TP noté : corrigé.

PTSI Lycée Eiffel

18 décembre 2014

I. Programmation de quelques algorithmes élémentaires sur les listes.

Question 1 : Écrire une fonction Python **appartient(l,a)** déterminant si un élément a apparait dans une liste l. La valeur retournée devra être un booléen (True ou False), mais si vous n'arrivez qu'à faire une version retournant un message, donnez-là quand même.

```
def appartient(l,a) :
    for i in l :
        if i==a :
            return True
    return False
```

On rappelle que, dans une définition de fonction, l'exécution s'arrête dès qu'une instruction return est rencontrée. Dans le programme ci-dessus, l'instruction de la dernière ligne ne sera donc exécutée que si on atteint la fin de la boucle, c'est-à-dire si on n'a pas croisé l'élément a en parcourant la liste.

Question 2 : Écrire une fonction Python **occurrences(l,a)** déterminant le nombre de fois où l'élément a apparait dans la liste l (la fonction renvoit 0 si l'élément n'est pas dans la liste).

```
def occurrences(l,a) :
    n=0
    for i in l :
        if i==a :
            n=n+1
    return n
```

Question 3 : Écrire une fonction Python **plusgrands(l,m)** déterminant tous les éléments de l qui sont supérieurs ou égaux à m. La fonction renverra donc une liste, qui sera vide si tous les éléments de l sont strictement inférieurs à m.

```
def plusgrands(l,m) :
    grands=[]
    for i in l :
        if i>=m :
            z.append(i)
    return z
```

Dans ce programme, la liste z est constituée de tous les éléments de la liste initiale plus grands que m.

Question 4 : Écrire une fonction Python **indices(l,a)** déterminant les indices de la liste où apparait la valeur a. La fonction renverra une liste, vide dans le cas où a n'apparait jamais dans la liste.

```
def indices(l,a) :
    z=[]
    for i in range(len(l)) :
        if l[i]==a :
            z.append(i)
    return z
```

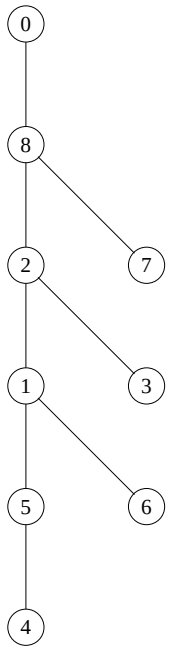
II. Présentation du problème : modélisation d'arbres généalogiques.

Question 5 (sur papier) : Compléter le tableau précédent et donner la liste **arbre** complète.

Membre de l'arbre	0	1	2	3	4	5	6	7	8
Père du membre	-1	8	1	0	1	8	8	5	0

La liste correspondante en Python sera **arbre=[-1,8,1,0,1,8,8,5,0]**.

Question 6 (sur papier) : Tracer l'arbre généalogique correspondant à la liste arbre2=[-1,2,8,2,5,1,1,8,0]. Combien de générations sont présentes sur cet arbre ?



Cet arbre contient cinq générations.

III. Programmation de quelques fonctionnalités simples sur les arbres généalogiques.

Question 7 : Écrire une fonction Python **pere(l,a)** déterminant le père de l'élément a dans la liste l (qui sera supposée représenter un arbre généalogique).

```
def pere(l,a) :
    return l[a]
```

Question 8 : Écrire une fonction Python **enfants(l,a)** déterminant la liste des enfants de l'élément a dans la liste l.

La fonction enfants est exactement la même que la fonction indices programmée dans la première partie du TP. Pour la suite de la correction, nous l'appelerons enfants et plus indices.

Question 9 : Écrire une fonction Python **generation(l,a)** déterminant à quelle génération appartient l'élément a dans la liste l

```
def generation(l,a) :
    g=0 ; e=a
    while e !=0 :
        g=g+1 ; e=l[e]
    return g
```

On parcourt l'arbre « vers le haut » en partant de l'élément a, en comptant le nombre de déplacements effectués.

Question 10 : Écrire une fonction Python **nbredescendants(l,a)** déterminant le nombre de descendants de l'élément a dans la liste l (on pourra commencer par écrire une fonction descendant(l,i,j) déterminant si j est un descendant de i dans la liste l).

```
def descendant(l,i,j) :
    e=j
    while e !=-1 :
        if e==i :
            return True
        e=l[e]
    return False

def nbredescendants(l,a) :
    z=[descendant(l,a,i) for i in range(len(l))]
    return occurrences(z,True)
```

Le programme descendant « remonte » dans l'arbre généalogique en partant de l'élément j jusqu'à être arrivé tout en haut de l'arbre, et s'arrête en renvoyant la valeur True s'il croise i en cours de route. Sinon, il ressortira False en fin de boucle. Le deuxième programme se contente de vérifier pour chaque élément de la liste s'il est ou non un descendant de i et compte le nombre de descendants trouvés.

Question 11 : Écrire une fonction Python **nbregenerations(l)** déterminant le nombre de générations présentes dans l'arbre représenté par la liste l.

```
def nbregenerations(l) :
    z=[generation(l,a) for a in range(len(l))]
    n=0
    for i in z :
        if i>n :
            n=i
    return n
```

C'est une classique recherche d'élément maximal dans une liste, ici celle constituée des générations de tous les éléments de la liste initiale.

IV. Des fonctionnalités plus avancées.

Question 12 : Écrire une fonction Python **plusprocheancetre(l,a,b)** déterminant le plus proche ancêtre commun de a et b dans la liste l (on pourra commencer par traiter le cas où a et b sont de la même génération).

```
def plusprocheancetre(l,a,b) :
    e=a
    while descendant(l,e,b)==False :
        e=l[e]
    return e
```

On se contente ici de remonter l'arbre à partir d'un des deux éléments donnés, jusqu'à tomber sur un élément dont l'autre élément donné est un descendant. On est alors sur l'ancêtre commun (je vous laisse y réfléchir). Si on préfère tenter d'exploiter l'indice de l'énoncé : pour deux éléments de même génération, on remonte dans l'arbre des deux côtés jusqu'a tomber sur le même élément, qui est alors l'ancêtre commun. Si a est de génération n et b de génération p, on fait d'abord remonter l'élément de génération maximale de $|n-p|$ générations pour le mettre au niveau de l'autre, puis on applique l'algorithme précédent.

Question 13 : Écrire une fonction Python **distance(l,a,b)** déterminant la distance entre a et b dans la liste l.

```
def distance(l,a,b) :
    return generation(l,a)+generation(l,b)-2*generation(l,plusprocheancetre(l,a,b))
```

Autant réutiliser les programmes précédents ! La distance entre un ancêtre et son descendant est simplement l'écart entre leurs générations.

Question bonus : Écrire une fonction Python **genealogique(l)** qui détermine si une liste d'entiers en Python (quelconque) est une liste représentant un arbre généalogique ou non.

Il existe en fait une condition très simple pour que la liste corresponde à un arbre généalogique : tout le monde doit être descendant de 0 (si ce n'est pas le cas, il y a forcément des cycles dans l'arbre, puisque chaque élément a toujours un père). Attention toutefois, on ne peut pas utiliser la fonctions descendants écrite plus haut pour le tester car celle-ci va justement ne jamais s'arrêter

si on ne remonte jamais jusqu'à la racine de l'arbre en partant de l'élément à tester. On va donc remonter à partir de l'élément i jusqu'à tomber sur -1 (on est contents) ou sur i lui-même (on n'est pas contents).

```
def genealogique(l) :
    if l[0] !=-1 :
        return False
    for i in range(len(l)) :
        e=l[i]
        while (e !=-1) and (e !=i) :
            e=l[e]
        if e==i :
            return False
    return True
```