# Problem Statement

In this problem, you will be given a directed unweighted graph **G** with **N** nodes (1 <= N <= 1000) and **E** edges (1 <= E <= 8000).

## Input Format

- The first line of input will contain two integers **N** and **E**.
- The next **E** lines will each contain two integers **A** and **B**, representing a directed edge from node **A** to node **B**.
- After that, there will be a single integer **X** (1 <=X <= N), denoting a node from the given graph.

## Output Format

- In the first line, print a single integer denoting the number of adjacent nodes of the given node **X**.
- In the second line, print **N** values, separated by a single space, where the **i-th** value denotes the minimum number of edges required to reach the **i-th** node. If a node is not reachable from **X**, the corresponding value should be -1. There should not be any leading or trailing spaces.

## Constraints

- (1 <= N <= 1000)
- (1 <= E <= 5000)
- (1 <=X <= N)

```
    // Read starting node X
    X = READ_INTEGER()

    // Part 1: Count adjacent nodes
    PRINT adj[X].SIZE()

    // Part 2: BFS to find shortest distances
    distances = ARRAY OF SIZE N+1 INITIALIZED TO -1
    distances[X] = 0

    queue = EMPTY QUEUE
    queue.PUSH(X)

    WHILE queue IS NOT EMPTY:
        current = queue.POP()

        FOR neighbor IN adj[current]:
            IF distances[neighbor] == -1:
                distances[neighbor] = distances[current] + 1
                queue.PUSH(neighbor)

    // Print distances (skip index 0)
    FOR i FROM 1 TO N:
        PRINT distances[i]
        IF i < N:
            PRINT " "

    PRINT NEWLINE
END FUNCTION
```

**Experiment:** Create a Graph Class with Depth-First Search (DFS) and Topological Sort in Java.

a. Create a Graph class where the constructor takes a filename as input and the graph is initialized using input from the file and stored in an adjacency list or matrix. Here's an example input for 5 vertices, 3 edges:

5 3
1 2
2 5
3 4

b. There should be functions to
   1. add vertex(n): Add n new vertices to the graph
   2. add edge(u,v): Add an edge to the graph
   3. get the number of vertices
   4. get adjacent vertices of a given vertex
   5. display the graph's adjacency list

Then you should implement the following features:

**1. Depth-First Search (DFS):** Perform DFS traversal on the graph and print the order of vertices visited.
**2. Topological Sort:** Implement a topological sorting algorithm using DFS, and return the vertices in topologically sorted order.

The graph should be initialized using input from a file. For example, given the following file: **[ Notice that the vertices can start from 0, so you have to keep this configurable ]**

6 6
5 2
5 0
4 0
4 1
2 3
3 1

This input represents a directed graph with 6 vertices and 6 edges.

**New Methods to Implement:**
- **DFS(v):** Perform DFS starting from vertex `v` and print the order of traversal.
- **topologicalSort():** Perform Topological Sorting and return the list of vertices in topologically sorted order.

**DFS(v):**
- Implement DFS traversal starting from a given vertex.
- Print the order of vertices visited during the DFS traversal.

**topologicalSort():**
- Implement Topological Sort using DFS for Directed Acyclic Graphs (DAG) only.
- The method should return a list of vertices in topologically sorted order.

## 📦 Graph Initialization from File

**Pseudocode:**

```mathematica
Graph(file):
    Read first line: V = number of vertices, E = number of edges
    Initialize adjacencyList as list of empty lists with size V
    For each of next E lines:
        Read u and v
        Add v to adjacencyList[u]
```

**Explanation:**
We read the number of vertices and edges, then for each edge, update the adjacency list. The graph is directed, so we only add `v` to `u`'s list.

## ➕ Add Vertex

**Pseudocode:**

```scss
addVertex(n):
    For i in 1 to n:
        Add an empty list to adjacencyList
```

**Explanation:**
Each new vertex gets its own empty list of neighbors in the adjacency list.

---

## ➕ Add Edge

**Pseudocode:**

```scss
addEdge(u, v):
    adjacencyList[u].add(v)
```

**Explanation:**

## 🔍 Get Adjacent Vertices

**Pseudocode:**

```kotlin
getAdjVertices(u):
    return adjacencyList[u]
```

**Explanation:**
Returns all vertices directly reachable from vertex `u`.

---

## 📊 Display Graph

**Pseudocode:**

```scss
displayGraph():
    For each vertex u in adjacencyList:
        Print u -> elements in adjacencyList[u]
```

**Explanation:**
Prints each vertex and its outgoing edges.

## 🔁 DFS Traversal

**Pseudocode:**

sql

```
DFS(v):
    visited = boolean array of size V, all false
    result = empty list

    dfsHelper(v, visited, result)

    Print result

dfsHelper(v, visited, result):
    visited[v] = true
    result.add(v)

    For each neighbor in adjacencyList[v]:
        If not visited[neighbor]:
            dfsHelper(neighbor, visited, result)
```

## □ Topological Sort (DFS Based)

**Pseudocode:**

arduino

```
topologicalSort():
    visited = boolean array of size V, all false
    stack = empty stack

    For each vertex v in 0 to V-1:
        If not visited[v]:
            topoSortDFS(v, visited, stack)

    result = reverse of stack
    return result

topoSortDFS(v, visited, stack):
    visited[v] = true

    For each neighbor in adjacencyList[v]:
        If not visited[neighbor]:
            topoSortDFS(neighbor, visited, stack)

    stack.push(v)
```

## 🧠 Bonus 1: Path Between Two Vertices (DFS based)

**Pseudocode:**

pgsql

Copy Edit

```pgsql
findPath(src, dest):
    visited = boolean array
    path = empty list
    found = dfsPath(src, dest, visited, path)
    If found: return path
    else: return "No path"

dfsPath(curr, dest, visited, path):
    visited[curr] = true
    path.add(curr)
    If curr == dest: return true

    For neighbor in adjacencyList[curr]:
        If not visited[neighbor]:
            If dfsPath(neighbor, dest, visited, path):
                return true

    path.remove last element
    return false
```

## 🧠 Bonus 2: Lexicographic DFS

**Pseudocode:**

sql

Copy Edit

```sql
lexDFS(v):
    visited = boolean array
    result = empty list
    dfsLexHelper(v, visited, result)
    print result

dfsLexHelper(v, visited, result):
    visited[v] = true
    result.add(v)

    neighbors = sorted(adjacencyList[v])
    For neighbor in neighbors:
        If not visited[neighbor]:
            dfsLexHelper(neighbor, visited, result)
```

**Explanation:**

Same as DFS, but sort neighbors before visiting for lexicographic order.

## ✅ Problem Summary

You are given a map of Sylhet as a graph:

- Each city is a node.
- Each road is an undirected edge.
- Anik can move at most K units (one unit of fuel per city hop).
- Starting from city L, count **how many different cities he can reach with K units of fuel.**

## 🧠 Pseudocode: BFS with fuel constraint

csharp

```csharp
function maxCitiesReachable(C, R, K, L, roadList):

    Initialize graph as an adjacency list of size (C+1)
    For each (u, v) in roadList:
        Add v to graph[u]
        Add u to graph[v]   // because roads are bidirectional

    visited = boolean array of size (C+1), all false
    queue = empty queue of (city, fuelUsed)

    Enqueue (L, 0) to queue  // start from city L with 0 fuel used
    visited[L] = true
    reach = 1   // starting city is counted

    while queue is not empty:
        (city, fuelUsed) = dequeue from queue

        if fuelUsed == K:
            continue  // can't go further, skip

        for each neighbor of city in graph:
            if neighbor is not visited:
                visited[neighbor] = true
                reach += 1
                Enqueue (neighbor, fuelUsed + 1)

    Print reach
```

**There are n cities and there are roads in between some of the cities. Somehow all the roads are damaged simultaneously. We have to repair the roads to connect the cities again. There is a fixed cost to repair a particular road.**
**Input is in the form of edges `{u, v, w}` where, `u` and `v` are city indices. `w` is the cost to rebuild the road between `u` and `v`. Print out the minimum cost to connect all the cities by repairing roads.**

### ✏️ Easy Pseudocode: (Using Kruskal's Algorithm)

java    Copy    Edit

```java
function minimumRepairCost(n, edges):

    Sort edges by cost (w) in ascending order
    Create parent array of size n to track connected components (DSU)

    Initialize totalCost = 0
    Initialize edgeCount = 0

    for each (u, v, w) in sorted edges:
        if findParent(u) != findParent(v):
            union(u, v)
            totalCost += w
            edgeCount += 1

            if edgeCount == n - 1:
                break  // All cities are connected

    if edgeCount != n - 1:
        return "Impossible to connect all cities"

    return totalCost

function findParent(node):
    if parent[node] == node:
        return node
    parent[node] = findParent(parent[node])  // Path compression
    return parent[node]

function union(a, b):
    parent[findParent(a)] = findParent(b)
```

### 🧠 Quick Comparison: Kruskal vs. Prim

| Feature | Kruskal's Algorithm | Prim's Algorithm |
|---|---|---|
| Best for | **Sparse graphs** (few edges) | **Dense graphs** (many edges) |
| Input type | List of edges | Adjacency list or matrix |
| Edge selection | Sort all edges and pick the smallest | Pick the smallest edge from visited nodes |
| Cycle detection | Uses **Disjoint Set Union (DSU)** | Not needed; just use a **visited[]** array |
| Time Complexity | `O(E log E)` (due to sorting edges) | `O(E log V)` using Min Heap (PriorityQueue) |
| Implementation Style | Greedy edge-by-edge with DSU | Grow tree from one node using heap |
| Easy to Implement | If edges are directly given | If graph is in adjacency list format |
| Works with | Works for disconnected graphs (forest | Needs a connected graph to work properly |

## 📘 Problem:

You are given a **weighted undirected graph** as an **adjacency list or matrix.**

Your goal is to:

- Connect all the cities (nodes) with **minimum total cable cost.**
- Start from a **source city.**
- Use **Prim's Algorithm** to expand the Minimum Spanning Tree (MST).

## ✅ Prim's Algorithm: Pseudocode

sql

```
function primMST(graph, n, source):

    visited = boolean array of size n, all false
    minHeap = priority queue (stores (cost, to_node))
    totalCost = 0

    push (0, source) into minHeap   // start from source with 0 cost

    while minHeap is not empty:
        (cost, current) = minHeap.pop()

        if visited[current]:
            continue  // already added to MST

        visited[current] = true
        totalCost += cost

        for each neighbor in graph[current]:
            (nextNode, weight) = neighbor
            if not visited[nextNode]:
                push (weight, nextNode) into minHeap

    return totalCost
```

A city named *Flower* on the western front, has been very quiet since some time after the deadly aftermath of war. But flowers are booming and things have started to move.

The new mayor, *Tris*, has decided to repair the country's damaged roads, which are currently in very bad condition. Tris has old maps, which have information about the previous road network. Each road of the Flower is bidirectional by default, and in the map, Tris saw an additional piece of information for each road regarding the number of vehicles that can move simultaneously through that road.

The budget is quite limited and Tris wants to maximize the total number of vehicles passing simultaneously all throughout the country. If there are **N** cities she wants to build exactly **(N-1)** roads so that all the country stays connected maximizing the total number of vehicles passing simultaneously. As there are a lot of cities and roads and Tris has been very busy managing the country, she has been looking for a programmer to help her solve this problem.

*Can you come forward to help her by designing a solution?*

Flower can be modeled through a graph network. Flower has $N$ cities (nodes) and $E$ bidirectional roads(edges). In the first two lines, you will be given two integer variables $N$ (1<=N<=5000) and $E$ (1<=E<=100000). In the following $E$ lines, you will be given the information regarding the roads. Each road's information holds three values $A$, $B$ and $C$ denoting an undirected edge between cities(nodes) $A$ and $B$ with the information $C$ representing the total number of cars that can simultaneously pass through the road(edge) $A->B$.

java Copy Edit

```java
function maxVehicleCapacity(N, E, edgeList):
    Sort edgeList in **descending order** by capacity (C)

    Initialize DSU (Disjoint Set Union) with N nodes
    totalCapacity = 0
    edgeCount = 0

    for each edge (A, B, C) in sorted edgeList:
        if findParent(A) != findParent(B):
            union(A, B)
            totalCapacity += C
            edgeCount += 1

            if edgeCount == N - 1:
                break  // All cities connected

    return totalCapacity


function findParent(node):
    if parent[node] == node:
        return node
    parent[node] = findParent(parent[node])  // path compression
    return parent[node]

function union(a, b):
    parent[findParent(a)] = findParent(b)
```

My solve:

```
class Edge:
    a, b, c  // cities a and b, c = number of vehicles (capacity)
```

Copy Edit

```
class UnionFind:
    parent[] = array of size N+1 where parent[i] = i

    function find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])  // path compression
        return parent[x]

    function union(a, b):
        rootA = find(a)
        rootB = find(b)
        if rootA == rootB:
            return false  // already connected
        parent[rootB] = rootA
        return true  // merged new components

main():
    Read n = number of cities
    Read m = number of roads
    edgeList = empty list

    for i = 1 to m:
        Read a, b, c
        Add Edge(a, b, c) to edgeList

    Sort edgeList by c (capacity) in descending order

    uf = new UnionFind(n + 1)
    totalCapacity = 0
    edgeCount = 0

    for each edge in edgeList:
        if uf.union(edge.a, edge.b):
            totalCapacity += edge.c
            edgeCount += 1
            if edgeCount == n - 1:
                break  // MST complete

    Print totalCapacity
```

A robot in a warehouse must deliver items from one storage unit to another. The warehouse floor is represented as a 2D grid of size n × m. Each cell in the grid has a time cost associated with entering that cell.

The robot can move in four directions — up, down, left, and right (no diagonal movement). Your task is to compute the minimum total time required for the robot to reach the destination cell from the starting cell. Also, find the shortest path from the source to the destination.

For the input,
First line: n rows, m columns
Next n lines: Each line has m integers (time to enter that cell)
Start: (sx, sy)
End: (ex, ey)

You need to print the minimum total time and the shortest path.



```
function minTimeShortestPath(grid, sx, sy, ex, ey):
    n = number of rows
    m = number of columns
    directions = [(0,1), (1,0), (0,-1), (-1,0)]  // right, down, left, up

    Create distance matrix: dist[n][m] = INF
    Create step matrix: steps[n][m] = INF
    Create parent matrix: parent[n][m] = (-1, -1)

    PriorityQueue pq = min-heap sorted by (totalCost, steps)
    Push (grid[sx][sy], 0, sx, sy) to pq
    dist[sx][sy] = grid[sx][sy]
    steps[sx][sy] = 0

    while pq is not empty:
        (cost, step, x, y) = pq.pop()

        for (dx, dy) in directions:
            nx = x + dx
            ny = y + dy

            if nx and ny are within bounds:
                newCost = cost + grid[nx][ny]
                newSteps = step + 1

                if newCost < dist[nx][ny] or
                   (newCost == dist[nx][ny] and newSteps < steps[nx][ny]):
                    dist[nx][ny] = newCost
                    steps[nx][ny] = newSteps
                    parent[nx][ny] = (x, y)
                    pq.push((newCost, newSteps, nx, ny))

    // Reconstruct path from destination
    path = empty list
    curr = (ex, ey)
    while curr != (sx, sy):
        path.add(curr)
        curr = parent[curr.x][curr.y]
    path.add((sx, sy))
    Reverse path

    return dist[ex][ey], path
```

## 🧩 2. Print Shortest Path in Dijkstra's Algorithm

### 🧠 Idea:

While running Dijkstra, we store each node's **parent** during relaxation. Then we **backtrack** from destination to source.

---

### 📝 Pseudocode (With Path Recovery):

pseudocode

```
function dijkstraWithPath(graph, source, destination):
    dist[] = INF for all nodes
    parent[] = -1 for all nodes
    visited[] = false
    dist[source] = 0

    minHeap = priority queue of (cost, node)
    push (0, source) to minHeap

    while minHeap is not empty:
        (cost, u) = pop from minHeap
        if visited[u]: continue
        visited[u] = true

        for each (v, weight) in graph[u]:
            if dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                parent[v] = u
                push (dist[v], v) into minHeap

    // Reconstruct path
    path = []
    curr = destination
    while curr != -1:
        path.add(curr)
        curr = parent[curr]
    reverse(path)

    return dist[destination], path
```

pseudocode

Copy Edit

```
function bellmanFord(V, E, edgeList, source):

    // Step 1: Initialize distances
    dist = array of size V, filled with INF
    parent = array of size V, filled with -1
    dist[source] = 0

    // Step 2: Relax all edges V-1 times
    for i = 1 to V-1:
        for each (u, v, w) in edgeList:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                parent[v] = u

    // Step 3: Check for negative weight cycles
    for each (u, v, w) in edgeList:
        if dist[u] + w < dist[v]:
            return "Negative cycle detected"

    return dist, parent
```

## 🔁 Path Reconstruction (Optional)

To print the path from source to any vertex `v`:

pseudocode

Copy Edit

```
function getPath(v, parent):
    path = []
    while v != -1:
        path.append(v)
        v = parent[v]
    reverse(path)
    return path
```

### Problem Statement 1: Binary Search Tree

Problem Statement: Write a program in Java to implement a Binary Search Tree (BST). Your program should include methods to:

1. insert(num): Add a node to the BST while maintaining the binary search tree property:
    a. All values in the left subtree of a node are less than the node's value.
    b. All values in the right subtree of a node are greater than the node's value.
2. search(key): Check whether a given value exists in the BST. It should return True/False.
3. inorder_traversal(): Print all values in the BST in ascending order using in-order traversal.

The input will have 3 lines

1. an input N - the number of nodes in the BST.
2. N comma-separated integers in a line, each to be inserted into the tree. After all nodes have been inserted, call the inorder_traversal() method.
3. The last line of the input will be the value to search in the three. If the value X is found, it should print "X is found in the BST.". If it is not found, it should print "X is not found in the BST"

```sql
Start:

1. Create empty BST (root = null)

2. Function insert(num):
    If root is empty:
        root = new Node(num)
    Else:
        Call insertHelper(root, num)

3. Function insertHelper(node, num):
    If num < node.value:
        If node.left is empty:
            node.left = new Node(num)
        Else:
            insertHelper(node.left, num)
    Else if num > node.value:
        If node.right is empty:
            node.right = new Node(num)
        Else:
            insertHelper(node.right, num)
```

Copy Edit

```
4. Function search(node, key):
    If node is empty:
        Return False
    If node.value == key:
        Return True
    Else if key < node.value:
        Return search(node.left, key)
    Else:
        Return search(node.right, key)

5. Function inorder(node):
    If node is empty:
        Return
    inorder(node.left)
    Print node.value
    inorder(node.right)

6. Main program:
    Read N
    Read list of N numbers

    For each number in list:
        insert(number)

    inorder(root)  // print sorted values

    Read X  // search value

    If search(root, X) is True:
        Print "X is found in the BST."
    Else:
        Print "X is not found in the BST."

End
```

## Problem Statement 2: Lowest Common Ancestor in Binary Search Tree

In this problem, you have to extend the previous problem's solution. You have to add a new function, **lca(n1, n2)** where Given two values n1 and n2 in a Binary Search Tree, find the Lowest Common Ancestor (LCA). You may assume that both values exist in the tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself). In other words, the LCA of n1 and n2 is the shared ancestor of n1 and n2 that is located farthest from the root [i.e., closest to n1 and n2]. **The time complexity of your implementation should be O(h) where h is the height of the tree.**

sql                                                    Copy    Edit

```sql
Function lca(node, n1, n2):
    If node is null:
        Return null

    If n1 < node.value AND n2 < node.value:
        // Both nodes are in left subtree
        Return lca(node.left, n1, n2)

    Else if n1 > node.value AND n2 > node.value:
        // Both nodes are in right subtree
        Return lca(node.right, n1, n2)

    Else:
        // Nodes are on different sides or one equals current node
        Return node
```

## Problem Statement 3: Best Time to Buy and Sell Stock

Problem Statement: You are given comma separated prices which contain the price of a stock on some days. You have to choose one day for buying the stock and a different one for selling it. Your task is to determine the **maximum profit** you can achieve from a single buy-sell transaction, where the buy day must come before the sell day. If no profit is possible, return 0.

**Example:**

**Input:**

- An integer n ($1 \le n \le 10^5$): the number of days
- A comma-separated list of n integers: the prices of the stock on each day

```mathematica
Function maxProfit(prices):
    minPrice = +∞
    maxProfit = 0

    For each price in prices:
        If price < minPrice:
            minPrice = price
        Else if price - minPrice > maxProfit:
            maxProfit = price - minPrice

    Return maxProfit


Main:
    Read n
    Read prices list of n integers

    result = maxProfit(prices)
    Print result
```

## Problem Statement 4: Minimum Number of Notes

**Problem Statement:** We are given a value of V and we need to give the change of it in Bangladesh notes. Given an integer V ($0 \le V \le 10^5$), find the minimum number of Bangladeshi notes/coins whose total value equals V. You have unlimited supply of each denomination in the set {1, 2, 5, 10, 20, 50, 100, 200, 500, 1000}. Output that minimum count. **If V = 0, output 0.**

```mathematica
Function minNotes(V):
    denominations = [1000, 500, 200, 100, 50, 20, 10, 5, 2, 1]
    count = 0

    For each note in denominations:
        If V == 0:
            Break

        notes_used = V // note        // integer division
        count += notes_used
        V = V % note                  // remaining amount

    Return count

Main:
    Read V
    result = minNotes(V)
    Print result
```

## Problem Statement 5: Fractional Knapsack Problem

Problem Statement: You are given the weights and values of N items. You need to put these items in a knapsack of capacity W in such a way that the final value of the knapsack is maximum. You can break down the items into fractions to attain the most optimal answer.

1. First line of input will be N
2. The following N lines will each have two integers - V and w for each item
3. Knapsack capacity W

```
Class Item:
    value
    weight
    ratio = value / weight

Function fractionalKnapsack(items, W):
    Sort items by ratio descending
    total_value = 0
    remaining_capacity = W

    For each item in items:
        If remaining_capacity == 0:
            Break

        If item.weight <= remaining_capacity:
            total_value += item.value
            remaining_capacity -= item.weight
        Else:
            fraction = remaining_capacity / item.weight
            total_value += item.value * fraction
            remaining_capacity = 0

    Return total_value

Main:
    Read N
    items = empty list

    For i = 1 to N:
        Read V, w
        Create item with value V, weight w
        Add item to items

    Read W (capacity)

    result = fractionalKnapsack(items, W)
    Print result
```

**Problem 1: Fibonacci Numbers**
You will implement the Fibonacci sequence using two approaches: **Memoization** and **Tabulation**.

**Methods to Implement:**

a. **fibMemo(int n, HashMap<Integer, Integer> memo)**: Computes Fibonacci numbers using a top-down memoization approach.
b. **fibTab(int n)**: Computes Fibonacci numbers using a bottom-up tabulation approach.

## a) Memoization (Top-down)

java Copy Edit

```java
import java.util.HashMap;

public static int fibMemo(int n, HashMap<Integer, Integer> memo) {
    if (n <= 1) return n;
    if (memo.containsKey(n)) return memo.get(n);

    int result = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
    memo.put(n, result);
    return result;
}
```

## b) Tabulation (Bottom-up)

java Copy Edit

```java
public static int fibTab(int n) {
    if (n <= 1) return n;
    int[] dp = new int[n + 1];
    dp[0] = 0; dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```

**Problem 2: Knapsack Problem**
You will solve the 0/1 Knapsack problem using a dynamic programming approach to maximize the value of items in the knapsack without exceeding its capacity.

**Method to Implement:**
**knapsack(int[] weights, int[] values, int capacity)**: Returns the maximum value that can fit in the knapsack of given capacity.

java Copy Edit

```java
public static int knapsack(int[] weights, int[] values, int capacity) {
    int n = weights.length;
    int[][] dp = new int[n + 1][capacity + 1];

    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            if (weights[i - 1] <= w) {
                dp[i][w] = Math.max(
                    values[i - 1] + dp[i - 1][w - weights[i - 1]],
                    dp[i - 1][w]
                );
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][capacity];
}
```

**Problem 3: Longest Common Subsequence (LCS)**
You will find the length of the Longest Common Subsequence (LCS) between two strings using dynamic programming.

**Method to Implement:**
**lcs(String s1, String s2)**: Returns the length of the longest common subsequence between two strings.

```java
public static int lcs(String s1, String s2) {
    int m = s1.length(), n = s2.length();
    int[][] dp = new int[m + 1][n + 1];

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[m][n];
}
```

**Problem 4: Rock Climbing Problem**
You will solve the rock climbing problem, where a climber has to reach the top of a wall with certain energy levels on each move. The goal is to maximize the energy collected.

**Method to Implement:**
**rockClimbing(int[][] wall)**: Given a 2D array representing energy values, calculate the maximum energy collected while climbing from bottom to top.

```java
public static int rockClimbing(int[][] wall) {
    int rows = wall.length;
    int cols = wall[0].length;

    // DP table: max energy collected at each cell
    int[][] dp = new int[rows][cols];

    // Initialize bottom row (start)
    for (int c = 0; c < cols; c++) {
        dp[rows - 1][c] = wall[rows - 1][c];
    }

    // Fill dp from bottom-1 row to top row
    for (int r = rows - 2; r >= 0; r--) {
        for (int c = 0; c < cols; c++) {
            int maxNext = dp[r + 1][c]; // move straight up

            if (c - 1 >= 0) maxNext = Math.max(maxNext, dp[r + 1][c - 1]); // diagonal left
            if (c + 1 < cols) maxNext = Math.max(maxNext, dp[r + 1][c + 1]); // diagonal right

            dp[r][c] = wall[r][c] + maxNext;
        }
    }

    // Result is max value in top row
    int maxEnergy = 0;
    for (int c = 0; c < cols; c++) {
        maxEnergy = Math.max(maxEnergy, dp[0][c]);
    }
    return maxEnergy;
}
```

In this problem you will be given the schedule of $N (1 <= N <= 1000)$ programs for a week. For each $i^{th}$ program you will given four pieces of information, $d_{1i}, t_{1i}, d_{2i}, t_{2i}$. The domain of $d_{1i}$ and $d_{2i}$ is $\{fri, sat, sun, mon, tue, wed, thu\}$ and $0 <= t_{1i}, t_{2i} <= 23$. $d_{1i}$ and $t_{1i}$ denote the starting time of the program (e.g., sun 10), similarly, $d_{2i}$ and $t_{2i}$ denote the ending time of the program.

Each program, will end within a day, meaning, sunday's program will end by sunday and not reach monday. The starting of the week will be considered from friday. Given these information, you want to `select the maximum number of non-overlapping programs to be watched` -- You can not watch multiple programs at the same time. However, after the ending of a program, you can immediately switch to watching another program, that is completely fine.

In the first line, there will be a singler integer value $N$. After that, there will be $N$ lines of information, each line having four values $d_{1i}, t_{1i}, d_{2i}, t_{2i}$ in the resepective manner. You need to print a single integer value denoting the maximum number of non-overlapping programs that can be selected. Print a newline at the end of output for passing all the test cases.

Copy Edit

```
Function getDayIndex(day):
    If day == "fri": return 0
    If day == "sat": return 1
    If day == "sun": return 2
    If day == "mon": return 3
    If day == "tue": return 4
    If day == "wed": return 5
    If day == "thu": return 6
    Return -1

Main:
    Read integer n  // number of intervals
    intervals = empty list

    For i from 0 to n-1:
        Read d1, t1, d2, t2  // start day/time and end day/time
        start = getDayIndex(d1) * 24 + parseInt(t1)
        end = getDayIndex(d2) * 24 + parseInt(t2)
        Add [start, end] to intervals

    Sort intervals by their end time (interval[1]) in ascending order

    count = 0
    lastEnd = -1

    For each interval in intervals:
        If interval.start >= lastEnd:
            count = count + 1
            lastEnd = interval.end

    Print count
```

In this problem you will be given a string $S$ (1<=|S|<=500) containing English lowercase letters. You need to calculate the `minimum number of letters` that are required to be deleted to make the given string a palindrome. A palindrome is a word, phrase, number, or sequence that reads the same backward as forward. For example, `madam`, `bab`, etc.

There will be a single line of input containing the string $S$. You need to print a single integer denoting the minimum number of operations required. Keep a newline after the output to pass all the test cases.

```
Main:
    Read string s
    n = length of s
    Create 2D array dp[n][n], initialized to 0

    For length from 2 to n:
        For i from 0 to n - length:
            j = i + length - 1

            If s[i] == s[j]:
                dp[i][j] = dp[i + 1][j - 1]
            Else:
                dp[i][j] = 1 + min(dp[i + 1][j], dp[i][j - 1])

    Print dp[0][n - 1]
```

```
Function lps(s):
    n = length of s
    Create dp[n][n] // dp[i][j] = length of LPS in s[i...j]

    For i from n-1 down to 0:
        dp[i][i] = 1

        For j from i+1 to n-1:
            If s[i] == s[j]:
                dp[i][j] = 2 + dp[i+1][j-1]
            Else:
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])

    Return dp[0][n-1]

Main:
    Read string s
    minDeletions = length(s) - lps(s)
    Print minDeletions
```

You are an adventurer exploring an ancient dungeon filled with treasures. There are N treasures in a chamber, each having a weight and a value. You can carry at most W units of weight in your magical backpack. You must choose a subset of treasures such that:

- You do not exceed the total weight limit W
- You maximize the total value of the treasures you carry
- You may either take or skip each treasure -- no fractional taking

Your Task:

1. Compute the maximum total value you can carry.
2. Output the indices (1-based) of the treasures that must be taken to achieve this.

Input Format:
First line: Two integers N and W
(1 ≤ N ≤ 1000, 1 ≤ W ≤ 10^5)

Next N lines: Each line contains two integers weight[i] and value[i]
(1 ≤ weight[i] ≤ W, 0 ≤ value[i] ≤ 10^9)



```
Main:
    Read N         // Number of items
    Read W         // Knapsack capacity

    Create arrays we[1..N], val[1..N]

    For i = 1 to N:
        Read we[i], val[i]    // weight and value of each item

    Create 2D array dp[0..N][0..W], initialized to 0

    For i = 1 to N:
        For w = 0 to W:
            If we[i] <= w:
                dp[i][w] = max(
                    dp[i-1][w],                // not take item i
                    dp[i-1][w - we[i]] + val[i]  // take item i
                )
            Else:
                dp[i][w] = dp[i-1][w]          // cannot take item i due to weight

    Print dp[N][W]    // max total value

    selected = empty list
    w = W
    For i = N down to 1:
        If dp[i][w] != dp[i-1][w]:
            Add i to selected    // item i was taken
            w = w - we[i]

    Sort selected in ascending order

    Print selected items
```