Épreuves écrites de NSI
Correction du sujet
Métropole 2 2021
Journée 1

Exercice 2

1. a. On peut écrire le code suivant :

```
pile1 = Pile()
pile1.empiler(7)
pile1.empiler(5)
pile1.empiler(2)
```

b. On obtient l'affichage suivant : `7, 5, 5, 2`

2. a.
* cas n° 1 : `3, 2`
* cas n° 2 : `3, 2, 5, 7`
* cas n° 3 : `3`
* cas n° 4 : on n'affiche rien

b. La fonction `mystere` donne une pile constituée des éléments situés au-dessus de `element` dans `pile` en incluant cet élément lui-même et en y mettant tous les éléments de `pile` s'il ne s'y trouve pas. Dans la pile renvoyée par la fonction `mystere`, ces éléments seront empilés dans l'ordre inverse de leur empilement dans `pile`.

3. On peut écrire le code suivant :

```
def etendre(pile1, pile2) :
 while not pile2.est_vide() :
      el = pile2.depiler()
      pile1.empiler(el)
```

4. On peut écrire le code suivant :

```
def supprime_toutes_occurrences(pile, element) :
    pile2 = Pile()
    while not pile.est_vide() :
        el = pile.depiler()
        if el != element :
            pile2.empiler(el)
    while not pile2.est_vide() :
        el = pile2.depiler()
        pile.empiler(el)
```

Exercice 4

Partie A

1. Il faut écrire l'instruction suivante :

```
lab2[1][0] = 2
```

2. On peut écrire le code suivant :

```
def est_valide(i, j, n, m) :
```

```
    return (0 <= i < n) and (0 <= j < m)
```

3. On peut écrire le code suivant :

```
def depart(lab) :
    n = len(lab)
    m = len(lab[0])
    for i in range(n) :
        for j in range(m) :
           if lab[i][j] == 2 :
               return (i, j)
```

4. On peut écrire le code suivant :

```
def nb_cases_vides(lab) :
    n = len(lab)
    m = len(lab[0])
    cpt = 0
    for i in range(n) :
        for j in range(m) :
            if lab[i][j] != 1 :
                cpt += 1
    return cpt
```

Partie B

1. L'appel `voisines(1, 2, [[1, 1, 4], [0, 0, 0], [1, 1, 0]])` renvoie `[(1, 1), (2, 2)]`.

2. a. La suite d'instructions se poursuit de la façon suivante :

```
...
chemin.append((3, 3))
chemin.append((3, 4))
chemin.pop()
chemin.pop()
chemin.pop()
chemin.append((1, 4))
chemin.append((1, 5))
```

b. On peut écrire le code suivant :

```
def solutions(lab) :
   chemin = [depart(lab)]
    case = chemin[0]
    i = case[0]
    j = case[1]
    while lab[i)[j] != 3 :
        lab[i)[j] = 4
        list_vois = voisines(i, j, lab)
        if len(list_vois) != 0 :
            case = list_vois[0]
            chemin.append(case)
            i = case[0]
            j = case[1]
        else :
```

```
            chemin.pop()
            case = chemin[len(chemin)-1] #ou chemin[-1]
            i = case[0]
            j = case[1]
    return chemin
```

Exercice 5

Questions préliminaires

1. Dans le tableau `[4, 8, 3, 7]` l'élément d'indice `1` est `8` est l'élément d'indice `3` est `7`. Or `1 < 3` et `8 > 7` donc le couple `(1, 3)` est une inversion pour ce tableau.

2. Pour ce même tableau, l'élément d'indice `2` est `3` est l'élément d'indice `3` est `7`. Or `2 < 3` et `3 < 7`. Donc le couple `(2, 3)` n'est pas une inversion pour ce tableau.

Partie A

1. a.
* cas n° 1 : la fonction renvoie 0, car il n'y a pas d'inversion dans le tableau de la forme `(0, j)` où `j > 0`.
* cas n° 2 : la fonction renvoie 1, car il y a une seule inversion dans le tableau de la forme `(1, j)` où `j > 1`.
* cas n° 3 : la fonction renvoie 2, car il y a deux inversion dans le tableau de la forme `(1, j)` où `j > 1`.

b. La fonction `fonction1(tab, i)` renvoie le nombre d'inversions dans le tableau `tab` de la forme `(i, j)` où `j > i`.

2. On peut écrire le code suivant :

```
def nombre_inversions(tab) :
    lg = len(tab)
    cpt = 0
    for i in range(lg - 1):
        cpt += fonction1(tab, i)
    return cpt
```

3. La fonction `nombre_inversions` est de complexité quadratique, donc en $O(n^2)$. En effet, la fonction `fonction1` comporte une boucle allant de `i+1` à la fin du tableau et que cette fonction est elle-même appelée à chaque passage dans boucle `for i in range(lg - 1),`.

Partie B

1. L'algorithme de tri par fusion est de complexité $O(n.\log_2(n))$ qui est donc meilleure que la complexité quadratique, soit en $O(n^2)$.

2. On peut écrire le code suivant :

```
def moitie gauche(tab) :
    lg = len(tab)
    m = (lg+1)//2
    res = []
    for i in range(m) :
        m.append(tab[i])
    return res
```

ou bien, avec la compréhension de liste :

```
def moitie_gauche(tab) :
    lg = len(tab)
    m = (lg+1)//2
    return [tab[i] for i in range(m)]
```

3. On peut écrire le code suivant :

```
def nb_inversions_rec(tab) :
    if len(tab) < 2 :
        return 0
    tab1 = moitie_gauche(tab)
    tab2 = moitie_droite(tab)
    n1 = nb_inversions_rec(tab1)
    n2 = nb_inversions_rec(tab1)
    tab1 = tri(tab1)
    tab2 = tri(tab2)
    return n1 + n2 + nb_inv_tab(tab1, tab2)
```