VISVESVARAYA TECHNOLOGICAL UNIVERSITY
JNANA SANGAMA, BELAGAVI, KARNATAKA 590018

LAB MANUAL

DESIGN AND ANALYSIS OF ALGORITHMS (21CS42)

Prepared by
Ms. Vinotha D
Professor, Dept. of CSE
The Oxford College of Engineering
Bangalore

विद्या सर्वत्र शोभते
Estd. 1974

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
THE OXFORD COLLEGE OF ENGINEERING
Hosur Road, Bommanahalli, Bangalore-560068

THE OXFORD COLLEGE OF ENGINEERING

Bommanahalli, Hosur Road, Bangalore – 560068
(Affiliated To Visvesvaraya Technological University, Belagavi)

DEPARTMENT VISION

To produce technocrats with creative technical knowledge and intellectual skills to sustain and excel in the highly demanding world with confidence.

DEPARTMENT MISSION

M1. To Produce the Best Computer Science Professionals with intellectual skills.

M2. To Provide a Vibrant Ambiance that promotes Creativity, Technology Competent and Innovations for the new Era.

M3. To Pursue Professional Excellence with Ethical and Moral Values to sustain in the highly demanding world.

THE OXFORD COLLEGE OF ENGINEERING

Bommanahalli, Hosur Road, Bangalore – 560068
(Affiliated To Visvesvaraya Technological University, Belagavi)

SYLLABUS

Module 1		
1	Sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the brute force method works along with its time complexity analysis: worst case, average case and best case.	**1**

Module 2		
1	Sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.	3
2	Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.	6

Module 5		
1	Design and implement C++/Java Program to find a subset of a given set S = {Sl, S2,..., Sn} of n positive integers whose SUM is equal to a given positive integer d. For example, if S = {1, 2, 5, 6, 8} and d= 9, there are two solutions {1, 2, 6} and {1, 8}. Display a suitable message, if the given problem instance doesn't have a solution.	**23**
2	Design and implement C++/Java Program to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle.	**25**

Module 1

Program 1

Sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the brute force method works along with its time complexity analysis: worst case, average case and best case.

```
import java.util.Arrays;

public class SelectionSort {
  public static void selectionSort(int[] arr) {
    int n = arr.length;

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

      for (int j = i + 1; j < n; j++) {
        if (arr[j] < arr[minIndex]) {
          minIndex = j;
        }
      }

      int temp = arr[minIndex];
      arr[minIndex] = arr[i];
      arr[i] = temp;
    }
  }

  public static void main(String[] args) {
    int[] sizes = {1000, 2000, 3000, 4000, 5000};

    for (int size : sizes) {
      int[] arr = generateRandomArray(size);

      long startTime = System.nanoTime();

      selectionSort(arr);

      long endTime = System.nanoTime();
      long duration = endTime - startTime;

      System.out.println("Sorted array of size " + size + " in " + duration + "
nanoseconds.");
    }
  }
```

```java
    public static int[] generateRandomArray(int size) {
        int[] arr = new int[size];
        Random rand = new Random();

        for (int i = 0; i < size; i++) {
            arr[i] = rand.nextInt(1000); // Generate random integers between 0 and 999
        }

        return arr;
    }
}
```

Module 2

Program 1

Sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.

```
import java.util.Arrays;
import java.util.Random;

public class QuickSort {
   /**
    * Sorts an integer array using the QuickSort algorithm.
    *
    * @param arr The array to be sorted
    */
   public static void quickSort(int[] arr) {
      quickSort(arr, 0, arr.length - 1);
   }

   /**
    * Recursive helper method to perform QuickSort on a subarray.
    *
    * @param arr  The array to be sorted
    * @param low  The starting index of the subarray
    * @param high The ending index of the subarray
    */
   private static void quickSort(int[] arr, int low, int high) {
      if (low < high) {
         int partitionIndex = partition(arr, low, high);

         // Recursively sort the subarrays before and after the partition
         quickSort(arr, low, partitionIndex - 1);
         quickSort(arr, partitionIndex + 1, high);
      }
   }

   /**
    * Partitions the array by selecting a pivot element and rearranging the elements
    * such that elements smaller than the pivot are placed to the left, and elements
    * greater than the pivot are placed to the right.
    *
    * @param arr  The array to be partitioned
    * @param low  The starting index of the subarray
```

```
     * @param high The ending index of the subarray
     * @return The partition index
     */
    private static 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, j);
            }
        }

        swap(arr, i + 1, high);

        return i + 1;
    }

    /**
     * Swaps two elements in an array.
     *
     * @param arr The array containing the elements
     * @param i  The index of the first element
     * @param j  The index of the second element
     */
    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        int[] sizes = {1000, 2000, 3000, 4000, 5000};

        // Iterate over different array sizes
        for (int size : sizes) {
            int[] arr = generateRandomArray(size);

            long startTime = System.nanoTime();

            quickSort(arr);

            long endTime = System.nanoTime();
            long duration = endTime - startTime;

            System.out.println("Sorted array of size " + size + " in " + duration + "
nanoseconds.");
        }
    }
```

```
    /**
     * Generates a random integer array of the specified size.
     *
     * @param size The size of the array
     * @return The generated random array
     */
    public static int[] generateRandomArray(int size) {
        int[] arr = new int[size];
        Random rand = new Random();

        for (int i = 0; i < size; i++) {
            arr[i] = rand.nextInt(1000); // Generate random integers between 0 and 999
        }

        return arr;
    }
}
```

Program 2

Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.

```
import java.util.Arrays;
import java.util.Random;

public class MergeSort {
   /**
    * Sorts an integer array using the MergeSort algorithm.
    *
    * @param arr The array to be sorted
    */
   public static void mergeSort(int[] arr) {
      mergeSort(arr, 0, arr.length - 1);
   }

   /**
    * Recursive helper method to perform MergeSort on a subarray.
    *
    * @param arr  The array to be sorted
    * @param low  The starting index of the subarray
    * @param high The ending index of the subarray
    */
   private static void mergeSort(int[] arr, int low, int high) {
      if (low < high) {
         int mid = (low + high) / 2;

         // Recursively sort the subarrays before and after the middle index
         mergeSort(arr, low, mid);
         mergeSort(arr, mid + 1, high);

         // Merge the sorted subarrays
         merge(arr, low, mid, high);
      }
   }

   /**
    * Merges two sorted subarrays into a single sorted subarray.
    *
    * @param arr  The array containing the subarrays
    * @param low  The starting index of the first subarray
    * @param mid  The ending index of the first subarray
```

```
 * @param high The ending index of the second subarray
 */
private static void merge(int[] arr, int low, int mid, int high) {
    int n1 = mid - low + 1;
    int n2 = high - mid;

    int[] left = new int[n1];
    int[] right = new int[n2];

    // Copy elements from the original array into temporary arrays
    for (int i = 0; i < n1; i++) {
        left[i] = arr[low + i];
    }

    for (int j = 0; j < n2; j++) {
        right[j] = arr[mid + 1 + j];
    }

    int i = 0;
    int j = 0;
    int k = low;

    // Merge the two subarrays by comparing elements
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    // Copy any remaining elements from the left subarray
    while (i < n1) {
        arr[k] = left[i];
        i++;
        k++;
    }

    // Copy any remaining elements from the right subarray
    while (j < n2) {
        arr[k] = right[j];
        j++;
        k++;
    }
}

public static void main(String[] args) {
    int[] sizes = {1000, 2000, 3000, 4000, 5000};
```

```
        // Iterate over different array sizes
        for (int size : sizes) {
            int[] arr = generateRandomArray(size);

            long startTime = System.nanoTime();

            mergeSort(arr);

            long endTime = System.nanoTime();
            long duration = endTime - startTime;

            System.out.println("Sorted array of size " + size + " in " + duration + "
nanoseconds.");
        }
    }

    /**
     * Generates a random integer array of the specified size.
     *
     * @param size The size of the array
     * @return The generated random array
     */
    public static int[] generateRandomArray(int size) {
        int[] arr = new int[size];
        Random rand = new Random();

        for (int i = 0; i < size; i++) {
            arr[i] = rand.nextInt(1000); // Generate random integers between 0 and 999
        }

        return arr;
    }
}
```

Module 3

Program 1

Write & Execute C++/Java Program to solve Knapsack problem using Greedy method.

```
import java.util.Arrays;

public class KnapsackGreedy {
    // Class representing an item with weight and value
    static class Item implements Comparable<Item> {
        int weight;
        int value;

        public Item(int weight, int value) {
            this.weight =  weight;
            this.value = value;
        }

        // Compare items based on their value-to-weight ratio
        public int compareTo(Item other) {
            double ratio1 = (double) value / weight;
            double ratio2 = (double) other.value / other.weight;
            if (ratio1 > ratio2) {
                return -1;
            } else if (ratio1 < ratio2) {
                return 1;
            } else {
                return 0;
            }
        }
    }

    // Method to solve the knapsack problem using a greedy approach
    public static int knapsackGreedy(int[] weights, int[] values, int capacity) {
        int n = weights.length;
        Item[] items = new Item[n];

        // Create an array of items with their respective weights and values
        for (int i = 0; i < n; i++) {
            items[i] = new Item(weights[i], values[i]);
        }

        // Sort the items in descending order based on their value-to-weight ratio
        Arrays.sort(items);

        int totalValue = 0;
        int currentWeight = 0;
```

```
        int i = 0;

        // Add items to the knapsack until the capacity is reached or all items are
considered
        while (currentWeight < capacity && i < n) {
            if (currentWeight + items[i].weight <= capacity) {
                // If the current item can be fully added, add its value and update the current
weight
                currentWeight += items[i].weight;
                totalValue += items[i].value;
            } else {
                // If the current item cannot be fully added, calculate the remaining weight
                // that can be added proportionally and update the total value and current
weight
                int remainingWeight = capacity - currentWeight;
                totalValue += (int) ((double) items[i].value / items[i].weight *
remainingWeight);
                currentWeight = capacity;
            }
            i++;
        }

        return totalValue;
    }

    public static void main(String[] args) {
        int[] weights = {10, 20, 30};
        int[] values = {60, 100, 120};
        int capacity = 50;

        int maxTotalValue = knapsackGreedy(weights, values, capacity);
        System.out.println("Maximum total value: " + maxTotalValue);
    }
}
```

Program 2

Write & Execute C++/Java Program to find shortest paths to other vertices from a given vertex in a weighted connected graph, using Dijkstra's algorithm.

```
import java.util.Arrays;
import java.util.PriorityQueue;

public class DijkstraShortestPaths {
    static class Edge {
        int target;
        int weight;

        public Edge(int target, int weight) {
            this.target = target;
            this.weight = weight;
        }
    }

    // Method to find the shortest paths from a given source vertex to all other vertices
in a graph
    public static void shortestPaths(int[][] graph, int source) {
        int n = graph.length;
        int[] distances = new int[n];
        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[source] = 0;

        // Use a priority queue to store edges based on their weights
        PriorityQueue<Edge> pq = new PriorityQueue<>((a, b) -> a.weight - b.weight);
        pq.offer(new Edge(source, 0));

        while (!pq.isEmpty()) {
            Edge edge = pq.poll();
            int u = edge.target;

            // Explore all neighboring vertices of the current vertex
            for (int v = 0; v < n; v++) {
                if (graph[u][v] != 0) {
                    int newDistance = distances[u] + graph[u][v];
                    if (newDistance < distances[v]) {
                        // If a shorter path is found, update the distance and add the edge to
the priority queue
                        distances[v] = newDistance;
                        pq.offer(new Edge(v, newDistance));
                    }
                }
            }
        }

        // Print the shortest paths from the source vertex to all other vertices
```

```
        System.out.println("Shortest paths from vertex " + source + ":");
        for (int i = 0; i < n; i++) {
            System.out.println("Vertex " + i + ": " + distances[i]);
        }
    }

    public static void main(String[] args) {
        int[][] graph = {
            {0, 4, 2, 0, 0},
            {4, 0, 1, 5, 0},
            {2, 1, 0, 8, 10},
            {0, 5, 8, 0, 2},
            {0, 0, 10, 2, 0}
        };
        int source = 0;

        shortestPaths(graph, source);
    }
}
```

Program 3

Write & Execute C++/Java Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm. Use Union-Find algorithms in your program.

```
import java.util.Arrays;

public class KruskalMinimumSpanningTree {
    static class Edge implements Comparable<Edge> {
        int source;
        int target;
        int weight;

        public Edge(int source, int target, int weight) {
            this.source = source;
            this.target = target;
            this.weight = weight;
        }

        public int compareTo(Edge other) {
            return this.weight - other.weight;
        }
    }

    static class UnionFind {
        int[] parent;
        int[] rank;

        public UnionFind(int n) {
            parent = new int[n];
            rank = new int[n];
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 0;
            }
        }

        public int find(int x) {
            if (parent[x] != x) {
                parent[x] = find(parent[x]);
            }
            return parent[x];
        }

        public void union(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);

            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
```

```
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }

    // Method to find the minimum spanning tree using Kruskal's algorithm
    public static void minimumSpanningTree(int[][] graph) {
        int n = graph.length;

        // Create an array to store all edges in the graph
        Edge[] edges = new Edge[n * n];
        int count = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (graph[i][j] != 0) {
                    edges[count] = new Edge(i, j, graph[i][j]);
                    count++;
                }
            }
        }

        // Sort the edges array based on their weights
        Arrays.sort(edges, 0, count);

        // Create an array to store the edges in the minimum spanning tree
        Edge[] mst = new Edge[n - 1];
        int mstCount = 0;

        // Create a UnionFind data structure to track connected components
        UnionFind uf = new UnionFind(n);

        // Iterate over the sorted edges
        for (int i = 0; i < count; i++) {
            Edge edge = edges[i];
            int source = edge.source;
            int target = edge.target;

            // Check if adding this edge creates a cycle
            if (uf.find(source) != uf.find(target)) {
                uf.union(source, target);
                mst[mstCount] = edge;
                mstCount++;
            }
        }

        // Print the minimum cost spanning tree
        System.out.println("Minimum Cost Spanning Tree:");
```

```
        for (int i = 0; i < mstCount; i++) {
            Edge edge = mst[i];
            System.out.println(edge.source + " - " + edge.target + " : " + edge.weight);
        }
    }

    public static void main(String[] args) {
        int[][] graph = {
            {0, 2, 0, 6, 0},
            {2, 0, 3, 8, 5},
            {0, 3, 0, 0, 7},
            {6, 8, 0, 0, 9},
            {0, 5, 7, 9, 0}
        };

        minimumSpanningTree(graph);
    }
}
```

Program 4

Write & Execute C++/Java Program To find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

```
import java.util.Arrays;

public class PrimMinimumSpanningTree {
    // Method to find the minimum spanning tree using Prim's algorithm
    public static void minimumSpanningTree(int[][] graph) {
        int n = graph.length;

        // Create arrays to store parent vertices, key values, and MST set
        int[] parent = new int[n];
        int[] key = new int[n];
        boolean[] mstSet = new boolean[n];

        // Initialize key values as infinity and parent values as -1
        Arrays.fill(key, Integer.MAX_VALUE);
        key[0] = 0;
        parent[0] = -1;

        // Construct the MST with (n-1) edges
        for (int count = 0; count < n - 1; count++) {
            // Find the vertex with the minimum key value among the vertices not yet in the
MST
            int u = minKey(key, mstSet);

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

            // Update key values and parent values of adjacent vertices of the selected
vertex
            for (int v = 0; v < n; v++) {
                if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {
                    parent[v] = u;
                    key[v] = graph[u][v];
                }
            }
        }

        // Print the minimum cost spanning tree
        System.out.println("Minimum Cost Spanning Tree:");
        for (int i = 1; i < n; i++) {
            System.out.println(parent[i] + " - " + i + " : " + graph[i][parent[i]]);
        }
    }

    // Method to find the vertex with the minimum key value among the vertices not yet
in the MST
```

```
    public static int minKey(int[] key, boolean[] mstSet) {
        int min = Integer.MAX_VALUE;
        int minIndex = -1;

        for (int v = 0; v < key.length; v++) {
            if (!mstSet[v] && key[v] < min) {
                min = key[v];
                minIndex = v;
            }
        }

        return minIndex;
    }

    public static void main(String[] args) {
        int[][] graph = {
            {0, 2, 0, 6, 0},
            {2, 0, 3, 8, 5},
            {0, 3, 0, 0, 7},
            {6, 8, 0, 0, 9},
            {0, 5, 7, 9, 0}
        };

        minimumSpanningTree(graph);
    }
}
```

Module 4

Program 1

Write & Execute C++/Java Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm.

```
public class FloydAllPairsShortestPaths {
    public static void floydWarshall(int[][] graph) {
        int n = graph.length;
        int[][] dist = new int[n][n];

        // Initialize the distance matrix with the given graph
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dist[i][j] = graph[i][j];
            }
        }

        // Perform the Floyd's algorithm
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (dist[i][k] != Integer.MAX_VALUE && dist[k][j] != Integer.MAX_VALUE &&
                            dist[i][k] + dist[k][j] < dist[i][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                    }
                }
            }
        }

        // Print the shortest distances between all pairs of vertices
        System.out.println("Shortest distances between all pairs of vertices:");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][j] == Integer.MAX_VALUE) {
                    System.out.print("INF\t");
                } else {
                    System.out.print(dist[i][j] + "\t");
                }
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        int[][] graph = {
            {0, 5, Integer.MAX_VALUE, 10},
            {Integer.MAX_VALUE, 0, 3, Integer.MAX_VALUE},
            {Integer.MAX_VALUE, Integer.MAX_VALUE, 0, 1},
```

```
            {Integer.MAX_VALUE, Integer.MAX_VALUE, Integer.MAX_VALUE, 0}
        };

        floydWarshall(graph);
    }
}
```

Program 2

Write & Execute C++/Java Program to solve Travelling Sales Person problem using Dynamic programming.

```
import java.util.Arrays;

public class TSPDynamicProgramming {

    public static int tsp(int[][] graph, int start) {
        int n = graph.length;
        int numSets = 1 << n;
        int[][] dp = new int[numSets][n];

        // Initialize the dp array with maximum distances
        for (int[] row : dp) {
            Arrays.fill(row,  Integer.MAX_VALUE);
        }

        // Base case: If there is only one city, return the distance from the start to that
city
        for (int i = 0; i < n; i++) {
            if (i == start) {
                dp[1 << i][i] = 0;
            } else {
                dp[1 << i][i] = graph[start][i];
            }
        }

        // Dynamic Programming: Compute the shortest path for each set of cities
        for (int set = 1; set < numSets; set++) {
            for (int last = 0; last < n; last++) {
                if ((set & (1 << last)) != 0) { // Check if the last city is in the current set
                    int subset = set ^ (1 << last); // Remove the last city from the set

                    for (int curr = 0; curr < n; curr++) {
                        if ((subset & (1 << curr)) != 0) { // Check if the current city is in the subset
                            dp[set][last] = Math.min(dp[set][last], dp[subset][curr] +
graph[curr][last]);
                        }
                    }
                }
            }
        }

        // Find the minimum distance by visiting all cities and returning to the start
        int minDistance = Integer.MAX_VALUE;
        for (int last = 1; last < n; last++) {
            minDistance = Math.min(minDistance, dp[numSets - 1][last] +
graph[last][start]);
        }
```

```
        return minDistance;
    }

    public static void main(String[] args) {
        int[][] graph = {
                {0, 10, 15, 20},
                {10, 0, 35, 25},
                {15, 35, 0, 30},
                {20, 25, 30, 0}
        };
        int start = 0;

        int minDistance = tsp(graph, start);
        System.out.println("Minimum Distance: " + minDistance);
    }
}
```

Program 3

Write & Execute C++/Java Program to solve 0/1 Knapsack problem using Dynamic Programming method.

```
public class KnapsackDynamicProgramming {
   public static int knapsack(int[] weights, int[] values, int capacity) {
      int n = weights.length;

      // Create a 2D array to store the maximum value that can be obtained for each item and capacity
      int[][] dp = new int[n + 1][capacity + 1];

      // Initialize the first row and column with zeros
      for (int i = 0; i <= n; i++) {
         dp[i][0] = 0;
      }
      for (int j = 0; j <= capacity; j++) {
         dp[0][j] = 0;
      }

      // Fill the dynamic programming table
      for (int i = 1; i <= n; i++) {
         for (int j = 1; j <= capacity; j++) {
            // If the current item can be included in the knapsack
            if (weights[i - 1] <= j) {
               // Choose the maximum value between including the current item and excluding it
               dp[i][j] = Math.max(values[i - 1] + dp[i - 1][j - weights[i - 1]], dp[i - 1][j]);
            } else {
               // If the current item cannot be included, take the value from the previous row
               dp[i][j] = dp[i - 1][j];
            }
         }
      }

      // Return the maximum value that can be obtained
      return dp[n][capacity];
   }

   public static void main(String[] args) {
      int[] weights = {2, 3, 4, 5};
      int[] values = {3, 4, 5, 6};
      int capacity = 7;

      int maxValue = knapsack(weights, values, capacity);
      System.out.println("Maximum value: " + maxValue);
   }
}
```

Module 5

Program 1

Design and implement C++/Java Program to find a subset of a given set S = {Sl, S2,..., Sn} of n positive integers whose SUM is equal to a given positive integer d. For example, if S = {1, 2, 5, 6, 8} and d= 9, there are two solutions {1, 2, 6} and {1, 8}. Display a suitable message, if the given problem instance doesn't have a solution.

```
import java.util.ArrayList;
import java.util.List;

public class SubsetSum {
    public static List<Integer> findSubset(int[] set, int targetSum) {
        List<Integer> subset = new ArrayList<>();
        boolean[][] dp = new boolean[set.length + 1][targetSum + 1];

        // Initialize the first column with true
        for (int i = 0; i <= set.length; i++) {
            dp[i][0] = true;
        }

        // Fill the dynamic programming table
        for (int i = 1; i <= set.length; i++) {
            for (int j = 1; j <= targetSum; j++) {
                if (set[i - 1] <= j) {
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        // Check if a solution exists and retrieve the subset
        if (dp[set.length][targetSum]) {
            int i = set.length;
            int j = targetSum;
            while (i > 0 && j > 0) {
                if (dp[i - 1][j]) {
                    i--;
                } else {
                    subset.add(set[i - 1]);
                    j -= set[i - 1];
                    i--;
                }
            }
        }

        return subset;
    }
```

```
    public static void main(String[] args) {
        int[] set = {1, 2, 5, 6, 8};
        int targetSum = 9;

        List<Integer> subset = findSubset(set, targetSum);

        if (subset.isEmpty()) {
            System.out.println("No subset found with the given sum.");
        } else {
            System.out.println("Subset with sum " + targetSum + ": " + subset);
        }
    }
}
```

Program 2

Design and implement C++/Java Program to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle.

```
import java.util.ArrayList;
import java.util.List;

public class HamiltonianCycle {
    private int[][] graph;
    private int numVertices;
    private boolean[] visited;
    private List<Integer> hamiltonianCycle;

    public List<List<Integer>> findHamiltonianCycles(int[][] g) {
        graph = g;
        numVertices = graph.length;
        visited = new boolean[numVertices];
        hamiltonianCycle = new ArrayList<>();
        List<List<Integer>> cycles = new ArrayList<>();

        // Start the search from each vertex
        for (int start = 0; start < numVertices; start++) {
            hamiltonianCycle.clear();
            hamiltonianCycle.add(start);
            visited[start] = true;
            backtrack(start, start, 1, cycles);
            visited[start] = false;
        }

        return cycles;
    }

    private void backtrack(int start, int current, int count, List<List<Integer>> cycles) {
        // Base case: All vertices are visited and a cycle is formed
        if (count == numVertices && graph[current][start] == 1) {
            cycles.add(new ArrayList<>(hamiltonianCycle));
            return;
        }

        // Recursive case: Visit the neighbors of the current vertex
        for (int next = 0; next < numVertices; next++) {
            if (!visited[next] && graph[current][next] == 1) {
                visited[next] = true;
                hamiltonianCycle.add(next);
                backtrack(start, next, count + 1, cycles);
                visited[next] = false;
                hamiltonianCycle.remove(hamiltonianCycle.size() - 1);
            }
        }
    }
```

```
    public static void main(String[] args) {
        int[][] graph = {
            {0, 1, 1, 0, 0},
            {1, 0, 1, 1, 1},
            {1, 1, 0, 0, 1},
            {0, 1, 0, 0, 1},
            {0, 1, 1, 1, 0}
        };

        HamiltonianCycle hc = new HamiltonianCycle();
        List<List<Integer>> cycles = hc.findHamiltonianCycles(graph);

        if (cycles.isEmpty()) {
            System.out.println("No Hamiltonian cycles found in the graph.");
        } else {
            System.out.println("Hamiltonian cycles found in the graph:");
            for (List<Integer> cycle : cycles) {
                System.out.println(cycle);
            }
        }
    }
}
```