RAMANUJAN COLLEGE

University Of Delhi

ALGORITHM AND ADVANCE DATA STRUCTURES

ASSIGNMENT

Submitted to:- Dr.Ashmita Patel Mam

Name:- Ravikant Maurya

College Roll No.:- 20221433

Exam Roll No.:- 20220570026

Course:- B.Sc.(Hons.)Computer Science

Q.1:- Write a program to sort the elements of an array using Randomized Quick Sort (the program should report the number of comparisons).

Source code

Screenshot of the input and sorted array

Number of comparisons reported

Explanation:-

1. Libraries and Namespace

- **`#include <iostream>`**: Provides functionality for input and output (e.g., `cin` and `cout`).
- **`#include <vector>`**: Allows the use of the dynamic array-like structure `std::vector`.
- **`#include <cstdlib>`**: Includes functions like `rand()` for generating random numbers.
- **`#include <ctime>`**: Provides functions like `time()` for random seed generation.
- **`using namespace std;`**: Avoids prefixing `std::` to standard library names (e.g., `std::cout` becomes `cout`).

2. Swap Function

- **Purpose**: Swaps the values of two integers using their references.
- **How it works**:
  - Temporary storage (`temp`) holds the value of one variable (`a`).
  - Values are exchanged between `a` and `b`.
  - Ensures no additional memory allocation, providing an efficient way to swap.

3. Randomized Partition Function

- **Purpose**: Divides the array into two parts around a randomly chosen pivot.
- **Key Steps**:
  1. **Random Pivot Selection**:
     - A random pivot index is calculated using `low + rand() % (high - low + 1)`.
     - The pivot element is swapped to the end of the array for ease of partitioning.
  2. **Partitioning**:
     - The pivot element is compared with each element in the array.
     - Elements smaller than or equal to the pivot are moved to the left.
     - Larger elements remain on the right.

        - A `comparisonCount` variable tracks the number of comparisons made.
    3. **Final Placement of Pivot**:
        - The pivot is placed at its correct position in the sorted array.

4. Randomized QuickSort Function

- **Purpose**: Sorts the array recursively using the divide-and-conquer approach.
- **Key Steps**:
    1. **Base Case**:
        - If the subarray size is 0 or 1 (`low >= high`), no sorting is needed.
    2. **Partitioning**:
        - The array is partitioned around a random pivot using `randomizedPartition`.
    3. **Recursive Calls**:
        - The subarray to the left of the pivot is sorted recursively.
        - The subarray to the right of the pivot is sorted recursively.

5. Main Function

- **Purpose**: Serves as the entry point for the program. Handles user input, sorting, and output.

Key Steps:

1. **Input**:
    - Prompts the user to enter the number of elements (`n`).
    - Takes input for the array elements.
2. **Initialization**:
    - Seeds the random number generator using `srand(time(0))` to ensure different pivot selections for each execution.
    - Initializes the `comparisonCount` variable to track the number of comparisons made during sorting.
3. **Sorting**:
    - Calls `randomizedQuickSort` to sort the array.
4. **Output**:
    - Displays the sorted array.
    - Prints the total number of comparisons performed during the sort.

6. Key Features

- **Random Pivot**: Improves performance on sorted or nearly sorted arrays, reducing the chances of worst-case complexity.
- **Comparison Counter**: Provides insight into the algorithm's performance for different inputs.
- **Recursion**: Makes the code cleaner and easier to understand.

7. Complexity

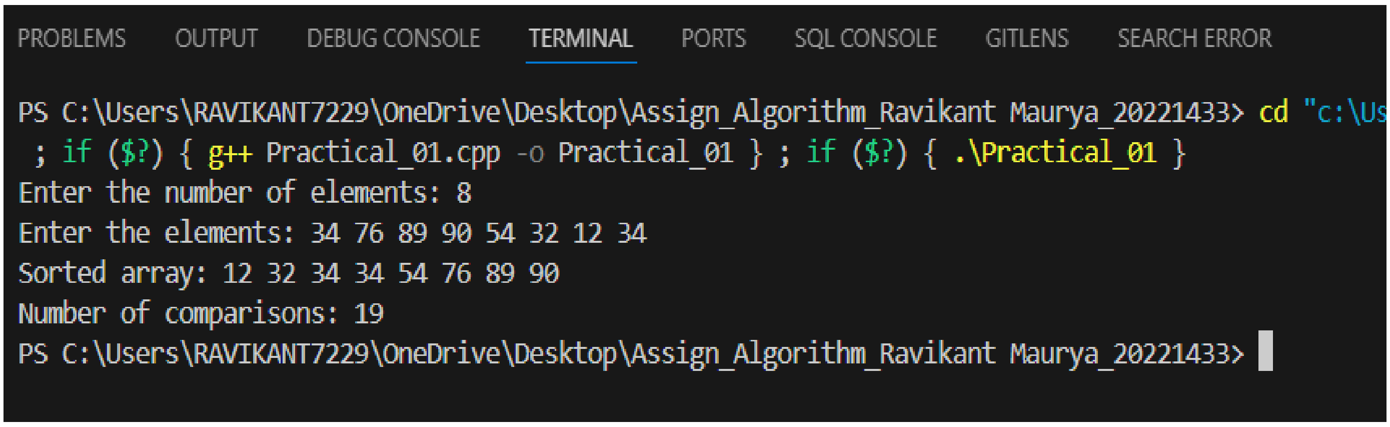
- **Time Complexity**:
    - Best/Average Case: O(nlog⁡n)O(n \log n)O(nlogn) (balanced partitions).

   - Worst Case: O(n2)O(n^2)O(n2) (highly unbalanced partitions, though randomized pivot reduces this likelihood).
- **Space Complexity**:
   - O(log⁡n)O(\log n)O(logn) for recursion stack in the best case, O(n)O(n)O(n) in the worst case.

Code:-

```cpp
#include <iostream>
#include <vector>
#include <cstdlib> // for rand()
#include <ctime>   // for srand()

using namespace std;

// Function to swap two elements in the array
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Partition function with a randomized pivot
// It rearranges the array such that elements smaller than the pivot
// are on the left, and elements greater than the pivot are on the right
int randomizedPartition(vector<int> &arr, int low, int high, int &comparisonCount) {
    // Select a random pivot index within the range [low, high]
    int pivotIndex = low + rand() % (high - low + 1);
    swap(arr[pivotIndex], arr[high]); // Place the random pivot at the end for convenience

    int pivot = arr[high]; // The pivot element
    int i = low - 1; // Index of the smaller element

    // Traverse the array and compare each element with the pivot
    for (int j = low; j < high; ++j) {
        comparisonCount++; // Increment the comparison counter
        if (arr[j] <= pivot) {
            i++; // Move the boundary of smaller elements
            swap(arr[i], arr[j]); // Place the smaller element in the correct position
        }
    }
    // Place the pivot in its correct sorted position
    swap(arr[i + 1], arr[high]);
    return i + 1; // Return the index of the pivot
}

// Randomized Quick Sort
// A divide-and-conquer algorithm that recursively sorts subarrays
void randomizedQuickSort(vector<int> &arr, int low, int high, int &comparisonCount) {
    if (low < high) { // Base condition for recursion
        // Partition the array and get the pivot index
        int pivotIndex = randomizedPartition(arr, low, high, comparisonCount);

        // Recursively sort the subarray before the pivot
        randomizedQuickSort(arr, low, pivotIndex - 1, comparisonCount);

        // Recursively sort the subarray after the pivot
        randomizedQuickSort(arr, pivotIndex + 1, high, comparisonCount);
    }
}

int main() {
    srand(time(0)); // Seed the random number generator with the current time

    int n; // Number of elements in the array
    cout << "Enter the number of elements: ";
    cin >> n;

    vector<int> arr(n); // Initialize a vector to hold the elements
    cout << "Enter the elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i]; // Input the elements
    }

    int comparisonCount = 0; // Initialize the comparison counter

    // Sort the array using randomized quicksort
    randomizedQuickSort(arr, 0, n - 1, comparisonCount);

    // Output the sorted array
    cout << "Sorted array: ";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " "; // Print each element of the sorted array
    }
    cout << endl;

    // Output the number of comparisons made during sorting
    cout << "Number of comparisons: " << comparisonCount << endl;

    return 0; // Indicate successful program termination
}
```

Output:-



```
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> cd "c:\Us
 ; if ($?) { g++ Practical_01.cpp -o Practical_01 } ; if ($?) { .\Practical_01 }
Enter the number of elements: 8
Enter the elements: 34 76 89 90 54 32 12 34
Sorted array: 12 32 34 34 54 76 89 90
Number of comparisons: 19
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433>
```

Q.2:- Write a program to find the ith smallest element of an array using Randomized Select.

Explanation:-

1.Swap Function

1. **Purpose**: Swaps the values

- of two integers using their references.
- **How It Works**:
  - A temporary variable stores one value while the other value is assigned.
  - The stored value is then assigned to the other variable.

2. Randomized Partition Function

- **Purpose**: Partitions the array into two parts based on a randomly chosen pivot.
- **Steps**:
  1. Selects a random pivot index and swaps it with the last element.
  2. Iterates through the array:
     - Elements smaller than or equal to the pivot are moved to the left.
     - Larger elements remain on the right.
  3. Places the pivot in its correct position.
  4. Returns the pivot index.

3. Randomized Select Function

- **Purpose**: Efficiently finds the i-th smallest element in the array.
- **Steps**:
  1. **Base Case**:
     - If there's only one element in the array, return it.
  2. **Partitioning**:
     - Divides the array into two parts using `randomizedPartition`.
     - Calculates the position (`k`) of the pivot in the sorted array.
  3. **Recursive Search**:
     - If `i == k`, the pivot is the desired element.
     - If `i < k`, search in the left subarray.
     - If `i > k`, search in the right subarray, adjusting `i` to reflect the new range.

4. Main Function

- **Purpose**: Handles user input/output and calls the Randomized Select function.
- **Key Steps**:
  1. Reads the number of elements and their values.
  2. Takes the 1-based index `i` as input and validates it.
  3. Calls `randomizedSelect` to find the i-th smallest element.
  4. Outputs the result.

Key Features:

- **Random Pivot Selection**: Improves performance by reducing the likelihood of worst-case complexity.
- **Efficient Search**: Avoids fully sorting the array, leading to better performance for finding a single element.
- **1-Based Indexing**: Aligns with human-readable indexing (starting from 1).

Complexity:

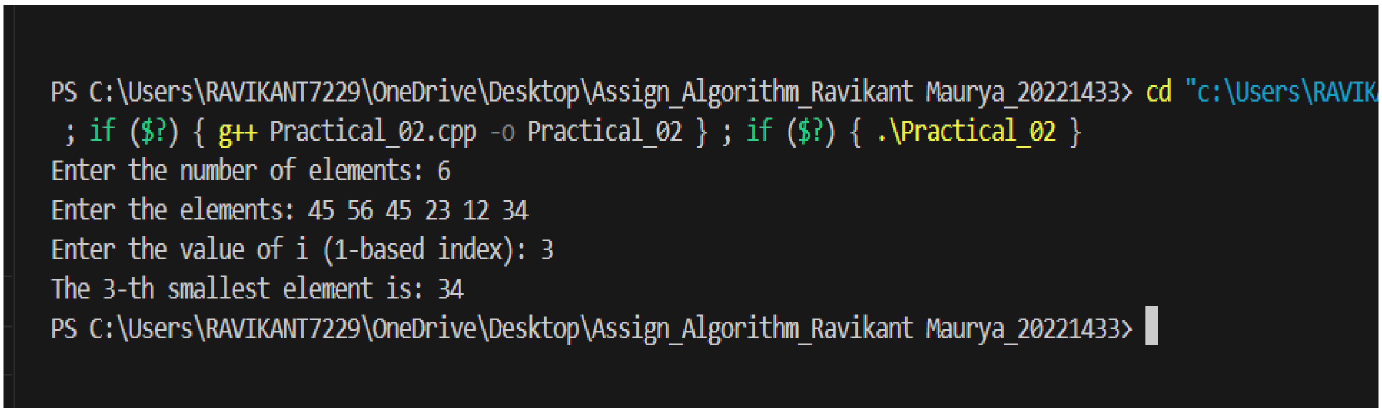
- **Time Complexity**:
  - Average Case: O(n)O(n)O(n)
  - Worst Case: O(n2)O(n^2)O(n2) (highly unlikely due to random pivot selection)
- **Space Complexity**:
  - O(log n)O(\log n)O(logn) for recursion stack in the average case.

Code:-

```cpp
#include <iostream>
#include <vector>
#include <cstdlib> // for rand()
#include <ctime>   // for srand()

using namespace std;

// Function to swap two elements in the array
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Partition function with a randomized pivot
// Rearranges the array such that elements smaller than the pivot are on the left,
// and elements greater than the pivot are on the right.
int randomizedPartition(vector<int> &arr, int low, int high) {
    // Select a random pivot index between low and high
    int pivotIndex = low + rand() % (high - low + 1);
    swap(arr[pivotIndex], arr[high]); // Place the random pivot at the end for convenience

    int pivot = arr[high]; // The pivot element
    int i = low - 1; // Index of the smaller element

    // Traverse the array and compare each element with the pivot
    for (int j = low; j < high; ++j) {
        if (arr[j] <= pivot) { // If the current element is smaller or equal to the pivot
            i++; // Increment the index of the smaller element
            swap(arr[i], arr[j]); // Swap the smaller element into its correct position
        }
    }
    // Place the pivot in its correct sorted position
    swap(arr[i + 1], arr[high]);
    return i + 1; // Return the index of the pivot
}

// Randomized Select algorithm
// Finds the i-th smallest element in the array without fully sorting it
int randomizedSelect(vector<int> &arr, int low, int high, int i) {
    if (low == high) { // Base case: Only one element in the array
        return arr[low];
    }

    // Partition the array and get the pivot index
    int pivotIndex = randomizedPartition(arr, low, high);
    int k = pivotIndex - low + 1; // The position of the pivot in the sorted array

    // Check the position of the pivot relative to the desired element
    if (i == k) {
        return arr[pivotIndex]; // The pivot is the i-th smallest element
    } else if (i < k) {
        // If the desired element is in the left part, search in the left subarray
        return randomizedSelect(arr, low, pivotIndex - 1, i);
    } else {
        // If the desired element is in the right part, search in the right subarray
        // Adjust i to reflect the new subarray's context
        return randomizedSelect(arr, pivotIndex + 1, high, i - k);
    }
}

int main() {
    srand(time(0)); // Seed the random number generator with the current time

    int n; // Number of elements in the array
    cout << "Enter the number of elements: ";
    cin >> n;

    vector<int> arr(n); // Array to store the elements
    cout << "Enter the elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i]; // Input the elements
    }

    int i; // Position of the element to find (1-based index)
    cout << "Enter the value of i (1-based index): ";
    cin >> i;

    // Check if the input index is valid
    if (i < 1 || i > n) {
        cout << "Invalid value of i. Must be between 1 and " << n << "." << endl;
        return 1; // Exit the program if the input is invalid
    }

    // Find the i-th smallest element using Randomized Select
    int ithSmallest = randomizedSelect(arr, 0, n - 1, i);
    cout << "The " << i << "-th smallest element is: " << ithSmallest << endl;

    return 0; // Successful program termination
```

```
}
```

Output:-



```
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> cd "c:\Users\RAVIK
 ; if ($?) { g++ Practical_02.cpp -o Practical_02 } ; if ($?) { .\Practical_02 }
Enter the number of elements: 6
Enter the elements: 45 56 45 23 12 34
Enter the value of i (1-based index): 3
The 3-th smallest element is: 34
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433>
```

Q.3:- Write a program to determine the minimum spanning tree of a graph using Kruskal’s algorithm.

Explanation:-

1. Edge Structure

- Represents a single edge in the graph with:
    - `src`: Source vertex.
    - `dest`: Destination vertex.
    - `weight`: Weight of the edge.

2. Compare Function

- Compares two edges based on their weight.
- Used to sort edges in ascending order for Kruskal's algorithm.

3. Disjoint Set Data Structure

- **Purpose**: Efficiently handles connected components of the graph.
- **Functions**:
    - `find(u)`: Determines the root of the set containing vertex `u` with path compression.
    - `unionSets(u, v)`: Merges two sets containing `u` and `v` based on rank optimization.

4. Kruskal's Algorithm

- **Steps**:
    1. **Sort Edges**: Sort all edges in non-decreasing order of weight.
    2. **Process Edges**:
        - Iterate over the sorted edges.
        - Use the disjoint set to check if the current edge forms a cycle.

        - If no cycle is formed, include the edge in the MST and union the sets of its vertices.
    3. **Output MST**:
        - Print all edges in the MST.
        - Print the total weight of the MST.

5. Main Function

- Takes input for the graph (number of vertices, edges, and edge details).
- Calls `kruskalMST` to compute and display the MST.

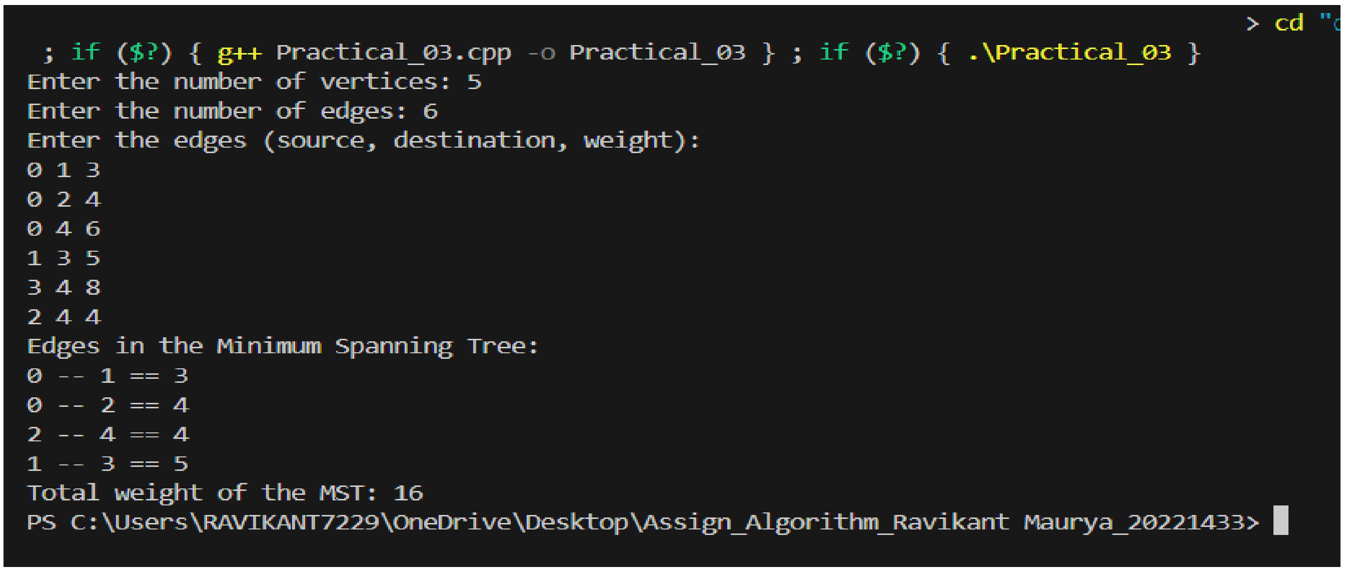
Complexity:

- **Sorting Edges**: O(Elog⁡E)O(E \log E)O(ElogE), where EEE is the number of edges.
- **Union-Find Operations**: Near constant time, O(α(V))O(\alpha(V))O(α(V)), for VVV vertices.
- **Total Complexity**: O(Elog⁡E)O(E \log E)O(ElogE).

Code:-

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for sort()

using namespace std;

// Structure to represent an edge in the graph
struct Edge {
    int src, dest, weight; // Source vertex, destination vertex, and weight of the edge
};

// Function to compare two edges based on their weights
// Used for sorting edges in ascending order of weight
bool compareEdges(const Edge &e1, const Edge &e2) {
    return e1.weight < e2.weight;
}

// Disjoint Set (Union-Find) Structure
// Helps efficiently manage connected components in the graph
class DisjointSet {
private:
    vector<int> parent, rank; // `parent` stores the parent of each node, `rank` is used to
optimize unions

public:
    // Constructor: initialize parent and rank arrays
    DisjointSet(int n) {
        parent.resize(n);
        rank.resize(n, 0); // Initially, all ranks are 0
        for (int i = 0; i < n; ++i) {
            parent[i] = i; // Each node is its own parent initially
        }
    }
    // Find the root of the set containing 'u' with path compression
    int find(int u) {
        if (u != parent[u]) { // If 'u' is not its own parent, recursively find its root
            parent[u] = find(parent[u]); // Path compression: flatten the tree
        }
        return parent[u];
    }

    // Union two sets based on rank
    void unionSets(int u, int v) {
        int rootU = find(u); // Find the root of the set containing 'u'
        int rootV = find(v); // Find the root of the set containing 'v'

        if (rootU != rootV) { // Only union if they are in different sets
            if (rank[rootU] < rank[rootV]) { // Attach the smaller tree under the larger tree
                parent[rootU] = rootV;
            } else if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else { // If ranks are equal, arbitrarily choose one as root and increment its
rank
                parent[rootV] = rootU;
                rank[rootU]++;
            }
        }
    }
};

// Kruskal's algorithm to find the Minimum Spanning Tree (MST)
void kruskalMST(int V, vector<Edge> &edges) {
    // Sort all edges in non-decreasing order of their weight
    sort(edges.begin(), edges.end(), compareEdges);

    DisjointSet ds(V); // Initialize disjoint set for the vertices

    vector<Edge> mst; // To store the edges included in the MST
    int mstWeight = 0; // To store the total weight of the MST

    for (const auto &edge : edges) {
        // Find the roots of the source and destination vertices of the current edge
        int rootSrc = ds.find(edge.src);
        int rootDest = ds.find(edge.dest);

        // If including this edge doesn't form a cycle
        if (rootSrc != rootDest) {
            mst.push_back(edge); // Include the edge in the MST
            mstWeight += edge.weight; // Add the weight of the edge to the total MST weight
            ds.unionSets(rootSrc, rootDest); // Union the two sets
        }
    }

    // Print the edges of the MST
    cout << "Edges in the Minimum Spanning Tree:\n";
    for (const auto &edge : mst) {
        cout << edge.src << " -- " << edge.dest << " == " << edge.weight << endl;
    }
    // Print the total weight of the MST
    cout << "Total weight of the MST: " << mstWeight << endl;
}

int main() {
    int V, E; // Number of vertices and edges in the graph
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    vector<Edge> edges(E); // Array to store the edges
    cout << "Enter the edges (source, destination, weight):\n";
    for (int i = 0; i < E; ++i) {
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight; // Input each edge
    }

    // Find and print the Minimum Spanning Tree
    kruskalMST(V, edges);

    return 0; // Program ends successfully
}
```

<u>***Output:-***</u>



```
                                                                                    > cd "c
 ; if ($?) { g++ Practical_03.cpp -o Practical_03 } ; if ($?) { .\Practical_03 }
Enter the number of vertices: 5
Enter the number of edges: 6
Enter the edges (source, destination, weight):
0 1 3
0 2 4
0 4 6
1 3 5
3 4 8
2 4 4
Edges in the Minimum Spanning Tree:
0 -- 1 == 3
0 -- 2 == 4
2 -- 4 == 4
1 -- 3 == 5
Total weight of the MST: 16
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433>
```

Q.4:- Write a program to implement the Bellman-Ford algorithm to find the shortest paths from a given source node to all other nodes in a graph.

<u>***Explanation:-***</u>

1. Data Structures

- **`Edge` Structure**:
    - Represents an edge with:
        - `src`: Source vertex.
        - `dest`: Destination vertex.
        - `weight`: Weight of the edge.
- **`distance` Vector**:
    - Stores the shortest distances from the source to each vertex.
    - Initialized to `INT_MAX` (infinity) to represent unreachable vertices.

2. Algorithm

- **Relaxation Phase**:
    - For each edge, update the distance to its destination vertex if:
        - The source vertex is reachable.
        - The new path through the edge provides a shorter distance.
    - Repeat this for ∣V∣−1|V| - 1∣V∣−1 iterations, as the longest possible shortest path in a graph has ∣V∣−1|V| - 1∣V∣−1 edges.
- **Negative-Weight Cycle Detection**:
    - After ∣V∣−1|V| - 1∣V∣−1 iterations, check all edges again.
    - If any edge can still be relaxed (i.e., a shorter path is found), then the graph contains a negative-weight cycle.

3. Input/Output

- **Input**:
    - `V`: Number of vertices.
    - `E`: Number of edges.
    - Edges with `source`, `destination`, and `weight`.
    - Source vertex to calculate distances from.
- **Output**:
    - If a negative-weight cycle is detected, print a message and terminate.
    - Otherwise, print the shortest distance to each vertex.

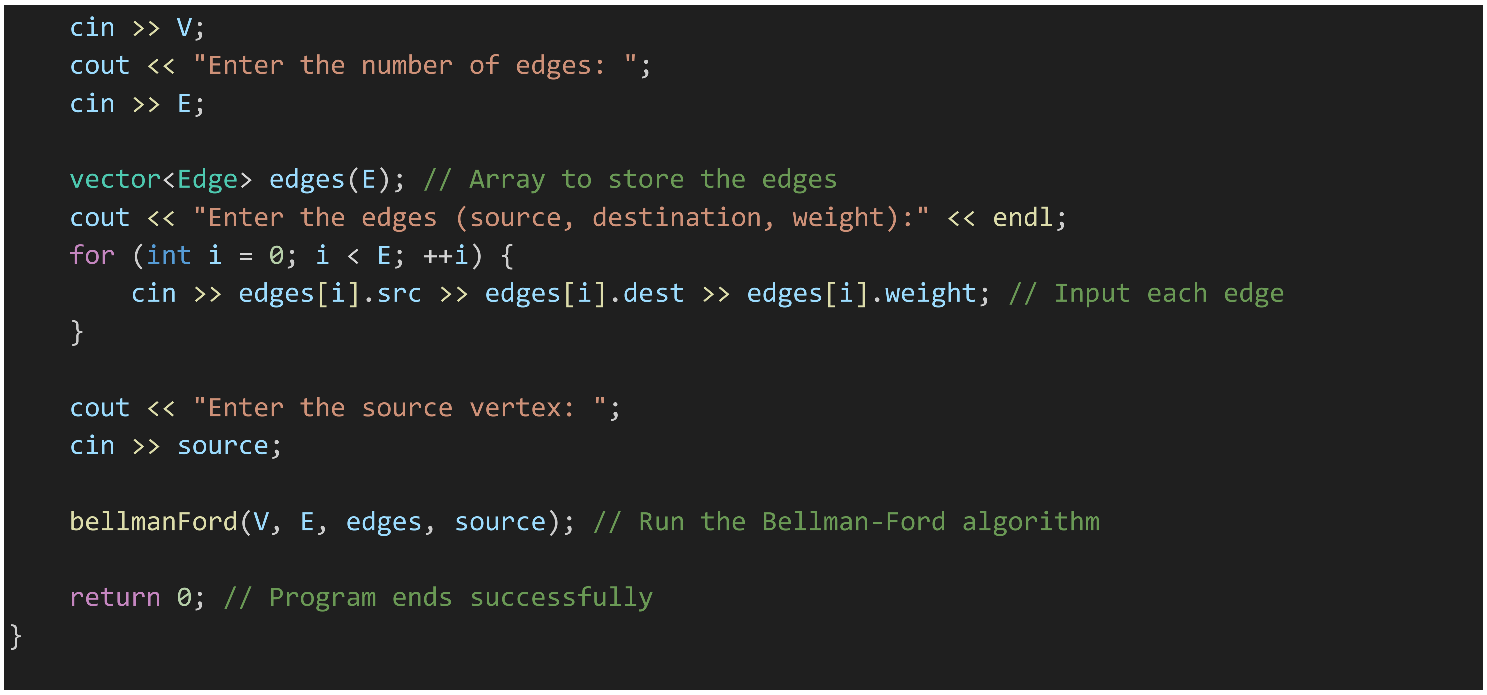
Complexity

- **Time Complexity**:
    - O(V⋅E)O(V \cdot E)O(V⋅E), where VVV is the number of vertices and EEE is the number of edges.
- **Space Complexity**:
    - O(V)O(V)O(V) for the `distance` array.

Code:-

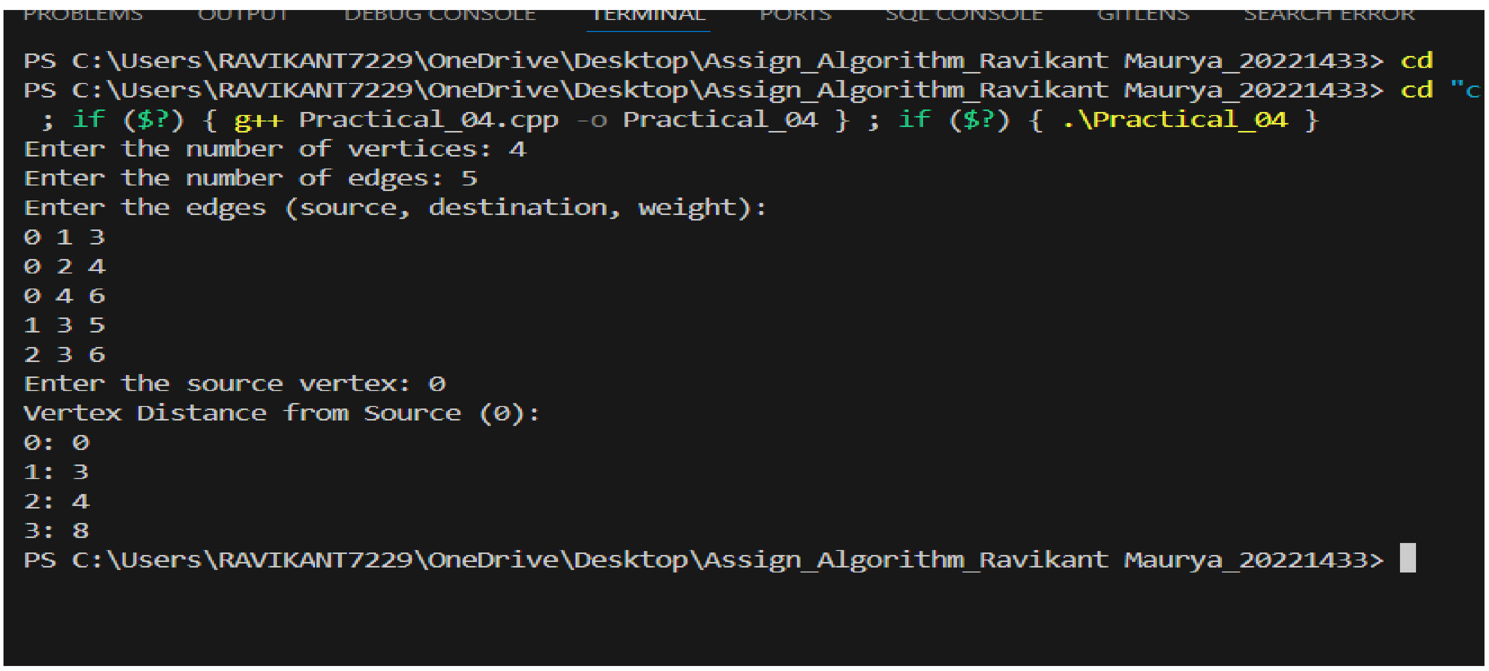
```
#include <iostream>
#include <vector>
#include <climits> // For INT_MAX to represent infinity

using namespace std;
```

```cpp
// Structure to represent an edge in the graph
struct Edge {
    int src, dest, weight; // Source vertex, destination vertex, and weight of the edge
};

// Bellman-Ford Algorithm
void bellmanFord(int V, int E, vector<Edge> &edges, int source) {
    // Initialize distances to all vertices as infinity (INT_MAX)
    vector<int> distance(V, INT_MAX);
    distance[source] = 0; // Distance to the source vertex is 0

    // **Relaxation Phase**
    // Relax all edges |V| - 1 times (where |V| is the number of vertices)
    for (int i = 1; i <= V - 1; ++i) {
        for (const auto &edge : edges) {
            int u = edge.src;     // Source vertex of the edge
            int v = edge.dest;   // Destination vertex of the edge
            int weight = edge.weight; // Weight of the edge

            // If the current distance to 'u' is not infinity and we find a shorter path to 'v'
            if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
                distance[v] = distance[u] + weight; // Update the distance to 'v'
            }
        }
    }

    // **Negative-Weight Cycle Detection**
    // Check all edges one more time to detect negative-weight cycles
    for (const auto &edge : edges) {
        int u = edge.src;     // Source vertex of the edge
        int v = edge.dest;   // Destination vertex of the edge
        int weight = edge.weight; // Weight of the edge

        // If we can still find a shorter path, a negative-weight cycle exists
        if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
            cout << "Graph contains a negative-weight cycle." << endl;
            return; // Terminate the function if a cycle is found
        }
    }

    // **Output the Shortest Distances**
    cout << "Vertex Distance from Source (" << source << "):" << endl;
    for (int i = 0; i < V; ++i) {
        if (distance[i] == INT_MAX) {
            cout << i << ": INF" << endl; // If a vertex is unreachable, its distance is "INF"
        } else {
            cout << i << ": " << distance[i] << endl; // Print the shortest distance to the 
vertex
        }
    }
}

int main() {
    int V, E, source; // Number of vertices, edges, and the source vertex
    cout << "Enter the number of vertices: ";
```



```cpp
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    vector<Edge> edges(E); // Array to store the edges
    cout << "Enter the edges (source, destination, weight):" << endl;
    for (int i = 0; i < E; ++i) {
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight; // Input each edge
    }

    cout << "Enter the source vertex: ";
    cin >> source;

    bellmanFord(V, E, edges, source); // Run the Bellman-Ford algorithm

    return 0; // Program ends successfully
}
```

Output:-



```
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> cd
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> cd "c
 ; if ($?) { g++ Practical_04.cpp -o Practical_04 } ; if ($?) { .\Practical_04 }
Enter the number of vertices: 4
Enter the number of edges: 5
Enter the edges (source, destination, weight):
0 1 3
0 2 4
0 4 6
1 3 5
2 3 6
Enter the source vertex: 0
Vertex Distance from Source (0):
0: 0
1: 3
2: 4
3: 8
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> 
```

Q.5:- Write a program to implement a B-Tree.

Explanation:-

BTree Node Structure:

- The **BTreeNode** class represents a node in the BTree.

- It holds an array of keys, pointers to child nodes, and some other important properties like the number of keys and whether it's a leaf node.

BTree Structure:

- The **BTree** class manages the root node and the minimum degree (t) of the tree.
- It provides methods to insert keys and traverse the tree.

BTreeNode Constructor:

- The constructor of the BTreeNode initializes the node with a minimum degree, and sets up arrays for keys and children.
- It also sets the number of keys (n) to zero and sets the isLeaf flag.

Insertion into a Non-Full Node:

- The insertNonFull method inserts a key into a node that isn't full.
- If the node is a leaf, it places the key in the correct position by shifting other keys.
- If the node is not a leaf, it finds the correct child node and recursively inserts the key.

Splitting Full Nodes:

- When a node becomes full (i.e., it has 2*t-1 keys), the splitChild method splits the node into two.
- The middle key moves up to the parent node, and a new child node is created to hold the split keys.

Main Function and Insertion:

- The main function demonstrates the use of the B Tree.
- It creates a BTree with a minimum degree of 3 and inserts several keys into it.
- Finally, it traverses the tree in order and prints the keys in the B Tree.

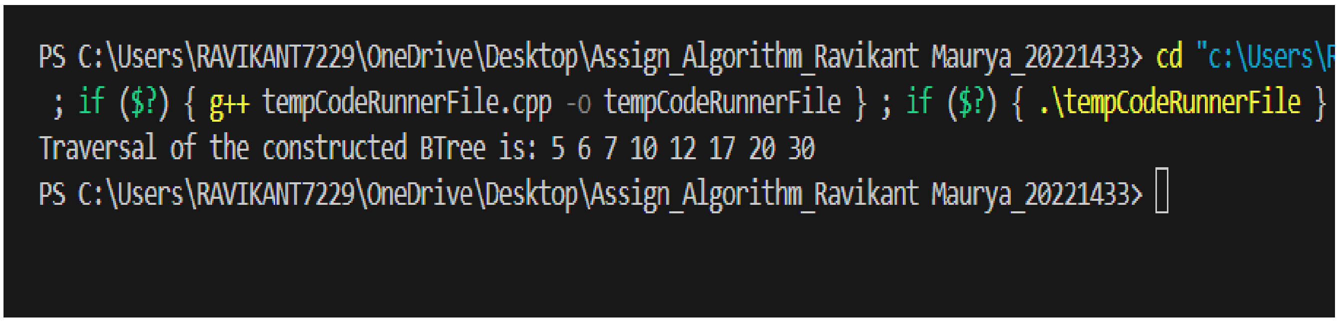
Code:-

```
#include <iostream>
using namespace std;

// A BTree Node
class BTreeNode {
public:
    int* keys;          // Array of keys in the node
    int t;              // Minimum degree (defines the range for the number of keys)
```

```cpp
    BTreeNode** children; // Array of child pointers
    int n;            // Current number of keys in the node
    bool isLeaf;      // True if the node is a leaf node

    // Constructor to initialize a BTreeNode
    BTreeNode(int t, bool isLeaf);

    // Traverse the tree in-order (from smallest to largest key)
    void traverse();

    // Insert a new key into a non-full node
    void insertNonFull(int k);

    // Split a child node that is full (number of keys = 2t - 1)
    void splitChild(int i, BTreeNode* y);

    // Allow BTree to access private members of BTreeNode
    friend class BTree;
};

// A BTree class
class BTree {
public:
    BTreeNode* root; // Pointer to the root node
    int t;           // Minimum degree

    // Constructor to initialize an empty BTree
    BTree(int t) {
        root = nullptr;
        this->t = t;
    }

    // Traverse the entire tree
    void traverse() {
        if (root != nullptr)
            root->traverse();
    }

    // Insert a key into the BTree
    void insert(int k);
};

// BTreeNode constructor
BTreeNode::BTreeNode(int t, bool isLeaf) {
    this->t = t;                       // Minimum degree
    this->isLeaf = isLeaf;             // Whether this node is a leaf
    keys = new int[2 * t - 1];         // Allocate space for keys
    children = new BTreeNode*[2 * t]; // Allocate space for child pointers
    n = 0;                             // Initialize number of keys to 0
}

// Traverse the subtree rooted at this node
void BTreeNode::traverse() {
    int i;
    // Traverse all children and keys
    for (i = 0; i < n; i++) {
        if (!isLeaf)                    // If the node is not a leaf, traverse the child before
the key
            children[i]->traverse();
        cout << keys[i] << " ";         // Print the key
    }

    // Traverse the last child
    if (!isLeaf)
        children[i]->traverse();
}

// Insert a key into a non-full node
void BTreeNode::insertNonFull(int k) {
    int i = n - 1; // Start from the rightmost key

    if (isLeaf) {
        // Shift keys to create space for the new key
        while (i >= 0 && keys[i] > k) {
            keys[i + 1] = keys[i];
            i--;
        }

        // Insert the new key into its correct position
        keys[i + 1] = k;
        n++;
    } else {
        // Find the child that will receive the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // Check if the found child is full
        if (children[i + 1]->n == 2 * t - 1) {
            // Split the child and adjust keys
            splitChild(i + 1, children[i + 1]);

            // After the split, decide which child to insert into
            if (keys[i + 1] < k)
                i++;
        }
        children[i + 1]->insertNonFull(k);
    }
}

// Split a child node y that is full
void BTreeNode::splitChild(int i, BTreeNode* y) {
    // Create a new node z to hold (t-1) keys of y
    BTreeNode* z = new BTreeNode(y->t, y->isLeaf);
    z->n = t - 1;

    // Copy the last (t-1) keys from y to z
    for (int j = 0; j < t - 1; j++)
        z->keys[j] = y->keys[j + t];

    // Copy the last t children from y to z (if not a leaf)
    if (!y->isLeaf) {
        for (int j = 0; j < t; j++)
            z->children[j] = y->children[j + t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Make space for the new child in this node
    for (int j = n; j >= i + 1; j--)
        children[j + 1] = children[j];

    // Link the new child z to this node
    children[i + 1] = z;

    // Move a key from y to this node
    for (int j = n - 1; j >= i; j--)
        keys[j + 1] = keys[j];

    keys[i] = y->keys[t - 1];
    n++; // Increment the number of keys in this node
}

// Insert a key into the BTree
void BTree::insert(int k) {
    if (root == nullptr) {
        // If the tree is empty, create the root
        root = new BTreeNode(t, true);
        root->keys[0] = k; // Insert the key
        root->n = 1;       // Initialize number of keys in root to 1
    } else {
        // If the root is full, split it
        if (root->n == 2 * t - 1) {
            BTreeNode* s = new BTreeNode(t, false);

            // Make the old root a child of the new root
            s->children[0] = root;

            // Split the old root and adjust
            s->splitChild(0, root);

            // Decide which child will receive the new key
            int i = (s->keys[0] < k) ? 1 : 0;
            s->children[i]->insertNonFull(k);

            // Update the root
            root = s;
        } else {
            // Insert into the non-full root
            root->insertNonFull(k);
        }
    }
}

// Main function to demonstrate BTree functionality
```

```cpp
int main() {
    BTree t(3); // Create a BTree with minimum degree 3

    // Insert elements
    t.insert(10);
    t.insert(20);
    t.insert(5);
    t.insert(6);
    t.insert(12);
    t.insert(30);
    t.insert(7);
    t.insert(17);

    cout << "Traversal of the constructed BTree is: ";
    t.traverse(); // Print the BTree
    cout << endl;

    return 0;
}
```

Output:-



```
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_2022143> cd "c:\Users\
 ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Traversal of the constructed BTree is: 5 6 7 10 12 17 20 30
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_2022143>
```

Q.6:- Write a program to implement the Tree Data structure, which supports the following operations:

a. Insert

b. Search

Explanation:-

BTreeNode Class:

- Represents a node in the BTree with an array of keys, child pointers, and information about the node's minimum degree (t), number of keys (n), and whether it is a leaf (isLeaf).
- Provides methods for searching (search()), inserting keys into a non-full node (insertNonFull()), and splitting a child node (splitChild()).

BTree Class:

- Represents the entire BTree with a root node (root) and a minimum degree (t).
- Provides methods for traversing the tree (traverse()), searching for a key (search()), inserting a key into the tree (insert()), and inserting multiple keys (insertMultiple()).

Insert Method:

- If the root is empty, a new root node is created. If the root is full, the tree grows by creating a new root, splitting the old root, and inserting the key into the appropriate child. Otherwise, the key is inserted into the root directly.

InsertNonFull Method:

- If the node is a leaf, the key is inserted into the correct position. If the node is not a leaf, it finds the correct child to insert the key, and if the child is full, it splits the child before inserting.

SplitChild Method:

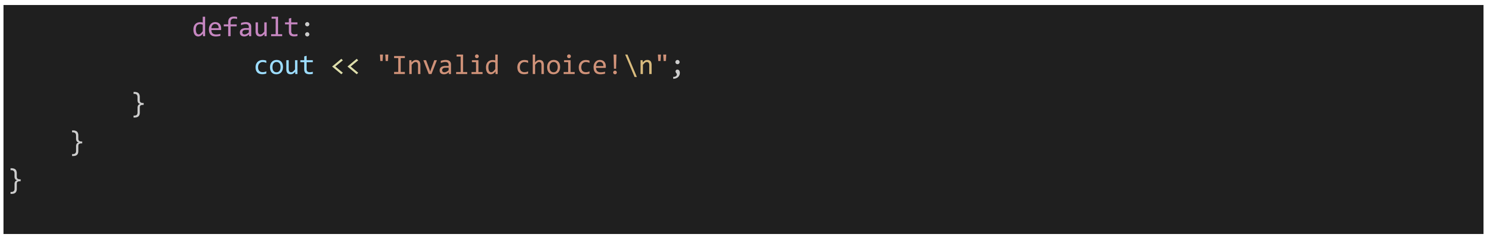
- Splits a full child node into two, promoting the middle key to the parent. The split ensures that the parent node maintains the BTree property (keys are ordered and nodes are half full).

InsertMultiple Method:

- Allows insertion of multiple space-separated keys by reading them from a string and calling the insert() method for each key.

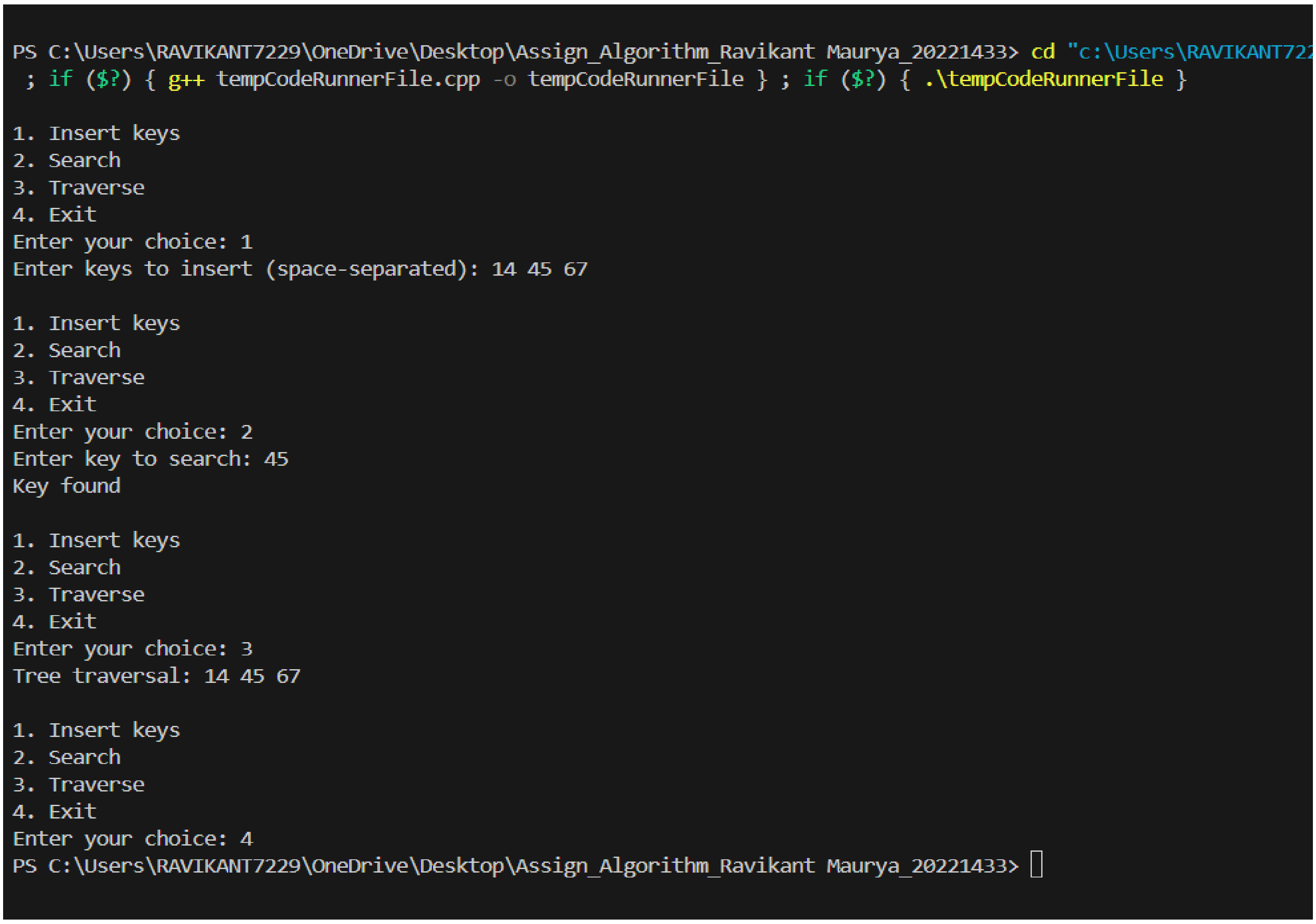
Code:-

```cpp
#include <iostream>
#include <sstream>
using namespace std;

// A BTree Node
class BTreeNode {
public:
    int* keys;             // Array to store keys in the node
    BTreeNode** children; // Array of child pointers
    int t;                 // Minimum degree of the BTree
    int n;                 // Current number of keys in this node
    bool isLeaf;           // Indicates whether the node is a leaf
    // Constructor to initialize a node
    BTreeNode(int t, bool isLeaf);

    // Search for a key in the subtree rooted at this node
    BTreeNode* search(int k);

    // Insert a key into a non-full node
    void insertNonFull(int k);

    // Split a full child into two
    void splitChild(int i, BTreeNode* y);

    // Traverse the tree rooted at this node and print its keys
    void traverse();

    // Allow access to private members from the BTree class
    friend class BTree;
};

// BTree class
class BTree {
public:
    BTreeNode* root; // Root of the BTree
    int t;           // Minimum degree of the BTree

    // Constructor: Initializes an empty tree with minimum degree `t`
    BTree(int t) {
        root = nullptr;
        this->t = t;
    }

    // Traverse the entire tree and print its keys
    void traverse() {
        if (root != nullptr) root->traverse();
    }

    // Search for a key in the BTree
    BTreeNode* search(int k) {
        return (root == nullptr) ? nullptr : root->search(k);
    }

    // Insert a single key into the tree
    void insert(int k);

    // Insert multiple keys from a space-separated string
    void insertMultiple(const string& keys);
};

// Constructor for BTreeNode
BTreeNode::BTreeNode(int t, bool isLeaf) {
    this->t = t;
    this->isLeaf = isLeaf;

    // Allocate space for the maximum possible number of keys and children
    keys = new int[2 * t - 1];
    children = new BTreeNode*[2 * t];
    n = 0; // Initially, there are no keys

    // Initialize keys and children to default values
    for (int i = 0; i < 2 * t - 1; i++) keys[i] = 0;
    for (int i = 0; i < 2 * t; i++) children[i] = nullptr;
}

// Search for a key in the subtree rooted at this node
BTreeNode* BTreeNode::search(int k) {
    int i = 0;

    // Find the first key that is greater than or equal to `k`
    while (i < n && k > keys[i]) i++;

    // If the key is found, return this node
    if (i < n && keys[i] == k) return this;

    // If this node is a leaf, the key is not in the tree
    if (isLeaf) return nullptr;

    // Recur to the appropriate child
    return children[i]->search(k);
}

// Traverse the BTree and print its keys in sorted order
void BTreeNode::traverse() {
    int i;
    for (i = 0; i < n; i++) {
        // If the node is not a leaf, traverse the child before the key
        if (!isLeaf && children[i]) children[i]->traverse();
        cout << keys[i] << " "; // Print the key
    }

    // Traverse the last child, if it exists
    if (!isLeaf && children[i]) children[i]->traverse();
}

// Insert a single key into the BTree
void BTree::insert(int k) {
    if (!root) {
        // If the tree is empty, create the root and insert the key
        root = new BTreeNode(t, true);
        root->keys[0] = k;
        root->n = 1; // Update the number of keys in the root
    } else {
        // If the root is full, grow the tree
        if (root->n == 2 * t - 1) {
            BTreeNode* s = new BTreeNode(t, false);

            // Make the old root a child of the new root
            s->children[0] = root;

            // Split the old root and update the new root
            s->splitChild(0, root);
            // Insert the new key into the appropriate child
            if (s->keys[0] < k) s->children[1]->insertNonFull(k);
            else s->children[0]->insertNonFull(k);

            root = s; // Update the root pointer
        } else {
            // If the root is not full, insert the key directly
            root->insertNonFull(k);
        }
    }
}

// Insert a key into a non-full node
void BTreeNode::insertNonFull(int k) {
    int i = n - 1;

    if (isLeaf) {
        // Find the correct position for the new key and shift keys as needed
        while (i >= 0 && keys[i] > k) {
            keys[i + 1] = keys[i];
            i--;
        }

        // Insert the key at the correct position
        keys[i + 1] = k;
        n++;
    } else {
        // Find the child that will receive the new key
        while (i >= 0 && keys[i] > k) i--;

        // If the child is full, split it
        if (children[i + 1]->n == 2 * t - 1) {
            splitChild(i + 1, children[i + 1]);

            // Adjust the index if the middle key moves up
            if (keys[i + 1] < k) i++;
        }

        // Recur to the appropriate child
        children[i + 1]->insertNonFull(k);
    }
}

// Split a full child into two
void BTreeNode::splitChild(int i, BTreeNode* y) {
    BTreeNode* z = new BTreeNode(y->t, y->isLeaf);
    z->n = t - 1; // z will have t-1 keys

    // Move the last t-1 keys of y to z
    for (int j = 0; j < t - 1; j++) z->keys[j] = y->keys[j + t];

    // Move the children of y to z, if applicable
    if (!y->isLeaf) {
        for (int j = 0; j < t; j++) z->children[j] = y->children[j + t];
    }

    y->n = t - 1; // Reduce the number of keys in y

    // Shift children of the current node to make space for z
    for (int j = n; j >= i + 1; j--) children[j + 1] = children[j];
    children[i + 1] = z;

    // Shift keys of the current node to make space for the middle key of y
    for (int j = n - 1; j >= i; j--) keys[j + 1] = keys[j];
    keys[i] = y->keys[t - 1]; // Move the middle key of y to the current node

    n++; // Update the key count of the current node
}

// Insert multiple keys separated by spaces
void BTree::insertMultiple(const string& keys) {
    stringstream ss(keys);
    int key;
    while (ss >> key) {
        insert(key);
    }
}

// Main function to demonstrate the program
int main() {
    BTree t(3); // BTree with minimum degree 3

    int choice;
    string input;
    while (true) {
        cout << "\n1. Insert keys\n2. Search\n3. Traverse\n4. Exit\nEnter your choice: ";
        cin >> choice;
        cin.ignore(); // Clear the newline character

        switch (choice) {
            case 1:
                cout << "Enter keys to insert (space-separated): ";
                getline(cin, input);
                t.insertMultiple(input);
                break;
            case 2:
                int key;
                cout << "Enter key to search: ";
                cin >> key;
                if (t.search(key)) cout << "Key found\n";
                else cout << "Key not found\n";
                break;
            case 3:
                cout << "Tree traversal: ";
                t.traverse();
                cout << endl;
                break;
            case 4:
                return 0;
```



```
            default:
                cout << "Invalid choice!\n";
        }
    }
}
```

Output:-



```
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> cd "c:\Users\RAVIKANT722
 ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }

1. Insert keys
2. Search
3. Traverse
4. Exit
Enter your choice: 1
Enter keys to insert (space-separated): 14 45 67

1. Insert keys
2. Search
3. Traverse
4. Exit
Enter your choice: 2
Enter key to search: 45
Key found

1. Insert keys
2. Search
3. Traverse
4. Exit
Enter your choice: 3
Tree traversal: 14 45 67

1. Insert keys
2. Search
3. Traverse
4. Exit
Enter your choice: 4
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> 
```

Q.7:- Write a program to search a pattern in a given text using the KMP algorithm.

Explanation:-

This code implements the Knuth-Morris-Pratt (KMP) algorithm, a pattern-matching technique designed to emciently search for a string (pattern) within a larger string (text). Unlike simpler methods that might recheck characters unnecessarily, KMP optimizes the process by utilizing an auxiliary array, the Longest Prefix Sumx (LPS) array, to skip redundant checks.

The **buildLPSArray** function constructs the LPS array for the given pattern. This

array
essentially stores the lengths of the longest prefix of the pattern that also serves as a sumx for every position in the pattern. For example, if part of the pattern matches but a mismatch occurs, the LPS array allows the algorithm to continue from a point where the match can potentially resume, instead of starting over from scratch. The function iterates through the pattern, updating the LPS array based on how much of the pattern has been successfully matched.

The **KMPSearch** function uses the LPS array to search for the pattern in the text. It compares characters from the pattern to the text one by one. If a match is found for all characters in the pattern, the function prints the starting index of the pattern in the text. In case of a
mismatch, it uses the LPS array to decide how much of the pattern can still be matched and skips unnecessary comparisons. This ensures that the algorithm doesn't repeatedly compare already matched characters.

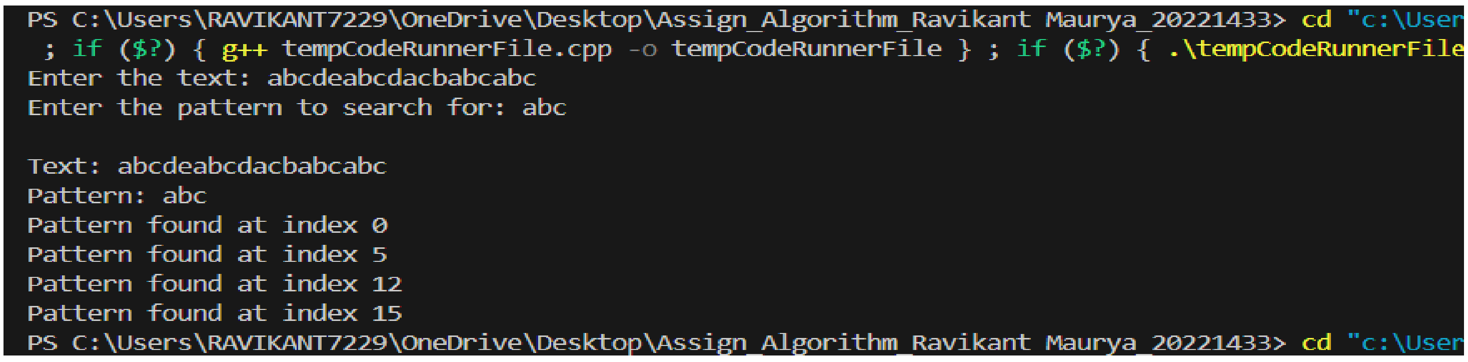
In the **main** function, the program takes input for both the text and the pattern from the user.
It then calls the KMPSearch function, which performs the search and outputs the indices where the pattern is found. This combination of preprocessing (building the LPS array) and
searching ensures emciency, especially for larger texts or patterns with repetitive structures.

Code:-

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// Function to build the LPS (Longest Prefix Suffix) array for the pattern
void buildLPSArray(const string& pattern, vector<int>& lps) {
    int m = pattern.length(); // Length of the pattern
    int length = 0;           // Length of the previous longest prefix suffix
    lps[0] = 0;               // LPS[0] is always 0
    int i = 1;                // Start from the second character of the pattern

    // Compute LPS array values for pattern[1...m-1]
    while (i < m) {
        if (pattern[i] == pattern[length]) {
            // If current character matches the character at the end of the current prefix
            length++;
            lps[i] = length; // Store the length of the current prefix suffix
            i++;
        } else {
            // If the characters do not match
            if (length != 0) {
                // Try the previous longest prefix suffix
                length = lps[length - 1];
            } else {
                // If no prefix suffix is found, set LPS[i] to 0
                lps[i] = 0;
                i++;
            }
        }
    }
}

// KMP pattern searching algorithm
void KMPSearch(const string& text, const string& pattern) {
    int n = text.length(); // Length of the text
    int m = pattern.length(); // Length of the pattern
    vector<int> lps(m); // LPS array to hold the longest prefix suffix values for the pattern

    // Build the LPS array for the pattern
    buildLPSArray(pattern, lps);

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

    // Loop through the text to find occurrences of the pattern
    while (i < n) {
        if (pattern[j] == text[i]) {
            // If characters match, move both indices forward
            i++;
            j++;
        }
```

```cpp
        // If the entire pattern is found
        if (j == m) {
            cout << "Pattern found at index " << i - j << endl;
            j = lps[j - 1]; // Get the next position to check in the pattern
        }
        // If there is a mismatch after j matches
        else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                // Skip the matched part of the pattern and try again
                j = lps[j - 1];
            } else {
                // If no match, move the text index forward
                i++;
            }
        }
    }
}

// Main function to demonstrate the KMP algorithm
int main() {
    string text, pattern;

    // Input text and pattern from the user
    cout << "Enter the text: ";
    getline(cin, text); // Get the full text input
    cout << "Enter the pattern to search for: ";
    getline(cin, pattern); // Get the pattern input

    // Display the input text and pattern
    cout << "\nText: " << text << endl;
    cout << "Pattern: " << pattern << endl;

    // Call KMPSearch to search for the pattern in the text
    KMPSearch(text, pattern);

    return 0;
}
```

Output:-



```
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> cd "c:\User
 ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile
Enter the text: abcdeabcdacbabcabc
Enter the pattern to search for: abc

Text: abcdeabcdacbabcabc
Pattern: abc
Pattern found at index 0
Pattern found at index 5
Pattern found at index 12
Pattern found at index 15
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> cd "c:\User
```

Q.8:- Write a program to implement a Suffix tree.

Explanation:-

This code builds a **suffix tree** for a given string. A sumx tree is a data structure that represents
all the sumxes of a string in a way that makes operations like substring search emcient.

The program has two main parts: the **SuffixTreeNode class** and the **SuffixTree class**.

1. SuffixTreeNode Class
   This class represents a single node in the sumx tree.
   - It has a children map that links the current node to its child nodes, based on characters.
   - The start and end variables define which part of the input string this node represents.
   - There's also a sumxIndex, which is used for leaf nodes to indicate where the sumx starts in the original string.
2. SuffixTree Class
   This class manages the whole sumx tree.
   - The main job of this class is to build the tree from the input string and display its structure.
   - It has a buildTree function to loop through all sumxes of the string and insert them into the tree using buildTreeUtil.

The tree is constructed by iterating through all sumxes of the input string. For each sumx, the buildTreeUtil function is called. This function inserts the sumx into the tree, one character at a time:

- If the character is not already a child of the current node, a new node is created. • The function continues down the tree until all characters of the sumx are added.
- Once the end of the sumx is reached, the node is marked with the sumx's starting index.

The root node is special, it doesn't represent any character and serves as the starting point for all sumxes.

The display function prints the structure of the tree. It's a recursive function that:

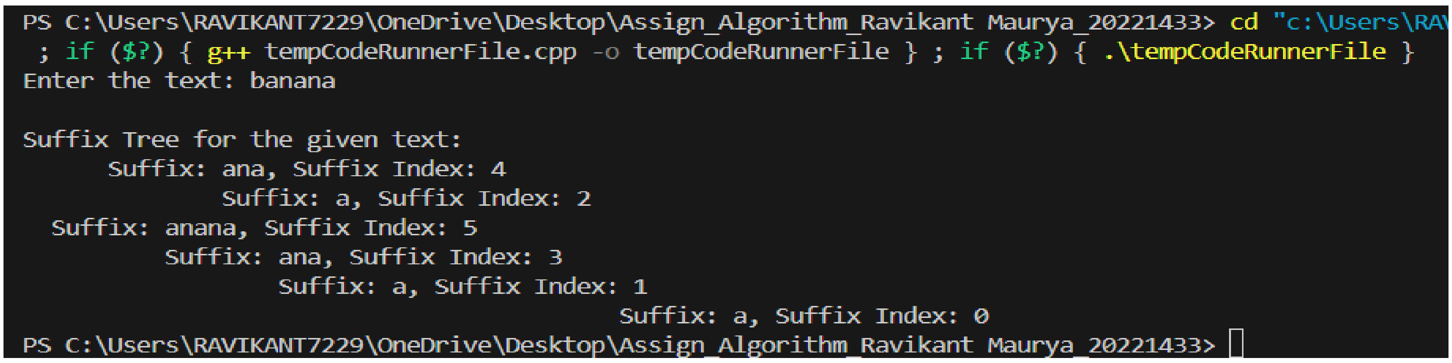
- Prints the substring represented by each node (using the start and end indices).

- For leaf nodes (nodes with a valid sumxIndex), it also prints the sumx index, showing where the sumx starts in the original string.

Code:-

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

using namespace std;

// Class representing a node in the suffix tree
class SuffixTreeNode {
public:
    unordered_map<char, SuffixTreeNode*> children; // Map to store child nodes (keyed by 
characters)
    int start, *end;                               // Start and end indices of the substring 
represented by the edge
    int suffixIndex;                               // Index of the suffix ending at this node

    // Constructor to initialize a node
    SuffixTreeNode(int start, int* end) {
        this->start = start;
        this->end = end;
        this->suffixIndex = -1; // Initialize suffix index to -1 (not a leaf initially)
    }
};

// Class representing the Suffix Tree
class SuffixTree {
public:
    string text;                // The input string for which the suffix tree is constructed
    SuffixTreeNode* root;       // Root node of the suffix tree
    int size;                   // Length of the input string

    // Constructor to initialize the suffix tree
    SuffixTree(string s) {
        text = s;
        size = s.length();
        root = new SuffixTreeNode(-1, new int(-1)); // Root node has no substring associated
    }

    // Function to build the suffix tree
    void buildTree() {
        for (int i = 0; i < size; i++) {
            // Build the suffix tree for each suffix starting at index `i`
            buildTreeUtil(i, size - 1, i);
        }
    }

    // Utility function to insert a suffix into the tree
    void buildTreeUtil(int start, int end, int suffixIndex) {
        SuffixTreeNode* currentNode = root; // Start from the root node

        // Traverse the tree and insert characters from the suffix
        for (int i = start; i <= end; i++) {
            char currentChar = text[i]; // Current character in the suffix

            // If there is no edge for the current character, create a new node
            if (currentNode->children.find(currentChar) == currentNode->children.end()) {
                SuffixTreeNode* newNode = new SuffixTreeNode(i, new int(end));
                currentNode->children[currentChar] = newNode; // Add the new node as a child
            }

            // Move to the next node in the tree
            currentNode = currentNode->children[currentChar];
        }

        // Mark the leaf node with the suffix index
        currentNode->suffixIndex = suffixIndex;
    }

    // Function to display the suffix tree
    void display(SuffixTreeNode* node, int level) {
        if (!node) return; // Base case: if the node is null, return

        // Print indentation for the current level
        for (int i = 0; i < level; i++) {
            cout << "  ";
        }

        // If the node represents a suffix (leaf node), print its details
        if (node->suffixIndex != -1) {
            cout << "Suffix: " << text.substr(node->start, *(node->end) - node->start + 1)
                 << ", Suffix Index: " << node->suffixIndex << endl;
        }

        // Recursively display all child nodes
        for (auto& child : node->children) {
            display(child.second, level + 1);
        }
    }
};

// Main function to demonstrate the suffix tree
int main() {
    string text;
    cout << "Enter the text: ";
    getline(cin, text); // Input the text from the user

    // Create a suffix tree
    SuffixTree suffixTree(text);
    suffixTree.buildTree();  // Build the suffix tree

    // Display the suffix tree
    cout << "\nSuffix Tree for the given text:\n";
    suffixTree.display(suffixTree.root, 0); // Display the tree starting from the root
    return 0;
}
```

Output:-



```
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433> cd "c:\Users\RA
 ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Enter the text: banana

Suffix Tree for the given text:
      Suffix: ana, Suffix Index: 4
              Suffix: a, Suffix Index: 2
  Suffix: anana, Suffix Index: 5
          Suffix: ana, Suffix Index: 3
                  Suffix: a, Suffix Index: 1
                                          Suffix: a, Suffix Index: 0
PS C:\Users\RAVIKANT7229\OneDrive\Desktop\Assign_Algorithm_Ravikant Maurya_20221433>
```

Thank You