# CONTRÔLE TERMINAL DÉCEMBRE 2022

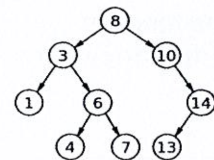
*documents autorisés : une feuille A4 manuscrite recto-verso*

**Ex 1 (2 pts) : Tri de bits**

On souhaite trier une liste de bits de manière à ce que tous les `0` se retrouvent à gauche, et tous les `1` à droite.
Ecrire une fonction `tri_bits` effectuant ce tri en vous inspirant du tri par comptage (aucune comparaison entre bits ne sera faite). Cette fonction renvoie une nouvelle liste triée.
Ex : l'appel `tri_bits([0,1,0,1,0,1,1,1,0])` renvoie la liste `[0,0,0,0,1,1,1,1,1]`.

**Ex 2 (4 pts) : Arbre binaire**

On souhaite écrire une classe implémentant un arbre binaire défini récursivement à partir de la valeur de sa racine (attribut valeur), de son fils gauche (attribut gauche) qui est un arbre, et de son fils droit (attribut droit) qui est aussi un arbre.
a) Donner le code de la classe Arbre, qui possède un constructeur et une fonction `est_vide` indiquant si l'instance courante est un arbre vide ou non (tous les attributs valent None).
b) Donner une méthode `postfixe(A)` prenant en paramètre une instance d'une telle classe et restituant les valeurs de l'arbre sous forme d'une chaîne de caractères, dans l'ordre où les rencontre un parcours postfixe de l'arbre.
Un tel parcours se déroule ainsi : parcours postfixe du sous-arbre gauche, puis parcours postfixe du sous-arbre droit, puis accès au nœud.
c) Dans quel ordre les nœuds de l'arbre ci-contre sont-ils visités lors de ce parcours ?



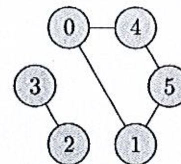

**Ex 3 (4 pts) : Palindrome**

Considérons la fonction suivante, qui indique si la chaîne **s** est un palindrome.

```
def palindrome(s):
  if len(s) <= 1:
    return True
  else:
    return s[0]  == s[-1] and palindrome(s[1 : -1])
```

a) Expliquer en une ou deux phrases son fonctionnement.
b) Combien d'appels récursifs sont-ils effectués lors de son exécution ?
c) Donner une version plus efficace de cette fonction, toujours sous forme récursive, en justifiant brièvement pourquoi elle est plus efficace. On pourra utiliser une fonction auxiliaire possédant des paramètres supplémentaires permettant d'accéder aux caractères à comparer.

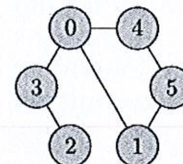
**Ex 4 (7 pts) : Graphes**
On considère le graphe non orienté suivant :




a) Combien possède-t-il de composantes connexes ? Est-il connexe ?

b) Donner la liste d'adjacences le représentant (sous forme de liste ou de dictionnaire).

c) Ecrire une fonction `ajoute(G, a)` qui prend en paramètre un graphe `G` (représenté par sa liste d'adjacence) et une arête `a`, et qui ajoute cette arête au graphe. Une arête est représentée par une liste de deux sommets [s1, s2]. Si l'un des sommets de l'arête fournie n'appartient pas au graphe, le graphe n'est pas modifié.
Ex : l'appel `ajoute(G, [0,3])` sur le graphe précédent modifie `G` pour obtenir le graphe ci-dessous.




d) Effectuer "à la main" un parcours en profondeur (DFS) du graphe ci-contre, à partir du sommet 0. Pour cela, on fera apparaître l'état de chaque structure de données lors de chaque itération, comme en TD.

e) Un parcours en largeur (BFS) d'un graphe permet d'obtenir la liste des distances reliant chaque sommet de ce graphe au sommet source du parcours. Pour rappel, ce calcul s'effectue de la façon suivante :

- on initialise les distances à -1
- pour chaque nouveau sommet `s` que l'on visite, si les distances de ses voisins n'ont pas encore été calculées, on leur donne pour valeur la distance de `s` augmentée de 1.

Durant ce parcours, il est possible de déterminer facilement si le graphe possède un cycle. Cela arrive si, lors du parcours des voisins `v` d'un sommet `s`, la distance de `v` est supérieure ou égale à celle de `s`. Si cela n'arrive pas, le graphe ne possède pas de cycle.

Ecrire une fonction `cycle_bfs(G,s0)` qui prend en paramètre un graphe `G` (représenté par sa liste d'adjacences) et un sommet source `s0`, et qui renvoie `True` si ce graphe possède un cycle, et `False` sinon. L'appel `cycle_bfs(G,0)` sur le graphe `G` de la question précédente renvoie `True` car la liste de sommets [0, 4, 5, 1] forme un cycle.

**Ex 5 (3 pts) : Problème du sac à dos**
On dispose d'un ensemble d'objets non fractionnables (on ne peut pas en prendre qu'une partie). Chaque objet possède une valeur et une masse. On souhaite prendre certains de ces objets dans un sac à dos, qui dispose d'une capacité limitée (en masse). On cherche à maximiser la somme des valeurs des objets que l'on met dans le sac à dos, sans en dépasser la capacité.
On suppose qu'on dispose d'une liste qui contient les objets disponibles, chaque objet étant représenté par un couple (valeur, masse). On souhaite donc disposer d'un algorithme qui prend en entrée cette liste et la capacité du sac à dos, et qui établit la liste des objets à prendre pour atteindre notre objectif.

Identifier trois types d'algorithme permettant de résoudre le problème.
Expliquer sommairement en quoi ils consistent, en donnant leurs avantages et/ou inconvénients.
On abordera notamment la question de leur complexité, ainsi que de leur optimalité. Aucun code n'est demandé.

**Ex 6 (2 pts) : Bonus : suite du problème du sac à dos**
Donner une fonction permettant de donner une solution au problème. Cette solution n'a pas l'obligation d'être optimale. Indiquer si elle est optimale, et à quelle catégorie d'algorithme elle appartient.