Correction du DS1

(d'après X/ENS PSI 2015)

1. On teste les éléments de `list`, un à un :

```
def estDansListe(liste,x) :
    for k in liste :
        if x == k :
            return True
    return False
```

On exécute au pire, un seul parcours de la liste donc la complexité est en $\boxed{O(n)}$

2. On utilise la fonction précédente avant d'ajouter x à la liste si besoin :

```
def ajouteDansListe(liste,x) :
    if not estDansListe(liste,x) :
        liste.append(x)
```

La complexité est à nouveau $\boxed{O(n)}$ à cause de l'utilisation de la fonction `estDansListe`.

3.
```
plan1 = {1:[2,3],2:[1,3,5],3:[1,2,4,5],4:[3,5],5:[2,3,4]}
plan2 = {1:[2],2:[1,3,4],3:[2],4:[2,5],5:[4]}
```

4. On crée un dictionnaire avec les entiers de $[\![1,n]\!]$ comme clefs, associées à des listes vides :

```
def creerPlanSansRoute(n) :
    return {k : [] for k in range(1,n+1)}
```

5. On teste si y est dans la liste des villes voisines de x, avec la fonction `estDansListe` :

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

6. On complète si besoin, grâce à la fonction `ajouteDansListe` pour éviter les répétitions, les deux listes d'adjacence pour que le graphe reste non orienté :

```
def ajouteRoute(plan,x,y) :
    ajouteDansListe(plan[x],y)
    ajouteDansListe(plan[y],x)
```

7. Pour chaque ville i, on compte le nombre de routes qui partent de i puis on divise le total par 2 (division entière) puisque chaque route est prise en compte deux fois dans le calcul précédent :

```
def nbRoutes(plan) :
    m = 0
    for i in plan :
        m += len(plan[i])
    return m//2
```

8. On remplit la liste `couleur` case par case, en mettant les valeurs particulières pour s et t :

```
def coloriageAleatoire(plan,k,s,t) :
    n = len(plan)
    couleur = [0]*(n+1)
    for i in plan :
        couleur[i] = randint(1,k+1)
    couleur[s] = 0
    couleur[t] = k+1
    return couleur
```

9. Avec une seule ville i, il est inutile d'utiliser la fonction `ajouteDansListe` pour éviter les répétitions puisque la liste des villes voisines de i est déjà sans répétition :

```
def voisinesDeCouleur(plan,couleur,i,c) :
    L =[]
    for k in plan[i] :
        if couleur[k] == c :
            L.append(k)
    return L
```

10. Il s'agit de faire la réunion des listes fournies par la fonction précédente, pour chaque élément de la liste `liste`. Cette fois, il est indispensable d'éviter les répétitions possibles :

```
def voisinesDeLaListeDeCouleur(plan,couleur,liste,c) :
    L = []
    for i in liste :
        for k in voisinesDeCouleur(plan,couleur,i,c) :
            ajouteDansListe(L,k)
    return L
```

La liste `liste` possède au plus n éléments ; pour chaque élément on exécute la fonction `voisineDeCouleur` de complexité $O(m)$ puisque de i partent au plus m routes. Puis pour chaque valeur de `k`, on exécute la fonction `ajouteDansListe` de complexité $O(n)$. On a donc une complexité en $O(n(m+mn)) = O(mn^2)$ à priori.
Une étude plus fine donne un résultat plus précis : en fait les deux premières boucles parcourent au maximum toutes les routes (dans les deux sens), elles donnent donc au plus $2m$ itérations ; on va cependant utiliser n fois la fonction `voisineDeCouleur` mais la somme des complexités de ces utilisation sera en $O(n+m)$ (ou $\max\{n,m\}$). La complexité de `voisinesDeLaListeDeCouleur` est donc en $\boxed{O(n(n+m))}$ (ce qui est plus conforme à l'objectif).

11. On applique la fonction précédente à partir de la liste `[s]` et si au cours des $k+2$ itérations, on n'a pas de ville de la bonne couleur, il n'existe pas de chemin arc en ciel ; comme `t` est la seule ville de couleur $k+1$, la dernière itération ne peut rajouter que la ville `t` (donc il est inutile de tester si `t` fait effectivement partie des villes ajoutées à la dernière itération).

```
def existeCheminArcEnCiel(plan,couleur,k,s,t) :
    v = [s]
    for j in range(1,k+2) :
        v = voisinesDeLaListeDeCouleur(plan,couleur,v,j)
        if len(v) == 0 :
            return False
    return True
```

On applique $k+1$ fois la fonction précédente donc la complexité est $\boxed{O(kn(n+m))}$

12. D'après le calcul de probabilité admis par le texte, il suffit d'appliquer k^k fois l'algorithme précédent, en changeant le coloriage du plan : complexité en $\boxed{O\left(k^{k+1}n(n+m)\right)}$

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

13. Une solution est de modifier les fonctions de façon à ce que les listes ne mémorisent pas que les villes par lesquelles on passe mais tout le chemin depuis la ville s, on construit donc des listes de listes :

```
def voisinesDeLaListeDeCouleur2(plan,couleur,liste,c) :
    L =[]
    for l in liste :            # l est une liste : le début de chemin
        for k in plan[l[-1]] :  # les voisins de la dernière étape
            if couleur[k] == c :
                ajouteDansListe(L,l.append(k))
    return L

def existeCheminArcEnCiel2(plan,couleur,k,s,t) :
    v = [s]
    for j in range(1,k+2) :
        v = voisinesDeLaListeDeCouleur2(plan,couleur,v,j)
        if len(v) == 0 : return False
    return v[0]          # on renvoie un des chemins

def existeCheminSimple2(plan,k,s,t) :
    for _ in range(k**k) :
        couleur = coloriageAleatoire(plan,k,s,t)
        r = existeCheminArcEnCiel2(plan,couleur,k,s,t) # pas tjs un booléen
        if r != False : return r
    return False
```