INFORMATIQUE MP DEVOIR 1

20 OCTOBRE 2017

Le soin et la clarté, ainsi que le respect de la syntaxe en Python sont partie intégrante de la notation finale.

EXERCICE 1 - TRIS ET ÉTUDES

Dans le but dé réaliser des études statistiques, on souhaite se doter d'une fonction de tri. On se donne la fonction **tri** suivante :

```
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. Détailler au fur et a mesure du passage en boucle l'état de la liste `L=[5,2,3,1,4]` lors de l'appel `tri(L)`.
2. 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`.
3. Proposer une fonction `tri_fusion(L)` qui applique le principe du tri fusion à une liste `L`. On pourra créer des fonctions auxiliaires.
4. Montrer que le tri fusion est plus rapide que le tri proposé dans l'énoncé.

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` inspiré de la fonction `tri` réalisant cette opération.

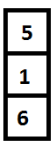
On considère toujours une liste constituée de couples (chaîne, entier). On souhaite cette fois mettre en place un tri par sélection. Le principe est simple : on déterminer le couple dont l'entier est le plus petit, et on échange sa place dans la liste avec l'élément en début de lisye. On recommence l'opération avec le second élément plus petit que l'on place en seconde place de la liste et ainsi de suite.

6. Écrire une fonction python `max_par_chaine(L,i)` qui renvoie la position du couple dans la liste `L` à partir de la position i dont l'entier correspondant est le minimum.
7. En déduire une fonction python `tri_selection_chaine` qui renvoie tri cette liste par sélection.

EXERCICE 2 - PILES

On souhaite réaliser une structure de piles de taille borné en Python.
Pour simuler une pile capacité `N` on considère un tableau `p = [n,a1,a2,a3,...,an,....]` de taille $N+1$ tel que :
- N soit la taille maximale de la pile
- n, la valeur de $p[0]$, soit le nombre d'éléments dans la pile.
- $a1, a2, a3, ..., an$ sont les n éléments de la pile
- l'espace restant est considéré comme vide dans la pile, quelque soit les valeurs du tableau.

Par exemple `p=[3,6,1,5,2,4,0]` représente la pile :
le sommet est en haut...



On considère que les fonctions suivantes sont déjà saisies dans Python :

```
def creer_pile(N):
    p = (N + 1) * [None]
    p[0] = 0
    return(p)
```

```
def taile(p):
    return(p[0])
```

1. Ecrire en Python les fonctions suivantes :
   — `est_vide(p)` : indique si la pile p est vide.
   — `depiler(p)` : dépile et renvoie le sommet de la pile p.
   — `empiler(p,v)` : empile la valeur v sur la pile p.

On considère que toutes ces fonctions sont maintenant écrites en Python.

2. Écrire une fonction `intervert(p)` qui échange les deux éléments en haut de la pile `p`.
3. Écrire une fonction `renverse(p)` qui renvoie une pile constituée des mêmes éléments dans l'ordre inverse.
4. On souhaite mélanger deux piles de la façon suivante : on créer une nouvelle pile et au fur et à mesure on choisit un élément au hasard au sommet d'un des deux piles, et on l'empile sur une cette nouvelle pile. Créer une fonction `melange(p,q)` qui réalise cela.

EXERCICE 3 - AUTOUR DE LA RÉCURSIVITÉ

Les parties sont indépendantes.

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?

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)` puis de l'appel `puissance_rapide(2,8)`.
   (b) Déterminer la complexité de cet algorithme.

Partie B

On considère une suite (u_n) définie par récurrence par : $u_{n+1} = 3u_n^2 - u_n + 5$ et $u_0 = 2$.
On considère la fonction récursive :

```
def u(n):
   assert n>=0
   if n == 0 :
      return 1
   else :
      return 3*(u(n-1))**2-u(n-1)+5
```

1. Détailler le déroulement d'un appel à `u(2)`.
2. Démontrer qu'un appel à `u(n)` termine et renvoie bien la valeur de u_n.
3. Déterminer le nombre $C(n)$ de multiplications effectuées lors d'un appel à `u(n)`.
4. Proposer une amélioration de cet algorithme au niveau de la complexité et démontrer que votre nouvelle version est plus rapide.