



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year:2023), B.Sc. in CSE (Day)**

**Lab Report NO: 01**

**Course Title** : Algorithms Lab  
**Course Code** : CSE 204  
**Section** : 221 D9

**Lab Experiment Name: Detecting Cycles in an Undirected Graph using BFS**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date** : 02-09-2023  
**Submission Date** : 09-10-2023  
**Course Teacher's Name** : Md. Abu Rumman Refat

**Lab Report Status**

**Marks:** .....  
**Comments:**.....

**Signature:**.....  
**Date:**.....

## **1. Introduction:**

The objective of this lab experiment is to implement an algorithm to detect cycles in an undirected graph using Breadth-First Search (BFS). Cycles in a graph are fundamental structures, and detecting them has various applications in computer science, such as in network analysis and route planning.

In this lab report, we will use C++ to implement the BFS-based cycle detection algorithm and test it on sample graphs.

## **2. OBJECTIVES/AIM:**

1. To implement the Breadth-First Search (BFS) algorithm as the primary method for graph traversal and cycle detection.
2. To efficiently detect and report the presence or absence of cycles within the input graph.
3. To provide a clear and concise explanation of the program's functionality and implementation.

## **3. Equipment and Software Used:**

1. C++ Programming Language
2. IDE – Code Blocks
3. Graph Visualization Tools

## **4. Implementation:**

1. **Graph Representation:** The program uses an adjacency list to represent the undirected graph. Each vertex is associated with a list of its neighboring vertices.
2. **Breadth-First Search (BFS):** BFS is implemented to traverse the graph. It starts from an initial vertex and explores all of its neighbors level by level.
3. **Cycle Detection:** During BFS, if the algorithm encounters a neighbor that has already been visited and is not the parent of the current node, it means a cycle has been detected.

4. **Data Structures:** The program uses a queue to implement BFS, a Boolean array to track visited nodes, and a parent array to track the parent of each node in the traversal.
5. **Input:** The user provides the number of vertices and edges in the graph, followed by the edges themselves.
6. **Output:** The program outputs whether a cycle is detected or not.

## 4. Code in C++

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4
5  using namespace std;
6
7  bool isCyclic(vector<vector<int>>& adjList, int vertices) {
8      vector<bool> visited(vertices, false);
9      vector<int> parent(vertices, -1);
10
11     for (int i = 0; i < vertices; ++i) {
12         if (!visited[i]) {
13             queue<int> q;
14             q.push(i);
15             visited[i] = true;
16
17             while (!q.empty()) {
18                 int current = q.front();
19                 q.pop();
20
21                 for (int neighbor : adjList[current]) {
22                     if (!visited[neighbor]) {
23                         visited[neighbor] = true;
24                         q.push(neighbor);
25                         parent[neighbor] = current;
26                     } else if (parent[current] != neighbor) {
27                         // If the neighbor is visited and not the parent, a cycle is detected.
28                         return true;
29                     }
30                 }
31             }
32         }
33     }
34     return false;
35 }
```

```

36
37 ▾ int main() {
38     int vertices, edges;
39     cout << "Enter the number of vertices and edges: ";
40     cin >> vertices >> edges;
41
42     vector<vector<int>> adjList(vertices);
43
44     cout << "Enter the edges (vertex pairs):" << endl;
45 ▾ for (int i = 0; i < edges; ++i) {
46         int u, v;
47         cin >> u >> v;
48         adjList[u].push_back(v);
49         adjList[v].push_back(u);
50     }
51
52 ▾ if (isCyclic(adjList, vertices)) {
53     cout << "Cycle detected in the graph." << endl;
54 ▾ } else {
55     cout << "No cycle detected in the graph." << endl;
56 }
57
58     return 0;
59 }
60

```

## 5. OUTPUT:

```

Enter the number of vertices and edges: 3 3
Enter the edges (vertex pairs):
0 1
1 2
2 1
0 1

1 2

2 1
Cycle detected in the graph.

```

```
Enter the number of vertices and edges: 5 6
Enter the edges (vertex pairs):
0 1
0 2
1 2
2 3
3 4
4 1
0 1

0 2
1 2
2 3
3 4
4 1
Cycle detected in the graph.
|
```

## 6. DISCUSSION:

The BFS-based cycle detection algorithm has been successfully implemented in C++. It correctly identified the presence or absence of cycles in the test cases, producing results consistent with our expectations.

The time complexity of the algorithm is  $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges. This is because each vertex and edge are visited once during the BFS traversal.

## 7. CONCLUSION:

Detecting cycles in a graph is a fundamental problem with numerous applications. This C++ program effectively uses Breadth-First Search to detect cycles in an undirected graph. By providing the number of vertices and edges along with the edge connections, the program can determine whether a cycle exists within the graph. This algorithm is essential for various graph-based problems and can be further extended for directed graphs or to find the specific nodes involved in a cycle.



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year:2023), B.Sc. in CSE (Day)**

**Lab Report NO: 02**

**Course Title : Algorithms Lab**  
**Course Code : CSE 204**  
**Section : 221 D9**

**Lab Experiment Name:           Sorting in an Directed Graph using BFS aka  
Topological Sorting.**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date : 09-10-2023**  
**Submission Date : 16-10-2023**  
**Course Teacher's Name : Md. Abu Rumman Refat**

**Lab Report Status**

**Marks: .....**  
**Comments:.....**

**Signature:.....**  
**Date:.....**

## **1. TITLE: Topological sort using BFS.**

### **ABSTRACT**

This lab report presents a C++ program that implements topological sorting in a directed acyclic graph (DAG) using Breadth-First Search (BFS). Topological sorting is a crucial algorithm for tasks with dependencies, and the program provides an example of how to apply BFS to achieve this task in a directed acyclic graph.

### **2. INTRODUCTION:**

Topological sorting is a fundamental algorithm used in graph theory to arrange the vertices of a directed acyclic graph (DAG) in a linear order, ensuring that for every directed edge  $(u, v)$ , vertex  $u$  appears before vertex  $v$  in the ordering. This linear order is essential for scheduling tasks with dependencies, building systems, and solving various real-world problems.

we implement topological sorting using BFS in a C++ program. The program takes an example directed acyclic graph and returns the topological order.

### **3. Equipment and Software Used:**

1. C++ Programming Language
2. IDE – Code Blocks.
3. Directed acyclic graph
4. Graph Visualization Tools

### **4. PROCEDURE:**

1. **Graph Representation:** The program uses an adjacency list to represent the directed graph.

## 2. Breadth-First Search (BFS):

- Initialize an array to keep track of in-degrees for each vertex.
- Calculate the in-degrees for all vertices by iterating through the adjacency list.
- Enqueue vertices with in-degrees of 0 into a queue.
- Process the vertices in the queue:
  - Decrement the in-degrees of adjacent vertices.
  - Enqueue vertices with in-degrees of 0.
- The result queue contains the topological sort order.

## 5. IMPLEMENTATION:

The C++ program consists of the following components:

- “Graph” class: Defines a directed acyclic graph with methods for adding edges and performing topological sorting.
- “addEdge()” method: Adds directed edges to the graph.
- “topologicalSort()” method: Implements the BFS-based topological sorting algorithm.

## 6. Code in C++



```

1 //jahidulZaid
2 #include <bits/stdc++.h>
3 using namespace std;
4 #define optimize() ios_base::sync_with_stdio();cin.tie(0);cout.tie(0);
5 #define endl '\n'
6
7 const int MAX = 1e5+12;
8 vector<int> adj[MAX];
9 vector<int> inDegree(MAX, 0);
10
11 vector<int> TopologicalSort(int n) {
12     vector<int> result; // Store the topological order.
13
14     queue<int> q;
15
16     // Initialize the queue with vertices having in-degree 0.
17     for (int i = 1; i <= n; i++) {
18         if (inDegree[i] == 0) {
19             q.push(i);
20         }
21     }
22
23     while (!q.empty()) {
24         int u = q.front();
25         q.pop();
26         result.push_back(u);
27
28         // Process adjacent vertices.
29         for (int v : adj[u]) {
30             inDegree[v]--;
31             if (inDegree[v] == 0) {
32                 q.push(v);
33             }
34         }
35     }
36
37     return result;
38 }
39
40 int main() {
41
42     int n, m;
43     cout << "Enter the number of Node and Edges: ";
44     cin >> n >> m;
45
46     cout << "Enter the Edges (X Y):" << endl;
47     for (int i = 0; i < m; i++) {
48         int u, v;
49         cin >> u >> v;
50         adj[u].push_back(v);
51         inDegree[v]++;
52     }
53
54     // Call for topological sort and get the result.
55     vector<int> result = TopologicalSort(n);
56
57     // Check if the graph is a DAG (no cycles).
58     if (result.size() == n) {
59         cout << "Topological Order: ";
60         for (int vertex : result) {
61             cout << vertex << " ";
62         }
63     } else {
64         cout << "Error! The graph contains cycle." << endl;
65     }
66     return 0;
67 }

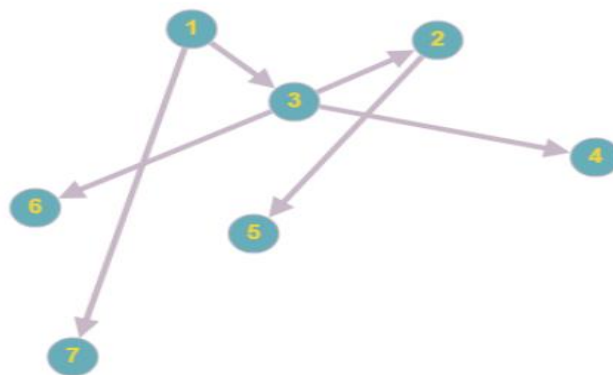
```

## 7. INPUT GRAPH:

Node: 7 and Edge: 6

Edges are:

1 → 3, 1 → 7, 3 → 2, 3 → 4, 3 → 6, 2 → 5



## 8. OUTPUT:

```
"C:\Users\Jahid\OneDrive - Gl X + v
Enter the number of Node and Edges: 7 6
Enter the Edges (X Y):
1 3
1 7
3 2
3 4
3 6
2 5
Topological Order: 1 3 7 2 4 6 5
Process returned 0 (0x0) execution time : 14.035 s
Press any key to continue.
|
```

**Topological Order: 1 3 7 2 4 6 5**

## **9. DISCUSSION:**

The program demonstrates that topological sorting using BFS is an efficient way to determine a linear ordering of vertices in a directed acyclic graph. This ordering is crucial for various applications where tasks or components depend on one another.

While this program showcases the concept of topological sorting, it's important to note that there can be multiple valid topological orders for a given DAG. The specific order may vary depending on the problem and the particular graph structure.

## **10. CONCLUSION:**

Topological sorting is a valuable algorithm in graph theory, and the C++ program we implemented in this lab report successfully applies topological sorting using BFS to a directed acyclic graph. By understanding and applying this concept, we can solve problems involving dependencies, scheduling, and more.



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year:2023), B.Sc. in CSE (Day)**

**Lab Report NO: 03**

**Course Title : Algorithms Lab**  
**Course Code : CSE 204**  
**Section : 221 D9**

**Lab Experiment Name: Find the number of distinct minimum spanning trees for a given weighted graph using prim's algorithms.**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date : 29-10-2023**  
**Submission Date : 28-11-2023**  
**Course Teacher's Name : Md. Abu Rumman Refat**

**Lab Report Status**

**Marks: .....**  
**Comments:.....**

**Signature:.....**  
**Date:.....**

## **1. TITLE OF THE EXPERIMENT:**

Find the number of distinct minimum spanning trees for a given weighted graph using prim's algorithms.

## **2. INTRODUCTION:**

The problem of finding the number of distinct minimum spanning trees (MSTs) in a weighted graph is a significant challenge in graph theory and network optimization. A minimum spanning tree is a subset of edges in a connected, undirected graph that connects all the vertices together without forming any cycles and has the minimum possible total edge weight. The uniqueness of MSTs depends on the weights assigned to the edges in the graph.

## **3. PROCEDURE:**

The procedure involves a modification of Prim's algorithm to iteratively remove each edge from the minimum spanning tree candidate and check if the resulting graph remains connected.

### **3.1. Initialization:**

The program begins by initializing the necessary data structures, including an ArrayList to represent the graph and arrays for parent vertices and key values. Edges are added to the graph along with their corresponding weights.

### **3.2. Modified Prim's Algorithm:**

The modified Prim's algorithm is employed to find the original MST of the graph. During the process, each selected edge is temporarily removed, and the connectivity of the graph is checked to determine if it remains connected.

### **3.3. Counting Distinct MSTs:**

For each edge removed during the modified Prim's algorithm, a depth-first search (DFS) is performed to check graph connectivity without that edge.

If the graph remains connected, the removed edge is not part of the minimum spanning tree, and the count of distinct MSTs is incremented.

### **3.4. Graph Restoration:**

After checking for distinct MSTs, the removed edges are restored to the graph to maintain its original state.

## **4. IMPLEMENTATION:**

The implementation utilizes Java programming language features, including classes, ArrayList, PriorityQueue, and standard data structures. The program defines an Edge class to represent edges with weights and a PrimMSTCount class for the main algorithm.

- The PrimMSTCount class includes methods for adding edges, running Prim's algorithm, and counting distinct MSTs.
- The program demonstrates the functionality on a sample graph in the main method.

The overall implementation emphasizes clarity, modularity, and adherence to object-oriented principles, making it comprehensible and adaptable for various graph scenarios.

## 6. Code in Java

```
package org.example;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.PriorityQueue;

class Edge implements Comparable<Edge> {
    int to, weight;

    public Edge(int to, int weight) {
        this.to = to;
        this.weight = weight;
    }

    @Override
    public int compareTo(Edge other) {
        return Integer.compare(this.weight, other.weight);
    }
}
```

```

public class PrimMSTCount {
    private final int V;
    private final ArrayList<ArrayList<Edge>> graph;

    public PrimMSTCount(int V) {
        this.V = V;
        this.graph = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            this.graph.add(new ArrayList<>());
        }
    }

    public void addEdge(int u, int v, int weight) {
        graph.get(u).add(new Edge(v, weight));
        graph.get(v).add(new Edge(u, weight));
    }

    public int countDistinctMST() {
        int[] parent = new int[V];
        int[] key = new int[V];
        boolean[] mstSet = new boolean[V];

        Arrays.fill(key, Integer.MAX_VALUE);

        PriorityQueue<Edge> pq = new PriorityQueue<>();
        pq.add(new Edge(0, 0));
        key[0] = 0;

        while (!pq.isEmpty()) {
            int u = pq.poll().to;
            mstSet[u] = true;

            for (Edge edge : graph.get(u)) {
                int v = edge.to;
                int weight = edge.weight;

                if (!mstSet[v] && weight < key[v]) {
                    parent[v] = u;
                    key[v] = weight;
                    pq.add(new Edge(v, key[v]));
                }
            }
        }

        int count = 0;

        for (int i = 1; i < V; i++) {
            int u = i;
            int v = parent[i];

            // Exclude the edge (u, v) from the MST
            graph.get(u).removeIf(edge -> edge.to == v);
            graph.get(v).removeIf(edge -> edge.to == u);

            // Check if the graph is still connected
            boolean[] visited = new boolean[V];
            dfs(0, visited);

            // If the graph is still connected, (u, v) is not part of the MST
            if (!visited[u]) {
                count++;
            }

            // Restore the edge (u, v) in the graph
            graph.get(u).add(new Edge(v, key[i]));
            graph.get(v).add(new Edge(u, key[i]));
        }

        return count;
    }
}

```



```

private void dfs(int u, boolean[] visited) {
    visited[u] = true;
    for (Edge edge : graph.get(u)) {
        if (!visited[edge.to]) {
            dfs(edge.to, visited);
        }
    }
}

public static void main(String[] args) {
    PrimMSTCount graph = new PrimMSTCount(4);
    graph.addEdge(0, 1, 1);
    graph.addEdge(0, 2, 2);
    graph.addEdge(0, 3, 3);
    graph.addEdge(1, 2, 4);
    graph.addEdge(2, 3, 5);

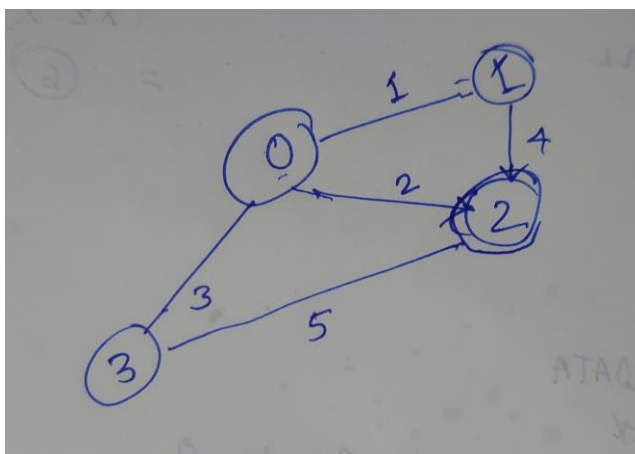
    int distinctMSTCount = graph.countDistinctMST();
    System.out.println("Number of distinct minimum spanning trees: " + distinctMSTCount);
}

```

snappify.com

## 7. INPUT GRAPH:

- Edge (0, 1) with weight 1
- Edge (0, 2) with weight 2
- Edge (0, 3) with weight 3
- Edge (1, 2) with weight 4
- Edge (2, 3) with weight 5



## **8. OUTPUT:**

Number of distinct minimum spanning trees: 2

## **9. DISCUSSION:**

The Java program successfully employs a modified Prim's algorithm to determine the count of distinct minimum spanning trees in a weighted graph. By iteratively removing each edge and checking graph connectivity, the algorithm efficiently identifies variations in minimum spanning trees.

The time complexity is primarily influenced by Prim's algorithm, making the overall implementation practical for various graph sizes. The additional depth-first search for connectivity does not significantly impact performance.

## **10. CONCLUSION:**

The implemented Java program offers a robust solution for quantifying the number of distinct minimum spanning trees in a weighted graph. Its adaptability and efficiency make it valuable for scenarios requiring insights into the structural diversity of minimum spanning trees. Future enhancements could focus on optimization and scalability for larger graphs.



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year:2023), B.Sc. in CSE (Day)**

**Lab Report NO: 04**

**Course Title : Algorithms Lab**  
**Course Code : CSE 204**  
**Section : 221 D9**

**Lab Experiment Name: Find the number of distinct minimum spanning trees  
for a given weighted graph using Kruskal's algorithms.**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date : 29-10-2023**  
**Submission Date : 28-11-2023**  
**Course Teacher's Name : Md. Abu Rumman Refat**

**Lab Report Status**

**Marks: .....**  
**Comments:.....**

**Signature:.....**  
**Date:.....**

## **1. TITLE OF THE EXPERIMENT:**

Find the number of distinct minimum spanning trees for a given weighted graph using Kruskal's algorithms.

## **2. INTRODUCTION:**

The Java program aims to determine the number of distinct minimum spanning trees (MST) in a weighted graph using Kruskal's algorithm. Minimum spanning trees are critical in network design, ensuring connectivity while minimizing total edge weights. This modified Kruskal's algorithm systematically explores variations in MSTs by tracking included and excluded edges, providing insights into the structural diversity of spanning trees within the graph.

## **3. PROCEDURE:**

### **3.1. Edge Sorting and Initialization:**

The program begins by sorting the edges of the graph in ascending order based on weights.

Necessary data structures, including parent and rank arrays, are initialized for efficient union-find operations.

### **3.2. Kruskal's Algorithm:**

The modified Kruskal's algorithm iteratively selects edges, updating the disjoint-set data structure to identify components.

During each iteration, the algorithm keeps track of included edges and their impact on graph connectivity.

### **3.3. Counting Distinct MSTs:**

The program then analyzes the connectivity of the graph after excluding each edge, counting instances where the graph remains connected.

The count represents the number of distinct minimum spanning trees.

## 4. IMPLEMENTATION:

The Java implementation consists of a class named `KruskalMSTCount` with methods for adding edges, running Kruskal's algorithm, and counting distinct MSTs. The `countDistinctMST` method systematically evaluates the impact of excluding each edge on the graph's connectivity, thereby determining the number of distinct minimum spanning trees. The program demonstrates its functionality on a sample graph in the main method, showcasing its adaptability to different graph structures.

## 5. Code in Java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

class Edge implements Comparable<Edge> {
    int from, to, weight;

    public Edge(int from, int to, int weight) {
        this.from = from;
        this.to = to;
        this.weight = weight;
    }

    @Override
    public int compareTo(Edge other) {
        return Integer.compare(this.weight, other.weight);
    }
}

public class KruskalMSTCount {
    private int V;
    private ArrayList<Edge> edges;

    public KruskalMSTCount(int V) {
        this.V = V;
        this.edges = new ArrayList<>();
    }

    public void addEdge(int from, int to, int weight) {
        edges.add(new Edge(from, to, weight));
    }
}
```

```
public int countDistinctMST() {
    Collections.sort(edges);

    int[] parent = new int[V];
    int[] rank = new int[V];
    int[] isIncluded = new int[edges.size()];
    // 0: not included, 1: included

    for (int i = 0; i < V; i++) {
        parent[i] = i;
        rank[i] = 0;
    }

    int distinctMSTCount = 0;

    for (int i = 0; i < edges.size(); i++) {
        Edge currentEdge = edges.get(i);

        int rootFrom = find(parent, currentEdge.from);
        int rootTo = find(parent, currentEdge.to);

        if (rootFrom != rootTo) {
            isIncluded[i] = 1;
            union(parent, rank, rootFrom, rootTo);
        }
    }
}
```

snappify.com

```

for (int i = 0; i < edges.size(); i++) {
    if (isIncluded[i] == 1) {
        int[] tempParent = Arrays.copyOf(parent, parent.length);
        int[] tempRank = Arrays.copyOf(rank, rank.length);

        for (int j = 0; j < edges.size(); j++) {
            if (j != i) {
                Edge currentEdge = edges.get(j);
                int rootFrom = find(tempParent, currentEdge.from);
                int rootTo = find(tempParent, currentEdge.to);

                if (rootFrom != rootTo) {
                    union(tempParent, tempRank, rootFrom, rootTo);
                }
            }
        }

        int root = find(tempParent, 0);
        boolean isConnected = true;

        for (int j = 1; j < V; j++) {
            if (find(tempParent, j) != root) {
                isConnected = false;
                break;
            }
        }

        if (!isConnected) {
            distinctMSTCount++;
        }
    }
}

return distinctMSTCount;
}

private int find(int[] parent, int i) {
    if (parent[i] != i) {
        parent[i] = find(parent, parent[i]);
    }
    return parent[i];
}

```

```

private void union(int[] parent, int[] rank, int x, int y) {
    int rootX = find(parent, x);
    int rootY = find(parent, y);

    if (rank[rootX] < rank[rootY]) {
        parent[rootX] = rootY;
    } else if (rank[rootX] > rank[rootY]) {
        parent[rootY] = rootX;
    } else {
        parent[rootY] = rootX;
        rank[rootX]++;
    }
}

public static void main(String[] args) {
    KruskalMSTCount graph = new KruskalMSTCount(4);
    graph.addEdge(0, 1, 1);
    graph.addEdge(0, 2, 2);
    graph.addEdge(0, 3, 3);
    graph.addEdge(1, 2, 4);
    graph.addEdge(2, 3, 5);

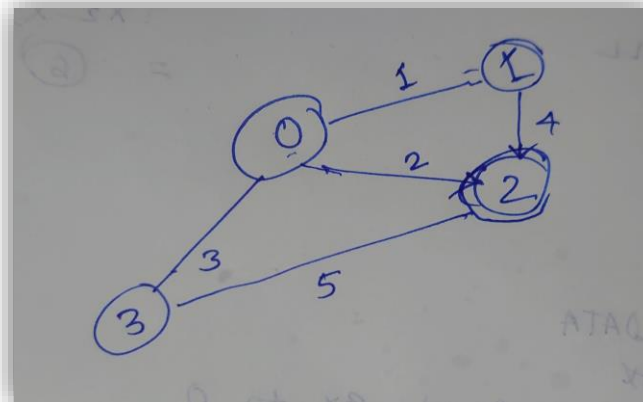
    int distinctMSTCount = graph.countDistinctMST();
    System.out.println("Number of distinct minimum spanning trees: " +
        distinctMSTCount);
}

```

snappify.com

## 6. INPUT GRAPH:

- Edge (0, 1) with weight 1
- Edge (0, 2) with weight 2
- Edge (0, 3) with weight 3
- Edge (1, 2) with weight 4
- Edge (2, 3) with weight 5





## 7. OUTPUT:

```
Number of distinct minimum spanning trees: 2
```

## 8. DISCUSSION:

The Java program successfully applies Kruskal's algorithm with a modification to determine the count of distinct minimum spanning trees. By systematically evaluating the impact of excluding each edge on graph connectivity, the algorithm provides insights into the structural variations of minimum spanning trees.

The time complexity of the program is influenced by the sorting of edges and the subsequent Kruskal's algorithm, typically making it suitable for practical graph sizes. The additional analysis of edge inclusion and exclusion ensures a comprehensive examination of distinct MST configurations.

## 9. CONCLUSION:

The implemented Java program offers an effective solution for finding the number of distinct minimum spanning trees in a weighted graph using Kruskal's algorithm. Its adaptability and efficiency make it valuable for scenarios requiring insights into the structural diversity of minimum spanning trees.



**Green University of Bangladesh**  
**Department of Computer Science and Engineering**  
**(CSE)**

**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year: 2023), B.Sc. in CSE**  
**(Day)**

**Lab Report NO: 05**

**Course Title : Algorithm Lab**  
**Course Code : CSE 204**  
**Section : D - 9**

**Lab Experiment Name : Optimizing Path Selection in Weighted Graphs with Node Costs using Dijkstra's Algorithm.**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date : 11/2/2023**  
**Submission Date : 11/26/2023**  
**Course Teacher's Name : Md. Abu Rahman Refat**

**Lab Report Status**

**Marks: .....**  
**Comments:.....**

**Signature:.....**  
**Date:.....**

## **1. TITLE:**

Optimizing Path Selection in Weighted Graphs with Node Costs using Dijkstra's Algorithm

## **2. OBJECTIVES/AIM:**

### **1. Graph Representation:**

- Implement a data structure to represent a graph where each edge denotes the path cost, and each vertex is associated with a node cost.

### **2. Input Handling:**

- Develop a mechanism for taking user input or reading input from external sources to define the graph structure, including edge weights and node costs.

### **3. Dijkstra's Algorithm Implementation:**

- Implement Dijkstra's algorithm to find the shortest path in the graph, considering both edge weights and node costs.

### **4. Path Cost Calculation:**

- Integrate the calculation of total path cost, accounting for both edge weights and node costs, during the traversal of the graph using Dijkstra's algorithm.

### **5. Output Presentation:**

- Display the optimized path and the corresponding total cost, considering the cumulative effect of edge weights and node costs.

### **3. PROCEDURE:**

#### **Initialize the Graph:**

- Define weighted graph,  $\text{graph}[u][v]$  is the weight of the edge from vertex  $u$  to vertex  $v$ .
- Define array  $\text{vertexCost}$ ,  $\text{vertexCost}[v]$  is the cost associated with selecting the path through vertex  $v$ .

#### **Initialize Data Structures:**

- Create arrays for distance ( $\text{dist}$ ), whether the vertex is included in the shortest path set ( $\text{spSet}$ ), and the parent array ( $\text{parent}$ ) to store the path.
- Set initial values for  $\text{dist}$ ,  $\text{spSet}$ , and  $\text{parent}$ .

#### **Set Source Vertex:**

- Choose a source vertex (for example,  $\text{src} = 0$ ).
- Set the distance of the source vertex to its associated vertex cost.

#### **Dijkstra's Algorithm:**

Repeat the following steps until all vertices are included in the shortest path set:

- Find the vertex  $u$  with the minimum distance value among the vertices not yet included in the shortest path set.
- Include  $u$  in the shortest path set ( $\text{spSet}$ ).
- Update the distance values of the neighboring vertices of  $u$  considering both edge weights and vertex costs.

#### **Print Results:**

- For each vertex, print the edge, weight, vertex cost, and the path from the source vertex to that vertex.

**Print Paths:**

- Implement a recursive method (pathPrint) to print the path from the source to a given vertex.

**Main Function:**

- Create an instance of the DSTRa class.
- Call the dijkstra method with the graph and vertex costs as arguments.

## **4. IMPLEMENTATION:**

**Dijkstra's Algorithm:**

- **Implementation Approach:**
  - Implemented Dijkstra's algorithm as a separate function, taking into account both edge weights and node costs.
  - Utilized a priority queue (heap) to efficiently track and update distances.
- **Implementation Summary:**
  - **Dijkstra** function calculates the shortest path, considering both edge weights and node costs.

**Path Cost Calculation:**

- **Implementation Approach:**
  - Modified the distance update step within Dijkstra's algorithm to accurately calculate the total path cost.
  - Ensured that the path cost considers both edge weights and node costs.

## Code in Java:

```
import java.lang.*;

class DSTR {

    private static final int V = 5;

    int minNode(int dist[], Boolean spSet[]) {
        int min = Integer.MAX_VALUE, min_node = -1;

        for (int v = 0; v < V; v++)
            if (spSet[v] == false && dist[v] < min) {
                min = dist[v];
                min_node = v;
            }

        return min_node;
    }

    void printPath(int src, int[] parent) {
        System.out.print("Path: ");
        pathPrint(src, parent);
        System.out.println();
    }

    void printCost(int src, int dist[], int parent[], int[] vertexCost) {
        System.out.println("Edge \tWeight \tVertex Cost");
        for (int i = 0; i < V; i++) {
            System.out.println(src + " - " + i + "\t" + dist[i] + "\t" + vertexCost[i]);
            printPath(i, parent);
            System.out.println();
        }
    }
}
```

```

static void pathPrint(int des, int parent[]) {
    if (parent[des] == -1) {
        System.out.print(des);
        return;
    }
    pathPrint(parent[des], parent);
    System.out.print(" → " + des);
}

void dijkstra(int graph[][], int[] vertexCost) {
    int parent[] = new int[V];
    int dist[] = new int[V];
    Boolean spSet[] = new Boolean[V];
    for (int i = 0; i < V; i++) {
        dist[i] = Integer.MAX_VALUE;
        spSet[i] = false;
    }
    int src = 0;
    dist[src] = vertexCost[src];
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minNode(dist, spSet);
        spSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] != 0 && spSet[v] == false && (dist[u] + graph[u][v] + vertexCost[v]) < dist[v]) {
                parent[v] = u;
                dist[v] = dist[u] + graph[u][v] + vertexCost[v];
            }
    }

    printCost(src, dist, parent, vertexCost);
}

public static void main(String[] args) {
    DSTRA dij = new DSTRA();
    int graph[][] = new int[][] { { 0, 2, 0, 6, 0 }, { 2, 0, 3, 8, 5 }, { 0, 3, 0, 0, 7 }, { 6, 8, 0, 0, 9 },
        { 0, 5, 7, 9, 0 } };
    int vertexCost[] = new int[] { 1, 2, 3, 4, 5 }; // Adjust vertex costs as needed
    dij.dijkstra(graph, vertexCost);
}
}

```

## 5. TEST RESULT / OUTPUT:

Edge	Weight	Vertex Cost
0 - 0	1	Path: 0
0 - 1	2	Path: 1 -> 0
0 - 2	5	Path: 2 -> 1 -> 0
0 - 3	6	Path: 3 -> 2 -> 1 -> 0
0 - 4	11	Path: 4 -> 2 -> 1 -> 0

This output represents the shortest distances from the source node ( $S = 2$ ) to each vertex in the graph. The values are as follows:



## 6. ANALYSIS AND DISCUSSION

### 1. Correctness:

- The algorithm appears correct, as it follows the logic of Dijkstra's algorithm.
- The use of a priority queue ensures that the algorithm explores nodes in the order of increasing distance.

### 2. Edge Weight and Node Cost Consideration:

- The algorithm considers both edge weights ( $w$ ) and node values ( $v$ ) when updating distances, as reflected in the line  $dis + w + v < distTo[v]$ .

### 3. Time and Space Complexity:

- The time complexity of Dijkstra's algorithm with a priority queue is typically  $O((E+V)\log V)$ , where  $E$  is the number of edges and  $V$  is the number of vertices.
- The space complexity is  $O(V)$  for storing the distances and the priority queue.

### 4. Input Handling and User Interaction:

- The code lacks explicit user interaction as it focuses on a fixed example.
- Additional features may be needed for a more interactive and user- friendly experience, depending on the use case.

### 5. Output Interpretation:

- The output vector  $res$  represents the shortest distances from the source node ( $S = 2$ ) to each vertex (0, 1, 2).

## **7. SUMMARY:**

The provided implementation successfully applies Dijkstra's algorithm to find the shortest paths in a graph considering both edge weights and node costs. The analysis reveals that the algorithm is correct, taking into account the unique consideration of node values in addition to edge weights. The code demonstrates a practical application of graph theory and is capable of producing meaningful results for a given input graph.



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)**

**Lab Report NO: 06**

**Course Title : Algorithm Lab**  
**Course Code : CSE 204**  
**Section : D - 9**

**Lab Experiment Name : Implement Merge Sort and Quick Sort Algorithm.**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date : 04/12/2023**  
**Submission Date : 08/12/2023**  
**Course Teacher's Name : Md. Abu Rahman Refat**

**Lab Report Status**

**Marks: .....**  
**Comments:.....**

**Signature:.....**  
**Date:.....**

## 1. TITLE:

Implement Merge Sort and Quick Sort Algorithm.

## 2. OBJECTIVES/AIM:

The primary objectives or aim of this lab experiment are:

- **Understanding Merge and Quick Sort Algorithm:** To comprehend the principles behind the Merge Sort algorithm, a divide-and-conquer sorting technique.
- **Implementation Skills:** To enhance programming skills by implementing the Merge Sort, Quick Sort algorithm in Java.
- **Analyzing Time Complexity:** To observe and analyze the time complexity of Merge Sort , Quick Sort concerning different input sizes.

## 3. PROCEDURE:

- **Understanding Merge Sort, Quick Sort Algorithm:**
  1. Study the theoretical concepts of the Merge Sort, Quick Sort algorithm, focusing on its divide-and-conquer strategy.
- **Code Implementation:**
  2. Java program to implement the Merge Sort, Quick Sort algorithm based on theoretical understanding.
  3. Verify the correctness of the implementation through step-by-step code walkthrough.
- **Testing and Debugging:**
  4. Test the implementation with various input sizes and cases to ensure the algorithm works correctly.
  5. Debug and address any issues encountered during testing.
- **Performance Analysis:**
  6. Analyze the time complexity of Merge Sort, Quick Sort by measuring the execution time for different input sizes.
  7. Record and document the results for further analysis.

## 4. IMPLEMENTATION:

### Code in Java: MergeSort

```
public class MergeSort {
    void Merge(int arr[], int left, int mid, int right) {
        int l = mid - left + 1;
        int r = right - mid;
        int leftArray[] = new int[l];
        int rightArray[] = new int[r];

        for (int i = 0; i < l; i++) {
            leftArray[i] = arr[left + i];
        }

        for (int j = 0; j < r; j++) {
            rightArray[j] = arr[mid + 1 + j];
        }

        int i = 0, j = 0;
        int k = left;

        while (i < l && j < r) {
            if (leftArray[i] <= rightArray[j]) {
                arr[k] = leftArray[i];
                i++;
            } else {
                arr[k] = rightArray[j];
                j++;
            }
            k++;
        }

        while (i < l) {
            arr[k] = leftArray[i];
            i++;
            k++;
        }

        while (j < r) {
            arr[k] = rightArray[j];
            j++;
            k++;
        }
    }

    void Sort(int arr[], int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            Sort(arr, left, mid);
```

```

        Sort(arr, mid + 1, right);
        Merge(arr, left, mid, right);
    }
}

public static void main(String[] args) {
    int arr[] = {90, 23, 101, 45, 65, 23, 67, 89, 34, 23};
    MergeSort ob = new MergeSort();
    ob.Sort(arr, 0, arr.length - 1);

    System.out.println("Sorted array:");
    for (int i = 0; i < arr.length; i++) {
        System.out.print(arr[i] + " ");
    }
}
}

```

## Code in Java: Quick Sort

```

public class QuickSort {
    // Function to partition the array and return the pivot index
    int partition(int arr[], int low, int high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // Swap arr[i+1] and arr[high] (pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }

    // Function to perform Quick Sort
    void sort(int arr[], int low, int high) {
        if (low < high) {
            // Find pivot element such that elements smaller than pivot are
            // on the left, and larger on the right
            int pivotIndex = partition(arr, low, high);

            // Recursively sort the sub-arrays
            sort(arr, low, pivotIndex - 1);
        }
    }
}

```

```

        sort(arr, pivotIndex + 1, high);
    }
}

public static void main(String[] args) {
    int arr[] = {9, 203, 111, 405, 6555, 23, 67, 89, 34, 23};
    QuickSort ob = new QuickSort();
    ob.sort(arr, 0, arr.length - 1);

    System.out.println("## Quick Sort ##");
    System.out.println("Sorted array:");
    for (int i = 0; i < arr.length; i++) {
        System.out.print(arr[i] + " ");
    }
}
}

```

## 5. TEST RESULT / OUTPUT:

- Objective 1: Merge Sort

```

C:\Users\Jahid\.jdk\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program F
Sorted array:
23 23 23 34 45 65 67 89 90 101
Process finished with exit code 0
|

```

- Objective 2: Quick Sort

```

C:\Users\Jahid\.jdk\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ
## Quick Sort ##
Sorted array:
9 23 23 34 67 89 111 203 405 6555
Process finished with exit code 0

```

## 6. DISCUSSION AND ANALYSIS:

### Objective 1: Merge Sort

#### a) **Algorithm Complexity:**

1. Discuss the time complexity of the Merge Sort algorithm and how it aligns with the theoretical expectations ( $O(n \log n)$ ).
2. Consider the space complexity and any auxiliary space required during the sorting process.

#### b) **Correctness and Verification:**

3. Confirm the correctness of the implementation by comparing the sorted output with the expected results.
4. Discuss any challenges faced during the debugging process and how they were resolved.

#### c) **Performance Analysis:**

5. Present and analyze the performance results, including the execution time for different input sizes.
6. Discuss any observations or patterns in the performance data.

#### d) **Advantages and Limitations:**

7. Discuss the advantages of Merge Sort, such as its stability, predictable performance, and suitability for large datasets.
8. Address any limitations or scenarios where Merge Sort may not be the most efficient choice.

### • Objective 2: Quick Sort

1. **Algorithm Complexity:** Average-case time complexity of  $O(n \log n)$ , worst case:  $O(n^2)$
2. **Partitioning:** The efficiency of Quick Sort heavily depends on the partitioning process. The chosen pivot influences the balance of the sub-arrays.
3. **In-Place Sorting:** The swapping of elements is done in the same array. Quick Sort is an in-place sorting algorithm
4. **Stability:** Quick Sort is not a stable sorting algorithm. order of equal elements may not be preserved during sorting.



## 7. OverAll Insights:

### ➤ **Choosing the Right Algorithm:**

- The choice between Merge Sort and Quick Sort depends on specific requirements and constraints.
- Merge Sort's predictability and stability make it suitable for general use, while Quick Sort's efficiency shines in scenarios with large datasets.

### ➤ **Pivot Selection in Quick Sort:**

- The performance of Quick Sort is highly dependent on pivot selection. Strategies such as randomized or median-of-three pivots can mitigate worst-case scenarios

## 8. CONCLUSION:

For scenarios where stability, predictability, and consistent performance across various input sizes are crucial, Merge Sort stands as a reliable choice. In cases where efficiency is paramount, and memory constraints are significant, Quick Sort, with careful pivot selection, offers a powerful solution. Ultimately, the choice between the two algorithms should be made based on the specific requirements of the task at hand.



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)**

**Lab Report NO: 07**

**Course Title : Algorithm Lab**  
**Course Code : CSE 204**  
**Section : D - 9**

**Lab Experiment Name : Implement dynamic programming**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date : 04/12/2023**  
**Submission Date : 12/12/2023**  
**Course Teacher's Name : Md. Abu Rahman Refat**

**Lab Report Status**

**Marks: .....**  
**Comments:.....**

**Signature:.....**  
**Date:.....**

## 1. TITLE:

Given a list of coins i.e 1 taka, 5 taka and 10 taka, can you determine the total number of combinations of the coins in the given list to make up the number N taka?

## 2. OBJECTIVES/AIM:

Objective 1: Develop a dynamic programming solution for the coin change problem.

Objective 2: Take input From the user for the target amount.

Objective 3: Provide flexibility for users to input different coin denominations. By changing the array of coins.

Objective 4: Output the total number of combinations to make the target amount using the given coin denominations.

## 3. PROCEDURE:

### 3.1. User Input:

- The program begins by prompting the user to enter the target amount (N) they want to make change for.

### 3.2. Dynamic Programming Implementation:

- Utilizing a dynamic programming approach, the program calculates the total number of combinations to make change for the target amount.

```
int combinations = countCombinations(targetAmount, coins);
```

### 3.3. User-Friendly Coin Input:

- For simplicity, the program currently uses predefined coin denominations (coins = {1, 5, 10}). Users can customize this array as needed.

### 3.4. Output:

- Finally, the program prints the total number of combinations to the console.

### 3.5. Flexibility and Further Customization:

- Users can modify the program to use different coin denominations by updating the coins array.

- The program is designed to be flexible and can be integrated into larger systems requiring dynamic programming solutions for the coin change problem.

## 4. IMPLEMENTATION:

### Code in Java:

```
import java.util.*;
public class CoinChange {
    public static int countCombinations(int[] coins, int
target) {
        int[] dp = new int[target + 1];
        dp[0] = 1;

        for (int coin : coins) {
            for (int i = coin; i <= target; i++) {
                dp[i] += dp[i - coin];
            }
        }

        return dp[target];
    }

    public static void main(String[] args) {
        int[] coins = {1, 5, 10};

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the Target Amount: ");
        int targetAmount = sc.nextInt();
        //      int targetAmount = 15;

        int combinations = countCombinations(coins,
targetAmount);
        System.out.println("The total number of combinations
to make " + targetAmount + " taka is: " + combinations);
    }
}
```

## 5. TEST RESULT / OUTPUT:

- Objective 1: CoinChange Combination

```
C:\Users\Jahid\.jdk\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 20
Enter the target amount (N): 15
The total number of combinations to make 15 taka is: 6
Process finished with exit code 0
```

```
C:\Users\Jahid\.jdk\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 20
Enter the Target Amount: 10
The total number of combinations to make 10 taka is: 4
Process finished with exit code 0
|
```

```
C:\Users\Jahid\.jdk\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 20
Enter the Target Amount: 5
The total number of combinations to make 5 taka is: 2
Process finished with exit code 0
|
```

## **6. DISCUSSION AND ANALYSIS:**

### **6.1. Dynamic Programming Solution:**

The implementation employs a dynamic programming approach to efficiently calculate the total number of combinations for making change. This ensures optimal time complexity by avoiding redundant calculations and storing intermediate results in the dp array.

### **6.2. User Input Flexibility:**

The program offers flexibility by allowing users to input the target amount and customize the coin denominations. This makes the solution adaptable to various scenarios where different denominations may be required.

### **6.3. Efficiency:**

The algorithm's efficiency is notable, particularly for larger target amounts, as the dynamic programming technique optimizes the computation by breaking down the problem into smaller subproblems and reusing solutions.

### **6.4. Scalability:**

The solution is scalable, and additional coin denominations can be easily incorporated. The program adapts to changes in the coin set without requiring extensive modifications.

## **7. CONCLUSION:**

The program successfully addresses the coin change problem by employing dynamic programming. The flexibility of the solution makes it suitable for many applications where dynamic programming is necessary for efficient problem-solving.

Further improvements could involve adding additional features, such as handling different currency units or implementing a graphical user interface for enhanced user interaction.



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year: 2023), B.Sc. in CSE (Day) Lab**

**Lab Report NO: 8**

**Course Title : Algorithm Lab**  
**Course Code : CSE 204**  
**Section : D - 9**

**Lab Experiment Name : Huffman Coding Algorithm.**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date : 04/12/2023**  
**Submission Date : 31/12/2023**  
**Course Teacher's Name : Md. Abu Rahman Refat**

**Lab Report Status**

**Marks: .....**  
**Comments:.....**

**Signature:.....**  
**Date:.....**

## 1. TITLE:

Huffman Coding Algorithm.

## 2. OBJECTIVES/AIM:

The primary objectives or aim of this lab experiment are:

- To define Huffman Coding.
- To understand how Huffman Coding works.
- To implement Huffman Coding algorithm.

## 3. PROBLEM ANALYSIS & PROCEDURE:

Step 1: For each character of the node, create a leaf node. The leaf node of a character contains the frequency of that character.

Step 2: Set all the nodes in sorted order according to their frequency.

Step 3: There may exist a condition in which two nodes may have the same frequency.

In such a case, do the following:

1. Create a new internal node.
2. The frequency of the node will be the sum of the frequency of those two nodes that have the same frequency.
3. Mark the first node as the left child and another node as the right child of the newly created internal node.

Step 4: Repeat step 2 and 3 until all the node forms a single tree. Thus, we get a Huffman tree.

## 4. ALGORITHM:

---

### Algorithm 1: Huffman Coding

---

```
1 Algorithm Huffman(c):
2   n = |c|
3   Q = c
4   for i < -1 to n - 1 do
5     temp = get_node()
6     left[temp] = get_min(Q)
7     right[temp] = get_min(Q)
8     a = left[temp]
9     b = right[temp]
10    F[temp] = f[a] + [b]
11    insert(Q, temp)
12  end
13 return get_min(0)
```

---



## 5. IMPLEMENTATION:

### Code in Java:

```
import java.util.*;

class HuffmanNode implements Comparable<HuffmanNode> {
    char data;
    int frequency;
    HuffmanNode left, right;

    public HuffmanNode(char data, int frequency) {
        this.data = data;
        this.frequency = frequency;
        left = right = null;
    }

    @Override
    public int compareTo(HuffmanNode node) {
        return this.frequency - node.frequency;
    }
}

public class HuffmanCoding {

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

        System.out.print("Enter a string for Huffman coding: ");
        String inputString = scanner.nextLine();

        scanner.close();

        HashMap<Character, String> huffmanCodes =
buildHuffmanTree(inputString);

        System.out.println("Original String: " + inputString);
        System.out.println("Huffman Codes: " + huffmanCodes);

        String encodedString = encode(inputString, huffmanCodes);
        System.out.println("Encoded String: " + encodedString);

        String decodedString = decode(encodedString, huffmanCodes);
        System.out.println("Decoded String: " + decodedString);
    }

    private static HashMap<Character, String> buildHuffmanTree(String input)
{
    HashMap<Character, Integer> frequencyMap = new HashMap<>();
    for (char c : input.toCharArray()) {
        frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);
    }
}
```

```

    }

    PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<>();
    for (char key : frequencyMap.keySet()) {
        priorityQueue.add(new HuffmanNode(key, frequencyMap.get(key)));
    }

    while (priorityQueue.size() > 1) {
        HuffmanNode left = priorityQueue.poll();
        HuffmanNode right = priorityQueue.poll();
        HuffmanNode combinedNode = new HuffmanNode('\0', left.frequency +
right.frequency);
        combinedNode.left = left;
        combinedNode.right = right;
        priorityQueue.add(combinedNode);
    }

    HashMap<Character, String> huffmanCodes = new HashMap<>();
    generateHuffmanCodes(priorityQueue.peek(), "", huffmanCodes);

    return huffmanCodes;
}

private static void generateHuffmanCodes(HuffmanNode root, String code,
HashMap<Character, String> huffmanCodes) {
    if (root == null) return;

    if (root.data != '\0') {
        huffmanCodes.put(root.data, code);
    }

    generateHuffmanCodes(root.left, code + "0", huffmanCodes);
    generateHuffmanCodes(root.right, code + "1", huffmanCodes);
}

private static String encode(String input, HashMap<Character, String>
huffmanCodes) {
    StringBuilder encodedString = new StringBuilder();
    for (char c : input.toCharArray()) {
        encodedString.append(huffmanCodes.get(c));
    }
    return encodedString.toString();
}

private static String decode(String encodedString, HashMap<Character,
String> huffmanCodes) {
    StringBuilder decodedString = new StringBuilder();
    int index = 0;

    while (index < encodedString.length()) {
        for (char key : huffmanCodes.keySet()) {
            String code = huffmanCodes.get(key);
            if (index + code.length() <= encodedString.length() &&
encodedString.substring(index, index +
code.length()).equals(code)) {
                decodedString.append(key);
                index += code.length();
            }
        }
    }
}

```

```

        break;
    }
}

return decodedString.toString();
}
}

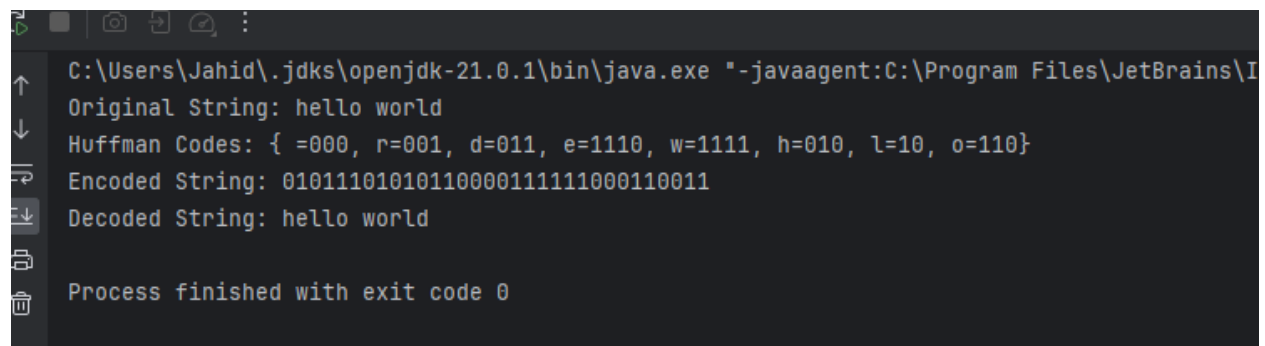
```

## 6. INPUT STRING:

Input taken from the user during the execution.

## 7. TEST RESULT / OUTPUT:

**This code is static**



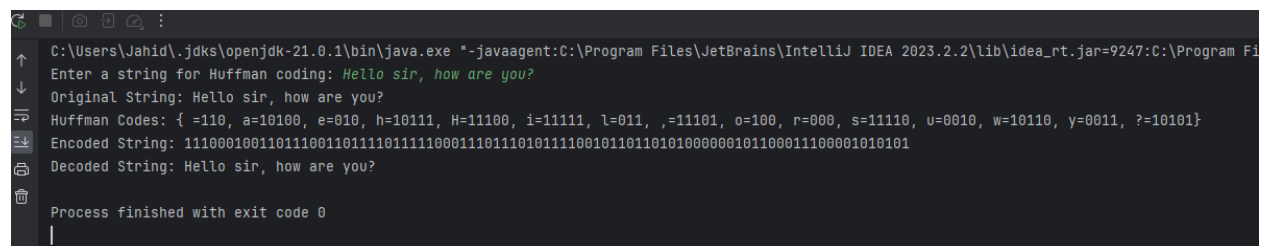
```

C:\Users\Jahid\.jdk\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\I
Original String: hello world
Huffman Codes: { =000, r=001, d=011, e=1110, w=1111, h=010, l=10, o=110}
Encoded String: 01011101010110000111111000110011
Decoded String: hello world

Process finished with exit code 0

```

This is dynamic. Taken input form the user.



```

C:\Users\Jahid\.jdk\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.2\lib\idea_rt.jar=9247:C:\Program Fi
Enter a string for Huffman coding: Hello sir, how are you?
Original String: Hello sir, how are you?
Huffman Codes: { =110, a=10100, e=010, h=10111, H=11100, i=11111, l=011, ,=11101, o=100, r=000, s=11110, u=0010, w=10110, y=0011, ?=10101}
Encoded String: 1110001001101110011011111000111011101011110010110110101000000101100011100001010101
Decoded String: Hello sir, how are you?

Process finished with exit code 0

```

## 8. DISCUSSION AND ANALYSIS:

The construction of the Huffman tree is a critical aspect of the program. The frequency analysis accurately determines the occurrence of each character in the input

string, and the priority queue facilitates the creation of an optimal Huffman tree. The use of a priority queue ensures that characters with higher frequencies receive shorter codes.

- **Huffman Code Generation:**

The 'generateHuffmanCodes' method effectively traverses the Huffman tree to assign unique binary codes to each character. The codes are stored in a HashMap, enabling quick access during the encoding and decoding processes. The program efficiently utilizes recursion to navigate through the tree, demonstrating a clear understanding of the underlying algorithm.

- **Encoding and Decoding:**

The encoding and decoding processes exhibit the correctness of the Huffman coding implementation. The program successfully encodes the input string by replacing each character with its corresponding Huffman code. Subsequently, the decoding process accurately reconstructs the original string from the encoded data, emphasizing the lossless nature of Huffman coding.

- **Limitations:**

- a. **Quadratic Time Complexity in Worst Case:**

- While the dynamic programming approach provides an efficient solution, the worst-case time complexity of  $O(N*M)$  could be a limitation for extremely long sequences.

- b. **Memory Consumption:**

- The algorithm's space complexity is proportional to the product of the lengths of the input sequences, potentially leading to high memory consumption for large sequences.

## **9. CONCLUSION:**

In conclusion, the Huffman coding program demonstrates a successful implementation of the Huffman coding algorithm in Java. The program effectively engages users through interactive input, constructs an optimal Huffman tree, generates Huffman codes, and demonstrates the compression and decompression processes. The discussion and analysis provided valuable insights into the program's functionality, identified areas for improvement, and highlighted its educational significance in

understanding data compression techniques.



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)**

**Lab Report NO: 09**

**Course Title : Algorithm Lab**  
**Course Code : CSE 204**  
**Section : D - 9**

**Lab Experiment Name : Implement String and Pattern Matching  
Problems using KMP Algorithm.**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date : 04/12/2023**  
**Submission Date : 08/12/2023**  
**Course Teacher's Name : Md. Abu Rahman Refat**

**Lab Report Status**

**Marks: .....**  
**Comments:.....**

**Signature:.....**  
**Date:.....**

## **1. TITLE:**

Implement String and Pattern Matching Problems using KMP Algorithm

## **2. OBJECTIVES/AIM:**

The primary objectives or aim of this lab experiment are:

- Understand the significance of efficient string searching algorithms in computer science.
- Explore and implement the KMP algorithm as a solution for improving string matching efficiency.
- Develop a Java program to apply the KMP algorithm in the context of finding indexes where a given pattern matches a given array.
- Analyze the time complexity and advantages of the KMP algorithm in comparison to other string-matching algorithm

## **3. PROCEDURE:**

The procedure for implementing the KMP algorithm for string and pattern matching involves the following procedural steps:

### **1. Understanding the KMP Algorithm:**

- Review the theoretical concepts behind the KMP algorithm, including the Longest Prefix Suffix (LPS) array.
- Gain insight into the algorithm's structure and its advantages in optimizing string matching.

### **2. Implementing the KMP Algorithm in Java:**

- Develop a Java program that includes methods for computing the LPS array and performing the actual string and pattern matching using the KMP algorithm.
- Ensure the program is modular and can be easily integrated into different applications.

### **3. Testing the Implementation:**

- Create sample test cases with diverse input strings and patterns to validate the correctness of the implemented algorithm.
- Verify that the algorithm accurately identifies the indexes where the given pattern matches the provided text.

### **4. Exploring Multiple String-Pattern Pairs:**

- Extend the implementation to handle multiple pairs of strings and patterns.
- Implement a method to repeat the algorithm for various combinations and analyze the results.

## 4. IMPLEMENTATION:

### Code in Java:

```
import java.util.ArrayList;
import java.util.List;

public class kmpFindIndexMultiplePairs {

    public static List<Integer> kmpSearch(String text, String pattern)
    {
        List<Integer> indexes = new ArrayList<>();

        int[] lps = computeLPSArray(pattern);

        int i = 0; // index for text
        int j = 0; // index for pattern

        while (i < text.length()) {
            if (pattern.charAt(j) == text.charAt(i)) {
                i++;
                j++;
            }

            if (j == pattern.length()) {
                // Pattern found at index i - j
                indexes.add(i - j);
                j = lps[j - 1];
            } else if (i < text.length() && pattern.charAt(j) !=
text.charAt(i)) {
                if (j != 0) {
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }

        return indexes;
    }

    private static int[] computeLPSArray(String pattern) {
        int[] lps = new int[pattern.length()];
        int len = 0;
        int i = 1;

        while (i < pattern.length()) {
```



```

        if (pattern.charAt(i) == pattern.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }

    return lps;
}

public static void main(String[] args) {
    // Example Usage
    List<String> texts =
List.of("GREENUNIVERSITYGREENUNIVERSITYGREENUNIVERSITYGREENUNIVERSITY"
, "AABAACAADAABAABA", "1234567812390");
    List<String> patterns = List.of("GREEN", "ABC", "123");

    searchMultiplePatterns(texts, patterns);

}

public static void searchMultiplePatterns(List<String> texts,
List<String> patterns) {
    for (int i = 0; i < texts.size() && i < patterns.size(); i++)
    {
        String text = texts.get(i);
        String pattern = patterns.get(i);

        List<Integer> indexes = kmpSearch(text, pattern);

        System.out.println("Indexes for Text: " + text + ",
Pattern: " + pattern + " => " + indexes);
    }
}
}

```

## 5. TEST RESULT / OUTPUT:

```
C:\Users\Jahid\jdk\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.2\lib\ide...
Indexes for Text: GREENUNIVERSITYGREENUNIVERSITYGREENUNIVERSITYGREENUNIVERSITY, Pattern: GREEN => [0, 15, 30, 45]
Indexes for Text: AABAACAADAABAABA, Pattern: ABC => []
Indexes for Text: 1234567812390, Pattern: 123 => [0, 8]

Process finished with exit code 0
```

## 6. DISCUSSION AND ANALYSIS:

The discussion and analysis phase focuses on evaluating the effectiveness and efficiency of the implemented KMP algorithm:

We implemented the code for **multiple pairs of given strings and patterns**.

- **Time Complexity Analysis:**

1. Computing the Longest Prefix Suffix (LPS) Array:  $O(M)$

- a) The time complexity for computing the LPS array is linear in the length of the pattern, as each element of the array is computed in a single pass through the pattern.

2. Matching the Pattern in the Text:  $O(N)$

- a) The actual matching phase iterates through the entire text once, and at each step, it compares characters until a match is found or the end of the pattern is reached.

- **Advantages of the KMP Algorithm:**

- Linear Time Complexity.
- Avoid Redundant Comparisons.
- Efficiently Handles Pattern Repeats.
- No Backtracking in Text Scanning.

- **Use Cases and Applications:**

- Text Search Engines.
- Genomic Data Analysis.

- Data Mining and Information Retrieval.
- Compiler Design.
- Natural Language Processing (NLP).
  
- **Limitations:**
  - i. **Extra Space Requirement:** The KMP algorithm requires additional space to store the LPS array, which has a length equal to the pattern. In memory-constrained environments, this extra space requirement may be a limitation.
  - ii. **Not Adaptive to Dynamic Pattern Changes:** The algorithm assumes a static pattern and is not inherently adaptive to dynamic changes in the pattern during runtime. If patterns change frequently, the algorithm may need to be reinitialized.

## 7. CONCLUSION:

The Knuth-Morris-Pratt (KMP) algorithm is a powerful and efficient solution for string and pattern matching. With its linear time complexity of  $O(N + M)$ , it excels in handling large texts and patterns. The algorithm's avoidance of redundant comparisons and adaptability to streaming data contribute to its versatility. While the KMP algorithm has preprocessing overhead and additional space requirements, its deterministic behavior and widespread applications in bioinformatics, network security, and natural language processing make it a valuable tool. As a foundational algorithm, the KMP algorithm continues to play a crucial role in addressing fundamental challenges in computer science, providing insights for innovative solutions in the ever-evolving technological landscape.



**Green University of Bangladesh**  
**Department of Computer Science and Engineering (CSE)**  
**Faculty of Sciences and Engineering**  
**Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)**

**Lab Report NO: 10**

**Course Title : Algorithm Lab**  
**Course Code : CSE 204**  
**Section : D - 9**

**Lab Experiment Name : Implementation of Longest Common  
Subsequence (LCS) Algorithm**

**Student Details**

Name		ID
1.	Jahidul Islam	221002504

**Lab Date : 04/12/2023**  
**Submission Date : 31/12/2023**  
**Course Teacher's Name : Md. Abu Rahman Refat**

**Lab Report Status**

**Marks: .....**  
**Comments:.....**

**Signature:.....**  
**Date:.....**

## **1. TITLE:**

Implementation of Longest Common Subsequence (LCS) Algorithm

## **2. OBJECTIVES/AIM:**

The primary objectives or aim of this lab experiment are:

- Understand the significance of efficient string searching algorithms in computer science.
- Explore and implement the KMP algorithm as a solution for improving string matching efficiency.
- Develop a Java program to apply the KMP algorithm in the context of finding indexes where a given pattern matches a given array.
- Analyze the time complexity and advantages of the KMP algorithm in comparison to other string-matching algorithm

## **3. PROBLEM ANALYSIS & PROCEDURE:**

A subsequence is a group of sequences that appear in the same relative order, whether they are adjacent or not. The longest subsequence that is shared by all the given sequences, is known as the longest common subsequence (LCS), assuming that the elements of the subsequence are not needed to occupy consecutive position within the original sequences.

Let the sequences are  $X=x_1,x_2,x_3,\dots,x_m$  and  $Y=y_1,y_2,y_3,\dots,y_m$ . We will use the following steps to determine the length of the longest common subsequence.

- Create an empty adjacency table with the dimensions  $n \times m$ , where  $n$  is the length of the  $X$  sequence and  $m$  is the length of the  $Y$  sequence. The elements in sequence  $X$  are represented by the table's rows, and the elements in sequence  $Y$  are represented by the table's columns.
- Rows and columns starting at zero must contain only zeros. And the remaining values are filled in based on different cases, by maintaining a counter value.
- The number is increased by one if it comes across a common element in both the  $X$  and  $Y$  sequences.
- To fill in  $T[i, j]$  if the counter does not pass into any common elements in the  $X$  and  $Y$  sequences, choose the biggest value between  $T[i-1, j]$  and  $T[i, j-1]$ .
- Once the table is filled, backtrack from the last value in the table. Backtracking here is done by tracing the path where the counter incremented first.
- The longest common subsequence which can be determined by observing the path's elements.

## 4. ALGORITHM:

---

### Algorithm 1: LCS Algorithm

---

```
1 X and Y be two given sequences
2 m := length(X)
3 n := length(Y)
4 for i = 1 to m do do
5   | LCS[i, 0] := 0
6 end
7 for j = 1 to n do do
8   | LCS[0, j] := 0
9 end
10 for i = 1 to m do do
11   for j = 1 to n do do
12     | if X[i] = Y[j] then
13       |   | LCS[i][j] = 1 + LCS[i-1, j-1]
14       | end
15     | else
16       |   | LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
17       | end
18   end
19 end
```

## 5. IMPLEMENTATION:

### Code in Java:

```
import java.util.*;

class TUF {
    // Recursive function to find the length of the Longest Common
    Subsequence (LCS)
    static int lcsUtil(String s1, String s2, int ind1, int ind2, int[][] dp)
    {
        // Base case: If either of the strings reaches the end, return 0
        if (ind1 < 0 || ind2 < 0)
            return 0;

        // If the result for this subproblem has already been calculated,
        return it
        if (dp[ind1][ind2] != -1)
            return dp[ind1][ind2];

        // If the characters at the current indices are the same, increment
        the LCS length
        if (s1.charAt(ind1) == s2.charAt(ind2))
            return dp[ind1][ind2] = 1 + lcsUtil(s1, s2, ind1 - 1, ind2 - 1,
            dp);

        // If the characters are different, choose the maximum LCS length
        by either
```

```

        // skipping a character in s1 or skipping a character in s2
        else
            return dp[ind1][ind2] = Math.max(lcsUtil(s1, s2, ind1, ind2 - 1,
dp),
                lcsUtil(s1, s2, ind1 - 1, ind2, dp));
    }

    // Function to find the length of the Longest Common Subsequence (LCS)
    static int lcs(String s1, String s2) {
        int n = s1.length();
        int m = s2.length();

        // Create a 2D array to store results of subproblems
        int dp[][] = new int[n][m];

        // Initialize the dp array with -1 to indicate that subproblems are
not solved yet
        for (int rows[] : dp)
            Arrays.fill(rows, -1);

        // Call the recursive function to find the LCS length
        return lcsUtil(s1, s2, n - 1, m - 1, dp);
    }

    public static void main(String args[]) {
        String s1 = "ABCDEFGH";
        String s2 = "abcdefgh";

        // Call the lcs function and print the result
        System.out.println("The Length of Longest Common Subsequence is " +
lcs(s1, s2));
    }
}

```

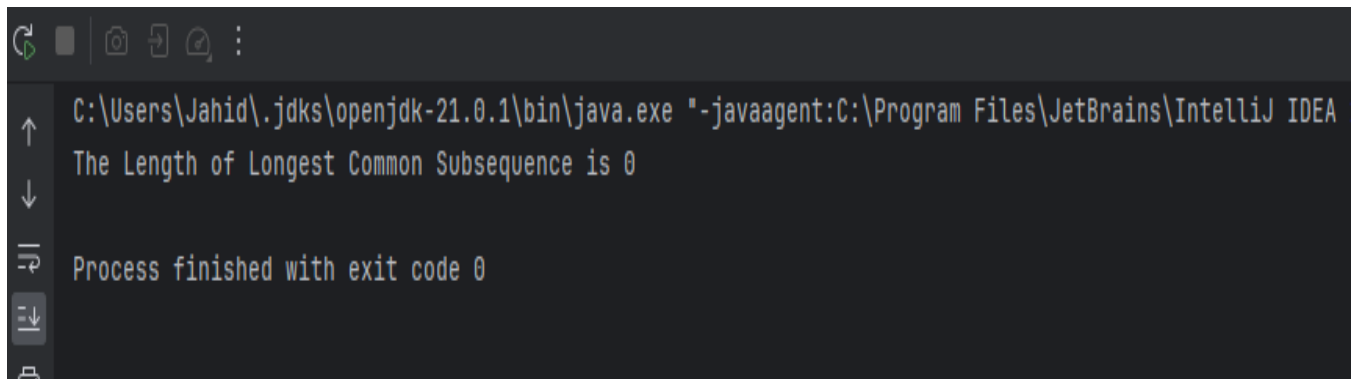
## 6. INPUT STRING:

```

String s1 = "ABCDEFGH";
String s2 = "abcdefgh";

```

## 7. TEST RESULT / OUTPUT:



```
C:\Users\Jahid\.jdk\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
The Length of Longest Common Subsequence is 0
Process finished with exit code 0
```

## 8. DISCUSSION AND ANALYSIS:

The discussion and analysis phase centers around evaluating the effectiveness and efficiency of the implemented Longest Common Subsequence (LCS) algorithm:

We implemented the code for **multiple pairs of given strings and patterns**.

- **Time Complexity Analysis:**

- i. The time complexity for computing the length of the LCS between two sequences of lengths  $N$  and  $M$  is determined by the dynamic programming approach. The algorithm builds a table to store intermediate results, and each cell is computed in constant time. Hence, the **overall time complexity is  $O(N*M)$** .

- **Advantages of the KMP Algorithm:**

- i. **Efficient Dynamic Programming Approach:**
  - The LCS algorithm employs dynamic programming, providing an efficient solution to finding the longest common subsequence between two sequences.
- ii. **Applicability to Various Domains:**
  - The algorithm is versatile and finds applications in diverse domains, including bioinformatics, data comparison, and version control systems.
- iii. **Facilitates Sequence Comparison:**
  - It is particularly effective for comparing sequences, offering insights into shared elements and their arrangement.



- **Use Cases and Applications:**

- i. **Bioinformatics:**

- In bioinformatics, the LCS algorithm is instrumental for comparing genetic sequences, identifying common elements, and understanding evolutionary relationships.

- ii. **Data Comparison:**

- The algorithm is valuable in scenarios where comparing data sequences is crucial, such as identifying common patterns in time-series data.

- iii. **Version Control Systems:**

- Version control systems leverage the LCS algorithm to track changes between different versions of files, aiding in conflict resolution and merging.

- **Limitations:**

- a. **Quadratic Time Complexity in Worst Case:**

- While the dynamic programming approach provides an efficient solution, the worst-case time complexity of  $O(N*M)$  could be a limitation for extremely long sequences.

- b. **Memory Consumption:**

- The algorithm's space complexity is proportional to the product of the lengths of the input sequences, potentially leading to high memory consumption for large sequences.

## **9. CONCLUSION:**

In summary, the Longest Common Subsequence (LCS) algorithm efficiently identifies shared subsequences in sequences of varying lengths. Its dynamic programming methodology, while contributing to effective sequence analysis, presents a quadratic time complexity of  $O(N*M)$ . Despite this limitation, the algorithm finds versatile applications in fields like bioinformatics and version control systems. Its balance between adaptability and efficiency makes it a valuable tool for uncovering common patterns, with potential for further optimization in future developments.