UNIT-V Notes

COMPETITIVE PROGRAMMING (CS602PC)

SYLLABUS:

UNIT-V Disjoint Set Union:

Disjoint set and its operations, Union Find Algorithm,

Applications: 1) Lexicographically Smallest Equivalent String, 2) Number of Distinct Islands, 3) Number of Connected Components in an Undirected Graph.

Introduction

Disjoint sets in mathematics are two sets that don't have any element in common. Sets can contain any number of elements, and those elements can be of any type. We can have a set of cars, a set of integers, a set of colors, etc. Sets also have various operations that can be performed on them, such as union, intersection, difference, etc.

We focus on **Disjoint Set Data Structure** and the operations we can perform. we will go through what a disjoint set data structure is, the disjoint set operations, as well as examples.

Disjoint Set Data Structure

Before we look at disjoint set operations, let us understand what the disjoint set data structure itself is.

A disjoint set data structure is a data structure that stores a list of disjoint sets. In other words, this data structure divides a set into multiple subsets - such that no 2 subsets contain any common element. They are also called union-find data structures or merge-find sets.

For example: if the initial set is [1,2,3,4,5,6,7,8].

A Disjoint Set Data Structure might partition it as - [(1,2,4), (3,5), (6,8),(7)].

This contains all of the elements of the original set, and no 2 subsets have any element in common.

The following partitions would be invalid:

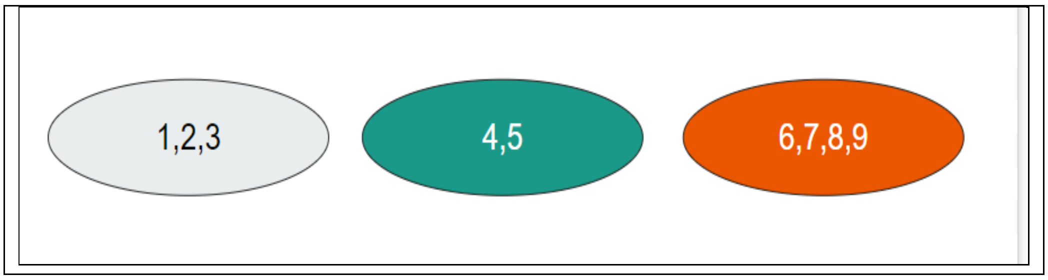
[(1,2,3),(3,4,5),(6,8),(7)] - invalid because 3 occurs in 2 subsets.

[(1,2,3),(5,6),(7,8)] -invalid as 4 is missing.

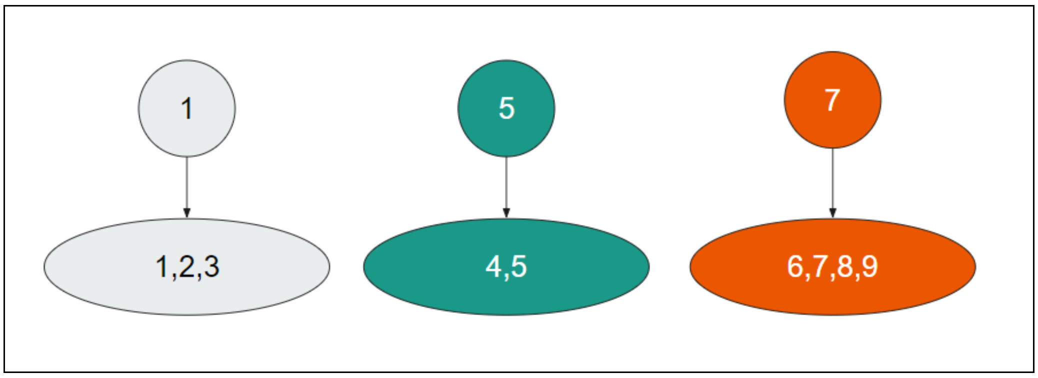
Representative Member

Each subset in a Disjoint Set Data Structure has a **representative member**. More commonly, we call this the **parent.** This is simply the member *responsible for identifying* the subset.

Let's understand this with an example.
Suppose this is the partitioning of sets in our Disjoint Set Data structure.



We wish to identify each subset, so we assign each subset a value by which we can identify it. The easiest way to do this is to choose a **member** of the subset and make it the **representative member**. This representative member will become the **parent** of all values in the subset.



Here, we have assigned representative members to each subset. Thus, if we wish to know which subset 3 is a part of, we can simply say - 3 is a part of the subset with representative member 1. More simply, we can say that the parent of 3 is 1.

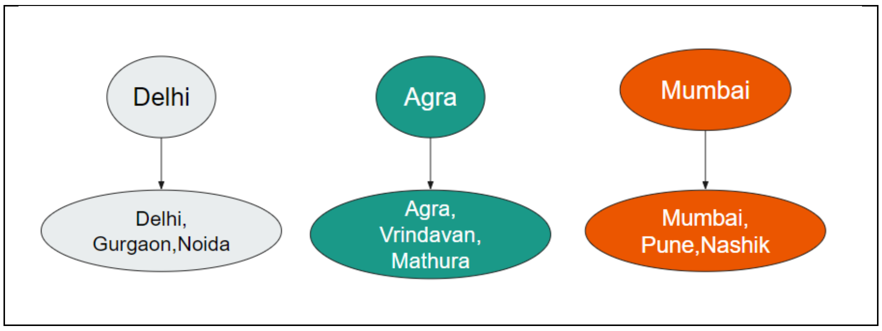
Here, 1 is its own parent.

Disjoint Set Operations

There are 2 major disjoint set operations in daa:

1. **Union/Merge** - this is used to merge 2 subsets into a single subset.
2. **Find** - This is used to find which subset a particular value belongs to.

We will take a look at both of them. To make things easier, we will change our sets from integers to Cities.

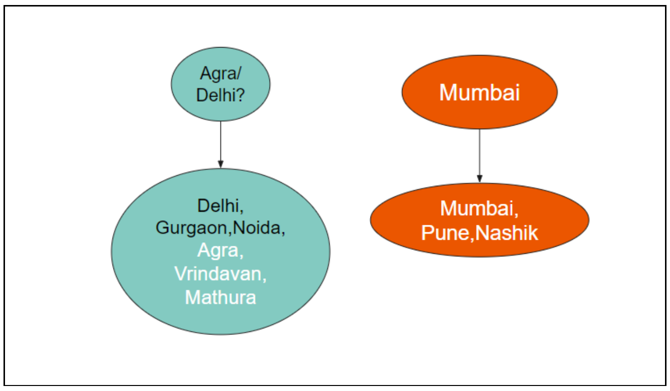


Let's say this data structure represents the places connected via express trains/ metro lines or any other means.

<u>Union/Merge:</u>

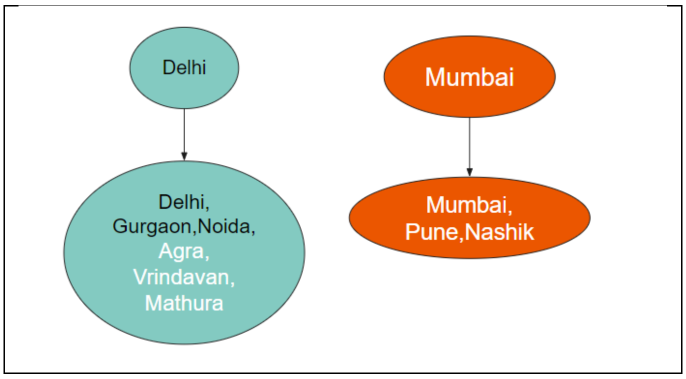
Suppose, in our example - a new express train started from Delhi to Agra. This would mean that *all* the cities connected to Delhi are now connected to *all* the cities connected to Agra.

This simply means that we can ***merge*** the subsets of Agra and Delhi into a single subset. It will look like this.



One key issue here is which out of Agra and Delhi will be the new Parent? For now, we will just choose any of them, as it will be sufficient for our purposes. Let's take Delhi for now.

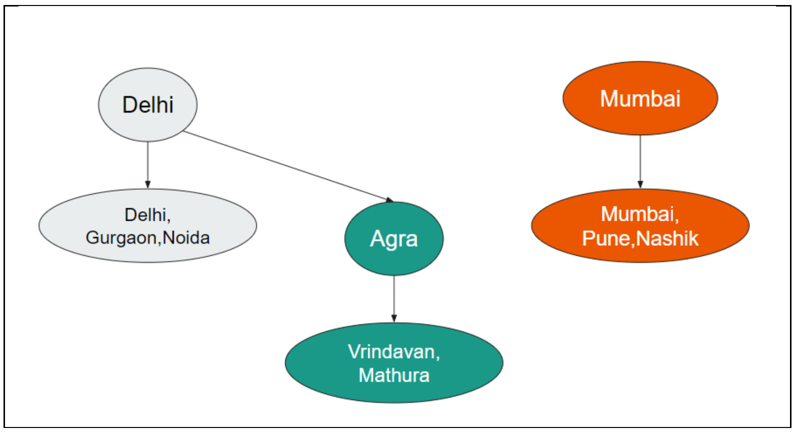
Further optimizations can be made to this by choosing the parent based on which subset has a larger number of values.



This is the new state of our Disjoint Set Data Structure. Delhi is the parent of Delhi, Gurgaon, Noida, Agra, Vrindavan, and Mathura.

This would mean that we would have to change the parent of Agra, Vrindavan, and Mathura (basically, all the children of Agra), to Delhi. This would be linear in time. If we perform the union operation m times, and each union takes O(n) times, we would get a quadratic O(mn) time complexity. This is not optimized enough.

Instead, we structure it like this:



Earlier, Agra was its own parent. Now, Delhi is the parent of Agra.
Now, you might be wondering - The parent of Vrindavan is still Agra. It should have been Delhi now.

This is where the next operation helps us - the **find** operation.

Find:

The find operation helps us find the parent of a node. As we saw above, the direct parent of a node might not be its actual (logical) parent. E.g., the logical parent of Vrindavan should be Delhi in the above example. But its direct parent is Agra.

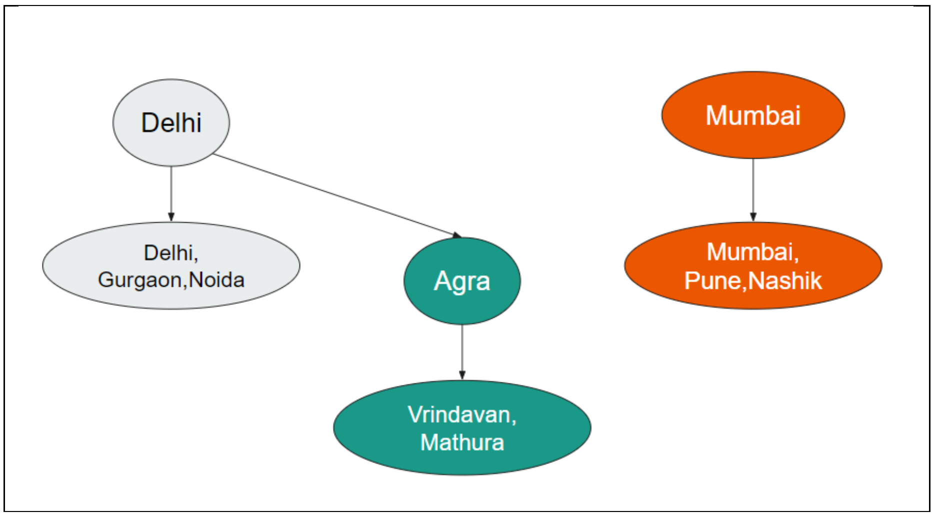
So, how do we find the actual parent?

The find operation helps us to find the actual parent. In pseudocode, the find operation looks like this:

```
find(node):
        if (parent(node)==node) return node;
        else return find(parent(node));
```

We don't return the direct parent of the node. We keep going up the tree of parents until we find a node that is its own parent.

Let's understand this via our example.

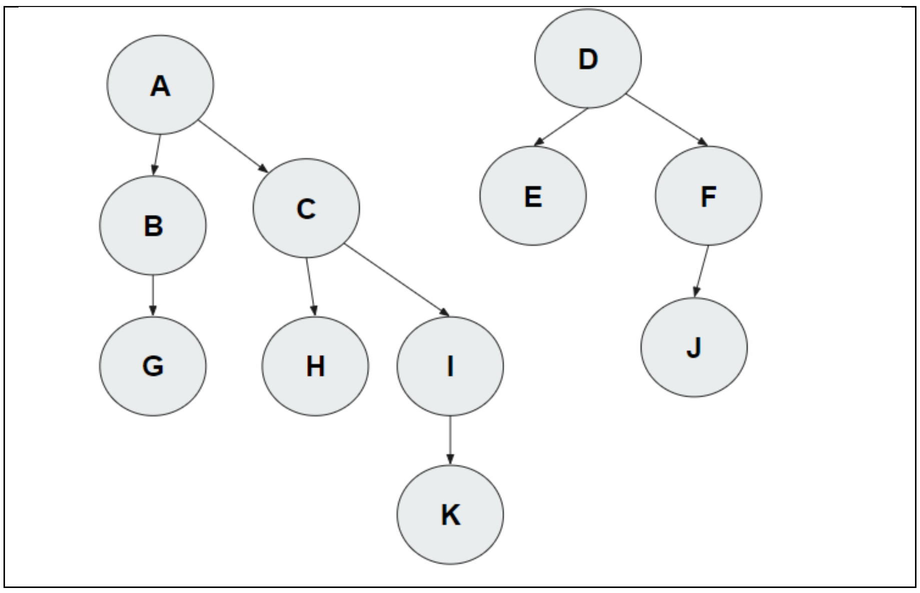


Suppose we have to find the parent of Mathura.

1. Check the direct parent of Mathura. It's Agra. Is Agra == Mathura?
   The answer is false. So, now we call the find operation on Agra.
2. Check the direct parent of Agra. It's Delhi. Is Delhi ==Agra?
   The answer is false. So now we call the find operation on Delhi.
3. Check the direct parent of Delhi. It's Delhi. Is Delhi==Delhi.
   The answer is true! So, our final answer for the parent of Mathura is Delhi.

This way, we keep going "up" the tree until we reach a root element. A root element is a node with its own parent - in our case, Delhi.

Example:

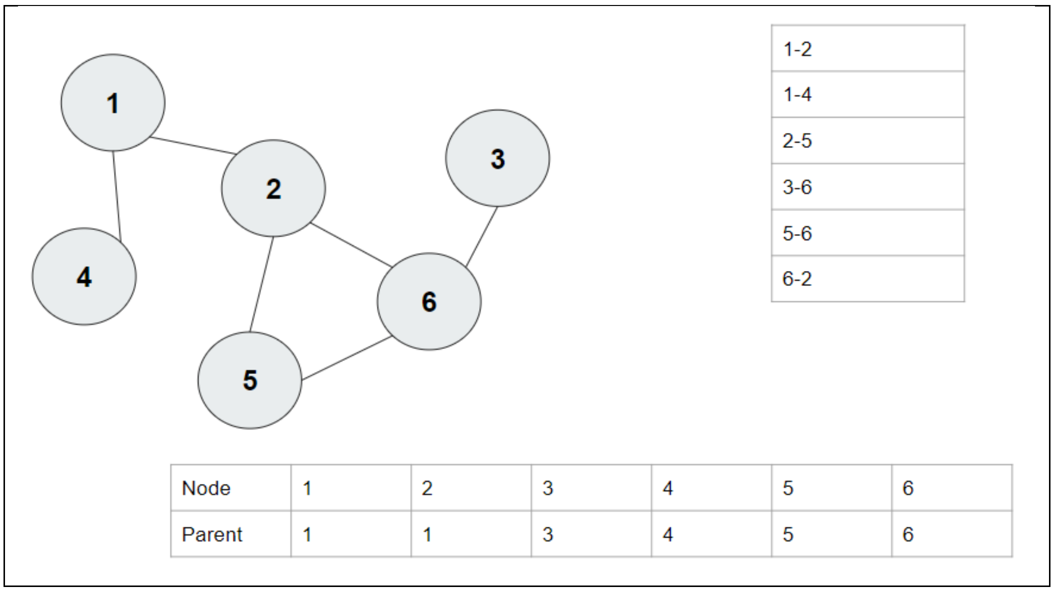


Let's try to find the parent of K in this example. (A and D are their own parents)

1. ParentOf(K) is I. Is I == K?
   False. So, we move upward, now using I.
2. ParentOf(I) is C. Is C== I?
   False. Move upward using C.
3. ParentOf(C) is A. Is A==C?
   False. Move upward using A.
4. ParentOf(A) is A. Is A==A?
   True! Our final answer is A.

Let's understand the use of a Disjoint Set Data Structure through a simple application - Finding a cycle in an undirected graph.

Find a Cycle in an Undirected Graph using Disjoint Set Data Structure
Let us understand the disjoint set operations in daa more clearly with the help of an example. Here, we will use a disjoint set data structure to find whether an undirected graph contains a cycle or not. We are assuming that our graph does not contain any self-loops.



1-2
1-4
2-5
3-6
5-6
6-2

Node	1	2	3	4	5	6
Parent	1	1	3	4	5	6

Let this be our graph. Each node is initialized as its own parent. The list of edges is shown on the top right, and the parents are shown at the bottom. We have shown each edge only once for simplicity (i.e., 1-2 and 2-1 aren't shown separately)

Algorithm

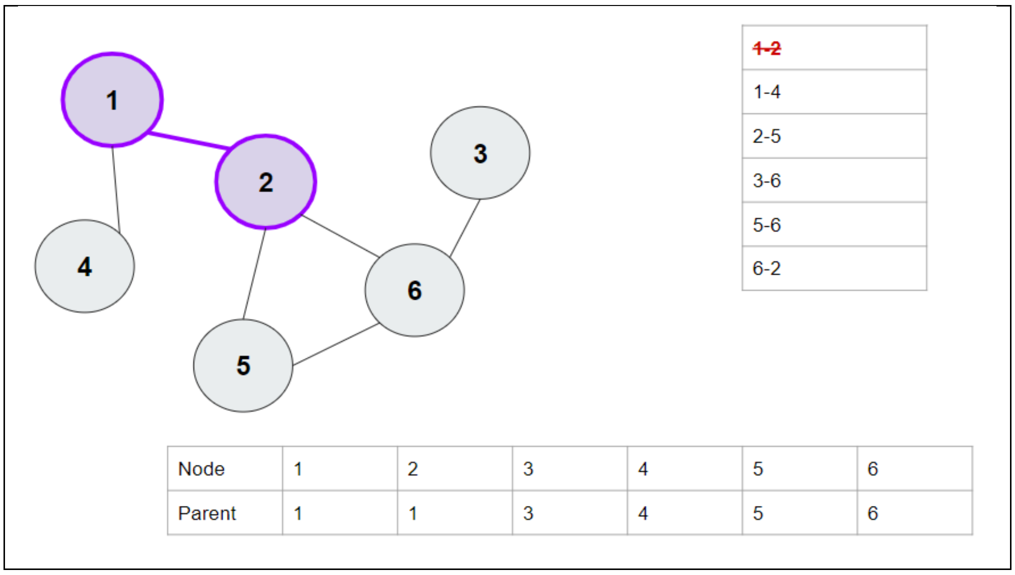
We will go through each edge and do the following. Let the edge be u-v.

1. Find the parent of u and v. If both parents are the same, it means u and v are part of the same subset, and we have found a cycle.v
2. If they are not the same, merge u and v. We don't merge u and v directly. Instead, we merge the parents of u and v. This ensures that all the siblings of u and all the siblings of v have the same common parent.

We do this for all edges.

Walkthrough

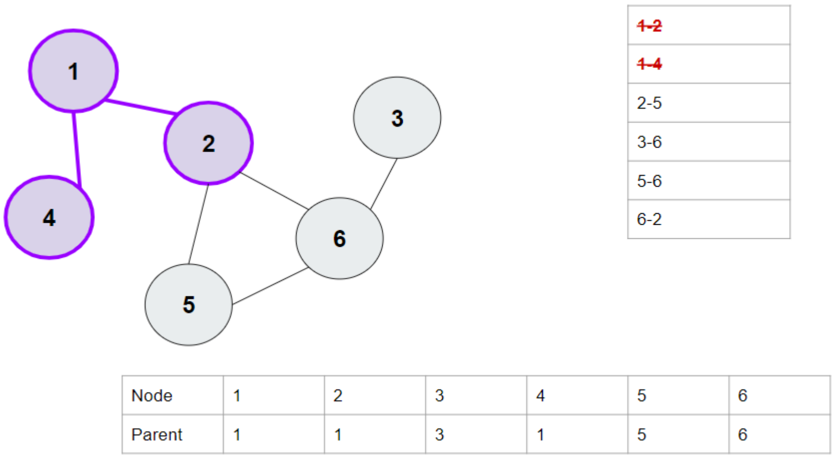
1. 1-2 Edge.
Parent of 1 is **1**, parent of 2 is **2**. They are different, so we will merge them. Set the parent of 2 as **1** (this is chosen randomly, setting the parent of 1 as 2 would also have been correct).



~~1-2~~
1-4
2-5
3-6
5-6
6-2

Node	1	2	3	4	5	6
Parent	1	1	3	4	5	6

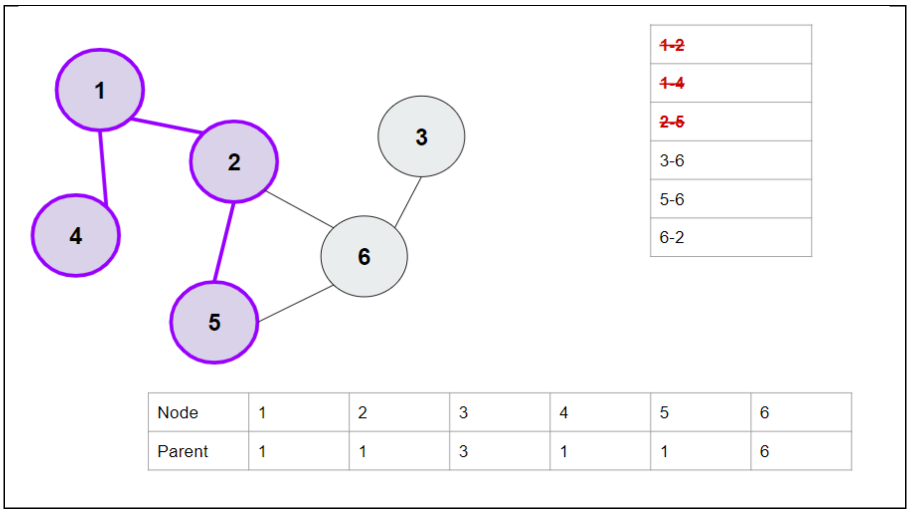
2. 1-4 Edge.
Parent of 1 is **1**. Parent of 4 is **4**. They are not the same, so merge 1 and 4. Set parent of **4** as **1.**



~~1-2~~
~~1-4~~
2-5
3-6
5-6
6-2

Node	1	2	3	4	5	6
Parent	1	1	3	1	5	6

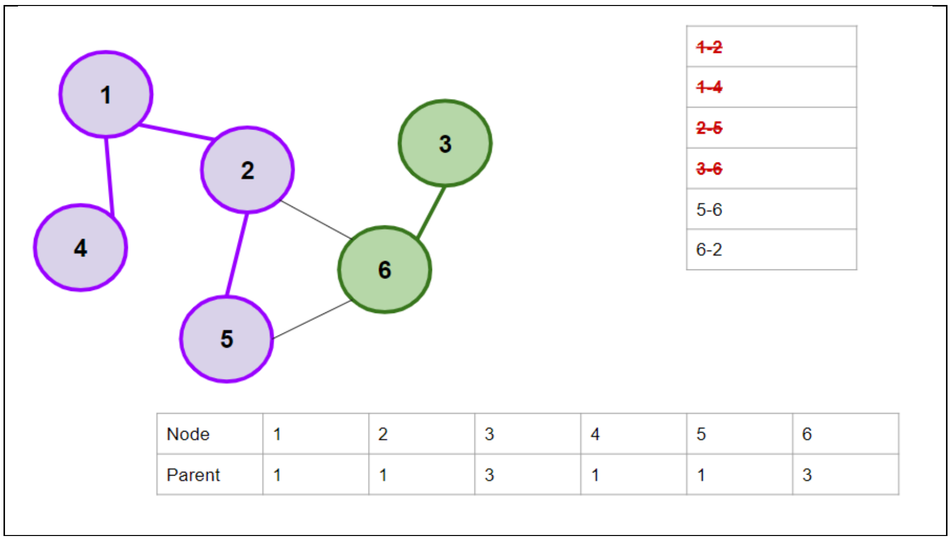
3. 2-5 Edge.
The parent of 2 is **1**, and the parent of 5 is **5**. They aren't the same, so we'll merge them. Set the parent of **5** as **1**.



~~1-2~~
~~1-4~~
~~2-5~~
3-6
5-6
6-2

Node	1	2	3	4	5	6
Parent	1	1	3	1	1	6

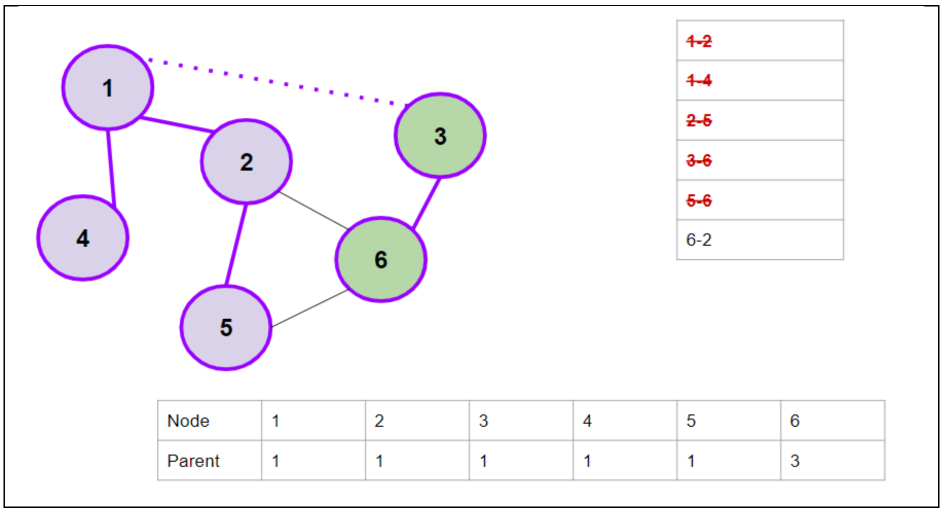
4. 3-6 Edge.
The parent of 3 is **3**, and the parent of 6 is **6**. They aren't the same, so we'll merge them. Set the parent of **6** as **3**.



~~1-2~~
~~1-4~~
~~2-5~~
~~3-6~~
5-6
6-2

Node	1	2	3	4	5	6
Parent	1	1	3	1	1	3

5. 5-6 Edge.
The parent of 5 is **1**, and the parent of 6 is **3**. They aren't the same, so we'll merge them. Set the parent of **3** as **1**.



~~1-2~~
~~1-4~~
~~2-5~~
~~3-6~~
~~5-6~~
6-2

Node	1	2	3	4	5	6
Parent	1	1	1	1	1	3

Note here - the direct parent of 6 is still 3. But, the actual parent becomes 1, because the parent of 3 is 1. (The dotted line is not an edge in the graph, it shows that the parent of 3 is 1)

6. 6-2 Edge.
The parent of 6 is 1 (actual parent), and the parent of 2 is also 1. **These are both the same.** Thus, our graph contains a cycle.

Complexities for Disjoint Set Operations

Regardless of the example or the purpose in which they are used, disjoint set operations in daa always have the same space and time complexities. The complexities for both find and union operations are:

The **space complexity** is O(n), where n is the number of vertices. The parent array stores the parent of each vertex. Hence, we get O(n) space complexity. We aren't considering the space complexity of the graph in this because the graph is the input here.

The **time complexity** is O(log N) in the average case because the find operation will have to go up a tree with n nodes. The height of a tree in the average case will be log N. In the worst case, we can have a skewed tree, and our time complexity will become O(n).

Applications

1) Lexicographically Smallest String

PROBLEM STATEMENT

You have given two strings ‘s’ and ‘t’ of same length, he knows that both strings are equivalent strings which means s[i], t[i] will follow all rules of any equivalence relation that are:

- Reflexive : s[i] = t[i]
- Symmetric: s[i] = t[i] => t[i] = s[i]
- Transitive: if s[i] = t[i] and t[i] = s[j] then s[i] = s[j]

You have to find the lexicographically smallest equivalent string of a given string ‘str’.

Note:

1. String ‘s’, ‘t’ and ‘str’ consist of only lowercase English letters from ‘a’ – ‘z’.
2. String ‘s’ and ‘t’ are of same length.

For example:

Let s = “abc” , t = “xyz” and str = “xbc” then all strings that we can generate are “abc”, “abz”, “ayc”,”ayz”, “xbc”, “xbz”, “xyc”, “xyz” and smallest of all these is “abc”.

Input Format:

The first line of input contains an integer 'T' representing the number of test cases.

The first and the only line of each test case contains three space-separated strings ‘s’, ‘t’ and ‘str’.

Output Format :

For each test case, print a single line containing a single string denoting the smallest equivalent string of ‘str’.

The output for each test case is printed in a separate line.

Note:

You do not need to print anything, it has already been taken care of. Just implement the given function.

Constraints :

1 <= T <= 5

1 <= N, M <= 5000

Where ‘T’ is the number of test cases, ‘N’ is the length of string ‘s’ and ‘t’ and ‘M’ is the length of ‘str’.

APPROACH :

The idea here is to use disjoint set union or union-find data structure.

It provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is.

Thus the basic interface of this data structure consists of only three operations:

- **union_sets(a, b)** - merges the two specified sets (the set in which the element **a** is located, and the set in which the element **b** is located), as we need the lexicographically smallest element as root so we will add one more condition while union two sets.

- **find_set(v)** - returns the representative (also called leader) of the set that contains the element **v**. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after **union_sets** calls). This representative can be used to check if two elements are part of the same set or not. **a** and **b** are exactly in the same set, if **find_set(a) == find_set(b)**. Otherwise, they are in different sets.

We iterate both string ‘s’ and ‘t’ and union all characters placed at the same index i.e. union(s[i],t[i]) where i goes from 0 -> s.length-1.

And to make the lexicographically smallest string we just return the find_set value of each character of ‘str’.

Algorithm:

- Declare an array parent of size 26 and initialize it with its index value.

  for(i : 0 -> s.length)

  - union_set(s[i],t[i])

- Declare an empty string res =””

- for(i : 0 -> str.length)

  - res += find_set(str[i])

- return res

  Description of union_set(int a, int b)

- first find representation of each set

  - a = find_set(a)

  - b = find_set(b)

- If (a != b)

  - if(a < b)

        - Parent[b] = a.
    - else
        - Parent[a] = b

Description of find_set(int v)

- if(v == parent[v]) return v.
- return parent[v] = find_set(v)

Program:

```
import java.util.*;
class Solution
{
        public String smallestEquivalentString(String s1, String s2, String s3)
        {
            UnionFind uf = new UnionFind();
            for (int i = 0; i < s1.length(); ++i)
            {
                int c1 = s1.charAt(i) - 'a';
                int c2 = s2.charAt(i) - 'a';
                uf.union(c1, c2);
            }

            // kmit ngit mgit

            StringBuilder sb = new StringBuilder();
            for (char ch : s3.toCharArray()) {
                sb.append((char)('a' + uf.find(ch - 'a')));
            }
            return sb.toString();
        }

        class UnionFind
        {
            int[] parent = new int[26];

             UnionFind()
            {
                for (int i = 0; i < 26; ++i)
                    parent[i] = i;
            }

            int find(int a)
            {
                if (a != parent[a]) {
                    parent[a] = find(parent[a]);
                }
                return parent[a];
            }
```

```
                void union(int c1, int c2)
                {
                        int parentC1 = find(c1);
                        int parentC2 = find(c2);

                        System.out.println("c1 " + c1 + " c2 " + c2 + " parentC1 " + parentC1
+ " parentC2 " + parentC2);

                        if (parentC1 == parentC2) return;

                        if (parentC1 < parentC2)
                                parent[parentC2] = parentC1;
                        else
                                parent[parentC1] = parentC2;
                }
          }
}

public class SmallestString
{
          public static void main(String args[])
          {
                Scanner sc=new Scanner(System.in);
                String s1=sc.next();
                String B=sc.next();
                String T=sc.next();
                System.out.println(new Solution().smallestEquivalentString(s1,B,T));
          }
}
```

Input Format:
Three space separated strings, A , B and T

Output Format:
Print a string, relatively smallest string of T.

Sample Input-1:
kmit ngit mgit

Sample Output-1:
ggit

Explanation:
The relative groups using A nd B are [k, n], [m, g], [i], [t] and
the relatively smallest string of T is "ggit"

Sample Input-2:
attitude progress apriori

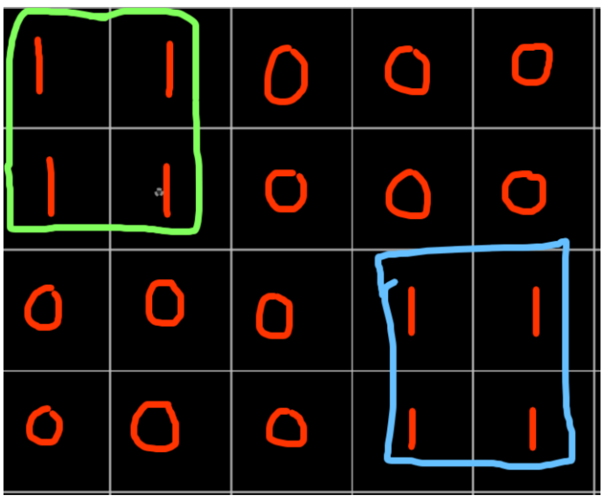
Sample Output-2:
aaogoog

Explanation:
The relative groups using A nd B are [a, p], [t, r, o], [i, g] and [u, e, d, s]
the relatively smallest string of T is "aaogoog"

2) Number of Distinct Islands:

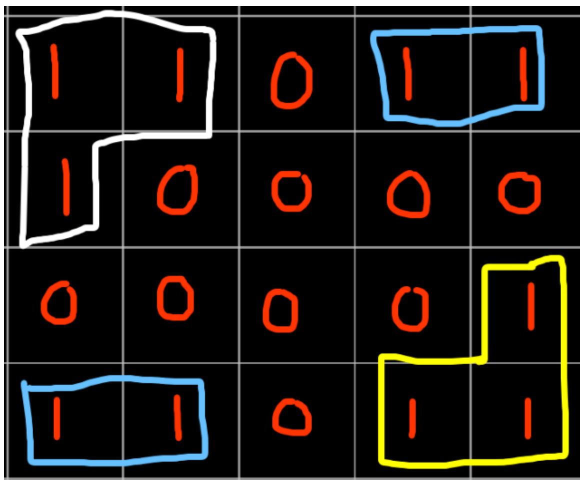
Number of Distinct Islands states that given a **n x m** binary matrix. An island is a group of 1's (representing land) connected **4-directionally** (horizontal or vertical).

An island is considered to be the same as another if and only if one island can be translated (and not rotated or reflected) to equal the other.
Example:
Sample input/output
Input: [[**1**,**1**,**0**,**0**,**0**],[**1**,**1**,**0**,**0**,**0**],[**0**,**0**,**0**,**1**,**1**],[**0**,**0**,**0**,**1**,**1**]]
Output: 1
Explanation:



- Check the above diagram for a better understanding.
- Note that we cannot rotate or reflect the orientation of the island.
- In the above figure, both the islands are identical, hence the number of distinct islands
- is 1.

Input: [[**1**,**1**,**0**,**1**,**1**],[**1**,**0**,**0**,**0**,**0**],[**0**,**0**,**0**,**0**,**1**],[**1**,**1**,**0**,**1**,**1**]]
Output: 3
Explanation:



- Check the above diagram for a better understanding.
- Note that islands on the top right corner and bottom left corner are identical, while islands on the top left corner and bottom right corner are different.
- Hence, the total number of distinct islands is 3.

Program:
```
import java.util.*;
class Solution
{
	private static final int[] dRow = {-1, 1, 0, 0};
	private static final int[] dCol = {0, 0, -1, 1};
	private class DisjointSet
	{
		private int[] parent;
		private int[] size;

		public DisjointSet(int V)
		{
			parent = new int[V];
			size = new int[V];
			for (int i=0; i<V; i++)
			{
				size[i] = 1;
				parent[i] = i;
			}
		}

		public int find(int u)
		{
			return parent[u] == u ? u : (parent[u] = find(parent[u]));
		}

		public void union(int u, int v)
		{
			int p1 = find(u);
			int p2 = find(v);
			if (p1 == p2)
```

```
                    return;

                if (size[p1] < size[p2])
                {
                    parent[p1] = p2;
                    size[p2] += size[p1];
                }
                else
                {
                    parent[p2] = p1;
                    size[p1] += size[p2];
                }
            }
        }

        public int numDistinctIslands(int[][] grid)
        {
            int nr = grid.length;
            int nc = grid[0].length;
            DisjointSet ds = new DisjointSet(nr * nc);

            for (int i=0; i<nr; i++)
            {
                for (int j=0; j<nc; j++)
                {
                    if (grid[i][j] == 1)
                    {
                        for (int k=0; k<4; k++)
                        {
                            int row = i + dRow[k];
                            int col = j + dCol[k];
                            if (row >= 0 && row < nr && col >= 0 && col <
                                                nc && grid[row][col] == 1)
                            {
                                ds.union(i*nc + j, row*nc + col);
                            }
                        }
                    }
                }
            }

            System.out.println("disjoint set parent " + Arrays.toString(ds.parent));
            System.out.println("disjoint set size " + Arrays.toString(ds.size));

            String []pattern = new String[nr*nc];

            for(int i=0; i < nr*nc; i++)
            pattern[i]= "";
            for(int i=0;i < nr; i++)
            {
```

```
                        for(int j=0; j<nc; j++)
                        {
                                if(grid[i][j]==0)
                                        continue;
                                int parent = ds.find(i*nc+j);
                                pattern[parent] += String.valueOf(i*nc + j-parent);
                        }
                }

                Set<String>tmp = new HashSet<>();
                for(int i=0;i<nr*nc; i++)
                {
                        if(pattern[i].length()!=0)
                        {
                                System.out.println("i " + i + " pattern[i] " + pattern[i]);
                                tmp.add(pattern[i]);
                        }
                }
                return tmp.size();
        }
}
public class DistinctIslands
{
        public static void main(String args[])
        {
                Scanner sc=new Scanner(System.in);
                int m=sc.nextInt();
                int n=sc.nextInt();
                int grid[][]=new int[m][n];
                for(int i=0;i<m;i++)
                        for(int j=0;j<n;j++)
                                grid[i][j]=sc.nextInt();
                System.out.println(new Solution().numDistinctIslands(grid));
        }
}
```

Input Format:
Line-1: Two space separated integers M and N, size of the wall.
Next M lines: N space separated integers, either 0 or 1.

Output Format:
Print an integer, Number of distinct shapes formed by Blue Lights.

Sample Input-1:
```
4 5
1 1 0 1 1
1 1 0 0 1
0 0 0 0 0
1 1 0 0 0
```
Sample Output-1:

3

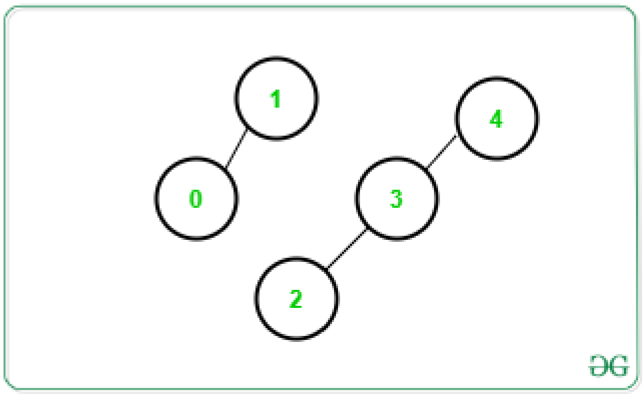
Sample Input-2:
5 5
1 1 0 1 1
1 0 0 0 1
0 0 0 0 0
1 0 0 0 1
1 1 0 1 1

Sample Output-2:
4

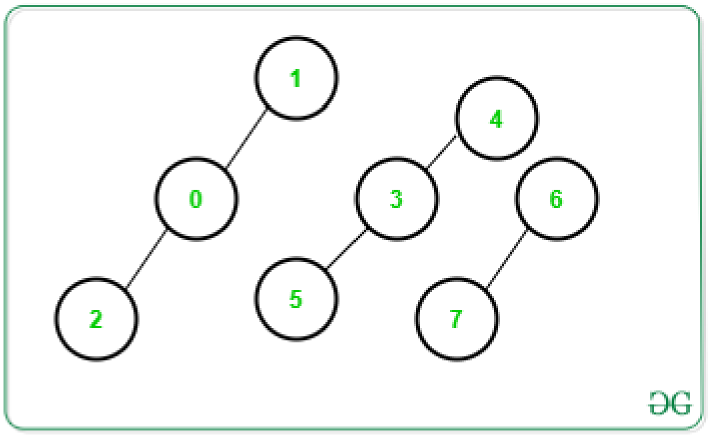
Note: The shapes,
```
1 1           1 1
1      and      1
```

3) Number of Connected Components in an Undirected Graph.

Given an undirected graph **G** with vertices numbered in the range **[0, N]** and an array **Edges[][]** consisting of **M** edges, the task is to find the total number of connected components in the graph using Disjoint Set Union algorithm.
Examples:
Input: *N = 5, Edges[][] = {{1, 0}, {2, 3}, {3, 4}}*
Output: *2*
Explanation: *There are only 2 connected components as shown below:*



Input: *N = 8, Edges[][] = {{1, 0}, {0, 2}, {3, 5}, {3, 4}, {6, 7}}*
Output: *3*
Explanation: *There are only 3 connected components as shown below:*



Approach: The problem can be solved using Disjoint Set Union algorithm. Follow the steps below to solve the problem:

- In DSU algorithm, there are two main functions, i.e. **connect()** and **root()** function.
- **connect():** Connects an edge.
- **root():** Recursively determine the topmost parent of a given edge.
- For each edge **{a, b},** check if **a** is connected to **b** or not. If found to be false, connect them by appending their top parents.
- After completing the above step for every edge, print the total number of the distinct top-most parents for each vertex.

Program:

```
import java.util.*;

class ConnectedComponents
{
        public int countComponents(int n, int[][] edges)
        {
                int[] parent = new int[n];
                int[] size = new int[n];
                for (int i = 0; i < n; i++)
                {
                        parent[i] = i;
                        size[i] = 1;
                }

                int components = n;
                for (int[] e : edges)
                {
                        int p1 = findParent(parent, e[0]);
                        int p2 = findParent(parent, e[1]);
                        System.out.println("p1 " + p1 + " p2 " + p2);
                        if (p1 != p2)
                        {
                                if (size[p1] < size[p2])
                                { // Merge small size to large size
                                        size[p2] += size[p1];
                                        parent[p1] = p2;
                                }
                                else
                                {
                                        size[p1] += size[p2];
                                        parent[p2] = p1;
                                }
                                components--;
                        }
                }
                for (int i = 0; i < n; i++)
                System.out.println("i " + i + " parent[i] " + parent[i] + " size[i] " + size[i]);

                return components;
        }
        private int findParent(int[] parent, int i)
        {
                if (i == parent[i]) return i;
                return parent[i] = findParent(parent, parent[i]); // Path compression
        }

        public static void main(String args[])
        {
                Scanner sc= new Scanner(System.in);
                int n=sc.nextInt();
                int e=sc.nextInt();
                int edges[][]=new int[e][2];
                for(int i=0;i<e;i++)
                        for(int j=0;j<2;j++)
                                edges[i][j]=sc.nextInt();
                System.out.println(new onnectedComponents().countComponents(n,edges));
        }
}
```

Input Format:

Line-1: Two space separated integers N and M, number of cities and routes
Next M lines: Two space separated integers city1, city2.

Output Format:
-
Print an integer, number of regions formed.

Sample Input-1:
--
```
5 4
0 1
0 2
1 2
```

```
3 4
```

Sample Output-1:

```
---
2
```

Sample Input-2:

```
--
5 6
0 1
0 2
2 3
1 2
1 4
2 4
```

Sample Output-2:

```
---
1
```