LAB MANUAL

Subject : Design and Analysis of Algorithm using C++
Code : LC–CSE-325

Department of Computer Science Engineering
RPS College of Engineering and Technology
Mahendergarh , Haryana - 123029

INSTRUCTIONS
### LIST OF PROGRAMS

<table>
<thead>
<tr>
<th>PROGRAM NO.</th>
<th>NAME OF PROGRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write a Program for iterative Binary Search</td>
</tr>
<tr>
<td>2</td>
<td>Write a Program for recursive Binary Search</td>
</tr>
<tr>
<td>3</td>
<td>Write a Program to sort a given set of elements using the Quick Sort</td>
</tr>
<tr>
<td>4</td>
<td>Write a Program to sort a given set of elements using the Merge Sort</td>
</tr>
<tr>
<td>5</td>
<td>Write a Program to sort a given set of elements using the Selection Sort</td>
</tr>
<tr>
<td>6</td>
<td>Write a Program for implementation of Fractional Knapsack problem using Greedy Method</td>
</tr>
<tr>
<td>7</td>
<td>Write a Program for implementation of 0/1 Knapsack problem using Dynamic Programming</td>
</tr>
<tr>
<td>8</td>
<td>Write a Program to find the shortest path from a given vertex to other vertices in a weighted connected graph using Dijkstra’s algorithm</td>
</tr>
<tr>
<td>9</td>
<td>Write a Program to find the minimum cost spanning tree (MST) of a given undirected graph using Kruskal’s algorithm</td>
</tr>
<tr>
<td>10</td>
<td>Write a Program to find the minimum cost spanning tree (MST) of a given undirected graph using Prim’s algorithm</td>
</tr>
</tbody>
</table>
COURSE OBJECTIVES:

• Implementation of various algorithms and to analyze the performance of algorithms.
• Demonstrate a familiarity with major algorithms and data structures.
• Apply important algorithmic design paradigms and methods of analysis.
• Synthesize efficient algorithms in common engineering design situations.

COURSE OUTCOMES:

• The course will help in improving the programming skills of the students.
• The design of algorithms for any problem will inculcate structured thinking process in the students and improve the analytical power.
PROGRAM 1 : Write a Program for iterative Binary Search

CODE

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

int binarySearch(vector<int>& arr, int target)
{
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target)
            return mid;
        if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1; // If target is not found
}

int main() {
    vector<int> arr = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int target = 23;
    int index = binarySearch(arr, target);
    if (index != -1)
        cout << "Element found at index: " << index << endl;
    else
        cout << "Element not found" << endl;
    return 0;
}
PROGRAM 2: Write a Program for recursive Binary Search

CODE

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

int binarySearch(vector<int>& arr, int target, int left, int right)
{
    if (left <= right)
    {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target)
            return mid;

        if (arr[mid] < target)
            return binarySearch(arr, target, mid + 1, right);
        else
            return binarySearch(arr, target, left, mid - 1);
    }
    return -1; // If target is not found
}

int main()
{
    vector<int> arr = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int target = 23;
    int index = binarySearch(arr, target, 0, arr.size() - 1);
    if (index != -1)
        cout << "Element found at index: " << index << endl;
    else
        cout << "Element not found" << endl;
    return 0;
}
```
PROGRAM 3: Write a Program to sort a given set of elements using the Quick Sort

CODE

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

// Function to partition the array
int partition(vector<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], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Function to implement Quick Sort
void quickSort(vector<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);
    }
}

int main()
{
    vector<int> arr = {12, 3, 9, 7, 14, 5, 23, 8};
    int n = arr.size();
    cout << "Original array: ";
    for (int num : arr)
    {
        cout << num << " ";
    }
    cout << endl;
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: ";
    for (int num : arr)
    {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```
PROGRAM 4: Write a Program to sort a given set of elements using the Merge Sort

CODE
#include <iostream>
#include <vector>
using namespace std;

// Function to merge two sorted subarrays into one sorted array
void merge(vector<int>& arr, int left, int mid, int right)
{
  int n1 = mid - left + 1;
  int n2 = right - mid;

  // Create temporary arrays
  vector<int> L(n1), R(n2);

  // Copy data to temporary arrays L[] and R[]
  for (int i = 0; i < n1; i++)
    L[i] = arr[left + i];
  for (int j = 0; j < n2; j++)
    R[j] = arr[mid + 1 + j];

  // Merge the temporary arrays back into arr[left..right]
  int i = 0, j = 0, k = left;
  while (i < n1 && j < n2)
  {
    if (L[i] <= R[j])
    {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = R[j];
      j++;
    }
    k++;
  }

  // Copy the remaining elements of L[], if there are any
  while (i < n1)
  {
    arr[k] = L[i];
    i++;
    k++;
  }

  // Copy the remaining elements of R[], if there are any
  while (j < n2)
  {
    arr[k] = R[j];
    j++;
    k++;
  }
}

// Function to implement Merge Sort
void mergeSort(vector<int>& arr, int left, int right)
{
  if (left < right)
```cpp
int mid = left + (right - left) / 2;
// Sort first and second halves
mergeSort(arr, left, mid);
mergeSort(arr, mid + 1, right);
// Merge the sorted halves
merge(arr, left, mid, right);
}

int main()
{
    vector<int> arr = {12, 3, 9, 7, 14, 5, 23, 8};
    int n = arr.size();
cout << "Original array: ";
for (int num : arr)
{
    cout << num << " ";
}
cout << endl;
mergeSort(arr, 0, n - 1);
cout << "Sorted array: ";
for (int num : arr)
{
    cout << num << " ";
}
cout << endl;
return 0;
}
```
PROGRAM 5: Write a Program to sort a given set of elements using the Selection Sort

CODE

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

void selectionSort(vector<int>& arr)
{
    int n = arr.size();
    for (int i = 0; i < n - 1; i++)
    {
        int minIndex = i;
        // Find the index of the minimum element in the unsorted part of the array
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }
        // Swap the minimum element with the first element of the unsorted part
        if (minIndex != i)
        {
            swap(arr[i], arr[minIndex]);
        }
    }
}

int main()
{
    vector<int> arr = {12, 3, 9, 7, 14, 5, 23, 8};
    int n = arr.size();
    cout << "Original array: ";
    for (int num : arr)
    {
        cout << num << " ";
    }
    cout << endl;
    selectionSort(arr);
    cout << "Sorted array: ";
    for (int num : arr)
    {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```
PROGRAM 6: Write a Program for implementation of Fractional Knapsack problem using Greedy Method

CODE
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Structure to represent items
struct Item
{
    int weight;
    int value;
};

// Comparison function for sorting items based on value per unit weight in descending order
bool compare(Item a, Item b)
{
    double ratio1 = (double)a.value / a.weight;
    double ratio2 = (double)b.value / b.weight;
    return ratio1 > ratio2;
}

// Function to solve the Fractional Knapsack problem
double fractionalKnapsack(vector<Item>& items, int capacity)
{
    // Sort items based on value per unit weight in descending order
    sort(items.begin(), items.end(), compare);
    double totalValue = 0.0;
    int remainingCapacity = capacity;
    // Iterate through sorted items and fill the knapsack
    for (int i = 0; i < items.size(); ++i)
    {
        if (remainingCapacity >= items[i].weight)
        {
            // If the entire item can be added to the knapsack
            totalValue += items[i].value;
            remainingCapacity -= items[i].weight;
        }
        else
        {
            // If only a fraction of the item can be added
            double fraction = (double)remainingCapacity / items[i].weight;
            totalValue += fraction * items[i].value;
            break; // Knapsack capacity is full
        }
    }
    return totalValue;
}

int main()
{
    vector<Item> items = {{10, 60}, {20, 100}, {30, 120}};
    int capacity = 50;
}
double maxValue = fractionalKnapsack(items, capacity);
cout << "Maximum value that can be obtained: " << maxValue << endl;
return 0;
}
**PROGRAM 7:** Write a Program for implementation of 0/1 Knapsack problem using Dynamic Programming

**CODE**
```cpp
#include <iostream>
#include <vector>
using namespace std;

// Function to solve 0/1 Knapsack problem using dynamic programming
int knapsack(vector<int>& values, vector<int>& weights, int capacity)
{
    int n = values.size();
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));
    for (int i = 1; i <= n; ++i)
    {
        for (int w = 1; w <= capacity; ++w)
        {
            if (weights[i - 1] <= w)
            {
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
            }
            else
            {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][capacity];
}

int main()
{
    vector<int> values = {60, 100, 120};
    vector<int> weights = {10, 20, 30};
    int capacity = 50;
    int maxValue = knapsack(values, weights, capacity);
    cout << "Maximum value that can be obtained: " << maxValue << endl;
    return 0;
}
```
PROGRAM 8: Write a Program to find the shortest path from a given vertex to other vertices in a weighted connected graph using Dijkstra’s algorithm

CODE:
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

// Structure to represent edges
struct Edge
{
    int destination;
    int weight;
};

// Structure to represent vertices
struct Vertex
{
    vector<Edge> adjList;
};

// Function to find the shortest path using Dijkstra's algorithm
vector<int> dijkstra(vector<Vertex>& graph, int source)
{
    int n = graph.size();
    vector<int> distance(n, INT_MAX);
    distance[source] = 0;
    // Min-heap to store vertices with their distances
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, source});
    while (!pq.empty())
    {
        int u = pq.top().second;
        pq.pop();
        for (const auto& edge : graph[u].adjList)
        {
            int v = edge.destination;
            int w = edge.weight;
            if (distance[v] > distance[u] + w)
            {
                distance[v] = distance[u] + w;
                pq.push({distance[v], v});
            }
        }
    }
    return distance;
}

int main()
{
    // Example graph
    int numVertices = 6;
    vector<Vertex> graph(numVertices);

/ Adding edges to the graph (0-indexed)

```cpp
graph[0].adjList.push_back({1, 2});
graph[0].adjList.push_back({2, 4});
graph[1].adjList.push_back({2, 1});
graph[1].adjList.push_back({3, 7});
graph[2].adjList.push_back({3, 3});
graph[2].adjList.push_back({4, 5});
graph[3].adjList.push_back({4, 1});
graph[3].adjList.push_back({5, 2});
graph[4].adjList.push_back({5, 6});
```

```cpp
int source = 0; // Source vertex
vector<int> shortestDistances = dijkstra(graph, source);
```

```cpp
// Output shortest distances from source to all other vertices
```

```cpp
cout << "Shortest distances from vertex " << source << " to all other vertices:" << endl;
for (int i = 0; i < shortestDistances.size(); ++i)
{
    cout << "Vertex " << i << ": " << shortestDistances[i] << endl;
}
return 0;
```
PROGRAM 9: Write a Program to find the minimum cost spanning tree (MST) of a given undirected graph using Kruskal’s algorithm

CODE
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Structure to represent edges
struct Edge {
    int source;
    int destination;
    int weight;
};

// Structure to represent disjoint sets for Union-Find
struct DisjointSets {
    vector<int> parent, rank;
    DisjointSets(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i)
            parent[i] = i;
    }
    int find(int u) {
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }
    void merge(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rank[rootX] > rank[rootY])
            parent[rootY] = rootX;
        else {
            parent[rootX] = rootY;
            if (rank[rootX] == rank[rootY])
                rank[rootY]++;
        }
    }
};

// Function to compare edges based on their weight
bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}
// Function to find the minimum spanning tree (MST) using Kruskal's algorithm
vector<Edge> kruskalMST(vector<Edge>& edges, int numVertices)
{
    vector<Edge> MST;
    DisjointSets ds(numVertices);
    // Sort edges based on their weight
    sort(edges.begin(), edges.end(), compareEdges);
    for (const auto& edge : edges)
    {
        int sourceRoot = ds.find(edge.source);
        int destinationRoot = ds.find(edge.destination);
        if (sourceRoot != destinationRoot)
        {
            MST.push_back(edge);
            ds.merge(sourceRoot, destinationRoot);
        }
    }
    return MST;
}

int main()
{
    // Example graph
    vector<Edge> edges = { {0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4} };
    int numVertices = 4;
    vector<Edge> MST = kruskalMST(edges, numVertices);
    cout << "Edges in the Minimum Spanning Tree (MST):" << endl;
    for (const auto& edge : MST)
    {
        cout << edge.source << " - " << edge.destination << " : " << edge.weight << endl;
    }
    return 0;
}
**PROGRAM 10**: Write a Program to find the minimum cost spanning tree (MST) of a given undirected graph using Prim’s algorithm

**CODE**
```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

// Structure to represent edges
struct Edge {
    int destination;
    int weight;
};

// Structure to represent vertices
struct Vertex {
    vector<Edge> adjList;
};

// Function to find the minimum spanning tree (MST) using Prim's algorithm
vector<Edge> primMST(vector<Vertex>& graph) {
    int numVertices = graph.size();
    vector<bool> visited(numVertices, false);
    vector<Edge> MST;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    // Start from vertex 0
    pq.push({0, 0});
    while (!pq.empty()) {
        int u = pq.top().second;
        int weight = pq.top().first;
        pq.pop();
        // If the vertex has already been visited, skip it
        if (visited[u])
            continue;
        // Mark vertex as visited
        visited[u] = true;
        // Add the edge to the MST
        if (u != 0)
            MST.push_back({u, weight});
        // Add adjacent vertices to the priority queue
        for (const auto& edge : graph[u].adjList)
            if (!visited[edge.destination])
                pq.push({edge.weight, edge.destination});
    }
    return MST;
}
```
return MST;
}
int main()
{
    // Example graph
    vector<Vertex> graph = {
        {{{1, 2}, {2, 1}, {3, 3}}, {{0, 2}, {2, 2}, {4, 4}}},
        {{{0, 1}, {1, 2}, {3, 1}, {4, 3}}, {{0, 3}, {2, 1}, {4, 2}}},
        {{{1, 4}, {2, 3}, {3, 2}}} };  
    vector<Edge> MST = primMST(graph);
    cout << "Edges in the Minimum Spanning Tree (MST):" << endl;
    for (const auto& edge : MST)
    {
        cout << edge.destination << " - " << edge.weight << endl;
    }
    return 0;
}