lab2

Trie

```java
import java.util.Arrays;
public class Trie {
    private Trie[] children;
    // private boolean isEnd;
    private int wordCount, prefixCount;
    public Trie() {
        children = new Trie[26];
        Arrays.fill(children, null);
        // isEnd=false;
        prefixCount = 0;
        wordCount = 0;
    }
    public void insert(String word) {
        Trie cur = this;
        for (char letter : word.toCharArray()) {
            int index = letter - 'a';
            if (cur.children[index] == null)
                cur.children[index] = new Trie();
            cur = cur.children[index];
            cur.prefixCount++;
        }
        // cur.isEnd=true;
        cur.wordCount++;
    }
    private Trie searchPrefix(String prefix) {
        Trie cur = this;
        for (char letter : prefix.toCharArray()) {
            int index = letter - 'a';
```

```
            if (cur.children[index] == null)
                return null;
            cur = cur.children[index];
        }
        return cur;
    }
    public boolean search(String word) {
        Trie node = searchPrefix(word);
        // return node!=null && node.isEnd;
        return node != null && node.wordCount > 0;
    }
    public void delete(String word) {
        Trie cur = this;
        Trie[] stack = new Trie[word.length()];
        int[] indices = new int[word.length()];
        int idx = 0;
        for (char letter : word.toCharArray()) {
            int index = letter - 'a';
            if (cur.children[index] == null)
                return;
            stack[idx] = cur;
            indices[idx] = index;

            cur = cur.children[index];
            cur.prefixCount--;
            idx++;

            if (cur.prefixCount == 0) {
                cur.children = null;
            }
        }
        cur.wordCount--;
        if (cur.wordCount == 0) {
            cur.children = null;
        }
```

```
        for (int jdx = word.length() - 1; jdx >= 0; jdx--) {
            Trie parent = stack[jdx];
            int index = indices[jdx];
            if (parent.children[index].prefixCount == 0 && parent.children[index].wordCount == 0) {
                parent.children[index] = null;
            } else {
                break;
            }
        }
    }
    public void print(Trie node, String prefix) {
        for (int idx = 0; idx < 26; idx++) {
            if (node.children != null && node.children[idx] != null) {
                char letter = (char) ('a' + idx);
                int count = node.children[idx].wordCount;
                if (count > 0) {
                    // Print the prefix and word count
                    System.out.println(prefix + letter + ": " + count);
                }
                // Recursively print the children
                print(node.children[idx], prefix + letter);
            }
        }
    }
}
```

Suffix Trie

```
import java.util.Arrays;
import java.util.Scanner;
public class Trie {
    private Trie[] children;
    // private boolean isEnd;
    private int wordCount, prefixCount;
```

```java
public Trie() {
    children = new Trie[26];
    Arrays.fill(children, null);
    // isEnd=false;
    prefixCount = 0;
    wordCount = 0;
}
public void insert(String word) {
    Trie cur = this;
    for (char letter : word.toCharArray()) {
        int index = letter - 'a';
        if (cur.children[index] == null)
            cur.children[index] = new Trie();
        cur = cur.children[index];
        cur.prefixCount++;
    }
    // cur.isEnd=true;
    cur.wordCount++;
}
public void printSuffixes() {
    StringBuilder str = new StringBuilder();  // Use StringBuilder instead of char[]
    printUtil(this, str);
}
private boolean isLeafNode(Trie node) {
    if (node == null)
        return false;
    for (Trie child : node.children) {
        if (child != null)
            return false;
    }
    return true;
}
private void printUtil(Trie root, StringBuilder str) {
    if (root.wordCount > 0) {
        System.out.println(str.toString());
```

```
        }
        for (int i = 0; i < 26; i++) {
            if (root.children[i] != null) {
                str.append((char)('a' + i));
                printUtil(root.children[i], str);
                str.setLength(str.length() - 1);  // Backtrack
            }
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String input = sc.next();
        Trie suffixTrie = new Trie();
        for (int idx = 0; idx < input.length(); idx++) {
            suffixTrie.insert(input.substring(idx));
        }
        suffixTrie.printSuffixes();
        sc.close();
    }
}
```

IndexPairs

```
import java.util.*;

class Node {
    Node[] children;
    boolean isEnd;
    Node() {
        children = new Node[26];
        isEnd = false;
    }
}
```

```java
public class CP_U3_SP4_IndexPairs {
    static int len;
    static String[] words;
    static String text;
    static Node root;
    static void insert(String word) {
        Node cur = root;
        for (char letter : word.toCharArray()) {
            int index = letter - 'a';
            if (cur.children[index] == null)
                cur.children[index] = new Node();
            cur = cur.children[index];
        }
        cur.isEnd = true;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        text = sc.nextLine();
        words = sc.nextLine().split("\\s+");
        len = words.length;
        root = new Node();
        for (String word : words) {
            insert(word);
        }
        Node cur = root;
        List<int[]> indices = new ArrayList<>();
        int index = 0;
        for (int start = 0; start < text.length(); start++) {
            cur = root;
            int end = start;
            while (end < text.length()) {
                index = text.charAt(end) - 'a';
                if (cur.children[index] == null)
                    break;
                cur = cur.children[index];
```

```
                if (cur.isEnd)
                    indices.add(new int[] { start, end });
                end++;
            }
        }
        for (int[] ind : indices) {
            System.out.println(Arrays.toString(ind));
        }
        sc.close();
    }
}
```

Longest word with all prefixes

```
import java.util.*;
class Node {
    public Node[] children;
    public boolean end;
    Node() {
        children = new Node[26];
        end = false;
    }
}
public class CP_U3_SP5_Longest_Word {
    static Node root;
    static String result = "";
    public static void insert(String word) {
        Node cur = root;
        for (char letter : word.toCharArray()) {
            int index = letter - 'a';
            if (cur.children[index] == null)
                cur.children[index] = new Node();
            cur = cur.children[index];
        }
```

```
        cur.end = true;
    }
    static void dfs(Node node, StringBuilder currentWord) {
        // if current node is not end of a word, stop exploring
        if (!node.end && currentWord.length() > 0)
            return;
        // update the result if the current word is valid
        String candidate = currentWord.toString();
        if (candidate.length() > result.length()
                || (candidate.length() == result.length() && candidate.compareTo(result) < 0)) {
            result = candidate;
        }
        //recursively explore all children
        for(int i=0;i<26;i++){
            if(node.children[i]!=null){
                currentWord.append((char)('a'+i));//add the character to current word
                dfs(node.children[i],currentWord);//recur for child node
                currentWord.deleteCharAt(currentWord.length()-1);
                //backtract for the loop to continue correctly
            }
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String[] words = sc.nextLine().split("\\s+");
        root = new Node();
        for (String word : words)
            insert(word);
        dfs(root, new StringBuilder());
        System.out.println(result);
        sc.close();
    }
}
```

TopKFrequentWords

```java
import java.util.PriorityQueue;
import java.util.Scanner;
public class TopKFrequentWords {
    static class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean endOfWord;
        String word = "";
        int freq;
    }
    private final TrieNode root = new TrieNode();
    private void insert(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {
            int idx = ch - 'a';
            if (node.children[idx] == null) node.children[idx] = new TrieNode();
            node = node.children[idx];
        }
        node.endOfWord = true;
        node.freq++;
        node.word = word;
    }
    private void dfs(TrieNode node, PriorityQueue<TrieNode> pq) {
        if (node == null) return;
        if (node.endOfWord) pq.offer(node);
        for (TrieNode child : node.children) dfs(child, pq);
    }
    private void printTopKWords(String[] words, int k) {
        for (String word : words) insert(word);
        PriorityQueue<TrieNode> pq = new PriorityQueue<>((a, b) ->
            a.freq == b.freq ? a.word.compareTo(b.word) : b.freq - a.freq);
        dfs(root, pq);
        for (int i = 0; i < k; i++) System.out.print(pq.poll().word + " ");
```

```
    }
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(System.in)) {
            String[] words = sc.nextLine().split(",");
            int k = sc.nextInt();
            new TopKFrequentWords().printTopKWords(words,k);
        }
    }
}
```

connected components

```
import java.util.*;
public class U1_SP1_Connected_Components_DFS {
    static int[][] grid; // Adjacency matrix to represent the graph
    static boolean[] visited; // Array to track visited nodes
    // DFS method to traverse the graph
    static void dfs(int node, int n) {
        visited[node] = true;
        for (int neighbor = 0; neighbor < n; neighbor++) {
            if (grid[node][neighbor] == 1 && !visited[neighbor]) {
                dfs(neighbor, n);
            }
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // Read number of cities (N) and routes (M)
        int n = sc.nextInt();
        int m = sc.nextInt();
        // Initialize adjacency matrix and visited array
        grid = new int[n][n];
        visited = new boolean[n];
        // Read the edges and populate the adjacency matrix
```

```
        for (int i = 0; i < m; i++) {
            int v1 = sc.nextInt();
            int v2 = sc.nextInt();
            grid[v1][v2] = 1;
            grid[v2][v1] = 1;
        }
        // Count the number of connected components
        int regions = 0;
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                dfs(i, n);
                regions++;
            }
        }
        // Print the number of regions
        System.out.println(regions);
        sc.close();
    }
}
```

bridges

```
import java.util.*;
class FindingBridges {
    private int V; // No. of vertices
    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];
    int time = 0;
    // Constructor
    FindingBridges(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList<Integer>();
```

```
}
// Function to add an edge into the graph
void addEdge(int v, int w) {
    adj[v].add(w); // Add w to v's list.
    adj[w].add(v); // Add v to w's list
}
// DFS based function to find all bridges
void bridge() {
    boolean visited[] = new boolean[V];
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];
    // Initialize parent and visited arrays
    Arrays.fill(parent, -1);
    Arrays.fill(visited, false);
    // Call the recursive helper function to find bridges
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            bridgeUtil(i, visited, disc, low, parent);
        }
    }
}
// A recursive function that finds and prints bridges using DFS traversal
void bridgeUtil(int u, boolean visited[], int disc[], int low[], int parent[]) {
    // Mark the current node as visited
    visited[u] = true;
    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
    // Iterate through all vertices adjacent to this vertex
    for (int v : adj[u]) {
        // If v is not visited, make it a child of u in DFS tree and recur
        if (!visited[v]) {
            parent[v] = u;
            bridgeUtil(v, visited, disc, low, parent);
            // Check if the subtree rooted at v has a connection back to one of the
```

```
                // ancestors of u
                low[u] = Math.min(low[u], low[v]);
                // If the lowest vertex reachable from subtree under v is below u in DFS tree,
                // then u-v is a bridge
                if (low[v] > disc[u]) {
                    System.out.println(u + " " + v);
                }
            }
            // Update low value of u for parent function calls
            else if (v != parent[u]) {
                low[u] = Math.min(low[u], disc[v]);
            }
        }
    }
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        try {
            int v = sc.nextInt();
            int e = sc.nextInt();
            FindingBridges g = new FindingBridges(v);
            for (int i = 0; i < e; i++) {
                int end1 = sc.nextInt();
                int end2 = sc.nextInt();
                g.addEdge(end1, end2);
            }
            g.bridge();
        } finally {
            sc.close(); // Close the scanner to prevent resource leaks
        }
    }
}
```

articulation points

```java
class FindingArticulationPoints {
    private int V;
    private LinkedList<Integer> adj[];
    int time = 0;
    // Constructor remains same as before
    // ...existing code...
    void AP() {
        boolean visited[] = new boolean[V];
        int disc[] = new int[V];
        int low[] = new int[V];
        int parent[] = new int[V];
        boolean ap[] = new boolean[V]; // Store articulation points
        Arrays.fill(parent, -1);
        Arrays.fill(visited, false);
        Arrays.fill(ap, false);
        // Call the recursive helper function for each undiscovered vertex
        for (int i = 0; i < V; i++)
            if (!visited[i])
                APUtil(i, visited, disc, low, parent, ap);
        // Print all articulation points
        for (int i = 0; i < V; i++)
            if (ap[i])
                System.out.print(i + " ");
    }
    void APUtil(int u, boolean visited[], int disc[], int low[],
                int parent[], boolean ap[]) {
        int children = 0;
        visited[u] = true;
        disc[u] = low[u] = ++time;
        for (Integer v : adj[u]) {
            if (!visited[v]) {
                children++;
                parent[v] = u;
                APUtil(v, visited, disc, low, parent, ap);
                low[u] = Math.min(low[u], low[v]);
```

```
                // Case 1: u is root and has two or more children
                if (parent[u] == -1 && children > 1)
                    ap[u] = true;
                // Case 2: u is not root and low value of one of its children
                // is more than discovery value of u
                if (parent[u] != -1 && low[v] >= disc[u])
                    ap[u] = true;
            }
            else if (v != parent[u])
                low[u] = Math.min(low[u], disc[v]);
        }
    }
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        try {
            int v = sc.nextInt();
            int e = sc.nextInt();
            FindingArticulationPoints g = new FindingArticulationPoints(v);
            for (int i = 0; i < e; i++) {
                int end1 = sc.nextInt();
                int end2 = sc.nextInt();
                g.addEdge(end1, end2);
            }
            g.AP();
        } finally {
            sc.close();
        }
    }
}
```

ford fulkerson

```
import java.util.Scanner;
public class FordFulkerson{
```

```java
static int visitedToken=1;
public static int fordFulkerson(int[][]caps,int source, int sink){
    int n=caps.length;
    int[]visited=new int[n];
    // boolean[] minCut=new boolean[n];
    for(int maxFlow=0;;){
        int flow=dfs(caps, visited, source, sink, Integer.MAX_VALUE);
        visitedToken++;
        maxFlow+=flow;
        if(flow==0){
            return maxFlow;
        }
    }
}
private static int dfs(int[][]caps, int[]visited, int node ,int sink, int flow){
    if(node==sink) return flow;
    int[]cap=caps[node];
    visited[node]=visitedToken;
    for(int i=0;i<cap.length;i++){
        if(visited[i]!=visitedToken && cap[i]>0){
            if(cap[i]<flow) flow=cap[i];
            int bottleneck=dfs(caps, visited, i, sink , flow);
            if(bottleneck>0){
                caps[node][i]-=bottleneck;
                caps[i][node]+=bottleneck;
                return bottleneck;
            }
        }
    }
    return 0;
}
public static void main(String[] args) {
    try(Scanner sc=new Scanner(System.in)){
        System.out.println("Enter number of nodes");
        int vertices=sc.nextInt();
```

```
            int[][] caps=new int[vertices][vertices];
            System.out.println("Enter capacity matrix");
            for(int i=0;i<vertices;i++){
                for(int j=0;j<vertices;j++){
                    caps[i][j]=sc.nextInt();
                }
            }
            System.out.println("Enter source");
            int source=sc.nextInt();
            System.out.println("Enter sink");
            int sink=sc.nextInt();
            System.out.println("Calculating max flow");
            int maxFlow=fordFulkerson(caps, source, sink);
            System.out.println("MaxFlow = "+maxFlow);
        }
    }
}
```

lca

```
import java.util.*;
class Node {
    public int data;
    public Node left;
    public Node right;
    public Node(int value) {
        data = value;
        left = null;
        right = null;
    }
}
public class LCA {
    // Helper method to build the tree from the input list
    private static Node buildTree(List<Integer> v) {
```

```
        if (v.isEmpty() || v.get(0) == -1)
            return null;
        Node root = new Node(v.get(0));
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);
        int i = 1;
        while (!queue.isEmpty() && i < v.size()) {
            Node current = queue.poll();
            if (i < v.size() && v.get(i) != -1) {
                current.left = new Node(v.get(i));
                queue.add(current.left);
            }
            i++;
            if (i < v.size() && v.get(i) != -1) {
                current.right = new Node(v.get(i));
                queue.add(current.right);
            }
            i++;
        }
        return root;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String[] arr = sc.nextLine().split(" ");
        String[] persons = sc.nextLine().split(" ");
        List<Integer> v = new ArrayList<>();
        int n = arr.length;
        for (int i = 0; i < n; i++) {
            v.add(Integer.parseInt(arr[i]));
        }
        Node P1 = new Node(Integer.parseInt(persons[0]));
        Node P2 = new Node(Integer.parseInt(persons[1]));
        // Write necessary code
        Node root = buildTree(v);
        Node res = new Solution().lowestCommonAscendant(root, P1, P2);
```

```
        System.out.println(res.data);
    }
}
class Solution {
    private Node dfs(Node root, Node p, Node q) {
        if (root == null || root.data == p.data || root.data == q.data) {
            return root;
        }
        Node left = dfs(root.left, p, q);
        Node right = dfs(root.right, p, q);
        if (left != null && right != null) {
            return root;
        }
        return (left != null) ? left : right;
    }
    public Node lowestCommonAscendant(Node root, Node p, Node q) {
        return dfs(root, p, q);
    }
}
```

topologicalSort

```
import java.util.*;
public class Topo {
    private static void dfs(int v, List<Integer>[] adj, boolean[] visited, Stack<Integer> stack) {
        visited[v] = true;
        for (int i : adj[v]) {
            if (!visited[i]) {
                dfs(i, adj, visited, stack);
            }
        }
        stack.push(v);
    }
    static List<Integer>[] constructAdj(int v, int[][] edges) {
```

```
    @SuppressWarnings("unchecked")
    List<Integer>[] adj = new ArrayList[v];
    for (int i = 0; i < v; i++)
        adj[i] = new ArrayList<>();
    for (int[] edge : edges)
        adj[edge[0]].add(edge[1]);
    return adj;
}
static int[] topologicalSort(int v, int[][] edges) {
    Stack<Integer> stack = new Stack<>();
    boolean[] visited = new boolean[v];
    List<Integer>[] adj = constructAdj(v, edges);
    for (int i = 0; i < v; i++) {
        if (!visited[i]) {
            dfs(i, adj, visited, stack);
        }
    }
    int[] result = new int[v];
    int index = 0;
    while (!stack.isEmpty()) {
        result[index++] = stack.pop();
    }
    return result;
}
public static void main(String[] args) {
    try (Scanner sc = new Scanner(System.in)) {
        int v = sc.nextInt(), e = sc.nextInt();
        int[][] edges = new int[v][2];
        for (int i = 0; i < e; i++) {
            edges[i][0] = sc.nextInt();
            edges[i][1] = sc.nextInt();
        }
        int[] result = topologicalSort(v, edges);
        for (int node : result) {
            System.out.print(node + " ");
```

```
            }
            System.out.println();
        }
    }
}
```

parallel course 2

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.Scanner;
class Graph{
    private final int v;
    private final List<List<Integer>> adj;
    private final int[] indegree;
    Graph(int v, int[][]edges){
        this.v=v;
        this.adj=new ArrayList<>();
        this.indegree=new int[v+1];
        build(edges);
    }
    private void build(int[][]edges){
        for(int i=0;i<=v;i++){
            adj.add(new ArrayList<>());
        }
        for(int[]edge:edges){
            adj.get(edge[0]).add(edge[1]);
            indegree[edge[1]]++;
        }
    }
    public int minSemesters(int k){
        Queue<Integer> queue=new LinkedList<>();
```

```
        for(int i=1;i<=v;i++) if(indegree[i]==0) queue.offer(i);
        int semesters=0, taken=0;
        while(!queue.isEmpty()){
            int canTake=Math.min(queue.size(),k);
            List<Integer> nextBatch=new ArrayList<>();
            for(int i=0;i<canTake;i++){
                int course=queue.poll();
                taken++;
                for(int neighbor:adj.get(course)){
                    if(--indegree[neighbor]==0) nextBatch.add(neighbor);
                }
            }
            queue.addAll(nextBatch);
            semesters++;
        }
        return (taken==v)?semesters:-1;
    }
}
public class ParallelCourses2 {
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(System.in)) {
            System.out.print("Enter number of courses: ");
            int n = sc.nextInt();
            System.out.print("Enter number of relations (edges): ");
            int e = sc.nextInt();
            int[][] edges = new int[e][2];
            System.out.println("Enter prerequisite relations (u v):");
            for (int i = 0; i < e; i++) {
                edges[i][0] = sc.nextInt();
                edges[i][1] = sc.nextInt();
            }
            System.out.print("Enter max courses per semester (k): ");
            int k = sc.nextInt();
            Graph g = new Graph(n, edges);
            int result = g.minSemesters(k);
```

```
            System.out.println("Minimum semesters required: " + result);
        }
    }
}
```

Lexicographically Smallest Equivalent String

```
import java.util.*;
class DisjointUnionSet {
    int[] parent;
    public DisjointUnionSet(int arraySize) {
        parent = new int[arraySize];
        for (int idx = 0; idx < arraySize; idx++) {
            parent[idx] = idx;
        }
    }
    // Returns representative of x's set
    int findParent(int num) {
        if (this.parent[num] != num)
            parent[num] = findParent(this.parent[num]);
        return parent[num];
    }
    // Unites the set that includes x and the set that includes x
    void union(int node1, int node2) {
        int parent1 = findParent(node1), parent2 = findParent(node2);
        if (parent1 != parent2)
            if (parent1 > parent2) {
                parent[parent1] = parent2;
            } else {
                parent[parent2] = parent1;
            }
    }
}
public class U5_SP1_Smallest_Equivalent_String {
```

```
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(System.in)) {
            StringBuilder result = new StringBuilder();
            char[] str1 = sc.next().toCharArray(), str2 = sc.next().toCharArray(), str = sc.next().toCharArray();
            DisjointUnionSet dus = new DisjointUnionSet(26);
            for (int idx = 0; idx < str1.length; idx++) {
                dus.union(str1[idx] - 'a', str2[idx] - 'a');
            }
            for (char character : str) {
                result.append((char) ('a' + dus.parent[character - 'a']));
            }
            System.out.println(result);
            sc.close();
        }
    }
}
```

Distinct Islands

```
import java.util.HashSet;
import java.util.Set;
class Solution {
    private int rows; // number of rows in the grid
    private int cols; // number of columns in the grid
    private int[][] grid; // grid representation
    private StringBuilder path; // used to store the path during DFS to identify unique islands
    public int numDistinctIslands(int[][] grid) {
        rows = grid.length; // set the number of rows
        cols = grid[0].length; // set the number of columns
        this.grid = grid; // reference the grid
        Set<String> uniqueIslands = new HashSet<>(); // store unique island paths as strings
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (grid[i][j] == 1) { // if it's part of an island
```

```
                    path = new StringBuilder(); // initialize the path
                    exploreIsland(i, j, 'S'); // start DFS with dummy direction 'S' (Start)
                    uniqueIslands.add(path.toString()); // add the path to the set
                }
            }
        }
        return uniqueIslands.size(); // the number of unique islands
    }
    private void exploreIsland(int i, int j, char direction) {
        grid[i][j] = 0; // mark as visited
        path.append(direction); // append the direction to path
        // directions represented as delta x and delta y
        int[] dX = { -1, 0, 1, 0 };
        int[] dY = { 0, 1, 0, -1 };
        char[] dirCodes = { 'U', 'R', 'D', 'L' }; // corresponding directional codes
        for (int dir = 0; dir < 4; ++dir) { // iterate over possible directions
            int x = i + dX[dir];
            int y = j + dY[dir];
            if (x >= 0 && x < rows && y >= 0 && y < cols && grid[x][y] == 1) { // check for valid next cell
                exploreIsland(x, y, dirCodes[dir]); // recursive DFS call
            }
        }
        path.append('B'); // append backtrack code to ensure paths are unique after recursion return
    }
}
```

count components

```
import java.util.*;
public class Connected_Components_DSU{
    static int[] parent;
    static int findParent(int node){
        if(parent[node]!=node) parent[node]=findParent(node);
        return parent[node];
```

```
    }
    static void union(int v1,int v2){
        int p1=findParent(v1),p2=findParent(v2);
        if(p1!=p2){
            parent[p1]=p2;
        }
    }
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        int nv=sc.nextInt();
        int ne=sc.nextInt();
        int result=nv;
        parent=new int[nv];
        for(int idx=0;idx<nv;idx++)parent[idx]=idx;
        for(int idx=0;idx<ne;idx++){
            int v1=sc.nextInt(),v2=sc.nextInt();
            int p1=findParent(v1), p2=findParent(v2);
            if(p1!=p2){
                result--;
                union(v1,v2);
            }
        }
        System.out.println(result);
        sc.close();
    }
}
```

Palindrome_Permutation

```
import java.util.*;
public class Palindrome_Permutation {
    public static void main(String[]args){
        Scanner sc=new Scanner(System.in);
        char[] letters=sc.nextLine().toCharArray();
```

```
        int res=0;
        boolean isPalin=false;
        for(char letter : letters){
            res ^= (1 << (letter - 'a'));
        }
        if(res==0 || (res & (res - 1))==0) isPalin=true;
        System.out.println(isPalin);
        sc.close();
    }
}
```

LongestIncreasingPath

```
import java.util.Scanner;
public class LongestIncreasingPath {
    static int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // up, down, left, right
    static int rows, cols;
    static int[][] matrix, memo;
    public static int longestIncreasingPath(int[][] matrix) {
        if (matrix == null || matrix.length == 0) return 0;
        rows = matrix.length;
        cols = matrix[0].length;
        memo = new int[rows][cols];
        int maxLength = 0;
        // Try starting from each cell
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                maxLength = Math.max(maxLength, dfs(matrix, i, j));
            }
        }
        return maxLength;
    }
    private static int dfs(int[][] matrix, int row, int col) {
        // If already computed, return memoized result
```

```
        if (memo[row][col] > 0) return memo[row][col];
        int max = 1; // Minimum path length is 1 (current cell)
        // Try all four directions
        for (int[] dir : directions) {
            int newRow = row + dir[0];
            int newCol = col + dir[1];
            // Check bounds and increasing condition
            if (newRow >= 0 && newRow < rows &&
                newCol >= 0 && newCol < cols &&
                matrix[newRow][newCol] > matrix[row][col]) {
                max = Math.max(max, 1 + dfs(matrix, newRow, newCol));
            }
        }
        // Memoize and return
        memo[row][col] = max;
        return max;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // Read matrix dimensions
        rows = sc.nextInt();
        cols = sc.nextInt();
        // Read matrix
        matrix = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = sc.nextInt();
            }
        }
        // Find and print result
        System.out.println(longestIncreasingPath(matrix));
        sc.close();
    }
}
```