Theseus

> *Wenn alle Teile der Aufgabe Theseus im Laufe der Zeit einzeln ausgetauscht werden, ab welchem Punkt — wenn überhaupt — ist es dann nicht mehr dieselbe Aufgabe?*

Wenn er nicht gerade über Abstraktes nachdenkt, erschlägt Theseus in seiner Freizeit Minotauren. Diesmal muss er jedoch erst ein dunkles und verschlungenes Labyrinth durchqueren. Da dies kein leichtes Unterfangen ist, bittet er Ariadne um Hilfe. Das Labyrinth kann man sich als verbundenen ungerichteten Graphen mit n Knoten (beschriftet von 1 bis n) und m Kanten vorstellen, mit einem besonderen Knoten t, wo der Minotaurus sitzt.

Theseus kann den Graphen nicht sehen, aber Ariadne kann ihn sehen. Sie wird mit Theseus eine Strategie ausarbeiten, damit er den Knoten, an dem sich der Minotaurus befindet, sicher erreichen kann: Sie wird an jede der m Kanten mit entweder 0 oder 1 beschriften. Danach wird Theseus das Labyrinth durch einen Knoten s betreten, den Ariadne vorher nicht kennt.

Da es sehr dunkel ist, kann er zu jedem Zeitpunkt nur

- den Index des Knotens, in dem er sich befindet,
- die Indizes der benachbarten Knoten, und
- die Bezeichnungen der angrenzenden Kanten sehen.

Ausserdem kann er sich aufgrund der verwinkelten Beschaffenheit des Labyrinths **niemals** an Informationen über frühere Knotenpunkte, die er besucht hat, erinnern.

Um den Minotaurus sicher zu erreichen, darf Theseus sich höchstens $min + C$ mal bewegen, wobei min die minimale Anzahl von Kanten auf dem Weg von s nach t ist und C eine Konstante.

Implementierungsdetails

Implementiere die folgenden zwei Funktionen:

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

- n: die Anzahl der Knoten
- $edges$: eine `vector` der Länge m, der die Kanten des Graphen beschreibt
- t: der Zielknoten
- Diese Funktion muss einen `vector` der Länge m zurückgeben, wobei das i-te Element entweder 0 oder 1 sein kann und die Beschriftung der i-ten Kante für alle $0 \leq i < m$ darstellt.

- Jede Kante muss entweder mit 0 oder 1 beschriftet sein. Die Beschriftung mit einem anderen Wert führt zu **undefiniertem Verhalten**.
- Diese Funktion wird für jeden Testfall **exakt einmal** aufgerufen.

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

- n: die Anzahl der Knoten im Graphen
- u: der aktuelle Knoten
- $neighbours$: eine Liste von Paaren (v, e), die angeben, dass es eine Kante zwischen u und v gibt, die mit e beschriftet ist
- Diese Funktion muss einen Nachbarknoten zurückgeben, zu dem man sich bewegen kann. Falls der Nachbarknoten t ist, wird das Programm automatisch beendet.
- Es ist garantiert, dass bei jedem Aufruf dieser Funktion u nicht gleich dem besonderen Knoten t ist.
- Ein Aufruf dieser Prozedur stellt eine Bewegung in dem Labyrinth dar. Daher kann diese Prozedur für jeden Testfall **so oft wie nötig** aufgerufen werden, um den Zielknoten zu erreichen.

Achtung! Das Programm soll keine globalen/statischen Variablen verwenden, um zwischen verschiedenen Instanzen von `label` oder `travel` zu kommunizieren. Jeder Versuch, dies zu umgehen, würde zu **undefiniertem Verhalten** führen.

Beschränkungen

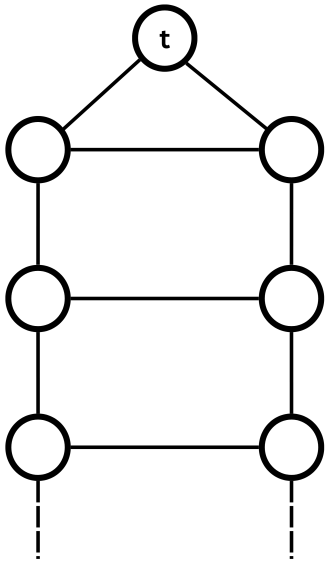
- $1 \leq n \leq 10000$
- $1 \leq m \leq 50000$
- $C = 14$
- Der Startknoten s wird für jeden Testfall festgelegt bevor `label` aufgerufen wird.

Teilaufgaben

1. (4 Punkte) Der Graph ist eine Clique (d.h. es gibt eine Kante zwischen allen Paaren von Knoten).
2. (10 Punkte) Die Entfernung zwischen dem Ziel und einem beliebigen Knoten im Graphen beträgt höchstens 2 Kanten.
3. (11 Punkte) Der Graph ist ein Baum.
4. (13 Punkte) Der Graph ist bipartit (d.h. es gibt eine Möglichkeit, die Knoten des Graphen in zwei Teilmengen zu unterteilen, so dass es keine Kante zwischen zwei Knoten aus derselben Teilmenge gibt).
5. (12 Punkte) Der Graph ist eine Leiter (siehe Definition unten).
6. (50 Punkte) Keine zusätzlichen Beschränkungen.

Anmerkung: Ein Leitergraph ist ein Graph, der aus zwei parallelen Pfaden (oder Ketten) gleicher Länge besteht, wobei jedes Paar entsprechender Knoten durch eine Kante verbunden ist und die Sprossen der Leiter bildet. Zusätzlich gibt es an einem Ende der Leiter einen speziellen Knoten —

den Zielknoten t —, der mit beiden Endpunkten der Leiter verbunden ist und somit als gemeinsamer Parent fungiert. Es ist garantiert, dass n für jeden solchen Graphen ungerade ist. Siehe das folgende Bild:



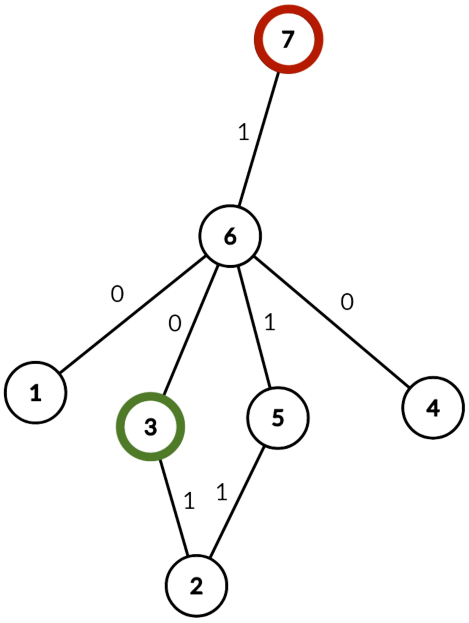
Beispiele

Beispiel 1

Nehmen wir an, wir haben einen Graphen mit 7 Knoten und 7 Kanten (unten abgebildet). Der Startknoten ist 3 (markiert mit **grün**) und der Zielknoten ist 7 (markiert mit **rot**). Der Grader wird zuerst den folgenden Funktionsaufruf machen:

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

Nehmen wir an, der Aufruf von `label` liefert `{0, 1, 1, 1, 0, 1, 0}`. Dann wird der resultierende Graph wie folgt aussehen:



Es folgt eine mögliche Abfolge von Aufrufen von `travel`, die zu einer korrekten Lösung führt:

Aufruf	Rückgabewert
`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

Wenn der zurückgegebene Wert 7 (d.h. das Ziel) ist, beendet sich das Programm.

Beispiel-Grader

Der Beispiel-Grader liest die Eingabe in folgendem Format:

- Zeile 1: n m
- Zeile 2: s t
- Zeile $3 + i$: a b, was bedeutet, dass es eine Kante zwischen den Knoten a und b gibt.

Zunächst ruft der Grader `label` mit den entsprechenden Parametern auf und beschriftet die Kanten des Graphen entsprechend dem zurückgegebenen Vektor.

Dann ruft er `travel` mit den Parametern n, s und den Nachbarn von s auf. Nach dem ersten Aufruf wird `travel` immer wieder aufgerufen, wobei der aktuelle Knoten derjenige ist, der vom vorherigen Aufruf zurückgegeben wurde, bis das Ziel t erreicht ist.

Am Ende gibt er die Anzahl der Bewegungen aus, die deine Lösung gemacht hat, und die Knoten, in der Reihenfolge, in der sie besucht wurden.