INFORMATIQUE MP DEVOIR 1

CORRECTION

EXERCICE 1 - TRIS ET ÉTUDES

On propose une version du tri par insertion en pseudo langage.

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

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. Soit L une liste non vide d'entiers ou de flottants. Montrer que ≪ la liste `L[0:i+1]` (en convention Python) est triée par ordre croissant à l'issue de l'itération `i` ≫ est un invariant de boucle. En déduire que `tri(L)` tri la liste `L`.

   Notons $P(i)$:" à l'issue de l'itération i, la variable L est telle que $L[0:i+1]$ triée dans l'ordre croissant."
   La liste $L[0:0+1]$ est la liste ayant une seule valeur $L[0]$ donc $P(0)$ est vraie.
   Soit $i \geqslant 0$. Supposons $P(i)$ vrai et $i+1 \leqslant n$ (i.e. on fait encore une itération). On a $L[0:i+1] = [x_0, ..., x_i]$ avec $x_0 \leqslant ... \leqslant x_i$. Au début de l'itération suivante j vaut $i+1$ et $x = L[i+1]$.
   La boucle "while" permet de faire décroître j et de s'arrêter dès que $L[j] \leqslant x$. Quand on s'arrête, on a donc dans $L[0:i+2]$ la valeur $[x_0, ..., x_{j-1}, x_j, x_j, ...x_i]$, puis la ligne 9 assure que $L[0:i+2] = [x_0, ...x_{j-1}, x, x_j, ...x_i]$ i.e. c'est bien $L[0:i+2]$ trié .
   Ainsi $P(i)$ est bien un invariant de boucle : la fonction trie bien le tableau.

3. Évaluer la complexité dans le meilleur et le dans le pire des cas de l'appel `tri(L)` en fonction du nombre d'éléments `n` de `L`.

   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.

4. Citer un algorithme de tri plus efficace dans le pire des cas. Quelle en est sa complexité dans le meilleur et le pire des cas ?
   Dans le meilleur et pire des cas le tri fusion est en $O(n \ln n)$.

On souhaite, partant d'une liste constituée de couples (chaîne, entier), trier la liste par ordre croissant de l'entier associé suivant le fonctionnement suivant :

```
>>> L = [['Bresil', 76], ['Kenya', 26017], ['Ouganda', 8431]]
>>> tri_chaine(L)
>>> L
[['Bresil', 76], ['Ouganda', 8431], ['Kenya', 26017]]
```

5. Écrire une fonction python `tri_chaine` réalisant cette opération.

```
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
```

On considère toujours une liste constituée de couples (chaîne, entier). On souhaite déterminer quelques indicateurs sans avoir à trier la liste.

6. Écrire une fonction python `max_chaine` qui renvoie la chaîne de caractère dont l'entier correspondant est le maximum.

```
def max_chaine(L):
    n = len(L)
    maxi = 0
    for i in range(1,n):
        if L[maxi][1]<L[i][1] :
            maxi = i
    return(L[maxi][0])
```

7. Écrire une fonction python `moy_chaine` qui renvoie la moyenne des entiers des couples de la liste.

```
def moy_chaine(L):
    n = len(L)
    moy = 0
    for i in range(n):
        moy += L[i][1]
    return(moy/n)
```

8. Écrire une fonction python `pays_moy_chaine` qui renvoie la liste des chaînes de caractères dont l'entier correspondant a un écart à la valeur moyenne des entiers de plus ou moins 10%.

```
def pays_moy_chaine(L):
    n = len(L)
    pays = []
    moy = moy_chaine(L)
    for i in range(n):
        if 0.9*moy < L[i][1] < 1.1*moy :
            pays.append(L[i][0])
    return(pays)
```

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. On considère deux piles `A` et `B`. On souhaite mélanger les deux piles de la manière suivante :

```
tant que B n'est pas vide:
    si A n'est pas vide
      on renverse A
      on dépile le sommet de A que l'on stocke
      on empile le sommet de B sur A
      on rempile le sommet de A sur A
```

Écrire une suite d'instructions réalisant ce processus.

```
while not est_vide(B):
    if not est_vide(A):
        A = renverse(A)
        x = depiler(A)
        empiler(A,depiler(B))
        empiler(A,x)
```

EXERCICE 3 - AUTOUR DE LA RÉCURSIVITÉ

Partie A

On s'intéresse d'abord au calcul de la puissance k-ième d'un entier naturel n.

1. On considère l'algorithme naïf :

```
def puissance(x,n):
    if n == 0:
       return 1
    else :
       return x * puissance(x,n-1)
```

Quelle est la complexité de cette algorithme ? On fait une multiplication par appel, et n appels récursif en tout : donc n multiplications.

2. On utlise maintenant l'algorithme :

```
def puissance_rapide(x,n):
    if n == 0:
       return 1
    else :
       r = puissance_rapide(x,n // 2)
       if n % 2 ==0:
           return r * r
        else :
           return x * r * r
```

(a) Décrire l'exécution de l'appel `puissance_rapide(2,7)`.

```
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
```

(b) Décrire l'exécution de l'appel `puissance_rapide(2,8)`.

```
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
```

(c) 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

```
def euclide(a,b):
   if b==0:
      return(a)
   else:
      r = a%b
      return(pgcd(b,r))
```

Partie C

```
def f(x,y):
   if x==0 and y==0:
      return(0)
   elif y==0:
      return 1 + f(0,x-1)
   else:
       return 1+f(x+1,y-1)
```