INFORMATIQUE MP DEVOIR 1

CORRECTION

EXERCICE 1 - TRIS ET ÉTUDES

1. Par passages en boucle :

```
Au début [5,2,3,1,4]
i=1 [5,2,3,1,4] --> [2,5,3,1,4] fin
i=2 [2,5,3,1,4] --> [2,3,5,1,4] fin
i=3 [2,3,5,1,4] --> [2,3,3,5,4] --> [2,2,3,5,4] --> [1,2,3,5,4] fin
i=4 [1,2,3,5,4] --> [1,2,3,4,5] fin
```

2. Dans le meilleur des cas la liste est déjà triée : dans la boucle for on fait un nombre constant de comparaisons et d'affectations, (on ne rentre pas dans le while), on a donc une complexité $O(1)$ par passage en boucle, soit une complexité $O(n)$ au total.
   Dans le pire des cas la liste est triée dans l'ordre décroissante : au $i$ème passage dans la boucle for on rentre $i-1$ dans la boule while, on a donc une complexité $O(n-i)$ par passage en boucle, soit une complexité $O(n^2)$ au total.

3.
```
def fusion(liste1,liste2):
    """ fusion de listes triées"""
    liste_fusionnee=[]
    i,j=0,0
    while (i<len(liste1)) and (j<len(liste2)):
        if liste1[i]<liste2[j]:
        liste_fusionnee.append(liste1[i])
        i+=1
        else:
            liste_fusionnee.append(liste2[j])
            j+=1
    if len(liste1)==i:liste_fusionnee+=liste2[j:]
    else:liste_fusionnee+=liste1[i:]
    return liste_fusionnee


def tri_fusion(liste):
    """ découpage et tri"""
    n=len(liste)
    if n==1: return liste
    if n>1:
        liste1=liste[0:n//2]
        liste2=liste[n//2:n]
        liste1=tri_fusion(liste1)
        liste2=tri_fusion(liste2)
        return fusion(liste1,liste2)
```

4. Cf cours $O(n \ln n)$.

5.
```
def tri_chaine(L):
    n = len(L)
    for i in range(1,n):
        j = i
        x = L[i]
        while 0 < j and x[1] < L[j-1][1] :
            L[j] = L[j-1]
            j = j-1
        L[j]=x
```

6. Bon la je l'ai appelé min

```
def min_par_chaine(L,i):
    n = len(L)
    pos = i
    for k in range(i,n):
        if L[pos][1]>L[k][1] :
            pos = k
    return(pos)
```

7. ```
   def tri_selection_chaine(L):
       for i in range(len(L)):
          m = min_par_chaine(L,i)
          L[i],L[m]=L[m],L[i]
   ```

EXERCICE 2 - PILES

1. ```
   def est_vide(p):
     return(taille(p)==0)

   def depiler(p):
      n= p[0]
      assert n > 0
      p[0] = n -1
      return(p[n])

   def empiler(p,v):
      n= p[0]
      assert n < len(p)-1
      n = n+1
      p[0] = n
      p[n] = v
   ```

2. Écrire une fonction `intervert(p)` qui échange les deux éléments en haut de la pile `p`.

   ```
   def intervert(p):
      assert(taille(p)>=2)
      x=depiler(p)
      y=depiler(p)
      empiler(p,x)
      empiler(p,y)
   ```

3. Écrire une fonction `renverse(p)` qui renvoie une autre pile constituée des mêmes éléments dans l'ordre inverse..

   ```
   def renverse(p):
       q = creer_pirle(taille(p))
       while not est_vide(p):
           empiler(q,depiler(p))
       return(q)
   ```

4. ```
   def melange(P,Q):
       M = creer_pile(taille(P)+taille(Q))
       while not (est_vide(P) or est_vide(Q)):
           if random.randint(0,1) == 1:
               empiler(M,depiler(P))
           else :
               empiler(M,depiler(Q))
       while not est_vide(P):
           empiler(M,depiler(P))
       while not est_vide(Q):
           empiler(M,depiler(Q))
       return(M)
   ```
   ```

EXERCICE 3 - AUTOUR DE LA RÉCURSIVITÉ

Partie A

1. On fait une multiplication par appel, et n appels récursif en tout : donc n multiplications.
2. (a)
```
puissance_rapide(2,7)
    r1 =  puissance_rapide(2,3)
        r2 =  puissance_rapide(2,1)
            r3 =  puissance_rapide(2,0) = 1
        r2 = 2 * 1 * 1 = 2
    r1 = 2 * 2 * 2 = 8
puissance\_rapide(2,7) = 2*8*8=128


puissance_rapide(2,8)
    r1 =  puissance_rapide(2,4)
        r2 =  puissance_rapide(2,2)
             r3 =  puissance_rapide(2,1) = 1
                 r4 = puissance_rapide(2,0) = 1
             r3 = 2*1*1=2
        r2 = 2 * 2  = 4
    r1 = 4 * 4  = 16
puissance\_rapide(2,7) = 16*16=256
```
   (b) Déterminer la complexité de cet algorithme. On note $C(n)$ le nombre d'opérations pour un appel à `puissance_rapide(x,n)`. On a :

$$C(n) \leqslant 2 + C(\lfloor n/2 \rfloor) \leqslant 4 + C(\lfloor n/2^2 \rfloor) \leqslant ... \leqslant 2 \times log_2(n)$$

   On a donc $C(n) = O(\log(n))$ soit une complexité logarithmique.

Partie B

1. Bon $u_0 = 1$ ou $u_0 = 2$ ça ne changeait rien...

```
u(2)
1er appel u(1)
    1er appel  u(0) = 1
    2e  appel  u(0) = 1
    u(1)=3 \times 1 ^2 - 1 +5 =7
2e appel u(1)
    1er appel  u(0) = 1
    2e  appel  u(0) = 1
    u(1)=3 \times 1 ^2 - 1 +5 =7
u(2) =  3 \times 7 ^2 - 7 +5 =145
```

2. On pose $H(n)$ " L'appel à `u(n)` termine et renvoie la valeur u_n".
   Initialisation : L'appelle `u(0)` renvoie $1 = u_0$ donc $H(0)$ est vraie.
   Hérédité : Soit $n > 0$, supposons que $H(n-1)$ est vraie.
   L'appel à `u(n)` fait deux appels à `u(n-1)` qui renvoient la valeur u_{n-1} par H.R. , l'appel à `u(n)` termine donc et renvoie $3u_{n-1}^2 - u_{n-1} + 5 = u_n$ donc $H(n)$ est vraie.
   Conclusion : $H(n)$ est bien vraie pour tout $n \in \mathbb{N}$
3. On a $C(0) = 0$ et $C(n) = 2 + 2C(n-1)$ donc $C(n) = 2^{n+1} - 2$ soit une complexité exponentielle.
4.
```
def u(n):
   assert n>=0
   if n == 0 :
      return 1
   else :
      x = u(n-1)
      return 3*x**2-x+5
```
   Ici la complexité vérifie $C(n) = 2 + C(n-1)$ et $C(0) = 0$, soit $C(n) = 2n$, une complexité linéaire.