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 des deux programmes ci-dessous. Ont-ils la même complexité ?

```
1  def fun(n):
2      for i in range(n):
3          for j in range(n):
4              print (n)
5
6  def fun2(n):
7      for i in range(n):
8          print (n)
9      for j in range(n):
10         print (n)
```

Solution

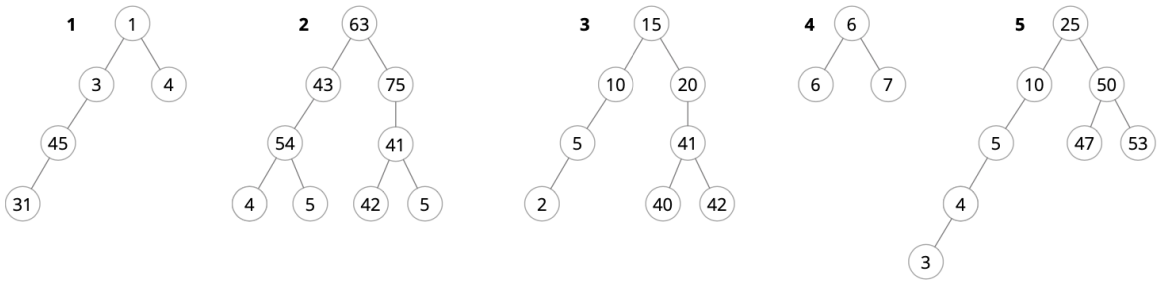
Non, ils n'ont pas la même complexité. **fun()** a une complexité de $O(n * n)$ car il s'agit de deux boucles imbriquées itérant chacune n fois dans le pire des cas. En revanche, **fun2()** a une complexité de $O(n + n)$ car elle contient deux boucles effectuant des opérations indépendantes.

Question 2: (*10 minutes*) **Programmation de base**
Ecrivez un programme Python qui imprime tous les nombres impairs à partir de 1 jusqu'à un nombre **n** défini par l'utilisateur. Ce nombre **n** doit être supérieur à 1. Exemple : si n = 6, résultat attendu : 1, 3, 5

Solution

```
1  def nombresImpairs(limite):
2      for nb in range(limite+1):
3          if nb % 2 == 1:
4              print(nb)
5
6  limit = int(input("Entrez une valeur maximale: "))
7
8  print("Nombres impairs compris entre 1 et " + str(limit) + " : ")
9  nombresImpairs(limit)
```

Question 3: (*10 minutes*) **Arbres binaires**
Lesquels de ces arbres sont des arbres binaires de recherche (binary search tree) ? Donnez leur hauteur (height).



Solution

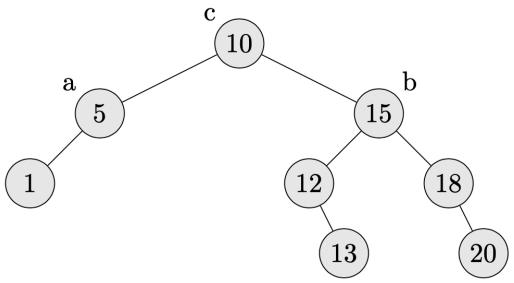
Seuls les arbres 3, 4 et 5 sont des arbres binaires de recherche selon la définition donnée en cours. En effet, l'ordre d'insertion des éléments est incorrect dans les arbres 1 et 2. Quant à leurs hauteurs, elles sont respectivement de 3, 3, 3, 1 et 4.

Question 4: (*20 minutes*) **Arbres binaires**

1. Ecrire un programme Python ayant une complexité temporelle de $O(logn)$ et qui prend comme argument un tableau trié **A[1, ..., n]** de n nombres et une clé k. Ce programme devra retourner **"OUI"** si **A** contient k et **"NON"** dans le cas contraire.
2. Quelle est la hauteur minimale et la hauteur maximale d'un arbre binaire de recherche ayant n éléments ? Exprimer votre réponse en fonction du nombre de noeuds n présents dans l'arbre.
3. Considerer l'arbre binaire suivant :

   Dessiner les arbres obtenus après execution de chacune des opérations suivantes (chaque opération est exécutée en commençant par l'arbre ci-dessus - les opérations ne sont pas exécutées de façon séquentielle).

$T =$



(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)

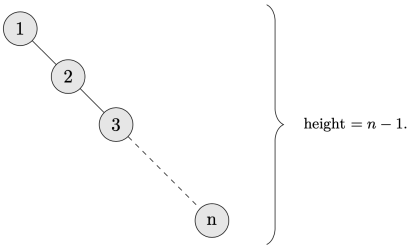
   — Si A[mid] $<$ k, alors cherchons k dans le tableau A[(mid + 1) . . . q] en appelant récursivement l'algorithme BINARY-SEARCH(A, k, mid+1, q)

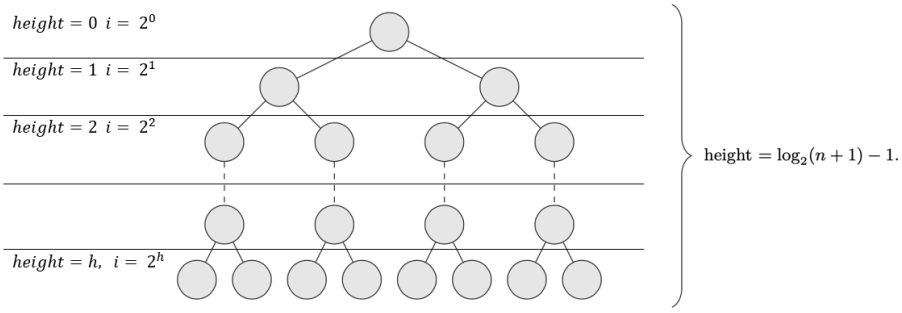
   Le programme permettant d'effectuer une recherche binaire est le suivant :

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

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

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





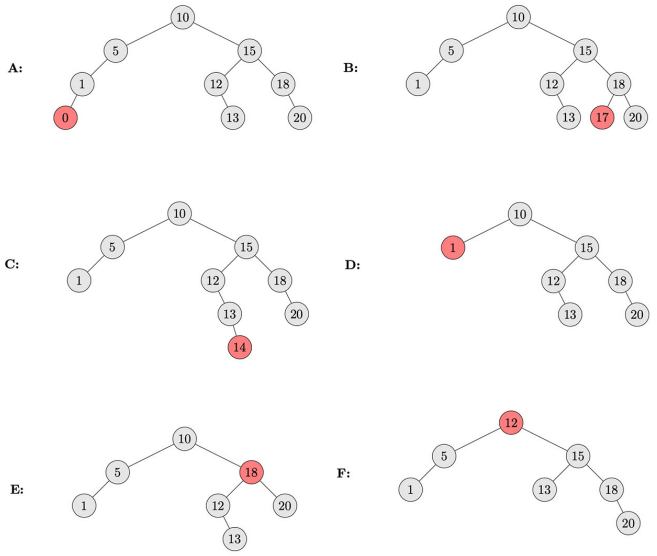
$n = \sum_{i=0}^{h} 2^i = 2^0 + 2^1 + \ldots\ldots\ldots\ldots 2^h$
$n = 2^{h+1} - 1$
$n + 1 = 2^{h+1} - 1$
$\log_2(n+1) = h + 1$
$h = \log_2(n+1) - 1$

n est le nombre de noeuds dans l'arbre,
h est la hauteur de l'arbre
i est le nombre de noeuds dans un niveau

3. Après execution de chaque opération, nous obtenons :

Solution

1. Figure A : TREE-INSERT(T, z) avec z.key = 0
2. Figure B : TREE-INSERT(T, z) avec z.key = 17
3. Figure C : TREE-INSERT(T, z) avec z.key = 14
4. Figure D : TREE-DELETE(T, a)
5. Figure E : TREE-DELETE(T, b)
6. Figure F : TREE-DELETE(T, c)



Question 5: (*10 minutes*) **Croissance de fonctions**

Nous avons vu comment exprimer la complexité d'un algorithme en terme de complexité temporelle dans le "pire des cas", notée $\mathcal{O}()$. L'exercice suivant permet de travailler avec différentes croissances de fonctions. Ordonner la liste de fonctions suivante selon leur croissance asymptotique : il faut trier les fonctions par ordre croissant de notation grand-$\mathcal{O}$. La notation grand-$\mathcal{O}()$ donne une borne supérieure du taux de croissance d'une fonction. Ainsi si $f(x) = 2x$ et $g(x) = 3x^2$, on dit que que $f(x)$ équivaut à $\mathcal{O}(x)$ et que $g(x)$ équivaut à $\mathcal{O}(x^2)$. On peut alors les ordonner pour obtenir : $\mathcal{O}(f(x)) < \mathcal{O}(g(x)) \Leftrightarrow \mathcal{O}(x) < \mathcal{O}(x^2)$.

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

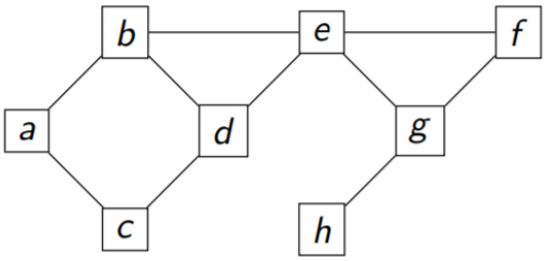
Solution

— 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: (*10 minutes*) **Breadth-First Search : Papier**

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

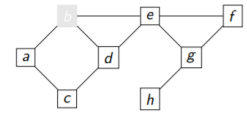
Appliquez l'algorithme **BFS** au graphe suivant en partant du sommet B :



Solution

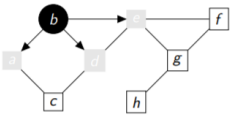
Dans cette solution, nous considèrons le sommet b comme sommet de départ et l'ordre alphabétique dans l'exploration des voisins.

a)



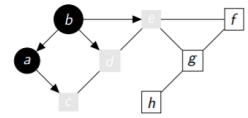
Q = [b]
0

b)



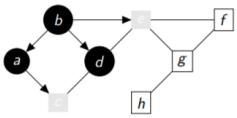
Q = [a, d, e]
1, 1, 1

c)



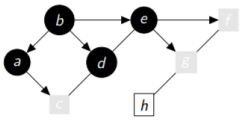
Q = [d, e, c]
1, 1, 2

d)



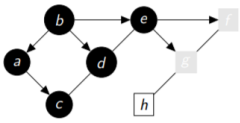
Q = [e, c]
1, 2

d)



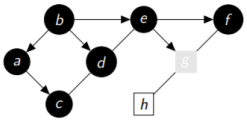
Q = [c, f, g]
2, 2, 2

e)



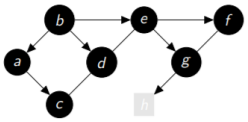
Q = [f, g]
2, 2

f)



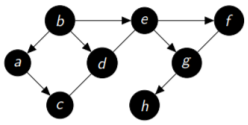
Q = [g]
2

g)



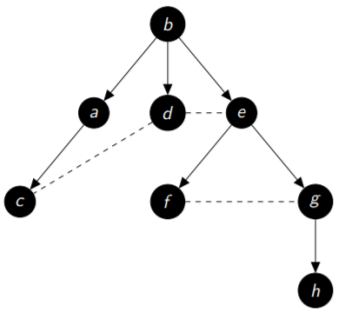
Q = [h]
3

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 (b, a, d, e, c, f, g, h).

Question 7: (*15 minutes*) **Breadth-First Search : 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 parcouru. Nous allons utiliser l'algorithme BFS car il renvoie toujours le chemin le plus court dans un graphe non pondéré (dont le poids entre tous les nœuds est inexistant ou virtuellement égal à 1). Implémentez l'algorithme en suivant les étapes suivantes :

1. Partir du sommet initial, construire un chemin avec le premier nœud et le mettre dans une **queue**.
2. Extraire le premier chemin de la **queue** et récupérer le dernier nœud du chemin. Si ce nœud n'a pas été visité, parcourir ses sommets adjacents (voisins). Pour chaque voisin construire un nouveau chemin et le mettre dans la **queue**.
3. Ajouter le nœud à la liste des nœuds **visités**. Une fois que cela est fait, supprimer le chemin parcouru de la queue.
4. Répéter les étapes 2 et 3 jusqu'à ce que la queue soit vide ou le nœud d'arrivée est atteint.

```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__':
    # Exemple de graphe implémenté sous la forme d'un dictionnaire
    graph = {'A': ['B', 'C', 'E'],
             'B': ['A', 'D', 'E'],
             'C': ['A', 'F', 'G'],
             'D': ['B', 'E'],
             'E': ['A', 'B', 'D'],
             'F': ['C'],
             'G': ['C']}
    print(bfs_shortest_path(graph, 'G', 'D'))  # devrait afficher  ['G', 'C', 'A', 'B', 'D']
```

Conseil

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

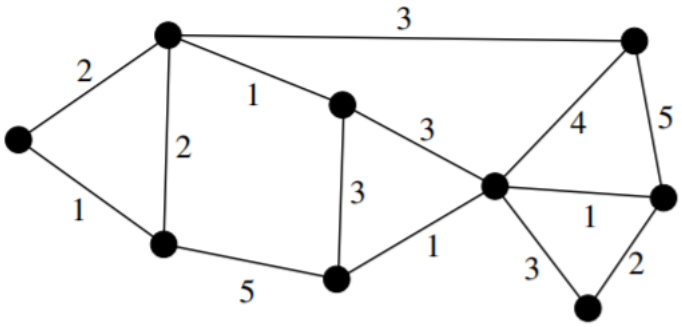
1. La file d'attente garde une trace des chemins possibles (implémentés sous forme de liste de nœuds) au lieu des nœuds.
2. lorsque l'algorithme recherche un nœud voisin, il doit vérifier si le nœud voisin correspond au nœud cible. Si c'est le cas, nous avons une solution et il n'est pas nécessaire de 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__':
    # Exemple de graphe sous forme de dictionnaire
    graph = {'A': ['B', 'C', 'E'],
             'B': ['A', 'D', 'E'],
             'C': ['A', 'F', 'G'],
             'D': ['B','E'],
             'E': ['A', 'B', 'D'],
             'F': ['C'],
             'G': ['C']}
    print(bfs_shortest_path(graph, 'G', 'D'))  # returns ['G', 'C', 'A', 'B', 'D']
```

Question 8: (*10 minutes*) **Algorithme de Kruskal : Papier**

Appliquez l'algorithme de Kruskal au graphe suivant :



Conseil

L'algorithme de Kruskal fonctionne de la façon suivante :

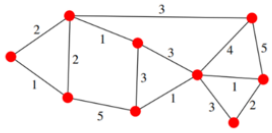
1. Classer les arêtes par ordre croissant de poids.
2. Prendre l'arête avec le poids le plus faible et l'ajouter à l'arbre (si 2 arêtes ont le même poids, choisir arbitrairement une des 2).
3. Vérifiez que l'arête ajoutée ne crée pas de cycle, si c'est le cas, supprimez la.
4. Répétez les étapes 2) et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

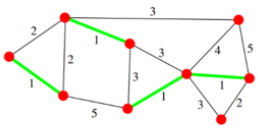
Solution

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

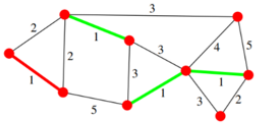
a)



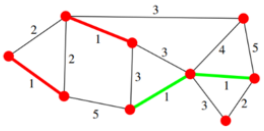
b)



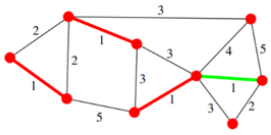
c)



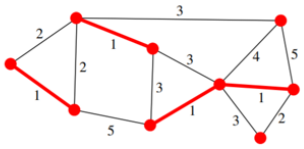
c)



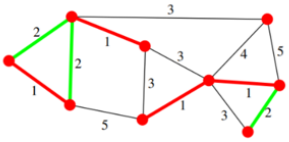
d)



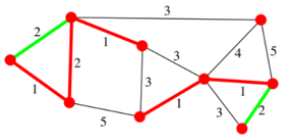
e)



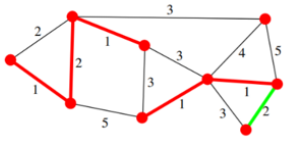
f)



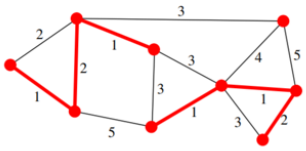
g)



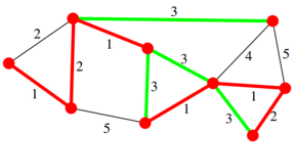
h)



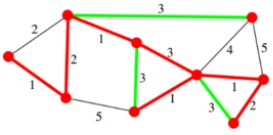
i)



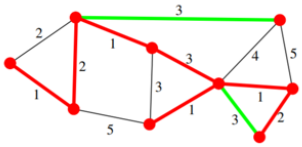
j)



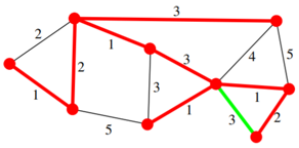
k)



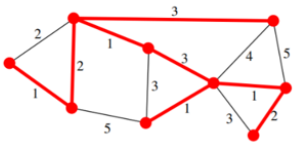
l)



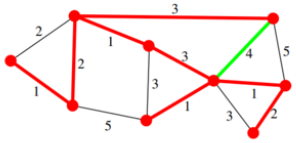
m)



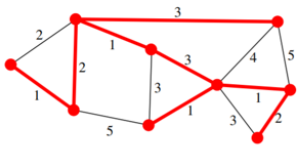
n)



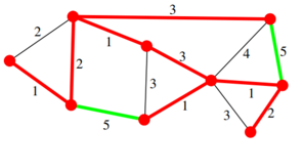
o)



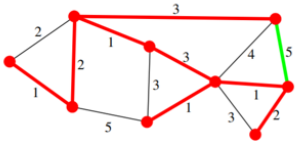
p)



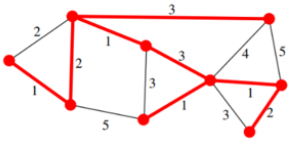
q)



r)



s)



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