X 2015 PSI-PT - Correction

Une remarque pour commencer : il faut bien comprendre que la notion de ***tableau*** introduite en début d'énoncé interdit l'usage des listes habituelles de Python.
Tout doit être fait à partir des fonctions données dans le préambule.
Il y a une bonne raison à cela : on veut faire un calcul de complexité sur les manipulations de listes, et pour cela il faut connaître la complexité des opérations élémentaires sur les listes. Or ***ceci n'est pas au programme*** - et est assez compliqué et flou.
On vous fait donc ***reprogrammer la structure de liste*** afin de pouvoir évaluer correctement la complexité des divers algorithmes.

1. ```python
   def creerListeVide(n):
       a = creerTableau(n+1)
       a[0] = 0
       return a
   ```

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

   Chaque opération s'effectue en temps constant à l'exception de la boucle : dans le pire des cas, l'élément `x` ne se trouve pas dans `liste` et la complexité de la fonction est donc en $\mathcal{O}(n)$.

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

   Si la liste est pleine initialement, le code provoque une erreur puisqu'il cherche à affecter une valeur à un élément inexistant de la liste.

   La complexité est celle de `estDansListe`, les autres opérations s'effectuant en temps constant : $\mathcal{O}(n)$ dans le pire des cas.

4. ```python
   plan1 = [[5,7],
            [2,2,3,0,0],
            [3,1,3,5,0],
            [4,1,2,4,5],
            [2,3,5,0,0],
            [3,2,3,4,0]]

   plan2 = [[5,4],
            [1,2,0,0,0],
            [3,1,4,3,0],
            [1,2,0,0,0],
            [2,2,5,0,0],
            [1,4,0,0,0]]
   ```

5. ```
def creerPlanSansRoute(n):
    res = creerTableau(n+1)
    el = creerTableau(2)
    el[0] = n
    el[1] = 0
    res[0] = el
    for i in range(n):
        el = creerListeVide(n)
        res[i+1] = el
    return res
```

6. ```
def estVoisine(plan, x, y):
    return estDansListe(plan[x],y)
```

7. ```
def ajouteRoute(plan, x, y):
    m = plan[0][1]
    if not estVoisine(plan,x,y):
        plan[0][1]=m+1
        ajouteDansListe(plan[x],y)
        ajouteDansListe(plan[y],x)
```

   Non, il n'y a aucun risque de dépassement de la capacité des listes puisqu'elles sont préformatées à la longueur maximale qu'elles pourraient avoir si toutes les villes étaient reliées par des routes.

8. ```
def afficheToutesLesRoutes(plan):
    m = plan[0][1]
    nbaffichees = 0
    aAfficher = 'Ce plan contient '+str(m)+' routes : '
    ville = 1
    while nbaffichees<m:
        for i in range(0,plan[ville][0]):
            j = plan[ville][i+1]
            if j>ville:
                aAfficher += '('+str(ville)+'-'+str(j)+') '
                nbaffichees += 1
        ville += 1
    affiche(aAfficher)
```

   Toutes les opérations sont en temps constant à l'exception des deux boucles :
   - la boucle `while` s'effectue m fois ;
   - la boucle `for`, intérieure, s'effectue au pire $n-1$ fois, puisqu'une ville peut, au pire, être reliée à toutes les autres.

   Donc la complexité de la fonction est en $\mathcal{O}(mn)$.

9. ```
def coloriageAleatoire(plan, couleur, k, s, t):
    n = plan[0][0]
    couleur[s] = 0
    couleur[t] = k+1
    for i in range(n):
        if i+1!=s and i+1!=t:
            couleur[i+1] = entierAleatoire(k)
```

10. ```
def voisinesDeCouleur(plan, couleur, i, c):
    n = plan[i][0]
    L = creerListeVide(n)
```

```python
    for k in plan[i][1:n+1]:
        if couleur[k]==c:
            ajouteDansListe(L,k)
    return L
```

11.
```python
def voisinesDeLaListeDeCouleur(plan, couleur, liste, c):
    nb = liste[0]
    n= plan[0][0]
    L = creerListeVide(n)
    for i in liste[1:nb+1]:
        p = plan[i][0]
        for k in plan[i][1:p+1]:
            if couleur[k]==c:
                ajouteDansListe(L,k)
    return L
```

Toutes les opérations sont en temps constant à l'exception de :
- `creerListeVide(n)`, probablement de complexité $\mathcal{O}(n)$ (ce n'est pas précisé par l'énoncé) ;
- `ajouteDansListe(L,k)` de complexité $\mathcal{O}(n)$ d'après la question 3 ;
- les deux boucles imbriquées dont la plus externe s'effectue `len(liste)` fois et celle interne qui s'effectue au pire m fois.

Il y a donc ambiguïté dans la question : doit-on évaluer la longueur de `len(liste)` en fonction de n et m ?
La question suivante laisse penser qu'il ne faut pas le faire, auquel cas la complexité de cette fonction est en $\mathcal{O}(mn \times len(liste))$.
Si au contraire il faut le faire, alors `len(liste)` est dans le pire des cas égal à n et la complexité est en $\mathcal{O}(mn^2)$.

12.
```python
def existeCheminArcEnCiel(plan, couleur, k, s, t):
    L = voisinesDeCouleur(plan,couleur,s,1)
    i=2
    while L[0]>0 and i<=k:
        L = voisinesDeLaListeDeCouleur(plan, couleur, L, i)
        i=i+1
    L = voisinesDeLaListeDeCouleur(plan, couleur, L, k+1)
    return L[0]>0
```

La boucle `while` s'effectue k fois, et pour chaque pas de cette boucle,
la complexité de `voisinesDeLaListeDeCouleur` est en $\mathcal{O}\big(mn \times len(L[k])\big)$.
La complexité de la présente fonction est donc en $\sum_k \mathcal{O}\big(mn \times len(L[k])\big) = \mathcal{O}\big(mn^2\big)$.

La complexité de `existeCheminArcEnCiel` est donc en $\mathcal{O}(mn^2k)$.

13.
```python
def existeCheminSimple(plan, k, s, t):
    n = plan[0][0]
    couleur = creerTableau(n+1)
    for i in range(k**k):
        coloriageAleatoire(plan, couleur, k, s, t)
        if existeCheminArcEnCiel(plan,couleur,k,s,t):
            return True
    return False
```

Dans le pire des cas, aucun chemin n'existe et la boucle `for` s'effectue k^k fois. La complexité de la fonction `existeCheminArcEnCiel` est en $\mathcal{O}(mn^2k)$ et les autres complexités sont négligeables devant celle-ci, donc la complexité de la présente fonction est en $\mathcal{O}\left(mn^2k^{k+1}\right)$.

14. Pour renvoyer un chemin lorsqu'il existe, il suffit de modifier `existeCheminArcEnCiel` pour qu'elle renvoie `False` lorsque le chemin n'existe pas, et le chemin lorsqu'il existe. On modifie alors aussi `existeCheminSimple` pour qu'elle renvoie ce chemin en cas d'existence.