Q.1 <https://www.geeksforgeeks.org/problems/minimum-spanning-tree/1>

IN MST we are greedy each node for taking adjancent we are seeing which edge weight is minimum this we do for all node

* So to solve the MST problem we used greedy logic and prim’s Algo
* First declare visited array of V and also create adjacency list if not given
* Create min heap
* Push start wt=0 and node=0
* While !pq.empty()
* Take out dist ,node
* Check if that node is visited no further execution of code do continue
* If’ node not visited mark them visited and wt to the sum
* Iterate through adjacency list
* Finout adjNode and wt if adjNode not vis push into queue wt,adjNode
* At last return sum

|  |
| --- |
| int spanningTree(int n, vector<vector<int>> adj[])  {  // code LogE for push into pq  priority\_queue<pair<int,int>,  vector<pair<int,int>>,  greater<pair<int,int>>> pq;  vector<int> vis(n,0);  pq.push({0,0});  int sum=0;  //this will run E time  while(!pq.empty())  {  int wt=pq.top().first;  int node=pq.top().second;  pq.pop();  // cout<<node<<endl;  if(vis[node]==1) continue;    vis[node]=1;    sum+=wt;  for(auto it:adj[node])  {  int adjNode=it[0];  int adjWt=it[1];  if(!vis[adjNode])  {  pq.push({adjWt,adjNode});  }  }  }  return sum;  } |
|  |

sTC: O(ElogE) SC:O(V)

Q.3 <https://www.geeksforgeeks.org/problems/disjoint-set-union-find/1?utm_source=youtube&utm_medium=collab_striver_ytdescription&utm_campaign=disjoint-set-union-find>

|  |
| --- |
| int find(int A[],int X)  {  //add code here  if(A[X]==X) return X;    else return A[X]=find(A,A[X]);    }  void unionSet(int A[],int X,int Z)  {  //add code here.    int X\_Parent=find(A,X);  int Z\_Parent=find(A,Z);    if(X\_Parent==Z\_Parent) return;    A[X\_Parent]=Z\_Parent;  } |

TC: O(4Alpha) SC:O(N)

Q.4 <https://www.naukri.com/code360/problems/kruskal-s-minimum-spanning-tree-algorithm_1082553?leftPanelTabValue=PROBLEM>

* So the kruskal Algo is implemented in 2 ways
* 1.union by rank
* 2.union by size
* So main logic
* 1. To find parent
* 2.perform union
* 1.Let’s see union by rank
* Here we are creating one class their we declare parent and rank vector
* Then write constructor

Resize parent and rank by 0

Iterate parent array assign it’s parent that i itself

* Find function

Here if we get node==parent[node] that means it’s top level guy who hasn’t had parent he is parent of that set so return that node

If not then return parent[node]=find(parent[node])

* Union:

1.find the parent of both node

If(both parent are same then return because they are in same set

2.check rank of first node if it’s less then first node parent=second node parent

3.if first node parent rank > then make first node parent of second node parent

3.else equal make anyone parent

Increase rank of parent by 1 which you’re make

* 2. Union by size
* Same as above just change in Union function
* Same logic just we have add size of anoth parent\_node size into node who has greater size

|  |
| --- |
| class disJointSet  {        public:          vector<int> parent,rank;          disJointSet(int n)          {              parent.resize(n+1);              rank.resize(n+1,0);              for(int i=0;i<=n;i++)              {                  parent[i]=i;              }          }          int Find(int node)          {              if(node==parent[node])              {                  return node;              }              return parent[node]=Find(parent[node]);          }          void Union(int node1,int node2)          {              int node1\_p=Find(node1);              int node2\_p=Find(node2);              if(node1\_p==node2\_p) return;              if(rank[node1\_p] < rank[node2\_p])              {                  parent[node1\_p]=node2\_p;                }              else if(rank[node1\_p] > rank[node2\_p])              {                  parent[node2\_p]=node1\_p;                }              else              {                  parent[node2\_p]=node1\_p;                  rank[node1\_p]+=1;              }          }  };  bool compare(vector<int> &a,vector<int> &b)  {      return a[2]<b[2];  }  int kruskalMST(int n, vector<vector<int>> &edges)  {      // Write your code here.      // vector<pair<int,pair<int,int>>> adj[n+1];      int mst=0;      sort(edges.begin(),edges.end(),compare);      disJointSet ds(n);      for(int i=0;i<edges.size();i++)      {          int u=edges[i][0];          int v=edges[i][1];          int wt=edges[i][2];          // adj[i].push\_back({wt,{u,v}});            if(ds.Find(u)!=ds.Find(v))          {              mst+=wt;              ds.Union(u,v);          }      }        return mst;  } |

|  |
| --- |
| class disJointSet  {        public:          vector<int> parent,rank;          disJointSet(int n)          {              parent.resize(n+1);              rank.resize(n+1,0);              for(int i=0;i<=n;i++)              {                  parent[i]=i;              }          }          int Find(int node)          {              if(node==parent[node])              {                  return node;              }              return parent[node]=Find(parent[node]);          }          void Union(int node1,int node2)          {              int node1\_p=Find(node1);              int node2\_p=Find(node2);              if(node1\_p==node2\_p) return;              if(rank[node1\_p] < rank[node2\_p])              {                  parent[node1\_p]=node2\_p;                }              else if(rank[node1\_p] > rank[node2\_p])              {                  parent[node2\_p]=node1\_p;                }              else              {                  parent[node2\_p]=node1\_p;                  rank[node1\_p]+=1;              }          }  };  bool compare(vector<int> &a,vector<int> &b)  {      return a[2]<b[2];  }  int kruskalMST(int n, vector<vector<int>> &edges)  {      // Write your code here.      // vector<pair<int,pair<int,int>>> adj[n+1];      int mst=0;      sort(edges.begin(),edges.end(),compare);      disJointSet ds(n);      for(int i=0;i<edges.size();i++)      {          int u=edges[i][0];          int v=edges[i][1];          int wt=edges[i][2];          // adj[i].push\_back({wt,{u,v}});            if(ds.Find(u)!=ds.Find(v))          {              mst+=wt;              ds.Union(u,v);          }      }        return mst;  } |

|  |
| --- |
| **Theory on DisJointSet and Union-Find**  The concepts of **disjoint sets** and **union-find** are crucial in graph theory, particularly in algorithms involving connected components, minimum spanning trees, and network connectivity. Let’s break down each concept.  **1. Disjoint Sets (Union-Find Data Structure)**  A **disjoint set** (or union-find) is a data structure that keeps track of a partition of a set into disjoint (non-overlapping) subsets. It supports two primary operations:   * **Union**: Merges two sets into one. * **Find**: Identifies which set a particular element belongs to. This can be used to check if two elements are in the same set.   The union-find data structure is particularly useful when you need to manage and merge sets dynamically.  **2. Operations in Union-Find**  **1. Find**   * **Purpose**: Determine the root or representative of the set containing a particular element. * **Implementation**: Typically implemented with path compression, where we make the tree flatter by making every node point directly to the root.   **2. Union**   * **Purpose**: Merge two sets into a single set. * **Implementation**: Typically implemented with union by rank or union by size, where we always attach the smaller tree under the root of the larger tree to keep the structure balanced.   **3. How It Works in Graphs**  Union-Find is commonly used in graph algorithms, particularly for:   * **Detecting Cycles**: In an undirected graph, we can use the union-find algorithm to detect cycles. If two vertices are already in the same set and we attempt to union them, this means adding this edge would form a cycle. * **Kruskal’s Algorithm**: For finding the Minimum Spanning Tree (MST) of a graph, Kruskal’s algorithm sorts all the edges by weight and adds them one by one, using the union-find structure to ensure no cycles are formed.   **4. Example of Union-Find in Practice**  Imagine you have a graph with 6 vertices:   1. Start with each vertex in its own set: {1}, {2}, {3}, {4}, {5}, {6}. 2. For each edge, perform a union operation if the vertices are in different sets:    * Edge (1, 2): Union {1} and {2} → {1, 2}, {3}, {4}, {5}, {6}    * Edge (2, 3): Union {1, 2} and {3} → {1, 2, 3}, {4}, {5}, {6}    * And so on. 3. Use the find operation to determine if adding an edge would form a cycle.   **5. Efficiency**   * **Time Complexity**: The operations are nearly constant time, specifically O(α(N)), where α(N) is the inverse Ackermann function, which grows extremely slowly, making it very efficient for large datasets.   **Summary**  The union-find data structure is a powerful tool in graph algorithms for managing disjoint sets, allowing for efficient union and find operations, which are fundamental in tasks like cycle detection and constructing minimum spanning trees |

TC:O(4alpha) SC:O(V)

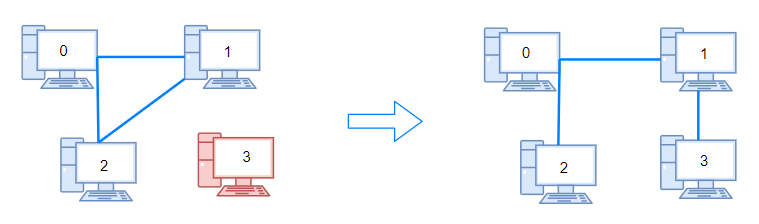
Q.5 <https://leetcode.com/problems/number-of-operations-to-make-network-connected/description/>

There are n computers numbered from 0 to n - 1 connected by ethernet cables connections forming a network where connections[i] = [ai, bi] represents a connection between computers ai and bi. Any computer can reach any other computer directly or indirectly through the network.

You are given an initial computer network connections. You can extract certain cables between two directly connected computers, and place them between any pair of disconnected computers to make them directly connected.

Return *the minimum number of times you need to do this in order to make all the computers connected*. If it is not possible, return -1.

**Example 1:**

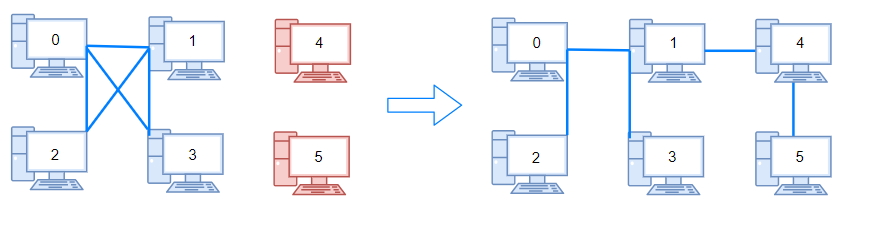


**Input:** n = 4, connections = [[0,1],[0,2],[1,2]]

**Output:** 1

**Explanation:** Remove cable between computer 1 and 2 and place between computers 1 and 3.

**Example 2:**



**Input:** n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]

**Output:** 2

* So here what I did is I used DSU-find technique or logic
* First we have to cnt extra no edges that edges are removed then also that network or component is still connect
* And we have count no of component or network set
* So one thing to observe is that if that n set or N component or N network set for connecting it we need N-1 edges
* So as it is Implement DSU class
* Then iterate through edges check if they are in same component or network set that mean same parent then do extraedges++ or else

Do union of it

* For count component it’s fix that their will only one parent of each set so if their 3 component then there are 3 parent that means i==parent[i]
* So iterate through vertices check whose parent[i]==i do component++
* After that ans=component-1
* Check if extraedges>=ans then return or else return -1

|  |
| --- |
| class Dsu  {      public:      vector<int>  parent,size;      Dsu(int n)      {          parent.resize(n);          size.resize(n,1);          for(int i=0;i<n;i++)          {              parent[i]=i;          }      }      int find(int node)      {          if(parent[node]==node) return node;          return parent[node]=find(parent[node]);      }      void Union(int n1,int n2)      {          int ul\_n1=find(n1);          int ul\_n2=find(n2);          if(ul\_n1==ul\_n2) return;          if(size[ul\_n1] > size[ul\_n2])          {              parent[ul\_n2]=ul\_n1;              size[ul\_n1]+=size[ul\_n2];          }          else if(size[ul\_n1] < size[ul\_n2])          {              parent[ul\_n1]=ul\_n2;              size[ul\_n2]+=size[ul\_n1];          }          else          {              parent[ul\_n1]=ul\_n2;              size[ul\_n2]+=size[ul\_n1];          }      }  };  class Solution {  public:      int makeConnected(int n, vector<vector<int>>& connections) {          int cntExtraEdges=0;          Dsu ds(n);          for(auto it:connections)          {              int u=it[0];              int v=it[1];              if(ds.find(u)==ds.find(v))              {                  cntExtraEdges++;              }              else              {                  ds.Union(u,v);              }          }          int Component=0;         for(int i=0;i<n;i++)         {          if(ds.parent[i]==i) Component++;         }          int ans=Component-1;          if(cntExtraEdges>=ans) return ans;          return -1;        }  }; |

TC:O(N+M)+O(N) +O(4alpha)

SC:O(N)

Q.6 <https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/submissions/1367468107/>

* First, from the stone information, we will find out the maximum row and the maximum column number so that we can get an idea about the size of the 2D plane(i.e. nothing but a matrix).
* Then, we need to create a disjoint set of sizes (maximum row index+maximum column index). For safety, we may take a size one more than required.
* Now it’s time to connect the cells having a stone. For that we will loop through the given cell information array and for each cell we will extract the row and the column number and do the following:
  + First, we will convert column no. to (column no. + maximum row index +1).
  + We will perform the union(***either unionBySize() or unionByRank()***) of the row number and the converted column number.
  + We will store the row and the converted column number in a map data structure for later use.
* Now, it’s time to calculate the number of components and for that, we will count the number of ultimate parents. Here we will refer to the previously created map.
  + We just need the nodes in the Disjoint Set that are involved in having a stone. So we have stored the rows and the columns in a map in step 3.3, as they will have stones. Now we just need to check them from the map data structure once for getting the number of ultimate parents.
* Finally, we will subtract the no. of components(i.e. no. of ultimate parents) from the total no. of stones and we will get our answer.

|  |
| --- |
| class Dsu  {      public:      vector<int>  parent,size;      Dsu(int n)      {          parent.resize(n+1);          size.resize(n+1);          for(int i=0;i<=n;i++)          {              parent[i]=i;              size[i]=1;          }      }      int find(int node)      {          if(parent[node]==node) return node;          return parent[node]=find(parent[node]);      }      void unionBySize(int n1,int n2)      {          int ul\_n1=find(n1);          int ul\_n2=find(n2);          if(ul\_n1==ul\_n2) return;          if(size[ul\_n1] > size[ul\_n2])          {              parent[ul\_n2]=ul\_n1;              size[ul\_n1]+=size[ul\_n2];          }          else if(size[ul\_n1] < size[ul\_n2])          {              parent[ul\_n1]=ul\_n2;              size[ul\_n2]+=size[ul\_n1];          }          else          {              parent[ul\_n1]=ul\_n2;              size[ul\_n2]+=size[ul\_n1];          }      }  };  class Solution {  public:      int removeStones(vector<vector<int>>& stones) {            int maxRow=INT\_MIN,maxCol=INT\_MIN;          int n=stones.size();          for(auto it:stones)          {              maxRow=max(maxRow,it[0]);              maxCol=max(maxCol,it[1]);          }            Dsu ds(maxRow + maxCol +1 );          unordered\_map<int,int> uniqueStones;          for(auto it:stones)          {              int rowNode=it[0];              int colNode=it[1]+maxRow+1;              ds.unionBySize(rowNode,colNode);              uniqueStones[rowNode]=1;              uniqueStones[colNode]=1;          }          int cnt=0;            for(auto it:uniqueStones)          {              if(ds.find(it.first)==it.first) cnt++;          }          return n-cnt;        }  }; |

TC:O(4lpha)+O(n)+O(map size) SC:O(maxRow+maxCol+1)

Q.7 <https://leetcode.com/problems/accounts-merge/>

🡪so here we have to Merge Account according email if the 2 of the person has same email then merge those account

* So write DisJoint-Union-Find for Merging account those having same email write whole class
* Second thing iterate through list i=0 to <n and inside it j=1 and j<n

Because for oth position at j there will be name so no need to check after that check if that mail is not in unordered\_map then assign mp[mail]=i

If’ it’s in map then do union

* Iterate through map before that create list of vector type and assign each mail to it’s correct parent
* Declare 2d string vector now iterate through that list i=0 to i<n and declare temp vector add name value in temp after that sort mergelist[i]

Then iterate through mergeList[i] takout one email and push\_back in after for loop push back 1d temp into 2d vector

* At last return 2d vector

|  |
| --- |
| class Dsu  {      public:      vector<int> parent,rank;      Dsu(int n)      {          parent.resize(n);          rank.resize(n,0);          for(int i=0;i<n;i++)          {              parent[i]=i;          }      }      int find(int node)      {          if(node==parent[node]) return node;          return parent[node]=find(parent[node]);      }      void unionByRank(int node1,int node2)      {          int ul\_p1=find(node1);          int ul\_p2=find(node2);          if(ul\_p1==ul\_p2) return;          if(rank[ul\_p1] > rank[ul\_p2])          {              parent[ul\_p2]=ul\_p1;          }          else if(rank[ul\_p1] < rank[ul\_p2])          {              parent[ul\_p1]=ul\_p2;          }          else          {              parent[ul\_p1]=ul\_p2;              rank[ul\_p2]+=1;          }      }  };  class Solution {  public:      vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {          int n=accounts.size();          Dsu ds(n);          unordered\_map<string,int> mailToNode;          for(int i=0;i<n;i++)          {              for(int j=1;j<accounts[i].size();j++)              {                  string mail=accounts[i][j];                  if(mailToNode.find(mail)==mailToNode.end())                  {                      mailToNode[mail]=i;                  }else                  {                      ds.unionByRank(i,mailToNode[mail]);                  }              }          }           vector<string> Mergedmail[n];          for(auto it:mailToNode)          {              string mail=it.first;              int node=ds.find(it.second);              Mergedmail[node].push\_back(mail);          }          vector<vector<string>> ans;          for(int i=0;i<n;i++)          {              vector<string> temp;              if(Mergedmail[i].size()==0) continue;              sort(Mergedmail[i].begin(),Mergedmail[i].end());              temp.push\_back(accounts[i][0]);              for(auto it:Mergedmail[i])              {                  temp.push\_back(it);              }              ans.push\_back(temp);          }          return ans;        }  }; |

TC:O(N\*M+LogN) SC:O(N\*M)

Q.8 <https://www.naukri.com/code360/problems/number-of-islands-ii_1266048>

* After each query we have to tell how many islands are their
* First declare visited vector take one cnt=0
* Iterate in queries
* Take out row and col
* Check if that is visited if yes then ans.push\_back(cnt)
* If not then mark visited[row][col]=1 and cnt++
* Iterate through it’s 4 direction take out it’s adjacent check they are valid if it is
* For node=row\*m+col and adjNode=nr\*m+nc
* If node and adjNode haven’t had same parent then decrement cnt do unionBySize on them or rank
* After for loop push back cnt into ans
* At last return ans

|  |
| --- |
| class Dsu  {      public:      vector<int> parent,rank;      Dsu(int n)      {          parent.resize(n+1);          rank.resize(n+1,0);          for(int i=0;i<n;i++)          {              parent[i]=i;          }      }      int find(int node)      {          if(parent[node]==node) return node;          return parent[node]=find(parent[node]);      }      void unionByRank(int node1,int node2)      {          int ul\_p1=find(node1);          int ul\_p2=find(node2);          if(ul\_p1==ul\_p2) return;          if(rank[ul\_p1] > rank[ul\_p2])          {              parent[ul\_p2]=ul\_p1;          }          else if(rank[ul\_p2] > rank[ul\_p1])          {              parent[ul\_p1]=ul\_p2;          }          else{              parent[ul\_p2]=ul\_p1;              rank[ul\_p1]+=1;          }      }  };  bool isValid(int row,int col,int n,int m)  {      return row>=0 && row<n && col>=0 && col<m;  }  vector<int> numOfIslandsII(int n, int m, vector<vector<int>> &q){      // Write your code here.      vector<vector<int>> vis(n,vector<int>(m,0));      int cnt=0;      vector<int> ans;      int drow[]={-1,0,1,0};      int dcol[]={0,-1,0,1};      Dsu ds(n\*m);      for(auto it:q)      {          int row=it[0];          int col=it[1];          if(vis[row][col]==1)          {              ans.push\_back(cnt);              continue;          }          vis[row][col]=1;          cnt++;          for(int i=0;i<4;i++)          {                    int adjRow=row+drow[i];                  int adjCol=col+dcol[i];                  if(isValid(adjRow,adjCol,n,m))                  {                      if(vis[adjRow][adjCol]==1)                      {                          int node=row\*m+col;                          int adjNode=adjRow\*m+adjCol;                          if(ds.find(node)!=ds.find(adjNode))                          {                              cnt--;                              ds.unionByRank(node, adjNode);                          }                      }                  }            }              ans.push\_back(cnt);        }      return ans;  } |

TC:O(N\*M\*4)+O(4alpha) SC:O(N\*M)

Q.9 <https://leetcode.com/problems/making-a-large-island/>

* So here we have to change at most one 0 to 1 and then return largest island
* First write DisJointSet by size for it
* Make Dsu(n\*n)
* Row =0 to row < n

Same col=0 and col<n if grid[row][col]==0 then continue

Check it’s adjacent 4 direction check they valid and also has grid[nr][nc]==0 findout it’s position in parent like

Node=row\*n+col adjNode=nr\*n+nc

Do union of node and adjNode

* Same as above iterate 2 for loop declare set here if grid[row][col]==1 continue check it’s adjacent they valid also has grid[nr][nc]==1 if that the case
* Then adjNode=nr\*n+nc and set.insert(ds.find(adjNode))
* After 4 direction take one total\_size=0
* Iterate through set and total\_size+=ds.size[it]
* Update max size
* After this 2 for loop completion
* Run 1 for loop from 0 to n\*n and findout each cell size and update it with mx=max(mx,ds.size[ds.find(cellNo)])
* At last return mx

|  |
| --- |
| class Dsu  {      public:      vector<int> parent,size;      Dsu(int n)      {          parent.resize(n+1);          size.resize(n+1);          for(int i=0;i<=n;i++)          {              parent[i]=i;              size[i]=1;          }      }          int find(int node)          {              if(parent[node]==node) return node;              return parent[node]=find(parent[node]);          }          void unionBySize(int node1,int node2)          {              int ul\_p1=find(node1);              int ul\_p2=find(node2);              if(ul\_p1==ul\_p2) return;              if(size[ul\_p1] > size[ul\_p2])              {                  parent[ul\_p2]=ul\_p1;                  size[ul\_p1]+=size[ul\_p2];              }              else if(size[ul\_p1] < size[ul\_p2])              {                  parent[ul\_p1]=ul\_p2;                  size[ul\_p2]+=size[ul\_p1];              }              else              {                  parent[ul\_p2]=ul\_p1;                  size[ul\_p1]+=size[ul\_p2];              }          }    };  class Solution {  public:      bool isValid(int row,int col,int n)      {          return row>=0 && row<n && col>=0 && col<n;      }      int largestIsland(vector<vector<int>>& grid) {       int n = grid.size();          Dsu ds(n \* n);          // step - 1          for (int row = 0; row < n ; row++) {              for (int col = 0; col < n ; col++) {                  if (grid[row][col] == 0) continue;                  int dr[] = { -1, 0, 1, 0};                  int dc[] = {0, -1, 0, 1};                  for (int ind = 0; ind < 4; ind++) {                      int newr = row + dr[ind];                      int newc = col + dc[ind];                      if (isValid(newr, newc, n) && grid[newr][newc] == 1) {                          int nodeNo = row \* n + col;                          int adjNodeNo = newr \* n + newc;                          ds.unionBySize(nodeNo, adjNodeNo);                      }                  }              }          }          // step 2          int mx = 0;          for (int row = 0; row < n; row++) {              for (int col = 0; col < n; col++) {                  if (grid[row][col] == 1) continue;                  int dr[] = { -1, 0, 1, 0};                  int dc[] = {0, -1, 0, 1};                  set<int> components;                  for (int ind = 0; ind < 4; ind++) {                      int newr = row + dr[ind];                      int newc = col + dc[ind];                      if (isValid(newr, newc, n)) {                          if (grid[newr][newc] == 1) {                              components.insert(ds.find(newr \* n + newc));                          }                      }                  }                  int sizeTotal = 0;                  for (auto it : components) {                      sizeTotal += ds.size[it];                  }                  mx = max(mx, sizeTotal + 1);              }          }          for (int cellNo = 0; cellNo < n \* n; cellNo++) {              mx = max(mx, ds.size[ds.find(cellNo)]);          }          return mx;        }  }; |