) du cas $n \neq 1$ par un appel à l'algorithme `Hanoi`.

3. Correction et terminaison. Soit $P_1(n)$ la propriété :

$P_1(n)$: l'algorithme `Hanoi(n, src, des, int)` déplace n disques de diamètre consécutif de la tour `src` vers la tour `des` en respectant les règles de déplacements (R1) et (R2).

- (a) Prouver la propriété $P_1(n)$, pour $n \geq 1$, et conclure que l'algorithme résout les tours de Hanoï.
- (b) Déduire également que l'algorithme termine.

4. Complexité.

- (a) Montrer que pour tout entier n , $\sum_{i=0}^n 2^i = 2^{n+1} - 1$.
- (b) On note c le coût du déplacement d'un disque d'une tour vers une autre, et $C(n)$ le coût de la résolution de `Hanoi(n, A, B, C)`. Donner l'expression du nombre $C(n)$.
- (c) (*) En utilisant les propriétés des opérateurs asymptotiques, montrez les propriétés suivantes.
 - i. Si $f(n) = 2^n + c$ avec $c \in \mathbb{R}$, alors $f = \Theta(2^n)$.
 - ii. Si $f_1 = \Theta(g_1)$ et $f_2 = \Theta(g_2)$ alors $f_1 \cdot f_2 = \Theta(g_1 \cdot g_2)$.
- (d) (*) En déduire la complexité asymptotique de l'algorithme `Hanoi`.

Les corrections des codes sont données à l'exercice 17.

Chapitre 4

Les piles

4.1 Exercices de base

Exercice 13. Piles à capacité bornée ou non (introduit en cours).

- (a) (P) Pour les piles à capacité bornée, écrire les fonctions `creer_pile`, `empiler`, `depiler`, `sommet`, `est_vide`, `taille` en veillant au bon traitement des cas exceptionnels.
(b) (P) Écrire `test_pile` qui utilise chacune de ces fonctions pour des arguments admissibles ou non.
- (a) Écrire les algorithmes de création, modification et observation de piles à capacité non bornée. Commenter les principales différences avec le cas borné.
(b) (P) Écrire les fonctions de la question 1a pour des piles à capacité non bornée.
(c) (P) Reprendre `test_pile` et observer le comportement des traitements exceptionnels.

```
# pilebornee.py
""" pile LIFO bornee — realisation par tableau """

def creer_pile(c):
    p = (c+1) * [None]
    p[0] = 0
    return p

# taille est tres vite utile (et remplace les lectures de p[0])
def taille(p):
    return p[0]

def depiler(p):
    assert taille(p) > 0, "depiler pile vide"
    p[0] = taille(p)-1

def empiler(p,e):
    assert taille(p) < len(p) -1, "empiler pile pleine" # = c
    p[0] = taille(p) + 1
    p[taille(p)] = e
    return p

def est_vide(p):
    return taille(p)==0

def sommet(p):
    assert taille(p) > 0, "sommet pile vide"
    return p[taille(p)]
```

```

# fonction utile mais pas de base
def afficher(p):
    for i in range(1, taille(p)):
        print(p[i], end=",")
    print(p[taille(p)]) # sans virgule finale

```

```

# pilenonbornee.py
""" pile LIFO non bornee — realisation listes python)"""

def creer_pile():
    return []

def depiler(p):
    assert len(p) > 0
    p.pop()

def empiler(p,e):
    return p.append(e)

def taille(p):
    return len(p)

def est_vide(p):
    return taille(p)==0

def sommet(p):
    assert taille (p) > 0
    return p[-1]

def afficher(p):
    #assert not(est_vide(p))
    for i in range(taille(p)):
        print(p[i], end=' ')
    print()

```

```

# piletst.py
""" Utilisation pile bornee ou non"""

# pour avoir les specs sous les yeux :
##def creer_pile(c):
##def depiler(p):
##def empiler(p,e):
##def taille(p):
##def est_vide(p):
##def sommet(p):
##def afficher(p):

# choix dynamique de la realisation bornee vs non bornee
bonnereponse = False
while not bonnereponse:
    rep = input("Choix pile a capacite bornee (b) ou non (n) : ?")
    if rep == 'b':
        bonnereponse = True
        print("pile a capacite bornee")
        from pilebornee import *
        pp = creer_pile(5) # cas borne
    elif rep == 'n':
        bonnereponse = True

```

```

        from pilenonbornee import *
        print("pile a capacite non bornee")
        pp = creer_pile() # cas non borne
    else:
        continue

# on y va
print("pile vide : "); print(taille(pp)); afficher(pp);
empiler(pp, 'A'); afficher(pp)
empiler(pp, 'B'); afficher(pp)
empiler(pp, 'C'); afficher(pp)
print("taille = ", taille(pp), "\n")

while not(est_vide(pp)):
    print("On depile : ", sommet(pp))
    depiler(pp) ;
    afficher(pp);
    print("taille =", taille(pp), "est_vide:", est_vide(pp))

print("L'appel suivant va lever une AssertionError (depiler pile vide)")
depiler(pp)

```

Exercice 14. Manipuler des piles.

1. (a) En utilisant exclusivement les fonctions de base des piles, écrire un algorithme qui duplique une pile et retourne cette copie.
 - (b) Évaluer le nombre d'appels aux fonctions de base. Évaluer le coût de l'occupation mémoire selon que le système optimise ou non cette occupation.
 - (c) (P) Écrire le programme `copier` correspondant.
2. (a) En utilisant exclusivement les fonctions de base des piles, écrire un algorithme retournant une pile qui contient, en ordre inverse, les éléments d'une pile `p` donnée, cette dernière n'étant pas modifiée (à l'issue du traitement).
 - (b) Écrire le programme `inverser` qui effectue ce traitement.

```

# manip.py
"""Manip de piles"""
from pilebornee import *

def copier(p):
    """q stocke la copie cherchée
    p est depilée et vidée pendant que
    r stocke les éléments de p depilés, donc en ordre inverse
    éléments ensuite re-empilés dans p et q"""
    q = creer_pile(taille(p))
    r = creer_pile(taille(p))
    while not est_vide(p):
        s = sommet(p)
        depiler(p)
        empiler(r, s)
    while not est_vide(r):
        s = sommet(r)
        depiler(r)
        empiler(p, s)
        empiler(q, s)
    return q

```

```

def inverser(p):
    """q duplique p : utilise copier(pile)
    r contient p en ordre inverse depilee de q"""
    q = copier(p)
    r = creer_pile(taille(p))
    while not est_vide(q):
        s = sommet(q)
        empiler(r, s)
        depiler(q)
    return r

def on_y_va():
    p = creer_pile(5)
    for i in range(5):
        empiler(p, 10+i)

    copie = copier(p)
    inv = inverser(p)

    #
    print("p", end=" : ")
    afficher(p)
    print("copie", end=" : ")
    afficher(copie)
    #
    print("p", end=" : ")
    afficher(p)
    print("inv", end=" : ")
    afficher(inv)

    print("RAPPEL : affecter n'est pas dupliquer (ni creer :)")
    print("p et encore_p = p : ")
    afficher(p)
    encore_p = p
    afficher(encore_p)
    print("on modifie p :")
    depiler(p)
    empiler(p, 99)
    print("p", end=" : ")
    afficher(p)
    print("encore_p ne devrait pas etre modifiee ...", end=" : ")
    afficher(encore_p)
    print("copie p n'est pas modifiee : ", end=" : ")
    afficher(copie)

#on_y_va()

```

Exercice 15. (P) Manipuler des piles (épisode 2).

Toujours en utilisant uniquement les fonctions de base des piles.

1. Écrire le programme `item` qui lit le n -ième élément d'une pile. La pile sera laissée inchangée et le cas où la pile n'est pas de taille suffisante sera traité.
2. Écrire le programme `tete_en_bas` qui place en fond de pile l'élément-sommet d'une pile non vide, les autres éléments restant dans l'ordre initial.

```

# manip-piles2.py
"""
Manipuler des piles : episode 2

```

```

"""
from pilebornee import *
from random import randint

##def creer_pile(c):
##def depiler(p):
##def empiler(p,e):
##def taille(p):
##def est_vide(p):
##def sommet(p):
##def afficher(p):

def item(p, n):
    """
    Retourne le n-ieme element de la pile p sans modifier cette derniere.
    Complexite en espace = O(n) car necessite de copier la pile
    """
    assert n <= taille(p), "n > taille(p)"
    q = creer_pile(n)
    for i in range(n):
        val = sommet(p)
        empiler(q, val)
        depiler(p)
    for i in range(n):
        empiler(p, sommet(q))
        depiler(q)
    return val

def tete_en_bas(p):
    """
    Place en fond de pile l'element-sommet d'une pile non vide,
    les autres elements restant dans l'ordre initial
    """
    assert not est_vide(p)
    s = sommet(p) # tete est conserve
    depiler(p) # tete inutile et maj taille(p)
    q = creer_pile(taille(p))
    while not est_vide(p):
        empiler(q, sommet(p))
        depiler(p)
    # q contient p (sauf tete) en ordre inverse
    empiler(p,s) # tete en bas
    while not est_vide(q):
        empiler(p, sommet(q))
        depiler(q)
    # pas de return

dim = 5
p = creer_pile(dim)
for e in range(dim):
    empiler(p, e)
print("p", end=" : ")
afficher(p)

# item
num = int(input("quel numero d'item dans p ? (0 pour finir) "))
while num > 0:
    print(item(p, num))

```

```

print("les numeros commencent en tete de pile :)")
num = int(input("quel numero d'item dans p ? (0 pour finir)"))

# tete en bas
print("p", end=" : ")
afficher(p)
tete_en_bas(p)
print("p tete en bas", end=" : ")
afficher(p)

```

Exercice 16. Deux piles et un seul tableau.

1. Proposer une réalisation de deux piles dans un seul tableau T de façon à ce qu'on puisse empiler au choix sur l'une d'entre elle tant que le tableau n'est pas plein.
2. (P) Écrire `empiler(e, b)` qui empile l'élément e sur l'une ou l'autre des piles selon la valeur du booléen b.

```

# deuxpiles.py
"""
Deux piles dans un seul tableau T
. g et d : deux piles
. T = [sommet g, vals de g, None, vals de d, sommet de d]
"""
from pilebornee import *
from random import randint

dim = 20
T = [0]*(dim+2)

def tailleg():
    return T[0]

def tailed():
    return T[-1]

def empiler_g(e):
    T[0] += 1
    T[tailleg()] = e

def empiler_d(e):
    T[-1] += 1
    T[-1-tailed()] = e

def empiler(e, b):
    assert tailleg() + tailed() < len(T)-2
    if b == False:
        empiler_g(e)
    else:
        empiler_d(e)

def init():
    for i in range(3):
        empiler_g(i)
    for i in range(3):
        empiler_d(i)
    for i in range(3):
        empiler_g(i+10)
    for i in range(3):

```



```

        empiler_d(i+20)

# test
init()
t = taille_g() + taille_d()
nb = [0,0] # compte le nb de valeur ajoutées a g et d
print(t, "valeurs stockées dans T pour un max de ", dim)
for i in range(dim-t):
    b = randint(0,1)
    nb[b] += 1
    empiler(int(b), b)
print("nb ajouts de 0 et 1 :", nb)
print(T)
# un de trop !
empiler(5,0)

```

4.2 Premières applications

Exercice 17. Les tours de Hanoï : seconde visite.

On revient sur les tours de Hanoï de l'exercice 12.

1. (P) Reprendre les programmes précédents pour calculer les états successifs des tours A, B, C au fur et à mesure de la résolution en utilisant le module de piles de l'exercice 13 pour représenter et manipuler les tours.
2. (P) Faire de même en comptant les nombres de déplacements de chaque disque, leur nombre total pour un n arbitraire et en affichant ces déplacements.

```

"""Hanoi : resolution et decompote des déplacements de 1 a 9 disques"""
import pile_bornee as p

# il est toujours déconseillé d'utiliser une variable globale :)
nbdep = 0
# voir hanoi3 pour une solution propre (ajout d'un paramètre nbdep dans
# hanoi)

def init_hanoi(nbdisk):
    g = p.creer_pile(nbdisk)
    m = p.creer_pile(nbdisk)
    d = p.creer_pile(nbdisk)
    for i in reversed(range(nbdisk)):
        p.empiler(g, i)
    return g, m, d

def hanoi(n, src, dest, interm):
    global nbdep
    if n==1:
        p.empiler(dest, p.sommet(src))
        p.depiler(src)
        #afficher_hanoi()
        print(src, " -> ", dest)
        nbdep += 1
    else:
        hanoi(n-1, src, interm, dest)
        hanoi(1, src, dest, interm)

```

```

        hanoi(n-1, interm, dest, src)

def afficher_hanoi():
    print("G : ", end='')
    p.afficher(g)
    print("M : ", end='')
    p.afficher(m)
    print("D : ", end='')
    p.afficher(d)

nbdisques = 3
res=[]
for n in range(1, nbdisques+1):
    nbdep = 0
    print("Hanoi avec ", n, " disques :")
    g, m, d = init_hanoi(n)
    print("depart")
    afficher_hanoi()
    hanoi(n, g, d, m)
    print("arrivee")
    afficher_hanoi()
    res = res + [(n, nbdep)]
    print("——")

print("(nb disques, nb déplacements)")
print(res)

```

```

"""Hanoi : detail des déplacements"""
import pilebornee as p

def init_hanoi(nbdisques):
    g = p.creer_pile(nbdisques)
    m = p.creer_pile(nbdisques)
    d = p.creer_pile(nbdisques)
    for i in reversed(range(nbdisques)):
        p.empiler(g, 10+i)
    return g, m, d

def hanoi_dep(n, src, dest, interm, src_name, dest_name, interm_name):
    """hanoi avec passage des identifiants des tours"""
    if n==1:
        print(p.sommet(src), ": sommet ", src_name, " -> ", dest_name)
        p.empiler(dest, p.sommet(src))
        p.depiler(src)
        #afficher_hanoi()
    else:
        hanoi_dep(n-1, src, interm, dest, src_name, interm_name, dest_name)
        hanoi_dep(1, src, dest, interm, src_name, dest_name, interm_name)
        hanoi_dep(n-1, interm, dest, src, interm_name, dest_name, src_name)

def afficher_hanoi():
    print("G : ", end='')
    p.afficher(g)
    print("M : ", end='')
    p.afficher(m)
    print("D : ", end='')
    p.afficher(d)
    print("————")

```

```

nbdisques = 4
g, m, d = init_hanoi(nbdisques)
print("depart")
afficher_hanoi()
hanoi_dep(nbdisques, g, d, m, "G", "D", "M")
print("arrivee")
afficher_hanoi()

```

```

"""Hanoi : decompte des deplacements par disque"""
import pilebornee as p

def init_hanoi(nbdisques):
    g = p.creer_pile(nbdisques)
    m = p.creer_pile(nbdisques)
    d = p.creer_pile(nbdisques)
    for i in reversed(range(nbdisques)):
        p.empiler(g, i)
    return g, m, d

def hanoi_dep(n, src, dest, interm, dep):
    """hanoi avec passage d'un tableau dep (val disque, nb dep)"""
    if n==1:
        dep[p.sommet(src)] += 1
        p.empiler(dest, p.sommet(src))
        p.depiler(src)
    else:
        hanoi_dep(n-1, src, interm, dest, dep)
        hanoi_dep(1, src, dest, interm, dep)
        hanoi_dep(n-1, interm, dest, src, dep)

def afficher_hanoi():
    print("G : ", end='')
    p.afficher(g)
    print("M : ", end='')
    p.afficher(m)
    print("D : ", end='')
    p.afficher(d)
    print("-----")

nbdisques = 5
deplacements = [0]*nbdisques
g, m, d = init_hanoi(nbdisques)
print("depart")
afficher_hanoi()
hanoi_dep(nbdisques, g, d, m, deplacements)
print("arrivee")
afficher_hanoi()
print("nb deplacements par disques :", deplacements)

```

Exercice 18. (P) Mots bien parenthésés.

1. (P) Écrire un programme `parentheses` qui vérifie la correction d'un mot parenthésé et qui affiche les paires des indices associés (ouvrant/fermant) comme vu en cours.
2. (P) Modifier le programme `parentheses` pour traiter des mots constitués de parenthèses "()", de crochets "[,]" et d'accolades "{,}". Le mot "([] {})" est bien parenthésé à la différence de "({ } [])".

3. (P) Modifier le programme `parentheses` pour traiter des mots constitués de parenthèses et d'autres caractères. Le mot "`3*(4-5*(6+7)-8)`" est bien parenthésé mais pas "`3*(4-5)*6`".

```
# parenthese.py
"""verifie le bon parenthesage d'un mot de ( et )
retourne un booleen et
imprime les indices ouvrant-fermant correspondants"""

from pilebornee import *

def verif(s):
    def parentheses(s):
        p = creer_pile(len(s))
        return p

    p = parentheses(s)
    for i in range(len(s)):
        if s[i] == '(':
            empiler(p, i)
        else:
            if est_vide(p):
                return False
            else:
                j = sommet(p)
                depiler(p)
                print((j,i), end=' ')
    return est_vide(p)

des_mots= [ '', '((( ( )))', '()()', '()()', '()()', '(((( ( ))))' ]
for s in des_mots:
    print(s, end=' : '), print(verif(s))
```

```
# parenthese2.py
""" verifie le bon parenthesage d'un mot de (, [, { et leurs fermants
retourne un booleen et
imprime les indices ouvrant-fermant correspondants"""
from pilebornee import *

def verif(s):
    def mot(s):
        """la pile p contient le mot"""
        p = creer_pile(len(s))
        return p

    p = mot(s)
    for i in range(len(s)):
        ch = s[i]
        if ch == '(' or ch=='{' or ch=='[':
            # ch est un ouvrant
            empiler(p, (ch, i))
        else:
            # ch est un fermant
            if est_vide(p):
                #pas assez de fermants
                return False
            else:
                # ch : fermant lu, i: son indice
                # ch0 : dernier ouvrant lu, j : son indice
```

```

        (ch0, j) = sommet(p)
        depiler(p)
        if ch=='(' and ch0=='(':
            print((ch0+ch, j, i), end=' ')
        elif ch=='}' and ch0=='{':
            print((ch0+ch, j, i), end=' ')
        elif ch==']' and ch0=='[':
            print((ch0+ch, j, i), end=' ')
        else:
            # ouvrant-fermant non correspondant
            return False
    return est_vide(p)

```

```

des_mots= [ '()[]{}', '() [({})]{{(())}}', '() []' ]
for s in des_mots:
    print(s, end=' : '), print(verif(s))

```

```

# parenthese3.py
""" verifie le bon parenthesage d'un mot de (, [, {, (et leurs fermants)
et de caracteres quelconques
retourne un booleen et
imprime les indices ouvrant-fermant correspondants """
from pilebornee import *

```

```

def verif(s):
    def mot(s):
        """la pile p contient le mot"""
        p = creer_pile(len(s))
        return p

    p = mot(s)
    for i in range(len(s)):
        ch = s[i]
        if ch == '(' or ch=='{' or ch=='[':
            # ch est un ouvrant
            empiler(p, (ch, i))
        elif ch == ')' or ch=='}' or ch==']':
            if est_vide(p):
                #pas assez de fermants
                return False
            # ch : fermant lu, i: son indice
            # ch0 : dernier ouvrant lu, j : son indice
            (ch0, j) = sommet(p)
            depiler(p)
            if ch=='(' and ch0=='(':
                print((ch0+ch, j, i), end=' ')
            elif ch=='}' and ch0=='{':
                print((ch0+ch, j, i), end=' ')
            elif ch==']' and ch0=='[':
                print((ch0+ch, j, i), end=' ')
            else:
                # ouvrant-fermant non correspondant
                return False
        else:
            # ch est un caractere quelconque
            continue
    return est_vide(p)

```

```

des_mots= [ '()[]{}', '() [({})]{{(())}}', '() []', '(3*[4-5]*{6+1})', \

```

```

    '(3*[4-5]*{6+1})', '(3*[4-5]*{6+1})', '3*(4-5)*(6+7)-8', \
    '3*(4-5)*6)'
for s in des_mots:
    print(s, end=' : '), print(verif(s))

```

4.3 Applications plus conséquentes

Exercice 19. Labyrinthe parfait.

Sur la grille discrète $[0, n-1] \times [0, n-1]$, on considère les n^2 noeuds $c = (i, j)$. Deux noeuds de la grille sont voisins si il existe une arête horizontale ou verticale les reliant. Un chemin entre deux noeuds c_1 et c_2 est la suite de noeuds successivement voisins $c_1, c_i, c_j, \dots, c_k, c_2$, d'extrémités c_1 et c_2 .

Un *labyrinthe parfait* est tel que, pour toute paire de noeuds de la grille, il existe un et un seul chemin entre ces noeuds. Le labyrinthe d'entrée c_1 et de sortie c_2 sera noté :

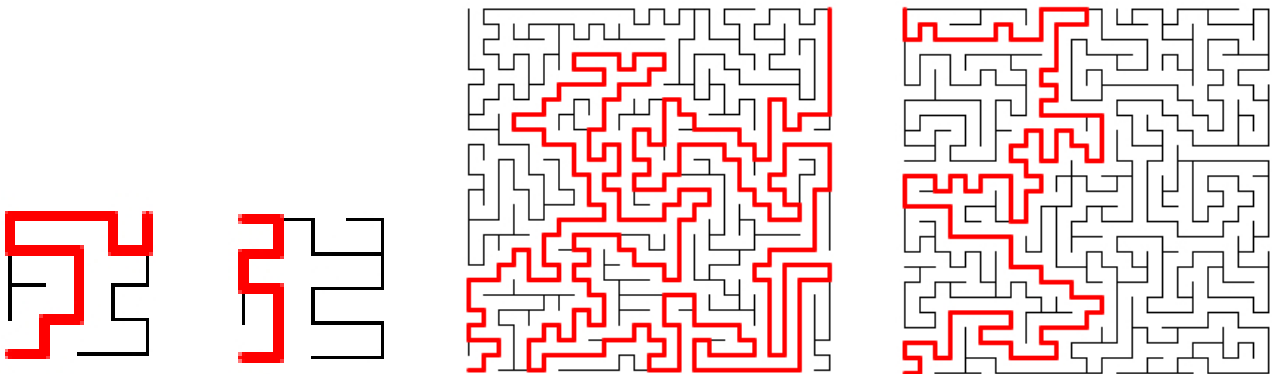
$$(c_1, c_i), (c_i, c_j), (c_i, c_k), \dots, (c_k, c_r), (c_j, c_l), (c_l, c_2)$$

ou aussi :

$$(c_1, c_i), (c_i, c_j), (c_j, c_l), (c_l, c_2), (c_i, c_k), \dots, (c_k, c_r).$$

Ici le chemin c_i, c_k, \dots, c_r où $r \neq 2$ mène à un cul-de-sac, et le chemin c_1, c_i, c_j, c_l, c_2 est l'unique chemin qui permet de sortir du labyrinthe.

Voici des exemples de labyrinthes parfaits de dimension 5×5 et 25×25 , et de chemins qui entrent au coin inférieur gauche $(0, 0)$ et sortent aux coins supérieurs droits ou gauche – resp. $(4, 4)$, $(24, 24)$ ou $(0, 4)$, $(0, 24)$.



À la différence des livre de jeux pour enfants, cheminer dans ces labyrinthes consiste à suivre les "lignes-chemins" plutôt qu'à se déplacer entre des "lignes-parois".

La construction d'un tel labyrinthe consiste à visiter une seule fois tous les noeuds de la grille. L'approche suivante profite de la structure de pile. On se donne :

- une grille `desVisites` de booléens pour identifier les noeuds déjà visités,
- une pile `p` des noeuds non encore visités,
- une pile `laby` qui stocke le labyrinthe en construction.

On part du noeud d'entrée, par exemple $(0, 0)$, que l'on empile sur `p` et que l'on marque dans `grilleDesVisites`.

Tant que la pile `p` n'est pas vide :

- i) on dépile le "noeud-sommet" c de la pile `p`,
- ii) on identifie ses noeuds voisins non encore visités et si il y en a au moins un :
 - . on en choisit un aléatoirement : s ,
 - .. on empile le chemin (c, s) sur `laby`,

... on empile c puis s sur p ,

... on marque s dans `grilleDesVisites` et on reprend.

1. Les questions suivantes vous accompagnent dans la mise en oeuvre de cet algorithme. La seule difficulté est d'éviter de sortir de la grille.
 - (a) Définir les trois structures de données `grilleDesVisites`, `p`, `laby` et des fonctions d'affichage adaptées.
 - (b) Écrire `visiter` qui met à jour la `grilleDesVisites`.
 - (c) Écrire `dejaVisite` qui retourne l'état de visite d'un noeud donné.
 - (d) Écrire `voisins` qui retourne la liste des noeuds voisins et non encore visité d'un noeud donné.
 - (e) Écrire `choisir` qui retourne aléatoirement le prochain noeud à visiter.
 - (f) Finalement, écrire `labyrinthe` qui construit et retourne un labyrinthe parfait sur une grille carrée de côté de longueur n arbitraire.
2. Utiliser le module `turtle` pour tracer ce labyrinthe.
3. (*) Compléter `labyrinthe` pour :
 - (a) qu'il retourne le chemin qui mène de l'entrée $(0, 0)$ jusqu'à la sortie en $(n - 1, n - 1)$,
 - (b) puis d'une entrée à une sortie arbitrairement choisies.

```
"""Labyrinthe parfait.
retourne le labyrinthe.
Version avec :
. choix de taille : grille carre de cote n
. verbose = 'pile': pile des cases visitees, 'res' : laby trouve, 'all' :
  tout
Appels :
. labyrinthe(5)
. labyrinthe(5, 'pile')
. labyrinthe(5, 'all')
"""
import random
from pilebornee import *

def aff(grille):
    """affiche le contenu d'une grille carree"""
    for i in range(len(grille)):
        print(grille[i])

def visiter(c, grille, verb):
    """ marque True dans visitees pour la case c visitée """
    (x,y) = c
    if x<0 or x>=len(grille) or y<0 or y>=len(grille[0]):
        # deplacement hors grille
        return
    grille[x][y] = True
    if verb=="all":
        aff(grille)

def deja_visitee(c, grille):
    """retourne valeur (True/False) de visitees pour c et True si hors
    grille (convention silencieuse)"""
    (x,y) = c
    if x<0 or x>=len(grille) or y<0 or y>=len(grille[0]):
```

```

        return True
    return grille[x][y]

def voisines(c, grille):
    """retourne la liste r des cases non visitees et voisines de c"""
    (x,y) = c
    r = []
    def ajouter(p):
        """ajoute a r une case non visitee"""
        if not deja_visitee(p, grille):
            r.append(p)
    ajouter((x-1,y))
    ajouter((x,y-1))
    ajouter((x+1,y))
    ajouter((x,y+1))
    return r

def choisir(L):
    """retourne une valeur de la liste L """
    l = len(L)
    assert l>0
    return L[random.randint(0, l-1)]

def labyrinthe(n, verb=False):
    """fonction principale
    . qui calcule le laby parfait dans une grille de cote n
    . qui retourne le laby comme une liste de couples reliant 2 cellules
      visitees
    . verb = False | 'pile' | 'res' | 'all' est un parametre de verboseite du
      calcul"""

    # grilleDesVisites est une grille de False (=non visitee) ou True
    grilleDesVisites = [[False] * n for i in range(n)]

    # les cases visitees qui ont peut-etre au moins une voisine a visiter
    pile = creer_pile(n*n)

    # laby : les chemins des cases visitees
    laby = creer_pile(n*n)

    # depart = en haut a gauche
    empiler(pile, (0,0))
    visiter((0,0), grilleDesVisites, verb)

    while not est_vide(pile):
        #on vient de visiter c
        c = sommet(pile)
        if verb=="all":
            print("on est en:", c)
        depiler(pile)
        lesvoisines = voisines(c, grilleDesVisites)
        # la liste des voisines de c non encore visitees
        if len(lesvoisines) > 0:
            # on n est pas dans un cul-de-sac...
            suivante = choisir(lesvoisines)
            #... donc on y va : suivante est une voisine de la case c
            if verb=="all":
                print("(choix) on va en:", suivante)
            # "relier" c a suivante dans laby

```



```

empiler(laby, (c, suivante))
# on conserve c pour visiter ses autres voisins au retour
empiler(pile, c)
# on continue a partir de suivante
empiler(pile, suivante)
visiter(suivante, grilleDesVisites, verb)
if verb=="pile":
    print("pile des cellules visitees :")
    afficher(pile)
else:
    if verb=="all":
        print(c, "n a plus de voisins non deja visitees")
if verb=="res":
    afficher(laby)

# on retourne le labyrinthe comme une liste de labys
# reliant 2 cellules visitees en partant de la derniere
ybal = []
while not est_vide(laby):
    ybal.append(sommet(laby))
    depiler(laby)
return ybal

```

```

"""Labyrinthe parfait.
retourne le labyrinthe et le laby de (0,0) vers (n-1, n-1).
Version avec :
. choix de taille : grille carree de cote n
. verbose = 'pile': pile des cases visitees, 'res' : laby trouve, 'all' :
    tout
Appels :
. labyrinthe(5)
. labyrinthe(5, 'pile')
. labyrinthe(5, 'all')
"""
import random
from pilebornee import *

def aff(grille):
    """affiche le contenu d'une grille carree"""
    for i in range(len(grille)):
        print(grille[i])

def visiter(c, grille, verb):
    """ marque True dans visitees pour la case c visitee """
    (x,y) = c
    if x<0 or x>=len(grille) or y<0 or y>=len(grille[0]):
        # deplacement hors grille
        return
    grille[x][y] = True
    if verb=="all":
        aff(grille)

def deja_visitee(c, grille):
    """retourne valeur (True/False) de visitees pour c et True si hors
    grille (convention silencieuse)"""
    (x,y) = c
    if x<0 or x>=len(grille) or y<0 or y>=len(grille[0]):
        return True
    return grille[x][y]

```

```

def voisins(c, grille):
    """retourne la liste r des cases non visitees et voisines de c"""
    (x,y) = c
    r = []
    def ajouter(p):
        """ajoute a r une case non visitee"""
        if not deja_visitee(p, grille):
            r.append(p)
    ajouter((x-1,y))
    ajouter((x,y-1))
    ajouter((x+1,y))
    ajouter((x,y+1))
    return r

def choisir(L):
    """retourne une valeur de la liste L """
    l = len(L)
    assert l>0
    return L[random.randint(0, l-1)]

def retour(grille, s, c):
    """Maj de grilleDesRetours : en s, on est arrive de c"""
    (x,y) = s
    grille[x][y] = c

def sortir(grille, entree, sortie):
    """retourne le chemin de la sortie vers l'entree"""
    ch = []
    c = sortie
    while c != entree:
        ch.append(c)
        (x,y) = c
        c = grille[x][y]
    return ch

def labyrinthe(n, entree, sortie, verb=False):
    """fonction principale
    . qui calcule un laby parfait dans une grille de cote n
    . qui retourne ce laby comme une liste de couples reliant 2 cellules
      visitees
    . qui retourne le chemin de l entree vers sortie
    . verb = False |'pile'|'res'|'all est un parametre de verbobite du
      calcul"""

    # grilleDesVisites est une grille de False (=non visitee) ou True
    grilleDesVisites = [[False] * n for i in range(n)]

    # les cases visitees qui ont peut-etre au moins une voisine a visiter
    pile = creer_pile(n*n)

    # laby : les chemins des cases visitees
    laby = creer_pile(n*n)

    # Retour(c) pour un pas en arriere vers l'entree
    grilleDesRetours = [(None,None)]*n for i in range(n)]

    # depart = en haut a gauche
    empiler(pile, (0,0))

```

```

visiter((0,0), grilleDesVisites, verb)

while not est_vider(pile):
    #on vient de visiter c
    c = sommet(pile)
    if verb=="all":
        print("on est en:", c)
    depiler(pile)
    lesvoisines = voisines(c, grilleDesVisites)
    # la liste des voisines de c non encore visitees
    if len(lesvoisines) > 0:
        # on n est pas dans un cul-de-sac...
        suivante = choisir(lesvoisines)
        #... donc on y va : suivante est une voisine de la case c
        if verb=="all":
            print("(choix) on va en:", suivante)
        # "relier" c a suivante dans laby
        empiler(laby, (c, suivante))
        # on conserve c pour visiter ses autres voisines au retour
        empiler(pile, c)
        # on continue a partir de suivante
        empiler(pile, suivante)
        visiter(suivante, grilleDesVisites, verb)
        retour(grilleDesRetours, suivante, c)
        if verb=="pile":
            print("pile des cellules visitees :")
            afficher(pile)
    else:
        if verb=="all":
            print(c, "n a plus de voisines non deja visitees")
if verb=="res":
    afficher(laby)

# calcul du chemin vers la sortie
#entree = (0,0)
#sortie = (n-1, n-1)
ch = sortir(grilleDesRetours, entree, sortie)
if verb=="all":
    print("e->s: ", ch)

# on retourne le labyrinthe comme une liste de labys
# reliant 2 cellules visitees en partant de la derniere
ybal = []
while not est_vider(laby):
    ybal.append(sommet(laby))
    depiler(laby)

# ybal et ch sont deux listes
return ybal, ch

```

```

"""Trace de labyrinthe parfait a l aide de turtle"""
import turtle
#from labyrinthe import labyrinthe
from labyrintheES import labyrinthe

# parametres deplacement
pas = 10

# n      0 1 2

```

```

# w . e      3 4 5
#   s        6 7 8
# la fontions ab construit la case
# de chemin a-b
# etat initial = etat final = angle nul = vers le nord
def sn():
    turtle.forward(pas)

def ns():
    turtle.backward(pas)

def ew():
    turtle.left(90)
    turtle.forward(pas)
    turtle.right(90)

def we():
    turtle.right(90)
    turtle.forward(pas)
    turtle.left(90)

# ^ y
# |
# |
# .---->x

##def etape_laby_sortie_vers_entree(t):
##     """interprete le chemin a-b, de b vers a selon n-w-s-e"""
##     xa, ya, xb, yb = t[0][0], t[0][1], t[1][0], t[1][1]
##     # on se positionne en b
##     turtle.up()
##     turtle.goto((xb*pas, yb*pas))
##     turtle.down()
##     # on trace une etape
##     if xa==xb: #n ou s
##         if ya > yb:
##             sn()
##         else:
##             ns()
##     else: # ya==yb: #e ou w
##         if xa > xb:
##             we()
##         else:
##             ew()

def etape_entree_vers_sortie(t):
    """interprete le chemin a-b, de a vers b selon n-w-s-e"""
    xa, ya, xb, yb = t[0][0], t[0][1], t[1][0], t[1][1]
    # on se positionne en a
    turtle.up()
    turtle.goto(xa*pas, ya*pas)
    turtle.down()
    # on trace une etape
    if xa==xb: #n ou s
        if ya > yb:
            ns()
        else:
            sn()

```

```

else: # ya==yb: #e ou w
    if xa > xb:
        ew()
    else:
        we()

##def etape_chemin_E_vers_S(t):
##    """interprete le chemin a-b selon n-e-s-w"""
##    xa, ya, xb, yb = t[0][0], t[0][1], t[1][0], t[1][1]
### on se positionne en a
##    turtle.up()
##    turtle.goto((xa*pas, ya*pas))
##    turtle.down()
##    # on trace une etape
##    if xa==xb: #n ou s
##        if ya > yb:
##            sn()
##        else:
##            ns()
##    else: # ya==yb: #e ou w
##        if xa > xb:
##            we()
##        else:
##            ew()

#parcours = ((0, 0), (1, 0)),((1, 0), (2, 0)),((2, 0), (2, 1)),((2, 1), (1,
1)),((1, 1), (0, 1)),((0, 1), (0, 2)),((0, 2), (0, 3)),((0, 3), (1, 3))
,((1, 3), (2, 3)),((2, 3), (3, 3)),((3, 3), (3, 2)),((3, 2), (2, 2))
,((2, 2), (1, 2)),((3, 2), (3, 1)),((3, 1), (3, 0))

n = int(input("Taille de la grille carree : "))
#
print("1. calcul chemin")
entree =(0,0)
sortie =(n-1,n-1)
(parcours, chemin_es) = labyrinthe(n, entree, sortie)
#print(parcours)
#print(chemin_es)

# trace
print("2. trace")
turtle.reset()
turtle.title("Labyrinthe parfait — Turtle")
l, h = 800, 680 # taille fenetre turtle
turtle.screensize(l,h)
turtle.hideturtle() # pour aller plus vite
turtle.speed(0)
turtle.left(90) # on regarde le nord

# tracer le labyrinthe du depart vers la sortie : ordre inverse de parcours
for e in reversed(parcours):
    etape_entree_vers_sortie(e)

# tracer le chemin gagnant
turtle.pencolor("red")
turtle.pensize(3)
turtle.hideturtle() # pour aller plus vite
turtle.up()

```

```

depart = (0,0)
turtle.goto(depart)
a = depart
for b in reversed(chemin_es):
    etape_entree_vers_sortie((a,b))
    a = b

turtle.mainloop()

```



Exercice 20. (P) La magie des coupes et des mélanges d'un jeu de cartes.

Nous étudions des manipulations de cartes, d'apparences magiques, proposées par Gilbreath entre 1958 et 1989. Pour cela nous utilisons trois articles récents de Lachal et Schott parus dans *Quadrature* en 2012 et 2013 — chercher “cartomagie, Lachal, Schott” sur le net.

1. Toujours en utilisant uniquement les fonctions de base des piles.

(a) Écrire le programme `couper` qui prend une pile `p` et coupe ses n éléments de tête, $0 < n < \text{taille}(p)$, et les regroupe dans le même ordre en une seconde pile, c'est-à-dire comme lorsqu'on coupe et “ré-empile” un jeu de cartes. On ajoute le cas $n = 0$ pour que la valeur effective de n soit choisie au hasard dans $\llbracket 0, \text{taille}(p) \rrbracket$. Par exemple la pile $[0, 1, 2, 3, 4]$ est coupée en $[0, 1, 2]$ et $[3, 4]$ si $n = 2$ (la tête de pile est la valeur de droite dans cette notation sous forme de liste).

(b) Écrire le programme `melanger` qui prend en argument deux piles `P1` et `P2` et retourne une troisième pile obtenue à partir du tirage aléatoire d'éléments-sommet de `P1` ou `P2`, et ce tant que ces piles le permettent — ces deux piles finissent vides. Ce mélange sera rendu stable : l'ordre relatif des éléments des piles initiales est conservé dans la pile mélangée.

Les mélanges $[3, 0, 1, 4, 2]$ et $[0, 1, 3, 2, 4]$ sont par exemple obtenus à partir des deux piles de la coupe précédente. Les fonctions `copier` et `inverser` de l'exercice 14 peuvent être utilisées.

Un tel mélange est appelé *mélange à la queue d'aronde* ou *mélange américain* dans les références mentionnées.

(c) `test` : écrire un programme qui teste `couper` et `melanger` pour des valeurs arbitraires des paramètres proposées par l'utilisateur, et ce tant qu'il le désire.

2. Lachal et Schott énoncent le premier principe de Gilbreath comme suit.

“Si un jeu de cartes classé en rouges et noires alternées une à une est coupé en deux paquets, avec une carte noire sur la face de l'un des paquets et une carte rouge sur la face de l'autre, et si ces deux paquets sont mélangés l'un dans l'autre au moyen d'un mélange à la queue d'aronde, alors chaque paire de cartes consécutives du jeu sera composée d'une carte rouge et d'une carte noire.”

`magiel` : construire un paquet de cartes en empilant p fois les mêmes paires de cartes. Par exemple, $p = 16$ paires rouge-noir pour construire un paquet de 32 cartes à deux couleurs successivement alternées.

`couper` ce paquet en deux paquets tels que leurs têtes soient de couleur différente : par exemple en choisissant n impair.

`melanger` ces paquets et observer.

Magie : comme annoncé par Gilbreath, le paquet final contient toujours p blocs de paires de cartes, ces paires pouvant apparaître dans un ordre différent. Ici : 16 paires rouge-noir ou noir-rouge.

Il pourra être commode de pouvoir afficher le contenu d'une pile (d'un jeu de cartes) par paquets de k éléments consécutifs.

3. Lachal et Schott rapportent le "truc" suivant pour ne pas contraindre le résultat de la coupe précédente, *i.e.* imposer des têtes de couleur différente.

"Si l'on préfère laisser le spectateur couper le jeu faces en bas sans se soucier de la couleur des cartes qui se trouvent sur la face des deux paquets coupés, il suffit, une fois le mélange effectué, de procéder à l'ajustement suivant : prenez le jeu, retournez-le faces en haut, [...], repérez deux cartes de même couleur, [...], coupez au milieu de cette paire et complétez la coupe [...]."

magie2 : Modifier magie1 en magie2 de façon à vérifier que ce "truc" fonctionne correctement. "Compléter la coupe" signifie "ré-empiler" les 2 paquets après la coupe.

4. Le second principe de Gilbreath dit :

"Si deux séries de cartes classées en ordre inverse l'une de l'autre sont mélangées ensemble avec un mélange à la queue d'aronde, les deux moitiés du groupe qui en résulte seront chacune composées de cartes similaires à celles des séries originales."

- (a) magie3 : choisir r lettres et construire deux paquets composés de deux séries : r minuscules et r majuscules. Inverser la seconde série. Mélanger les deux séries et vérifier que le second principe de Gilbreath s'applique.
- (b) Modifier magie3 pour ne manipuler qu'un seul paquet de "cartes".
- (c) Modifier magie3 pour vérifier que si le principe reste vrai après un nombre arbitraire de coupes et de mélanges. Justifier votre observation.

5. Le principe précédent a été généralisé par Hudson (1966).

"Lorsqu'un nombre quelconque de cartes sont classées en séries répétées, elles peuvent être mélangées entre elles au moyen d'un mélange à la queue d'aronde sans altérer le contenu de chaque série, à condition que les cartes de l'un des deux paquets qui doit être mélangé avec l'autre soient inversées."

magie4 : proposer une vérification de cette généralisation.

```
# magiepile.py
"""
Manipuler des cartes avec des piles : episode magique de Gilbreath
"""
from pilebornee import *
from manippiles import inverser, copier
from random import randint

def couper(p, n):
    """coupe un jeu de cartes
    en un paquet de taille n>0 et son complement
    si n==0, n est choisi au hasard
    """
    assert not est_vide(p), "pile vide"
    assert n < taille(p), "taille de coupe trop grande"
    if n==0:
        n = randint(1, taille(p))
        print("taille aleatoire = ",n)
    q = creer_pile(n)
    for i in range(n):
        empiler(q, sommet(p))
    depiler(p)
```

```

return inverser(q)

def melanger(p1, p2):
    """
    melange americain (insertions successives apres coupe)
    prend en argument deux piles p1 et p2
    retourne une troisieme pile obtenue a partir
    du tirage aleatoire d'elements-sommet de p1 ou p2
    """
    q = creer_pile(taille(p1)+taille(p2))
    while not(est_vide(p1) or est_vide(p2)):
        # on a le choix : p1 et p2 non vide
        un_ou_deux = randint(1,2)
        if un_ou_deux == 1:
            p = p1
        else:
            p = p2
        empiler(q, sommet(p))
        depiler(p)
    # on n'a plus le choix : on vide ce qui reste
    while not est_vide(p1):
        empiler(q, sommet(p1))
        depiler(p1)
    while not est_vide(p2):
        empiler(q, sommet(p2))
        depiler(p2)
    # on veut retrouver la tete des paquets en haut du melange
    return inverser(q)

def aff(p, k):
    """affiche p par groupe de k>0 elements"""
    assert k>0
    q = copier(p)
    while not est_vide(q):
        for i in range(k):
            print(sommet(q), end=" , ")
            depiler(q)
        print()

def test():
    n = 0
    while n >=0:
        dim = 8
        p = creer_pile(dim)
        for e in range(dim):
            empiler(p, e)
        print("p", end=" : ")
        afficher(p)
        # couper
        if n==0:
            print("coupe au hasard")
            print("on coupe")
        q = couper(p, n)
        print("p", end=" : ")
        afficher(p)
        print("q", end=" : ")
        afficher(q)
        # melanger

```



```

    print("on melange")
    r = melanger(p,q)
    print("melange(p,q)", end=" : ")
    afficher(r)
    #aff(r, 2)
    print("taille de la coupe (<", taille(p)," et -1 pour arreter) : ?"
        )
    n = int(input(""))

def magie1():
    """Gilbreath 1 : coupe dirigee et melange d'un jeu a deux couleurs
    alternees"""
    dim = 32
    k = 2
    cartes = creer_pile(dim)

    onjoue = True
    n = 0 # premier tirage au hasrad
    while onjoue:
        for e in range(dim//k):
            empiler(cartes, 'R')
            empiler(cartes, 'N')
        afficher(cartes)
        if n==0:
            print("on coupe au hasard:")
        else:
            print("on coupe")
        tas = couper(cartes, n)
        print(taille(cartes), taille(tas))
        afficher(cartes)
        afficher(tas)
        print("on melange")
        jeu = melanger(cartes, tas)
        print(taille(jeu), taille(cartes), taille(tas))
        aff(jeu, k)
        val = input("On recommence (O/N) ?")
        if val=='O':
            onjoue == True
            print("taille de la coupe (<", dim,") : ?")
            n = int(input(""))
        else:
            onjoue = False

def magie2():
    """Gilbreath 1 + truc = coupe et melange d'un jeu a deux couleurs
    alternees"""
    dim = 32
    k = 2
    cartes = creer_pile(dim)

    onjoue = True
    n = 0 # premier tirage au hasard
    while onjoue:
        for e in range(dim//k):
            empiler(cartes, 'R')
            empiler(cartes, 'N')
        afficher(cartes)
        if n==0:
            print("on coupe au hasard:")

```

```

else:
    print("on coupe")
# coupe
tas = couper(cartes , n)
print(taille(cartes) , taille(tas))
afficher(cartes)
afficher(tas)
print("on melange")
# melange
jeu = melanger(cartes , tas)
print(taille(jeu) , taille(cartes) , taille(tas))
aff(jeu , k)
# coupe de rattrapage
print("coupe magique")
tastmp = creer_pile(taille(jeu))
tete = sommet(jeu)
empiler(tastmp , tete)
depiler(jeu)
while tete != sommet(jeu):
    empiler(tastmp , sommet(jeu))
    depiler(jeu)
    tete = sommet(tastmp)
# jeu et tas ont deux tetes de mm couleur : on complete la coupe
inverser(tastmp)
while not est_vide(jeu):
    empiler(cartes , sommet(jeu))
    depiler(jeu)
while not est_vide(cartes):
    empiler(tastmp , sommet(cartes))
    depiler(cartes)
aff(tastmp , k)
#
val = input("On recommence (O/N) ?")
if val=='O':
    onjoue == True
    print("taille de la coupe (< , dim , ) : ?")
    n = int(input(""))
else:
    onjoue = False

def magie3():
    """ Gilbreath 2 """
# minusc = "abcd"
# majusc = "ABCD"
minusc = "abcdefgh"
majusc = "ABCDEFGH"
r = len(minusc)

jeu = creer_pile(2*r)
for c in minusc:
    empiler(jeu , c)
for c in reversed(majusc):
    empiler(jeu , c)
afficher(jeu)
jeudesecours = copier(jeu)
print("on coupe au milieu")
tas = couper(jeu , r)
afficher(jeu)
afficher(tas)

```

```

print("on melange ")
res = melanger(jeu , tas)
afficher(res)
#
onjoue = True
while onjoue == True:
    print("on coupe et on melange encore ")
    jeu = res
    tas = couper(jeu , r)
    afficher (jeu)
    afficher(tas)
    res = melanger(jeu , tas)
    print("Caramba : encore manque !")
    afficher(res)
    val = input("On recommence (O/N) ?")
    if val=='O':
        onjoue == True
    else:
        onjoue = False

def magie4():
    serie = [x for x in range(5)]
    m = 3 # la serie est repetee 2 m fois
    cartes = creer_pile(2*m*len(serie))

    # preparation du jeu magique
    for i in range(m):
        for e in serie:
            empiler(cartes , e)
    for i in range(m):
        for e in reversed(serie):
            empiler(cartes , e)
    afficher(cartes)
    # coupe et melange
    print("on coupe :")
    tas = couper(cartes , m*len(serie))
    afficher(cartes)
    afficher(tas)
    print("on melange")
    jeu = melanger(cartes , tas)
    afficher(jeu)
    print("Magique !")
    aff(jeu , len(serie))
    # on recommence a loisir
    onjoue = True
    while onjoue == True:
        cartes = jeu
        print("encore : on coupe :")
        tas = couper(cartes , m*len(serie))
        afficher(cartes)
        afficher(tas)
        print("encore : on melange")
        jeu = melanger(cartes , tas)
        afficher(jeu)
        print("Caramba ! ")
        aff(jeu , len(serie))
        #
        val = input("On recommence (O/N) ?")
        if val=='O':

```

```
    onjoue == True
else:
    onjoue = False
```

Chapitre 5

Trier

5.1 Les tris vus en cours

Exercice 21. À la main

Traiter les questions suivantes pour chacun des algorithmes vus en cours : tri insertion, tri fusion et tri rapide.

1. Détailler leur “déroulement à la main” sur le vecteur $[15, 4, 2, 8, 17, 23, 0, 1]$. Comptabiliser les comparaisons et les affectations et comparer aux résultats du cours.
2. Proposer un exemple de pire et de meilleur cas pour un vecteur d’entrée de taille 8.

Exercice 22. (P) Tri insertion

1. Écrire `triinsertion` vu en cours.
2. Que peut-on dire de la complexité du tri insertion lorsqu’on double le nombre de valeurs à trier ?
3. Observations expérimentales des complexités.
 - (a) Modifier `triinsertion` pour compter le nombre de comparaisons.
 - (b) Écrire des fonctions qui génèrent les meilleurs et les pires cas du tri insertion pour un nombre de valeurs à trier (dimension) arbitraire.
 - (c) Écrire une fonction qui génère un échantillon de cas quelconques de dimension arbitraire.
 - (d) Écrire `testcomptriinsertion` pour illustrer les complexités vues en cours. Le résultat attendu est un tracé `matplotlib` pour des dimensions qui doublent de 100 à 6400 (par exemple).
 - i. Complexités des meilleurs et des pires cas ;
 - ii. complexité moyenne obtenues sur des échantillons de cas quelconques de façon à illustrer le résultat vu en cours.
4. Reprendre la question précédente en observant les temps d’exécutions du tri insertion. Proposer un tracé qui exhibe les résultats de la question 2.

```
# triinsertion.py
from random import randint

def triinsertion(t):
    for i in range(1, len(t)):
        v = t[i]
        j = i
        while j > 0 and v < t[j-1]:
```

```

        t[j] = t[j-1]
        j = j-1
    t[j] = v
    return t

# un tableaux de valeurs aleatoires
def test():
    dim = 5
    a = [randint(0,50) for i in range(dim)]
    print(a)
    print("Trions !")
    b = triinsertion(a)
    print(a)
    print(b)
    print("attention : affecter un tab n'est pas creer un nouveau tab")

# pire et meilleur cas
def pirecasinsertion(n):
    a = [x for x in reversed(range(n))]
    return a

def meilleurcasinsertion(n):
    a = [x for x in range(n)]
    return a

```

```

# triinsertion.py
from random import randint
from numpy import log
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

valmax = 99

def triinsertion(t):
    for i in range(1, len(t)):
        v = t[i]
        j = i
        while j > 0 and v < t[j-1]:
            t[j] = t[j-1]
            j = j-1
        t[j] = v
    return t

def comptriinsertion(t):
    """ version avec decompte nb comparaisons """
    c = 0
    for i in range(1, len(t)):
        v = t[i]
        j = i
        c += 1 # au moins une (derniere) comparaison
        while j > 0 and v < t[j-1]:
            t[j] = t[j-1]
            j = j-1
            c += 1
        t[j] = v
        if j == 0:
            c = c-1 # sauf si c'est j qui arrete la boucle
    return t, c

```

```

# un tableaux de valeurs aleatoires
def tabalea(n):
    a = [randint(0, valmax) for i in range(n)]
    return a

def pirecasinsertion(n):
    a = [x for x in reversed(range(n))]
    return a

def meilleurcasinsertion(n):
    a = [x for x in range(n)]
    return a

def testcomp10():
    n = 10
    a = tabalea(n)
    w = pirecasinsertion(n)
    b = meilleurcasinsertion(n)
    tabs = [b, w, a]
    print("Trions !")
    for t in tabs:
        print(t)
        res, c = comptriinsertion(t)
        print(res)
        print(c, "comparaisons pour n = ", n)

def testcomp():
    #dims = (10, 50)
    dims = (10, 50, 100, 200, 300, 500, 1000)
    nbcompalea = []*len(dims)
    nbcompbest = []*len(dims)
    nbcompworst = []*len(dims)
    for n in dims:
        # meilleur cas
        b = meilleurcasinsertion(n)
        res, c = comptriinsertion(b)
        nbcompbest.append(c)
        #print("au moins ", c, "comparaisons pour n = ", n)
        # pire cas
        w = pirecasinsertion(n)
        res, c = comptriinsertion(w)
        nbcompworst.append(c)
        #print("au plus ", c, "comparaisons pour n = ", n)
        # une moyenne de p cas aleatoires
        p = 10
        ctot = 0
        for t in range(p):
            a = tabalea(n)
            res, c = comptriinsertion(a)
            #print(res)
            #print(c, "comparaisons pour n = ", n)
            ctot += c
        cmoy = int(ctot/p)
        nbcompalea.append(cmoy)
        #print(cmoy, "comparaisons en moyenne pour n = ", n, p, "
            echantillons")
    return dims, nbcompbest, nbcompworst, nbcompalea

```

```

def plottestcomp():
    # calculs
    n, b, w, m = testcomp()

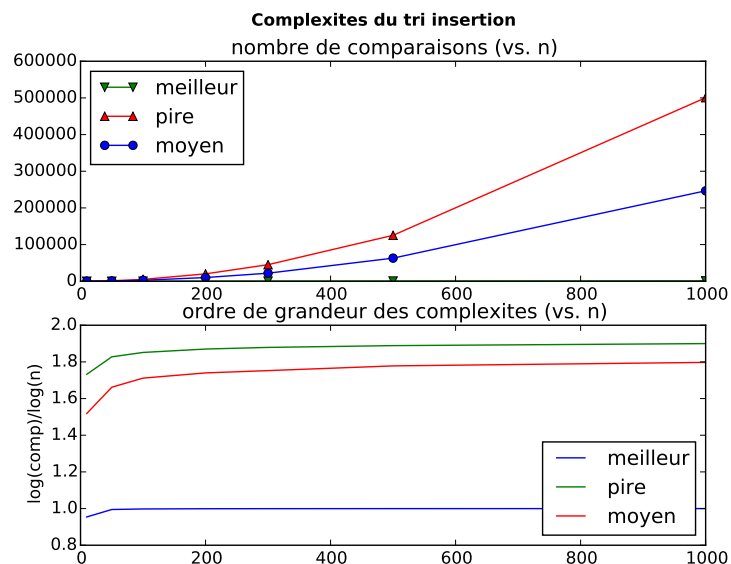
    # traces
    fig = plt.figure()
    fig.suptitle("Complexites du tri insertion", fontweight='bold')

    # echelle lineaire
    #fig1 = fig.add_subplot(1,1,1)
    plt.subplot(211)#2lignes, 1 col, fig1
    plt.title("nombre de comparaisons (vs. n)")
    plt.plot(n, b, 'g-v', label="meilleur")
    plt.plot(n, w, 'r-^', label="pire")
    plt.plot(n, m, 'b-o', label="moyen")
    plt.legend(loc=2) # legende en haut a gche

    # echelles log
    plt.subplot(212)
    plt.title("ordre de grandeur des complexites (vs. n)")
    #plt.semilogy()
    plt.ylabel("log(comp)/log(n)")
    plt.plot(n, log(b)/log(n), label="meilleur")
    plt.plot(n, log(w)/log(n), label="pire")
    plt.plot(n, log(m)/log(n), label="moyen")
    plt.legend(loc=4)

    # sortie ecran ou pdf
    #plt.show()
    pp = PdfPages('./plt-comp-triinsertion.pdf')
    pp.savefig()
    pp.close()

```



```

# testtimetriinsertion.py
"""Mesure et trace des timings mesures du tri insertion
. time vs. n
. ordre de complexite : t(2n)/t(n)"""

```



```

from random import randint
import time
from numpy import log
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
import pickle as pk

valmax = 99

def triinsertion(t):
    for i in range(1, len(t)):
        v = t[i]
        j = i
        while j > 0 and v < t[j-1]:
            t[j] = t[j-1]
            j = j-1
        t[j] = v
    return t

# un tableau de valeurs aleatoires
def tabalea(n):
    a = [randint(0, valmax) for i in range(n)]
    return a

def pirecasinsertion(n):
    a = [x for x in reversed(range(n))]
    return a

def meilleurcasinsertion(n):
    a = [x for x in range(n)]
    return a

def timer_tri(dims):
    """time tri insertion (best, worst, aver) pour taille n in dims"""
    #
    timing_alea = []*len(dims)
    timing_best = []*len(dims)
    timing_worst = []*len(dims)
    for n in dims:
        # meilleur cas
        b = meilleurcasinsertion(n)
        t0 = time.time()
        c = triinsertion(b)
        t = time.time() - t0
        timing_best.append(t)
        #print("au moins ", t, " pour n = ", n)
        # pire cas
        w = pirecasinsertion(n)
        t0 = time.time()
        c = triinsertion(w)
        t = time.time() - t0
        timing_worst.append(t)
        #print("au plus ", t, " pour n = ", n)
        # une moyenne de p cas aleatoires
        p = 10
        tot = 0
        for essai in range(p):

```

```

        a = tabalea(n)
        t0 = time.time()
        c = triinsertion(a)
        t = time.time() - t0
        #print(t , " pour n = ", n)
        tot += t
    tmoy = tot/p
    timing_alea.append(tmoy)
    #print(tmoy, " en moyenne pour n = ", n, p, " echantillons")
return dims, timing_best, timing_worst, timing_alea

def save_time(n, t1, t2, t3, fichier):
    """save dans fichier les dims,
    les timings de meilleurs, pires et moyens cas"""
    # on transforme en t2D
    t = [n, t1, t2, t3]
    f = open(fichier, "wb")
    pk.dump(t, f)
    f.close()

def read_time(fichier):
    """recuperation n et timings best, worst, avrg"""
    f = open(fichier, "rb")
    vals = pk.load(f)
    n = vals[0]
    t1 = vals[1]
    t2 = vals[2]
    t3 = vals[3]
    f.close()
    return n, t1, t2, t3

def plot_time(n, b, w, m):
    """n = les dims,
    b, w, m = timings best, worts, avg pour les dims"""
    # traces
    fig = plt.figure()
    fig.suptitle("Complexites du tri insertion", fontweight='bold')

    # echelle lineaire
    #fig1 = fig.add_subplot(1,1,1)
    plt.subplot(211)#2lignes, 1 col, fig1
    plt.title("timing vs. n")
    plt.plot(n, b, 'g-v', label="meilleur")
    plt.plot(n, w, 'r-^', label="pire")
    plt.plot(n, m, 'b-o', label="moyen")
    plt.legend(loc=2) # legende en haut a gche

    # ratios
    plt.subplot(212)
    plt.title("timings : facteur d'augmentation entre t(2n) et t(n)")
    #plt.semilogy()
    plt.ylabel("t(2n)/t(n)")
    # np.array commode pour div vecteurs
    bb = np.array(b)
    ww = np.array(w)
    mm = np.array(m)
    print(bb,ww,mm)
    nbval = len(n)
    nt = np.array(n[1:nbval]) # n sans la premiere valeur pour ratios

```

```

ratio_b = bb[1:nbval]/bb[0:nbval-1]
ratio_w = ww[1:nbval]/ww[0:nbval-1]
ratio_m = mm[1:nbval]/mm[0:nbval-1]
print(ratio_b, ratio_w, ratio_m)
plt.plot(nt, ratio_b, 'g-v', label="meilleur")
plt.plot(nt, ratio_w, 'r-^', label="pire")
plt.plot(nt, ratio_m, 'b-o', label="moyen")
plt.legend(loc=4)

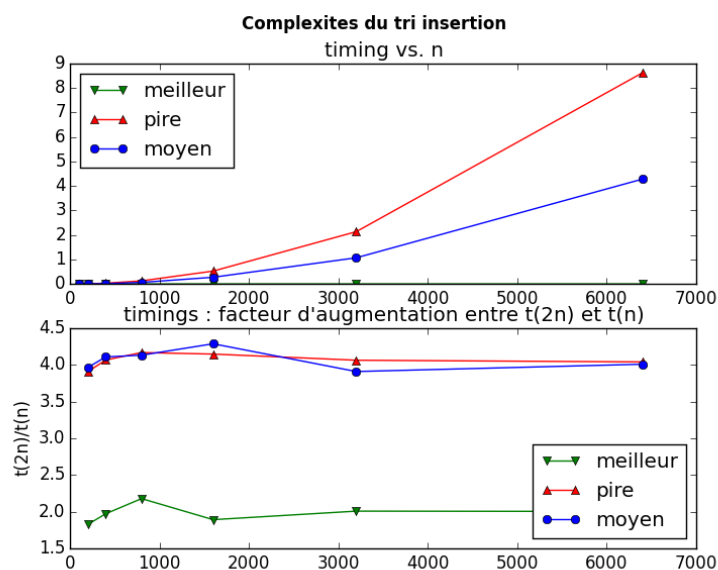
# sortie ecran ou pdf
plt.show()
#pp = PdfPages('./plt-times-triinsertion.pdf')
#pp.savefig()
#pp.close()

# tailles tableaux a trier
#dims = (10, 50)
dims = (100, 200, 400, 800, 1600, 3200, 6400)

# tris et save
n, b, w, m = timer_tri(dims)
#save_time(n, b, w, m, "timingsinsertion")

# load et trace
#n, b, w, m = read_time("timingsinsertion")
plot_time(n, b, w, m)

```



Exercice 23. (P) Tri fusion

1. Que peut-on dire de la complexité du tri fusion lorsqu'on double le nombre de valeurs à trier ?
2. Écrire les différentes versions de fusion vues en cours. On suppose que les arguments sont des (sous-)tableaux de longueur arbitraire mais triés.
 - (a) `fusion0` qui fusionne les tableaux `t1` et `t2` en un tableau `t`.
 - (b) `fusion1` qui fusionne les sous-tableaux `t[d, m]` et `t[m, d]` avec recopie et sentinelles.
 - (c) `fusion2` qui fusionne les sous-tableaux `t[d, m]` et `t[m, d]` avec recopie.

- (d) Dans chacun des cas, écrire `testfusion` adapté à des vérifications.
3. (a) Écrire `trifusion` basé sur l'une des fusions précédentes.
 - (b) Écrire une version de `trifusion` qui explicite la succession des appels récursifs.
 - (c) Écrire `testtrifusion` pour se rassurer.
 4. (a) Modifier `trifusion` pour compter le nombre de comparaisons et d'affectations (des éléments du tableau d'entrée).
 - (b) À la manière des questions 3 et 4 de l'exercice 22, écrire `testcomptrifusion` et `testtimestrifusion` pour illustrer les complexités (pire cas, meilleur cas, en moyenne) vues en cours.
 5. Analyse plus fine : fusion avec recopie et sentinelles *vs.* fusion avec recopie seulement.
 - (a) Écrire `trifusion1` et `trifusion2` qui utilisent respectivement les fusions avec copie locales et sentinelles ou sans ces dernières, resp. `fusion1` et `fusion2` de la question 2.
 - (b) Modifier chacune de ces versions pour compter le plus précisément possible nombre de comparaisons.
 - (c) Reprendre la question 4b en observant plus spécialement les comparaisons dans le pire cas et en moyenne. Faire le lien avec ce qui a été présenté en cours.
 6. Reprendre la question précédente en observant les temps d'exécutions du tri fusion. Proposer un tracé qui exhibe les résultats de la question 1.

```
# trifusion.py
from random import randint

# rappel
def verifCopieSousTab(t, g, m, d):
    """Affecter un tab ne cree pas un nouveau tab
    MAIS affecter une tranche d'un tab cree un tab"""
    # copies locales G et D
    tt = t
    tg = t[g: m]
    td = t[m: d]
    print(t)
    print(tt)
    print(tg)
    print(td)

    t[g] = 10
    t[d-1] = 99
    print(t)
    print(tt)
    print(tg)
    print(td)

#fusion : 3 versions
def fusion0(t1, t2):
    t = [None]*(len(t1) + len(t2))
    i1 = i2 = k = 0
    while i1 < len(t1) and i2 < len(t2):
        if t1[i1] < t2[i2]:
            t[k] = t1[i1]
            i1 += 1
        else:
            t[k] = t2[i2]
            i2 += 1
        k += 1
```

```

if i1 == len(t1):
    for j in range(i2, len(t2)):
        t[k] = t2[j]
        k += 1
else:
    for j in range(i1, len(t1)):
        t[k] = t1[j]
        k += 1
return t

def fusion1(t, g, m, d):
    """ fusion avec sentinelles sur tableaux locaux """
    infini = 9999
    # copies locales G et D avec sentinelles
    G = t[g:m] + [infini]
    D = t[m:d] + [infini]
    # leurs indices de tete : G et D commencent a 0 !!
    i1 = i2 = 0
    # on peut ranger dans t maintenant
    for k in range(g, d): # k parcourt t
        if G[i1] < D[i2]:
            t[k] = G[i1]
            i1 += 1
        else:
            t[k] = D[i2]
            i2 += 1
    return t

def fusion2(t, g, m, d):
    """ fusion sur tableaux locaux """
    # copies locales G et D
    G = t[g:m]
    D = t[m:d]
    # leurs indices de tete
    i1 = i2 = 0
    # on peut ranger dans t maintenant
    for k in range(g, d):
        if i1 < len(G) and (i2==len(D) or G[i1] < D[i2]):
            # G n est pas vide et tete G < tete D : on prend la tete de G
            t[k] = G[i1]
            i1 += 1
        else: # G est vide ou si (D n est pas vide et tete G > tete D)
            # on prend la tete de D
            t[k] = D[i2]
            i2 += 1
    return t

# test fusion 0, 1, 2
from triinsertion import triinsertion
def testfusion():
    # deux tableaux de taille differente et de valeurs aleatoires
    dim1 = 5
    dim2 = 4
    a = [randint(0,50) for i in range(dim1)]
    b = [randint(0,50) for i in range(dim2)]
    # trie de chaque sous-tab
    a = triinsertion(a)
    b = triinsertion(b)
    print(a)

```

```

print(b)
c = a + b      # c est pret
print(c)
# fusion
#c = fusion0(a,b)
#fusion1(c, 0, dim1, dim1+dim2)
fusion2(c, 0, dim1, dim1+dim2)
print(c)

# tri fusion : version 0 et test
def trifusion0(t, g, d):
    """ version avec 3 parametres :
    terminaison de la recursion :
    . m est strictement decroissante jusqu'a valoir g ou d
    . si t ne contient qu'une seule valeur alors il n'y a rien a faire, il
      est trie"""
    if d > g + 1:
        # t contient au moins deux valeurs
        m = (g+d)//2
        trifusion0(t, g, m)
        trifusion0(t, m, d)
        #fusion1(t, g, m, d) # avec sentinelles
        fusion2(t, g, m, d) # sans sentinelle
    else: # rien a faire
        return # pour etre plus explicite

def trifusion(t):
    """ version avec le tab t comme seul parametre """
    trifusion0(t, 0, len(t))

def trifusion0_verbose(t, g, d, s):
    """ version avec affichage arbre des appels
    s : niveau indentation des appels"""
    if g < d - 1:
        # t contient au moins deux valeurs
        m = (g+d)//2
        s = s+4
        indent = s*" " #chaine de s blancs

        print(indent, "trifusion(", g, ", m, ")")
        trifusion0_verbose(t, g, m, s)

        print(indent, "trifusion(", m, ", d, ")")
        trifusion0_verbose(t, m, d, s)

        print(indent, "fusion(", g, ", m, d, ")")
        fusion1(t, g, m, d)
        #fusion2(t, g, m, d)

    else :
        s = s-2
        return

def testtrifusion(verb=False):
    # un tableau de valeurs aleatoires
    dim1 = 8
    a = [randint(0,10) for i in range(dim1)]
    print(a)

```

```

# tri fusion
if verb == True:
    print("trifusion(", 0, len(a), ")")
    trifusion0_verbos(a, 0, len(a), 0)
else:
    trifusion(a)
print(a)

```

```

# testcomptrifusion.py
from random import randint
from numpy import log, log2
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

def fusion1(t, g, m, d, cmp, aff):
    """ fusion avec sentinelles sur tableaux locaux
    version avec compteurs comparaison et deplacement
    compteurs mis a jour et renvoyes """
    infini = 9999
    # copies locales G et D avec sentinelles
    G = t[g: m] + [infini]
    D = t[m: d] + [infini]
    aff = aff + len(G) + len(D) - 2
    # leurs indices de tete : G et D commencent a 0 !!
    i1 = i2 = 0
    # on peut ranger dans t maintenant
    for k in range(g, d): # k parcourt t
        if G[i1] < D[i2]:
            t[k] = G[i1]
            i1 += 1
            cmp += 1
            aff += 1
        else:
            t[k] = D[i2]
            i2 += 1
            cmp += 1
            aff += 1
    return cmp, aff

def fusion2(t, g, m, d, cmp, aff):
    """ fusion sur tableaux locaux """
    # copies locales G et D
    G = t[g: m]
    D = t[m: d]
    aff = aff + d - g
    # leurs indices de tete
    i1 = i2 = 0
    # on peut ranger dans t maintenant
    for k in range(g, d):
        if i1 < len(G) and (i2 == len(D) or G[i1] < D[i2]):
            # G n est pas vide et tete G < tete D : on prend la tete de G
            t[k] = G[i1]
            aff += 1
            if i1 < len(G) and i2 == len(D): # cas ou G[i1] < D[i2] est
                evalue
                cmp += 1
            i1 += 1
        else: # G est vide ou si (D n est pas vide et tete G > tete D)
            # on prend la tete de D

```

```

        t[k] = D[i2]
        aff += 1
        if i1 == len(G) or i2 < len(D) :
            cmp += 1
            i2 += 1
    return cmp, aff

def trifusion(t, g, d, cmp, aff, sentinelles):
    """ cmp et aff transmis et renvoyes
    choix version avec ou sans sentinelle """
    if sentinelles == True:
        fusion = fusion1
    else:
        fusion = fusion2

    if d > g + 1:
        # t contient au moins deux valeurs
        m = (g+d)//2
        cmp, aff = trifusion(t, g, m, cmp, aff, sentinelles)
        cmp, aff = trifusion(t, m, d, cmp, aff, sentinelles)
        cmp, aff = fusion(t, g, m, d, cmp, aff)
        #print(cmp, aff)
    # rien a faire
    return cmp, aff

def comptrifusion(t, sentinelles):
    """ version classique avec decompte """
    cmp, aff = 0, 0
    cmp, aff = trifusion(t, 0, len(t), cmp, aff, sentinelles)
    return cmp, aff

# un tableaux de valeurs aleatoires
valmax = 99
def tabalea(n):
    a = [randint(0, valmax) for i in range(n)]
    return a

def pirecasinsertion(n):
    a = [x for x in reversed(range(n))]
    return a

def meilleurcasinsertion(n):
    a = [x for x in range(n)]
    return a

##def testcomp10():
##    n = 10
##    a = tabalea(n)
##    w = pirecasinsertion(n)
##    b = meilleurcasinsertion(n)
##    tabs = [b,w,a]
##    print("Trions !")
##    t2 = [None]*n
##    for t in tabs:
##        t2[0:len(t)] = t # copie non trie !
##        print(t)
##        cmp, aff = comptrifusion1(t)
##        #print(t)

```



```

##         print("v1: ", cmp , "comparaisons pour n = ", n)
##         print(aff , "affectations pour n = ", n)
##         #print(t2)
##         cmp, aff = comprifusion2(t2)
##         #print(t2)
##         print("v2 :", cmp , "v2 : comparaisons pour n = ", n)
##         print(aff , "affectations pour n = ", n)
##         print(" ")
##     print("pour info : nlog2(n) = ", n*log2(n))

def testcomp(sentinelles):
    """choix version avec ou sans sentinelles"""
    #dims = (10, 100)
    #dims = (10, 50, 100, 200, 300, 500, 1000)
    dims = (100, 200, 400, 800, 1600, 3200, 6400)

    nbcompalea = []*len(dims)
    nbcompbest = []*len(dims)
    nbcompworst = []*len(dims)
    nbaffalea = []*len(dims)
    nbaffbest = []*len(dims)
    nbaffworst = []*len(dims)

    # choix version tri fusion
    f = comprifusion

    for n in dims:
        # meilleur cas
        b = meilleurcasinsertion(n)
        comp, aff = f(b, sentinelles)
        nbcompbest.append(comp)
        nbaffbest.append(aff)
        #print("au moins ", c , "comparaisons pour n = ", n)
        # pire cas
        w = pirecasinsertion(n)
        comp, aff = f(w, sentinelles)
        nbcompworst.append(comp)
        nbaffworst.append(aff)
        #print("au plus ", c , "comparaisons pour n = ", n)
        # une moyenne de p cas aleatoires
        p = 10
        ctot, atot = 0, 0
        for t in range(p):
            a = tabalea(n)
            comp, aff = f(a, sentinelles)
            #print(res)
            #print(c , "comparaisons pour n = ", n)
            ctot += comp
            atot += aff
        cmoy = int(ctot/p)
        amoy = int(atot/p)
        nbcompalea.append(cmoy)
        nbaffalea.append(amoy)
        #print(cmoy, "comparaisons en moyenne pour n = ", n, p, "
            echantillons")
    return dims, nbcompbest, nbcompworst, nbcompalea, nbaffbest, nbaffworst,
        nbaffalea

```

```

def plottestcomp(sentinelles , ecran=False):
    """choix version avec ou sans sentinelles
    essai ecran ou pdf"""
    # calculs
    n, b, w, m, ab, aw, am = testcomp(sentinelles)
    nlogn = [x*log2(x) for x in n]

    def le_plot(subplt):
        # comparaisons : trait continu
        subplt.plot(n, b, 'g-v', label="meilleur")
        subplt.plot(n, w, 'r-^', label="pire")
        subplt.plot(n, m, 'b-o', label="moyen")
        # nb affectations : trait pointille
        subplt.plot(n, ab, 'g-v', label="meilleur")
        subplt.plot(n, aw, 'r-^', label="pire")
        subplt.plot(n, am, 'b-o', label="moyen")
        # complexite theorique : diamant
        subplt.plot(n, nlogn, 'yD', label='nlog2(n)')
        #
        #subplt.set_title("nombre de comparaisons (-) et d'affectations (-) (vs. n)")
        subplt.set_xlabel('n')
        subplt.set_ylabel('nb comp. ')

    plt.close('all')
    # traces
    fig = plt.figure(figsize=(6,12))

    # partage de la fig en 3+1 subfig
    sub1= plt.subplot(411) # 3 lignes + 1 pour place legend, 1 col, sous-fig 1
    sub2= plt.subplot(412) # 3 lignes, 1 col, sous-fig 2
    sub3= plt.subplot(413) # 3 lignes, 1 col, sous-fig 3

    # echelle lineaire
    le_plot(sub1)
    if sentinelles == True:
        sub1.set_title("Complexites du tri fusion — version avec sentinelles", fontweight='bold', fontsize=14)
    else:
        sub1.set_title("Complexites du tri fusion — version sans sentinelle", fontweight='bold', fontsize=14)

    # echelle log2
    le_plot(sub2)
    sub2.loglog(basex=10, basey=10)
    sub2.set_title("idem en echelles log10", fontsize=12)

    # axes en log10
    le_plot(sub3)
    sub3.set_title("idem en echelles log2", fontsize=12)
    sub3.loglog(basex=2, basey=2)

    # legend en dessous
    # bbox_to_anchor [left, bottom, width, height]
    # bbox_to_anchor [left, bottom]
    #plt.legend(bbox_to_anchor=(1.05, 1), loc='upper right',
    #           ncol=1, borderaxespad=0., fontsize=14)

```

```

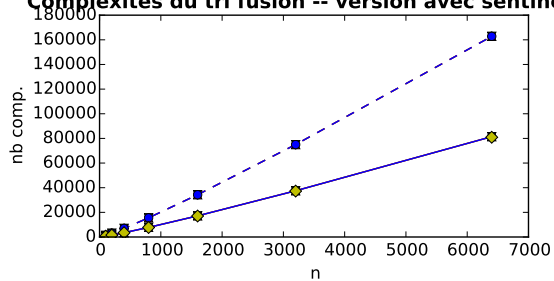
plt.legend(bbox_to_anchor=(0, -1.3, 1, -0.2), loc='lower left',
           ncol=3, mode="expand", borderaxespad=0., fontsize=14)

plt.tight_layout(True) # pour aerer entre les subfig
#plt.subplots_adjust(top=0.95) # pour le supitle

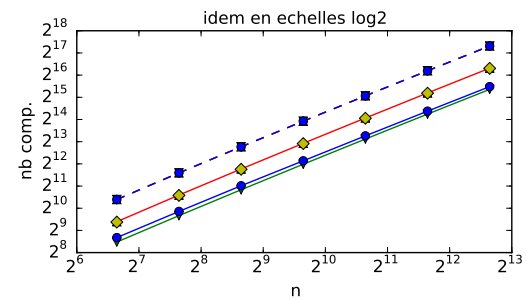
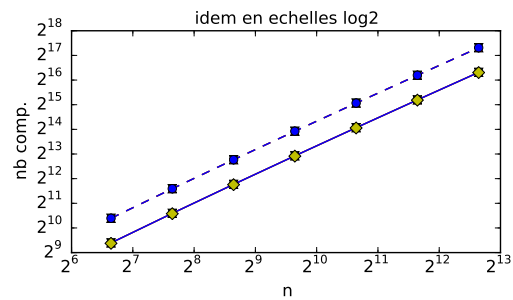
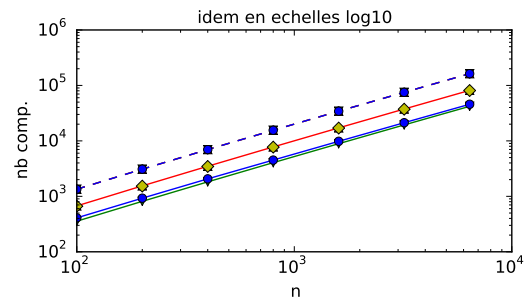
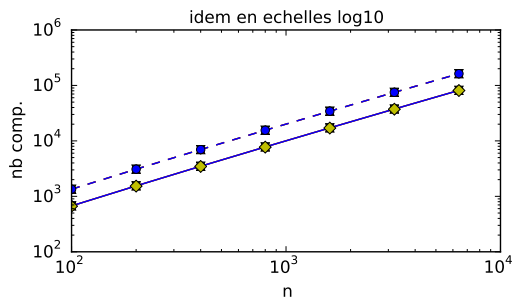
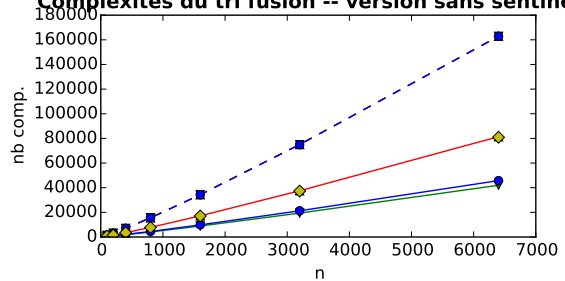
# sortie ecran
if ecran:
    plt.show()
else:
    # sortie pdf
    if sentinelles == True:
        #pp = PdfPages('./plt-test.pdf')
        pp = PdfPages('./plt-comp-trifusion-all.pdf')
    else:
        pp = PdfPages('./plt-comp-trifusion2-all.pdf')
    pp.savefig()
    pp.close()

```

Complexites du tri fusion -- version avec sentinelles



Complexites du tri fusion -- version sans sentinelle



```

# testtimetrifusion.py
"""Mesure et trace des timings mesures du tri fusion
. time vs. n
. ordre de complexite : t(2n)/t(n)"""
from random import randint
from numpy import log, log2
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
import time
import pickle as pk
from trifusion import trifusion
from testtimetriinsertion import pirecasinsertion, meilleurcasinsertion
valmax = 99

def tabalea(n):
    a = [randint(0, valmax) for i in range(n)]
    return a

def timer_tri(tri, dims):
    """time tri fusion (best, worst, aver) pour taille n in dims"""
    #
    timing_alea = []*len(dims)
    timing_best = []*len(dims)
    timing_worst = []*len(dims)
    for n in dims:
        # meilleur cas
        b = meilleurcasinsertion(n)
        t0 = time.time()
        c = tri(b)
        t = time.time() - t0
        timing_best.append(t)
        #print("au moins ", t, " pour n = ", n)
        # pire cas
        w = pirecasinsertion(n)
        t0 = time.time()
        c = tri(w)
        t = time.time() - t0
        timing_worst.append(t)
        #print("au plus ", t, " pour n = ", n)
        # une moyenne de p cas aleatoires
        p = 10
        tot = 0
        for essai in range(p):
            a = tabalea(n)
            t0 = time.time()
            c = tri(a)
            t = time.time() - t0
            #print(t, " pour n = ", n)
            tot += t
        tmoy = tot/p
        timing_alea.append(tmoy)
        #print(tmoy, " en moyenne pour n = ", n, p, " echantillons")
    return dims, timing_best, timing_worst, timing_alea

def save_time(n, t1, t2, t3, fichier):
    """save dans fichier les dims,

```

```

les timings de meilleurs , pires et moyens cas """
# on transforme en t2D
t = [n, t1, t2, t3]
f = open(fichier, "wb")
pk.dump(t, f)
f.close()

def read_time(fichier):
    """recuperation n et timings best, worst, avrg"""
    f = open(fichier, "rb")
    vals = pk.load(f)
    n = vals[0]
    t1 = vals[1]
    t2 = vals[2]
    t3 = vals[3]
    f.close()
    return n, t1, t2, t3

def plot_time(n, b, w, m):
    """n = les dims,
    b, w, m = timings best, worts, avg pour les dims"""
    # traces
    fig = plt.figure()
    fig.suptitle("Complexites du tri fusion sans sentinelle : timings",
                 fontweight='bold')

    # echelle lineaire
    #fig1 = fig.add_subplot(1,1,1)
    plt.subplot(211)#2lignes, 1 col, fig1
    plt.title("timing vs. n")
    plt.plot(n, b, 'g-v', label="meilleur")
    plt.plot(n, w, 'r-^', label="pire")
    plt.plot(n, m, 'b-o', label="moyen")
    plt.legend(loc=2) # legende en haut a gche

    # ratios
    plt.subplot(212)
    plt.title("timings : facteur d'augmentation entre t(2n) et t(n)")
    plt.semilogx()
    plt.ylabel("t(2n)/t(n)")
    # np.array commode pour div vecteurs
    bb = np.array(b)
    ww = np.array(w)
    mm = np.array(m)
    #print(bb,ww,mm)
    nbval = len(n)
    nt = np.array(n[1:nbval]) # n sans la premiere valeur pour ratios
    ratio_b = bb[1:nbval]/bb[0:nbval-1]
    ratio_w = ww[1:nbval]/ww[0:nbval-1]
    ratio_m = mm[1:nbval]/mm[0:nbval-1]
    #print(ratio_b, ratio_w, ratio_m)
    plt.plot(nt, ratio_b, 'g-v', label="meilleur")
    plt.plot(nt, ratio_w, 'r-^', label="pire")
    plt.plot(nt, ratio_m, 'b-o', label="moyen")
    plt.legend(loc='lower left')

    # sortie ecran ou pdf
    plt.show()
    #pp = PdfPages('./plt-times-trifusion2.pdf')

```

```

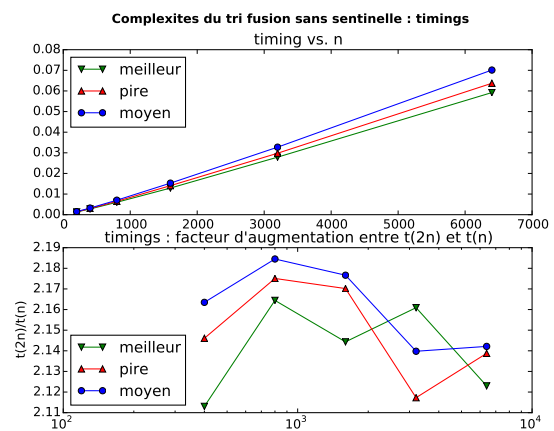
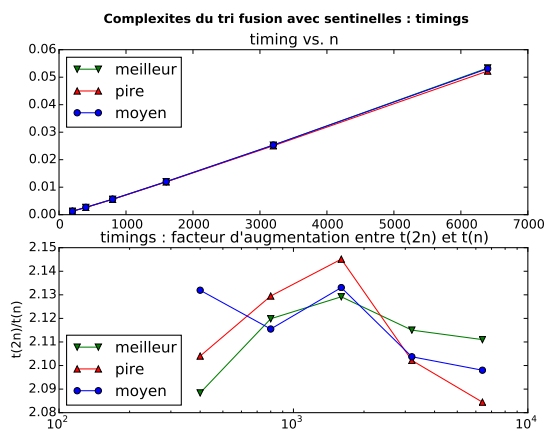
#pp.savefig()
#pp.close()

# tailles tableaux a trier
#dims = (10, 50)
dims = (200, 400, 800, 1600, 3200, 6400)
#dims = (400, 800, 1600, 3200, 6400, 12800, 25600)

# tris et save
n, b, w, m = timer_tri(trifusion, dims)
#save_time(n, b, w, m, "timingsfusion")

# load et trace
#n, b, w, m = read_time("timingsfusion")
plot_time(n, b, w, m)

```



Exercice 24. (P) Tri rapide

1. Que peut-on dire de la complexité du tri rapide lorsqu'on double le nombre de valeurs à trier ?
2. Écrire `trirapide` vu en cours.
3. Modifier `trirapide` pour compter le nombre d'appels récursifs.
4. (a) Modifier `trirapide` en `trirapide_comp` pour compter le nombre de comparaisons et d'affectations (des éléments du tableau d'entrée).
(b) La fonction `permutations` de la bibliothèque `itertools` retourne, sous la forme d'un itérateur, l'ensemble des permutations des composantes d'un vecteur d'entrées. Par exemple, la boucle :

```

for t in permutations([0, 1, 2]):
    print(t)

```

retourne les 6 t-uplets suivants :

```

(0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
(2, 0, 1)
(2, 1, 0)

```

Exploiter cette fonction pour tester exhaustivement `trirapide_comp` – be careful : size matters !

5. À la manière des questions 3 et 4 de l'exercice 22, écrire `testcomp`, `trirapide` et `testtimet`, `trirapide` pour illustrer les complexités (pire cas, meilleur cas, en moyenne) vues en cours. Proposer les tracés adaptés.

```

#trirapide.py
""" tri rapide """
from random import randint
from numpy import log2

def partition(t, g, d):
    """ invariant : avant la boucle i
    si g<=k <= m, t[k] <= pivot
    si m+1<=k <= i-1, t[k] > pivot
    si k = d-1, t[k] = pivot.
    Cad : t[g:m] <= pivot < t[m+1:d] (*)
    avec pivot = dernier elemnt
    et m+1 indice du premier > pivot """
    assert g < d
    pivot = t[d-1]
    m = g-1
    for i in range(g, d-1): #d-1 jamais atteint
        if t[i] <= pivot:
            tmp = t[i] #permuter t[m+1] et t[i]
            t[i] = t[m+1]
            t[m+1] = tmp
            m = m+1 # m reste le dernier <= pivot
    # placer pivot a sa place definitive
    t[d-1] = t[m+1]
    t[m+1] = pivot
    #print("partition :", t[g:d])
    return m+1

def trirapide0(t, g, d):
    if d - g > 1: #il reste au moins 2 elmnts ds cette sous-partie de t
        m = partition(t, g, d)
        trirapide0(t, g, m) # m est exclu
        trirapide0(t, m+1, d)

def trirapide(t):
    trirapide0(t, 0, len(t))

### versions recurso-iterative (pb limite python)
def trirapide11(t, g, d):
    while d - g > 1: #il reste au moins 2 elmnts ds cette sous-partie de t
        m = partition(t, g, d)
        trirapide11(t, g, m) # m est exclu
        g = m+1

def trirapide1(t):
    trirapide11(t, 0, len(t))

### version aleatoire
def trirapidealea(t, g, d):
    """ choix aleatoire du pivot """
    if d - g > 1: #il reste au moins 2 elmnts ds cette sous-partie de t
        i_pivot = randint(g,d)
        #permutation dernier et pivot aleatoire

```

```

    tmp = t[i_pivot]
    t[i_pivot] = t[d-1]
    t[d-1] = tmp
    # on y va
    m = partition(t, g, d)
    trirapide0(t, g, m) # m est exclu
    trirapide0(t, m+1, d)

def trirapidealea(t):
    trirapidealea(t, 0, len(t))

### pires cas pour comparaison et permutation
def pirepermtrirapide(dim):
    """pire cas des permutations"""
    assert dim % 2 == 0 # cas pair uniquement
    w = [None]*dim
    for i in range(0, dim//2):
        w[i] = dim - 1 - 2*i
        w[-i-1] = dim - 2 - 2*i
    return w

def pirecomptrirapide(dim):
    return [x for x in range(dim)]

```

```

#trirapide.py
"""Decomptes des appels, et des meilleurs cas du tri rapide (deterministe)
"""

from random import randint
from math import factorial
from numpy import log2
from itertools import permutations
from trirapide import pirepermtrirapide, pirecomptrirapide

#### nb appels
def trirapide0_appels(t, g, d, app):
    """compte le nbre d'appels recursifs"""
    app += 1
    if d - g > 1:
        m = partition(t, g, d)
        app = trirapide0_appels(t, g, m, app) # m est exclu
        app = trirapide0_appels(t, m+1, d, app)
    return app

def trirapide_appels(t):
    app = 0
    app = trirapide0_appels(t, 0, len(t), app)
    return app

#### nb comp et nb permutations
def partition_comp(t, g, d, cmp, perm):
    """invariant : avant la boucle i
    si g<=k <= m, t[k] <= pivot
    si m+1<=k <= i-1, t[k] > pivot
    si k = d-1, t[k] = pivot.
    Cad : t[g:m] <= pivot < t[m+1:d] (*)
    avec pivot = dernier elemnt
    et m+1 indice du premier > pivot"""
    assert g < d
    pivot = t[d-1]

```



```

m = g-1
for i in range(g, d-1): #d-1 jamais atteint
    cmp += 1
    if t[i] <= pivot:
        #print("permutation")
        m = m+1 # m reste le dernier <= pivot
        tmp = t[i] #permuter t[m+1] et t[i]
        t[i] = t[m]
        t[m] = tmp
        perm += 1
# placer pivot a sa place definitive
t[d-1] = t[m+1]
t[m+1] = pivot
perm +=1
#print("partition :", g, d, "donne : ", t)
return m+1, cmp, perm

def trirapide0_comp(t, g, d, cmp, perm):
    """compte le nbre de comparaisons et permutations"""
    if d - g > 1 :
        m, cmp, perm = partition_comp(t, g, d, cmp, perm)
        cmp, perm = trirapide0_comp(t, g, m, cmp, perm) # m est exclu
        cmp, perm = trirapide0_comp(t, m+1, d, cmp, perm)
    return cmp, perm

##def trirapide_comp(t):
##    cmp = perm = 0
##    cmp, perm = trirapide0_comp(t, 0, len(t), cmp, perm)
##    return cmp, perm

#
def partition_comp_opt(t, g, d, cmp, perm):
    """suppression des permutations inutiles"""
    assert g < d
    pivot = t[d-1]
    m = g-1
    for i in range(g, d-1): #d-1 jamais atteint
        cmp += 1
        if t[i] <= pivot:
            #print("permutation")
            m = m+1 # m reste le dernier <= pivot
            if i != m:
                tmp = t[i] #permuter t[m+1] et t[i]
                t[i] = t[m]
                t[m] = tmp
                perm += 1
# placer pivot a sa place definitive
if d-1 != m+1:
    t[d-1] = t[m+1]
    t[m+1] = pivot
    perm +=1
#print("partition :", g, d, "donne : ", t)
return m+1, cmp, perm

def trirapide0_comp_opt(t, g, d, cmp, perm):
    """compte le nbre d'appels recursifs"""
    if d - g > 1 :
        m, cmp, perm = partition_comp_opt(t, g, d, cmp, perm)

```

```

    cmp, perm = trirapide0_comp_opt(t, g, m, cmp, perm) # m est exclu
    cmp, perm = trirapide0_comp_opt(t, m+1, d, cmp, perm)
return cmp, perm

```

```

def trirapide_comp(t, opt):
    cmp = perm = 0
    if opt == True:
        cmp, perm = trirapide0_comp_opt(t, 0, len(t), cmp, perm)
    else:
        cmp, perm = trirapide0_comp(t, 0, len(t), cmp, perm)
    return cmp, perm

```

```

dim = 1
# cf compttrifusion_4.txt
n = dim
print("n, nlogn, n*n =", n, int(n*log2(n)), n*n)
print("nb permutations ", factorial(dim))
a = [x for x in range(dim)]
# tous les cas
opt = True
num_perm = 0
for tuple_t in permutations(a):
    num_perm += 1
    t = list(tuple_t)
    print(num_perm, ":", t, end = "")
    print("trie en nb comp, nb perm =", trirapide_comp(t, opt))

# pire et meilleurs : test exhaustifs
num_perm = 0
maxc = maxp = 0
minc = minp = dim*dim
minsum = minc+minp
for tuple_t in permutations(a):
    num_perm += 1
    t = list(tuple_t)
    c, p = trirapide_comp(t, opt)
    #
    maxc = max(c, maxc)
    maxp = max(p, maxp)
    if c == maxc or p == maxp:
        print("num_perm du max comp ou perm=", num_perm, "et", maxc, maxp)
    minc = min(c, minc)
    minp = min(p, minp)
    minsum = min(minsum, c+p)
    if minsum == c+p:
        print("num_perm du minsum=", num_perm, "et minc, minp", c, p, "et
            minsum=", minsum)

```

```

"""A FINIR : BUG QUI PLANTE TRI RAPIDE POUR N > 900
MEME LA VERSION AVEC UNE SEULE RECURSION
PAS DE MESSAGE .... """

```

```

# testtimetrirapide.py
"""Mesure et trace des timings mesures du tri rapide
. time vs. n
. ordre de complexite : t(2n)/t(n)
"""

```

```

from random import randint
from numpy import log, log2
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
import time
import pickle as pk
from trirapide import trirapide1, pirepermtrirapide, pirecomptrirapide
from triinsertion import meilleurcasinsertion, pirecasinsertion

valmax = 99

def tabalea(n):
    a = [randint(0, valmax) for i in range(n)]
    return a

def timer_tri(tri, dims):
    """time tri fusion (best, worst, aver) pour taille n in dims"""
    #
    timing_alea = []*len(dims)
    timing_best = []*len(dims)
    timing_worst = []*len(dims)
    for n in dims:
        # un pire cas
        b = pirepermtrirapide(n)
        t0 = time.time()
        c = tri(b)
        t = time.time() - t0
        timing_best.append(t)
        #print("au moins ", t, " pour n = ", n)
        # un autre pire cas
        w = pirecomptrirapide(n)
        t0 = time.time()
        c = tri(w)
        t = time.time() - t0
        timing_worst.append(t)
        #print("au plus ", t, " pour n = ", n)
        # une moyenne de p cas aleatoires
        p = 10
        tot = 0
        for essai in range(p):
            a = tabalea(n)
            t0 = time.time()
            c = tri(a)
            t = time.time() - t0
            #print(t, " pour n = ", n)
            tot += t
        tmoy = tot/p
        timing_alea.append(tmoy)
        #print(tmoy, " en moyenne pour n = ", n, p, " echantillons")
    return dims, timing_best, timing_worst, timing_alea

def save_time(n, t1, t2, t3, fichier):
    """save dans fichier les dims,

```

```

les timings de meilleurs , pires et moyens cas """
# on transforme en t2D
t = [n, t1, t2, t3]
f = open(fichier, "wb")
pk.dump(t, f)
f.close()

def read_time(fichier):
    """recuperation n et timings best, worst, avrg"""
    f = open(fichier, "rb")
    vals = pk.load(f)
    n = vals[0]
    t1 = vals[1]
    t2 = vals[2]
    t3 = vals[3]
    f.close()
    return n, t1, t2, t3

def plot_time(n, b, w, m):
    """n = les dims,
    b, w, m = timings best, worts, avg pour les dims"""
    # traces
    fig = plt.figure()
    fig.suptitle("Complexites du tri fusion sans sentinelle : timings",
                 fontweight='bold')

    # echelle lineaire
    #fig1 = fig.add_subplot(1,1,1)
    plt.subplot(211)#2lignes, 1 col, fig1
    plt.title("timing vs. n")
    plt.plot(n, b, 'g-v', label="pire permutations")
    plt.plot(n, w, 'r-^', label="pire comparaisons")
    plt.plot(n, m, 'b-o', label="moyen")
    plt.legend(loc=2) # legende en haut a gche

    # ratios
    plt.subplot(212)
    plt.title("timings : facteur d'augmentation entre t(2n) et t(n)")
    plt.semilogx()
    plt.ylabel("t(2n)/t(n)")
    # np.array commode pour div vecteurs
    bb = np.array(b)
    ww = np.array(w)
    mm = np.array(m)
    #print(bb,ww,mm)
    nbval = len(n)
    nt = np.array(n[1:nbval]) # n sans la premiere valeur pour ratios
    ratio_b = bb[1:nbval]/bb[0:nbval-1]
    ratio_w = ww[1:nbval]/ww[0:nbval-1]
    ratio_m = mm[1:nbval]/mm[0:nbval-1]
    #print(ratio_b, ratio_w, ratio_m)
    plt.plot(nt, ratio_b, 'g-v', label="pire permutations")
    plt.plot(nt, ratio_w, 'r-^', label="pire comparaisons")
    plt.plot(nt, ratio_m, 'b-o', label="moyen")
    plt.legend(loc='lower left')

    # sortie ecran ou pdf
    plt.show()
    #pp = PdfPages('./plt-times-trirapide.pdf')

```

```

#pp.savefig()
#pp.close()

# tailles tableaux a trier
#dims = (10, 50)
dims = (200, 900)#, 3200, 6400)
#dims = (400, 800, 1600, 3200, 6400, 12800, 25600)

# tris et save
n, b, w, m = timer_tri(trirapide1, dims)
#save_time(n, b, w, m, "timingsrapide")

# load et trace
#n, b, w, m = read_time("timingsrapide")
plot_time(n, b, w, m)

```

Exercice 25. Comparaison des algorithmes de tri

Reprendre les trois exercices précédents pour produire une courbe qui permet de comparer les timings moyens des algorithmes de tri vus en cours.

5.2 Autres algorithmes de tri et compléments

Exercice 26. Tri par insertion dichotomique

L'étape d'insertion du tri par insertion peut être réalisée par dichotomie.

1. Justifier pourquoi.
2. Quelle modification apporter à la preuve de la correction du tri par insertion ?
3. Quelle est la complexité en espace de cet algorithme ?
4. Quelle est la complexité en temps dans le pire cas de cet algorithme ? Commenter et expliciter un pire cas.
5. (P) Coder ce tri par insertion dichotomique. Observer le gain par rapport à la version classique.

Exercice 27. Autres versions du tri rapide

1. Autre choix déterministe.
 - (a) (P) Écrire une version du tri rapide qui choisisse comme pivot la première valeur du tableau à trier.
 - (b) Prouver la correction de cette version.
 - (c) Quels sont les meilleur et pire cas ?
2. Version randomisée.
 - (a) (P) Écrire une version du tri rapide qui choisisse comme pivot une valeur aléatoirement choisie dans le tableau à trier.
 - (b) Expliciter l'invariant de cette version.
 - (c) Qu'en est-il des meilleur et pire cas ?

Exercice 28. Tri par sélection

Le tri par sélection consiste à parcourir successivement tous les éléments du tableau pour en extraire la valeur minimale et la placer en tête de tableau (tri par ordre croissant). Ce principe est répété itérativement sur le reste du tableau jusqu'à obtenir le tableau complètement trié.

1. Proposer deux stratégies pour la gestion de la valeur minimale dont une de complexité en espace constante (et minimale) pour tout type d'éléments de tableau.
2. (P) Coder cet algorithme.
3. Démontrer la correction de cet algorithme.
4. Évaluer sa complexité en espace. Qu'en déduire ?
5. Évaluer sa complexité en temps en exhibant les meilleur et pire cas.

Exercice 29. (P) Application du tri par sélection

Le tri par insertion est souvent utilisé pour accélérer les autres algorithmes de tri lorsque le nombre d'éléments à trier est petit, c'est-à-dire inférieur à une certaine taille fixée à l'avance — à 5 par exemple.

1. Modifier le tri rapide avec cette accélération. Observer le gain.
2. Modifier le tri fusion avec cette accélération. Observer le gain.

Exercice 30. Tri à bulles

Le tri à bulles consiste à parcourir le tableau de la gauche vers la droite en permutant successivement chaque paire d'éléments consécutifs "mal" ordonnée. Ce parcours est répété tant que des permutations ont lieu.

1. Appliquer ce principe à la main sur le vecteur $[15, 4, 2, 8, 17, 23, 0, 1]$.
2. (P) Coder l'algorithme après avoir codé une fonction de permutation.
3. Quel est l'effet du premier parcours du tableau ? du second ? Expliquer sa dénomination.
4. Prouver la correction de l'algorithme.
5. Évaluer sa complexité en espace. Qu'en déduire ?
6. Évaluer sa complexité en temps en exhibant les meilleur et pire cas.

Exercice 31. Un tri linéaire : le tri par dénombrement

L'optimalité de la complexité semi-log des tris par comparaison peut être améliorée si on peut trier sans ...comparer.

Le tri par dénombrement s'applique quand on connaît toutes les valeurs du tableau à trier. Par exemple : $t[i] \in \{1, p\}$ pour $i \in \{1, n\}$. Il suffit alors de compter le nombre d'occurrences dans t de chacune des p valeurs possibles.

1. (P) Écrire et coder cet algorithme.
2. Quelle est sa complexité en temps dans le pire cas ? Exhiber un tel pire cas.
3. Qu'en est-il dans le meilleur cas ?
4. Quelle est sa complexité en espace ? Qu'en déduire ?

Exercice 32. Un autre tri linéaire : le tri-base ou tri radix ou encore *radix sort*

Compléter votre culture et affuter votre anglais en consultant la page wikipedia :
http://en.wikipedia.org/wiki/Radix_sort

Chapitre 6

Couleurs, images et traitements

6.1 Principes

Nous utiliserons les modules `pyplot` et `image` de `matplotlib`, les fichiers d'image et les tableaux `numpy`. Pour cela, il faut auparavant "avoir tout le matos".

Soit : `importer numpy, matplotlib.image` et `matplotlib.pyplot` qui seront respectivement appelés : `np`, `plt` et `mpimg`

Traitement basique. Les 3 grandes étapes du traitement avec `matplotlib` :

`file` → `np.ndarray` → `plt.img` → `file`

`mpimg.imread()` : `file` → `ndarray`

Lire un fichier `jpeg` ou `png` (par exemple) contenant une image. L'objet `ndarray t` créé est un tableau `numpy`. Le type de ces valeurs dépend du format du fichier image. Il est décrit par l'attribut `dtype`.

`float32` avec des valeurs comprises entre 0.0 et 1.0 si `file` est de type `png` ;

`uint8` avec des valeurs généralement comprises entre 0 et 255 pour les autres formats de fichiers image (`jpeg` en particulier). Attention : dans ce cas, l'arithmétique de ces valeurs est celle de $\mathbb{Z}/256\mathbb{Z}$.

C'est une étape de "pixelisation" : le tableau `t` est un ensemble de lignes ; chaque ligne est un ensemble de valeurs. Ces valeurs sont les pixels. Le tableau `t` permet tous les traitements numériques possibles ; voir l'exemple suivant et les exercices.

`plt.imshow()` : `ndarray` → `img`

Convertir un tableau `numpy` en un objet-image.

`plt.show()` permet alors de visualiser l'image en cours et, le cas échéant, de corriger son traitement.

`plt.savefig()` : `img` → `file`

Enregistrer l'objet-image dans un fichier `png`, `pdf`, `ps`, `eps` ou `svg` . :

Attention : cette commande est incompatible avec `plt.show()` (au moins dans mon environnement). Commenter cette dernière avant l'étape de sauvegarde.

Un exemple :

```
""" traitement image avec matplotlib.image et pyplot """
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

f = 'arago.jpg'
t = mpimg.imread(f) #t est un objet ndarray
# traitement global tres simple :) et en place
```

```

t = 255 - t
#
img = plt.imshow(t) #img est une image matplotlib

plt.axis('off') # pas d'axes x,y avec echelles
# choisir une des deux options suivantes (choix exclusif)
#plt.show()
plt.savefig('arago-inv.png', transparent=True, frameon= False , bbox_inches=
'tight', pad_inches=0.0) # ou .pdf si on veut

```



Remarque. On est parti d'un fichier jpeg sans obtenir un fichier transformé de même format. D'une part, le format png (compression sans perte) est tout à fait adapté aux affichages numériques et les formats pdf, ps (eps) adaptés aux affichages papier. Si du jpeg est vraiment nécessaire, plusieurs solutions sont possibles dont l'utilisation de la bibliothèque PIL à la place de matplotlib. C'est en fait cette dernière qui est appelée par matplotlib pour les formats autres que png. On peut donc espérer l'intégration totale de ces traitements dans matplotlib dans des prochaines versions. Mais tout ça, c'est de la bidouille : les concepts "fichier/objet image/tableau" sont invariants !

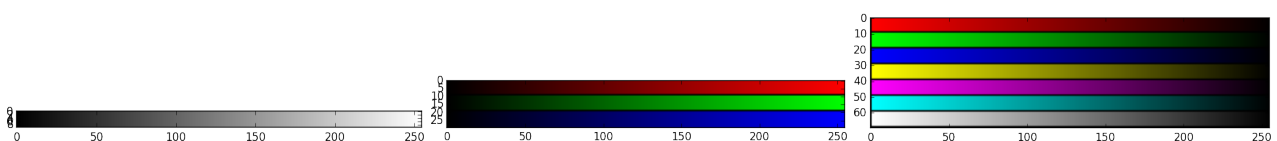
Tous les détails et les nombreuses autres commandes sont à l'URL : http://matplotlib.org/users/image_tutorial.html

6.2 On commence en couleurs !

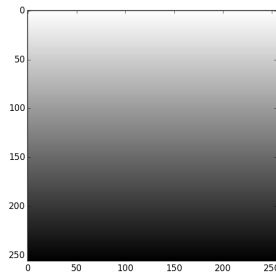
Exercice 33. (P) Des palettes en barres, en carrés et un premier traitement.

Plusieurs petits programmes pour être à l'aise avec les couleurs et l'attirail de traitement.

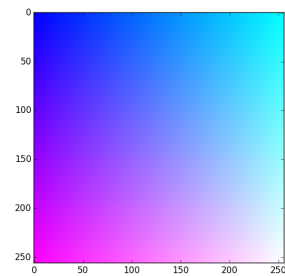
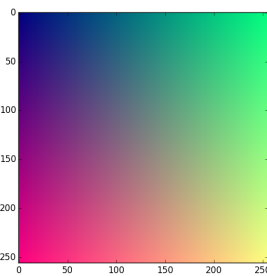
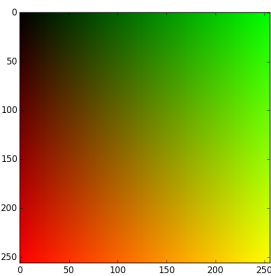
1. Écrire les programmes qui produisent les affichages suivants sous forme de barre 1D (ou presque) :
 - (a) le spectre des gris ;
 - (b) le spectre des trois couleurs R, V, B ;
 - (c) le spectre précédent complété des 4 mélanges de couleurs possibles.



2. Générer et tracer une palette carrée de gris.
3. La totalité des couleurs s'observe en 3D. Des palettes presque complètes sont possibles en 2D en paramétrant la valeur d'une des couleurs, le bleu par exemple.



- (a) Écrire une fonction `palettecouleurpart(b)` qui génère (sans tracé) une palette 2D de couleurs (R,V,B) pour une valeur de paramètre de B.
- (b) Écrire la fonction de tracé correspondante qui donne par exemple les figures suivantes pour $b = 0, 128, 255$.



6.3 Des images et du traitement

Le traitement avec des `ndarray` en `float32` issus du format `png` est moins délicat que pour les `uint8` obtenus à partir des autres formats, `jpg` en particulier.

Les images `arago.jpg` et `ballon.png` seront par exemple utilisées dans les traitements suivants.

Exercice 34. (P) Griser des photos couleur `png` et `jpeg`.

1. Les deux triplets (α, β, γ) suivants définissent des transformations classiques d'un pixel (R, V, B) en un niveau de gris $G = \alpha R + \beta V + \gamma B$. Pour les images naturelles, $(\alpha, \beta, \gamma) = (0.2125, 0.7154, 0.0721)$ et pour les images numériques : $(\alpha, \beta, \gamma) = (0.299, 0.587, 0.114)$.
2. Proposer et coder une solution simple pour transformer une image couleur `jpeg` (`uint8`) en niveaux de gris. Comparer l'image obtenue avec celles issues qu'un choix arbitraire de R, V ou B .
3. À partir de l'image en niveaux de gris, générer une image partagée horizontalement en 2 parties : l'une originale et l'autre en négatif. Généraliser ce traitement pour un nombre arbitraire de découpages.

Exercice 35. (P) Des traitements classiques

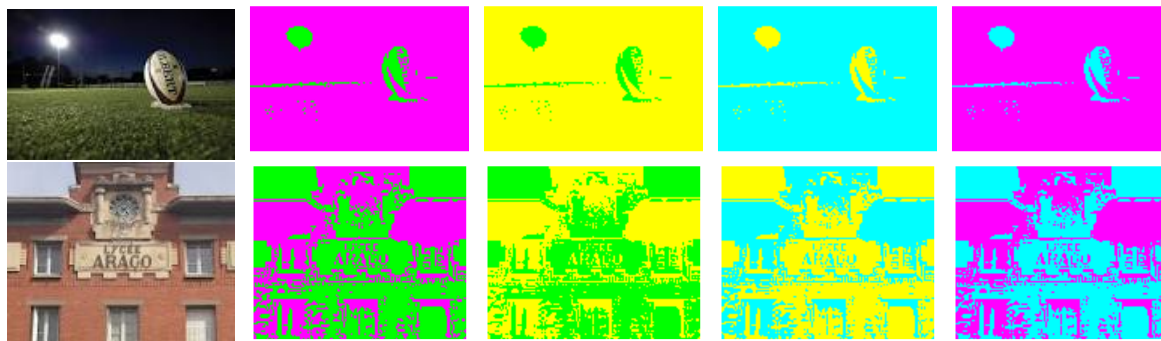
Réaliser les traitements suivants à partir d'une image en niveaux de gris.

- Contrastes** 1. La solution "brute force" consiste à générer une image en noir et blanc. Pour cela, il suffit de choisir un seuil, de "blanchir" les pixels plus clairs que ce seuil et de noircir les autres. On pourra essayer plusieurs valeurs de seuil et discuter de leur intérêt pour des images plutôt claires ou plutôt sombres.

2. Définir une fonction qui augmente le contraste de façon plus continue que la précédente. Essayer d'abord sur $[0, 1]$ puis sur $[0, 255]$. Coder et observer.
3. Proposer et coder une stratégie pour diminuer le contraste.

Luminance Ajouter ou retrancher une constante à la valeur de chaque pixel. Coder et observer.

Pop Art et wharolitude Proposer et coder une stratégie pour obtenir des images de ce genre :



Réduire la taille Proposer et coder une stratégie pour diminuer, par 2 par exemple, la taille d'une image.

- Fusionner deux images**
1. Préparatifs : dégager la table et poser un objet, votre stylo par exemple. Prendre une photo de la table avec cet objet de façon à ce que l'objet ne soit pas omniprésent par rapport à l'ensemble de la photo. **Sans trop bouger** l'appareil, déplacer significativement l'objet et prendre une autre photo.
 2. Fusion additive : après avoir transformé ces photos en niveaux de gris, les fusionner en une seule image par addition pixel avec pixel.
 3. Fusion soustractive : à quoi peut-elle servir ? Essayer en bougeant très très peu le stylo entre les 2 photos et toujours sans bouger l'appareil !

Troisième partie
Devoir maison

Chapitre 7

Novembre 2014

Exercice 1. Dans toute cette partie, on s'interdit l'usage de la fonction `min` préprogrammée en Python et permettant d'obtenir directement le minimum d'une liste donnée en argument. En revanche, on se donne la fonction `mini` suivante, écrite en Python.

```
1 def mini(t):
2     '''Calcule le minimum d'un tableau d'entiers ou de flottants.'''
3     if len(t) == 0:
4         return None
5     p=t[0]
6     for i in range(len(t)):
7         if t[i] <= p:
8             p = t[i]
9     return p
```

1. Expliquer le déroulement pas à pas (évolution de la valeur des variables) lors de l'appel `mini([6, 2, 15, 2, 15])`, puis donner la valeur renvoyée.
2. Prouver que lorsque `t` est une liste non vide d'entiers ou de flottants, `mini(t)` renvoie la valeur minimale des éléments de `t`. On exhibera un invariant de boucle précis.
3. Évaluer la complexité temporelle de l'appel `mini(t)` en fonction du nombre `n` d'éléments de `t`.
4. Proposer une modification de la fonction `mini` pour que la valeur renvoyée soit le maximum et non le minimum. On pourra utiliser la numérotation des lignes pour préciser le lieu d'éventuelles modifications et ainsi éviter de réécrire toute la fonction.

On souhaite récupérer non plus le minimum d'une liste mais la (une) position dans le tableau où le minimum est atteint. Dans l'exemple vu plus haut, il y a deux positions où ce minimum est atteint : 1 et 3.

5. Expliquer le principe d'une fonction réalisant cette opération et en particulier le rôle des variables manipulées.
6. Cette question ne sera lue que si la question précédente a été traitée. Écrire une fonction `position_mini` réalisant effectivement cette opération.
7. Préciser l'indice renvoyé si le minimum est présent plusieurs fois dans la liste. Proposer une modification permettant de changer ce comportement

On souhaite maintenant déterminer la valeur minimale d'un tableau bidimensionnel d'entiers/de flottants. Un appel de cette fonction pourrait être :

```
>>> mini2D([[10, 3, 15], [5, 13, 10]])
3
```

8. Expliquer le principe d'une fonction réalisant ce travail.
9. Programmer effectivement cette fonction (on supposera les listes internes de taille non nulle).
10. Évaluer la complexité temporelle de cette fonction.

On souhaite, partant d'une liste constituée de couples (chaîne, entier), déterminer la/une chaîne pour laquelle l'entier associé est minimal :

```
>>> chaine_mini(['Tokyo', 7000], ['Paris', 6000], ['Londres', 8000])
'Paris'
```

11. Écrire une fonction `chaine_mini` réalisant effectivement cette opération.
12. Écrire enfin une fonction `majores_par` prenant en entrée une liste `t` d'entiers ainsi qu'un entier seuil et renvoyant le nombre d'éléments de `t` majorés (au sens strict) par seuil :

```
>>> majores_par([12, -5, 10, 9], 10)
2
```

① Déroulement pas à pas de $\text{mini}([6, 2, 15, 2, 15])$.

3. $\text{len}(t) = 5 \neq 0$

5. $p = t[0] = 6$

6. $i = 0$

7. $t[0] \leq p$ vrai ($6 \leq 6$)

6. $i = 1$
7. $t[1] = 2 \leq p = 6$ vrai

6. $i = 2$

7. $t[2] = 15 \leq p = 6$ faux

8. $p = t[0] = 6$

8. $p = t[1] = 2$

6. $i = 3$

7. $t[3] = 2 \leq p = 2$ vrai

6. $i = 4$

7. $t[4] = 15 \leq p = 2$ faux

8. $p = t[3] = 2$

9. return 2 (= p).

mini renvoie la valeur $2 = p[3] = \text{min}(t)$.

② Invariant de boucle et preuve $\text{mini}(t) = \text{min}(t)$ si t est non vide.

• Au préalable : si t est vide, $\text{mini}(t)$ termine en renvoyant la valeur None : l4.

Si t est non vide, $\text{mini}(t)$ termine en renvoyant p , initialisé en l5 et mis à jour dans la boucle for (l8) qui termine pour $i = \text{len}(t)$.

• Démontrons l'invariant de la boucle for avant pour $i = 1$: l4.

(s) : Avant l'itération i (l6), $p = \text{min}\{t[0], t[1], \dots, t[i-1]\}$

Initialisation : Avant l'itération $i = 1$, $p = t[0]$ depuis la l5.

L'itération $i = 0$ ne modifie pas $p = t[0] = \text{min}\{t[0]\}$.

Rang (*) : On choisit $i = 1$ pour simplifier l'écriture de l'invariant et on montre que l'itération 0 ne modifie pas l'invariant. On aurait pu écrire :

$$i = 0.. \text{len}(t) \quad \text{et} \quad p = \text{min}\{t[0], t[1], \dots, t[\max(0, i-1)]\}$$

(ce qui est moins naturel pour une boucle qui calcule le min de t et qui nous conforte dans l'idée que cette boucle aurait dû être écrite $\text{range}(1, \text{len}(t))$)

- Conservation: Avant l'iteration i , $p = \min \{t[0], t[1], \dots, t[i-1]\}$, $i \geq 1$.

Pour $i < \text{len}(t)$, - si $t[i] \leq p$ (17), $t[i] \leq \min \{t[0], t[1], \dots, t[i-1]\}$
donc $t[i] = \min \{t[0], \dots, t[i-1], t[i]\}$. La lt modifie p qui
vaut maintenant $t[i] = \min \{t[0], \dots, t[i]\}$.

- Si $t[i] > p = \min \{t[0], t[1], \dots, t[i-1]\}$ alors $p = \min \{t[0], \dots, t[i-1], t[i]\}$.

Donc dans les 2 cas, avant l'iteration suivante $i+1$, $p = \min \{t[0], \dots, t[i]\}$.

- Termination: La premiere fois que le corps de la boucle n'est pas execute,

$i = \text{len}(t)$ et p vaut $\min \{t[0], \dots, t[\text{len}(t)-1]\}$.

t de longueur $\text{len}(t)$ et $t = \{t[0], \dots, t[\text{len}(t)-1]\}$.

Mini(t) renvoie $p = \min(t)$.

③ Complexite' temporelle de mini(t): Pour $n = \text{len}(t)$;

$T(n)$ compte les comparaisons et les affectations (hors celle des indices de boucle).

les lignes 13 et 15 introduisent 2 telles instructions.

la boucle 16-17-18 est executee exactement $\text{len}(t)$ fois.

Chaque iteration de boucle introduit exactement 1 comparaison et 0 ou 1 affecta-

-tion. L'iteration $i=0$ introduit systematiquement une affectation.

Soit $n+3 \leq T(n) \leq 2n+2$.

Asymptotiquement $T(n) = \Theta(n)$ et mini(t) est de complexite' lineaire.

④ mini(t) \rightarrow maxi(t) en remplaçant la lt par $t[i] > p$ ou $t[i] \geq p$.

⑤ Position du minimum: On ajoute une variable entiere pos, apres la
ligne 8 dans le corps de la boucle for, qui va stocker l'indice i du
minimum courant. pos peut être initialisee à 0 apres la ligne 15

(mais ce n'est pas necessaire ici car la premiere iteration executee toujours 18).

⑥ Voir codes

⑦ Position du minimum renvoyé.

A l'issue de la dernière itération ($i=n-1$), pos vaut l'indice de la dernière occurrence du minimum de t .

C.a.d : $t[pos] = \min \{t[0], \dots, t[n-1]\}$ et $pos \geq j$ pour j tel que $t[j] = t[pos]$, $j=0, n-1$.

Si on souhaite renvoyer la position de la première occurrence du minimum de t , il suffit de remplacer la ligne 17 par $t < p[i]$. (**).

⑧ Minimum d'un tableau bidimensionnel.

Le parcours d'un tableau bidimensionnel nécessite d'introduire deux boucles for imbriquées :

- la première parcourt les listes une à une :
- la seconde parcourt des éléments de chacune de ces listes.

Ce qui s'écrit de 2 façons en python pour un tableau bidim t .

<pre>for i in range(len(t)): for j in range(len(t[i])): ... t[i][j] ...</pre>	<pre>for l in t: for e in l: ... e ...</pre>
---	--

⑨ Voir codes.

⑩ Complexité temporelle de mini 2D :

Soit n le nombre de listes $t[0] t[1] \dots t[n-1]$,
soit $m = \max(\text{len}(t[0]), \text{len}(t[1]), \dots, \text{len}(t[n-1]))$.

$T(n, m)$ compte le nombre de comparaisons et d'affectations de mini 2D (hors celles des indices de boucles).

La bande extérieure est parcourue exactement n fois,

et à chaque fois la bande intérieure est parcourue au plus m fois.

Le corps de bande effectue exactement 1 comparaison et 0 ou 1 affectation à chaque itération. Donc la complexité de la bande intérieure est $\leq 2m$, et la complexité de la bande extérieure est $\leq n \times 2m$.

Les listes intérieures ont toujours non vides donc $m \geq 1$ et $T(n, m) \geq n$.

En ajoutant les comparaisons (" l_3 ") et affectations (" l_5 "), on a donc :

$$\underline{n+2 \leq T(n, m) \leq 2mn+2}$$

Le meilleur cas est obtenu pour un minimum ^{unique} $t[0][0]$ et le pire cas pour n listes triées par ordre décroissant et globalement intérieures 2 à 2.

Si le tableau bidim. contient $m \times n$ valeurs (matrice rectangulaire $m \times n$)

alors $T(n, m) = \Theta(m \times n)$. Mini 2D est asymptotiquement linéaire en le nombre d'éléments du tableau.

⑪ la liste est composée de couples (chaîne de caractères, entier).

Chaîne_minim renvoie la chaîne correspondant à la valeur entière minimale.

Voir codes.

⑫ On introduit un compteur, c.à.d. une variable entière, et on parcourt le contenu de chaque liste en comparant avec la valeur de seuil. On incrémente le compteur si besoin et on retourne la valeur de ce compteur en fin de traitement.

** (7) : Autre solution plus compliquée mais qui permet de inverser le sens de range (start, stop, step).

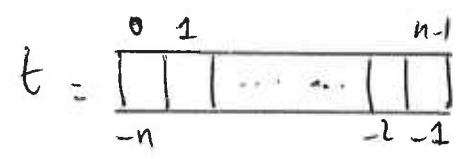
On veut la première occurrence du mini de t. On peut parcourir la liste t de la fin vers le début (sans modifier la comparaison t).

l6 devient alors : for i in range (len(t)-1, 0, -1) :

et l'initialisation l5 devient : p = t[-1] (= t[len(t)-1])

Rappel : range ([start], stop, [step]) avec $\begin{cases} \text{start} + i * \text{step} < \text{stop} \\ n * \text{step} > 0 \end{cases}$ ou $\begin{cases} \text{start} + i * \text{step} > \text{stop} \\ n * \text{step} < 0 \end{cases}$
Options = 0 sinon = 1 sinon.

Autre solution "propre" à python



(l6) devient for i in range (-1, -len(t), -1) :

et l'init (l5) devient (encore) p = t[-1].

```

'''Correction sujet zero CCM : donnees meteo'''
#
def mini(t):
    if len(t) == 0:
        return None
    p = t[0]
    for i in range(len(t)):
        if t[i] <= p:
            p = t[i]
    return p

#Q6
def pos_mini(t):
    if len(t) == 0:
        return None
    p = t[0]
    pos = 0
    for i in range(len(t)):
        if t[i] <= p:
            p = t[i]
            pos = i
    return pos

#Q7
def premiere_pos_mini(t):
    """parcours en partant de la fin de t"""
    if len(t) == 0:
        return None
    p = t[len(t)-1]
    pos = len(t)-1
    for i in range(len(t)-1,0,-1):
        if t[i] <= p:
            p = t[i]
            pos = i
    return pos

#Q7 autre solution
def premiere_pos_mini2(t):
    """changement du test : <= devient <"""
    if len(t) == 0:
        return None
    p = t[0]
    pos = 0
    for i in range(len(t)):
        if t[i] < p:
            p = t[i]
            pos = i
    return pos

t=[6,2,15,2,15]
print("Q6-7 : ", t)
print("mini = ", mini(t))
print("pos = ", pos_mini(t))
print("premiere pos = ", premiere_pos_mini(t))
print("premiere pos (solution 2) = ", premiere_pos_mini2(t))

```

```

#Q9
def mini2D(t):
    if len(t) == 0:
        return None
    p = t[0][0]
    for num_liste in range(len(t)):          #nombre de listes de t
        # parcours liste apres liste
        for j in range(len(t[num_liste])):  #nombre d'elements de t[.]
            # parcours de la liste i
            if t[num_liste][j] <= p:
                p = t[num_liste][j]
    return p

#Q9 : autre ecriture
def mini2D2(t):
    """ avec des boucles python : on itere sur les elements (sans indice)
        """
    if len(t) == 0:
        return None
    p = t[0][0]          # t est une liste de listes , p le min de toutes ces
                        val
    for e in t:          # e est une liste : parcours de t
        for f in e:     # f est une valeur de e : parcours de e
            if f <= p:
                p = f
    return p

#Q9 : autre ecriture moins propre et plus cher (2 parcours de t)
def mini2D_concat(t):
    """concatene t en une liste 1D puis appel a mini"""
    if len(t) == 0:
        return None
    t1D = []
    for e in t:          # e est une liste de t
        t1D = t1D + e   # t1D concatene t2D
    print("t1D : ", t1D)
    return mini(t1D)

tt= [[10,3,15],[5,13,10]]
print("Q9 : ", tt)
print("len t2d = ", len(tt)) ; print("len t2d[0] = ", len(tt[0]))
print("mini2D = ", mini2D(tt)); print("mini2D v2= ", mini2D2(tt))
print("mini2D_concat = ", mini2D_concat(tt))

# Q11
def chaine_mini(t):
    """ version ou on manipule entier et chaine separement"""
    if len(t) == 0:
        return None
    minival = t[0][1]          # est un entier
    pos = 0
    for i in range(len(t)):   # pour chaque couple (chaine, entier)
        if t[i][1] <= minival: # l'entier
            minival = t[i][1]
            pos = i           # sa position
    return t[pos][0]         # la chaine

```

```

def chaine_mini2(t):
    """ version ou on manipule un couple (chaine-entier)
    et avec des boucles python (sans indice) """
    if len(t) == 0:
        return None
    min = t[0] # est un couple (chaine-entier)
    for couple in t:
        # parcours liste apres liste
        if couple[1] <= min[1]: # cas ou min = couple (chaine, entier)
            min = couple
    return min[0] #la chaine

ch = [['Tokyo', 7000], ['Paris', 6000], ['Londres', 8000]]
print("Q11 : ",ch)
print("chaine_mini : ", chaine_mini(ch))
print("chaine_mini : ", chaine_mini2(ch))

# Q12
def majores_par2(t, seuil):
    nb_occ = 0
    for e in t:
        if e < seuil:
            nb_occ += 1
    return nb_occ

ch2 = [12,-5,10,9] ; s = 10
print(majores_par2(ch2, s),"valeurs superieures strictes a ", s, "dans ",
ch2)

```