1 Exercice : palindrome

Un palindrome est un texte ou un mot dont l'ordre des lettres reste identique qu'on le lise de gauche à droite ou de droite à gauche.

Exemples : *radar*,*été*, *erre*, *kayak*, *rotor* sont des palindromes.

En utilisant une structure de pile, écrire une fonction `estPalindrome(s)` qui renvoie *True* si la chaîne de caractère *s* est un palindrome, *False* sinon.

Estimer la complexité de votre algorithme en espace et en temps.

[ ]:
```
def estPalindrome(s):
    '''
    entrée: s  = chaîne de caractère
               = mot à analyser
    sortie: booléen = Vrai si s est un palindrome
    '''
```

[3]:
```
# Script de validation de la fonction estPalindrome
# on utilise des mots avec un nombre de lettres qui est pair et impair

listeTest=['radar','été','math','erre','maths']
for mot in listeTest:
    print(mot,'\t',estPalindrome(mot))
```

```
radar    True
été      True
math     False
erre     True
maths    False
```

2 Exercice : notation polonaise inversée

La **notation polonaise inversée**, également appelée * notation post-fixée*, permet d'écrire de façon non ambiguë des formules arithmétiques sans utiliser de parenthèses.

Au lieu d'écrire les opérandes autour de l'opérateur (comme $4+5$), on écrit d'abord les opérandes, puis l'opérateur (`[ 4 , 5 , "+" ]`).

Par exemple, l'expression $3 \times (4+7)$ peut s'écrire:

`[ 3 , 4 , 7 , "+" , "*" ]`

ou `[ 4 , 7 , "+" , 3 , "*" ]`

De même l'expression

$$A = ((23-4) \times 4 + 11/3$$

peut s'écrire:

```
[ 24 , 4 , "-" , 4 , "*" , 11 , 3 , "/" , "+" ]
```

En utilisant une pile, écrire une fonction `evalueNPI(expr)` qui permet d'évaluer une expression en notation polonaise inversée. On utilisera en entrée une liste d'éléments étant : - ou bien des nombres (entiers ou flottants) pour les opérandes, - ou bien des caractères "+", "-", "*", "/" pour les opérateurs.

[ ]:
```
def evalueNP2(expr) :
    '''entrée = expr, expression étant une liste sous la forme␣
 ↪opérande, ou opérateur
    Sortie = valeur de l expression évaluée, ou erreur sinon'''
```

[6]:
```
# Validation de la fonction à l'aide de 4 expressions
E1=[4,5,'+']
E2=[24,4,"-",4,"*",11,3,"/","+"]
E3=[3,4,7,"+","*"]
E4=[4,7,"+",3,"*"]
listExp=[E1,E2,E3,E4]
for exp in listExp:
    print(exp,'valeur = ',evaluateNPI(exp))
```

```
[4, 5, '+'] valeur =  9
[24, 4, '-', 4, '*', 11, 3, '/', '+'] valeur =  83.66666666666667
[3, 4, 7, '+', '*'] valeur =  33
[4, 7, '+', 3, '*'] valeur =  33
```

[ ]:
```

```

3 Exercice : mélange de cartes

On souhaite simuler un mélange de cartes à l'aide de piles : - on prend le paquet, on le coupe au hasard. On obtient deux paquets, soit un dans chaque main. - on mélange les deux paquets en choisissant au hasard une carte de la main gauche ou une carte de la main droite jusqu'à épuisement des deux paquets.

Un paquet de cartes est assimilé à une liste d'entiers parmi $[\![1, n]\!]$.

1. Ecrire une fonction `coupe(paquet)` qui admet la pile *paquet* comme paramètre d'entrée, qui retire de cette pile un nombre aléatoire d'éléments de son sommet et renvoie une seconde pile comportant les éléments ayant été retiré.

Exemple : pour la pile `paquet = [ 1 , 2 , 3 , 4, 5 , 6 , 7 ]`, on tire au hasard un élément de $[\![1\ 6]\!]$.

Si on obtient 3, alors la fonction renvoie : `[ 5 , 6 , 7 ]` (les 3 éléments situés au dessus du paquet) et la pile initiale devient `[ 1 , 2 , 3 , 4 ]`

[4]:
```
from random import randrange
help(randrange)
```

```
Help on method randrange in module random:

randrange(start, stop=None, step=1, _int=<class 'int'>) method of␣
↪random.Random
instance
    Choose a random item from range(start, stop[, step]).

    This fixes the problem with randint() which includes the
    endpoint; in Python this is usually not what you want.
```

[ ]:
```
def coupe(paquet):
    '''Entrée: paquet est une pile d'entiers de n éléments
    Sortie : la pile initiale à laquelle on a retiré entre 1 et n-1␣
↪éléments
    et la pile des éléments retirés, dans le même ordre que la pile␣
↪initiale'''
```

[8]:
```
# Validation de la fonction coupe
paq1=[k for k in range(12)]
paq2=coupe(paq1)
print(paq1)
print(paq2)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[9, 10, 11]
```

2. Ecrire une fonction `melange(paquet1,paquet2)` qui prend en entrée peux piles et renvoie la pile mélangée.

[ ]:
```
def melange(paquet1, paquet2) :
    ''' Principe : on initialise une pile VIDE.
    ON boucle tant que paquet1 et paquet2 sont non vides,
    tirer au hasard un des paquets
    empiler la carte du dessus dans la pile résultat.
    Dès qu'un des paquets est vide, vide le plein dans le résulat.'''
```

[10]:
```
#Validation du mélange
paq1=[k for k in range(12)]
paq2=coupe(paq1)
print(paq1,paq2)
paq3=melange(paq1,paq2)
print(paq3)
```

```
print(paq1,paq2)
```

```
[0, 1, 2, 3, 4, 5, 6, 7] [8, 9, 10, 11]
[7, 6, 5, 11, 10, 4, 3, 9, 2, 8, 1, 0]
[] []
```

4 Exercice : Empiler, dépiler, rempiler, ...

4.1 Renversement d'une pile

Ecrire une fonction `renverse(p)` qui prend une pile en argument et renvoie une autre pile constituée des mêmes éléments placés dans l'ordre inverse. On s'autorise à vide la pile fournie en argument. Quelle est la complexité en temps et en espace de cette fonction~?

[ ]:
```
def renverse(p):
```

4.2 Accès à un élément par son indice

Ecrire une fonction `renvoie(p,n)` qui lit le n-ième élément d'une pile en comptant depuis son sommet :

pour n=0, la fonction renvoie l'élément du haut de la pile, pour n=1, celui immédiatement en dessus, etc...

On s'assurera que la pile en sortie contient toujours les mêmes éléments. On prévoira le cas où la pile n'est pas de taille suffisante pour que le n-ième élément existe, dans ce cas, la fonction ne renvoie rien.

Indication : on pourra utiliser une deuxième pile.

[ ]:
```
def renvoie1(p,n) :
    ''' Principe, on utilise une pile auxiliaire qui sert
    à stocker les n valeurs renversées puis on reconstitue toute la␣
 ↪pile
    en dépilant à noueau la pile auxilaire dans la pile initiale'''
```

4.3 Mise au fond

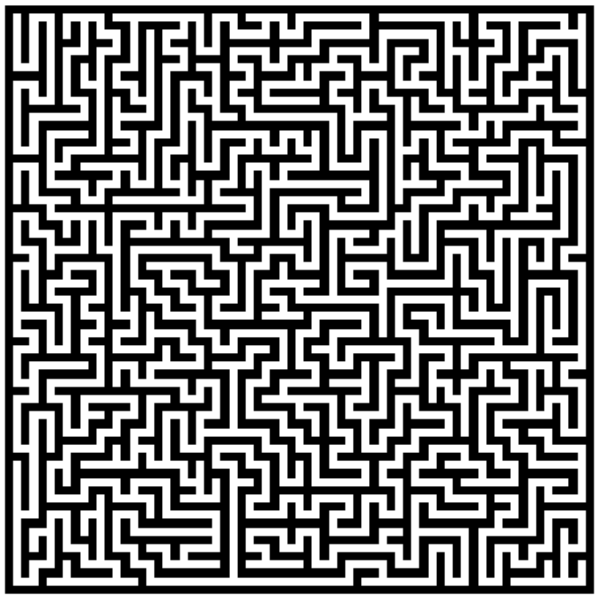
Ecrire une fonction `top2groud(p)` qui prend une pile non vide en argument et place l'élément situé à son sommet tout au fond de la pile, en conservant l'odre des autres éléments.

Quelle est sa complexité en temps et en espace ?

[5]:
```
def top2groud(p):
    '''Principe : on stocke le 1er élément, on dépile tout, on empile␣
 ↪le 1er élément
    On réempile tout'''
```

5 Projet : labyrinthe parfait

On cherche à construire un labyrinthe parfait de dimensions données. Il s'agit d'un labyrinthe où, pour toute paire de points, il existe un et un seul chemin entre ces deux points. Voici un exemple de labyrinthe parfait de dimension 30×30.



On considère le script ci-dessous. Voici son descriptif à lire attentivement.

On construit une matrice de booléens (n,n), *atteinte* indiquant, pour chaque case, si elle a déjà été atteinte par une chemin (initialement `False`), lignes (2 et 3).

On se donne deyx fonctions `visiter` et `est_atteinte` permettant respectivement de modifier (lignes 7 à 11) et consulter (lignes 13 à 17) le contenu de la matrice *atteinte*.

La fonction `choix` réalise l'opération suivante : étant donné une case *c*, (position (x,y)), elle renvoie les positions adjacentes non encore visitées (haut, bas, gauche ou droite). Le résultat est renvoyé sous la forme d'un tableau de 0 à 4 éléments.

On notera que cette fonction `choix` fait appel à la *fonction locale*, `ajouter(p)`, qui remplit ajoute la case `p` au tableau `r` si cette case n'est pas atteinte.

La dernière fonction auxilaire est la fonction `tirage(L)` qui prend un élément au hasard le tableau `L` fourni en argument et le renvoie. Elle sert à choisir aléatoirement parmi les directions possibles renvoyées par la fonction `choix` précédente. Cette fonction utilise la méthode randint() de la bibliothèqe random pour générer aléatoirement un nombre entier en *0* et *n-1* inclus.

Enfin, la fonction de construction du labyrinthe `labyrinthe` utilise une pile nommé *pile* contenant les emplacements à partir desquels on est susceptible de se déplacer. Initialement, on y place la case (0,0) (en haut à gauche) et on la marque comme visitée.

Puis, tant que la pile n'est pas vide, on extrait son sommet `cellule` (lignes 40 et 41). On examine alors les déplacements encore possibles (ligne 42) : s'il en existe au moins un, on en choisit un au hasard avec la fonction `tirage` (lignes 43 et 44). Cette nouvelle cellule choisie est notée `suivante`.

On **relie alors les cases** `cellule` et `suivante`: c'est la partie que vous avez à compléter en utilisant le tableau image `image0` dans la dimension doit être judicieusement choisie...

Enfin, les lignes 48 à 50 permettent de noter la case `suivante` comme visitée, puis d'empiler successivement ces deux cases dans la pile `pile`.

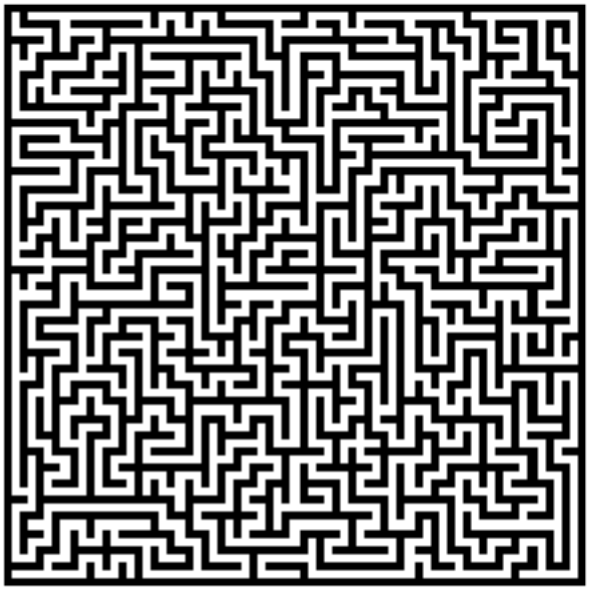
[3]:
```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import numpy as np
#construction d'un labyrinthe parfait
n = 35
atteinte = [[False] * n for i in range(n)]
def visiter(c): # écrit la case c comme atteinte.
    (x,y) = c
    if x<0 or x>=n or y<0 or y>=n:
        return # on sort de la fonction
    atteinte[x][y] = True

def est_atteinte(c): # donne l'état (atteinte ou non) de la case c.
    (x,y) = c
    if x<0 or x>=n or y<0 or y>=n:
        return True
    return atteinte[x][y]

def choix(c):
    (x,y)=c
    r=[]
    def ajouter(p):
        if not est_atteinte(p):
            r.append(p)
    ajouter ((x-1,y))
    ajouter ((x+1,y))
    ajouter ((x,y-1))
    ajouter ((x,y+1))
    return r
```

```python
def tirage(L):
    n=len(L)
    assert n>0
    return L[np.random.randint(0,n)]

def labyrinthe():
    # pile = creer pile # A MODIFIER
    # pile =  empile(pile,(0,0))# A MODIFIER
    visiter((0,0))
    while not pile# est_vide(pile) : A MODIFIER
        cellule# = dépiler(piler)
        c = choix(cellule)
        if len(c)>0:
            suivante=tirage(c)
            # relier les cases cellulee et suivante
            # A COMPLETER
            # ...
            visiter(suivante)
             # empiler(pile,cellue) A MODIFIER
             # empiler(pile,suivante) A MODIFIER
image0 = np.zeros((#,#, 3), dtype=np.uint8) # A COMPLETER
labyrinthe()
plt.imshow(image0)
plt.axis('off')
mpimg.imsave("laby1.png",image0)
```



Travail à faire : on demande de compléter le programme ci-dessus de manière à générer convenablement le labryinthe sous forme d'une image à l'écran et d'un fichier *.png* enregistré sur le disque dur.