Piles

I. Les piles

I.1. Notion de structure de données

Une des problématiques importantes de l'informatique est le stockage des données. Pour traiter efficacement ces dernières, il faut les ranger de manière adéquate, en fonction du problème posé. L'objet informatique qui stocke des valeurs en mémoire s'appelle une *structure de données* ; elle est caractérisée par les opérations qu'elle permet et le coût de ces opérations.
La principale structure de données en Python est la structure de liste, mais il y a aussi les *tuples*, les *ensembles*, les *dictionnaires* ; `numpy`, quant à lui, permet de manipuler des *tableaux*.

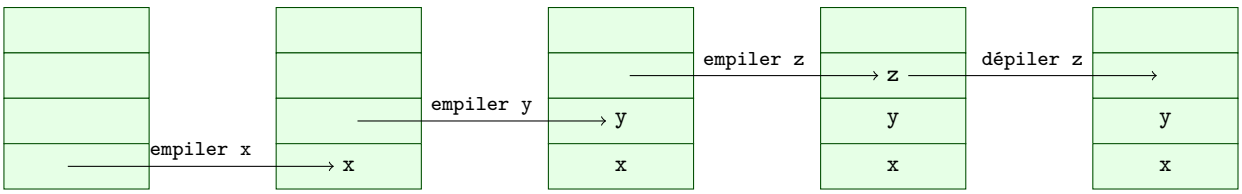
Nous allons maintenant aborder la structure de *pile*, qui n'est pas implémentée nativement en Python, mais que l'on peut simuler à l'aide des listes.

I.2. La structure de pile

Déf 1:

On dit qu'une liste de données est organisée en pile (« stack » en Anglais) si seule la dernière donnée ajoutée à la liste est directement accessible.

Ainsi, dans une pile, tout ajout se fait devant le dernier élément de la liste (appelé sommet de la pile), et toute suppression ne peut concerner que le sommet. Une telle structure est dite de type LIFO (Last In, First Out).
L'ajout d'un élément à une pile s'appelle empilage ; la suppression du sommet s'appelle le dépilage .



Pile vide

I.3. Les primitives sur les piles

L'implémentation de la structure de pile doit comprendre les fonctions suivantes (et uniquement celles-ci, on s'astreindra à n'utiliser aucune autre fonction Python sur les listes) :

- une fonction de création d'une pile vide ;
- une fonction d'empilage (**`empiler`** ou **`push`**) et une fonction de dépilage (**`depiler`** ou **`pop`**) ;
- une fonction permettant de vérifier si une pile est vide ;
- éventuellement, une fonction (**`sommet`** ou **`top`**) permettant de retrouver le sommet de la pile mais sans l'enlever.

En utilisant directement les fonctions sur les listes disponibles en Python, cela donne les procédures ci-dessous.

```
def creer_pile():
    return []
def est_vide(p):
    return len(p) == 0
def empiler(p, x):
    p.append(x)
def depiler(p):
    assert len(p) >0, 'Pile vide' # ne devrait pas se produire
    return p.pop()
def sommet(p):
    assert len(p) >0, 'Pile vide' # ne devrait pas se produire
    return p[-1]
```

II. Exemples d'utilisation d'une pile

II.1. Contrôle du parenthésage d'une expression

Il s'agit ici d'écrire une fonction qui contrôle si une expression mathématique, donnée sous forme d'une chaîne de caractères, est bien parenthésée, c'est-à-dire s'il y a autant de parenthèses ouvrantes que de fermantes, et qu'elles sont bien placées.

Version simplifiée :

On parcourt l'expression de gauche à droite. Chaque fois qu'une parenthèse « (» est trouvée, on l'empile ; quand on trouve une parenthèse «) », on dépile si c'est possible (sinon, il y a une erreur. Si l'expression est correcte, la procédure doit se terminer sur une pile vide.

Algorithme 1 : Vérification parenthèses, version simplifiée

Données : *expr* : expression sous forme d'une chaîne de caractères
Résultat : Vrai ou Faux

def *VerifieParenthese(expr)* **:**
  $p \leftarrow$ Pile Vide;
  pour *car parcourant expr* **faire**
    si *car* $=$ '(' **alors**
      empiler *car* dans p
    finsi
    si *car* $=$ ')' **alors**
      si p *est vide* **alors**
        retourner *Faux*
      sinon
        depiler p
      finsi
    finsi
  finpour
  si p *est vide* **alors**
    retourner *Vrai*
  sinon
    retourner *Faux*
  finsi
fin

Cela donne le programme Python ci-dessous.

Vérification du parenthésage d'une expression, version simplifiée

```
from DefPiles import *
# le fichier DefPiles.py contient les primitives sur les piles
def test_parenthesage1(expr):
    p = creer_pile()
    for elt in expr:
        if elt == '(' :
            empiler(p, '(')
        if elt == ')' :
            if est_vide(p):
                return False
            else:
                depiler(p)
    return est_vide(p)

liste_essais=['2+(3*4)', '2+(3*4)*2)', '(x+(y*z)*(2)']
for expr in liste_essais:
    print (expr, ' ----> ',test_parenthesage1(expr))
```

```
2+(3*4)  ----> True
2+(3*4)*2)  ----> False
(x+(y*z)*(2)  ----> False
```

Exercices :

1. Améliorer le programme précédent pour afficher le type d'erreur (parenthèse ouvrante ou fermante en trop), et l'indice où elle s'est produit.
2. Améliorer le programme précédent pour gérer aussi les accolades « { } » et les crochets « [ ] ».

II.2. Évaluation d'une expression mathématique écrite sous forme postfixée

L'écriture postfixée (ou notation polonaise inverse) d'une expression algébrique consiste à placer les opérateurs après son ou ses opérande(s).

Par exemple : « $a+b$ » sera écrit « $a\ b\ +$ » ; « $\sqrt{a}$ » sera écrit « $a\ \sqrt{}$ » ; « $\sqrt{a^2+b^2}$ » s'écrira « $a\ 2\ \hat{}\ b\ 2\ \hat{}\ +\ \sqrt{}$ » ; « $a-((b+c*d/e-f)*g)/(h^2)$ » s'écrira « $a\ b\ c\ d\ *\ e\ /\ +\ f\ -\ g\ *\ h\ 2\ \hat{}\ /\ -$ ».

Dans une expression postfixée, l'usage des parenthèses est superflu.

Pour évaluer une expression postfixée, dont les éléments sont supposés rangés dans une liste (les opérateurs étant sous forme d'une chaine de caractères), on procèdera de la façon suivante : on utilise une pile initialement vide puis on parcourt les éléments de l'expression à évaluer en appliquant les règles suivantes :

- si l'élément est un nombre ou une variable (donc si ce n'est pas un opérateur), il est empilé ;
- si l'élément est un opérateur unaire, le sommet de la pile est dépilé, cet opérateur lui est appliqué et le résultat est ré-empilé ;
- si l'élément est un opérateur binaire , deux éléments de la pile sont dépilés, l'opérateur leur est appliqué et le résultat est ré-empilé.

Si l'expression est syntaxiquement correcte, à la fin du traitement la pile ne contient qu'un élément : c'est le résultat du calcul.

Exercice :

On suppose donnée une implémentation de la structure `Pile`, ainsi que les constantes et procédures suivantes :

```
import math
# liste des opérateurs autorisés:
op_unaire = ['sqrt', 'exp', 'ln', 'sin', 'cos']
op_binaire = ['+', '-', '*', '/', '^']

def calc1(operateur, op):
    # résultat d'une opération unaire
    if operateur == 'sqrt':
        return math.sqrt(op)
    elif operateur == 'exp':
        return math.exp(op)
    elif operateur == 'ln':
        return math.log(op)
    elif operateur == 'cos':
        return math.cos(op)
    elif operateur == 'sin':
        return math.sin(op)
    else:
        raise NameError('Opérateur inconnu')

def calc2(operateur, op1, op2):
    # résulat d'une opération binaire
    if operateur == '+':
        return op1 + op2
    elif operateur == '-':
        return op1 - op2
    elif operateur == '*':
        return op1 * op2
    elif operateur == '/':
        return op1 / op2
    elif operateur == '^':
        return op1 ** op2
    else:
        raise NameError('Opérateur inconnu')
```

Écrire une procédure qui permet d'évaluer une expression postfixée (supposée syntaxiquement correcte).

II.3. Transformation d'une expression en expression postfixée

Il s'agit maintenant de transformer une expression écrite sous forme algébrique classique en une expression écrite sous forme postfixée, afin de pouvoir ensuite l'évaluer en utilisant le programme précédent.
Pour simplifier (du moins dans un premier temps, vous pourrez raffiner ensuite), nous supposerons que les expressions manipulées comprennent seulement :

- des opérandes formés d'une seule lettre : a, b, $\ldots$, x, y, $\ldots$;
- des opérateurs binaires d'un seul caractère : $+$, $-$, $*$, $/$, $\hat{}$;
- des parenthèses (et).

L'algorithme est le suivant.

Algorithme 2 : Transformation en postfixée

Données : *expr* : expression algébrique sous forme d'une chaîne de caractères
Résultat : expression sous forme postfixée dans la variable *result*

```
def Transforme(expr) :
    p ← Pile Vide;
    result ← liste vide;
    car ← premier caractère de expr;
    tant que car non vide ou p non vide faire
        si car est un opérande alors
            ajouter car à result;
            car ← caractère suivant
        finsi
        sinon si car == '(' alors
            empiler car;
            car ← caractère suivant
        finsi
        sinon si car est un opérateur et (p est vide ou sommet(p) == '(' ) alors
            empiler car;
            car ← caractère suivant
        finsi
        sinon si car et sommet(p) sont des opérateurs alors
            si car plus prioritaire (strictement) que sommet(p) alors
                empiler car;
                car ← caractère suivant
            finsi
            sinon
                dépiler;
                ajouter le sommet(p) ainsi dépilé à result;
                /* car ne change pas !                                  */
            finsi
        finsi
        sinon si (car est vide ou égal à ')' ) et sommet(p) est un opérateur alors
            dépiler;
            ajouter le sommet(p) ainsi dépilé à result;
            /* car ne change pas !                                      */
        finsi
        sinon si car = ')' et sommet(p)='(' alors
            dépiler;
            car ← caractère suivant
        finsi
        sinon
            Erreur !;
        finsi
    fintq
fin
```

Par exemple, si on applique cet algorithme à l'expression :

$$a + (b - c) * d/e$$

on obtient les contenus suivants des variables :

car	p avant	p après	result
a	vide	vide	[a]
+	vide	+	[a]
(	+	+(	[a]
b	+(	+(	[a,b]
−	+(	+(−	[a,b]
c	+(−	+(−	[a,b,c]
)	+(−	+(	[a,b,c,−]
...	...	...	...
...	...	...	...

II.4. Tour de magie de Gilbreath

1. Écrire une fonction `couper` qui prend une pile et la coupe en enlevant de son sommet un certain nombre d'éléments (tiré au hasard) qui sont renvoyés dans une seconde pile en ordre inverse. Exemple : si la pile initiale est `[1, 2, 3, 4, 5]` et si le nombre d'éléments retirés vaut 2, alors la pile ne contient plus que `[1, 2, 3]` et la pile renvoyée contient `[5, 4]`.
2. Écrire une fonction `melange` qui prend en arguments deux piles et qui mélange leurs éléments dans une troisième pile de la façon suivante : tant qu'une pile au moins n'est pas vide, on retire aléatoirement un élément au sommet d'une des deux piles et on l'empile sur la pile résultat. Exemple : un mélange possible des piles `[1, 2, 3]` et `[5, 4]` est `[3, 2, 4, 1, 5]`. Note : à l'issue du mélange, les deux piles de départ sont donc vides.
3. Construire un paquet de cartes en empilant n fois les mêmes k cartes (par exemple, pour un paquet de 32 cartes, on empile $n = 16$ paquets de paires rouge/noir). Couper alors le paquet avec la fonction `couper` ci-dessus, puis mélanger les deux paquets obtenus à l'aide de la fonction `melange`. On observe alors que le paquet final contient toujours n blocs des mêmes k cartes (même si ces dernières peuvent apparaître dans un ordre différent au sein de chaque bloc). Sur l'exemple des 16 paquets rouge/noir, on obtient toujours 16 paquets rouge/noir ou noir/rouge.

```
from DefPiles import *
import random

def couper(pile):
    n = random.randint(1, len(pile)-1)

   ....


def melange(p1, p2):


   ....


paquet = 8*['Coeur', 'Carreau', 'Trèfle', 'Pique']
paquet2 = couper(paquet)
print(melange(paquet, paquet2), '\n')

paquet = 4*[1,2,3,4,5,6,7,8]
paquet2 = couper(paquet)
print(melange(paquet, paquet2))

>>>
['Pique', 'Coeur', 'Trèfle', 'Carreau', 'Carreau', 'Coeur', 'Trèfle', 'Pique',
 'Coeur', 'Carreau', 'Trèfle', 'Pique', 'Coeur', 'Carreau', 'Trèfle', 'Pique',
 'Coeur', 'Carreau', 'Trèfle', 'Pique', 'Coeur', 'Carreau', 'Trèfle', 'Pique',
 'Coeur', 'Carreau', 'Trèfle', 'Pique', 'Coeur', 'Carreau', 'Trèfle', 'Pique']

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

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

II.5. Un tri

On considère une liste pile `p` contenant des nombres que l'on veut trier.
Pour cela, on utilise deux piles `p1` et `p2` vérifiant toujours les propriétés suivantes :

- les éléments de `p1` sont empilés en ordre décroissant ;
- les éléments de `p2` sont empilés en ordre croissant ;
- le sommet de `p1` est strictement supérieur au sommet de `p2`.

Le but est de répartir les éléments de `p` dans les piles `p1` et `p2` en conservant ces restrictions. Lorsque `p` sera vide, il suffira de réempiler les éléments de `p2` sur ceux de `p1` pour avoir une pile triée par ordre décroissant (ou `p1` sur `p2` pour un ordre croissant).
On procédera donc ainsi :

- Au départ, on met dans `p1` (respectivement `p2`) le plus grand (respectivement le plus petit) des 2 sommets de `p`.
- Puis, tant que la pile `p` n'est pas vide, on compare le sommet `s` de `p` aux sommets `s1` et `s2` de `p1` et de `p2` respectivement :
  - si `s=s1`, on empile `s` sur `p1` ;
  - si `s=s2`, on empile `s` sur `p2` ;
  - si `s1>s>s2`, on empile `s` sur `p1` (par exemple), ou, mieux, sur celle des deux piles qui est la plus petite ;
  - si `s1<s` : on dépile `p1`, et on empile ses éléments sur `p2` jusqu'à ce que le sommet de `p1` devienne $\geqslant$ `s` (si jamais `p1` devient vide, on y met seulement `s`).
  - on procède de façon analogue si `s<s2`.
- On termine comme indiqué ci-dessus, en « fusionnant » les piles `p1` et `p2`.

II.6. Coloriage

II.6.1. Manipulation des images en Python

1. Les images informatiques

Une image matricielle (ou *carte de points*, *bitmap*) est une matrice de points colorés appelés pixels (= *picture element*).
Chaque case de la matrice contient la couleur du pixel correspondant. On convient de numéroter les colonnes en partant de la gauche, et les lignes en partant du haut.
Cette couleur est représenté par un triplet de nombres entre 0 et 255 (3 octets) : un nombre pour chaque couleur primaire rouge, vert et bleu ; c'est le format RGB (*red, green, blue*). La couleur définitive de chaque pixel est obtenue par addition de ces couleurs primaires.
Exemple :

(0,0,0)	noir	(255,255,255)	blanc
(255,0,0)	rouge	(0,255,0)	vert
(0,0,255)	bleu	(0,255,255)	cyan
(255,0,255)	magenta	(255,255,0)	jaune
(148,129,43)	kaki	(0,86,27)	vert impérial
(248,142,85)	saumon	...	...

Cela fait $256^3 = 16\,777\,216$ couleurs possibles.
Certaines images sont encodées en RGBA ; dans ce cas, un 4^e octet est utilisé (*canal alpha*) pour gérer la transparence, de 0 pour un pixel totalement transparent à 255 pour un pixel complètement opaque.

2. Les outils Python

Pour lire, écrire et modifier des images, il existe de nombreux modules en Python (qu'il ne faut pas utiliser en même temps).
Nous utiliserons ici la bibliothèque `pillow`, qui est la version maintenue à jour de la bibliothèque PIL (Python Imaging Library). Vous pouvez éventuellement trouver les fichiers d'installation de `pillow` ici : `https://pypi.python.org/pypi/Pillow/`, et une documentation (en Anglais) ici :
`http://pillow.readthedocs.org/en/3.0.x/handbook/index.html` ou là :
`https://media.readthedocs.org/pdf/pillow/latest/pillow.pdf` .

Pour simplifier, les images utilisées seront placées dans le même répertoire que votre programme (cela évite d'écrire tout le chemin d'accès).
Les instructions de base sont illustrées dans le petit programme (sans utilité) ci-dessous.

```python
from PIL import Image

# lecture du fichier image
image = Image.open('Joconde.jpg')
print(image.size, image.mode, image.format)
# image.size = largeur (=abcisses), hauteur(=ordonnées)
# image.mode = type d'image: 'L' si en gris, 'RGB' si en 3 couleurs etc...
# image.format = extension du fichier: Jpeg, Tif, ...
tailleX, tailleY = image.size

# on peut extraire des pixels depuis l'image avec getpixel  et réécrire dans l'image avec putp
for x in range(tailleX):
    for y in range(tailleY):
        pixel = image.getpixel( (x,y) )
        if sum(pixel)>500:
            image.putpixel( (x,y), (255,255,255) )
        if sum(pixel)< 80:
             image.putpixel( (x,y), (255,0,0) )

# affichage
image.show()
# sauvegarde sous un nouveau format
image.save("Joconde-defiguree.png")
```

```
(620, 701) RGB JPEG
```

Voici le résultat de ce programme stupide :

Figure 1 – Joconde

Figure 2 – Joconde défigurée

II.6.2. Coloriage

On veut colorier une partie d'une image noir et blanc délimitée par un contour.

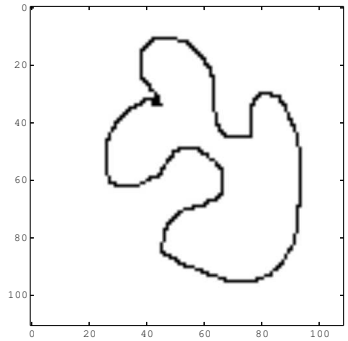


Figure 3 – Figure initiale

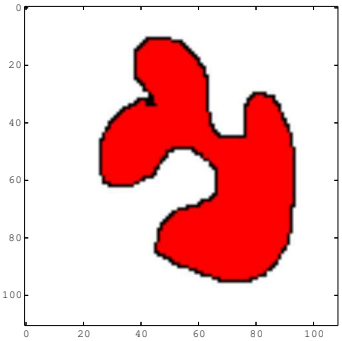


Figure 4 – Figure finale

Pour cela, on part d'un point de la région choisie, puis on parcourt les pixels de la région en modifiant la couleur au fur et à mesure de leur rencontre tant que l'on a des pixels de la même couleur.
On utilisera une pile de la façon suivante : le pixel à colorier est celui qui est au sommet de la pile ; s'il est de la même couleur que le pixel de départ, on le colorie puis on empile ses voisins (sous réserve de ne pas déborder de l'image). Le procédé s'arrêtera lorsque la pile sera vide.

On peut aussi colorier plusieurs parties d'une même image. Essayez d'obtenir la figure ci-dessous :

Figure 5 – Figure initiale

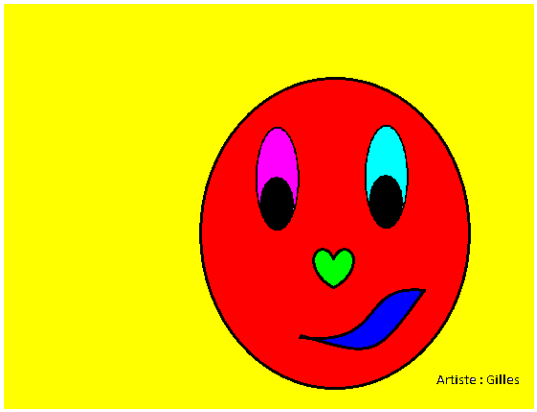
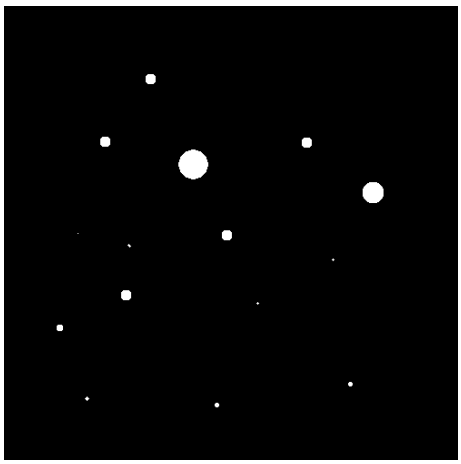


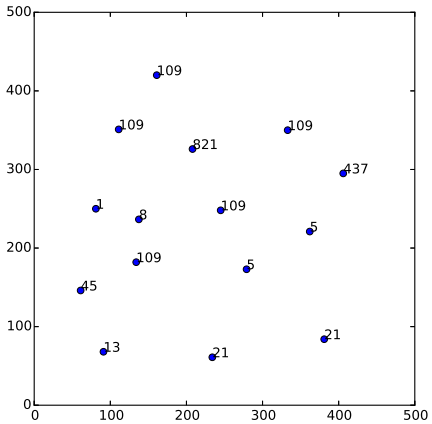
Figure 6 – Figure finale

II.6.3. Extraction de taches

On considère une image binaire (noir et blanc), qui comporte des taches plus ou moins grosses ; il s'agit d'énumérer ces taches et de déterminer l'isobarycentre de chacune d'entre elles ainsi que son nombre de pixels. Par exemple, pour l'image :



le programme donnera le résultat :



Pour cela, on parcourt l'image par lignes successives. Dès que l'on rencontre un pixel blanc, on entre dans une boucle qui effectue le coloriage de la région liée à ce pixel (par un programme semblable au précédent) et calcule l'isobarycentre et le nombre de pixels.

II.7. Génération d'un labyrinthe parfait

Un labyrinthe est une grille rectangulaire $n \times m$, dont les éléments sont appelés des cellules, qui sont séparées par des murs. Un labyrinthe est dit parfait si deux cellules quelconques sont reliées par un chemin et un seul.

Pour construire un labyrinthe parfait, il suffit d'énumérer les mn cases dans l'ordre du chemin qui les relie, en partant d'une case arbitraire. On utilisera ici une méthode exhaustive. Chaque cellule contient un booléen qui vaut `True` si elle appartient au chemin et `False` sinon ; au départ, toutes les cellules sont donc à `False`.
Voici la démarche de construction, qui utilise une pile contenant les emplacements à partir desquels on est susceptible de se déplacer :

- la pile est initialisée à `vide` et le dessin du labyrinthe est initialisé à une grille $n \times m$ de cases blanches toutes séparées par des murs noirs ;
- on part d'une cellule au hasard, par exemple celle de coordonnées $(0, 0)$, dont on met la valeur à `True` et que l'on empile ;
- tant que la pile n'est pas vide, on examine s'il y a des cellules disponibles dans le voisinage du sommet.
  - si non, on dépile ;
  - si oui, on choisit au hasard une des cellules disponibles, que l'on empile, et sur le dessin on casse le mur entre la case précédente et celle-ci.

Ci-dessous un exemple de labyrinthe obtenu :

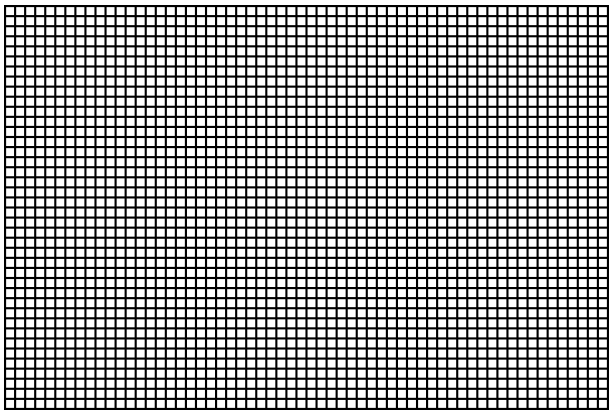


Figure 7 – Labyrinthe 40 x 60 avant construction

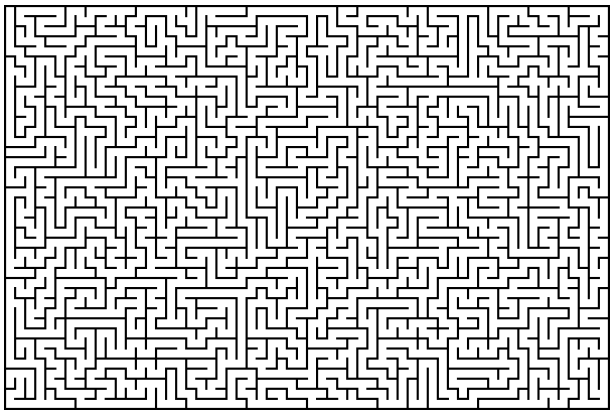


Figure 8 – Labyrinthe 40 x 60 final

Ci-dessous une partie du programme, à compléter.

```
from DefPile import *
import random
import numpy as np
from PIL import Image

def labyrinthe(n, m):
    # crée un labyrinthe de taille (n,m)

    def possibles(case):
        # donne la liste des cases contigues à case et disponibles
        # ces cases doivent être dans le tableau et valeur de laby à False


        [ A COMPLÉTER]


    def choix(L):
        # choisit au hasard un élément de L supposée non vide
        return L[random.randint(0, len(L) - 1)]

    def casse_mur(c1, c2):
        # efface le mur entre les cases c1 et c2
        x1, y1 = c1
        x2, y2 = c2
```

```
            if x1 > x2:
                x1, x2 = x2, x1
            if x2 > x1:
                deb_x, fin_x = x2*ep_totale, x2*ep_totale+ep_mur
                deb_y, fin_y = y1*ep_totale+ep_mur, (y1+1)*ep_totale
                dessin[deb_x:fin_x, deb_y:fin_y, :] = trait_blanc_h
            if y1 > y2:
                y1, y2 = y2, y1
            if y2 > y1:
                deb_x, fin_x = x1*ep_totale+ep_mur, (x1+1)*ep_totale
                deb_y, fin_y = y2*ep_totale, y2*ep_totale+ep_mur
                dessin[deb_x:fin_x, deb_y:fin_y,:] = trait_blanc_v

        p = creer_pile()
        laby = [ [False]*m for i in range(n) ]

        # initialisation du dessin: des cases blanches avec des murs autour
        ep_mur = 4
        ep_case = 15
        ep_totale = ep_mur +ep_case
        hauteur = ep_totale*n + ep_mur
        largeur = ep_totale*m + ep_mur
        dessin = np.zeros( (hauteur, largeur, 3), dtype='uint8')
        dessin.fill(255)
        mur_h = np.zeros( (ep_mur, largeur, 3), dtype='uint8')
        mur_h.fill(0)
        mur_v = np.zeros( (hauteur, ep_mur, 3), dtype='uint8')
        mur_v.fill(0)
        for i in range(0, n+1):
            dessin[i*ep_totale:i*ep_totale+ep_mur, :, :] = mur_h
        for j in range(0, m+1):
            dessin[:, j*ep_totale:j*ep_totale+ep_mur,:] = mur_v
        trait_blanc_h = np.zeros( (ep_mur, ep_case, 3) , dtype='uint8')
        trait_blanc_h.fill(255)
        trait_blanc_v = np.zeros( (ep_case, ep_mur, 3), dtype='uint8')
        trait_blanc_v.fill(255)

        laby[0][0] = True
        empiler(p, (0,0))
        while not est_vide(p):
            case = depiler(p)
            # print(case)
            L = possibles(case)
            if len(L) != 0:


                [ A COMPLÉTER]




        return Image.fromarray(dessin)

    im = labyrinthe(40,60)
    #im.save('Laby-40-60.png')
    im.show()
```

II.8. Résolution d'un Sudoku

Une grille de Sudoku est une grille de taille 9×9, découpée en 9 carrés de tailles 3×3.
Le but est de la remplir avec des chiffres de $[\![1\,;9]\!]$, de sorte que chaque ligne, chaque colonne, et chacun des 9 carrés de tailles 3×3 contienne une et une seule fois chaque entier de $[\![1\,;9]\!]$.

On représentera une grille de Sudoku par une liste de taille 9×9, c'est-à-dire une liste de 9 listes de tailles 9, dans laquelle les cases non remplies sont associées au chiffre 0. Ainsi la grille :

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

est représentée par la liste :

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

Les lignes du Sudoku sont alors les éléments accessibles par `L[i]` pour $0 \leqslant i \leqslant 8$, et l'élément de la case (i, j) est accessible par `L[i][j]` .
Pour parcourir la grille, il peut être plus commode de représenter une case par un numéro de l'ensemble $[\![0\,;80]\!]$, 0 correspondant à la case en haut à gauche et 80 à celle en bas à droite ; plus précisément, la case de numéro `num` a pour coordonnées `i=num//9` et `j=num%9` .
L'idée pour résoudre une grille est de procéder par « retour arrière » (*backtracking*). L'objectif est de compléter la grille en testant toutes les combinaisons en commençant par la première case vide, et jusqu'à la dernière ; si l'on obtient un conflit avec les règles, on est obligé de revenir en arrière.

Pour cela, on utilise une pile qui contient la suite des essais ; au départ, la pile contient la grille à résoudre. Ensuite tant que la pile n'est pas vide on fait :

1. dépiler une grille
2. dans cette grille, rechercher la première case vide ; s'il n'y en a pas, la grille est remplie et on l'ajoute à la liste des solutions (qui est une variable globale).
3. pour cette case, on examine si on peut y mettre une valeur qui respecte les règles du jeu ; si ce n'est pas possible, on retourne en **1.**, sinon on y met une valeur, on empile la grille ainsi complétée puis l'on retourne en **1.**