Building Roads

Byteland has n cities, and m roads between them. The goal is to construct new roads so that there is a route between any two cities.

Your task is to find out the minimum number of roads required, and also determine which roads should be built.

Input

The first input line has two integers n and m: the number of cities and roads. The cities are numbered 1,2, . . . ,n.

After that, there are m lines describing the roads. Each line has two integers a and b: there is a road between those cities.

A road always connects two different cities, and there is at most one road between any two cities.

Output

First print an integer k: the number of required roads.

Then, print k lines that describe the new roads. You can print any valid solution.

Constraints

1 <= n <= 10^5
1 <= m <= 2 ⋅ 10^5
1 <= a,b <= n

Example

Input:

```
4 2
1 2
3 4
```

Output:

```
1
2 3
```

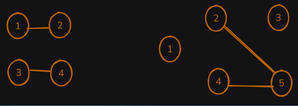
Problem Breakdown

Lets understand the problem first

You're given n cities and m roads.

Some cities may already be connected.

Your goal is to build the minimum number of new roads so that all cities become reachable (i.e., form a connected graph).



Recognize Graph Concepts

Each city = node

Each road = edge

Already connected groups = connected components

So, we're finding how many connected components there are.

What do we need to do?

To connect k components, we need at least k - 1 roads.

Why? Because connecting two components reduces the number of components by 1.

Plan the Algorithm

Step-by-step:

1. Read graph input.
2. Traverse the graph using DFS/BFS.
3. Each time you find a new unvisited node:
   - Mark it as a new connected component.
   - Save this node as the representative of that component.
4. After traversal, connect all component representatives using the first one.

Algorithm

```cpp
#include <bits/stdc++.h>
using namespace std;

#define endl '\n'

int n, m;
vector<vector<int>> graph;
vector<bool> visited;

void dfs(int node) {
    visited[node] = true;
    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor);
        }
    }
}

void solve() {
    cin >> n >> m;
    graph.resize(n + 1);
    visited.assign(n + 1, false);

    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    vector<int> representatives;

    for (int i = 1; i <= n; ++i) {
        if (!visited[i]) {
            representatives.push_back(i);
            dfs(i);
        }
    }

    int numNewRoads = representatives.size() - 1;
    cout << numNewRoads << endl;

    for (int i = 1; i < representatives.size(); ++i) {
        cout << representatives[0] << " " << representatives[i] << endl;
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    solve();
    return 0;
}
```

Traversing the graph using DFS/BFS

Reading the graph input

Save this node as the representative of that component.

After traversal, connect all component representatives using the first one.

Time Complexity

O(N + M)

we visit every cell at most once.

Space Complexity

O(N + M)

for queues and auxiliary arrays