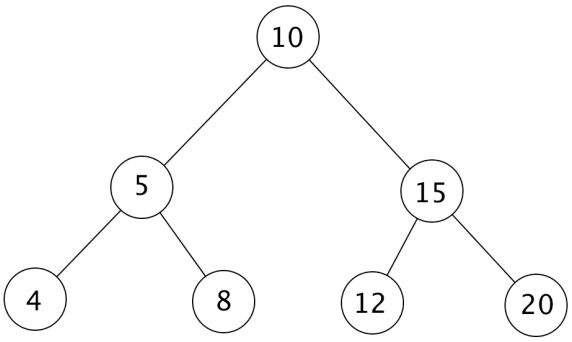
Épreuves écrites de NSI
Correction du sujet
Métropole 2 2021
Journée 2

Exercice 3

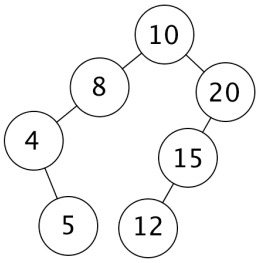
1. a. Sa taille est de 7.

b. Sa hauteur est de 4

2. On peut proposer l'arbre suivant :



3. On obtient l'arbre suivant :



4. On peut écrire le code suivant :

```
def hauteur(self) :
    self.racine.hauteur()
```

5. On peut écrire les codes suivants :

• pour la classe `Noeud`, en inspirant du code de la méthode `hauteur` :

```
def taille(self) :
    if self.gauche == None and self.droit == None:
        return 1
    if self.gauche == None:
        return 1+self.droit.taille()
    elif self.droit == None:
        return 1+self.gauche.taille()
    else:
        tg = self.gauche.taille()
        td = self.droit.taille()
        return tg + td + 1
```

• pour la classe `Arbre` :

```
def taille(self) :
    self.racine.taille()
```

6. a. La taille maximale d'un arbre binaire de hauteur `h` est $2^h - 1$. En utilisant la définition de « bien construit » donnée dans l'énoncé, on en déduit que **l**a taille minimale d'un arbre binaire de recherche bien construit de hauteur `h` est de 2^{h-1}.

b. On admet qu'un arbre binaire de recherche de hauteur `h` est bien construit si et seulement si sa taille est supérieure ou égale à 2^{h-1}.
On en déduit le code de la méthode `bien_construit` :

```
def bien_construit (self) :
    t = self.taille()
    h = self.hauteur()
    return t >= 2**(h-1)
```

Exercice 4

1. Dans le code de la fonction `echange`, l'affection `lst[i2] = lst[i1]` remplace la valeur du tableau `lst` à l'indice `i2` par la valeur à l'indice `i1`. L'affectation suivante `lst[i1] = lst[i2]` ne change donc pas la valeur à l'indice `i1`. Pour effectuer l'échange des valeurs, il faut :

* soit utiliser une variable intermédiaire de la façon suivante :

```
def echange (lst, i1, i2) :
    temp = lst[i2]
    lst[i2] = lst[i1]
    lst[i1] = temp
```

* soit utiliser l'affectation multiple en Python de la façon suivante :

```
def echange (lst, i1, i2) :
    lst[i1], lst[i2] = lst[i2], lst[i1]
```

2. L'appel `randint(0, 10)` peut renvoyer les valeurs `1`, `9` ou `10`.

3. a. On suppose que la valeur de l'argument `ind` lors d'un appel de la fonction `melange` est toujours un entier positif. La fonction `melange` se termine toujours car, à chaque appel récursif de la fonction, la valeur du deuxième argument `ind` est décrémentée de 1. Il est donc nécessaire que cette argument devienne égal à 0 en un nombre fini d'étapes. Or, dans le cas où l'argument `ind` est nul, il n'y a pas d'appel récursif.

b. Le plus grand indice possible de la liste `lst` est `n-1`. Au premier rappel récursif, l'argument `ind` sera donc égal à `n-2` et au dernier récursif il sera égal à `0`. Il y aura donc `n-1` appels récursifs.

c. On obtient les affichages suivants :

```
...
[0, 3, 4, 1, 2]
[0, 3, 4, 1, 2]
[3, 0, 4, 1, 2]
```

d. On peut proposer la version itérative suivante (si on y inclue l'impression de la liste à chaque étape) :

```
def melange_iter(lst) :
```

```
    lg = len(lst)
    for i in range(1, lg) :
        print(lst)
        j = randint(0, i)
        echange(lst, i, j)
```

Exercice 5

1. a. Si tous les éléments sont positifs la solution est la somme de tous les éléments du tableau.

b. Si tous les éléments sont négatifs, la solution est la plus grand valeur du tableau.

2. a. On peut écrire le code suivant :

```
def somme_sous_sequence(lst, i, j) :
    s = 0
    for k in range(i, j+1) :
        s += lst[k]
    return s
```

b. Dans le cas d'un tableau de 10 valeurs, lorsque `i` est égal à 0, `j` prend 10 valeurs dans la boucle interne et il y a alors 10 comparaisons pour cette valeur de i. A chaque fois que `i` est incrémentée de 1, le nombre de valeurs prises par `j` est décrémentée de 1. le nombre de comparaison au total est donc

$$10 + 9 + \cdots + 1 = \frac{10 \times 11}{2} = 55$$

c. On peut écrire le code suivant :

```
def pgsp(lst):
    n = len(lst)
    somme_max = lst[0]
    deb = 0
    fin = 0
    for i in range(n):
        for j in range(i, n):
            s = somme_sous_sequence(lst, i, j)
            if s > somme_max :
                somme_max = s
                deb = i
                fin = j
    return (somme_max, deb, fin)
```

3. a. On obtient le tableau suivant :

i	0	1	2	3	4	5	6	7
lst[i]	-8	-4	6	8	-6	10	-4	-4
S(i)	$\mathbf{-8}$	$\mathbf{-4}$	**6**	**14**	**8**	**18**	**14**	**10**

b. En appliquant la remarque du 3., on obtient le code suivant :

```
def pgsp2(lst):
    sommes_max = [lst[0]]
    for i in range(1, len(lst)):
        if sommes_max[i-1]<=0 :
            sommes_max.append(lst[i])
```

```
        else :
            sommes_max.append(lst[i]+sommes_max[i-1])
    return max(sommes_max)
```

c. On admet que l'algorithme de la fonction `max` est, au pire, de complexité linéaire. La fonction `pgsp2` est alors elle aussi de complexité linéaire. Cette deuxième version est bien plus efficace que la première version qui était quant à elle de complexité quadratique.