BHARATI VIDYAPEETH'S

INSTITUTE OF COMPUTER APPLICATIONS & MANAGEMENT

(Affiliated to Guru Gobind Singh Indraprastha University,

Approved by AICTE, New Delhi)

Design and Analysis of Algorithms (MCA- 261) Practical File

Submitted To:

Dr. Saumya Bansal

(Assistant Professor)

Submitted By:

Shipra Vashist (03111604423)

MCA 3rd Sem, Sec. 1

INDEX

S. No	Problem Description	Sign
1	Write a program to implement Bubble Sort.	
2	Write a program to implement Quick Sort.	
3	Write a program to implement Merge Sort.	
4	Write a program to implement Binary Search on the given list of values.	
5	Write a program to perform Radix sort on a given list of numbers.	
6	Write a program to perform Bucket sort on a given list of numbers.	
7	Write a program to perform Counting sort on a given list of numbers.	
8	Given two matrices, perform Strassen's matrix multiplication.	
9	To Implement Matrix Chain Multiplication.	
10	Write a program to perform a naïve string-matching algorithm.	
11	Implement and analyze the disjoint data structure algorithm.	
12	Implement fractional Knapsack using Greedy approach and analyze the algorithm.	
13	To implement Huffman Coding and analyze its time complexity.	
14	Implement the Dijkstra Algorithm using Greedy and analyze the algorithm.	
15	Implement the Prims' and Kruskal Algorithm using Greedy and analyze the algorithm.	
16	Implement the Longest Common Subsequence using Dynamic Programming and analyze the algorithm.	
17	Implement Matrix Chain Multiplication using Dynamic Programming and analyze the algorithm.	
18	Implement Floyd-Warshell algorithm for a given graph.	

P1 Write a program to implement bubble sort

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

void bubbleSort(int arr[], int n) {
  for (int i = 0; i < n-1; i++) {
    // Last i elements are already sorted
    for (int j = 0; j < n-i-1; j++) {
      // Swap if the element is greater than the next
      if (arr[j] > arr[j+1]) {
        int temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }
    }
  }
}

void printArray(int arr[], int n) {
  for (int i = 0; i < n; i++)
    cout << arr[i] << " ";
  cout << endl;
}

int main() {
  int arr[] = {64, 34, 25, 12, 22, 11, 90};
```

```
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Unsorted array: ";
    printArray(arr, n);

    bubbleSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Output   Clear

```
Unsorted array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90


=== Code Execution Successful ===
```

P2 Write a program to implement quick sort.

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

int partition(int arr[], int low, int high) {
  int pivot = arr[high];
  int i = (low - 1);

  for (int j = low; j <= high - 1; j++) {
    if (arr[j] < pivot) {
      i++;
      swap(arr[i], arr[j]);
    }
  }
  swap(arr[i + 1], arr[high]);
  return (i + 1);
}

void quickSort(int arr[], int low, int high) {
  if (low < high) {
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}
```

```
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int arr[] = {10, 80, 30, 90, 40, 50, 70};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Output Clear

```
Unsorted array: 10 80 30 90 40 50 70
Sorted array: 10 30 40 50 70 80 90


=== Code Execution Successful ===
```

P3 Write a program to implement merge sort.

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

void merge(int arr[], int l, int m, int r) {
   int n1 = m - l + 1;
   int n2 = r - m;
   int L[n1], R[n2];

   for (int i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (int i = 0; i < n2; i++)
      R[i] = arr[m + 1 + i];

   int i = 0, j = 0, k = l;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      } else {
         arr[k] = R[j];
         j++;
      }
      k++;
   }
```

```
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
  }
}

void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
  }
}

void printArray(int arr[], int n) {
  for (int i = 0; i < n; i++)
    cout << arr[i] << " ";
  cout << endl;
```

```
}

int main() {
  int arr[] = {12, 11, 13, 5, 6, 7};
  int n = sizeof(arr) / sizeof(arr[0]);
  mergeSort(arr, 0, n - 1);
  cout << "Sorted array: ";
  printArray(arr, n);
  return 0;
}
```

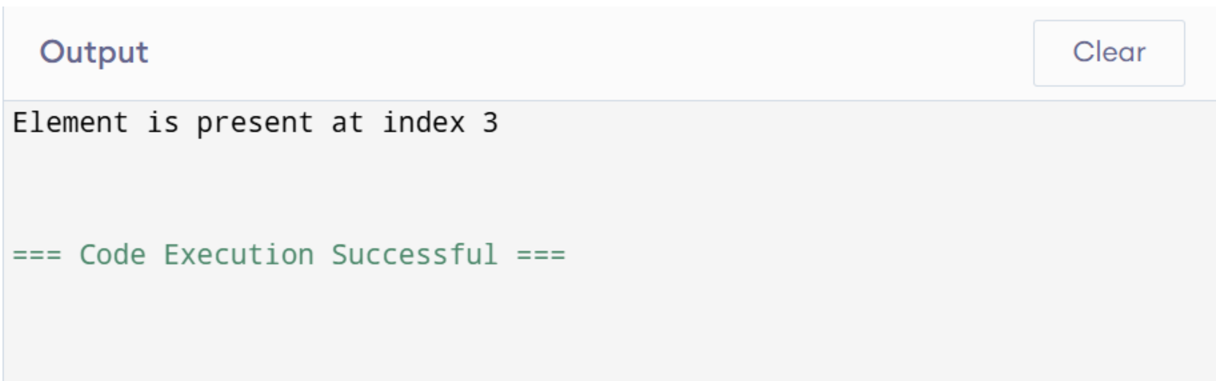
Output Clear

```
Sorted array: 5 6 7 11 12 13


=== Code Execution Successful ===
```

P4 Write a program to implement binary search on the given list of values.

```
#include <iostream>

using namespace std;


int binarySearch(int arr[], int l, int r, int x) {

   while (l <= r) {

      int mid = l + (r - l) / 2;


      if (arr[mid] == x)

         return mid;


      if (arr[mid] < x)

         l = mid + 1;

      else

         r = mid - 1;

   }

   return -1;

}


int main() {

   int arr[] = {2, 3, 4, 10, 40};

   int x = 10;

   int n = sizeof(arr) / sizeof(arr[0]);


   int result = binarySearch(arr, 0, n - 1, x);

   if (result == -1)
```

```
        cout << "Element is not present in array\n";
    else
        cout << "Element is present at index " << result << endl;

    return 0;
}
```



P5 Write a program to perform radix sort on a given list of numbers.

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

int getMax(int arr[], int n) {
  int mx = arr[0];
  for (int i = 1; i < n; i++)
    if (arr[i] > mx)
      mx = arr[i];
  return mx;
}

void countSort(int arr[], int n, int exp) {
  int output[n];
  int count[10] = {0};

  for (int i = 0; i < n; i++)
    count[(arr[i] / exp) % 10]++;

  for (int i = 1; i < 10; i++)
    count[i] += count[i - 1];

  for (int i = n - 1; i >= 0; i--) {
    output[count[(arr[i] / exp) % 10] - 1] = arr[i];
    count[(arr[i] / exp) % 10]--;
  }
```

```
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

void radixSort(int arr[], int n) {
    int m = getMax(arr, n);

    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);
    radixSort(arr, n);
    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Output

Clear

```
Sorted array: 2 24 45 66 75 90 170 802


=== Code Execution Successful ===
```

P6 Write a program to perform bucket sort on a given list of numbers.

```
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;


void bucketSort(float arr[], int n) {

  vector<float> buckets[n];


  // 1. Put elements into different buckets

  for (int i = 0; i < n; i++) {

    int bucketIndex = n * arr[i];  // Index in bucket

    buckets[bucketIndex].push_back(arr[i]);

  }


  // 2. Sort each bucket individually

  for (int i = 0; i < n; i++) {

    sort(buckets[i].begin(), buckets[i].end());

  }


  // 3. Concatenate all buckets into the original array

  int index = 0;

  for (int i = 0; i < n; i++) {

    for (size_t j = 0; j < buckets[i].size(); j++) {

      arr[index++] = buckets[i][j];

    }
```

```
    }
}

void printArray(float arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    float arr[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};
    int n = sizeof(arr) / sizeof(arr[0]);

    bucketSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Output

Clear

```
Sorted array: 0.1234 0.3434 0.565 0.656 0.665 0.897


=== Code Execution Successful ===
```

P7 Write a program to perform counting sort on a given list of numbers.

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

void countSort(int arr[], int n) {
  // Find the maximum element in the array
  int max = arr[0];
  for (int i = 1; i < n; i++) {
    if (arr[i] > max) {
      max = arr[i];
    }
  }

  // Create a count array to store the count of individual elements
  int count[max + 1] = {0};

  // Store the count of each element
  for (int i = 0; i < n; i++) {
    count[arr[i]]++;
  }

  // Modify count array to store cumulative count
  for (int i = 1; i <= max; i++) {
    count[i] += count[i - 1];
  }
```

```
    // Output array to store sorted elements
    int output[n];
    for (int i = n - 1; i >= 0; i--) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }

    // Copy the sorted elements back into the original array
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int arr[] = {4, 2, 2, 8, 3, 3, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    countSort(arr, n);

    cout << "Sorted array: ";
```

```
    printArray(arr, n);

    return 0;
}
```

Output   Clear

```
Sorted array: 1 2 2 3 3 4 8


=== Code Execution Successful ===
```

P8 Given two matrices, perform Strassen's matrix multiplication.

```
#include <iostream>

using namespace std;


// Function to add two matrices

void add(int A[2][2], int B[2][2], int C[2][2]) {

   for (int i = 0; i < 2; i++) {

      for (int j = 0; j < 2; j++) {

         C[i][j] = A[i][j] + B[i][j];

      }

   }

}


// Function to subtract two matrices

void subtract(int A[2][2], int B[2][2], int C[2][2]) {

   for (int i = 0; i < 2; i++) {

      for (int j = 0; j < 2; j++) {

         C[i][j] = A[i][j] - B[i][j];

      }

   }

}


// Strassen's Matrix Multiplication

void strassen(int A[2][2], int B[2][2], int C[2][2]) {

   int M1 = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]);

   int M2 = (A[1][0] + A[1][1]) * B[0][0];
```

```
    int M3 = A[0][0] * (B[0][1] - B[1][1]);
    int M4 = A[1][1] * (B[1][0] - B[0][0]);
    int M5 = (A[0][0] + A[0][1]) * B[1][1];
    int M6 = (A[1][0] - A[0][0]) * (B[0][0] + B[0][1]);
    int M7 = (A[0][1] - A[1][1]) * (B[1][0] + B[1][1]);

    // Calculating the final values for C
    C[0][0] = M1 + M4 - M5 + M7;
    C[0][1] = M3 + M5;
    C[1][0] = M2 + M4;
    C[1][1] = M1 - M2 + M3 + M6;
}

int main() {
    int A[2][2] = { {1, 2}, {3, 4} };
    int B[2][2] = { {5, 6}, {7, 8} };
    int C[2][2]; // Result matrix

    // Applying Strassen's Algorithm
    strassen(A, B, C);

    cout << "Resultant Matrix C (A * B):" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << C[i][j] << " ";
        }
```

```
        cout << endl;
    }

    return 0;
}
```

Output   Clear

```
Resultant Matrix C (A * B):
19 22
43 50


=== Code Execution Successful ===
```

P9 To Implement Matrix Chain Multiplication.

```
#include <iostream>
#include <climits> // For INT_MAX
using namespace std;

// Function to perform Matrix Chain Multiplication
int matrixChainMultiplication(int dims[], int n) {
   // Create a 2D table to store the minimum multiplication costs
   int dp[n][n];

   // Initialize the diagonal elements of dp to 0
   for (int i = 1; i < n; i++) {
      dp[i][i] = 0;
   }

   // l is the chain length
   for (int l = 2; l < n; l++) {
      for (int i = 1; i < n - l + 1; i++) {
         int j = i + l - 1;
         dp[i][j] = INT_MAX;

         // Try different places to split the product and calculate the cost
         for (int k = i; k < j; k++) {
            int cost = dp[i][k] + dp[k + 1][j] + dims[i - 1] * dims[k] * dims[j];

            // Update the minimum cost
```

```
            if (cost < dp[i][j]) {
                dp[i][j] = cost;
            }
        }
    }
}

// The minimum cost to multiply matrices A1...An-1 will be in dp[1][n-1]
return dp[1][n - 1];
}

int main() {
    // Array representing dimensions of matrices A1, A2, ..., An
    // If A1 is 10x30, A2 is 30x5, and A3 is 5x60, the array is {10, 30, 5, 60}
    int dims[] = {10, 30, 5, 60};
    int n = sizeof(dims) / sizeof(dims[0]);

    // Call the function
    int minCost = matrixChainMultiplication(dims, n);

    cout << "Minimum number of multiplications is " << minCost << endl;

    return 0;
}
```

Output

Clear

```
Minimum number of multiplications is 4500


=== Code Execution Successful ===
```

P10 Write a program to perform a naïve string matching algorithm.

```
#include <iostream>

#include <string>

using namespace std;


// Function to perform Naïve String Matching

void naiveStringMatching(string text, string pattern) {

  int n = text.length();

  int m = pattern.length();


  // Loop over each position in the text where pattern can potentially match

  for (int i = 0; i <= n - m; i++) {

    int j;


    // Check the characters of the pattern with the text

    for (j = 0; j < m; j++) {

      if (text[i + j] != pattern[j]) {

        break;

      }

    }


    // If all characters match, we found the pattern at index i

    if (j == m) {

      cout << "Pattern found at index " << i << endl;

    }

  }
```

```
}

int main() {
    string text = "ABABDABACDABABCABAB";
    string pattern = "ABABCABAB";

    // Call the function
    naiveStringMatching(text, pattern);

    return 0;
}
```

Output   Clear

```
Pattern found at index 10


=== Code Execution Successful ===
```

P12 Implement fractional Knapsack using Greedy approach and analyze the algorithm.

```
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;


// Structure to store weight and value of an item

struct Item {

   int weight;

   int value;

};


// Comparator function to sort items by value-to-weight ratio

bool compare(Item a, Item b) {

   double r1 = (double)a.value / a.weight;

   double r2 = (double)b.value / b.weight;

   return r1 > r2; // Sort in descending order of value/weight

}


// Function to calculate the maximum value that can be obtained in a fractional knapsack

double fractionalKnapsack(int W, vector<Item> items, int n) {

   // Sort items by value-to-weight ratio

   sort(items.begin(), items.end(), compare);


   double totalValue = 0.0;  // Result (maximum value)
```

```
    // Iterate through all items
    for (int i = 0; i < n; i++) {
        // If the item can be included fully
        if (items[i].weight <= W) {
            W -= items[i].weight;      // Decrease the weight of the knapsack
            totalValue += items[i].value; // Add the item's value
        }
        // If only a fraction of the item can be included
        else {
            totalValue += items[i].value * ((double)W / items[i].weight); // Add fractional value
            break;  // Since the knapsack is full
        }
    }

    return totalValue;
}

int main() {
    int W = 50; // Knapsack capacity
    vector<Item> items = {{10, 60}, {20, 100}, {30, 120}}; // List of items with weights and
values

    int n = items.size();

    // Calculate and display the maximum value
    double maxValue = fractionalKnapsack(W, items, n);
```

```
    cout << "Maximum value in Knapsack = " << maxValue << endl;

    return 0;
}
```

Output   Clear

```
Maximum value in Knapsack = 240


=== Code Execution Successful ===
```

P13 To implement Huffman Coding and analyze its time complexity.

```
#include <iostream>

#include <queue>

#include <unordered_map>

#include <vector>

using namespace std;


// A node in the Huffman Tree

struct Node {

    char ch;        // Character

    int freq;       // Frequency of the character

    Node *left, *right;


    // Constructor

    Node(char c, int f) {

        ch = c;

        freq = f;

        left = right = nullptr;

    }

};


// Comparator to order nodes in the priority queue (min-heap)

struct Compare {

    bool operator()(Node* a, Node* b) {

        return a->freq > b->freq; // Min-heap based on frequency

    }

};
```

```
// Function to generate Huffman Codes from the tree
void generateCodes(Node* root, string code, unordered_map<char, string>& huffmanCodes) {
    if (!root) return;

    // If it's a leaf node, add the character and its code
    if (!root->left && !root->right) {
        huffmanCodes[root->ch] = code;
    }

    // Recur for left and right children
    generateCodes(root->left, code + "0", huffmanCodes);
    generateCodes(root->right, code + "1", huffmanCodes);
}

// Huffman Encoding Function
void huffmanCoding(vector<char> chars, vector<int> freqs) {
    int n = chars.size();

    // Step 1: Create a min-heap
    priority_queue<Node*, vector<Node*>, Compare> pq;

    // Step 2: Add all characters to the min-heap
    for (int i = 0; i < n; i++) {
        pq.push(new Node(chars[i], freqs[i]));
    }
```

```
    // Step 3: Build the Huffman Tree
    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop(); // Smallest freq
        Node* right = pq.top(); pq.pop(); // Second smallest freq

        // Create a new internal node with combined frequency
        Node* combined = new Node('\0', left->freq + right->freq);
        combined->left = left;
        combined->right = right;

        pq.push(combined); // Add the combined node back to the heap
    }

    // The remaining node is the root of the Huffman Tree
    Node* root = pq.top();

    // Step 4: Generate Huffman Codes
    unordered_map<char, string> huffmanCodes;
    generateCodes(root, "", huffmanCodes);

    // Step 5: Print the Huffman Codes
    cout << "Huffman Codes:" << endl;
    for (auto pair : huffmanCodes) {
        cout << pair.first << ": " << pair.second << endl;
    }
}
```

```
// Main function
int main() {
    vector<char> chars = {'a', 'b', 'c', 'd', 'e'};
    vector<int> freqs = {5, 9, 12, 13, 16};

    huffmanCoding(chars, freqs);

    return 0;
}
```

Output Clear

```
Huffman Codes:
e: 11
b: 101
a: 100
d: 01
c: 00
```

P14 Implement the Dijkstra Algorithm using Greedy and analyze the algorithm.

```
#include <iostream>

#include <vector>

#include <queue>

#include <utility>

#include <climits>


using namespace std;


// Function to implement Dijkstra's Algorithm

vector<int> dijkstra(int V, vector<vector<pair<int, int>>>& graph, int src) {

  vector<int> dist(V, INT_MAX);

  dist[src] = 0;


  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

  pq.push({0, src}); // (distance, vertex)


  while (!pq.empty()) {

    int u = pq.top().second;

    pq.pop();


    // Traverse all adjacent vertices

    for (auto& neighbor : graph[u]) {

      int v = neighbor.first;

      int weight = neighbor.second;
```

```
            // Relaxation step
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}

// Main function to demonstrate Dijkstra's algorithm
int main() {
    int V = 5; // Number of vertices
    vector<vector<pair<int, int>>> graph(V);

    // Adding edges (u, v, weight)
    graph[0].emplace_back(1, 10);
    graph[0].emplace_back(4, 5);
    graph[1].emplace_back(2, 1);
    graph[2].emplace_back(3, 2);
    graph[4].emplace_back(1, 3);
    graph[4].emplace_back(2, 9);
    graph[4].emplace_back(3, 2);
    graph[3].emplace_back(2, 6);
```

```
    vector<int> distances = dijkstra(V, graph, 0);

    cout << "Vertex Distance from Source\n";
    for (int i = 0; i < V; i++) {
        cout << i << "\t\t" << distances[i] << endl;
    }

    return 0;
}
```

Output Clear

```
Vertex Distance from Source
0		0
1		8
2		9
3		7
4		5


=== Code Execution Successful ===
```

P15 Implement the Prims' and Kruskal Algorithm using Greedy and analyze the algorithm.

```
#include <iostream>

#include <vector>

#include <queue>

#include <utility>

#include <climits>


using namespace std;


// Function to implement Prim's Algorithm

void prim(int V, vector<vector<pair<int, int>>>& graph) {

  vector<int> key(V, INT_MAX);

  vector<bool> inMST(V, false);

  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;


  key[0] = 0; // Starting vertex

  pq.push({0, 0}); // (key, vertex)


  while (!pq.empty()) {

    int u = pq.top().second;

    pq.pop();

    inMST[u] = true; // Include vertex in MST


    // Traverse all adjacent vertices

    for (auto& neighbor : graph[u]) {
```

```
            int v = neighbor.first;
            int weight = neighbor.second;

            // If not in MST and weight is less than key
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                pq.push({key[v], v});
            }
        }
    }

    // Output the total weight of the MST
    int totalWeight = 0;
    for (int i = 0; i < V; i++) {
        totalWeight += key[i];
    }
    cout << "Total weight of MST using Prim's: " << totalWeight << endl;
}

// Main function to demonstrate Prim's algorithm
int main() {
    int V = 5; // Number of vertices
    vector<vector<pair<int, int>>> graph(V);

    // Adding edges (u, v, weight)
    graph[0].emplace_back(1, 2);
```

```
    graph[0].emplace_back(3, 6);
    graph[1].emplace_back(0, 2);
    graph[1].emplace_back(2, 3);
    graph[1].emplace_back(3, 8);
    graph[1].emplace_back(4, 5);
    graph[2].emplace_back(1, 3);
    graph[2].emplace_back(4, 7);
    graph[3].emplace_back(0, 6);
    graph[3].emplace_back(1, 8);
    graph[4].emplace_back(1, 5);
    graph[4].emplace_back(2, 7);

    prim(V, graph);

    return 0;
}
```

Output   Clear

```
Total weight of MST using Prim's: 16


=== Code Execution Successful ===
```

Kruskal

```
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;


// Structure to represent an edge

struct Edge {

   int src, dest, weight;

   Edge(int s, int d, int w) : src(s), dest(d), weight(w) {}

};


// Comparator to sort edges by weight

bool compareEdges(Edge a, Edge b) {

   return a.weight < b.weight;

}


// Find function for Union-Find (with path compression)

int findParent(int node, vector<int>& parent) {

   if (parent[node] == node) {

      return node;

   }

   return parent[node] = findParent(parent[node], parent); // Path compression

}


// Union function for Union-Find
```

```
void unionSets(int u, int v, vector<int>& parent, vector<int>& rank) {
   int rootU = findParent(u, parent);
   int rootV = findParent(v, parent);

   if (rootU != rootV) {
      if (rank[rootU] < rank[rootV]) {
         parent[rootU] = rootV;
      } else if (rank[rootU] > rank[rootV]) {
         parent[rootV] = rootU;
      } else {
         parent[rootV] = rootU;
         rank[rootU]++;
      }
   }
}

// Kruskal's Algorithm
void kruskal(int V, vector<Edge>& edges) {
   // Step 1: Sort all edges by weight
   sort(edges.begin(), edges.end(), compareEdges);

   // Initialize Union-Find data structures
   vector<int> parent(V);
   vector<int> rank(V, 0);
   for (int i = 0; i < V; i++) {
      parent[i] = i;
```

```
    }

    // Resultant MST
    vector<Edge> mst;
    int mstCost = 0;

    // Step 2: Process edges in sorted order
    for (Edge& edge : edges) {
        int u = edge.src;
        int v = edge.dest;

        // Check if adding this edge creates a cycle
        if (findParent(u, parent) != findParent(v, parent)) {
            mst.push_back(edge);
            mstCost += edge.weight;
            unionSets(u, v, parent, rank);
        }
    }

    // Step 3: Output the MST
    cout << "Edges in the Minimum Spanning Tree:" << endl;
    for (Edge& edge : mst) {
        cout << edge.src << " -- " << edge.dest << " == " << edge.weight << endl;
    }
    cout << "Total Cost of MST: " << mstCost << endl;
}
```

```
int main() {
    int V = 5; // Number of vertices
    vector<Edge> edges;

    // Graph edges (src, dest, weight)
    edges.push_back(Edge(0, 1, 10));
    edges.push_back(Edge(0, 2, 6));
    edges.push_back(Edge(0, 3, 5));
    edges.push_back(Edge(1, 3, 15));
    edges.push_back(Edge(2, 3, 4));

    kruskal(V, edges);

    return 0;
}
```

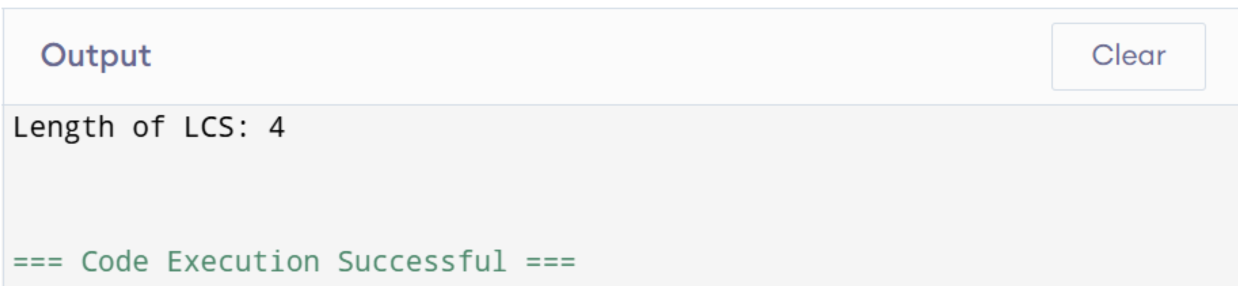
Output   Clear

```
Edges in the Minimum Spanning Tree:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Total Cost of MST: 19
```

P16 Implement the Longest Common Subsequence using Dynamic Programming and analyze the algorithm.

```
#include <iostream>

#include <vector>

#include <string>


using namespace std;


// Function to find LCS using Dynamic Programming

int lcs(const string& s1, const string& s2) {

  int m = s1.length();

  int n = s2.length();

  vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));


  // Build the dp table

  for (int i = 1; i <= m; i++) {

    for (int j = 1; j <= n; j++) {

      if (s1[i - 1] == s2[j - 1]) {

        dp[i][j] = dp[i - 1][j - 1] + 1; // If characters match

      } else {

        dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); // If not

      }

    }

  }


  return dp[m][n]; // Length of LCS
```

```
}

// Main function to demonstrate LCS
int main() {
   string s1 = "AGGTAB";
   string s2 = "GXTXAYB";
   cout << "Length of LCS: " << lcs(s1, s2) << endl;
   return 0;
}
```



}

P17 Implement Matrix Chain Multiplication using Dynamic Programming and analyze the algorithm

```
#include <iostream>

#include <vector>

#include <limits.h>


using namespace std;


// Function to find the minimum number of multiplications

int matrixChainOrder(const vector<int>& p) {

  int n = p.size() - 1; // Number of matrices

  vector<vector<int>> dp(n, vector<int>(n, 0));


  // L is the chain length

  for (int L = 2; L <= n; L++) {

    for (int i = 0; i < n - L + 1; i++) {

      int j = i + L - 1;

      dp[i][j] = INT_MAX;


      // Calculate the minimum cost

      for (int k = i; k < j; k++) {

        int q = dp[i][k] + dp[k + 1][j] + p[i] * p[k + 1] * p[j + 1];

        dp[i][j] = min(dp[i][j], q);

      }

    }

  }
```

```
    return dp[0][n - 1]; // Minimum cost
}

// Main function to demonstrate Matrix Chain Multiplication
int main() {
    vector<int> p = {1, 2, 3, 4}; // Dimensions
    cout << "Minimum number of multiplications: " << matrixChainOrder(p) << endl;
    return 0;
}
```

Output Clear

```
Minimum number of multiplications: 18


=== Code Execution Successful ===
```

P18 Implement Floyd-Warshell algorithm for a given graph.

```
#include <iostream>

#include <vector>

#include <limits.h>


using namespace std;


// Function to implement Floyd-Warshall Algorithm

void floydWarshall(vector<vector<int>>& graph) {

   int V = graph.size();


   // Updating distances

   for (int k = 0; k < V; k++) {

      for (int i = 0; i < V; i++) {

         for (int j = 0; j < V; j++) {

            if (graph[i][k] != INT_MAX && graph[k][j] != INT_MAX) {

               graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);

            }

         }

      }

   }

}


// Main function to demonstrate Floyd-Warshall Algorithm

int main() {

   vector<vector<int>> graph = {
```

```
        {0, 5, INT_MAX, 10},
        {INT_MAX, 0, 3, INT_MAX},
        {INT_MAX, INT_MAX, 0, 1},
        {INT_MAX, INT_MAX, INT_MAX, 0}
    };

    floydWarshall(graph);

    cout << "Shortest distances between every pair of vertices:\n";
    for (const auto& row : graph) {
        for (int dist : row) {
            if (dist == INT_MAX) cout << "INF ";
            else cout << dist << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output Clear

```
Shortest distances between every pair of vertices:
0 5 8 9
INF 0 3 4
INF INF 0 1
INF INF INF 0


=== Code Execution Successful ===
```