Algorithmes et Pensée Computationnelle

Consolidation 2

Les exercices de cette série sont une compilation d'exercices semblables à ceux vus lors des semaines précédentes. Le but de cette séance est de consolider les connaissances acquises lors des travaux pratiques des dernière semaines.

Question 1: (*10 minutes*) **Complexité**
Analysez la complexité pour les deux codes suivants. Est-ce la même pour fun() et fun2() ? Pourquoi ?

```
def fun(n):
  for i in range(n):
    for j in range(n):
      print (n)

def fun2(n):
  for i in range(n):
    print (n)
  for j in range(n):
    print (n)
```

Solution

Non, ils n'ont pas la même complextié. Fun() a une complexité de O(n*n) car il s'agit de deux boucles imbriquées. En revanche, fun2() a une complexité de O(n+n) car il s'agit de deux boucles indépendante.

Question 2: (*10 minutes*) **Complexité**
Quel est la complexité de ce code ?

```
def fun(k):
  if k == 1:
    print ('Done')
  else:
    k = k/2
fun(k)
```

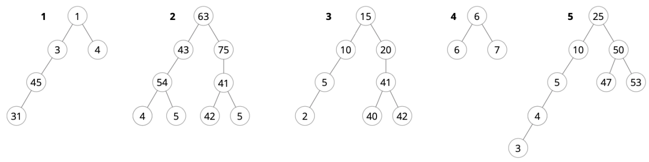
Solution

Question 3: (*10 minutes*)
Ecrivez un programme python qui permet d'imprimer tous les nombres impairs à partir de 1 jusqu'à un nombre n défini par l'utilisateur (qui doit être supérieur à 1) : Exemple : si n = 6, résultat attendu : 1, 3, 5

Solution

```
def displayOddNumbers(limit):
  for nb in range(limit+1):
    if nb % 2 == 1:
      print(nb)

limit = int(input("What is the limit of the function displayOddNumbers(limit)? "))

print("Odd numbers between 0 and " + str(limit) + ": ")
displayOddNumbers(limit)
```

Question 4: ( ) Lesquels (ou lequel) de ces arbres est un arbre binaire (binary tree) ? Donnez leur hauteur (height).



>_ Solution

1. Il ne s'agit pas d'un arbre binaire car il n'y a pas decondition qui est remplie à chaque noeuds. Sa hauteur est de 3.
2. Il ne s'agit pas d'un arbre binaire car il n'y a pas de condition qui est remplie à chaque noeuds. Sa hauteur est de 3.
3. Il s'agit d'un arbre binaire car une condition peut être appliquée à chaque noeud. Sa hauteur est de 3.
4. Il s'agit d'un arbre binaire car une condition peut être appliquée à chaque noeud. Sa hauteur est de 1.
5. Il s'agit d'un arbre binaire car une condition peut être appliquée à chaque noeud. Sa hauteur est de 4.

Question 5: (🕒 *10 minutes*) **Trie complexité**
Trier la liste de fonctions suivante selon leur croissance assymptotique :

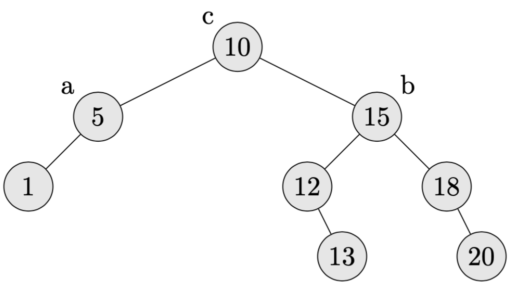
$$n^{\sqrt{n}}, n \cdot log(n), n^{1/log(n)}, log(log(n)), \sqrt{n}, 3^n/n^5, 2^n \tag{1}$$

>_ Solution

— D'abord la constante : $n^{1/log(n)} = (2^{log(n)})^{1/log(n)}$
— Ensuite le $log(log(n))$
— Puis $n^{\sqrt{n}} = 2^{\sqrt{n} \cdot log_2(n)}$
— Puis 2^n
— Enfin $3^n/n^5$

Question 6: (🕒 *20 minutes*) **Théorie**

1. Donner en pseudo code un algorithme ayant une complexité temporelle de O(log n) qui prend comme argument un tableau trié A[1, ..., n] de n nombres et une clé k et qui retourne "OUI" si A contient k et "NON" sinon.
2. Quelle est la hauteur maximum et la hauteur minimale d'un arbre binaire de recherche ayant n éléments ? Quel arbre est meilleur ? Justifier.
3. Considerer l'arbre binaire suivant :
   Dessiner les arbres obtenus après executions de chacune des opérations suivantes (chaque opération

$T =$ 

est exécutée en commençant par l'arbre ci-dessus - les opérations ne sont pas exécutée séquentiellement).

(a) TREE-INSERT(T, z) avec z.key = 0
(b) TREE-INSERT(T, z) avec z.key = 17
(c) TREE-INSERT(T, z) avec z.key = 14
(d) TREE-DELETE(T, a)
(e) TREE-DELETE(T, b)
(f) TREE-DELETE(T, c)

Solution

1. Etant donné que les nombres du tableau A sont triés, nous utilisons l'algorithme de recherche binaire. L'algorithme de recherche binaire prend comme argument un tableau A, une clé k, des indices p et q et retourne "OUI" si A[p . . . q] contient la clé k et "NON" autrement. Comme A[p . . . q] est trié, nous pouvons comparer k avec l'élément du milieu mid = $\lfloor (p+q)//2 \rfloor$ et :

   — Si A[mid] = k return "OUI"

   — Si A[mid] $>$ k, alors cherchons k dans le tableau A[p . . . (mid-1)] en appelant récursivement l'algorithme BINARY-SEARCH(A, k, p, mid-1)

   — Si A[mid] $<$ k, alors cherchons k dans le tableau A[(mid + 1) . . . q] en appelant récursivement l'algorithme BINARY-SEARCH(A, k, mid+1, q)

   The pseudo-code of the procedure is as follows :

```
def binary_search(A, k, p, q):
  if (q < p):
    return "NON" # array is empty so it doesn't contain k
  else:
    mid = (p+q)//2
  if (A[mid] == k):
    return "OUI"
  elif (A[mid] > k):
    return binary_search(A, k, p, mid−1)
  else: # A[mid] < k
    return binary_search(A, k, mid+1, q)
```

   Note that we solve the original problem by calling BINARY-SEARCH(A, k, 1, n).

2. Hauteur minimale et maximale d'un arbre binaire.
   — La hauteur maximum d'un arbre binaire est atteinte quand l'arbre n'est constitué d'une seule branche.

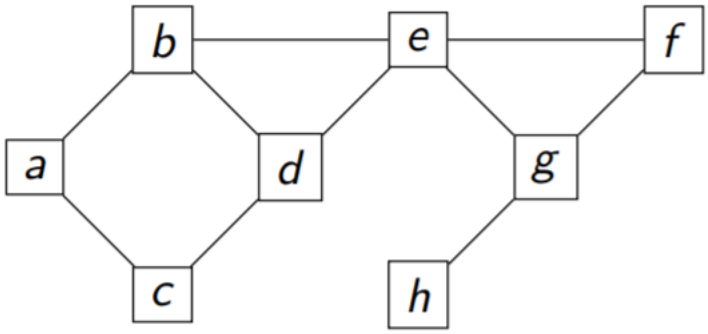
   — La hauteur minimum est atteinte lorsque l'arbre binaire est "complet" : nous ne pouvons pas ajouter un noeud sans augmenter la hauteur de l'arbre hauteur de un.

3. Après execution de chaque opération, nous obtenons :
   (a) Figure A : TREE-INSERT(T, z) avec z.key = 0
   (b) Figure B : TREE-INSERT(T, z) avec z.key = 17
   (c) Figure C : TREE-INSERT(T, z) avec z.key = 14
   (d) Figure D : TREE-DELETE(T, a)
   (e) Figure E : TREE-DELETE(T, b)
   (f) Figure F : TREE-DELETE(T, c)

Question 7: (*5 minutes*) **Breadth-First Search algorithm : Papier**

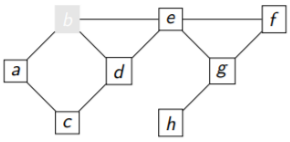
Le but du Breadth-first search (BFS) ou parcours en largeur est d'explorer le graphe à partir d'un sommet donné (sommet de départ ou sommet source).

Appliquez l'algorithme de BFS au graphe suivant :



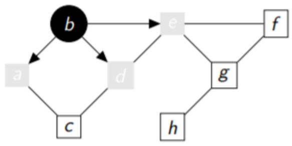
>_ Solution

a)



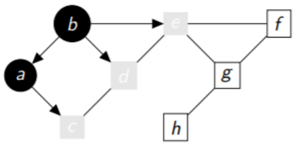
Q = [b]
0

b)



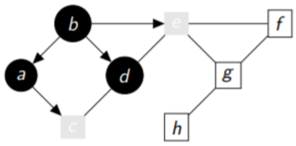
Q = [a, d, e]
1, 1, 1

c)



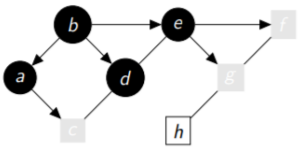
Q = [d, e, c]
1, 1, 2

d)



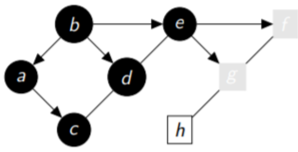
Q = [e, c]
1, 2

d)



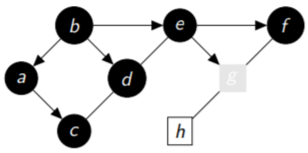
Q = [c, f, g]
2, 2, 2

e)



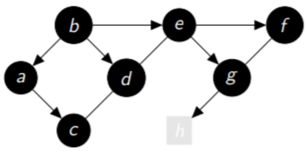
Q = [f, g]
2, 2

f)



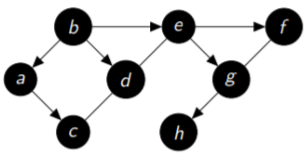
Q = [g]
2

g)



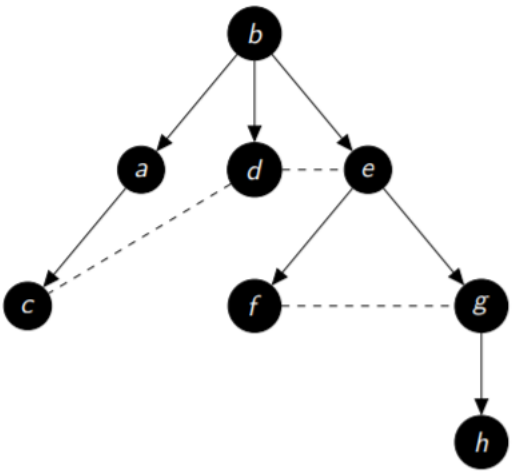
Q = [h]
3

h)



Q = Ø

Ci-dessous l’arborescence associée au parcours.



L’ordre de parcours est : ligne après ligne (de la racine vers les feuilles) et de gauche à droite pour une ligne.

Question 8: (*15 minutes*) Breadth-First Search algorithm : Python

Dans cet exercice, nous aimerions identifier le chemin le plus court entre deux nœuds d'un graphe. La fonction que nous implémentons doit pouvoir accepter comme argument un graphe, un nœud de départ et un nœud de fin. Si l'algorithme est capable de connecter les nœuds de départ et d'arrivée, il doit renvoyer le chemin. nous allons utiliser l'algorithme BFS car il renvoie toujours le chemin le plus court.

```python
# # trouve le chemin le plus court entre 2 noeuds d'un graphe en utilisant BFS
def bfs_shortest_path(graph, start, goal):
    #TODO


if __name__ == '__main__':
  # sample graph implemented as a dictionary
  graph = {'A': ['B', 'C', 'E'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F', 'G'],
        'D': ['B'],
        'E': ['A', 'B', 'D'],
        'F': ['C'],
        'G': ['C']}
  print(bfs_shortest_path(graph, 'G', 'D'))  # prints  ['G', 'C', 'A', 'B', 'D']
```

Conseil

Il existe quelques différences principales entre l'implémentation e de BFS pour l'application du plus court chemin.

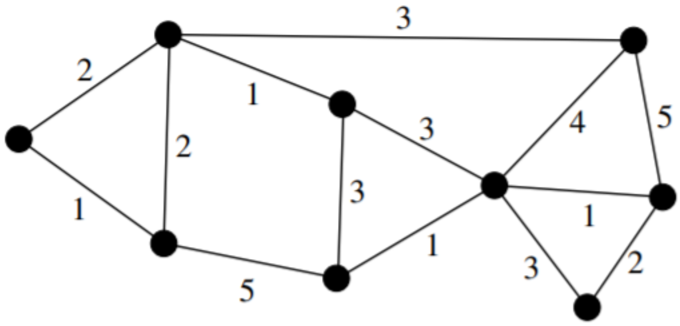
1. La file d'attente garde une trace des chemins possibles (implémentés sous forme de liste de nœuds) au lieu des nœuds.
2. lorsque l'algorithme recherche un nœud voisin, il doit vérifier si le nœud voisin correspond au nœud cible. Si c'est le cas, nous avons une solution et il n'est pas nécessaire de continuer à explorer le graphique.

Solution

```python
# # trouve le chemin le plus court entre 2 noeuds d'un graphe en utilisant BFS
def bfs_shortest_path(graph, start, goal):
    # garder une trace des noeuds visites
    visited = []
    # garder une trace de tous les chemins à vérifier
    queue = [[start]]

    # retourner le chemin si le noeud de départ est celui d'arrivée
    if start == goal:
        return ”start = goal”

    # continue de boucler jusqu'à ce que tous les chemins possibles aient été vérifiés
    while len(queue)>0:
        # extraire le premier chemin de la file d'attente
        path = queue.pop(0)
        # récupérer le dernier noeud du chemin
        node = path[-1]
        if node not in visited:
            neighbours = graph[node]
            # parcourir tous les noeuds voisins, construire un nouveau chemin et
            # le mettre dans la file d'attente
            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
                # retourner le path si le voisin est le noeud d'arrivée
                if neighbour == goal:
                    return new_path

            # marquer le noeud comme visité
            visited.append(node)

    # au cas où il n'y aurait pas de chemin entre les 2 noeuds
    return ”il n'y a pas de chemin reliant les deux noeuds”




if __name__ == '__main__':
    # sample graph implemented as a dictionary
    graph = {'A': ['B', 'C', 'E'],
             'B': ['A', 'D', 'E'],
             'C': ['A', 'F', 'G'],
             'D': ['B'],
             'E': ['A', 'B', 'D'],
             'F': ['C'],
             'G': ['C']}
    print(bfs_shortest_path(graph, 'G', 'D'))  # returns ['G', 'C', 'A', 'B', 'D']
```

Question 9: (🕓 *5 minutes*) Algorithme de Kruskal : Papier

Appliquez l'algorithme de Kruskal au graphe suivant :



Conseil

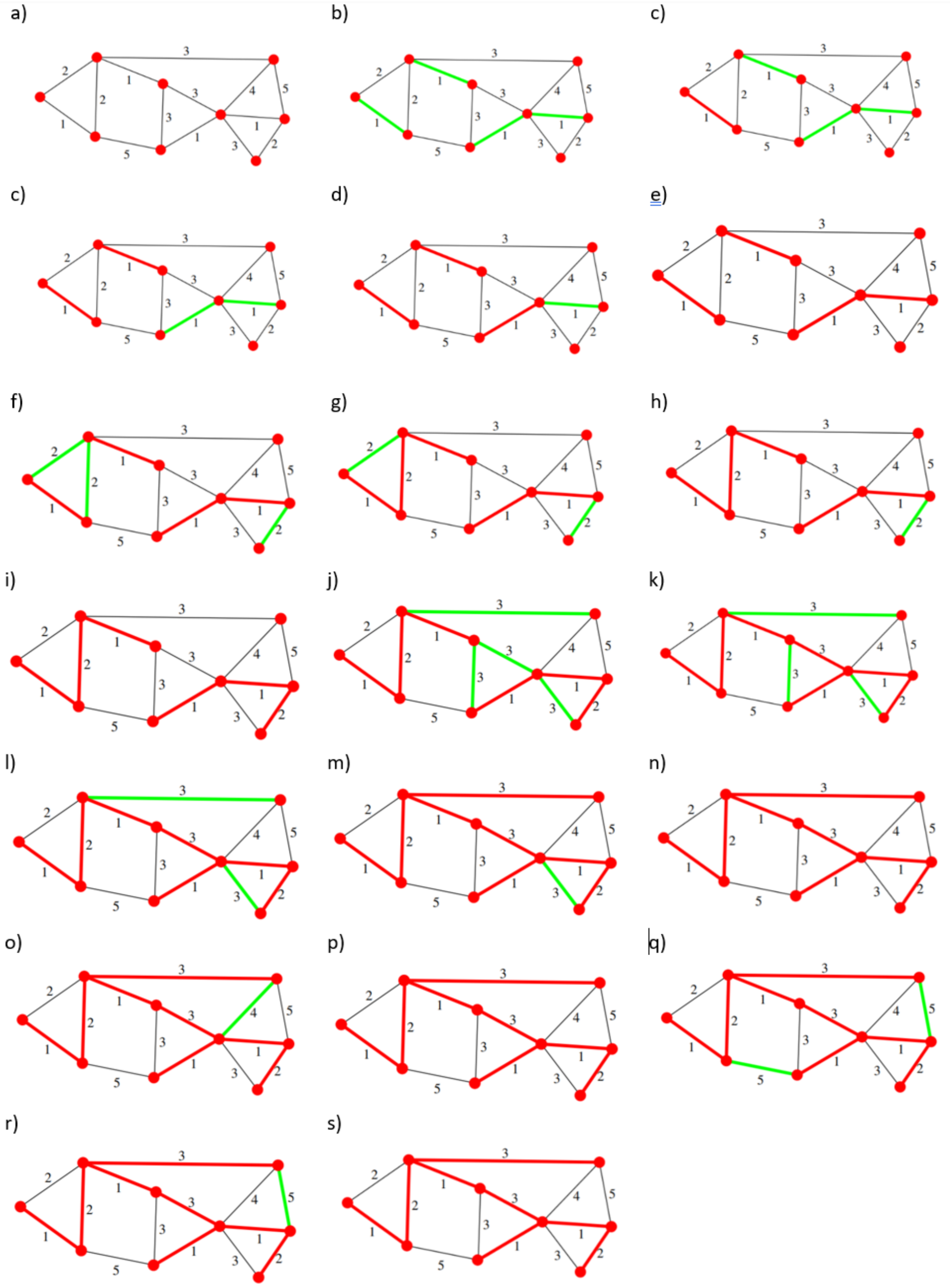
L'algorithme de Kruskal fonctionne de la façon suivante :

1. Classer les arêtes par ordre croissant de poids.
2. Prendre l'arête avec le poids le plus faible et l'ajouter à l'arbre (si 2 arêtes ont le même poids, choisir arbitrairement une des 2).
3. Vérifiez que l'arête ajoutée ne crée pas de cycle, si c'est le cas, supprimez la.
4. Répétez les étapes 2) et 3) jusqu'à ce que tous les sommets aient été atteints.

Un Minimum Spanning Tree, s'il existe, a toujours un nombre d'arêtes égal au nombre de sommets moins un. Par exemple, ici notre graphe a 9 sommets. L'algorithme devrait donc s'arrêter lorsque 8 arêtes ont été choisies.

>_ Solution

Vous trouverez ci-dessous les étapes de la construction du MST(Minimum spanning tree) avec l'algorithme de Kruskal :



L'algorithme s'arrête car tous les sommets ont été atteints. On voit bien que seules 8 arêtes ont été nécessaires.