TP récursivité

1 Listes imbriquées

On suppose qu'une liste L peut contenir parmi ses éléments d'autres listes, qui à son tour contiennent d'autres listes et ainsi de suite, et on souhaite obtenir une liste formée des "mêmes" éléments que la première mais ne contenant, elle, pas de liste...

Autrement dit, de la liste `[1, [2, [3, 4], [5, [6, 7]]], [8], [[[9]]] ]` on voudrait obtenir la liste `[1, 2, 3, 4, 5, 6, 7, 8, 9]`.

Ecrire pour ce faire une fonction : `def aplatitListe(L)` qui admet pour argument la liste à traiter `L`, et qui renvoie la liste aplatie obtenue.

Pour tester si un objet est une liste, on peut utiliser la fonction `type` :

```
>>> type([2])
list
```

2 Fibonacci

On l'a vu, la formule de récurrence $u_n = u_{n-1} + u_{n-2}$ pour la suite de Fibonacci (de conditions initiales $u_0 = 0$ et $u_1 = 1$) conduit à une fonction récursive de complexité exponentielle, à moins de faire appel à la mémoïsation, où on retrouve la complexité linéaire attendue de l'algorithme impératif classique (sous l'hypothèse simplificatrice, et vite fausse bien sûr, que les opérations arithmétiques sur les entiers : somme, produit, s'exécutent en temps constant.)

On peut établir les deux formules suivantes : si $n = 2p$ est pair, alors $u_n = 2u_{p-1}u_p + u_p^2$ et si $n = 2p+1$ est impair, alors $u_n = u_p^2 + u_{p+1}^2$.

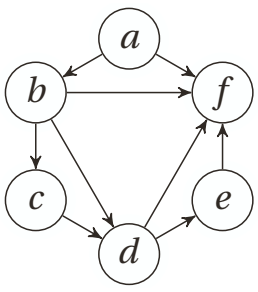
(Ce n'est pas utile à l'exercice, mais une méthode pour justifier ces formules peut être de poser A la matrice $\begin{pmatrix} u_2 & u_1 \\ u_1 & u_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. On observe en effet que pour tout n, $A^n = \begin{pmatrix} u_{n+1} & u_n \\ u_n & u_{n-1} \end{pmatrix}$ et en écrivant A^{2p} les formules ci-dessus s'obtiennent assez facilement.)

A l'aide des deux formules précédentes, écrire une fonction récursive `def fibo(n)` qui calcule le terme d'indice n de la suite de Fibonacci.

La question qui suit est à traiter à la maison (ou à la fin du TP si vous êtes efficace !) : Justifier la terminaison, la correction et préciser la complexité de la fonction `fibo(n)`.

3 Graphes acycliques orientés

Un graphe est dit orienté quand une arête joignant deux noeuds est munie d'une direction. Un cycle est un chemin dont l'origine et l'extrémité coïncident. Exemple d'un graphe acyclique orienté :



Pour représenter un graphe en python, on choisira des listes d'adjacence. Si bien que dans l'exemple précédent, en numérotant les sommets a à f de 0 à 5, le graphe sera connu par la liste : $[[1,5],[2,3,5],[3],[4,5],[5],[]]$

3.1 Recherche de chemin dans un graphe acyclique orienté

L'absence de cycle rend la recherche d'un chemin d'un noeud à un autre plus facile, et se prête bien à une implémentation récursive. Ecrire donc une fonction `chemin(G, n1, n2)` qui prend en argument un graphe `G` (ou plutôt sa liste d'adjacence), les indices de deux noeuds `n1` et `n2` et qui retourne `True` s'il existe un chemin de `n1` à `n2`, et `False` sinon. (Indication : s'il existe une arête de `n1` à `n2`, alors la réponse est `True`, sinon, on regarde les noeuds voisins de `n1` et s'il existe un chemin de l'un d'entre-eux vers `n2`.)

3.2 Création aléatoire de graphe acyclique orienté

Pour l'algorithme qui suivra, il pourra être utile d'être en mesure de créer à la demande des graphes acycliques orientés. Pour ce faire, en partant d'un certain nombre de noeuds et aucune arête, on va rajouter un certain nombre d'arêtes, en prenant garde à chaque nouvelle arête de ne pas créer de cycle. On utilisera la fonction `randint` du module `numpy.random`

Pour créer un graphe acyclique orienté comprenant n noeuds, on commence par créer une liste d'adjacence formée de n listes vides. (Attention à l'initialiser correctement...)

Puis on rajoute, ou on tente de rajouter un certain nombre d'arêtes.

Ecrire une fonction `ajouteUneArete(G)` qui essaie d'ajouter une arête au graphe G. (Essaie seulement, car il pourrait arriver qu'aucune arête ne puisse être ajoutée au graphe G) En tirant au hasard les indices de deux noeuds, si les deux indices sont distincts, si l'arête n'existe pas déjà et si aucun cycle n'est créé avec cette arête, alors l'arête est ajoutée au graphe. (La fonction pourra retourner `True` si une arête a été ajoutée, et `False` sinon.

Ecrire également une fonction `creeGraphe(n)` qui crée un graphe acyclique orienté formé de n noeuds. (Par exemple en tentant pour n noeuds de créer n^2 arêtes, mais on pourra tenter d'autres valeurs.

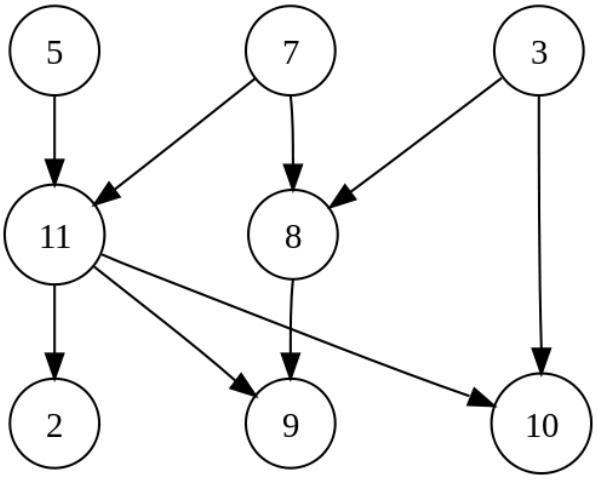
3.3 Tri topologique

Un graphe acyclique orienté est utile pour présenter un ensemble de tâches dont certaines doivent impérativement être réalisées avant d'autres. Par exemple : la mise à jour de certains programmes peut nécessiter la mise à jour de bibliothèques qui elles-mêmes supposent que d'autres bibliothèques soient mises à jour.

Il est alors utile d'ordonner l'ensemble des noeuds de telle manière que si le noeud `n1` est placé avant le noeud `n2`, alors il n'existe aucun chemin de `n2` vers `n1` (l'existence d'un tel chemin indiquerait que la tâche associée au noeud `n2` doit être exécutée avant celle liée au noeud `n1`.)

Dans le cas de l'exemple proposé précédemment, la seule manière d'ordonner les noeuds est a, b, c, d, e, f, mais en règle générale, il n'y a pas unicité d'un ordre adéquat.

Exemple :



Déterminer plusieurs tris topologiques du graphe ci-dessus.

Un algorithme simple pour obtenir un tri topologique d'un graphe orienté acyclique est le suivant : on retire un noeud n et on réalise un tri topologique du graphe obtenu, puis on insère dans la liste triée des noeuds le noeud n.

On passe pour ce faire en revue les noeuds de la liste triée, et on insère le noeud n retiré initialement après le dernier noeud qui admet n comme voisin (ou en tête de liste si n n'est l'extrémité d'aucune arête).

Bien sûr, on procèdera avec une procédure récursive.

Indication : Plutôt que de modifier effectivement le graphe pour en retirer un noeud, on suggère d'ajouter à la fonction qu'on va écrire un argument optionnel entier. Si celui-ci n'est pas fourni, on tâche de réaliser le tri topologique du graphe entier, et avec un argument entier a, on réalise le tri topologique du graphe réduit à ses a premiers noeuds, ainsi la fonction pourrait commencer ainsi :

```
def triTopologique(G, a = None):
    """ avec un second argument a, retourne un tri topologique
    du sous-graphe formé des a premiers noeuds de G. Sans le second
    argument, trie l'intégralité de G."""
    if a == None: # on traite tout le graphe
        a = len(G)
    if a == 1:       # un seul noeud, le tri est déjà terminé !
        return [0]
    ....
```

3.4 Pour aller plus loin

Quelle est la complexité de l'algorithme de `triTopologique` étudié ci-dessus ? Comment pourrait-on l'améliorer ?

Quid de la complexité de la création d'un graphe aléatoire ? Comment ici aussi améliorer celle-ci ?

3.5 Annexe

Sur `http://cahier-de-prepa.fr/mp-kju` On pourra télécharger le fichier `decorateurs.py` qui introduit deux fonctions nommées `trace` et `memoize` qui définissent ce qu'on appelle des décorateurs, car elles viennent apporter une fonctionnalité supplémentaire à une fonction donnée, en précédant la définition d'une fonction par, ou bien `@trace`, ou bien `@memoize`.

Exemple :

```
from decorateurs import *

@memoize
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

@trace
def pgcd(a, b):
    if b == 0:
        return a
    else:
        return pgcd(b, a % b)
```

(Essayez fib(100), pgcd(24, 15) par exemple...)

Une précision technique : dans le processus de memoïzation, à chaque fois qu'une valeur est calculée, donnant la valeur de retour pour des arguments donnés, une entrée est ajoutée dans un dictionnaire, avec comme clé le tuple formé des arguments d'appels, et comme valeur la valeur de retour de la fonction.

Or un dictionnaire ne peut admettre comme clé que des objets immuables et, pour cette raison, le décorateur échouera si parmi les arguments d'une fonction donnée figure un objet mutable, comme une liste par exemple. Aucun souci avec les entiers de la fonction précédente par exemple...