Épreuve d'informatique X/ENS - 2015 - PSI/PT
Quand la taille n'est pas un problème

Partie I. Préliminaires : Listes sans redondance

Question 1 : On commence par créer un tableau de $(n+1)$ cases (ligne 2) et on met un zéro en première case (ligne 3) pour signaler que la liste créée est vide. Enfin, on retourne la liste ainsi créée (ligne 4).

```
1 def creerListeVide(n):
2     a = creeTableau(n+1)
3     a[0] = 0
4     return(a)
```

Question 2 : On récupère n le nombre d'éléments présents dans la liste (ligne 2). Dans la boucle des lignes 3 à 5, on parcourt les éléments de la liste. Si l'un des éléments correspond à l'élément x cherché, on quitte la fonction en retournant `True` (ligne 5). Si l'élément n'est pas présent dans la liste, la boucle se finit et on retourne `False` en ligne 6.

```
1 def estDansListe(liste, x):
2     n = liste[0]
3     for i in range(1,n+1):
4         if x == liste[i]:
5             return(True)
6     return(False)
```

Dans le pire des cas, l'élément x n'est pas dans le tableau, on le parcourt en entier. La complexité est en $O(n)$.

Question 3

```
1 def ajouteDansListe(liste, x):
2     if not(estDansListe(liste,x)):
3         n = liste[0]
4         liste[0] = n+1
5         liste[n+1] = x
```

Si la liste est initialement pleine, à l'exécution de la ligne 5, l'interpréteur Python va signaler un problème de débordement de tableau et le programme va s'arrêter avec un message d'erreur.
Dans le pire des cas, la complexité de la fonction `ajouteDansListe` est conditionnée par la complexité de la fonction `estDansListe` (les opérations des lignes 3 à 5 se faisant à coût constant), la complexité est donc en $O(n)$.

Partie II. Création et manipulation de plans (PSI*)

Question 4 : Le plan 1 peut être représenté par la variable `plan1` suivant :

```
plan1=[  [5  7],
         [2  2  3  *  *],
         [3  1  3  5  *],
         [4  1  2  4  5],
         [2  3  5  *  *],
         [3  2  3  4  *]  ]
```

Le plan 2 peut être représenté par la variable `plan2` suivant :

```
plan2=[  [5  4],
         [1  2  *  *  *],
         [3  1  3  4  *],
         [1  2  *  *  *],
         [2  2  5  *  *],
         [1  4  *  *  *]  ]
```

Question 5 : Pour créer un plan à n villes, on commence par créer un tableau à $(n+1)$ éléments (ligne 2). On remplit la première case du tableau avec un tableau à deux éléments (ligne 3). Dans la boucle des lignes 6 à 7, on affecte chacun des éléments du tableau avec les informations correspondant à n villes et zéro route. On remplit les autres cases du tableau correspondant au plan avec des listes vides ayant une capacité de $(n-1)$ éléments. On retourne le plan ainsi créé à la ligne 8.

```
1 def creerPlanSansRoute(n):
2     plan = creerTableau(n+1)
3     plan[0] = creerTableau(2)
4     plan[0][0] = n
5     plan[0][1] = 0
6     for i in range(1,n+1):
7         plan[i] = creerListeVide(n-1)
8     return(plan)
```

Question 6 : Pour voir si la ville y est voisine de la ville x, il suffit de vérifier que la ville y fait partie des villes reliées à la ville x par une route, c'est-à-dire vérifier que y est dans la liste `plan[x]`.

```
1 def estVoisine(plan, x, y):
2     return(estDansListe(plan[x],y))
```

Question 7 : On vérifie que les villes x et y ne sont pas reliées par une route (ligne 2). Si ce n'est pas le cas, on ajoute une nouvelle route avec la mise à jour du compteur de routes `plan[0][1]` et l'ajout de y parmi les voisines de x (ligne 4) et symétriquement, l'ajout de x parmi les voisines de y (ligne 5).

```
1  def ajouteRoute(plan, x, y):
2      if (y != x) and not(estVoisine(plan,x,y)):
3          plan[0][1] += 1
4          ajouteDansListe(plan[x],y)
5          ajouteDansListe(plan[y],x)
```

Il n'y a pas de risque de dépassement si les paramètres x et y correspondent bien à des villes du plan. Une ville d'un plan à n villes ayant au plus $(n-1)$ voisines, pour chaque x dans $[\![n]\!]$, la liste `plan`$[x]$ a le nombre juste suffisant de cases.

Question 8 : On commence par afficher le nombre de routes (ligne 2). Puis, dans la double boucle des lignes 4 à 7, on affiche les différents routes. La route `(i-j)` n'est affichée que si $i < j$ ce qui évite les doubles affichages.

```
1  def afficheToutesLesRoutes(plan):
2      affiche("Ce plan contient ",plan[0][1]," route(s): ")
3      n = plan[0][0]
4      for i in range(1,n+1):
5          for j in range(1,plan[i][0]+1):
6              if plan[i][j] > i:
7                  affiche("(",i,"-",plan[i][j],") ")
```

La complexité de la procédure `afficheToutesLesRoutes` est en $O(n+2m)$, on traite chacune de n villes et chaque route est explorée deux fois (mais affichée qu'une seule fois).

Partie III. Recherche de chemins arc-en-ciel

Question 9 : On récupère le nombre de villes (ligne 2). On affecte les couleurs spéciales 0 et $k+1$ aux villes de départ et d'arrivée (lignes 3 et 4). Dans la boucle des lignes 5 à 7, on affecte une couleur aléatoire entre 1 et k à toutes les villes distinctes des villes de départ et d'arrivée.

```
1  def coloriageAleatoire(plan, couleur, k, s, t):
2      n = plan[0][0]
3      couleur[s] = 0
4      couleur[t] = k+1
5      for i in range(1,n+1):
6          if i != s and i !=t:
7              couleur[i] = entierAleatoire(k)
```

Question 10 : On commence par récupérer le nombre de villes (ligne 2), créer une liste vide (ligne 3) et récupérer le nombre de voisins de la ville i (ligne 4). Dans la boucle des lignes 5 à 7, on étudie chaque ville voisine de la ville i. Si sa couleur est la couleur c, on l'ajoute à la liste résultat (ligne 8).

```
1  def voisinesDeCouleur(plan, couleur, i, c):
2      n = plan[0][0]
3      voisines = creerListeVide(n)
4      nb_voisins = plan[i][0]
5      for j in range(1,nb_voisins+1):
6          voisin = plan[i][j]
7          if couleur[voisin] == c:
8              ajouteDansListe(voisines,voisin)
9      return(voisines)
```

Question 11 : On commence par récupérer le nombre de villes (ligne 2), créer une liste vide (ligne 3) et récupérer le nombre de villes présentes dans la liste `liste` (ligne 4). Dans la boucle des lignes 5 à 10, on étudie chaque ville de la liste. Pour chaque ville de la liste `liste`, dans les boucle des lignes 8 à 10, on ajoute les voisines de couleur `c` dans la liste `voisines`. À la sortie de la boucle, la liste `voisines` contient toutes les villes de couleur `c` voisines d'une des villes présente dans la liste sans redondance `liste` dans le plan `plan` colorié par le tableau `couleur`.

```
1  def voisinesDeLaListeDeCouleur(plan, couleur, liste, c):
2      n = plan[0][0]
3      voisines = creerListeVide(n)
4      p = liste[0]
5      for k in range(1,p+1):
6          i = liste[k]
7          nb_voisins = plan[i][0]
8          for j in range(1,nb_voisins+1):
9              if couleur[plan[i][j]] == c:
10                 ajouteDansListe(voisines,plan[i][j])
11     return(voisines)
```

Dans le pire des cas, toutes les villes sont présentes dans la liste `liste` ($p = n$), on parcourt n fois la boucle des lignes 5 à 10. Chaque ville a au plus m voisines et le nombre d'appels à la fonction `ajouteDansListe` ne peut pas excéder $2m$. La complexité de la fonction `ajouteDansListe` étant en $O(n)$, on en déduit une complexité globale en $O(n+nm)$.

Question 12 On commence par vérifier que la ville `s` est de couleur 0 et la ville `t` est de couleur $k+1$, sinon on retourne `False` (lignes 2 et 3). On crée une liste vide dans laquelle on ajoute `s` (lignes 5 et 6). Dans la boucle conditionnelle des lignes 8 à 10, on calcule de proche en proche la liste des villes de couleur c situées à une distance c de la ville s. On quitte la boucle si la liste est vide ou si on a atteint une ville de couleur $k+1$, c'est-à-dire la ville t si le tableau des couleurs est correct. En sortie de boucle (ligne 11), il suffit de retourner le booléen caractérisant la présence de t dans la liste `liste`.

```
1  def existeCheminArcEnCiel(plan, couleur, k, s, t):
2      if couleur[s] != 0 or couleur[t] != k+1:
3          return(False)
4      n = plan[0][0]
5      liste = creerListeVide(n)
6      ajouteDansListe(liste,s)
7      c = 0
8      while c < k+1 and liste[0] != 0:
9          c += 1
10         liste = voisinesDeLaListeDeCouleur(plan,couleur,liste,c)
11     return(estDansListe(liste,t))
```

Dans le pire des cas, on fait $k+1$ appels à la fonction `voisinesDeLaListeDeCouleur`. La complexité est en $O((k+1)(n+nm))$, les autres opérations étant négligeables.

Partie IV. Recherche de chemin passant par exactement k villes intermédiaires distinctes

Question 13 : On commence par créer un tableau des couleurs (ligne 2). On exécute la boucle conditionnelle des lignes 5 à 8 tant que l'on n'a pas prouvé l'existence d'un chemin de s à t passant par exactement k villes intermédiaires et que l'on n'est pas passé plus de k^k fois dans la boucle. Le résultat retourné est le booléen `True` si on a l'existence d'un tel chemin avec une probabilité au moins égale à $1-1/e$.

```
1  def existeCheminSimple(plan, k, s, t):
2      couleur = creerTableau(plan[0][0]+1)
3      resultat = False
4      compteur = 0
5      while not(resultat) and (compteur < k**k):
6          compteur += 1
7          coloriageAleatoire(plan,couleur,k,s,t)
8          resultat = existeCheminArcEnCiel(plan, couleur, k, s, t)
9      return(resultat)
```

La complexité de la fonction `coloriage` est négligeable devant celle de la fonction `existeCheminArcEnCiel`. Dans le pire des cas, on passe k^k fois dans la boucle, la complexité est en $O(k^k.(k+1).(n+nm)) = O(f(k)\times g(n,m))$ avec $f(k)=(k+1)k^k$ et $g(n,m)=(n+nm)$.

Question 14 : L'idée est de faire appel à un tableau auxiliaire `predecesseur` en modifiant la fonction `voisinesDeLaListeDeCouleur` qui pour toute ville de couleur $i+1$ accessible par un chemin depuis la ville de départ s indique quelle est la ville de couleur i qui précède la ville de couleur $(i+1)$.
Si un chemin de s à t existe, on reconstitue le chemin à l'envers, en partant de t puis en remontant vers s grâce à la suite des prédécesseurs.

```
1  def voisinesDeLaListeDeCouleur(plan, couleur, liste, c, predecesseur):
2      n = plan[0][0]
3      voisines = creerListeVide(n)
4      p = liste[0]
5      for k in range(1,p+1):
6          i = liste[k]
7          nb_voisins = plan[i][0]
8          for k in range(1,nb_voisins+1):
9              j = plan[i][k]
10             if couleur[j] == c:
11                 ajouteDansListe(voisines,j)
12                 predecesseur[j] = i
13     return(voisines)
14
15 def existeCheminArcEnCiel(plan, couleur, k, s, t):
16     if couleur[s] != 0 or couleur[t] != k+1:
17         return(False)
18     n = plan[0][0]
19     liste = creerListeVide(n)
20     ajouteDansListe(liste,s)
21     predecesseur = creeTableau(n+1)
22     c = 0
23     while c < k+1 and liste[0] != 0:
24         c += 1
25         liste = voisinesDeLaListeDeCouleur(plan,couleur,
26                                            liste,c,predecesseur)
27     if estDansListe(liste,t):
28         chemin = [t]
29         pt = t
30         for i in range(k+1):
31             pt = predecesseur[pt]
32             chemin.append(pt)
33         chemin.reverse()
34         print(chemin)
35         return(True)
36     else:
37         return(False)
```