Theseus

> *Якщо всі частини корабля Тесея поступово замінювати одну за одною, то в який момент — і чи взагалі — цей корабель перестає бути тим самим?*

Коли він не поринає глибоко в абстрактне, Тесей у вільний час убиває мінотаврів. Цього разу, однак, йому спершу потрібно пройти через темний і заплутаний лабіринт. Оскільки це нелегке завдання, він просить допомоги в Даші, щоб вона скерувала його. Лабіринт можна розглядати як неорієнтований граф із n вершинами, пронумерованими від 1 до n, і m ребрами, зі спеціальною вершиною t, де знаходиться Мінотавр.

Тесей не бачить графа, але Даша бачить. Разом вони придумають стратегію, щоб він безпечно дістався вершини, де знаходиться Мінотавр: Даша поставить мітку 0 або 1 на кожному з m ребер. Після цього Тесей увійде до лабіринту через вершину s, яку Даша заздалегідь не знає.

Оскільки навколо дуже темно, у будь-який момент часу він може бачити лише індекс вершини, у якій перебуває, індекси сусідніх вершин та мітки прилеглих ребер. Також через заплутаність лабіринту він ніколи **не може згадати** жодної інформації про вершини, які відвідав раніше.

Щоб безпечно дістатися до Мінотавра, Тесею потрібно зробити не більше ніж $min + C$ ходів, де min — це мінімальна кількість ребер на шляху від s до t, а C — константа.

Деталі реалізації

Вам потрібно імплементувати дві функції:

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

- n: кількість вершин
- $edges$: список довжини m, що описує ребра графа
- t: вершина призначення
- Ця функція повинна повертати список міток довжини m, де i-й елемент може бути або 0, або 1 і представляє мітку i-го ребра для всіх $0 \leq i < m$.
- Кожне ребро має бути промарковане або 0, або 1. Маркування іншим значенням призведе до **невизначеної поведінки**.
- Ця функція викликається **рівно один раз** для кожного тестового випадку.

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

- n: кількість вершин у графі
- u: поточна вершина
- $neighbours$: список пар (v, e), що показують наявність ребра між u та v, промаркованого e
- Ця функція повинна повертати сусідню вершину, до якої слід перейти. Якщо сусідня вершина дорівнює t, програма завершується автоматично.
- Гарантується, що для будь-якого виклику цієї функції u не дорівнює спеціальній вершині t.
- Виклик цієї функції означає рух через лабіринт. Тому для кожного тестового випадку цю функцію можна викликати **стільки разів, скільки необхідно**, щоб дістатися вершини призначення.

Увага! У програмі заборонено оголошувати будь-які глобальні або статичні змінні та використовувати їх для обміну даними між різними викликами `label` чи `travel`. Будь-яка спроба обійти це призведе до **невизначеної поведінки**.

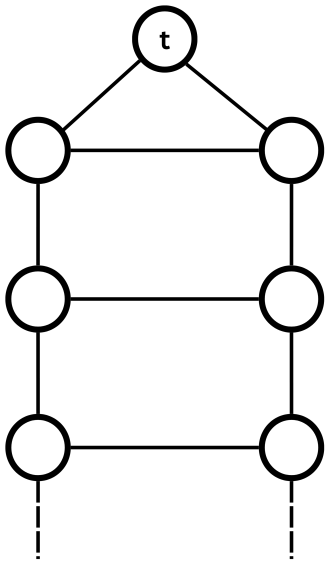
Обмеження

- $1 \leq n \leq 10000$
- $1 \leq m \leq 50000$
- $C = 14$
- Початкова вершина s фіксується для кожного тесту перед викликом функції `label`.

Підзадачі

1. (4 бали) Граф є клікою (тобто існує ребро між будь-якими двома вершинами $1 \leq u \leq v \leq n$).
2. (10 балів) Відстань між вершиною призначення та будь-якою вершиною графа не перевищує 2 ребер.
3. (11 балів) Граф є деревом.
4. (13 балів) Граф є двочастковим (тобто можна розбити вершини графа на дві підмножини так, щоб не існувало ребра між двома вершинами з однієї підмножини).
5. (12 балів) Граф буде «драбинкою» (див. визначення нижче).
6. (50 балів) Без додаткових обмежень.

Примітка: граф «драбинка» — це граф, що складається з двох паралельних шляхів (або ланцюгів) однакової довжини, причому кожна пара відповідних вершин з'єднана ребром, утворюючи «сходинки» драбинки. Крім того, на одному кінці драбинки є спеціальна вершина — вершина призначення t — яка з'єднана з обома кінцевими вершинами драбинки, фактично виконуючи роль спільного батька. Гарантується, що n буде непарним для будь-якого такого графа. Подивіться малюнок нижче.



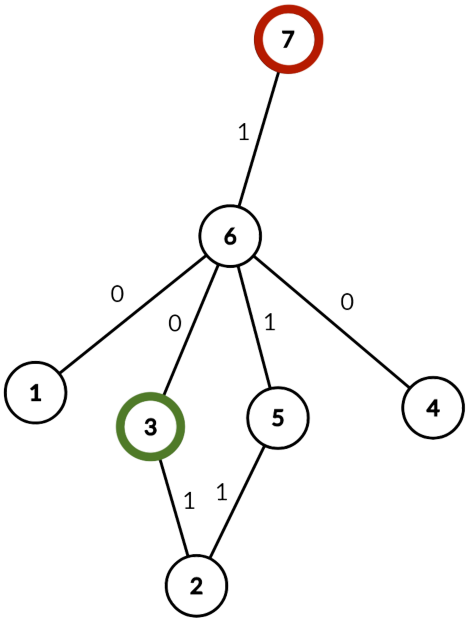
Приклади

Приклад 1

Припустимо, що маємо граф із 7 вершинами та 7 ребрами (переліченими нижче). Початкова вершина — 3 (позначена **зеленим**), а вершина призначення — 7 (позначена **червоним**). Спершу градер викличе:

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

Припустимо, що виклик `label` повертає `{0, 1, 1, 1, 0, 1, 0}`. Тоді отриманий граф виглядатиме так:



Нижче наведено можливу послідовність викликів `travel`, яка приведе до правильного розв'язку:

Виклик	Повернуте число
`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

Коли повертається число 7 (вершина призначення) — програма завершується.

Приклад градера

Зчитує вхідні дані у такому форматі:

- рядок 1: $n\ m$
- рядок 2: $s\ t$
- рядок $3 + i$: $a\ b$, що позначає наявність ребра, яке з'єднує вершини a та b

Спочатку градер викликає `label` з відповідними параметрами й маркує ребра графа згідно з повернутим вектором.

Потім він викликає `travel` з параметрами n, s та сусідами s. Після першого виклику він продовжуватиме послідовно викликати `travel`, де поточною вершиною буде та, яку повернув попередній виклик, доки не буде досягнуто вершини призначення t.

У кінці він виведе кількість ходів, які зробив ваш розв'язок, та вершини, які були пройдені у тому ж порядку.