Коледни играчки

locked

Problem | Submissions | Leaderboard | Discussions

Украсявайки коледната си елха, Иванчо се натъкна на интересен проблем. Той има N на брой коледни играчки, номерирани с числата от 1 до N. Преди да ги сложи на елхата, Иванчо иска да оцвети всяка от тях в някакъв цвят. Тъй като е малко капризен, той има M изисквания за двойки играчки, които трябва да бъдат едноцветни. Напишете програма, която намира най-голямия брой цветове, които Иванчо може да използва за оцветяването на играчките.

Input Format

На първия ред на стандартния вход са зададени числата **N** и **M**. Следват **M** реда, на всеки от които са записани по две числа – номерата на две играчки, които трябва да бъдат едноцветни. Между дадените двойки играчки може да има повтарящи се.

Constraints

1 ≤ N ≤ 10^9

1 ≤ M ≤ 10^5

Output Format

Изведете едно число - най-голямия брой цветове, които Иванчо може да използва за боядисване на играчките.

Sample Input 0

```
10 5
1 2
1 3
2 3
1 4
2 1
```

Sample Output 0

```
7
```

Submissions: 101
Max Score: 100
Difficulty: Medium

Rate This Challenge:

More

Current Buffer (saved locally, editable) C++

```cpp
#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <algorithm>
#include <unordered_map>
#include <list>
#include <forward_list>
#include <map>
int counter=0;
using namespace std;
struct Node{
  vector<int> neighbours;
  bool hasNeighbour(int index){
    for(auto it:neighbours){
      if(it==index){
        return true;
      }
    }
    return false;
  }
  void addNeighbour(int index){
    neighbours.push_back(index);
  }
};
struct Graph{
  map<int,Node> nodes;
  map<int,bool> visited;
  void connect(int from, int to) {
    if(nodes.find(from)!=nodes.end()&& nodes.find(to)!=nodes.end()){
      nodes[from].addNeighbour(to);
      nodes[to].addNeighbour(from);

    }
    else if(nodes.find(from)==nodes.end()&& nodes.find(to)!=nodes.end()){
      Node n;
      nodes.insert({from,n});
      visited.insert({from,false});
      nodes[from].addNeighbour(to);
      nodes[to].addNeighbour(from);

    }
    else if(nodes.find(from)!=nodes.end()&& nodes.find(to)==nodes.end()){
      Node n;
      nodes.insert({to,n});
      visited.insert({to,false});
      nodes[from].addNeighbour(to);
      nodes[to].addNeighbour(from);


    }
    else if(nodes.find(from)==nodes.end()&& nodes.find(to)==nodes.end()){
      Node n;
      Node n1;
      nodes.insert({to,n});
       nodes.insert({from,n1});
      visited.insert({from,false});
      visited.insert({to,false});
      nodes[from].addNeighbour(to);
      nodes[to].addNeighbour(from);

    }
  }
  void _DFS(int current) {
    visited[current] = true;
    for (auto neighbour : nodes[current].neighbours) {
      if (!visited[neighbour]) {
        _DFS(neighbour);
      }
    }
  }
  void DFSAll() {


    for (auto it:nodes) {
      if (!visited[it.first]) {
         counter++;
        _DFS(it.first);

      }
    }
  }
};
int main() {
  int toys;
  int doubles;
  cin>>toys;
  cin>>doubles;

  Graph g;
  int first;
  int second;

  for(int i=0;i<doubles;i++){
    cin>>first;
    cin>>second;
    g.connect(first,second);
  }
   g.DFSAll();
  cout<<toys-g.nodes.size()+counter;
    return 0;
}
```

Line: 1 Col: 1

Upload Code as File | Test against custom input | Run Code | Submit Code