DS 1 : Piles & Récursivité
Éléments de correction

N°	Elts de rép.	Pts	Note
00-00	**Titre de l'exo**	0	0
0	éléments de réponse	0	0

01-08	**Les piles**		
1	L'acronyme LIFO signifie que l'on a accès uniquement au dernier élément ajouté dans une pile.	1	
2	Pour mettre en œuvre les historiques de consultation avec les fonctions annuler et avancer.	1	
3	avantages : nombre de variable inconnu à l'avance, limitation d'accès à la seule dernière variable utilisée, structure de donnée plus facile à mettre en place. inconvénient : si on veut pouvoir accéder à n'importe quel élément, il vaut mieux utiliser un tableau	1	
4	On peut utiliser une liste. En effet elles sont de taille variable et les méthodes .append() et .pop() réalisent les fonctions dépiler et empiler des piles.	1	
5	La fonction empiler a deux arguments : une pile p qui a la structure d'une liste et un élément v, elle modifie la liste p en ajoutant l'élément v à la fin de la liste avec la méthode .append() , et elle n'a pas de sortie car la liste p est modifié comme souhaité par la fonction empiler sur une pile. La fonction depiler a un seul argument : une pile p qui a la structure d'une liste, elle vérifie d'abord que la liste est non vide avec un assert, puis elle utilise la méthode .pop() pour retirer le dernier élément de la liste p, et a comme sortie le dernier élément retiré.	1	

6

écrivons une première fonction renverser :

```
    def renverser(p):
      p1 = creer_pile()
      p2 = creer_pile()
      while not(est_vide(p)):
        empiler(p1,depiler(p))
      while not(est_vide(p1)):
        empiler(p2,depiler(p1))
      while not(est_vide(p2)):
        empiler(p,depiler(p2))


```

Puis la fonction superposer

```
    def superposer(p1,p2):
      renverser(p2)
      while not(est_vide(p2)):
        empiler(p1,depiler(p2))


```

1

7

```
    (0, 1)
    (3, 4)
    (5, 6)
    (2, 7)
    True

```

1

8

```
    def parentheses+acolades(s):
      p = creer_pile() # on cree une pile p
      for i in range(len(s)): # on parcourt tous les
      caracteres du 'mot' de la gauche vers la droite
        if s[i] == '(': # si on rencontre une parenthese
      ouvrante, on vient d'ouvrir une parenthese
          empiler(p, '(') # alors on note dans la liste
      qu'on a ouvert une parenthese
        elif s[i] == '{': # si on rencontre une acolade
      ouvrante, on vient d'ouvrir une acolade
          empiler(p, '{') # alors on note dans la liste
      qu'on a ouvert une acolade
        elif s[i] == '}': # si on rencontre une acolade
      fermante, on vient de fermer une acolade
          if est_vide(p): # si la pile est vide ca veut
      dire qu'on n'a pas ouvert d'acolade
            return False # donc le mot est mal parenthese
          elif depiler(p) == '(': # si le dernier
      caractere ouvert est une parenthese
            return False # le mot est mal parenthese car
      de la forme '{(})', en verifiant on a retire l'
      element ouvrant corespondant
        else: # sinon c'est que c'est une parenthese
      fermante
          if est_vide(p): # si la pile est vide ca veut
      dire qu'on n'a pas ouvert de parenthese
            return False # donc le mot est mal parenthese
          elif depiler(p) == '{': # si le dernier
      caractere ouvert est une acolade
            return False # le mot est mal parenthese car
      de la forme '({)}', en verifiant on a retire l'
      element ouvrant corespondant
      return est_vide(p) # une fois le mot parcouru, on
      verifie qu'il ne reste pas de parenthese ou d'
      acolade ouverte

```

1

09 - 20	**La Récursivité**		
9	Une fonction est dite récursive si elle s'appelle elle même.	1	
10	factorielle1 n'appelle pas factorielle1 dans sa définition elle n'est pas récursive factorielle2(n) appelle factorielle2(n-1) dans sa définition, elle s'appelle elle même, elle est récursive	1	
11	factorielle2(n) possède comme cas d'arrêt le cas n = 1 et elle s'appelle elle même avec un argument décroissant n-1. Donc pour toute valeur de n < 1, factorielle2 n'atteint jamais son cas d'arrêt. D'autre part le nombre n décroit que par pas de 1 donc on atteint le cas d'arrêt que si n est entier. Le programme ne fonctionne donc pas pour toute valeur de n non entière.	1	
12	 `import numpy as np` `def u(n):` `  if n == 0 :` `  return 0` `  elif n == 1 :` `  return 1` `  else :` `  retrun np.sqrt(u(n _ 1)+u(n _ 2))`	1	
13	il faut ré-écrire légèrement la relation de récurrence comme $v_{n+3} = \frac{1}{4} \times (v_{n+2}^2 - v_n)$ `def v(n):` `  if n == 0 :` `  return 0` `  elif n == 1 :` `  return 1` `  elif n == 2 :` `  return 1` `  else :` `  retrun (v(n _ 1)**2 _ v(n _ 3))/4`	1	

14	 ``` def f(n): if n == 0 : return 0 else : retrun g(n _ 1) def g(n): if n == 0 : return 1 else : retrun f(n _ 1) ``` $f(1) = g(0) = 1$ $f(2) = g(1) = f(0) = 0$ $f(3) = g(2) = f(1) = g(0) = 1$ $f(4) = g(3) = f(2) = g(1) = f(0) = 0$ cette fonction teste la parité de son argument et renvoie le reste de sa division par 2.	1	
15	La fonction écrite à la question 12 appelle sa propre fonction pour les arguments n-1 et n-2, donc sa complexité $C(n) = C(n-1) + C(n-2)$ avec $C(0) = C(1) = 1$. On peut remarquer que chaque appel récursif multiplie par 2 le nombre d'appel de la fonction, on va donc avoir une complexité exponentielle. On peut montrer que pour n grand $C(n)$ est équivalent à Φ^n avec Φ solution de $\Phi^n = \Phi^{n-1} + \Phi^{n-2}$ soit $\Phi = \dfrac{1+\sqrt{5}}{2}$	1	
16	 ``` import numpy as np def u(n): l = [0,1] i = 2 while i<=n : l.append(np.sqrt(l[i _ 1]+l[i _ 2])) i += 1 return l[n] ``` On doit faire n étape de boucle donc la complexité est $C(n) = n \times 1 = O(n)$, c'est une complexité linéaire.	1	

17	Les programmes récursifs permettent d'écrire de manière naturelle et concise, notamment sans boucle les fonctions définies par récurrence. Mais il faut faire attention à la complexité des programmes récursifs qui peut devenir exponentielle.	1	
18	La fonction cercle prend en argument les coordonnée x et y, ainsi que le rayon r, et trace le cercle de centre (x,y) et de rayon r, à l'aide de 100 points tracé tous les 3,6° et relié entre eux.	1	
19	 `import matplotlib.pyplot as plt` `import numpy as np` `def cercle(x,y,r):` `  theta = np.linspace(0, 2*np.pi, 100)` `  X = r*np.cos(theta)+x` `  Y = r*np.sin(theta)+y` `  plt.plot(X,Y)` `def bulles1(n):` `  def bulles(n,x,y,r):` `  cercle(x,y,r)` `  if n > 1 :` `  bulles(n _ 1,x+3*r/2,y,r/2)` `  bulles(n _ 1,x,y _ 3*r/2,r/2)` `  bulles(n,0,0,1)`	1	
20	Il faut faire attention à ne pas tracer de cercle dans les cercles déjà présent. Pour cela il faut garder la trace d'où se situe le cercle tracé 'haut', 'bas', 'droite', 'gauche'. `import matplotlib.pyplot as plt` `import numpy as np` `def cercle(x,y,r):` `  theta = np.linspace(0, 2*np.pi, 100)` `  X = r*np.cos(theta)+x` `  Y = r*np.sin(theta)+y` `  plt.plot(X,Y)` `def bulles2(n):` `  def bulles(n,x,y,r,d):` `  cercle(x,y,r)` `  if n > 1 :` `  if d!='bas' :` `  bulles(n _ 1,x,y+3*r/2,r/2,'haut')` `  if d!='haut' :` `  bulles(n _ 1,x,y _ 3*r/2,r/2,'bas')` `  if d!='gauche' :` `  bulles(n _ 1,x+3*r/2,y,r/2,'droite')` `  if d!='droite' :` `  bulles(n _ 1,x _ 3*r/2,y,r/2,'gauche')` `  bulles(n,0,0,1,'')`	1	