Louis CHARLES
DM Informatique Mars 2019
Sujet : X PSI et PT 2015

Préliminaires

Les deux fonctions ci-dessous sont considérées comme à disposition.
En voilà une implémentation en python.

```
"""
Fonction
Renvoie un entier aléatoire entre 1<=n<=k
"""
def entierAleatoire(k):
    return randint(1,k)

"""
Fonction
Crée un tableau de capacité n initialisé à None (vide)
"""
def creerTableau(n):
    return [None] * n;
```

Partie I

Question 1

```
"""
Fonction
Crée une liste vide de capacité n
(Remplie de None avec nombre d'éléments à 0 dans res[0]
"""
def creerListeVide(n):
    res = creerTableau(n + 1)
    res[0] = 0
    return res
```

Question 2

On trouvera systématiquement la description de la complexité en commentaire

```
"""
Fonction
Indique si l'élément x est dans la liste liste

Complexité :
    Unité :         comparaison
    Paramètres :    n (capacité de la liste)
    Nature :        Pire des cas :
                        - la liste de capacité n est pleine
                        - x est en dernière position dans la liste
    Nom :           cpxEstDansListe(n)
    Formule :       Lecture : 1 test par passage dans la boucle
                    cpxEstDansListe(n) = n
                    cpxEstDansListe(n) = O(n)
"""
def estDansListe(liste, x):
    res = False
    for i in range(1, liste[0] + 1):
        if x == liste[i]:
            res = True
            break
    return res
```

Question 3

```
"""
Procédure
Ajoute l'élément x dans la liste liste s'il n'est pas présent

Complexité :
    Unité :         comparaison
    Paramètres :    n (capacité de la liste)
    Nature :        Pire des cas :
                        - la liste de capacité n est pleine
                        - x est en dernière position dans la liste
    Nom :           cpxAjouterDansListe(n)
    Formule :       Lecture : complexité de estDansListe
                    cpxAjouterDansListe(n) = cpxEstDansListe(n)
                    cpxAjouterDansListe(n) = O(n)
"""
def ajouterDansListe(liste, x): # OutOfBound si plein
    if not estDansListe(liste, x):
        liste[liste[0]+1] = x
        liste[0] = liste[0] + 1
```

Si la liste comporte déjà n éléments utiles, nous obtiendrons un dépassement de capacité de celle-ci, car nous ne vérifions pas si le nombre d'éléments initialisés (dans liste[0]) n'a pas déjà atteint len(liste)+1.

Partie II

Question 4

Plan 1	Plan 2
[[5 ;7] ; [2 ;2 ;3 ;None ;None ;None] ; [3 ;2 ;3 ;5 ;None ;None] ; [4 ;1 ;2 ;4 ;5 ;None] ; [2 ;3 ;5 ;None ;None ;None] ; [3 ;2 ;3 ;4 ;None ;None] ;]	[[5 ;4] ; [1 ;2 ;None ;None ;None ;None] ; [3 ;1 ;3 ;4 ;None ;None] ; [1 ;2 ;None ;None ;None ;None] ; [2 ;2 ;5 ;None ;None ;None] ; [1 ;4 ;None ;None ;None ;None] ;]

Question 5

```
"""
Fonction
Crée et retourne un plan sans aucune route
"""
def creerPlanSansRoute(n):
    plan = creerTableau(n+1)  # le "header" et un tableau par ville
    plan[0] = [n, 0]
    for i in range (1, n+1):
        plan[i] = creerListeVide(n)
    return plan
```

Question 6

```
"""
Fonction
Retourne un booléen indiquant si x et y sont voisines dans plan

Complexité :
    Unité :          comparaison
    Paramètres :     n (nombre de villes du plan)
    Nature :         Pire des cas :
                         - la ville x a n-1 voisines
                         - y est en dernière position dans la liste des voisines de x
    Nom :            cpxEstVoisine(n)
    Formule :        Lecture : complexité de estDansListe(n)
                     cpxEstVoisine(n) = cpxEstDansListe(n)
                     cpxEstVoisine(n) = O(n)
"""
def estVoisine(plan, x, y):
    return estDansListe(plan[x], y)
```

Question 7

```
"""
Procédure
Ajoute une route entre les villes x et y dans le plan p

Complexité :
    Unité :          comparaison
    Paramètres :     n (nombre de villes du plan)
    Nature :         Pire des cas :
                         - la ville x a n-1 voisines
                         - y est en dernière position dans la liste des voisines de x
    Nom :            cpxAjouteRoute(n)
    Formule :        Lecture : complexité de estVoisine et deux fois celle de ajouterDansListe
                     cpxAjouteRoute(n) = cpxEstVoisine(n)+2*cpxAjouterDansListe(n)
                     cpxAjouteRoute(n) = O(n)
"""
def ajouteRoute(plan, x, y):    # Pas de contrôle des arguments (ni x != y)
    if not estVoisine(plan, x, y):
        plan[0][1]=plan[0][1] + 1
        ajouterDansListe(plan[x],y)
        ajouterDansListe(plan[y],x)
```

Il n'y a pas de risque de dépassement de capacité car ajouterDansListe ne fait effectivement l'ajout que si la ville n'est pas déjà dans la liste des voisines. Comme chaque liste de voisines a une capacité totale du nombre de villes, il est impossible de dépasser.

Par contre, on ne vérifie pas ici que la route qu'on ajoute ne relie pas une ville x à elle-même, ni que les villes x et y ont bien un numéro inférieur à n (nombre de villes que l'on pourrait trouver dans plan[0][0].

Question 8

```
"""
Fonction
Retourne sous forme de liste intelligente tous les routes du plan (couple de (x,y))

Complexité :
    Unité :          affectation (append)
    Paramètres :     n (nombre de villes du plan) et m (nombre de routes du plan)
    Nature :         Exact : Nous parcourons en intégralité toutes les voisines
                     (qui sont présentes 2m fois dans le tableau plan)
    Nom :            cpxAfficheToutesLesRoutes(m)
    Formule :        Lecture : nous parcourons au maximum 2m routes
                     cpxAfficheToutesLesRoutes(m) = 2m
                     cpxAfficheToutesLesRoutes(m) = O(m)
"""
def afficheToutesLesRoutes(plan):
    n,m = plan[0]
    routes = [m]
    for i in range (1, n+1):
        for j in range (1, plan[i][0]+1):
            if plan[i][j] > i:
                routes.append([(i, plan[i][j])])
    return routes
```

Si on affiche *routes*, on obtient une liste « intelligente » (i.e. le premier élément contient le nombre) des couples de villes reliées par une route, et ce, sans doublon.

Partie III

Question 9

```
"""
Procédure
Colorie les villes de manière aléatoire avec une couleur entre 1 et k
Affecte la couleur 0 à s et k+1 à t

Complexité :
    Unité :         affectation
    Paramètres :    n (nombre de villes du plan)
    Nature :        Exact
    Nom :           cpxColoriageAleatoire(n)
    Formule :       Lecture : nous parcourons exactement n villes + s et t
                    cpxColoriageAleatoire(n) = n+2
                    cpxColoriageAleatoire(n) = O(n)
"""
def coloriageAleatoire(plan, couleur, k, s, t):
    for i in range (1, plan[0][0]+1):
        couleur[i] = entierAleatoire(k)
    couleur[s] = 0
    couleur[t] = k+1
```

Question 10

```
"""
Fonction
Retourne la liste des villes voisines de x, et de couleur c

Complexité :
    Unité :         test
    Paramètres :    n (nombre de villes du plan)
    Nature :        Pire des cas : la ville x a n-1 voisines
    Nom :           cpxVoisinesDeCouleur(n)
    Formule :       Lecture : nous parcourons exactement n-1 villes
                    cpxVoisinesDeCouleur(n) = n-1
                    cpxVoisinesDeCouleur(n) = O(n)
"""
def voisinesDeCouleur(plan, couleur, i, c):
    voisines = []
    for j in range(1, plan[i][0]+1):
        if couleur[plan[i][j]] == c:
            voisines.append(plan[i][j])
    return voisines
```

Question 11

```
"""
Fonction
Retourne la liste sans doublon des villes voisines des villes de liste, et de couleur c

Complexité :
    Unité :          test
    Paramètres :     n (nombre de villes du plan), m (nombre de coutes du plan)
    Nature :         Pire des cas :
                         - liste contient n villes
                         - on est amené à parcourir tout le tableau plan, soit 2m routes
    Nom :            cpxVoisinesDeLaListeDeCouleur(n,m)
    Formule :        Lecture :
                         - nous parcourons au maximum (n-1) villes et appelons voisinesDeCouleur
                           => (n-1)(n-1)
                         - dans les boucles en i et j, nous parcourons 2m chemins (aller/retour des m routes)
                           Nous y appellons ajouterDansListe de complexite n
                           => 2m(n-1)
                     => cpxVoisinesDeLaListeDeCouleur(n,m) = (n-1)(n-1) + 2m(n-1) = (n-1)(2m + n-1)
                     cpxVoisinesDeCouleur(n,m) = O(n(n+m))
"""
def voisinesDeLaListeDeCouleur(plan, couleur,liste, c):
    voisines = creerListeVide(plan[0][0])
    for i in range(1, liste[0]+1):                                          # maximum n-1
        voisinesDeCouleurDeI = voisinesDeCouleur(plan, couleur, liste[i], c)    # complexité n-1
        for j in range(0, len(voisinesDeCouleurDeI)):
            ajouterDansListe(voisines, voisinesDeCouleurDeI[j])             # complexité n - réalisé 2m
fois maximum
    return voisines
```

Question 12

```
"""
Fonction
Retourne un booleen indiquant s'il existe un chemin arcenciel entre s et t, avec k étapes exactement

Complexité :
    Unité :          test
    Paramètres :     n (nombre de villes du plan), m (nombre de routes du plan), k (nombre de villes étapes)
    Nature :         Pire des cas (par héritage des complexités des fonctions appelées)
    Nom :            cpxExisteCheminArcEnCiel(n,m,k)
    Formule :        Lecture :
                         - nous colorions le plan : complexité n+2
                         - nous appelons voisinesDeLaListeDeCouleur k fois => k((n-1)(2m + n-1))
                           Nous y appellons ajouterDansListe de complexite n
                           => 2m(n-1)
                     => cpxExisteCheminArcEnCiel(n,m,k) = n+2 + k((n-1)(2m + n-1))
                     cpxExisteCheminArcEnCiel(n,m,k) = O(kn(n+m))
"""
def existeCheminArcEnCiel(plan, couleur, k, s, t):
    coloriageAleatoire(plan, couleur, k, s, t)      # complexité n+2
    listeVilles = creerListeVide(plan[0][0])
    ajouterDansListe(listeVilles, s)                # complexité exacte : 1
    coul = 0
    while coul < k+1:
        coul = coul + 1
        listeVilles = voisinesDeLaListeDeCouleur(plan, couleur, listeVilles, coul)  #Maxi k fois
        if listeVilles[0] == 0:
            return False
    return True
```

Partie 4

Question 13

```
"""
Fonction
Retourne un booleen indiquant avec une probabilité au moins 1-1/e s'il existe un chemin entre s et t
    qui passe par exactement k villes intermédiaires distinctes

Complexité :
    Unité :         test
    Paramètres :    n (nombre de villes du plan), m (nombre de coutes du plan), k (nombre de villes étapes)
    Nature :        Pire des cas (par héritage des complexités des fonctions appelées)
    Nom :           cpxExisteCheminSimple(n,m,k)
    Formule :       Lecture : nous appelons au maximum existeCheminArcEnCiel k**k fois
                    => cpxExisteCheminSimple(n,m,k) = k**k*n + k((n-1)(2m + n-1)))
                    cpxExisteCheminSimple(n,m,k) = O(k**(k+1)n(n+m)) (f(k) = k**(k+1))
"""
def existeCheminSimple(plan, couleur, k, s, t):
    for _ in range(0, k**k):
        if existeCheminArcEnCiel(plan, couleur, k, s, t):
            return True
    return False
```

Question 14

Pour fournir le premier chemin trouvé, il faut mémoriser tous les chemins qu'on parcourt.
Ainsi, au lieu de travailler sur des villes, on travaille sur des chemins. Il faut alors modifier 3 fonctions ci-dessus, la ville considérée étant alors la dernière ville du chemin.

Cela donne :

```
"""
Fonction
Retourne la liste sans doublon des villes voisines de la dernière ville des chemins de liste, et de couleur
c
"""
def voisinesDeLaListeDesCheminsDeCouleur(plan, couleur,liste, c):
    cheminsPossibles = []
    for i in range(len(liste)):
        voisinesDeCouleurDeI = voisinesDeCouleur(plan, couleur, liste[i][-1], c) # Dernière ville du chemin
        for j in range(0, len(voisinesDeCouleurDeI)):
            chemin = liste[i].copy();
            if not voisinesDeCouleurDeI[j] in chemin:
                chemin.append(voisinesDeCouleurDeI[j])
            cheminsPossibles = cheminsPossibles + [chemin]
    return cheminsPossibles
```

Pour la recherche du chemin arc-en-ciel, cela donne :

```
"""
Fonction
Retourne s'il existe un chemin arc-en-ciel
"""
def existeEtFournitCheminArcEnCiel(plan, couleur, k, s, t):
    coloriageAleatoire(plan, couleur, k, s, t)
    listeChemins = [[s]]
    coul = 0
    while coul < k+1:
        coul = coul + 1
        listeChemins = voisinesDeLaListeDesCheminsDeCouleur(plan, couleur,
listeChemins, coul)
        if len(listeChemins) == 0:
            return False, None
    return True, listeChemins
```

Et enfin, pour la fonction de détection et de fourniture du chemin trouvé :

```
"""
Fonction
Retourne s'il existe un chemin avec une probabilité au moins 1-1/e
"""
def existeEtFournitCheminSimple(plan, couleur, k, s, t):
    for _ in range(0, k**k):
        existe, chemin = existeEtFournitCheminArcEnCiel(plan, couleur, k, s, t)
        if existe:
            return True, chemin
    return False, None
```

Compléments

On peut lancer un grand nombre de fois cette fonction alors qu'on sait q'il n'y a qu'un seul chemin allant de s à t et passant par k villes exactement. On trouve alors un taux de détection proche de 1-1/e. Lorsque le nombre de chemins solution augmente, alors le taux de détection augmente aussi

On écrit une fonction qui permet de détecter la présence d'un chemin passant par k villes exatement, allant de s à t, avec une probabilité de détection d'au moins p, de la manière suivante :

```
"""
Fonction
Retourne s'il existe un chemin avec une probabilité au moins égale à p
"""
def detecteEtTrouveChemin(plan, couleur, k, s, t, p):
    for i in range (floor(-log(1-p))+1):
      detecte, chemins = existeEtFournitCheminSimple(plan, couleur, k,s,t)
      if detecte :
          return True, chemins
          break
    return False, None
```

Exécution :

```
plan = creerPlanSansRoute(11)
couleur=[None]*12
ajouteRoute(plan, 1, 2)
ajouteRoute(plan, 1, 6)
ajouteRoute(plan, 2, 1)
ajouteRoute(plan, 2, 7)
ajouteRoute(plan, 3, 4)
ajouteRoute(plan, 3, 8)
ajouteRoute(plan, 4, 8)
ajouteRoute(plan, 5, 6)
ajouteRoute(plan, 5, 10)
ajouteRoute(plan, 6, 7)
ajouteRoute(plan, 7, 8)
ajouteRoute(plan, 8, 9)
ajouteRoute(plan, 9, 11)
ajouteRoute(plan, 10, 11)

print(detecteEtTrouveChemin(plan, couleur, 5, 6, 4, 0.99))

(True, [[6, 5, 10, 11, 9, 8, 4]])
```