Algorithmique
Correction Contrôle n° 2 (C2)

Info-sup S2 – Epita

2 mars 2020 - 10 : 00

Solution 1 **(Un peu de cours... –** ***4 points*****)**

L'arbre général **A** étant le suivant :

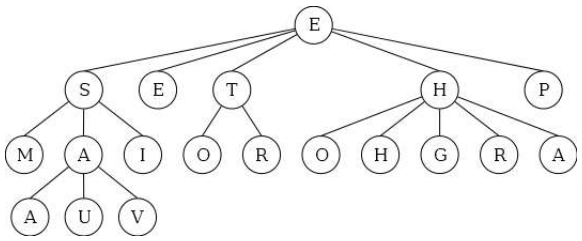

Figure 1 – Arbre général **A**

1. La taille de l'arbre `A` est : `19`
2. La hauteur de l'arbre `A` est : `3`
3. La longueur de cheminement interne de l'arbre `A` est : `5`
4. La profondeur moyenne externe de l'arbre `A` est : `29/14` $\simeq$ `2.07`
5. La liste des sommets de l'arbre `A` rencontré en ordre suffixe est :
   `{M, A, U, V, A, I, S, E, O, R, T, O, H, G, R, A, H, P, E}`
6. La liste des nœuds de l'arbre `A` rencontré en ordre hiérarchique est :
   `{E, S, E, T, H, P, M, A, I, O, R, O, H, G, R, A, A, U, V}`

Solution 2 **(Carré magique –** ***4 points*****)**

Spécifications :

La fonction `Siamese(`n`)` construit et retourne un carré magique d'ordre n (n est un entier supérieur à 2 impair).

```
1        def Siamese(n):
2            """
3            n natural, odd
4            """
5            S = matrix.init(n, n, 0)
6            (i, j) = (n - 1, n // 2)
7            for val in range(1, n*n + 1):
8                S[i][j] = val
9                if val % n == 0:
10                   i = i - 1
11               if i == -1:
12                   i = n-1
13           else:
14               (i, j) = ((i + 1) % n, (j + 1) % n)
15           return S
```

Solution 3 **(Sous-liste – *5 points*)**

Spécifications :

La fonction `sub_line(`M`, `L`)` vérifie si la liste L est incluse dans une des lignes de la matrice M (supposée non vide).

```
1   def sub_line(M, L):
2       (lineM, colM) = (len(M), len(M[0]))
3       n = len(L)
4       if n > colM:
5           return False
6       else:
7           i = 0
8           ok = False
9           while i < lineM and not ok:
10              j = 0
11              while j < colM - n and not ok:
12                  k = 0
13                  while k < n and L[k] == M[i][j+k]:
14                      k += 1
15                  ok = (k == n)
16                  j += 1
17              i += 1
18          return ok
19
20  # two functions
21
22  def equalList(LM, L, start) :
23      (i, n) = (0, len(L))
24      while i < n and LM[start+i] == L[i] :
25          i += 1
26      return i == n
27
28  def sub_line2(M, L) :
29      (lb, cb, n) = (len(M), len(M[0]), len(L))
30      if n > cb :
31          return False
32      else:
33          i = 0
34          j = (cb - n) + 1
35          while i < lb and j > (cb-n) :
36              j = 0
37              while j <= (cb-n) and not equalList(M[i], L, j):
38                  j += 1
39              if j > (cb-n) :
40                  i += 1
41          return i < lb
```

***Solution 4* (Arbre partiellement ordonné – *3 points*)**

Spécifications :

La fonction `priority(B)` vérifie si l'arbre binaire B (dont les clés sont des entiers strictement positifs) est partiellement ordonné.

```
    def __test(B, p):
        """
        p: B's parent
        """
        if B == None:
            return True
        else:
            if B.key < p:
                return False
            else:
                return __test(B.left, B.key) and __test(B.right, B.key)

    def priority(B):
        return __test(B, 0)



    def priority2(B, p=0):
        """
        p: B's parent
        """
        if B == None:
            return True
        else:
            if B.key < p:
                return False
            else:
                return priority2(B.left, B.key) and priority2(B.right, B.key)
```

Version sans passer la clé du père en paramètre :

```
    def __priority3(B):
        """
        B not empty
        """
        test = True
        if B.left != None:
            if B.key > B.left.key:
                test = False
            else:
                test = __priority3(B.left)
        if test and B.right != None:
            if B.key > B.right.key:
                test = False
            else:
                test = __priority3(B.right)
        return test

    def priority3(B):
        return B == None or __priority3(B)
```

Solution 5 **(Largeur –** ***4 points*****)**

Spécifications :

La fonction `width(`B`)` calcule la largeur de l'arbre binaire B.

```
    # with level change marks (None)

    def width(B):
        w_max = 0
        if B:
            q = queue.Queue()
            q.enqueue(B)
            q.enqueue(None)
            w = 0
            while not q.isempty():
                B = q.dequeue()
                if B == None:
                    w_max = max(w, w_max)
                    if not q.isempty():
                        q.enqueue(None)
                        w = 0
                else:
                    w = w + 1
                    if B.left:
                        q.enqueue(B.left)
                    if B.right:
                        q.enqueue(B.right)
        return w_max


    # another way to manage levels, with two queues.

    def width2(B):
        w_max = 0
        if B != None:
            q = queue.Queue() #current
            q.enqueue(B)
            q_next = queue.Queue() #next level
            w = 0
            while not q.isempty():
                B = q.dequeue()
                w = w + 1
                if B.left != None:
                    q_next.enqueue(B.left)
                if B.right != None:
                    q_next.enqueue(B.right)
                if q.isempty():
                    w_max = max(w, w_max)
                    (q, q_next) = (q_next, q)
                    w = 0
        return w_max
```