MICROSOFT PREPARATION

DAY : 19

Redundant Connections :

Problem Link : https://leetcode.com/problems/redundant-connection/

Test Cases Passed : 39 / 39

Time Used : 10.15

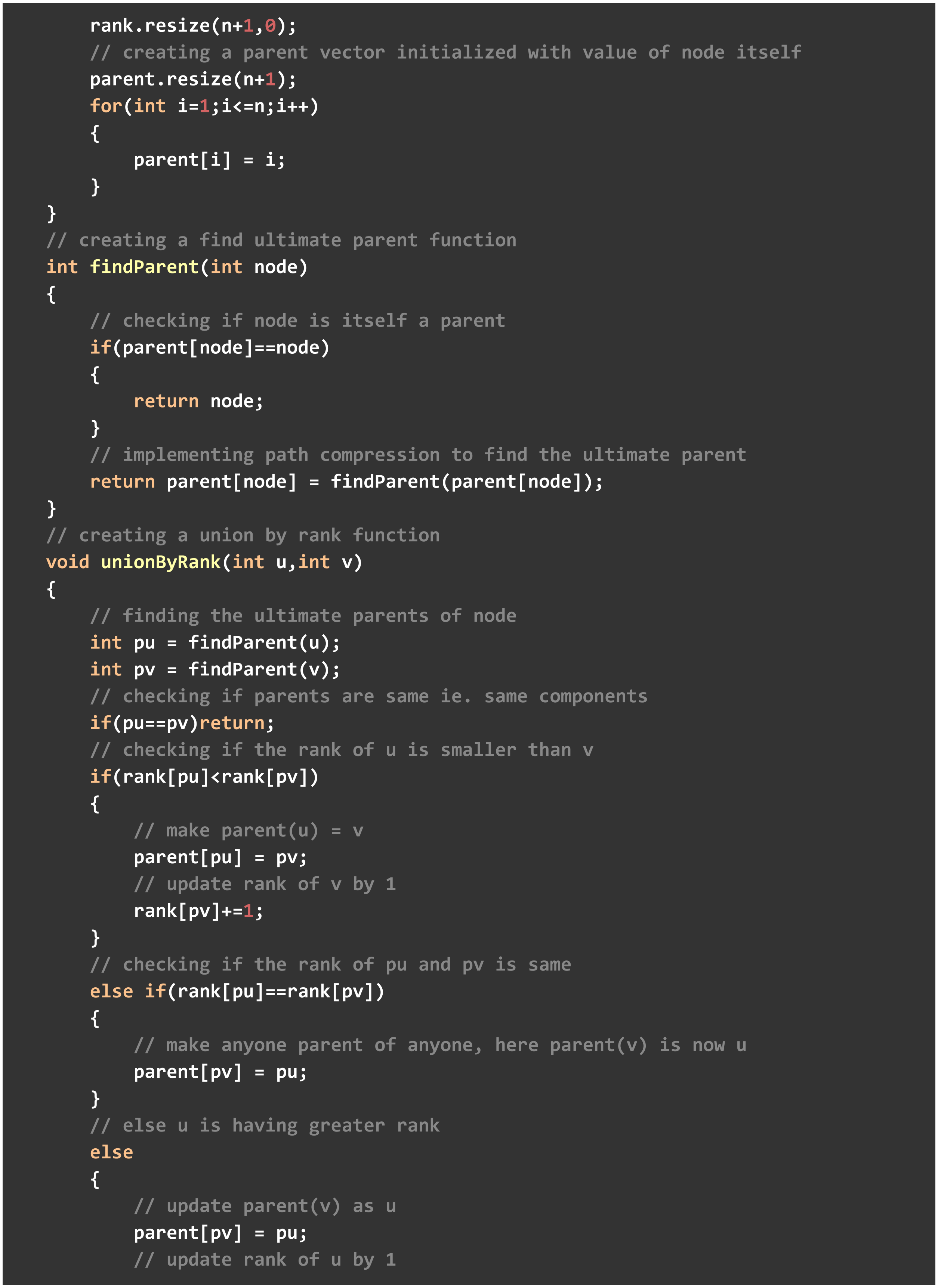
Difficulty Level : MEDIUM

Approach Used :

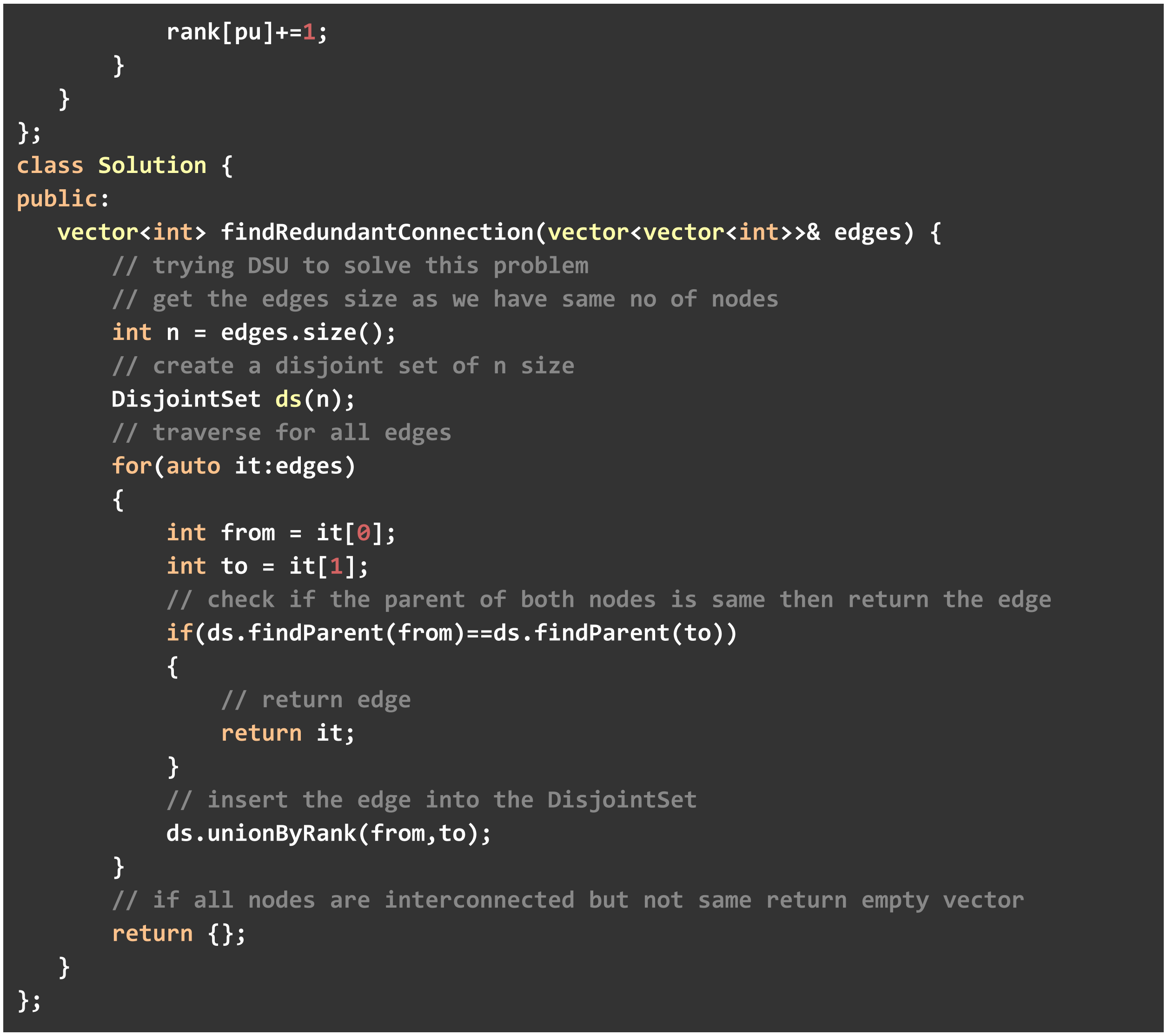
- Calculate the number of nodes as n
- Create a disjoint set of n elements
- Traverse for all edges :
    - Check if the parent(from) same as parent(to) ie. already connected / redundant edge :
        - Return edge
- Return empty vector

Solution :

```
class DisjointSet
{
   private:
   // creating a parent and rank vector
   vector<int> parent;
   vector<int> rank;
   public:
   // creating a constructor to construct the disjoint set for n nodes
   DisjointSet(int n)
   {
       // 1 based indexing
       // creating a rank vector initialized with 0
```



```
        rank.resize(n+1,0);
        // creating a parent vector initialized with value of node itself
        parent.resize(n+1);
        for(int i=1;i<=n;i++)
        {
            parent[i] = i;
        }
    }
    // creating a find ultimate parent function
    int findParent(int node)
    {
        // checking if node is itself a parent
        if(parent[node]==node)
        {
            return node;
        }
        // implementing path compression to find the ultimate parent
        return parent[node] = findParent(parent[node]);
    }
    // creating a union by rank function
    void unionByRank(int u,int v)
    {
        // finding the ultimate parents of node
        int pu = findParent(u);
        int pv = findParent(v);
        // checking if parents are same ie. same components
        if(pu==pv)return;
        // checking if the rank of u is smaller than v
        if(rank[pu]<rank[pv])
        {
            // make parent(u) = v
            parent[pu] = pv;
            // update rank of v by 1
            rank[pv]+=1;
        }
        // checking if the rank of pu and pv is same
        else if(rank[pu]==rank[pv])
        {
            // make anyone parent of anyone, here parent(v) is now u
            parent[pv] = pu;
        }
        // else u is having greater rank
        else
        {
            // update parent(v) as u
            parent[pv] = pu;
            // update rank of u by 1
```



```
           rank[pu]+=1;
       }
   }
};
class Solution {
public:
   vector<int> findRedundantConnection(vector<vector<int>>& edges) {
       // trying DSU to solve this problem
       // get the edges size as we have same no of nodes
       int n = edges.size();
       // create a disjoint set of n size
       DisjointSet ds(n);
       // traverse for all edges
       for(auto it:edges)
       {
           int from = it[0];
           int to = it[1];
           // check if the parent of both nodes is same then return the edge
           if(ds.findParent(from)==ds.findParent(to))
           {
               // return edge
               return it;
           }
           // insert the edge into the DisjointSet
           ds.unionByRank(from,to);
       }
       // if all nodes are interconnected but not same return empty vector
       return {};
   }
};
```