SYLLABUS:

Disjoint Set Union: Disjoint set and its operations, Union Find Algorithm, Lexicographically Smallest Equivalent String, Number of Distinct Islands, and Number of Connected Components in an Undirected Graph.

Introduction:

- In this section we study the use of forests in the representation of sets. We shall assume that the elements of the sets are the numbers 1, 2, 3, ..., n.
- These numbers might, in practice, be indices into a symbol table in which the names of the elements are stored. We assume that the sets being represented are pairwise disjoint (that is, if S_i and S_j $i \neq j$, are two sets, then there is no element that is in both S_i and S_j).
- For example, when n = 10, the elements can be partitioned into three disjoint sets,
- $S_1 = \{1, 7, 8, 9\}$, $S_2 = \{2, 5, 10\}$, and $S_3 = \{3, 4, 6\}$.
- Below Figure 1 shows one possible representation for these sets. In this representation, each set is represented as a tree.
- Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children.
- The reason for this change in linkage becomes apparent when we discuss the implementation of set operations.

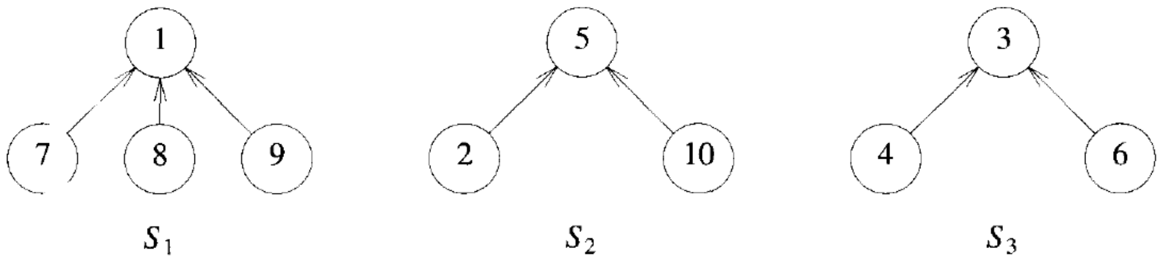


Figure 1: Possible tree representation of sets.

The operations we wish to perform on these sets are:

1. **Disjoint set union:** If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = all elements *x* such that *x* is in S_i or S_j. i.e $\mathbf{S_i \cap S_j = \emptyset}$. Thus, $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$
   Since we have assumed that all sets are disjoint, we can assume that following the union of S_1 and S_2, the sets S_1 and S_2 do not exist independently; that is, they are replaced by $S_1 \cup S_2$ in the collection of sets.

2. **Find(*i*).** Given the element *i*, find the set containing *i*. Thus, 4 is in set S_3, and 9 is in set S_1.

Union and Find Operations:

Let us consider the union operation first. Suppose that we wish to obtain the union of S_1 and S_2 (from Figure 1). Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other. $S_1 \cup S_2$ could then have one of the representations of Figure 2.

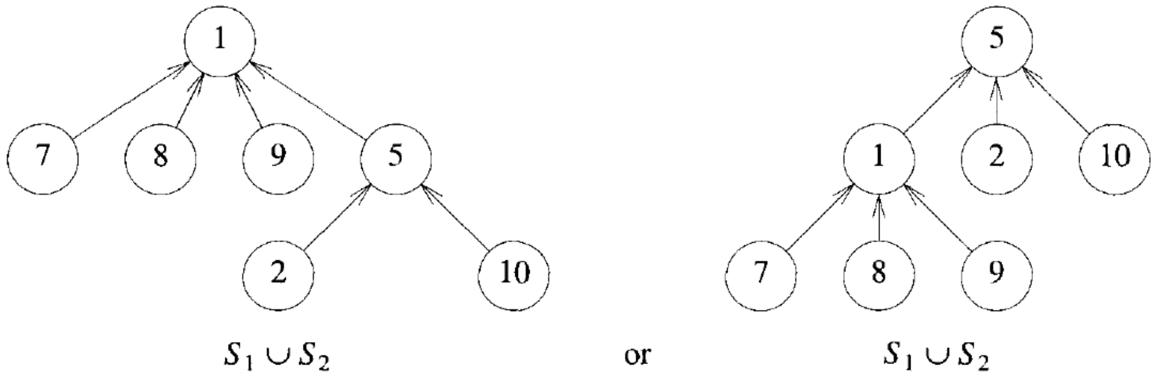


Figure 2: Possible tree representation of $S_1 \cup S_2$.

- To obtain the union of two sets, all that has to be done is to set the parent field of one of the roots to the other root.
- This can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set.
- If, in addition, each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name.
- The data representation for S_1, S_2, and S_3 may then take the form shown in Figure 3.

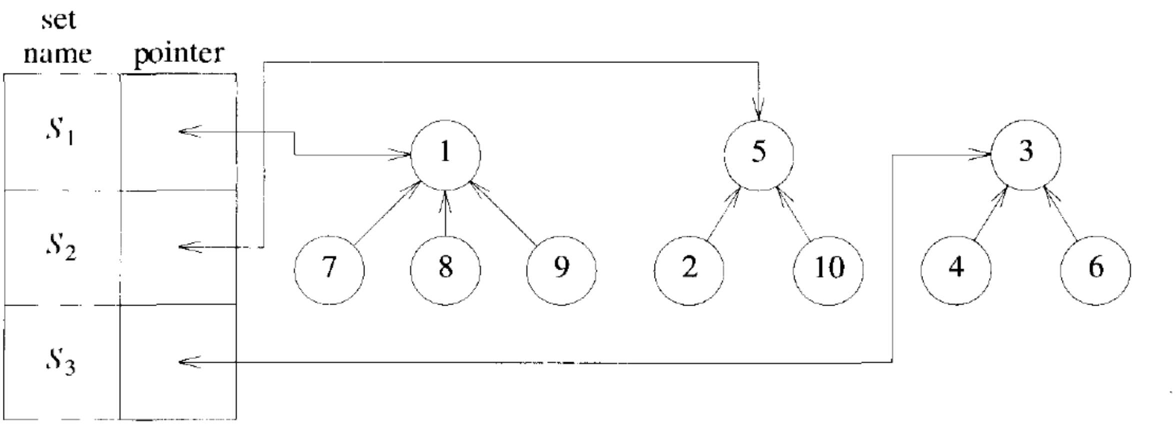


Figure 3: Data representation for S_1, S_2 ***and*** S_3.

- **In presenting the union and find algorithms**, we ignore the set names and identify sets just by the roots of the trees representing them.
- This simplifies the discussion. The transition to set names is easy. If we determine that element *i* is in a tree with root *j*, and *j* has a pointer to entry k in the set name table, then the set name is just name[k].
- If we wish to unite sets S_i and S_j, then we wish to unite the trees with roots ***FindPointer(S_i)*** and ***FindPointer(S_j)***.
- Here **FindPointer is a function that takes a set name and determines the root of the tree that represents it**. This is done by an examination of the [set name, pointer] table.
- In many applications the set name is just the element at the root.
- The operation of Find(i) now becomes: Determine the root of the tree containing element i.

- The function Union(i, j) requires two trees with roots i and j be joined. Also to simplify, assume that the set elements are the numbers 1 through n.
- Since the set elements are numbered 1 through n, we represent the tree nodes using an array p[1 : n], where n is the maximum number of elements.
- The i^{th} element of this array represents the tree node that contains element i.
- This array element gives the parent pointer of the corresponding tree node.
- **Figure 4**. shows this representation of the sets S_1, S_2, and S_3 of Figure 1. Notice that root nodes have a parent of −1.

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	−1	5	−1	3	−1	3	1	1	1	5

Figure 4: Array representation of $\mathbf{S_1, S_2}$ ***and*** $\mathbf{S_3}$ of **Figure 1.**

- We can now implement Find(i) by following the indices, starting at *i* until we reach a node with parent value −1.
- For example, Find(6) starts at 6 and then moves to 6's parent, 3. Since *p*[3] is negative, we have reached the root.
- The operation Union(i, j) is equally simple. We pass in two trees with roots *i* and *j*. Adopting the convention that the first tree becomes a subtree of the second, the statement p[i] := j; accomplishes the union.

```
Algorithm SimpleUnion(i, j)
{
    p[i] := j;
}

Algorithm SimpleFind(i)
{
    while (p[i] ≥ 0) do i := p[i];
    return i;
}
```

Algorithm-1: Simple Algorithms for union and Find

- **Algorithm-1** gives the descriptions of the union and find operations just discussed.
- Although these two algorithms are very easy to state, their performance characteristics are not very good.
- For instance, if we start with *q* elements each in a set of its own (that is, $S_i=\{i\}, 1 \le i \le q$), then the initial configuration consists of a forest with *q* nodes, and $p[i]=0, 1\le i\le q$. Now let us process the following sequence of *union–find* operations:

Union(1,2), Union(2,3), Union(3,4), Union(4,5),…,Union(n−1,n)

Find(1), Find(2),…, Find(n)

- This sequence results in the degenerate tree of **Figure 5**.
- Since the time taken for a union is constant, the n−1 unions can be processed in time O(n).
- However, each **find** requires following a sequence of parent pointers from the element to be found to the root.
- Since the time required to process a find for an element at level i of a tree is O(i), the total time needed to process the n **finds** is

$$O\left(\sum_{i=1}^{n} i\right) = O(n^2)$$

- **We can improve the performance of our union and find algorithms by avoiding the creation of degenerate trees.**
- **To accomplish this, we make use of a weighting rule for Union(i,j).**

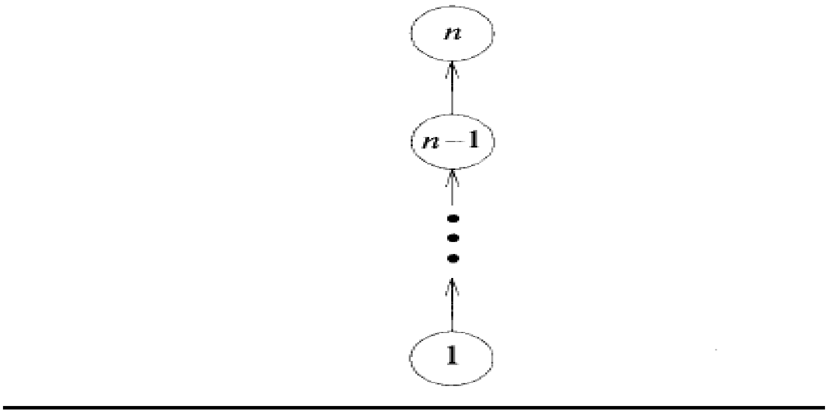
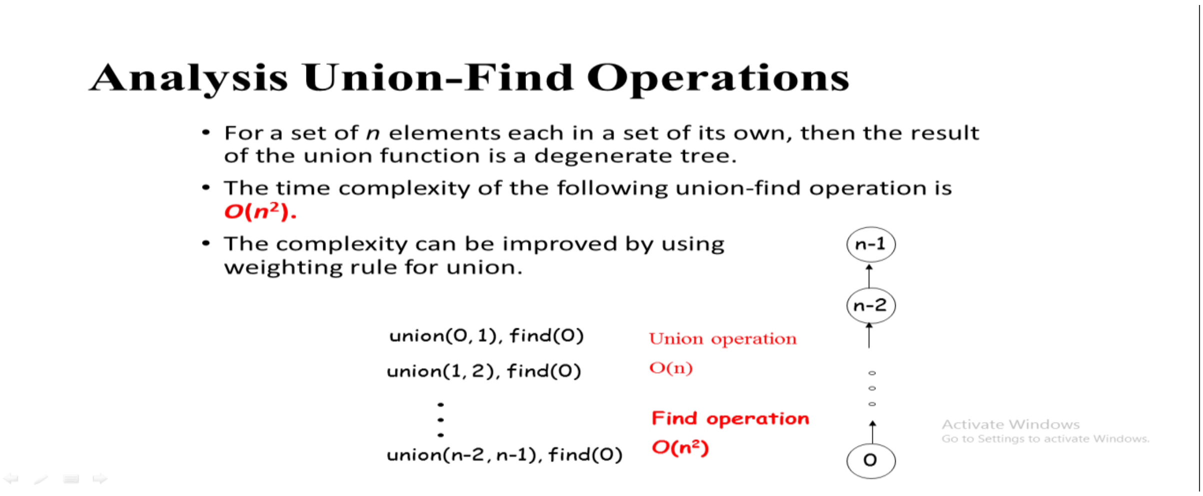


Figure 5: Degenerate Tree



Forest Tree: In data structures**,** forest is a set of zero or more disjoint trees**.** Each tree in a forest is a separate hierarchical structure. No node is shared between trees. The trees are independent of each other.

Weighted Rule for Union (i,j):

Definition: "*If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i ; otherwise make i the parent of j.*"

- When we use the weighting rule to perform the sequence of set unions given before, we obtain the trees of Figure 6.
- In this figure, the unions have been modified so that the input parameter values correspond to the roots of the trees to be combined.
- To implement the weighting rule, we need to know how many nodes there are in every tree.
- To do this easily, we maintain a count field in the root of every tree.
- If i is a root node, then count[i] equals the number of nodes in that tree.
- Since all nodes other than the roots of trees have a positive number in the *p* field, we can maintain the count in the *p* field of the roots as a negative number.
- Using this convention, we obtain Algorithm 2. In this algorithm the time required to perform a union has increased somewhat but is still bounded by a constant (that is, it is O(1)).
- The find algorithm remains unchanged. The maximum time to perform a find is determined by **Lemma 1**.

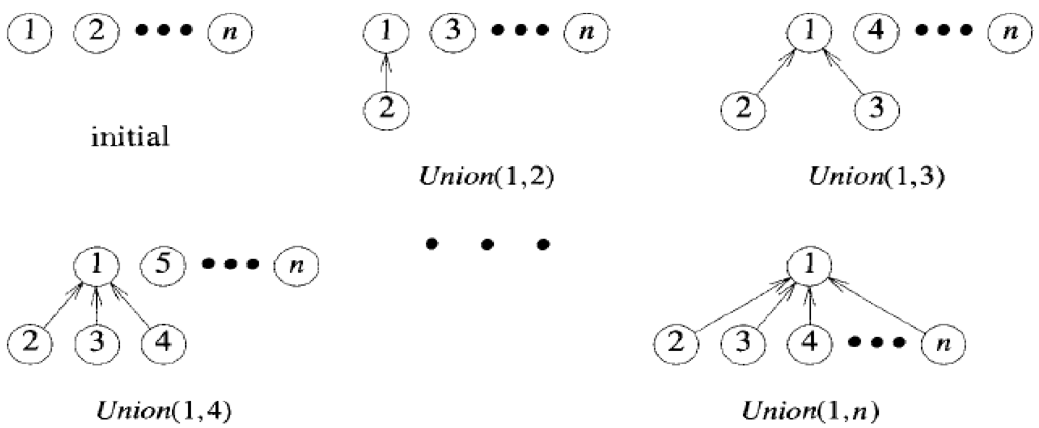


Figure 6: Trees obtained using weighted rule

```
Algorithm WeightedUnion(i, j)
// Union sets with roots i and j, i ≠ j, using the
// weighting rule. p[i] = −count[i] and p[j] = −count[j].
{
    temp := p[i] + p[j];
    if (p[i] > p[j]) then
    { // i has fewer nodes.
        p[i] := j; p[j] := temp;
    }
    else
    { // j has fewer or equal nodes.
        p[j] := i; p[i] := temp;
    }
}
```

Algorithm-2: Union Algorithm with weighting rule.

Lemma-1: Assume that we start with a forest of trees, each having one node. Let *T* be a tree with *m* nodes created as a result of a sequence of unions each performed using Weighted Union. **The height of *T* is no greater than $\lfloor \log_2 m \rfloor + 1$.**

Example-1: Consider the behavior of WeightedUnion on the following sequence of unions starting from the initial configuration p[i] = −count[i] = −1, $1 \le i \le 8 = n$:

Union(1,2), Union(3,4), Union(5,6), Union(7,8), Union(1,3), Union(5,7), Union(1,5)

The trees of **Figure 7** are obtained. As is evident, the height of each tree with mm nodes is $\lfloor \log 2m \rfloor + 1$

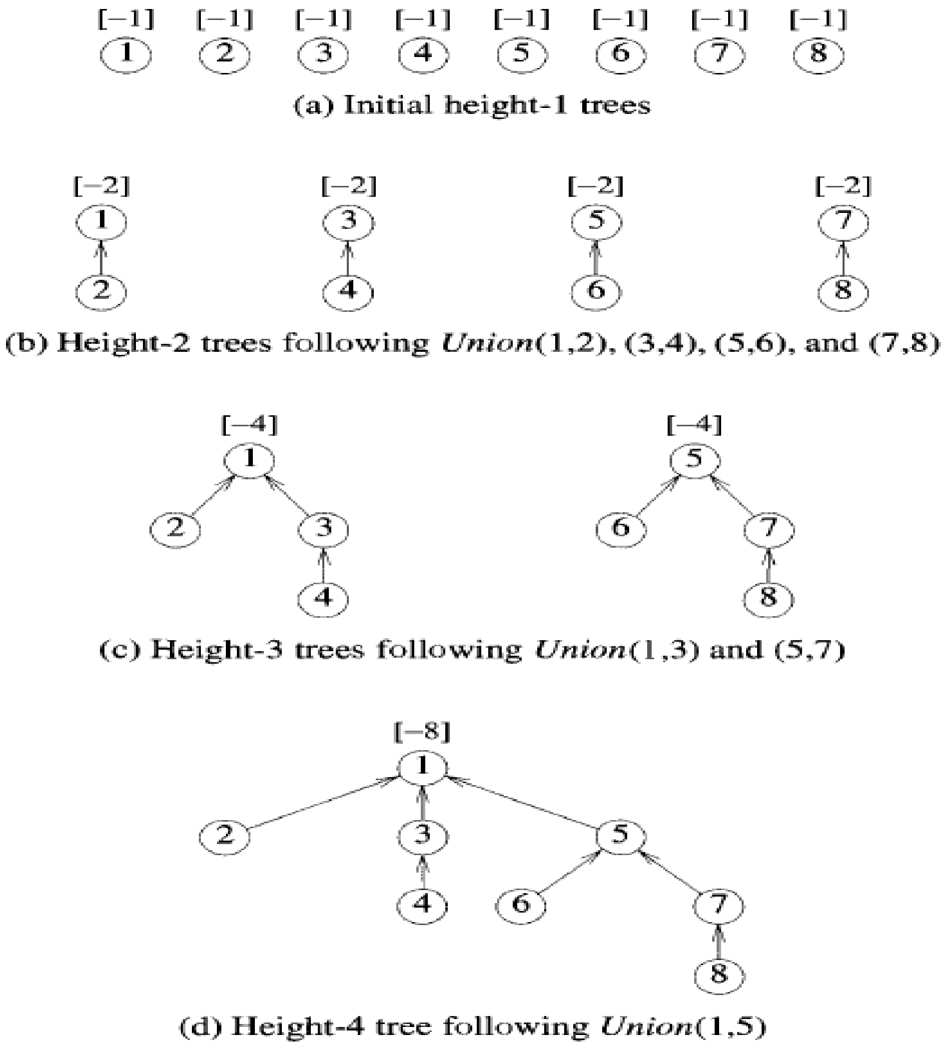


Figure 7: Trees achieving worst-case bound

- From Lemma 1, it follows that the time to process a **find is O(log m)** if there are *m* elements in a tree.
- If an intermixed sequence of u−1 union and *f* find operations is to be processed, the time becomes O(u+ f log u), as no tree has more than u nodes in it.
- Of course, we need O(n) additional time to initialize the *n-tree* forest.

Collapsing Rule(Path Compression):

Definition: *"If j is a node on the path from i to its root and parent[i]≠ root(i), then set parent[j] to root(i)."*

- **CollapsingFind (Algorithm 3)** incorporates the Collapsing Rule

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i. Use the
// collapsing rule to collapse all nodes from i to the root.
{
    r := i;
    while (p[r] > 0) do r := p[r]; // Find the root.
    while (i ≠ r) do  // Collapse nodes from i to root r.
    {
        s := p[i]; p[i] := r; i := s;
    }
    return r;
}
```

Algorithm-3: Find Algorithm with Collapsing rule.

Example-2: Consider the tree created by WeightedUnion on the sequence of unions of Example-1. Now process the following eight finds:

Find(8), Find(8),…, Find(8)

- If **SimpleFind** is used, each Find(8) requires going up three parent link fields for a total of 24 moves to process all eight finds.
- When CollapsingFind is used, the first Find(8) requires going up three links and then resetting two links.
- Note that even though only two parent links need to be reset, CollapsingFind will reset three (the parent of 5 is reset to 1).
- Each of the remaining seven finds requires going up only one link field.
- The total cost is now only 13 moves.

Time Complexity:

Without Path Compression:

- **Union**: $O(1)$ per operation (with weighted union).
- **Find** $O(\log n)$ per operation (since tree height is logarithmic due to weighted union).
- **Total for *m* operations**: $O(m \log n)$.

With Path Compression + Weighted Union:

- Amortized Time per Operation: $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function (extremely slow-growing, effectively a constant for practical purposes).
- Total for m operations: $O(m \cdot \alpha(n))$.

Java Pprgram for Weighted Union and Collapsing Find: WeightedUnion.java

```
import java.util.*;

public class WeightedUnion
{
   private int[] parent; // parent[i] = root of i (if >= 0) or size of set (if < 0)
   private int count;    // Number of disjoint sets

   public WeightedUnion(int n)
   {
      parent = new int[n];
      count = n;
      Arrays.fill(parent, -1); // Initialize all as roots with size 1
   }
   // Performs Weighted Union (Union by Size) on sets containing i and j.
   public void weightedUnion(int i, int j)
   {
      int rootI = find(i);
      int rootJ = find(j);

      if (rootI == rootJ) return; // Already connected

      // Union by size: smaller tree attaches to the larger one
      if (parent[rootI] < parent[rootJ])
      { // rootI has larger size
         parent[rootI] += parent[rootJ];
         parent[rootJ] = rootI;
      } else {
         parent[rootJ] += parent[rootI];
         parent[rootI] = rootJ;
      }
      count--;
   }

   //Finds the root of i with path compression.

   public int find(int i)
  {
      if (parent[i] < 0) return i;
      parent[i] = find(parent[i]); // Path compression
      return parent[i];
   }

   public int getCount()
   {
      return count;
   }

   public static void main(String[] args)
```

```
    {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter number of elements: ");
        int n = scanner.nextInt();
        WeightedUnion uf = new WeightedUnion(n);

        System.out.println("Enter union operations (pairs of indices, -1 to stop):");
        while (true)
        {
            int i = scanner.nextInt();
            if (i == -1) break;
            int j = scanner.nextInt();
            uf.weightedUnion(i, j);
        }

        System.out.println("Enter find queries (indices, -1 to stop):");
        while (true) {
            int i = scanner.nextInt();
            if (i == -1) break;
            System.out.println("Root of " + i + ": " + uf.find(i));
        }
        // Final state
        System.out.println("Number of disjoint sets: " + uf.getCount());
        System.out.println("Parent array: " + Arrays.toString(uf.parent));
    }
}
```

Input:

```
Enter number of elements: 5
Enter union operations (pairs of indices, -1 to stop):
0 1
2 3
0 4
3 4
-1
Enter find queries (indices, -1 to stop):
4
Root of 4: 1
2
Root of 2: 1
-1
```

Output=

```
Number of disjoint sets: 1
Parent array: [1, -5, 1, 1, 1]
```

Applications:

1. **Lexicographically Smallest Equivalent String.**
2. **Number of Distinct Islands.**
3. **Number of Connected Components in an Undirected Graph.**

1. Lexicographically Smallest Equivalent String:

You are given two strings of the same length s1 and s2 and a string baseStr.
We say s1[i] and s2[i] are equivalent characters.

For example, if s1 = "abc" and s2 = "cde", then we have 'a' == 'c', 'b' == 'd', and 'c' == 'e'.
Equivalent characters follow the usual rules of any equivalence relation:
Reflexivity: 'a' == 'a'.
Symmetry: 'a' == 'b' implies 'b' == 'a'.
Transitivity: 'a' == 'b' and 'b' == 'c' implies 'a' == 'c'.

For example, given the equivalency information from s1 = "abc" and s2 = "cde", "acd" and "aab" are equivalent strings of baseStr = "eed", and "aab" is the lexicographically smallest equivalent string of baseStr.
Return the lexicographically smallest equivalent string of baseStr by using the equivalency information from s1 and s2.

Example 1:
Input: s1 = "parker", s2 = "morris", baseStr = "parser"
Output: "makkek"
Explanation: Based on the equivalency information in s1 and s2, we can group their characters as [m,p], [a,o], [k,r,s], [e,i].
The characters in each group are equivalent and sorted in lexicographical order.
So the answer is "makkek".

Example 2:
Input: s1 = "hello", s2 = "world", baseStr = "hold"
Output: "hdld"
Explanation: Based on the equivalency information in s1 and s2, we can group their characters as [h,w], [d,e,o], [l,r].
So only the second letter 'o' in baseStr is changed to 'd', the answer is "hdld".

Example 3:
Input: s1 = "leetcode", s2 = "programs", baseStr = "sourcecode"
Output: "aauaaaaada"
Explanation: We group the equivalent characters in s1 and s2 as [a,o,e,r,s,c], [l,p], [g,t] and [d,m], thus all letters in baseStr except 'u' and 'd' are transformed to 'a',
the answer is "aauaaaaada".

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)

Java Program for Lexicographically Smallest Equivalent String:

LexSmallestEquivalentString.java

```
import java.util.*;
class LexSmallestEquivalentString
{

   private class UnionFind
   {
      private int[] parent;

      private UnionFind(int n)
      {
         parent = new int[n];
         for(int i=0;i<n;i++)
         {
            parent[i] = i;
         }
      }
      private int getAbsoluteParent(int i)
      {
         if(parent[i]==i)
         {
            // absolute parent
            return i;
         }
         parent[i]=getAbsoluteParent(parent[i]);
         return parent[i];
      }

      private void unify(int i, int j)
      {
         int absoluteParentI = getAbsoluteParent(i);
         int absoluteParentJ = getAbsoluteParent(j);
         if(absoluteParentI<absoluteParentJ)
         {

            parent[absoluteParentJ] = absoluteParentI ;
         }
         else
         {

            parent[absoluteParentI] = absoluteParentJ ;
         }
      }

   }
   public String smallestEquivalentString(String s1, String s2, String baseStr)
    {
```

```
        UnionFind uf = new UnionFind(26);

        StringBuilder sb  = new StringBuilder();

        for(int i=0 ; i<s1.length() ; i++){

            int charS1 = s1.charAt(i)-'a';
            int charS2 = s2.charAt(i)-'a';

            uf.unify(charS1, charS2 );
        }

        for(int i=0 ; i<baseStr.length() ; i++)
        {
            int smallestMappedChar  = uf.getAbsoluteParent(baseStr.charAt(i)-'a');

            sb.append((char)(smallestMappedChar+'a'));
        }

        return sb.toString();
    }


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

Example 1:
Input =attitude progress apriori
Output =aaogoog

Example 2:
Input =kmit ngit mgit
Output =ggit

Example 3:
Input =hello world hold
Output =hdld

2. Number of Distinct Islands:

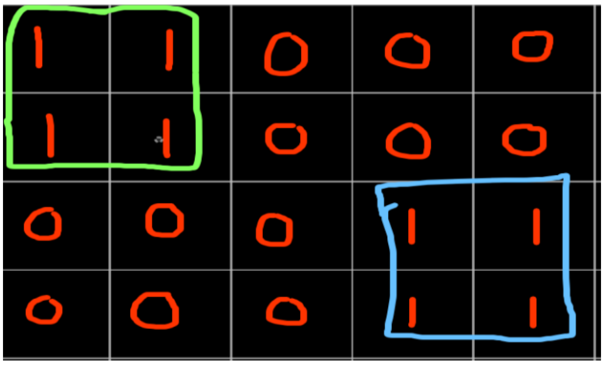
Number of Distinct Islands states that given a **n x m** binary matrix. An island is a group of 1's (representing land) connected **4-directionally** (horizontal or vertical).

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-1:
Input

Output: 1

Explanation:



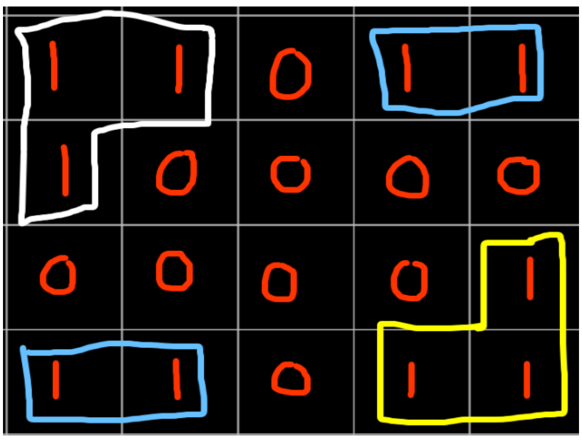
- Check the above diagram for a better understanding.
- Note that we cannot rotate or reflect the orientation of the island.
- In the above figure, both the islands are identical, hence the number of distinct islands is 1.

Example-2:

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.

Procedure:

Step 1: Initialize Structures

1. Let distinctShapes be an empty HashSet to store unique island shape strings.
2. Create a UnionFind class instance with size rows * cols (to handle each cell as a node).
3. Define 4 directions: up, down, left, right as dirs = {{1,0}, {-1,0}, {0,1}, {0,-1}}.

Step 2: Traverse the Grid

4. Loop through each cell (i, j):
   - If grid[i][j] == 1:
     - Create List<int[]> islandCoords = new ArrayList<>().
     - Call dfs(grid, i, j, i, j, islandCoords) to collect all connected land cells of the island.

Step 3: DFS (Depth-First Search)

5. In DFS:
   - Base case: return if i or j is out of bounds, already visited (grid[i][j] != 1).
   - Mark the cell as visited: grid[i][j] = 0.
   - Add the cell (i, j) to islandCoords.
   - Recurse in all 4 directions.

Step 4: Union-Find Mapping

6. After DFS collects all island coordinates:
   - Let baseX = islandCoords.get(0)[0], baseY = islandCoords.get(0)[1].
   - Convert the 2D coordinate (x, y) to a 1D index using formula: index = x * cols + y
   - Let baseIndex = baseX * cols + baseY.
   - For each (x, y) in islandCoords:
     - Compute idx = x * cols + y.
     - Call union(baseIndex, idx) to group all cells of this island in DSU.

Step 5: Normalize Island Shape

7. Sort islandCoords based on (row, col).
8. Let (baseX, baseY) = first coordinate in the sorted list.
9. Build a string by storing each coordinate as relative positions:
   - For each (x, y) in islandCoords:
     - Append (x - baseX):(y - baseY), to a StringBuilder.

   Example:
   For island at coordinates [(2,3), (2,4), (3,3), (3,4)]
   Normalized = 0:0,0:1,1:0,1:1,

Step 6: Store Shape

10. Add the normalized shape string to the distinctShapes set.

Step 7: Final Output

11. After processing all cells, return distinctShapes.size() as the number of unique island shapes.

Java Program for Number of Distinct Islands: DistinctIslandsDSF.java

```
import java.util.*;

class DistinctIslandsDSU
{

    static class UnionFind
    {
        int[] parent;

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

        int find(int x)
        {
            if (x != parent[x])
            {
                parent[x] = find(parent[x]); // path compression
            }
            return parent[x];
        }

        void union(int x, int y)
        {
            int rootX = find(x);
            int rootY = find(y);
            if (rootX != rootY)
            {
                parent[rootY] = rootX;
            }
        }
    }

    static int[][] dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
```

```
public static int numDistinctIslands(int[][] grid)
{
    int rows = grid.length;
    int cols = grid[0].length;


    Set<String> distinctShapes = new HashSet<>();
    UnionFind uf = new UnionFind(rows * cols);

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            if (grid[i][j] == 1)
            {
                List<int[]> islandCoords = new ArrayList<>();

                dfs(grid,i,j, i, j, islandCoords);

                // Union all positions in the island
                int baseIndex = islandCoords.get(0)[0] * cols + islandCoords.get(0)[1];
                for (int[] coord : islandCoords)
                {
                    int idx = coord[0] * cols + coord[1];
                    uf.union(baseIndex, idx);
                }

                // Normalize shape
                 islandCoords.sort((a, b) ->
                {
                     if (a[0] == b[0])
                    {
                     return Integer.compare(a[1], b[1]);
                    }
                    else
                    {
                     return Integer.compare(a[0], b[0]);
                     }
                });
                int baseX = islandCoords.get(0)[0];
                int baseY = islandCoords.get(0)[1];
                StringBuilder shape = new StringBuilder();
                for (int[] coord : islandCoords)
                {
                    shape.append((coord[0] - baseX)).append(":").append((coord[1] - baseY)).append(",");
                }
                distinctShapes.add(shape.toString());
            }
        }
    }
```

```
        return distinctShapes.size();
    }

    static void dfs(int[][] grid,int baseX, int baseY, int i, int j, List<int[]> coords)
    {
        int rows = grid.length, cols = grid[0].length;
        if (i < 0 || j < 0 || i >= rows || j >= cols || grid[i][j] != 1)
            return;

        grid[i][j] = 0;
        coords.add(new int[]{i, j});
        for (int[] dir : dirs)
        {
            dfs(grid,baseX, baseY, i + dir[0], j + dir[1], coords);
        }
    }

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);


        int rows = sc.nextInt();

        int cols = sc.nextInt();

        int[][] grid = new int[rows][cols];

        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                grid[i][j] = sc.nextInt();

        int result = numDistinctIslands(grid);
        System.out.println(result);
    }
}
```

Sample input's and output's:

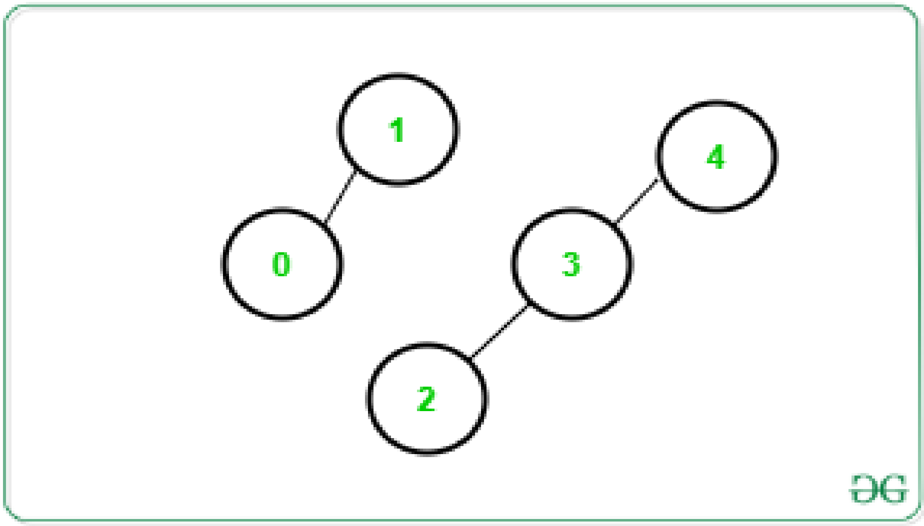
Example: 1

Input =4 5
1 1 0 0 0
1 1 0 0 0
0 0 0 1 1
0 0 0 1 1
Output =1

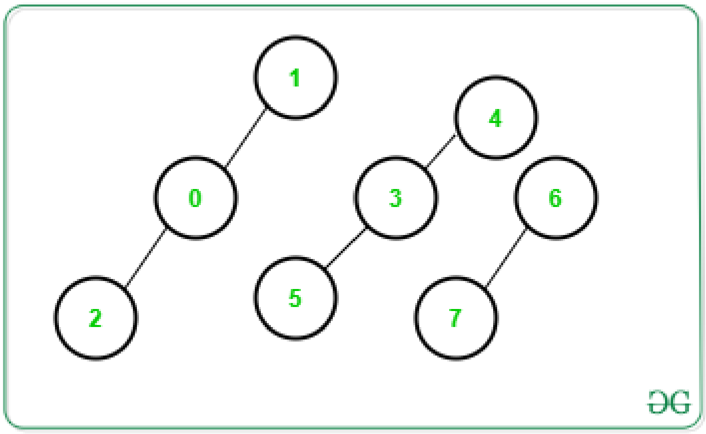
3. Number of Connected Components in an Undirected Graph:

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.

<u>Java Program for Number of Connected Components in an Undirected Graph:</u>

ConnectedComponents.java

```
import java.util.*;

class ConnectedComponents
{
   int[] parent;
   int[] size;
   public int countComponents(int n, int[][] edges)
   {
      parent = new int[n];
      size = new int[n];
      for (int i = 0; i < n; i++)
      {
         parent[i] = -1;
         size[i] = 1;
      }
      int components = n;
      for (int[] e : edges)
      {
         int p1 = find(e[0]);
         int p2 = find(e[1]);
         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--;
         }
      }
      return components;
   }
   private int find(int i)
   {
                  while(parent[i]>=0)
```

```
                i = parent[i];
            return i;
    }

    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 ConnectedComponents().countComponents(n,edges));
        }
}
```

Case =1
Input =5 6
0 1
0 2
2 3
1 2
1 4
2 4
Output =1
Case =2
Input =5 4
0 1
0 2
1 2
3 4
Output =2
Case =3
Input =6 3
0 1
2 3
4 5
Output =3