Corrigé : quand la taille n'est pas un problème (X-ENS psi 2015)

Partie I. Préliminaires : listes sans redondance

Question 1.

```
def creerListeVide(n):
    lst = creerTableau(n+1)
    lst[0] = 0
    return lst
```

Question 2.

```
def estDansListe(lst, x):
    for i in range(1, lst[0]+1):
        if lst[i] == x:
            return True
    return False
```

La complexité temporelle de cette fonction est un $O(n)$, où n désigne le nombre maximal d'éléments dans la liste.

Question 3.

```
def ajouteDansListe(lst, x):
    if not estDansListe(lst, x):
        lst[0] += 1
        lst[lst[0]] = x
```

Lorsque la liste est pleine initialement et que l'élément x n'y figure pas, la dernière ligne de cette procédure provoque une erreur car la case destinée à recevoir la valeur x n'existe pas.
L'assignation d'une valeur à une case du tableau est de coût constant donc le coût temporel de cette fonction est celui de la fonction `estDansListe`, à savoir $O(n)$.

Partie II. Création et manipulation de plans

Question 4. Les deux plans dessinés sont respectivement représentés par les tableaux :

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

et

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

Question 5.

```
def creerPlanSansRoute(n):
    lst = creerListeVide(n)
    lst[0] = [n, 0]
    for i in range(1, n+1):
        lst[i] = creerListeVide(n-1)
    return lst
```

Question 6.

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

Question 7.

```
def ajouteRoute(plan, x, y):
    if not estVoisine(plan, x, y):
        plan[0][1] += 1
        ajouteDansListe(plan[x], y)
        ajouteDansListe(plan[y], x)
```

Il n'y a ici aucun risque de débordement puisque chaque ville ne peut être reliée qu'à au plus $n-1$ autres villes.

Question 8. Pour ne faire apparaître chaque route qu'une seule fois on convient de n'afficher la route reliant les villes x et y que lorsque $x < y$.

```
def afficheToutesLesRoutes(plan):
    n, m = plan[0]
    affiche('Ce plan contient ', m, ' route(s) :')
    for x in range(1, n+1):
        for i in range(1, plan[x][0]+1):
            y = plan[x][i]
            if x < y:
                affiche(' (', x, '-', y, ')')
    affiche('\n')
```

Les listes des voisins de chacune des villes vont être parcourues une fois chacune. Il y a n listes à parcourir et la somme des longueurs de celles-ci est égale à $2m$ donc le coût de cette fonction est un $O(n+m)$.

Partie III. Recherche de chemins arc-en-ciel

Question 9.

```
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.

```
def voisinesDeCouleur(plan, couleur, i, c):
    lst = creerListeVide(plan[0][0])
    for j in range(1, plan[i][0]+1):
        if couleur[plan[i][j]] == c:
            ajouteDansListe(lst, plan[i][j])
    return lst
```

Question 11.

```
def voisinesDeLaListeDeCouleur(plan, couleur, liste, c):
    lst = creerListeVide(plan[0][0])
    for i in range(1, liste[0]+1):
        for j in range(1, plan[liste[i]][0]+1):
            if couleur[plan[liste[i]][j]] == c:
                ajouteDansListe(lst, plan[liste[i]][j])
    return lst
```

Cette fonction parcourt chacune des listes de voisins des villes présentes dans `liste`. Celle-ci contient au maximum n villes et la somme des longueurs des listes des voisins est majorée par $2m$. Enfin, pour chaque voisin considéré il faut déterminer sa couleur (ce qui se fait en coût constant) puis le cas échéant l'ajouter dans `lst`, ce qui a un coût en $O(n)$. Le coût total de cette fonction est donc un $O(n(n+m))$.

Question 12. Rappelons que seule la ville t est de couleur $k+1$, ce qui permet de rédiger la fonction demandée de la manière suivante :

```
def existeCheminArcEnCiel(plan, couleur, k, s, t):
    liste = creerListeVide(plan[0][0])
    ajouteDansListe(liste, s)
    for c in range(1, k+2):
        liste = voisinesDeLaListeDeCouleur(plan, couleur, liste, c)
        if liste[0] == 0:
            return False
    return True
```

La création de `liste` est un $O(n)$; ensuite, pour chaque valeur de c dans l'intervalle $[\![1, k+1]\!]$ on applique la fonction `voisinesDeLaListeDeCouleur`, de coût $O(n(n+m))$. Le coût dans le pire des cas est donc un $O(kn(n+m))$.

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

Question 13.

```
def existeCheminSimple(plan, k, s, t):
    couleur = creerListeVide(plan[0][0])
    for _ in range(k**k):
        coloriageAleatoire(plan, couleur, k, s, t)
        if existeCheminArcEnCiel(plan, couleur, k, s, t):
            return True
    return False
```

La création du tableau `couleur` a un coût en $O(n)$. La fonction `coloriageAleatoire` a un coût en $O(n)$ et la fonction `existeCheminArcEnCiel` un coût en $O(kn(m+n))$. La fonction `existeCheminSimple` a donc un coût en $O(k^{k+1}n(m+n))$, conforme à l'objectif du problème.

Question 14. Pour renvoyer un chemin détecté avec succès, on peut modifier la fonction `existeCheminArcEnCiel` pour mémoriser le chemin arc-en-ciel, s'il en existe. Pour cela, on modifie la fonction `voisinesDeLaListeDeCouleur` : celle-ci prend maintenant en argument une liste représentant non plus un ensemble de sommets mais un ensemble de chemins (représenté par le type *list*), et pour chacun d'eux rajoute un sommet de couleur c si cela s'avère possible :

```
def voisinesDeLaListeDeCouleur2(plan, couleur, liste, c):
    lst = creerListeVide(plan[0][0])
    for i in range(1, liste[0]+1):
        for j in range(1, plan[liste[i][-1]][0]+1):
            if couleur[plan[liste[i][-1]][j]] == c:
                ajouteDansListe(lst, liste[i] + [plan[liste[i][-1]][j]])
    return lst
```

On modifie ensuite la fonction `existeCheminArcEnCiel` en conséquence, pour retourner un couple formé d'un booléen et d'une solution, s'il en existe :

```
def existeCheminArcEnCiel2(plan, couleur, k, s, t):
    liste = creerListeVide(plan[0][0])
    ajouteDansListe(liste, [s])
    for c in range(1, k+2):
        liste = voisinesDeLaListeDeCouleur2(plan, couleur, liste, c)
        if liste[0] == 0:
            return False, []
    return True, liste[1]
```

Il reste alors à modifier légèrement la fonction principale :

```
def existeCheminSimple2(plan, k, s, t):
    couleur = creerListeVide(plan[0][0])
    for _ in range(k**k):
        coloriageAleatoire(plan, couleur, k, s, t)
        r, lst = existeCheminArcEnCiel2(plan, couleur, k, s, t)
        if r:
            return lst
    return False
```

♦ ♦

♦