2493. Divide Nodes Into the Maximum Number of Groups

You are given a positive integer `n` representing the number of nodes in an **undirected** graph. The nodes are labeled from `1` to `n`.

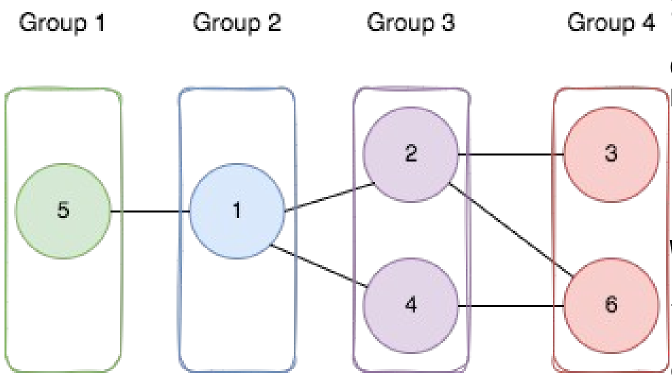
You are also given a 2D integer array `edges`, where `edges[i] = [ai, bi]` indicates that there is a **bidirectional** edge between nodes `ai` and `bi`. **Notice** that the given graph may be disconnected.

Divide the nodes of the graph into `m` groups (**1-indexed**) such that:

- Each node in the graph belongs to exactly one group.
- For every pair of nodes in the graph that are connected by an edge `[ai, bi]`, if `ai` belongs to the group with index `x`, and `bi` belongs to the group with index `y`, then `|y - x| = 1`.

Return *the maximum number of groups (i.e., maximum* `m`*) into which you can divide the nodes*. Return `-1` *if it is impossible to group the nodes with the given conditions*.

Example 1:



```
Input: n = 6, edges = [[1,2],[1,4],[1,5],
[2,6],[2,3],[4,6]]
Output: 4
Explanation: As shown in the image we:
- Add node 5 to the first group.
- Add node 1 to the second group.
- Add nodes 2 and 4 to the third group.
- Add nodes 3 and 6 to the fourth group.
We can see that every edge is satisfied.
It can be shown that that if we create a
fifth group and move any node from the
third or fourth group to it, at least on
of the edges will not be satisfied.
```

Example 2:

```
Input: n = 3, edges = [[1,2],[2,3],[3,1]]
Output: -1
Explanation: If we add node 1 to the first group, node 2 to the second group, and
node 3 to the third group to satisfy the first two edges, we can see that the third
edge will not be satisfied.
It can be shown that no grouping is possible.
```

Constraints:

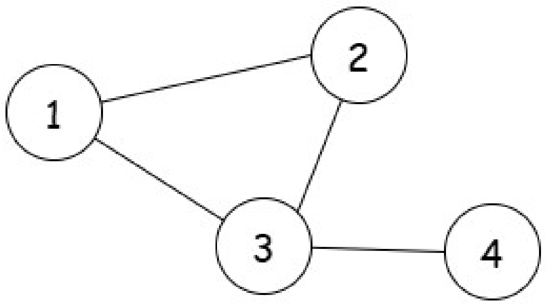
- `1 <= n <= 500`
- `1 <= edges.length <= 104`
- `edges[i].length == 2`
- `1 <= ai, bi <= n`
- `ai != bi`
- There is at most one edge between any pair of vertices.

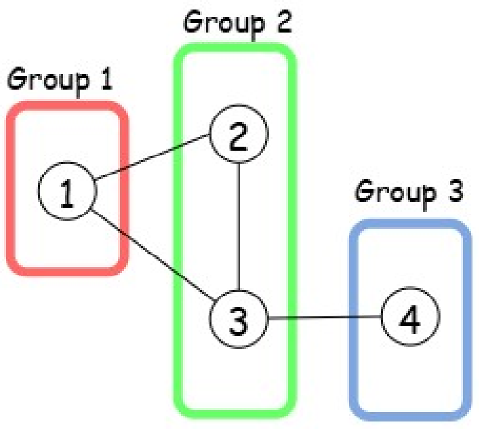
Overview

We are given a graph with `n` nodes, represented by a 2D array edges, where `edges[i] = [u, v]` means there is a bidirectional edge between nodes `u` and `v`. Our task is to divide the nodes into the largest number of numbered groups (1, 2, 3, ...) such that:

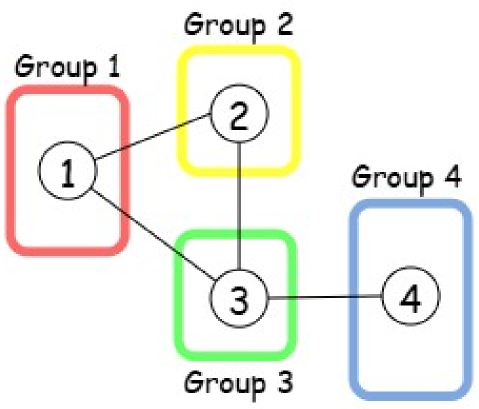
- Each node belongs to exactly one group.
- If there is an edge `[u, v]`, and `u` is in group `x`, then `v` must be in either group `x - 1` or `x + 1`.

Sometimes, this kind of split is not possible. For example, consider this graph:





Invalid Split: Nodes 2 and 3 are directly connected and belong to the same group.

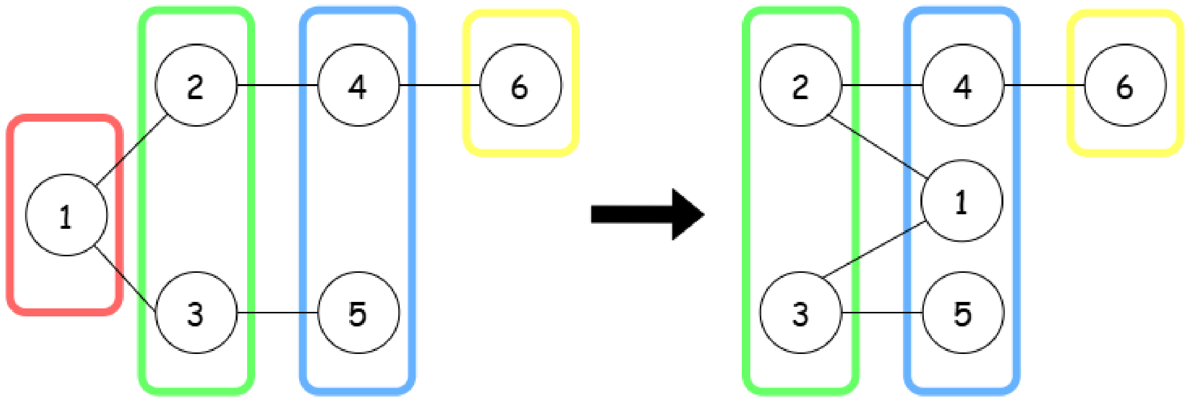


Invalid Split: Nodes 1 and 3 are directly connected but they do not belong to consecutive groups.

Here, no valid split exists. In such cases, we return `-1`.

A key observation is that if it’s possible to divide the nodes into `x` groups (`x > 2`), we can also divide them into `x - 1` groups. Intuitively, this works because the nodes in the first and third groups can’t be directly connected, but they must all connect to nodes in the second group. By combining groups `1` and `3`, we get a valid split with `x - 1` groups.

Combining groups 1 and 3 results in a valid split of 3 groups.



So, to check if a valid split is possible, we just need to see if the graph can be split into two groups—in other words, whether it is *bipartite*.

A graph is bipartite when we can divide its nodes into two distinct sets where:

- *All edges connect vertices from one set to vertices in the other set.*
- *No edges exist between vertices within the same set.*

Another key detail to consider is that the given graph is not always connected. In this case, we calculate the largest number of groups for each connected part of the graph, and then take the sum of these numbers.

To sum up, the problem boils down to these two steps:

1. Check if the graph is bipartite to see if a valid split exists.
2. For each connected part of the graph, find the largest number of groups we can divide the nodes into and return their sum.

2493. Divide Nodes Into the Maximum Number of Groups

```
/*
    Build the graph: <O(m),O(mn)>
    Check if graph is bipartitable using DSU: <O(nα(n)),O(3n)>
    Get all graph components using DSU: <O(mα(n)+n+n),O(5n)>
    Cumulate the number of groups that we can make of each component: <O(n),O(1)>
    Overall time complexity: O((n+m)α(n))
    Overall sapce complexity: O(mn+8n)
    n: #nodes
    m: #edges
*/
```

126 ms | Beats 83.26%

53.64 MB | Beats 59.00%

```
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
class DSU{
    public:
        vi parent;
        vi group;
        DSU(int _N){
            parent.resize(_N);
            group.resize(_N,1);
            for(int i=0;i<_N;++i){
                parent[i]=i;
            }
        }
        int find(int p){
            if(p==parent[p]) return p;
            return parent[p]=find(parent[p]);
            // Iterative
            /*int root=p;
            while(root!=parent[root]){
                root=parent[root];
            }
            while(parent[p]!=root){
                int next=parent[p];
                parent[p]=root;
                p=next;
            }
            return root;*/
        }
```

```
void unify(int p,int q){
            int parent_p=find(p);
            int parent_q=find(q);
            if(parent_p==parent_q) return;

            if(group[parent_p]<group[parent_q]){
                group[parent_q]+=group[parent_p];
                group[parent_p]=1;
                parent[parent_p]=parent_q;

            }
            else{
                group[parent_p]+=group[parent_q];
                group[parent_q]=1;
                parent[parent_q]=parent_p;
            }
        }
};

class Solution {
    private:
        vvi graph,components;
        vi depth;

    public:
        void build_graph(int n, vvi& edges){
            graph.resize(n);
            for(auto& edge: edges){
                int u=--edge[0];
                int v=--edge[1];
                graph[u].push_back(v);
                graph[v].push_back(u);
            }
        }
    // Check if graph is bipartitable using union find
    // https://www.youtube.com/watch?v=f7xY_7wTa3M&list=PL-FHy1OmEFZsAhLhVAUYzguW4zzNXHE15&index=3
    bool is_bipartitable(int n){
      DSU dsu=DSU(n);
      for(int u=0;u<n;++u){
        for(auto& v: graph[u]){
          int v0=graph[u][0];
          if(dsu.find(u)==dsu.find(v)) return false;
          dsu.unify(v,v0);
        }
      }
      return true;
    }
```

```
    // Store all graph components using union find
    void get_components(int n,vvi& edges){
      std::unordered_map<int,vi> comp;
      DSU dsu=DSU(n);
      for(auto& edge: edges){
        int u=edge[0];
        int v=edge[1];
        dsu.unify(u,v);
      }
      for(int node=0;node<n;++node){
        int parent=dsu.find(node);
        comp[parent].push_back(node);
      }

      for(auto&[_,c]: comp){
        components.push_back(c);
      }
    }
```

```
// Count the maximum number of groups that we can made using BFS
int count_groups(int n,int node){
    vi depth(n,0);

    std::queue<int> q;
    q.push(node);

    depth[node]=1;
    int goup_count=1;
    while(!q.empty()){
        int cur_node=q.front();
        q.pop();
        for(auto& ne: graph[cur_node]){
            if(!depth[ne]){
                depth[ne]=depth[cur_node]+1;
                goup_count=std::max(goup_count,depth[ne]);
                q.push(ne);
            }

        }
    }
    return goup_count;
}
```

```
int magnificentSets(int n, vvi& edges){
        // Construct the graph adjacency list
        build_graph(n,edges);

        // Check the graph is bipartitable
        if(!is_bipartitable(n)) return -1;

        // Determine all the components of the graph
        get_components(n,edges);

        // For each component:
        int ans=0;
        for(auto& cur_comp: components){
            // Starting from each node if the current component
            // Count the maximum number of groups that we can made
            int cnt=0;
            for(auto& node: cur_comp){
                cnt=std::max(cnt,count_groups(n,node));
            }
            // Cumulate the number of groups
            ans+=cnt;
        }

        return ans;
    }
};
```