Thésée

> *Si toutes les pièces du bateau de Thésée sont remplacées une par une au fil du temps, à quel moment — s'il y en a un — cesse-t-il d'être le même bateau ?*

Quand il ne s'interroge pas profondément sur l'abstrait, Thésée passe son temps libre à tuer des minotaures. Cette fois, cependant, il doit d'abord traverser un labyrinthe sombre et tortueux. Comme ce n'est pas une tâche facile, il demande l'aide d'Ariane pour le guider. Le labyrinthe peut être vu comme un graphe non orienté connexe comportant n nœuds (numérotés de 1 à n) et m arêtes, avec un nœud spécial t, où se trouve le Minotaure.

Thésée ne peut pas voir le graphe du tout, mais Ariane le peut. Elle et Thésée vont élaborer une stratégie afin qu'il puisse atteindre en toute sécurité le nœud où se trouve le Minotaure : elle apposera une étiquette valant soit 0, soit 1 sur chacune des m arêtes. Ensuite, Thésée entrera dans le labyrinthe par un nœud s qu'Ariane ne connaît pas à l'avance.

Comme il fait très sombre, à tout moment, il ne peut voir que l'indice du nœud où il se trouve, les indices des nœuds voisins, ainsi que les étiquettes des arêtes adjacentes. De plus, en raison de la nature tortueuse du labyrinthe, il ne peut **jamais se souvenir** d'aucune information concernant les nœuds qu'il a déjà visités.

Pour atteindre le Minotaure en toute sécurité, Thésée doit effectuer au maximum $min + C$ déplacements, où min est le nombre minimal d'arêtes sur le chemin allant de s à t, et C est une constante.

Détails de l'implémentation

Vous devrez implémenter deux fonctions :

```
std::vector<int> label(int n, std::vector<std::pair<int,int>> edges, int t);
```

- n : le nombre de nœuds
- $edges$: une liste de longueur m décrivant les arêtes du graphe
- t : le nœud de destination
- Cette procédure doit retourner une liste d'étiquettes de longueur m où le i-ième élément vaut soit 0, soit 1, et représente l'étiquette de la i-ième arête pour tout $0 \leq i < m$.
- Chaque arête doit obligatoirement être étiquetée avec 0 ou 1. Utiliser une autre valeur entraînera un **comportement indéfini**.
- Cette procédure est appelée **exactement une fois** pour chaque cas de test.

```
int travel(int n, int u, std::vector<std::pair<int,int>> neighbours);
```

- n : le nombre de nœuds dans le graphe
- u : le nœud actuel
- $neighbours$: une liste de paires (v, e) indiquant qu'il existe une arête entre u et v étiquetée avec e
- Cette procédure doit retourner un nœud voisin vers lequel se déplacer. Si le nœud voisin est t, le programme se termine automatiquement.
- Il est garanti que, pour tout appel à cette fonction, u ne sera jamais égal au nœud spécial t.
- Un appel à cette procédure représente un déplacement dans le labyrinthe. Ainsi, pour chaque cas de test, cette procédure peut être appelée **autant de fois que nécessaire** afin d'atteindre le nœud de destination.

Attention ! Le programme ne doit pas utiliser de variables globales ou statiques pour communiquer entre différentes instances de `label` ou `travel`. Toute tentative de contourner cette règle entraînera un **comportement indéfini**.

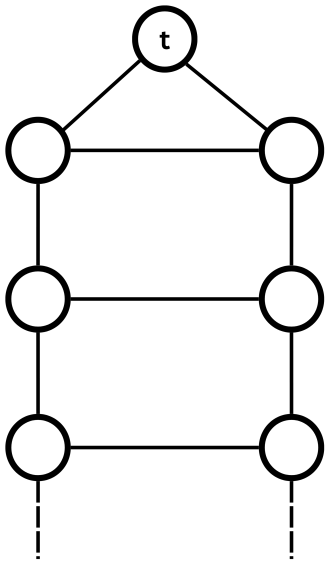
Limites

- $1 \leq n \leq 10000$
- $1 \leq m \leq 50000$
- $C = 14$
- Le nœud de départ s est fixé pour chaque test avant l'appel de la fonction `label`.

Sous-tâches

1. (4 points) Le graphe est une clique (c'est-à-dire qu'il existe une arête entre chaque paire de nœuds $1 \leq u < v \leq n$).
2. (10 points) La distance entre le nœud de destination et n'importe quel nœud du graphe est d'au plus 2 arêtes.
3. (11 points) Le graphe est un arbre.
4. (13 points) Le graphe est biparti (c'est-à-dire qu'il est possible de diviser les nœuds du graphe en deux sous-ensembles tels qu'il n'existe aucune arête entre deux nœuds d'un même sous-ensemble).
5. (12 points) Le graphe est une échelle (voir la définition ci-dessous).
6. (50 points) Aucune contrainte supplémentaire.

Remarque : Un graphe en échelle est un graphe constitué de deux chemins (ou chaînes) parallèles de même longueur, où chaque paire de nœuds correspondants est reliée par une arête, formant ainsi les barreaux de l'échelle. De plus, à une extrémité de l'échelle se trouve un nœud spécial — le nœud de destination t — qui est connecté aux deux extrémités de l'échelle, jouant effectivement le rôle de parent commun. Il est garanti que n sera impair pour tout graphe de ce type. Voir l'image ci-dessous.



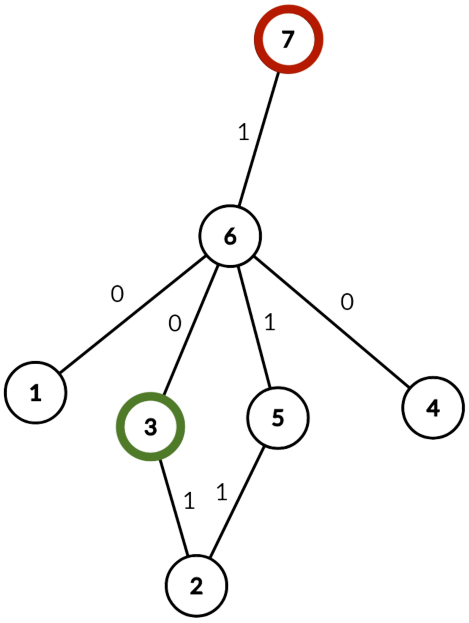
Exemples

Exemple 1

Supposons que nous ayons un graphe avec 7 nœuds et 7 arêtes (listées ci-dessous). Le nœud de départ sera 3 (marqué en **vert**) et le nœud de destination est 7 (marqué en **rouge**). Le grader appellera d'abord :

```
label(7, {{1, 6}, {7, 6}, {2, 5}, {3, 2}, {3, 6}, {6, 5}, {6, 4}}, 7)
```

Supposons que l'appel à `label` retourne `{0, 1, 1, 1, 0, 1, 0}`. Le graphe obtenu ressemblera alors à ceci :



Ce qui suit est une séquence possible d'appels à `travel` qui mènera à une solution correcte :

Appel	Valeur retournée
`travel(7, 3, {{2, 1}, {6, 0}})`	2
`travel(7, 2, {{5, 1}, {3, 1}})`	5
`travel(7, 5, {{6, 1}, {2, 1}})`	6
`travel(7, 6, {{3, 0}, {5, 1}, {1, 0}, {4, 0}, {7, 1}})`	4
`travel(7, 4, {{6, 0}})`	6
`travel(7, 6, {{3, 0}, {5, 1}, {1, 0}, {4, 0}, {7, 1}})`	7

Lorsque la valeur retournée est 7 (c'est-à-dire le nœud de destination), le programme s'arrête.

Grader d'exemple

Le grader d'exemple lit l'entrée dans le format suivant :

- ligne 1 : n m
- ligne 2 : s t
- ligne $3 + i$: a b, indiquant qu'il existe une arête reliant les nœuds a et b.

Le grader appellera d'abord `label` avec les paramètres correspondants et étiquettera les arêtes du graphe selon le vecteur retourné.

Ensuite, il appellera `travel` avec les paramètres n, s et les voisins de s. Après le premier appel, il continuera à faire des appels successifs à `travel`, avec le nœud courant étant celui retourné par l'appel précédent, jusqu'à ce que le nœud de destination t soit atteint.

À la fin, il affichera le nombre de déplacements effectués par votre solution ainsi que les nœuds parcourus dans l'ordre.