LAB 5

Kruskal’s Algorithm:

Kruskal’s algorithm is a greedy approach for finding the Minimum Spanning Tree (MST) by selecting edges in ascending order of weight, ensuring no cycles form. It efficiently uses the Disjoint Set (Union-Find) data structure to manage connected components.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct Edge {
    int start, end, weight;
};

bool compareEdges(Edge a, Edge b) {
    return a.weight < b.weight;
}

class DisjointSet {
    vector<int> parent, rank;
public:
    DisjointSet(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    int findParent(int node) {
        if (parent[node] != node)
            parent[node] = findParent(parent[node]);
        return parent[node];
    }

    void unionSets(int node1, int node2) {
        int root1 = findParent(node1);
        int root2 = findParent(node2);
        if (root1 != root2) {
            if (rank[root1] > rank[root2])
                parent[root2] = root1;
            else if (rank[root1] < rank[root2])
                parent[root1] = root2;
            else {
                parent[root2] = root1;
                rank[root1]++;
            }
        }
    }
};

int kruskalAlgorithm(int totalNodes, vector<Edge> &edges) {
    sort(edges.begin(), edges.end(), compareEdges);
    DisjointSet ds(totalNodes);
    int totalWeight = 0, selectedEdges = 0;

    for (Edge edge : edges) {
        if (ds.findParent(edge.start) != ds.findParent(edge.end)) {
            ds.unionSets(edge.start, edge.end);
            totalWeight += edge.weight;
            selectedEdges++;
            if (selectedEdges == totalNodes - 1) break;
        }
    }
    return totalWeight;
}

int main() {
    int totalNodes = 4;
    vector<Edge> edges = {
        {0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}
    };
```

```cpp

    cout << "Minimum Spanning Tree Cost: " << kruskalAlgorithm(totalNodes, edges) << endl;
    return 0;
```

Prim's Algorithm:

Prim's algorithm starts from any node and **grows the MST step by step** by adding the smallest available edge connecting a new vertex. It uses a **priority queue (min-heap)** for efficient edge selection, making it preferable for **dense graphs**.

```cpp
#include <iostream>
#include <vector>

using namespace std;

#define V 5
#define INF 99999

int findMin(vector<int>& key, vector<bool>& visited) {
    int min = INF, index = -1;
    for (int i = 0; i < V; i++) {
        if (!visited[i] && key[i] < min) {
            min = key[i];
            index = i;
        }
    }
    return index;
}

void primMST(vector<vector<int>>& graph) {
    vector<int> parent(V, -1);
    vector<int> key(V, INF);
    vector<bool> visited(V, false);

    key[0] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = findMin(key, visited);
        visited[u] = true;

        for (int v = 0; v < V; v++) {
            if (graph[u][v] && !visited[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " : " << graph[i][parent[i]] << "\n";
}

int main() {
    vector<vector<int>> graph = {
        {2, 2, 0, 6, 8},
        {2, 5, 3, 8, 5},
        {0, 3, 8, 0, 7},
        {6, 8, 9, 9, 9},
        {0, 5, 7, 9, 0}
    };

    primMST(graph);

    return 0;
}



```

```
 } ; if ($?) { .\tempCodeRunnerFile }
0 - 1 : 2
1 - 2 : 3
0 - 3 : 6
1 - 4 : 5
PS C:\Users\mohit\Desktop\STUDY\DAA LAB\27>
```

LEETCODE 1

The problem **"Longest Substring Without Repeating Characters"** requires finding the longest contiguous substring in a given string s where no character appears more than once.

For example:

- **Input:** "abcabcbb"
- **Output:** 3 (Longest substring: "abc")

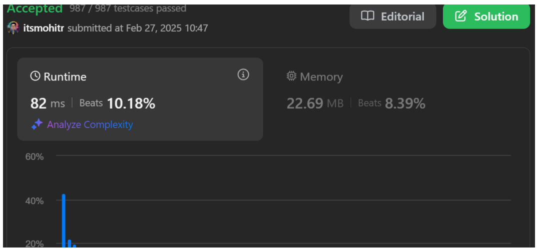
A common approach to solve this is using the **Sliding Window** technique with a **HashSet or HashMap** to track seen characters efficiently. The optimal solution runs in **O(n) time complexity**.

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
       int maxLength = 0;
        int n = s.length();

        for (int i = 0; i < n; i++) {
            string str = "";
            for (int j = i; j < n; j++) {
                if (str.find(s[j]) != string::npos) {
                    break;
                }
                str += s[j];
                maxLength = max(maxLength, (int)str.length());
            }
        }
        return maxLength;



}
```



LEETCODE 2

The problem "Same Tree" requires checking whether two binary trees p and q are identical.

Conditions for Two Trees to be the Same:

1. **Both trees must have the same structure.**
2. **Corresponding nodes must have the same values.**

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (p == q) return true;
        if (!p || !q || p->val != q->val) return false;
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```