Algorithmes et Pensée Computationnelle

Consolidation 2

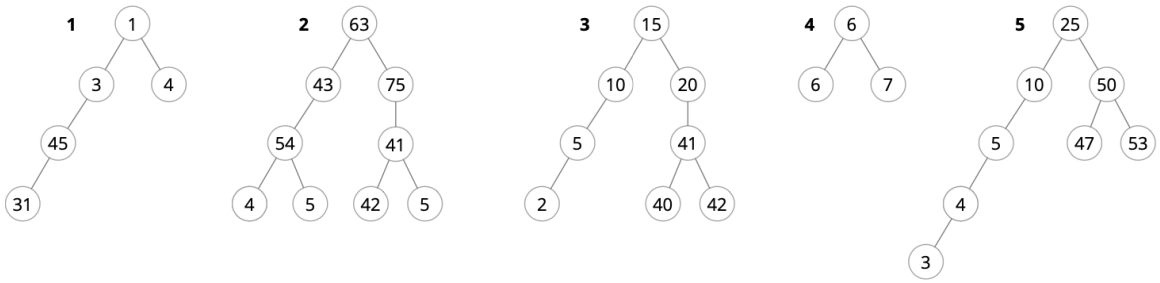
Les exercices de cette série sont une compilation d'exercices semblables à ceux vus lors des semaines précédentes. Le but de cette séance est de consolider les connaissances acquises lors des travaux pratiques des dernière semaines.

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)
```

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

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



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).

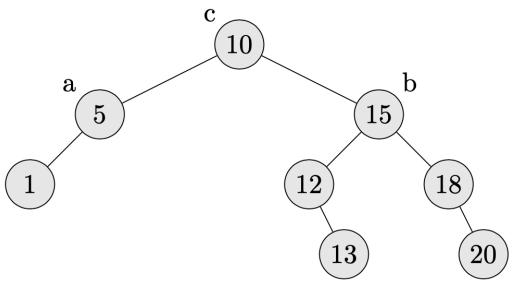
   (a) TREE-INSERT(T, z) avec z.key = 0

   (b) TREE-INSERT(T, z) avec z.key = 17

   (c) TREE-INSERT(T, z) avec z.key = 14

   (d) TREE-DELETE(T, a)

$T =$



(e) TREE-DELETE(T, b)

(f) TREE-DELETE(T, c)

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

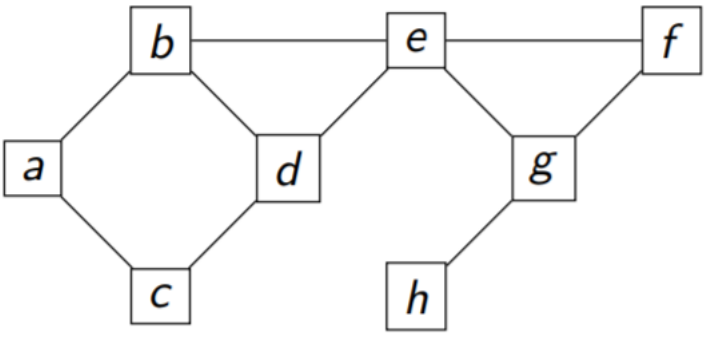
Nous avons vu comment exprimer la complexité d'un algorithme en terme de complexité temporelle dans le "pire des cas", notée $\mathcal{O}()$. L'exercice suivant permet de travailler avec différentes croissances de fonctions. Ordonner la liste de fonctions suivante selon leur croissance asymptotique : 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 : $\mathcal{O}(f(x)) < \mathcal{O}(g(x)) \Leftrightarrow \mathcal{O}(x) < \mathcal{O}(x^2)$.

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

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 :



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': ['A', 'B', 'D'],
             'F': ['C'],
             'G': ['C']}
    print(bfs_shortest_path(graph, 'G', 'D'))  # devrait afficher  ['G', 'C', 'A', 'B', 'D']
```

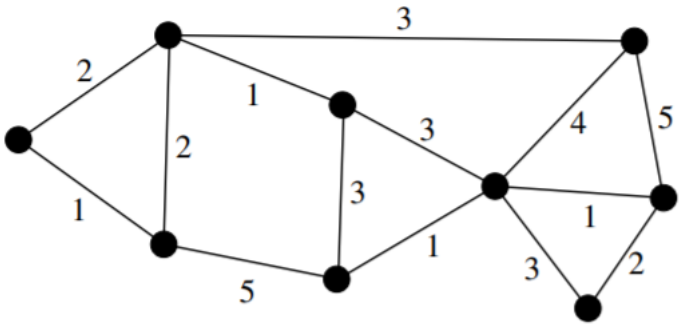
Conseil

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.

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.