Exercice *Dans les questions suivantes, les seules fonctions autorisées sur les piles sont celles définies ci-dessus.*

a) Ecrire les fonctions dup, swap, rot, agissant sur une pile *p* (passée en paramètre) définies comme suit :
   - dup duplique l'élément de tête de la pile
   - swap permute les deux éléments de tête de la pile
   - rot effectue une permutation circulaire des trois éléments de tête
   - rotn effectue une permutation circulaire des *n* éléments de tête

b) Ecrire deux fonctions, copier_pile et inverser_pile, de paramètre une pile *p*, qui renvoient respectivement une copie de *p* et une autre pile contenant les éléments de *p* dans l'ordre inverse sans modifier *p*.

```
def dup(p):
    empiler(p,lire_sommet(p))

def swap(p):
    a = depiler(p)
    b = depiler(p)
    empiler(p,a)
    empiler(p,b)

def rot(p):
    a = depiler(p)
    b = depiler(p)
    c = depiler(p)
    empiler(p,b)
    empiler(p,a)
    empiler(p,c)

def rotn(p,n):
    m = pile_vide()
    a = depiler(p)
    for i in range(n-1):
        empiler(m,depiler(p))
    empiler(p,a)
    for i in range(n-1):
        empiler(p,depiler(m))

def inverser_pile(p):
    m = pile_vide()
    n = pile_vide()
    while not est_vide(p):
        sommet = depiler(p)
        empiler(m,sommet)
        empiler(n,sommet)
    while not est_vide(n):
        empiler(p,depiler(n))
    return m

def copier_pile(p):
    m = pile_vide()
    n = pile_vide()
    while not est_vide(p):
        empiler(n,depiler(p))
    while not est_vide(n):
        sommet = depiler(n)
        empiler(p,sommet)
        empiler(m,sommet)
    return m
```

1. Exercices

Exercice 1

Dans cet exercice, les seules fonctions que l'on utilisera avec des piles sont pile_vide(), empiler(p,x), depiler(p), est_vide(p).

Un programmeur a écrit un petit script gérant l'inscription à une activité. Les personnes intéressées entrent leurs coordonnées sur un ordinateur. Leurs noms, adresses et téléphones sont stockés au fur et à mesure de leur inscription dans des piles nommées pile_nom, pile_adresse, pile_phone.

Ecrire une fonction compile(pile_nom, pile_adresse, pile_phone) qui renvoie une pile contenant les données des trois piles précédentes sous forme de tuples du type (*nom, adresse, téléphone*). La fonction affichera un message d'erreur et renverra une pile vide si les trois piles n'ont pas le même nombre d'éléments.

```
def compile(pile_nom, pile_adresse, pile_phone):
    pilecompile = pile_vide()
    while not (est_vide(pile_nom) or est_vide(pile_adresse) or
                                                    est_vide(pile_phone)):
        nom = depiler(pile_nom)
        adresse = depiler(pile_adresse)
        phone = depiler(pile_phone)
        empiler(pilecompile, (nom, adresse, phone))
    if est_vide(pile_nom) and est_vide(pile_adresse) and
                                                    est_vide(pile_phone):
        return pilecompile
    else:
        print("Les trois piles n'ont pas le même nombre d'éléments.")
        return []
```

Rappel : not (A or B) = (not A) and (not B)

Exercice 2 : Implémentation d'une file à l'aide de deux piles

On dispose de deux piles A et B, muni des fonctions habituelles, et on souhaite les utiliser comme une file. L'ajout des éléments se fait toujours dans la pile A. On retire les éléments de la pile B. Si celle-ci est vide, on transfère d'abord tous les éléments de la pile A vers la pile B.

a. Programmer les fonctions enfiler et defiler suivant le principe précédent.
b. On utilise cette file pour enfiler puis défiler *n* éléments. Quel est la complexité de l'algorithme correspondant ?

a.

```
def enfiler(A, B, x):
    empiler(A, x)

def defiler(A, B):
    if not est_vide(B):
        return depiler(B)
    while not est_vide(A):
        empiler(B, depiler(A))
    return depiler(B)
```

b.

Enfiler est en temps constant, donc enfiler *n* éléments est en O(*n*).
Défiler le premier va être en O(*n*), puis en temps constant pour chacun des *n* – 1 suivants donc en O(*n*) pour l'ensemble. L'algorithme complet est en O(*n*) (somme de 3 O(*n*)).

Exercice 3

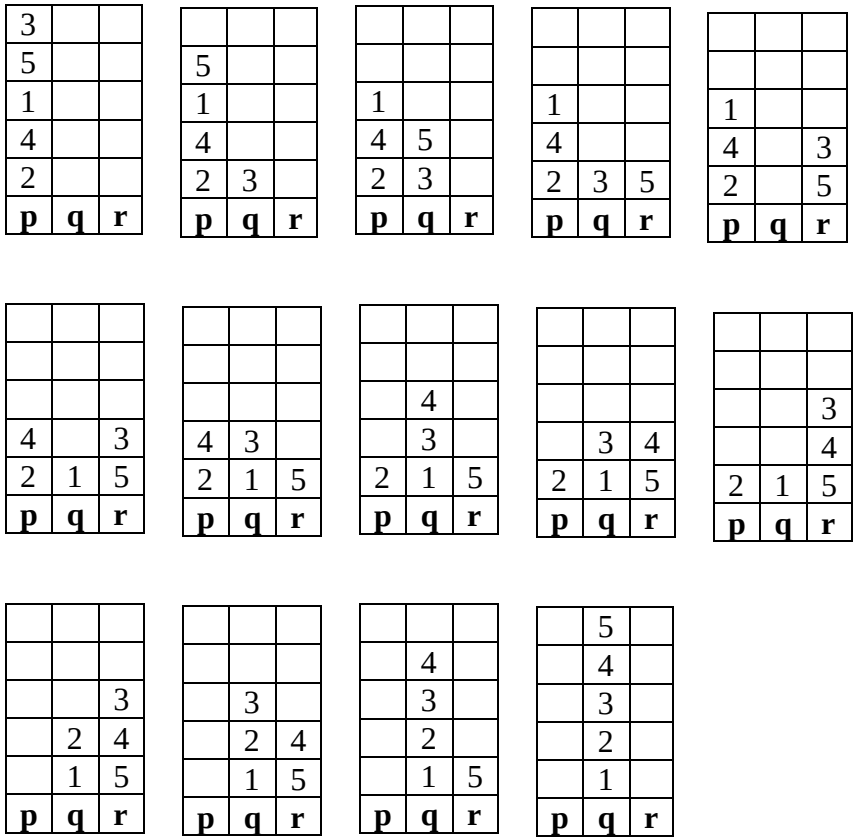
```
def parenthesage(chaine):
    pile = pile_vide()
    for c in chaine:
        if c in ['(', '[', '{']:
            empiler(pile, c)
        elif c in [')', ']', '}']:
            if est_vide(pile):
                return False
            d = depiler(pile)
            if (d, c) not in [('(',')'), ('[',']'), ('{','}')]:
                return False
    return est_vide(pile)
```

Exercice 4

```
def switch(pile):
    if est_vide(pile):
        return pile_vide()
    npile = pile_vide()
    tpile = pile_vide()        # pile temporaire
    s = depiler(pile)
    empiler(npile, s)
    empiler(tpile, s)
    if est_vide(pile):         # cas pile à un élément
        empiler(pile, s)
        return npile
    while not est_vide(pile):
        empiler(tpile, depiler(pile))
    nsommet = depiler(tpile)
    empiler(pile, nsommet)
    while not est_vide(tpile):
        s = depiler(tpile)
        empiler(npile, s)
        empiler(pile, s)
    depiler(npile)         # on supprime l'ancien sommet de npile
    empiler(npile, nsommet)
    return npile
```

Exercice 5 : Tri d'une pile

1.



2. Ecrire une fonction tri_pile de paramètre une pile *p* contenant des entiers renvoyant la pile *p* triée. Cette fonction n'utilisera pour seules variables que deux autres piles *q* et *r*, munies de leurs fonctions habituelles.

```
def tri_pile(p):
    q = pile_vide()
    r = pile_vide ()
    while not est_vide(p):
        if est_vide(q) or lire_sommet(p) > lire_sommet(q):
            empiler(q, depiler(p))
            while not est_vide(r):
                empiler(q, depiler(r))
        else:
            empiler(r, depiler(q))
    return q
```

L'algorithme finalement programmé est un peu différent de l'algorithme exécuté à la main. Dans l'algorithme programmé, après avoir déplacé un élément de la pile p vers la pile q, on dépile à chaque fois toute la pile r dans la pile q, ce qui est un peu moins efficace, mais plus simple à programmer. On peut bien sûr programmer l'algorithme tel qu'il a été exécuté à la main.

Algorithme programmé

3		
5		
1		
4		
2		
p	**q**	**r**

5		
1		
4		
2	3	
p	**q**	**r**

1		
4	5	
2	3	
p	**q**	**r**

1		
4		
2	3	5
p	**q**	**r**

1		
4		3
2		5
p	**q**	**r**

4		3
2	1	5
p	**q**	**r**

4	3	
2	1	5
p	**q**	**r**

	5	
4	3	
2	1	
p	**q**	**r**

4	3	
2	1	5
p	**q**	**r**

	4	
	3	
2	1	5
p	**q**	**r**

	5	
	4	
	3	
2	1	
p	**q**	**r**

	4	
	3	
2	1	5
p	**q**	**r**

	3	4
2	1	5
p	**q**	**r**

		3
		4
2	1	5
p	**q**	**r**

		3
	2	4
	1	5
p	**q**	**r**

	3	
	2	4
	1	5
p	**q**	**r**

	4	
	3	
	2	
	1	5
p	**q**	**r**

	5	
	4	
	3	
	2	
	1	
p	**q**	**r**