Lec-59

26/11/25 10:13 AM

$0 \leftarrow \left(\frac{n(n-1)}{2} - m\right)$

Ujan has a lot of useless stuff in his drawers, a considerable part of which are his math notebooks: it is time to sort them out. This time he found an old dusty graph theory notebook with a description of a graph.

It is an undirected weighted graph on n vertices. It is a complete graph: each pair of vertices is connected by an edge. The weight of each edge is either 0 or 1; exactly m edges have weight 1, and all others have weight 0.

Since Ujan doesn't really want to organize his notes, he decided to find the weight of the minimum spanning tree of the graph. (The weight of a spanning tree is the sum of all its edges.) Can you find the answer for Ujan so he stops procrastinating?

Input

The first line of the input contains two integers n and m ($1 \le n \le 10^5$, $0 \le m \le \min(\frac{n(n-1)}{2}, 10^5)$), the number of vertices and the number of edges of weight 1 in the graph.

The i-th of the next m lines contains two integers a_i and b_i ($1 \le a_i, b_i \le n$, $a_i \ne b_i$), the endpoints of the i-th edge of weight 1.

It is guaranteed that no edge appears twice in the input.

Output

Output a single integer, the weight of the minimum spanning tree of the graph.

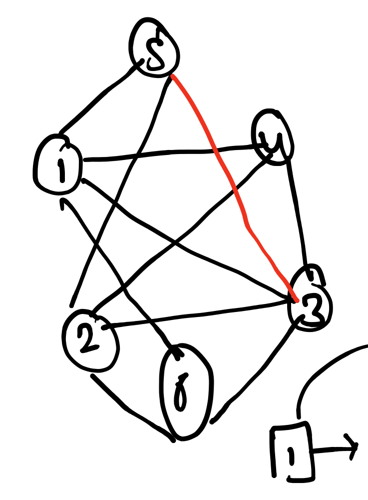
Examples

n — 6 11 → m

1 3
1 4
1 5
1 6
2 3
2 4
2 5
2 6
3 4
3 5
3 6

$n \times (n-1) = n^2$

1 | 2, n
2 | (1, n) 2



1 → [3,4,5,6]
[2] → [3,4,5,6]
[3] → [1,2, 4,5,6]
[4] → [1,2,3]
5 → [1,2]
6 → [1,2]

1 1–0

[1,2,3,4,5,6]

[2]

2

```java
private static void dfs(int i) {
	// TODO Auto-generated method stub
	Stack<Integer> st = new Stack<>();
	for(int x:set) { [2,3,4,5,6]
		if(!map.get(i).contains(x)) {
			st.push(x);
		}
	}

}
```

```java
private static void dfs(int i) {
	// TODO Auto-generated method stub
	Stack<Integer> st = new  Stack<>();
	for(int x:set) {
		if(!map.get(i).contains(x)) {
			st.push(x);
		}
	}
	while(!st.isEmpty()) {
		int s=st.pop();
		set.remove(s);
		for(int x:set) {
			if(!map.get(s).contains(x)) {
				st.push(x);
			}
		}
	}

}
```

1,2,3,4,5,6

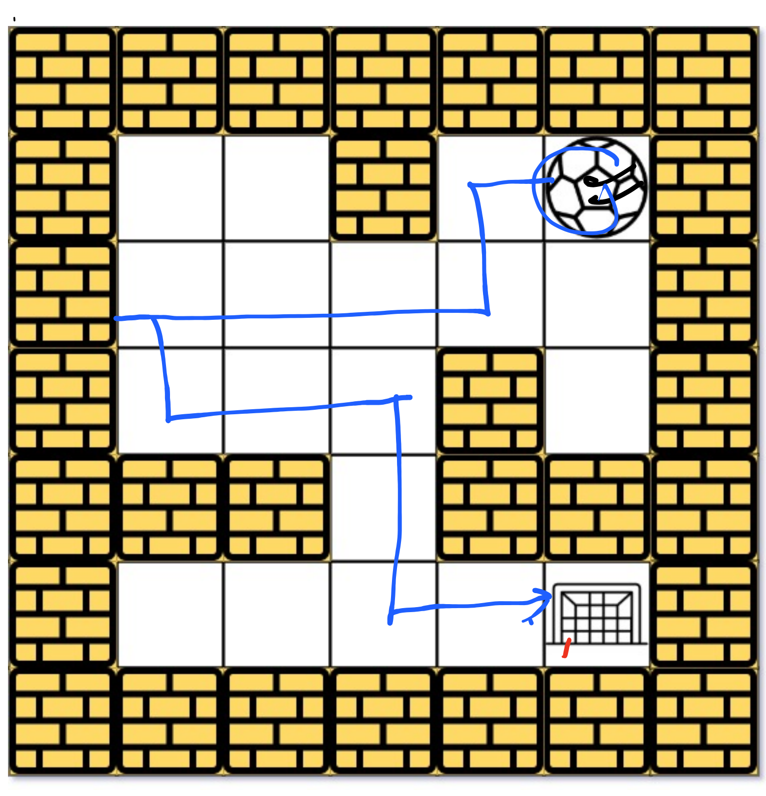
2

6
5

6

1 → [3,4,5,6]
[2] → [3,4,5,6]
[3] → [1,2, 4,5,6]
[4] → [1,2,3]
5 → [1,2]
6 → [1,2]

cr, cc, cost



There is a ball in a `maze` with empty spaces (represented as `0`) and walls (represented as `1`). The ball can go through the empty spaces by rolling **up, down, left or right**, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

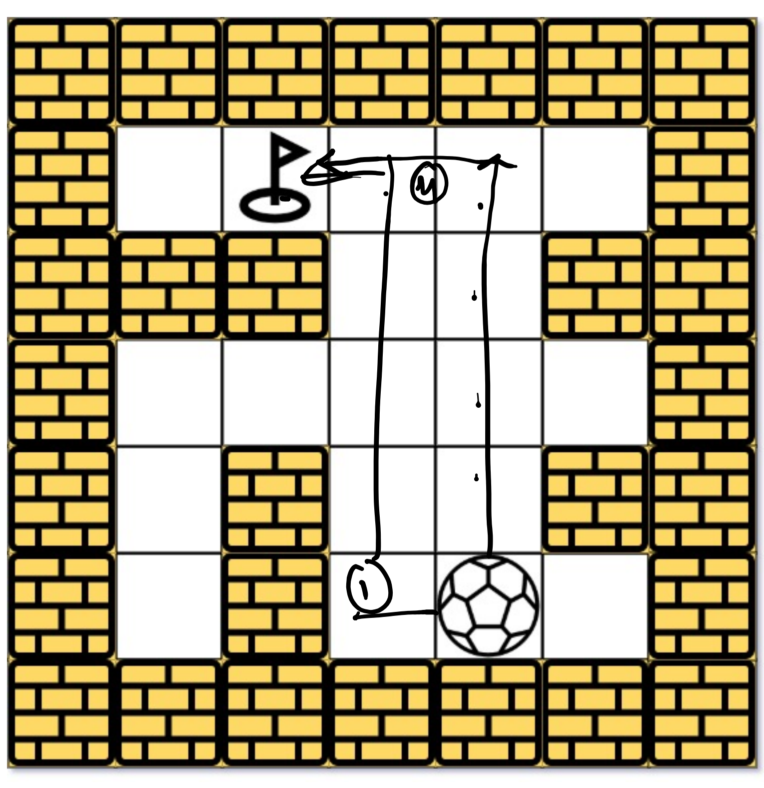
Given the `m x n` `maze`, the ball's `start` position and the `destination`, where `start = [start_row, start_col]` and `destination = [destination_row, destination_col]`, return *the shortest **distance** for the ball to stop at the destination*. If the ball cannot stop at `destination`, return `-1`.

The **distance** is the number of **empty spaces** traveled by the ball from the start position (excluded) to the destination (included).

You may assume that **the borders of the maze are all walls** (see examples).

```
Input: maze =
[[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4],
destination = [4,4]
Output: 12
```

0 | 4 | 0



6 UL

6 UUL

There is a ball in a `maze` with empty spaces (represented as `0`) and walls (represented as `1`). The ball can go through the empty spaces by rolling **up, down, left or right**, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction (must be different from last chosen direction). There is also a hole in this maze. The ball will drop into the hole if it rolls onto the hole.

Given the `m x n` `maze`, the ball's position `ball` and the hole's position `hole`, where `ball = [ball_row, ball_col]` and `hole = [hole_row, hole_col]`, return *a string* `instructions` *of all the instructions that the ball should follow to drop in the hole with the **shortest distance** possible*. If there are multiple valid instructions, return the **lexicographically minimum** one. If the ball can't drop in the hole, return `"impossible"`.

If there is a way for the ball to drop in the hole, the answer `instructions` should contain the characters `'u'` (i.e., up), `'d'` (i.e., down), `'l'` (i.e., left), and `'r'` (i.e., right).

The **distance** is the number of **empty spaces** traveled by the ball from the start position (excluded) to the destination (included).

You may assume that **the borders of the maze are all walls** (see examples).

(a,n) → if(a.a== --' (

a -