INDEX

SL No.	Questions	Page No.
1	Implement binary search procedure using divide and conquer method.	
2	Implement divide and conquer method for finding the maximum and minimum number.	
3	Write a program to measure the performance using time function between bubble sort and quick sort.	
4	Implement the fractional knapsack problem that will generate an optimal solution for the given set of instance.	
5	Implement the 0/1 knapsack problem that will generate an optimal solutions for the given set of instance such as no of elements n, cost p, and weight wi 5.	
6	Write a program to find the minimum cost spanning tree using Prim's algorithm.	
7	Write a program to find the single source shortest path.	
8	Write a program to implement dynamic programming method for all pair's shortest path problem.	
9	Using backtracking algorithm implement N-queens problem.	
10	Write a program for coloring a graph.	

1. **Implement binary search procedure using divide and conquer method**

Program:

```c
#include <stdio.h>

#define MAX_SIZE 100

// Binary search function
int binarySearch(int arr[], int left, int right, int target) {
    if (right >= left) {
        int mid = left + (right - left) / 2;

        // If the target element is present at the middle
        if (arr[mid] == target)
            return mid;

        // If the target element is smaller than the middle element, search in the left subarray
        if (arr[mid] > target)
            return binarySearch(arr, left, mid - 1, target);

        // If the target element is greater than the middle element, search in the right subarray
        return binarySearch(arr, mid + 1, right, target);
    }

    // Element is not present in the array
    return -1;
}

// Bubble sort function
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Main function
int main() {
    int n, target;
```

```c
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    if (n > MAX_SIZE) {
        printf("Array size exceeds the maximum limit.\n");
        return 1;
    }

    int arr[MAX_SIZE];

    printf("Enter the elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Sort the array using bubble sort
    bubbleSort(arr, n);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    printf("Enter the target element to search: ");
    scanf("%d", &target);

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

    if (result == -1)
        printf("Element not found in the array.\n");
    else
        printf("Element found at index %d.\n", result);

    return 0;
}
```

Output:

```
Enter the number of elements in the array: 8
Enter the elements of the array:
12 45 234  657 234 4 2345 23
Sorted array: 4 12 23 45 234 234 657 2345
Enter the target element to search: 23
Element found at index 2.
```

2. Implement divide and conquer method for finding the maximum and minimum number.

Program:

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

// Function to find the maximum and minimum number using divide and conquer
void findMaxMin(int arr[], int left, int right, int& maxNum, int& minNum) {
    if (left == right) {
        // Only one element in the array
        maxNum = arr[left];
        minNum = arr[left];
    } else if (left == right - 1) {
        // Two elements in the array
        if (arr[left] < arr[right]) {
            maxNum = arr[right];
            minNum = arr[left];
        } else {
            maxNum = arr[left];
            minNum = arr[right];
        }
    } else {
        // More than two elements in the array
        int mid = (left + right) / 2;
        int maxNum1, maxNum2, minNum1, minNum2;

        // Divide the array into two halves and recursively find the maximum and
minimum in each half
        findMaxMin(arr, left, mid, maxNum1, minNum1);
        findMaxMin(arr, mid + 1, right, maxNum2, minNum2);

        // Combine the results from the two halves to get the overall maximum and
minimum
        maxNum = (maxNum1 > maxNum2) ? maxNum1 : maxNum2;
        minNum = (minNum1 < minNum2) ? minNum1 : minNum2;
    }
}

// Main function
int main() {
    int n;

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

    int arr[n];
```

```
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int maxNum = arr[0];
    int minNum = arr[0];
    findMaxMin(arr, 0, n - 1, maxNum, minNum);

    cout << "Maximum number: " << maxNum << endl;
    cout << "Minimum number: " << minNum << endl;

    return 0;
}
```

Output:

```
Enter the number of elements in the array: 7
Enter the elements of the array: 3 2 5 7 4 2 1
Maximum number: 7
Minimum number: 1
```

3. **Write a program to measure the performance using time function between bubble sort and quick sort.**

Program:

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

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Bubble sort function
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}

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

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

    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// Quick sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
```

```
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}

// Function to generate random numbers in the array
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
}

// Function to measure the performance of sorting algorithms
void measurePerformance(int n) {
    int* arr1 = new int[n];
    int* arr2 = new int[n];

    generateRandomArray(arr1, n);
    for (int i = 0; i < n; i++) {
        arr2[i] = arr1[i];
    }

    clock_t start, end;
    double timeTaken;

    // Measure time for bubble sort
    start = clock();
    bubbleSort(arr1, n);
    end = clock();
    timeTaken = double(end - start) / CLOCKS_PER_SEC;
    cout << "Bubble Sort:\n";
    cout << "Time taken: " << timeTaken * 1000 << " milliseconds (" << timeTaken <<
" seconds)\n\n";

    // Measure time for quick sort
    start = clock();
    quickSort(arr2, 0, n - 1);
    end = clock();
    timeTaken = double(end - start) / CLOCKS_PER_SEC;
    cout << "Quick Sort:\n";
    cout << "Time taken: " << timeTaken * 1000 << " milliseconds (" << timeTaken <<
" seconds)\n";

    delete[] arr1;
    delete[] arr2;
}
```

```cpp
// Main function
int main() {
    int n;

    cout << "Enter the size of the array: ";
    cin >> n;

    measurePerformance(n);

    return 0;
}
```

Output:

```
Enter the size of the array: 100000
Bubble Sort:
Time taken: 20844 milliseconds (20.844 seconds)

Quick Sort:
Time taken: 187 milliseconds (0.187 seconds)
```

4. **Implement the fractional knapsack problem that will generate an optimal solution for the given set of instance.**

Program:

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

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

// Function to compare items based on their value per unit weight
bool compareItems(const Item& item1, const Item& item2) {
    double valuePerUnitWeight1 = double(item1.value) / item1.weight;
    double valuePerUnitWeight2 = double(item2.value) / item2.weight;
    return valuePerUnitWeight1 > valuePerUnitWeight2;
}

// Function to calculate the maximum achievable value
double fractionalKnapsack(int capacity, vector<Item>& items) {
    // Sort items based on value per unit weight
    sort(items.begin(), items.end(), compareItems);

    double totalValue = 0.0;  // Total value achieved
    int currentWeight = 0;    // Current weight in the knapsack

    // Iterate through each item
    for (const Item& item : items) {
        // Check if the current item can be fully included
        if (currentWeight + item.weight <= capacity) {
            currentWeight += item.weight;
            totalValue += item.value;
        }
        // If the current item cannot be fully included, include a fraction of it
        else {
            int remainingCapacity = capacity - currentWeight;
            double fraction = double(remainingCapacity) / item.weight;
            currentWeight += remainingCapacity;
            totalValue += fraction * item.value;
            break;  // Knapsack is full, so exit the loop
        }
    }
```

```
    return totalValue;
}

// Main function
int main() {
    int numItems;
    cout << "Enter the number of items: ";
    cin >> numItems;

    vector<Item> items(numItems);

    cout << "Enter the weight and value of each item:\n";
    for (int i = 0; i < numItems; i++) {
        cout << "Item " << i + 1 << ":\n";
        cout << "Weight: ";
        cin >> items[i].weight;
        cout << "Value: ";
        cin >> items[i].value;
    }

    int capacity;
    cout << "Enter the capacity of the knapsack: ";
    cin >> capacity;

    double maxValue = fractionalKnapsack(capacity, items);

    cout << "Maximum achievable value: " << maxValue << endl;

    return 0;
}
```

Output:

```
Enter the number of items: 4
Enter the weight and value of each item:
Item 1:
Weight: 5
Value: 10
Item 2:
Weight: 3
Value: 8
Item 3:
Weight: 2
Value: 6
Item 4:
Weight: 7
Value: 12
Enter the capacity of the knapsack: 10
Maximum achievable value: 24
```

5. **Implement the 0/1 knapsack problem that will generate an optimal solutions for the given set of instance such as no of elements n, cost p, and weight wi 5.**

Program:

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

// Structure to represent an item
struct Item {
    int cost;
    int weight;
};

// Function to calculate the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve the 0/1 knapsack problem
int knapsack(int capacity, vector<Item>& items) {
    int n = items.size();

    // Create a table to store the maximum values
    vector<vector<int>> table(n + 1, vector<int>(capacity + 1, 0));

    // Fill the table using dynamic programming approach
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= capacity; j++) {
            if (items[i - 1].weight <= j) {
                table[i][j] = max(items[i - 1].cost + table[i - 1][j - items[i -
1].weight], table[i - 1][j]);
            } else {
                table[i][j] = table[i - 1][j];
            }
        }
    }

    return table[n][capacity];
}

// Main function
int main() {
    int n;
    cout << "Enter the number of items: ";
    cin >> n;
```

```
    vector<Item> items(n);

    cout << "Enter the cost and weight of each item:\n";
    for (int i = 0; i < n; i++) {
        cout << "Item " << i + 1 << ":\n";
        cout << "Cost: ";
        cin >> items[i].cost;
        cout << "Weight: ";
        cin >> items[i].weight;
    }

    int capacity;
    cout << "Enter the capacity of the knapsack: ";
    cin >> capacity;

    int maxValue = knapsack(capacity, items);

    cout << "Maximum achievable value: " << maxValue << endl;

    return 0;
}
```

Output:

```
Enter the number of items: 5
Enter the cost and weight of each item:
Item 1:
Cost: 10
Weight: 5
Item 2:
Cost: 20
Weight: 10
Item 3:
Cost: 30
Weight: 15
Item 4:
Cost: 40
Weight: 20
Item 5:
Cost: 50
Weight: 25
Enter the capacity of the knapsack: 50
Maximum achievable value: 100
```

6. **Write a program to find the minimum cost spanning tree using Prim's algorithm.**

Program:

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
using namespace std;

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

// Function to find the minimum cost spanning tree using Prim's algorithm
void primMST(vector<vector<int>>& graph) {
    int numVertices = graph.size();

    // Create a vector to store the parent of each vertex in the MST
    vector<int> parent(numVertices, -1);

    // Create a vector to store the minimum weight of each vertex in the MST
    vector<int> minWeight(numVertices, numeric_limits<int>::max());

    // Create a vector to store whether a vertex is included in the MST
    vector<bool> inMST(numVertices, false);

    // Start with the first vertex as the root
    int root = 0;
    minWeight[root] = 0;

    // Create a priority queue to store the vertices based on their minimum weight
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    // Insert the root vertex into the priority queue
    pq.push(make_pair(0, root));

    // Loop through all vertices
    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

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

        // Update the minimum weight and parent for its adjacent vertices
        for (int v = 0; v < numVertices; v++) {
            if (graph[u][v] != 0 && !inMST[v] && graph[u][v] < minWeight[v]) {
```

```
                parent[v] = u;
                minWeight[v] = graph[u][v];
                pq.push(make_pair(minWeight[v], v));
            }
        }
    }

    // Print the minimum cost spanning tree
    cout << "Minimum Cost Spanning Tree:\n";
    for (int i = 1; i < numVertices; i++) {
        cout << "Edge: " << parent[i] << " - " << i << " \tWeight: " << graph[parent[i]][i]
<< endl;
    }
}

// Main function
int main() {
    int numVertices;
    cout << "Enter the number of vertices: ";
    cin >> numVertices;

    // Create an adjacency matrix to represent the graph
    vector<vector<int>> graph(numVertices, vector<int>(numVertices));

    cout << "Enter the weight of the edges:\n";
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            cin >> graph[i][j];
        }
    }

    primMST(graph);

    return 0;
}
```

Output:

```
Enter the number of vertices: 4
Enter the weight of the edges:
0 2 0 6
2 0 3 8
0 3 0 0
6 8 0 0
Minimum Cost Spanning Tree:
Edge: 0 - 1     Weight: 2
Edge: 1 - 2     Weight: 3
Edge: 0 - 3     Weight: 6
```

7. Write a program to find the single source shortest path

Program:

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
using namespace std;

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

// Function to find the single source shortest path using Dijkstra's algorithm
void dijkstraShortestPath(vector<vector<Edge>>& graph, int source) {
    int numVertices = graph.size();

    // Create a vector to store the distances from the source vertex
    vector<int> distance(numVertices, numeric_limits<int>::max());

    // Create a priority queue to store the vertices based on their distances
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    // Start with the source vertex
    distance[source] = 0;
    pq.push(make_pair(0, source));

    // Loop through all vertices
    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        // Update the distances for the adjacent vertices
        for (const auto& edge : graph[u]) {
            int v = edge.destination;
            int weight = edge.weight;

            if (distance[u] + weight < distance[v]) {
                distance[v] = distance[u] + weight;
                pq.push(make_pair(distance[v], v));
            }
        }
    }

    // Print the shortest distances from the source vertex
    cout << "Shortest Distances from Source Vertex " << source << ":\n";
    for (int i = 0; i < numVertices; i++) {
```

```
        cout << "Vertex " << i << ": " << distance[i] << endl;
    }
}

// Main function
int main() {
    int numVertices;
    cout << "Enter the number of vertices: ";
    cin >> numVertices;

    // Create a vector of vectors to represent the graph
    vector<vector<Edge>> graph(numVertices);

    cout << "Enter the number of edges: ";
    int numEdges;
    cin >> numEdges;

    cout << "Enter the edges and their weights:\n";
    for (int i = 0; i < numEdges; i++) {
        int source, destination, weight;
        cin >> source >> destination >> weight;
        graph[source].push_back({destination, weight});
    }

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

    dijkstraShortestPath(graph, sourceVertex);

    return 0;
}
```

Output:

```
Enter the number of vertices: 5
Enter the number of edges: 6
Enter the edges and their weights:
0 1 4
0 2 2
1 2 1
1 3 5
2 3 8
3 4 3
Enter the source vertex: 0
Shortest Distances from Source Vertex 0:
Vertex 0: 0
Vertex 1: 4
Vertex 2: 2
Vertex 3: 9
Vertex 4: 12
```

8. **Write a program to implement dynamic programming method for all pair's shortest path problem.**

Problem:

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

#define nV 4
#define INF 999

void printShortestDistances(int distances[][nV]);

void floydWarshall(int graph[][nV]) {
  int distances[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      distances[i][j] = graph[i][j];

  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (distances[i][k] + distances[k][j] < distances[i][j])
          distances[i][j] = distances[i][k] + distances[k][j];
      }
    }
  }
  printShortestDistances(distances);
}

void printShortestDistances(int distances[][nV]) {
  cout << "Shortest Distances:\n";
  for (int i = 0; i < nV; i++) {
    for (int j = 0; j < nV; j++) {
      if (distances[i][j] == INF)
        cout << "INF\t";
      else
        cout << distances[i][j] << "\t";
    }
    cout << endl;
  }
}

int main() {
  int graph[nV][nV] = {{0, 3, INF, 5},
                       {2, 0, INF, 4},
                       {INF, 1, 0, INF},
```

```
                      {INF, INF, 2, 0}};
  floydWarshall(graph);

  return 0;
}
```

Output:

```
Shortest Distances:
0      3      7      5
2      0      6      4
3      1      0      5
5      3      2      0
```

9. Using backtracking algorithm implement N-queens problem.

Program:

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

// Function to check if a queen can be placed at the given position
bool isSafe(vector<vector<int>>& board, int row, int col, int N) {
    // Check if there is a queen in the same column
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 1) {
            return false;
        }
    }

    // Check if there is a queen in the upper-left diagonal
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    // Check if there is a queen in the upper-right diagonal
    for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    return true;
}

// Function to solve the N-Queens problem using backtracking
bool solveNQueensUtil(vector<vector<int>>& board, int row, int N) {
    if (row == N) {
        // All queens have been placed successfully
        return true;
    }

    for (int col = 0; col < N; col++) {
        if (isSafe(board, row, col, N)) {
            board[row][col] = 1;  // Place the queen

            // Recursively solve the problem for the next row
            if (solveNQueensUtil(board, row + 1, N)) {
                return true;
```

```
            }

            board[row][col] = 0;  // Backtrack and remove the queen
        }
    }

    return false;  // No solution found
}

// Function to solve the N-Queens problem and print the board
void solveNQueens(int N) {
    vector<vector<int>> board(N, vector<int>(N, 0));

    if (solveNQueensUtil(board, 0, N)) {
        // Solution found, print the board
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                cout << board[i][j] << " ";
            }
            cout << endl;
        }
    } else {
        cout << "No solution found for N = " << N << endl;
    }
}

int main() {
    int N;
    cout << "Enter the value of N: ";
    cin >> N;

    solveNQueens(N);

    return 0;
}
```

Output:

```
Enter the value of N: 4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```

10. Write a program for coloring a graph.

Program:

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

// Function to check if it's safe to color a vertex with a particular color
bool isSafe(vector<vector<int>>& graph, vector<int>& colors, int vertex, int color)
{
    for (int i = 0; i < graph.size(); i++) {
        if (graph[vertex][i] && colors[i] == color) {
            return false;
        }
    }
    return true;
}

// Function to color the graph using backtracking
bool colorGraphUtil(vector<vector<int>>& graph, vector<int>& colors, int vertex, int
numColors) {
    if (vertex == graph.size()) {
        return true; // All vertices have been colored successfully
    }

    for (int color = 1; color <= numColors; color++) {
        if (isSafe(graph, colors, vertex, color)) {
            colors[vertex] = color; // Assign color to the vertex

            if (colorGraphUtil(graph, colors, vertex + 1, numColors)) {
                return true;
            }

            colors[vertex] = 0; // Backtrack and try a different color
        }
    }

    return false; // No valid coloring found
}

// Function to color the graph and print the colors assigned to each vertex
void colorGraph(vector<vector<int>>& graph, int numColors) {
    vector<int> colors(graph.size(), 0);

    if (colorGraphUtil(graph, colors, 0, numColors)) {
        cout << "Graph can be colored using " << numColors << " colors." << endl;
        cout << "Colors assigned to vertices:" << endl;
```

```
        for (int i = 0; i < colors.size(); i++) {
            cout << "Vertex " << i << ": Color " << colors[i] << endl;
        }
    } else {
        cout << "Graph cannot be colored using " << numColors << " colors." << endl;
    }
}

int main() {
    int numVertices, numColors;

    cout << "Enter the number of vertices in the graph: ";
    cin >> numVertices;

    cout << "Enter the number of colors available: ";
    cin >> numColors;

    vector<vector<int>> graph(numVertices, vector<int>(numVertices, 0));

    cout << "Enter the adjacency matrix representing the graph:" << endl;
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            cin >> graph[i][j];
        }
    }

    colorGraph(graph, numColors);

    return 0;
}
```

Output:

```
Enter the number of vertices in the graph: 4
Enter the number of colors available: 3
Enter the adjacency matrix representing the graph:
0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0
Graph can be colored using 3 colors.
Colors assigned to vertices:
Vertex 0: Color 1
Vertex 1: Color 2
Vertex 2: Color 3
Vertex 3: Color 1
```