Correction du DS 1

Informatique pour tous, deuxième année

Julien Reichert

Exercice 1

Voir le cours et le TP 1, les deux implémentations, naturelle et à l'aide de tableaux, étaient acceptées. Toute autre éventuelle implémentation cohérente pouvait également être acceptée.

Exercice 2

Naïvement, il suffit de tester la taille et d'agir en conséquence. Trier une liste de taille si petite est excessif, en pratique. Vérifier que la taille est entre un et trois n'est pas nécessaire car la spécification exclut les autres tailles.

```
def moyenne(l):
    assert 1 <= len(l) <= 3, "Taille incorrecte"
    if len(l) == 1:
        return l[0]
    if len(l) == 2:
        return .6 * max(l) + .4 * min(l)
    return .6 * max(l) + .1 * min(l) + .3 * (sum(l)-max(l)-min(l)) # Autres formules possibles
```

On peut même retirer les tests sur la taille en la mettant artificiellement à 3. Pour retirer toute utilisation d'un `if`, on réécrit même quelques fonctions...

```
def abs(nombre):
    signe = str(nombre)[0] # soit '-' (ord donne 45) soit un caractère chiffre (ord donne 48 à 57)
    return nombre*(ord(signe) > 46)-nombre*(ord(signe) < 46)
# Évite de transformer un entier en flottant avec la racine du carré

def min(l):
    mini = l[0]
    for i in range(1,len(l)):
        mini = (mini + l[i] - abs(mini - l[i]))//2
    return mini

def max(l):
    return -min([-x for x in l])

def moyennebis(l):
    1 / len(l) + 1 / (4-min([4,len(l)])) # Erreur si len(l) est 0 ou > 3
    for i in range (3-len(l)):
        l.append(min(l)) # On peut constater que c'est équivalent
    return .6 * max(l) + .1 * min(l) + .3 * (sum(l)-max(l)-min(l)) # Autres formules possibles
```

Exercice 3

Il s'agit d'une extension d'un exercice du TP 3. On a ici besoin de connaître le contenu des piles. Une solution intuitive revient à mettre en argument les piles. À ce stade, il est plus simple de considérer celles-ci comme des variables globales (favoriser le langage par rapport au paradigme de programmation), et donc la fonction ne sera pas elle-même récursive mais appellera une sous-fonction récursive après une préparation des variables.

```
def hanoi_explicite(n):
    piles = [list(range(n,0,-1)), [], []]
    print ("Départ :",piles[0],piles[1],piles[2])
    def hanoi_aux(i,origine,arrivee):
        if i > 0:
         transfert = 3-origine-arrivee
         hanoi_aux(i-1,origine,transfert)
         print(origine+1,"->",arrivee+1)
         piles[arrivee].append(piles[origine].pop())
         print ("Piles :",piles[0],piles[1],piles[2])
         hanoi_aux(i-1,transfert,arrivee)
    hanoi_aux(n,0,2)
```

Exercice 4

D'après la formule du cours, on a $c_n = ac_{\frac{n}{b}} + \mathcal{O}(n^\alpha)$ avec $\log_b(a) = \alpha = 0$. Ainsi, $c_n = \mathcal{O}(n^0 \log n)$ soit une complexité logarithmique. C'est le cas d'une dichotomie simple.

Exercice 5

C'est la combinaison de deux exercices du TP 2 :

```
def multiparenthesage(L):
    parentheses = []
    couples = []
    for i in range(1,len(L)+1):
        if L[-i] < 0:
            parentheses.append((L[-i],len(L)-i))
        elif L[-i] > 0:
            try:
                (type,position) = parentheses.pop()
                assert(type == -L[-i])
                couples.append((len(L)-i,position))
            except:
                raise ValueError("Mauvais parenthesage")
    if parentheses == []:
        return list(reversed(couples))
    else:
        raise ValueError("Une parenthese fermante au moins est de trop")
```

On notera que la lecture de la liste se fait à l'envers. La raison est qu'alors les couples seront ordonnés par position croissante des parenthèses ouvrantes.