Consignes

- Pour tous les exercices de cette planche on aura préalablement écrit l'implémentation des piles à capacité limitée ou illimitée vue en cours, et l'on ne s'autorisera à n'utiliser sur les piles que leurs seules primitives :

  creer_pile(), depiler(), empiler(,), top(,), taille(), est_vide()

- On travaillera au choix avec des piles à capacité limitée ou illimitée. (*Indication :* il est légèrement plus simple d'utiliser des piles à capacité illimitée, on n'a pas à s'occuper de leur capacité).

- Le corrigé utilise les piles à capacité limitée ; il s'adapte presque immédiatement aux piles à capacité illimitée en ignorant tout ce qui concerne la capacité des piles.

Exercice 1 : créer une pile renversée

Exercice 1.

1. Ecrire une fonction `renverse(pile)` prenant en argument une pile et qui retourne la pile obtenue en inversant l'ordre des éléments de `pile`. La pile `pile` pourra être vidée.
2. Même question, sauf qu'à la fin l'argument `pile` doit être inchangé.

Exercice 1 : créer une pile renversée - Correction

a) On s'autorise à vider `pile`.

```
def renverse(pile):
    n = taille(pile)
    pile2 = creer_pile(n) # supprimer n pour une pile illimitée
    while not(est_vide(pile2)):
        empiler(pile2,depiler(pile))
    return pile2
```

Exercice 1 : créer une pile renversée - Correction

b) A la fin `pile` doit être inchangée. On stocke les résultats dépilés dans une pile 'tampon', que l'on utilisera à la fin pour reconstituer le contenu de '`pile`'.

```
def renverse1(pile):
    n = taille(pile)
    pile2 = creer_pile(n)
    pile3 = creer_pile(n) # pour stoker les éléments dépilés
    while not(est_vide(pile)):
        lu = depiler(pile)
        empiler(pile2,lu)
        empiler(pile3,lu)
    while not(est_vide(pile3)): # Reremplir pile
        empiler(pile,depiler(pile3))
    return pile2
```

Exercice 2 : suppression d'un élément

Exercice 2. Ecrire une fonction `supprime(pile,p)` qui supprime dans la pile `pile` son p-ième élément compté en partant du sommet (indication : il faudra utiliser une deuxième pile).
Si p dépasse la taille de la pile la fontion retournera `False`, et sinon retournera `True`.

Exercice 2 : suppression d'un élément - Correction

• On utilise une deuxième pile pour stocker les $p-1$ premiers éléments dépilés avant de supprimer le p-ième élément puis de rempiler ces $p-1$ éléments.

```
def suprime(pile,p):
    if taille(pile) < p:
        return False
    pile2 = creer_pile(p-1) # supprimer p-1 si pile illimitée
    for i in range(p-1):
        empiler(pile2,depiler(pile))
    depiler(pile)
    for i in range(p-1):
        empiler(pile,depiler(pile2))
    return True
```

Exercice 3 : Echange de deux éléments

Exercice 3. Ecrire une fonction `echange(pile,p,q)` qui échange dans la pile `pile` ses p-ième et q-ième éléments, comptés en partant du sommet.

Si p ou q dépasse la taille de la pile la fonction retournera `False`, et sinon retournera `True`.

Exercice 3 : Echange de deux éléments - Correction

```
def echange(pile,p,q):
    assert isinstance(p,int) and isinstance(q,int)
    if taille(pile)<p or taille(pile)<q:    # Cas d'impossibilité
        return False
    if p == q:
        return True    # Ici rien à faire
    if p > q: # pour que p soit inférieur à q
        p,q = q,p
    pile2 = creer_pile(p)
    pile3 = creer_pile(q-p)
    for i in range(p):
        empiler(pile2,depiler(pile))
    for i in range(q-p):
        empiler(pile3,depiler(pile))
    empiler(pile,depiler(pile2))
    empiler(pile2,depiler(pile3))
    for i in range(q-p-1):
        empiler(pile,depiler(pile3))
    for i in range(p):
        empiler(pile,depiler(pile2))
    return True
```

Exercice 4 : Parenthésage (), [], {}

Adapter l'algorithme vu en cours pour déterminer si un mot est bien parenthésé pour englober les cas des 3 types de parenthèses (), [], {}.

Exercice 4 : Parenthésage (), [], {} - avec 3 + 1 piles

```
def parenthesage(mot):
    pile1 = creer_pile(len(mot))    # création de la pile ()
    pile2 = creer_pile(len(mot))    # création de la pile []
    pile3 = creer_pile(len(mot))    # création de la pile {}
    pile0 = creer_pile(len(mot))    # création de la pile 'mémoire"
    for char in mot:                 # Parcours du mot
        if char == '(':             # Si (
            empiler(pile1,0); empiler(pile0,1)  # on empile pile1 et on mémorise
        elif char == '[':             # Si [
            empiler(pile2,0); empiler(pile0,2)  # on empile pile2 et on mémorise
        elif char == '{':             # Si {
            empiler(pile3,0); empiler(pile0,3)  # on empile pile3 et on mémorise
        elif char == ')':          # Si )
            if est_vide(pile1) or depiler(pile0) != 1: return False
            else: depiler(pile1)
        elif char == ']':          # Si ]
            if est_vide(pile2) or depiler(pile0) != 2: return False
            else: depiler(pile2)
        elif char == '}':          # Si }
            if est_vide(pile3) or depiler(pile0) != 3: return False
            else: depiler(pile3)
    return est_vide(pile1) and est_vide(pile2) and est_vide(pile3)
```

Exercice 4 : Parenthésage (), [], {} - avec 1 pile

Ou encore utiliser une seule liste et empiler les différentes parenthèses ouvrantes :

```
def parenthesage(mot):
    pile = creer_pile(len(mot))    # création de la pile
    for char in mot:                # Parcours du mot
        if char == '(':             # Si (
            empiler(pile,'(')      # on empile '('
        elif char == '[':           # Si [
            empiler(pile,'[')      # on empile '['
        elif char == '{':           # Si {
            empiler(pile,'{')      # on empile '{'
        elif char == ')':           # Si )
            if est_vide(pile) or depiler(pile) != '(': return False
        elif char == ']':           # Si ]
            if est_vide(pile) or depiler(pile) != '[': return False
        elif char == '}':           # Si }
            if est_vide(pile) or depiler(pile) != '{': return False
    return est_vide(pile)
```

Cette approche est moins pratique que la précédente si l'on souhaite adapter le programme pour qu'il retourne aussi les positions des différentes parenthèses.

Exercice 5. Correction

• On adapte le programme vu en cours pour qu'il empile les positions des parenthèses ouvrantes rencontrées.

```
def parentheses(mot):
    pile = creer_pile(len(mot))
    result = [ ]
    for i in range(len(mot)):
        if mot[i] == '(':
            empiler(pile, i+1)
        elif mot[i] == ')':
            if est_vide(pile):
                return False
            result.append((depiler(pile), i+1))
    if est_vide(pile):
        return result
    else:
        return False
```

Exercice 5. Correction : 2ème version

• Plutôt que de parcourir la séquence "par indice" on peut mettre à profit l'instruction :

`for i, c in enumerate(seq)`

qui implémente une boucle `for` où :

– `c` parcourt les éléments de la séquence `seq`, tandis que

– `i` parcourt les indices correspondants.

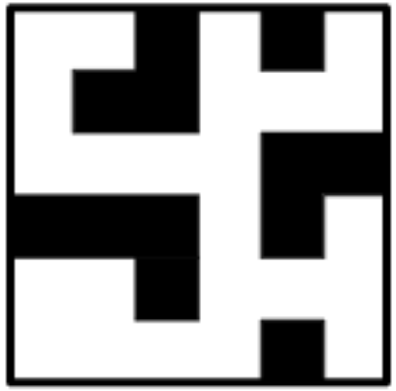
```
def parentheses(mot):
    pile = creer_pile(len(mot))
    result = [ ]
    for i, c in enumerate(mot):
        if c == '(':
            empiler(pile,i+1)
        elif c == ')':
            if est_vide(pile):
                return False
            result.append((depiler(pile),i+1))
    if est_vide(pile):
        return result
    else: return False
```

Exercice 6. Résolution d'un labyrinthe

On se donnera un labyrinthe carré par un tableau carré de type `ndarray` (`numpy`). Un 0 symbolisera un passage, un 1 un mur. On prendra par exemple :

```
import numpy as np
T = np.array([
[0,0,1,0,1,0],
[0,1,1,0,0,0],
[0,0,0,0,1,1],
[1,1,1,0,1,0],
[0,0,1,0,0,0],
[0,0,0,0,1,0]
])
```

pour :



De chaque case on peut se déplacer sur une case libre voisine horizontalement ou verticalement. Entrée et sortie du labyrinthe seront des cases données par leurs indices, par exemple $(0,0)$ et $(5,5)$.

Exercice 6. Résolution d'un labyrinthe

Le but de l'exercice est d'écrire une fonction qui détermine un chemin de l'entrée à la sortie du labyrinthe par une recherche en profondeur utilisant une pile. Le principe est :

- Chaque case sera symbolisée par le couple `(i,j)` de ses indices dans le tableau.
- Lorsqu'une case aura été déjà visité, sa valeur dans le tableau sera mise à -1.
- Une fonction `voisin()` prendra en paramètre le tableau et les indices d'une case et retournera la liste de ses cases voisines libres et non déjà visitées.
- De chaque case on se déplacera sur une telle case voisine libre et non déjà visitée.
- Une pile à capacité illimitée constituera le 'fil d'ariane' : on empilera les cases visitées successivement. Dès qu'une case n'aura plus aucune case voisine non déjà visitée on la dépilera, pour poursuivre le trajet à partir de la case précédente.

Exercice 6. Résolution d'un labyrinthe

1. Ecrire la fonction `voisin()` comme décrite ci-dessus.
2. Ecrire la fonction `trajet()` prenant en paramètre le tableau du labyrinthe, et les couples d'indices de l'entrée et de la sortie, et qui retourne la pile des cases successivement visitées constituant un trajet de l'entrée à la sortie.
3. La tester sur le tableau ci-dessus avec pour entrée `(0,0)` et sortie `(5,5)`.

Exercice 6. Résolution d'un labyrinthe

On utilise des piles à capacités illimitées.

1)

```
def voisins(T,v):
    V = []
    N = np.shape(T)[0]
    i,j = v[0],v[1]
    for a in (-1,1):
        if 0<= i+a <N:
            if T[i+a,j] == 0:
                V.append((i+a,j))
        if 0<= j+a <N:
            if T[i,j+a] == 0:
                V.append((i,j+a))
    return V
```

Exercice 6. Résolution d'un labyrinthe

```
from copy import deepcopy

def labyrinthe(T,entree,sortie):
    T = deepcopy(T)
    P = creer_pile()
    Recherche = True
    v = entree
    empiler(P,v)
    while Recherche:
        vois = voisins(T,v)
        if vois == []:
            depiler(P)
            if est_vide(P):
                return False
            v = top(P)
        else:
            i,j = vois[0]
            T[i,j] = -1
            empiler(P,(i,j))
            v = (i,j)
            if v == sortie:
                Recherche = False
    return P
```