ÉCOLE POLYTECHNIQUE — ÉCOLES NORMALES SUPÉRIEURES

CONCOURS D'ADMISSION 2015 **FILIÈRES PSI et PT**

ÉPREUVE D'INFORMATIQUE (XCR)

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation sera **obligatoirement** Python.

⋆ ⋆ ⋆

Quand la taille n'est pas un problème

Notations. On désignera par $[\![n]\!]$ l'ensemble des entiers de 1 à n : $[\![n]\!] = \{1, \ldots, n\}$.

Objectif. Le but de cette épreuve est de décider s'il existe, entre deux villes données, un chemin passant par exactement k villes intermédiaires *distinctes*, dans un plan contenant au total n villes reliées par m routes. L'algorithme d'exploration naturel s'exécute en temps $O(n^k m)$. L'objectif est d'obtenir un algorithme qui s'exécute en un temps $O(f(k) \times n(n+m))$, qui croit polynomialement en la taille $(n+m)$ du problème quelle que soit la valeur de k demandée.

Complexité. La complexité, ou le temps d'exécution, d'un programme Π (fonction ou procédure) est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc...) nécessaires à l'exécution de Π. Lorsque cette complexité dépend de plusieurs paramètres n, m et k, on dira que Π a une complexité en $O(\phi(n,m,k))$, lorsqu'il existe quatre constantes absolues A, n_0, m_0 et k_0 telles que la complexité de Π soit inférieure ou égale à $A \times \phi(n,m,k)$, pour tout $n \geqslant n_0$, $m \geqslant m_0$ et $k \geqslant k_0$.

Lorsqu'il est demandé de préciser la complexité d'un programme, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Implémentation. On suppose que l'on dispose d'une fonction `creerTableau(n)` qui alloue un tableau de taille `n` indexé de 0 à `n` − 1 (les valeurs contenues dans le tableau initialement sont arbitraires). L'instruction `b = creerTableau(n)` créera un tableau de taille `n` dans la variable `b`.

On pourra ainsi créer un tableau `a` de p tableaux de taille q par la suite d'instructions suivante :

```
a = creerTableau(p)
for i in range(p):
    a[i] = creerTableau(q)
```

On accédera par l'instruction `a[i][j]` à la `j`-ème case du `i`-ème tableau contenu dans le tableau `a` ainsi créé. Par exemple, la suite d'instructions suivante remplit le tableau `a` avec les sommes des indices `i` et `j` de chaque case :

```
for i in range(p):
    for j in range(q):
        a[i][j] = i+j
```

On supposera l'existence de deux valeurs booléennes `True` et `False`.

On supposera l'existence d'une procédure : `affiche(...)` qui affiche le contenu de ses arguments à l'écran. Par exemple, `x = 1; y = x+1; affiche("x = ",x," et y = ",y);` affiche à l'écran :

```
x = 1 et y = 2
```

Dans la suite, nous distinguerons *fonction* et *procédure* : les fonctions renvoient une valeur (ou un tableau) tandis que les procédures ne renvoient aucune valeur.

La plus grande importance est donnée à la lisibilité du code produit par les candidats. Ainsi, les candidats sont encouragés à introduire des procédures ou fonctions intermédiaires lorsque cela en simplifie l'écriture.

Partie I. Préliminaires : Listes sans redondance

On souhaite stocker en mémoire une liste *non-ordonnée* d'au plus n entiers *sans redondance* (i.e. où aucun entier n'apparaît plusieurs fois). Nous nous proposons d'utiliser un tableau `liste` de longueur $n+1$ tel que :

— `liste[0]` contient le nombre d'éléments dans la liste

— `liste[i]` contient le `i`-ème élément de la liste (non-ordonnée) pour $1 \leqslant$ `i` $\leqslant$ `liste[0]`

Question 1. Écrire une fonction `creerListeVide(n)` qui crée, initialise et renvoie un tableau de longueur `n` $+ 1$ correspondant à la liste vide ayant une capacité de `n` éléments.

Question 2. Écrire une fonction `estDansListe(liste, x)` qui renvoie `True` si l'élément `x` apparaît dans la liste représentée par le tableau `liste`, et renvoie `False` sinon.

Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre

maximal n d'éléments dans la liste ?

Question 3. Écrire une procédure `ajouteDansListe(liste, x)` qui modifie de façon appropriée le tableau `liste` pour y ajouter `x` si l'entier `x` n'appartient pas déjà à la liste, et ne fait rien sinon.

Quel est le comportement de votre procédure si la liste est pleine initialement ? (On ne demande pas de traiter ce cas)

Quelle est la complexité en temps de votre procédure dans le pire cas en fonction du nombre maximal n d'éléments dans la liste ?

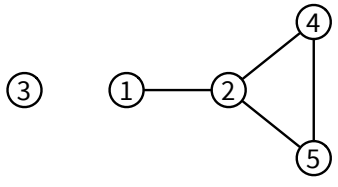
Partie II. Création et manipulation de plans

Un *plan* P est défini par : un ensemble de n villes numérotées de 1 à n et un ensemble de m routes (toutes à double-sens) reliant chacune deux villes ensemble. On dira que deux villes $x, y \in [\![n]\!]$ sont *voisines* lorsqu'elles sont reliées par une route, ce que l'on notera par $x \sim y$. On appellera *chemin* de longueur k toute suite de villes $v_1, \ldots, v_k$ telle que $v_1 \sim v_2 \sim \cdots \sim v_k$. On représentera les villes d'un plan par des ronds contenant leur numéro et les routes par des traits reliant les villes voisines (voir Fig. 1).

Structure de données. Nous représenterons tout plan P à n villes par un tableau `plan` de $(n+1)$ tableaux où :

- `plan[0]` contient un tableau à deux éléments où :
  - `plan[0][0] =` n contient le nombre de villes du plan
  - `plan[0][1] =` m contient le nombre de routes du plan
- Pour chaque ville `x` $\in [\![n]\!]$, `plan[x]` contient un tableau à n éléments représentant la liste à au plus $n-1$ éléments des villes voisines de la ville `x` dans P dans un ordre arbitraire en utilisant la structure de liste sans redondance définie dans la partie précédente. Ainsi :
  - `plan[x][0]` contient le nombre de villes voisines de `x`
  - `plan[x][1]`, ..., `plan[x][plan[x][0]]` sont les indices des villes voisines de x.

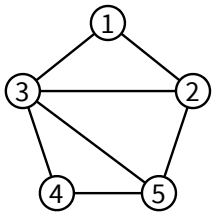
La figure 1 donne un exemple de plan et d'une représentation possible sous la forme de tableau de tableaux (les `*` représentent les valeurs non-utilisées des tableaux).



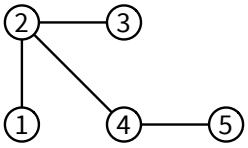

```
plan = [ [5  4],
         [1  2   *   *   *],
         [3  4   1   5   *],
         [0  *   *   *   *],
         [2  2   5   *   *],
         [2  4   2   *   *] ]
```

FIGURE 1 – Un plan à 5 villes et 4 routes et une représentation possible en mémoire sous forme d'un tableau de tableaux `plan`.

Question 4. Représenter sous forme de tableaux de tableaux les deux plans suivants :



Plan 1



Plan 2

On pourra utiliser dans la suite, les fonctions et procédures de gestion de listes définies dans la partie précédente.

Question 5. Écrire une fonction `creerPlanSansRoute(n)` qui crée, remplit et renvoie le tableau de tableaux correspondant au plan à `n` villes n'ayant aucune route.

Question 6. Écrire une fonction `estVoisine(plan, x, y)` qui renvoie `True` si les villes `x` et `y` sont voisines dans le plan codé par le tableau de tableaux `plan`, et renvoie `False` sinon.

Question 7. Écrire une procédure `ajouteRoute(plan, x, y)` qui modifie le tableau de tableaux `plan` pour ajouter une route entre les villes `x` et `y` si elle n'était pas déjà présente et ne fait rien sinon. (On prendra garde à bien mettre à jour toutes les cases concernées dans le tableau de tableaux `plan`.)

Y a-t-il un risque de dépassement de la capacité des listes ?

Question 8. Écrire une procédure `afficheToutesLesRoutes(plan)` qui affiche à l'écran la liste des routes du plan codé par le tableau de tableaux `plan` où chaque route apparaît exactement une seule fois. Par exemple, pour le graphe codé par le tableau de tableaux de la figure 1, votre procédure pourra afficher à l'écran :

```
Ce plan contient 4 route(s): (1-2) (2-4) (2-5) (4-5)
```

Quelle est la complexité en temps de votre procédure dans le pire cas en fonction de n et m ?

Partie III. Recherche de chemins arc-en-ciel

Étant données deux villes distinctes s et $t \in [\![n]\!]$, nous recherchons un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes. L'objectif de cette partie et de la suivante est de construire une fonction qui va détecter en temps linéaire en $n(n+m)$, l'existence d'un tel chemin avec une probabilité indépendante de la taille du plan $n+m$.

Le principe de l'algorithme est d'attribuer à chaque ville $i \in [\![n]\!] \setminus \{s,t\}$ une couleur aléatoire codée par un entier aléatoire uniforme `couleur[`i`]` $\in \{1,\dots,k\}$ stocké dans un tableau `couleur` de taille $n+1$ (la case 0 n'est pas utilisée). Les villes s et t reçoivent respectivement les couleurs spéciales 0 et $k+1$, i.e. `couleur[`s`] = 0` et `couleur[`t`] =` k`+1`. L'objectif de cette partie est d'écrire une procédure qui détermine s'il existe un chemin de longueur $k+2$ allant de s à t dont la j-ème ville intermédiaire a reçu la couleur j. Dans l'exemple de la figure 2, le chemin

$6 \sim 7 \sim 8 \sim 3 \sim 4$ de longueur $5 = k + 2$ qui relie $s = 6$ à $t = 4$ vérifie cette propriété pour $k = 3$.

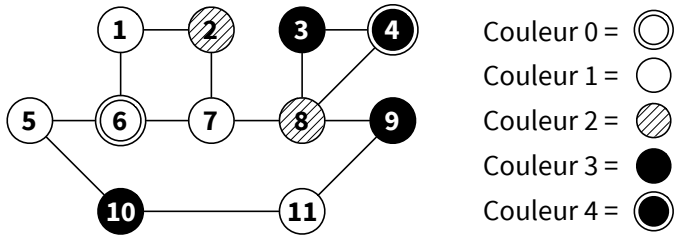


FIGURE 2 – Exemple de plan colorié pour $k = 3$, $s = 6$, $t = 4$.

On suppose l'existence d'une fonction `entierAleatoire(k)` qui renvoie un entier aléatoire uniforme entre 1 et `k` (i.e. telle que $\Pr\{$`entierAleatoire(k)` $= c\} = 1/$`k` pour tout entier $c \in \{1, \ldots,$ `k`$\}$).

Question 9. Écrire une procédure `coloriageAleatoire(plan, couleur, k, s, t)` qui prend en argument un plan de n villes, un tableau `couleur` de taille $n + 1$, un entier `k`, et deux villes s et $t \in [\![n]\!]$, et remplit le tableau `couleur` avec : une couleur aléatoire uniforme dans $\{1, \ldots,$ `k`$\}$ choisie indépendamment pour chaque ville $i \in [\![n]\!] \setminus \{s, t\}$; et les couleurs 0 et `k` $+ 1$ pour s et t respectivement.

Nous cherchons maintenant à écrire une fonction qui calcule l'ensemble des villes de couleur c voisines d'un ensemble de villes donné. Dans l'exemple de la figure 2, l'ensemble des villes de couleur 2 voisines des villes $\{1, 5, 7\}$ est $\{2, 8\}$.

Question 10. Écrire une fonction `voisinesDeCouleur(plan, couleur, i, c)` qui crée et renvoie un tableau codant la liste sans redondance des villes de couleur `c` voisines de la ville `i` dans le plan `plan` colorié par le tableau `couleur`.

Question 11. Écrire une fonction `voisinesDeLaListeDeCouleur(plan, couleur, liste, c)` qui crée et renvoie un tableau codant la liste sans redondance des 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`.

Quelle est la complexité de votre fonction dans le pire cas en fonction de n et m ?

Question 12. Écrire une fonction `existeCheminArcEnCiel(plan, couleur, k, s, t)` qui renvoie `True` s'il existe dans le plan `plan`, un chemin $s \sim v_1 \sim \cdots \sim v_k \sim t$ tel que `couleur[`v_j`]` `=` j pour tout $j \in \{1, \ldots, k\}$; et renvoie `False` sinon.

Quelle est la complexité de votre fonction dans le pire cas en fonction de k, n et m ?

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

Si les couleurs des villes sont choisies aléatoirement et uniformément dans $\{1, \ldots, k\}$, la probabilité que j soit la couleur de la j-ème ville d'une suite fixée de k villes, vaut $1/k$ indépendamment pour tout j. Ainsi, étant données deux villes distinctes s et $t \in [\![n]\!]$, s'il existe dans le plan `plan` un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes et si le coloriage `couleur` est choisi aléatoirement conformément à la procédure `coloriageAleatoire(plan, couleur,` k`,` s`,` t`)`, la procédure `existeCheminArcEnCiel(plan, couleur,` k`,` s`,` t`)` répond `True` avec probabilité au moins k^{-k} ; et répond toujours `False` sinon. Ainsi, si un tel chemin existe, la probabilité qu'une parmi k^k exécutions indépendantes de `existeCheminArcEnCiel` réponde `True` est supérieure ou égale à $1 - (1 - k^{-k})^{k^k} = 1 - \exp(k^k \ln(1 - k^{-k})) \geqslant 1 - 1/e > 0$ (admis).

Question 13. Écrire une fonction `existeCheminSimple(plan, k, s, t)` qui renvoie `True` avec probabilité au moins $1 - 1/e$ s'il existe un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes dans le plan `plan` ; et renvoie toujours `False` sinon.

Quelle est sa complexité en fonction de k, n et m dans le pire cas ? Exprimez-la sous la forme $O(f(k) \times g(n, m))$ pour f et g bien choisies.

Question 14. Expliquer comment modifier votre programme pour renvoyer un tel chemin s'il est détecté avec succès.

Note : L'algorithme présenté partiellement dans les parties III et IV de ce sujet est dû à Dániel Marx. Pour en savoir plus, on pourra consulter sa page web (`http://www.cs.bme.hu/~dmarx`) et plus particulièrement les transparents sur la complexité paramétrée (Part 1: Algorithmic techniques, page 66 et suivantes).

En utilisant une meilleure structure de liste sans redondance que celle proposée dans le sujet, on peut facilement obtenir une complexité qui soit linéaire en la taille $(n+m)$ du problème quelle que soit la valeur de k demandée.