# Eléments de correction

## 1 Listes imbriquées

Par exemple :

```
def aplatitListe2(L):
    M = []
    for i in L:
        if type(i) == list:
            M += aplatitListe2(i)
        else:
            M.append(i)
    return M
```

## 2 Fibonacci

Une réponse possible :

```
def fibo(n):
    if n < 2:
        return n
    p = n // 2
    if n % 2:
        return fibo(p) ** 2 + fibo(p + 1) ** 2
    else:
        fp = fibo(p)
        return 2*fibo(p-1)*fp + fp ** 2
```

Bien sûr, il vaut bien mieux éviter d'écrire `fibo(p)*fibo(p)` par exemple, pour éviter de calculs redondants et coûteux (et la variable temporaire `fp` ci-dessus est là pour la même raison).

Pour la terminaison de `fibo`, on peut remarquer que l'argument `n` de la fonction est un variant, car si $n \leqslant 1$, la fonction s'interrompt, et que sinon, les valeurs invoquées le sont pour des nouvelles valeurs de $n$ strictement inférieures.

Pour la correction, elle provient bien sûr de l'exactitude des deux formules de récurrences données dans l'énoncé (lesquelles se déduisent comme indiqué par l'énoncé par le calcul matriciel de $A^{2p}$ en fonction de $A^p$.)

Pour ce qui concerne la complexité, en notant $T(n)$ le temps nécessaire à l'exécution de `fibo(n)`, alors on peut approcher $T(2p)$ par $T(p) + T(p-1) + c$ où $c$ est une constante et $T(2p+1)$ par $T(p) + T(p+1) + c$.

On devine une complexité linéaire selon $n$... Le montrer est un peu délicat. En posant $u_n = T(2^n)$, on approche $u_{n+1}$ par $2u_n + c$ et le calcul montre alors que $u_n = 2^n(u_0 - c) + c$ donc $T(2^n)$ est en effet proportionnel à $2^n$... L'hypothèse de croissance de $T$ permet alors de justifier que $T(n)$ est en $O(n)$.

## 3 Tri topologique

```
def add_node(G, n1, n2):
    G[n1].append(n2)

def initialiseGraphe(n):
    G = [[] for i in range(n)]
    return G
```

```
def chemin(G, n1, n2):
    """ retourne True s'il existe un
    chemin de n1 vers n2, False sinon """
    if n2 in G[n1]:
        return True
    for n in G[n1]:
        if chemin(G, n, n2):
            return True
    return False

def ajouteUneArete(G):
    """ Choisit deux noeuds distincts. Si les deux noeuds
    sont indentiques, ou si l'arête existe déjà, ou si
    le graphe obtenu a un cycle, ne fait rien, sinon ajoute
    une arête à G """
    nbNoeuds = len(G)
    n1 = randint(nbNoeuds)
    n2 = randint(nbNoeuds)
    if n1==n2 or n2 in G[n1] or chemin(G, n2, n1):
        return False
    else:
        G[n1].append(n2)
    return True

def creeGraphe(n):
    G = initialiseGraphe(n)
    for i in range(n*n):
        ajouteUneArete(G)
    return G

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:
        a = len(G)
    if a == 1:
        return [0]

    tri = triTopologique(G, a-1)
    min_i = 0
    i = a-2
    while i >= 0 and min_i == 0:
        if a-1 in G[tri[i]]:
            min_i = i+1
        i -= 1
    tri.insert(min_i, a-1)
    return tri
```

La complexité de la fonction `triTopologique` est en $O(n*(n+a))$ où $n$ est le nombre de noeuds du graphe à trier, et $a$ son nombre d'arêtes. En effet, en notant $T(n)$ le nombre d'instructions nécessaires à obtenir un tri topologique d'un graphe de $n$ noeuds, on observe que $T(1) = 0$, et pour tout $n$, $T(n) = T(n-1) + O(n+a)$ car une fois trié un graphe de $n-1$ noeuds, on passe en revue les arcs issus de ces $n-1$ noeuds (au nombre de $a$ au plus) puis on insère le noeud restant dans une liste de $n-1$ termes.

Un meilleur algorithme pour réaliser le tri tropologique d'un graphe acyclique orienté est le suivant :

- On compte pour chaque noeud le nombre d'arêtes qui y aboutissent. Une fois ce travail effectué, alors les

candidats pour commencer notre liste triée sont faciles à trouver : ce sont ceux qui ne sont l'extrémité d'aucune arête, donc ceux pour lesquels le nombre obtenu vaut 0.

- On crée une liste formée de ces noeuds (qui peuvent tous être initiaux) et on retire n'importe lequel d'entre eux (autant utiliser bien sûr `pop` pour des raisons d'efficacité...), et on met à jour la liste des nombres d'arêtes obtenue précédemment (en considérant qu'on a retiré du graphe le sommet choisi). Si de ce fait, des noeuds se retrouvent sans noeud prédécesseur, on les rajoute à la liste des noeuds qui ont cette propriété...

Une implémentation de cet algorithme est la suivante :

```
def triTopologique2(G):
    n = len(G)
    nbPred = [0] * n
    for i in G:
        for j in G[i]:
            nbPred[j] += 1
    L = []
    for i in range(n):
        if nbPred[i] == 0:
            L.append(i)
    tri = []
    while len(L) > 0:
        i = L.pop()
        tri.append(i)
        for j in G[i]:
            nbPred[j] -= 1
            if nbPred[j] == 0:
                L.append(j)
    return tri
```

Sa complexité est en $O(n + a)$ où $n$ est le nombre de noeuds et $a$ le nombre d'arêtes. En effet, les deux premières boucles imbriquées comptent $n + a$ itérations, la seconde boucle for en compte $n$ et la boucle conditionnelle qui termine la fonction compte $n$ itérations, et le nombre total d'itérations de la boucle for à l'intérieur de celle-ci compte $a$ itérations...

Une autre approche équivalente en termes de complexité et très élégante est de parcourir en profondeur le graphe, en gardant en mémoire les noeuds déjà visités. Voici l'implémentation de l'un de vos prédécesseurs :

```
def triTopologique2(G):
    n = len(G)
    L, visites = [], [False for i in range(n)]
    for i in range(n):
        if not visites[i]:
            parcoursProfondeur(G, i, visites, L)
    return L[::-1]

def parcoursProfondeur(G, n, visites, L):
    visites[n] = True
    for v in G[n]:
        if not visites[v]:
            parcoursProfondeur(G, v, visites, L)
    L.append(n)
```

Pour ce qui concerne la création aléatoire de graphe acyclique orienté, la complexité de l'algorithme naïf proposé est élevée à cause de la vérification effectuée à chaque arête ajoutée qu'un cycle n'est pas créé. Cette vérification est faite par un parcours en profondeur qui, pour un graphe de $n$ noeuds et $a$ arêtes a une complexité en $O(n+a)$.

Comme il y a au plus $n^2$ arêtes pour ce que propose l'énoncé, la complexité obtenue est ici de l'ordre de $O(n^4)$.

Une amélioration pourrait être de garder trace au fur et à mesure de la construction du graphe de l'existence ou non d'un chemin du noeud $i$ au noeud $j$. On peut s'aider d'une matrice de $n$ lignes et colonnes pour ce faire. L'ajout d'une arête conduit à mettre à jour une colonne de la matrice et est ainsi une opération linéaire (selon le nombre $n$ de noeuds) ce qui conduit à une complexité en $O(n^3)$.

Une idée astucieuse, due à un élève, pour améliorer la complexité de cet algorithme est de partir du principe qu'un graphe acyclique orienté admet toujours un tri topologique, et d'imposer en quelque sorte celui-ci en mélangeant les noeuds, puis en n'ajoutant d'arêtes entre les noeuds $i$ et $j$ qu'à condition que le noeud $i$ précède $j$ dans la liste mélangée des noeuds.

Une première version est de complexité en $O(n^3)$ (car le test de présence d'une arête se fait en temps linéaire) :

```
def randint2(n):
    i1, i2 = randint(n), randint(n)
    if i2 < i1:
        i1, i2 = i2, i1
    return (i1, i2)

def creeGraphe2(n):
    L = list(range(n))
    shuffle(L)
    G = [[] for i in range(n)]
    for i in range(n*n):
        i1, i2 = randint2(n)
        if i1 != i2:
            n1, n2 = L[i1], L[i2]
            if not n2 in G[n1]:
                G[n1].append(n2)
    return G
```

et une seconde version avec une matrice d'adjacence pour vérifier l'existence d'une arête entre deux noeuds en temps constant (ce qui conduit alors à une complexité en $O(n^2)$ qui ne peut qu'être optimale bien sûr...) :

```
def creeGraphe3(n):
    M = [[False for i in range(n)] for j in range(n)]
    L = list(range(n))
    G = [[] for i in range(n)]
    shuffle(L)
    for i in range(n*n):
        i1, i2 = randint2(n)
        if i1 != i2:
            n1, n2 = L[i1], L[i2]
            if not M[n1][n2]:
                G[n1].append(n2)
                M[n1][n2] = True
    return G
```

Remarque : pour battre efficacement un jeu de cartes, des algorithmes existent en temps linéaire, pourvu bien sûr de disposer d'un générateur efficace de nombres aléatoire. La fonction `shuffle` utilisée ci-dessus vient du module `numpy.random`.