SRINIVAS UNIVERSITY
INSTITUTE OF ENGINEERING AND TECHNOLOGY
MUKKA, MANGALURU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

DESIGN AND ANALYSIS OF ALGORITHMS LAB
SUBJECT CODE: 23SCS044

COMPILED BY:

Mrs. VARSHA GANGADHAR BANGERA
Assistant Professor

Course Objectives

1. To introduce students to advanced sorting techniques:
2. To understand graph algorithms and explore optimization problems:
3. To solve computational problems using dynamic programming and enhance programming and analytical skills

Course Outcomes

Upon successful completion of this course, students will be able to:

1. **Apply sorting algorithms:** Demonstrate and analyze the performance of Quick Sort and Merge Sort for large datasets, including their best, average, and worst-case complexities.
2. **Solve graph-related problems:** Develop efficient Java programs to compute MST using Prim's and Kruskal's algorithms and solve shortest path problems using Dijkstra's and Floyd's algorithms.
3. **Implement optimization strategies:** Apply Greedy and Dynamic Programming approaches to solve the 0/1 Knapsack problem and other optimization problems.
4. **Work with dynamic programming problems:** Formulate and solve problems like the Travelling Salesperson and subset sum problems using

LABORATORY EXPERIMENTS (Total 15 Hours of Practical, 02 hours lab session per week per batch)

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 non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate using Java how the divide and-conquer method works along with its time complexity analysis: worst case, average case and best case

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 non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate using Java how the divide and-conquer method works along with its time complexity analysis: worst case, average case and best case.

3. Find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

4. Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm. Use Union-Find algorithms in your program.

5. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. Write the program in Java.

6. Implement in Java, the 0/1 Knapsack problem using Greedy method. Page 40

7. Implement in Java, the 0/1 Knapsack problem using Dynamic Programming method.

8. Write Java programs to (a) Compute the Transitive Closure of a given directed graph using Warshall’s Algorithm. (b) Implement All-Pairs Shortest Paths problem using Floyd's algorithm.

9. Implement Travelling Sales Person problem using Dynamic programming.

10. Design and implement in Java 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.

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 non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate using Java how the divide and-conquer method works along with its time complexity analysis: worst case, average case and best case

Quick Sort Implementation

1. **Algorithm:**
    - Pick a pivot element.
    - Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.
    - Recursively apply the same strategy to the sub-arrays.
    - Combine the results.
2. **Time Complexity Analysis:**
    - **Best Case:** O(nlogn)O(n \log n)O(nlogn) - Pivot divides the array into two equal halves.
    - **Average Case:** O(nlogn)O(n \log n)O(nlogn) - Random pivot results in reasonably balanced partitions.
    - **Worst Case:** O(n2)O(n^2)O(n2) - Pivot always divides array into highly uneven partitions (e.g., already sorted data with the first/last element as pivot).

Java Program

```
import java.util.Random;


public class QuickSortDemo {

   public static 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);

      }

   }
```

```
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;
}

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 = {5000, 10000, 20000, 40000, 80000};
    Random random = new Random();

    for (int size : sizes) {
        int[] array = random.ints(size, 1, 100000).toArray();
```

```
            long startTime = System.nanoTime();

            quickSort(array, 0, array.length - 1);

            long endTime = System.nanoTime();

              System.out.println("Quick Sort for n=" + size + " took " + (endTime - startTime) /
    1e6 + " ms");

          }

       }

    }
```

The program dynamically generates arrays of sizes 500050005000, 100001000010000, 200002000020000, 400004000040000, and 800008000080000 and sorts them using Quick Sort. The time taken to sort each array is measured in nanoseconds and converted to milliseconds for display.

Here's an **example output** (execution time may vary depending on the system's performance):

Quick Sort for n=5000 took 2.5 ms

Quick Sort for n=10000 took 5.4 ms

Quick Sort for n=20000 took 11.3 ms

Quick Sort for n=40000 took 23.7 ms

Quick Sort for n=80000 took 48.5 ms

Explanation of the Output

1. **Generated Arrays**: The Random class generates random integers in the range 111 to 100000100000100000 for the specified sizes.
2. **Sorting Time**: Time taken depends on:
    - o System performance (CPU speed, memory).
    - o Input data (random arrays simulate average-case performance for Quick Sort).
3. **Observation**: Sorting time approximately doubles as the array size doubles, consistent with the O(nlogn)O(n \log n)O(nlogn) complexity.

To run the program:

Compile the code using**: javac QuickSortDemo.java**

Execute the program using: **java QuickSortDemo**

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 non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate using Java how the divide and-conquer method works along with its time complexity analysis: worst case, average case and best case

Merge Sort Implementation

1. **Algorithm:**
   - Divide the array into two halves.
   - Recursively sort each half.
   - Merge the two sorted halves.
2. **Time Complexity Analysis:**
   - **Best Case, Average Case, Worst Case:** O(nlogn)O(n \log n)O(nlogn) - Recursion depth is O(logn)O(\log n)O(logn) and merging takes O(n)O(n)O(n).

Java Program

```
import java.util.Random;

public class MergeSortDemo {

    public static void mergeSort(int[] arr, int left, int right) {

        if (left < right) {

            int mid = left + (right - left) / 2;

            mergeSort(arr, left, mid);

            mergeSort(arr, mid + 1, right);

            merge(arr, left, mid, right);

        }

    }

    private static void merge(int[] arr, int left, int mid, int right) {

        int n1 = mid - left + 1;

        int n2 = right - mid;
```

```
    int[] L = new int[n1];
    int[] R = new int[n2];

    System.arraycopy(arr, left, L, 0, n1);
    System.arraycopy(arr, mid + 1, R, 0, n2);
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i++];
        } else {
            arr[k] = R[j++];
        }
        k++;
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
public static void main(String[] args) {
    int[] sizes = {5000, 10000, 20000, 40000, 80000};
    Random random = new Random();
    for (int size : sizes) {
        int[] array = random.ints(size, 1, 100000).toArray();
        long startTime = System.nanoTime();
        mergeSort(array, 0, array.length - 1);
        long endTime = System.nanoTime();
```

```
            System.out.println("Merge Sort for n=" + size + " took " + (endTime - startTime) /
1e6 + " ms");

        }

    }

}
```

The program dynamically generates arrays of sizes 500050005000, 100001000010000, 200002000020000, 400004000040000, and 800008000080000 and sorts them using Merge Sort. The time taken to sort each array is measured in nanoseconds and displayed in milliseconds.

Here's an **example output** (the actual execution time may vary depending on your system's performance):

Merge Sort for n=5000 took 3.4 ms

Merge Sort for n=10000 took 7.2 ms

Merge Sort for n=20000 took 14.9 ms

Merge Sort for n=40000 took 30.8 ms

Merge Sort for n=80000 took 64.5 ms

1. **Generated Arrays**:
    - The Random class creates random integers between 111 and 100001000000100000.
    - Each array size is independently generated and sorted.
2. **Sorting Time**:
    - Sorting time approximately doubles when the array size doubles, which aligns with the O(nlogn)O(n \log n)O(nlogn) time complexity of Merge Sort.
3. **Performance Comparison**:
    - Merge Sort has consistent performance across different inputs due to its guaranteed O(nlogn)O(n \log n)O(nlogn) complexity.
    - Unlike Quick Sort, Merge Sort doesn't degrade for certain input patterns (e.g., sorted arrays).

To Run the Program

1. Save the code in a file named MergeSortDemo.java.
2. Compile the program**: javac MergeSortDemo.java**
3. Execute the program: **java MergeSortDemo**

3.Find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

- **Input Representation**:

- The graph is represented as an adjacency matrix `graph[][]`.
- A weight of `0` means there is no edge between the corresponding vertices.

🎬 **Prim's Algorithm**:

- Starts from an arbitrary vertex and grows the MST by repeatedly adding the minimum weight edge from the set of edges connecting a vertex in the MST to a vertex outside the MST.

Java Program

```
import java.util.*;

public class PrimsAlgorithm {

    // Method to find the vertex with the minimum key value

    static int findMinVertex(int[] key, boolean[] mstSet, int vertices) {

        int minKey = Integer.MAX_VALUE, minIndex = -1;



        for (int v = 0; v < vertices; v++) {

            if (!mstSet[v] && key[v] < minKey) {

                minKey = key[v];

                minIndex = v;

            }

        }

        return minIndex;

    }



    // Method to implement Prim's Algorithm

    static void primMST(int[][] graph, int vertices) {

        int[] parent = new int[vertices]; // To store the resulting MST
```

```
int[] key = new int[vertices];   // Minimum edge weight to include a vertex
boolean[] mstSet = new boolean[vertices]; // To track vertices included in MST
// Initialize key values to infinity and mstSet[] as false
Arrays.fill(key, Integer.MAX_VALUE);
Arrays.fill(mstSet, false);
// Start from the first vertex
key[0] = 0;
parent[0] = -1; // First node is always the root of MST
for (int count = 0; count < vertices - 1; count++) {
    // Pick the minimum key vertex not yet included in MST
    int u = findMinVertex(key, mstSet, vertices);
    mstSet[u] = true; // Include the vertex in MST
    // Update key and parent values of adjacent vertices
    for (int v = 0; v < vertices; v++) {
        // Update key[v] if the edge u-v is smaller than key[v] and v is not in MST
        if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {
            parent[v] = u;
            key[v] = graph[u][v];
        }
    }
}

// Print the MST and its total cost
```

```
    printMST(parent, graph, vertices);
}
// Method to print the constructed MST
static void printMST(int[] parent, int[][] graph, int vertices) {
    System.out.println("Edge \tWeight");
    int totalCost = 0;
    for (int i = 1; i < vertices; i++) {
        System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);
        totalCost += graph[i][parent[i]];
    }
    System.out.println("Total cost of Minimum Spanning Tree: " + totalCost);
}
public static void main(String[] args) {
    // Example graph represented as an adjacency matrix
    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}
    };

    int vertices = graph.length;
```

```
        primMST(graph, vertices);
    }
}
```

Output:

- The edges of the MST are printed with their weights.
- The total cost of the MST is also displayed.

For the input 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}
}
```

The output will be:

```
Edge    Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
Total cost of Minimum Spanning Tree: 16
```

To Run

1. Save the code in a file named `PrimsAlgorithm.java`.
2. Compile and run the program

```
javac PrimsAlgorithm.java
java PrimsAlgorithm
```

4.Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm. Use Union-Find algorithms in your program.

Explanation

1. **Graph Representation**:
   - The graph is represented as a list of edges (`List<Edge>`), where each edge has a source, destination, and weight.
2. **Union-Find Algorithm**:
   - `find(parent, vertex)`: Determines the parent (root) of a vertex, with path compression for efficiency.
   - `union(parent, rank, x, y)`: Merges two subsets into one, using rank to keep the tree shallow.

3. **Kruskal's Algorithm**:
    - Sort all edges by weight.
    - Initialize a parent array for union-find and an empty list for the MST.
    - Iterate through the sorted edges:
        - If the current edge does not form a cycle, include it in the MST and perform a union operation.
    - Stop when the MST has V−1V - 1V−1 edges (where VVV is the number of vertices).

Java Program

```
import java.util.*;


class Edge implements Comparable<Edge> {
    int src, dest, weight;


    // Constructor
    public Edge(int src, int dest, int weight) {
        this.src = src;
        this.dest = dest;
        this.weight = weight;
    }


    // Sorting edges by weight@Override
    public int compareTo(Edge other) {
        return this.weight - other.weight;
    }
}


public class KruskalsAlgorithm {
    // Find the parent of a node (with path compression)
    static int find(int[] parent, int vertex) {
```

```
    if (parent[vertex] != vertex)
        parent[vertex] = find(parent, parent[vertex]);
    return parent[vertex];
}

// Union of two subsets
static void union(int[] parent, int[] rank, int x, int y) {
    int xRoot = find(parent, x);
    int yRoot = find(parent, y);

    // Attach smaller rank tree under the larger rank tree
    if (rank[xRoot] < rank[yRoot]) {
        parent[xRoot] = yRoot;
    } else if (rank[xRoot] > rank[yRoot]) {
        parent[yRoot] = xRoot;
    } else {
        parent[yRoot] = xRoot;
        rank[xRoot]++;
    }
}

// Kruskal's Algorithm to find MST
static void kruskalMST(List<Edge> edges, int vertices) {
    // Sort edges by weight
    Collections.sort(edges);

    // Result MST
```

```
List<Edge> mst = new ArrayList<>();
int[] parent = new int[vertices];
int[] rank = new int[vertices];

// Initialize subsets with single elements
for (int i = 0; i < vertices; i++) {
    parent[i] = i;
    rank[i] = 0;
}

int edgeCount = 0;
int totalCost = 0;

for (Edge edge : edges) {
    if (edgeCount == vertices - 1) break;

    int x = find(parent, edge.src);
    int y = find(parent, edge.dest);

    // If including this edge doesn't form a cycle
    if (x != y) {
        mst.add(edge);
        union(parent, rank, x, y);
        edgeCount++;
        totalCost += edge.weight;
    }
}
```

```
    // Print the MST
    System.out.println("Edge \tWeight");
    for (Edge edge : mst) {
        System.out.println(edge.src + " - " + edge.dest + "\t" + edge.weight);
    }
    System.out.println("Total cost of Minimum Spanning Tree: " + totalCost);
}

public static void main(String[] args) {
    int vertices = 6; // Number of vertices in the graph
    List<Edge> edges = new ArrayList<>();

    // Add edges (source, destination, weight)
    edges.add(new Edge(0, 1, 4));
    edges.add(new Edge(0, 2, 4));
    edges.add(new Edge(1, 2, 2));
    edges.add(new Edge(1, 0, 4));
    edges.add(new Edge(2, 3, 3));
    edges.add(new Edge(2, 5, 2));
    edges.add(new Edge(2, 4, 4));
    edges.add(new Edge(3, 4, 3));
    edges.add(new Edge(5, 4, 3));

    // Run Kruskal's algorithm
    kruskalMST(edges, vertices);
}
```

}

Output:

- Displays the edges of the MST with their weights.
- Shows the total cost of the MST.

Sample Output

For the graph in the example:

Edge Weight

1 - 2 2

2 - 5 2

2 - 3 3

3 - 4 3

0 - 1 4

Total cost of Minimum Spanning Tree: 14

How to Run

1. Save the code as `KruskalsAlgorithm.java`.
2. Compile the program: javac KruskalsAlgorithm.java
3. Execute the program: java KruskalsAlgorithm

5. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. Write the program in Java

Explanation

1. **Graph Representation:**
   - The graph is represented as an adjacency matrix where graph[i][j] indicates the weight of the edge between vertices i and j. If there's no edge, the value is 0.
2. **Initialization:**
   - All distances are initialized to Integer.MAX_VALUE (representing infinity).
   - The distance of the source vertex to itself is 0.
3. **Find Minimum Vertex:**
   - The findMinVertex method identifies the unvisited vertex with the smallest known distance.
4. **Distance Update:**
   - For each vertex adjacent to the current vertex, the algorithm updates the shortest distance if a shorter path is found.

Java Program

```
import java.util.*;



class Dijkstra {

   // Method to find the vertex with the minimum distance value

   static int findMinVertex(boolean[] visited, int[] distance) {

      int minVertex = -1;

      for (int i = 0; i < distance.length; i++) {

         if (!visited[i] && (minVertex == -1 || distance[i] < distance[minVertex])) {

            minVertex = i;

         }

      }

      return minVertex;
```

```
}

// Dijkstra's Algorithm implementation
static void dijkstra(int[][] graph, int source) {
    int vertices = graph.length;
    int[] distance = new int[vertices]; // Stores shortest distance from source
    boolean[] visited = new boolean[vertices]; // Tracks visited vertices

    // Initialize distances to infinity and visited to false
    Arrays.fill(distance, Integer.MAX_VALUE);
    distance[source] = 0;

    for (int i = 0; i < vertices - 1; i++) {
        int minVertex = findMinVertex(visited, distance);
        visited[minVertex] = true;

        // Update distance of adjacent vertices
        for (int j = 0; j < vertices; j++) {
            if (graph[minVertex][j] != 0 && !visited[j]) {
                int newDist = distance[minVertex] + graph[minVertex][j];
                if (newDist < distance[j]) {
                    distance[j] = newDist;
                }
```

```
            }
        }
    }

    // Print the shortest distances
    System.out.println("Vertex\tDistance from Source");
    for (int i = 0; i < vertices; i++) {
        System.out.println(i + "\t\t" + distance[i]);
    }
}

public static void main(String[] args) {
    // Graph represented as an adjacency matrix
    int[][] graph = {
        {0, 10, 0, 0, 0, 0},
        {10, 0, 5, 0, 0, 0},
        {0, 5, 0, 20, 1, 0},
        {0, 0, 20, 0, 2, 3},
        {0, 0, 1, 2, 0, 9},
        {0, 0, 0, 3, 9, 0}
    };

    int source = 0; // Starting vertex
```

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

Output:

- The shortest distances from the source vertex to all other vertices are displayed.

Vertex	Distance from Source
0	0
1	10
2	15
3	36
4	16
5	39

How it Works with Example

- The graph has 6 vertices. The algorithm starts at vertex 0 (source).
- It finds the shortest path to vertex 1 (distance = 10).
- From vertex 1, it discovers a shorter path to vertex 2 (distance = 15).
- The process continues until all vertices are visited

How to Run

1. Save the code as Dijkstra`.java`
2. Compile the program: javac Dijkstra.java
3. Execute the program: java Dijkstra

5. Implement in Java, the 0/1 Knapsack problem using Greedy method.

Explanation

1. **Item Representation:**
    - Each item is represented by its value and weight using the KnapsackItem class.
2. **Sorting:**
    - The items are sorted in descending order of their value-to-weight ratio using a custom comparator.
3. **Greedy Selection:**
    - The algorithm iterates through the sorted items:
        - If the item fits within the remaining capacity, it is added entirely.
        - Otherwise, it is skipped (since the problem is 0/1, fractional parts cannot be taken).

Java Program

```
import java.util.*;


class KnapsackItem {
   int value, weight;


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


public class KnapsackGreedy {
```

```
    // Method to solve the 0/1 Knapsack problem using the Greedy method
    static int knapsackGreedy(KnapsackItem[] items, int capacity) {
        // Sort items by value-to-weight ratio in descending order
            Arrays.sort(items, (a, b) -> Double.compare((double) b.value / b.weight, (double) a.value / a.weight));

        int totalValue = 0;
        int remainingCapacity = capacity;

        for (KnapsackItem item : items) {
            if (item.weight <= remainingCapacity) {
                // Take the entire item
                totalValue += item.value;
                remainingCapacity -= item.weight;
            } else {
                // Skip the item (since it's 0/1 Knapsack, partial items are not allowed)
                continue;
            }
        }

        return totalValue;
    }

    public static void main(String[] args) {
```

```
        // Define items with value and weight
        KnapsackItem[] items = {
            new KnapsackItem(60, 10),
            new KnapsackItem(100, 20),
            new KnapsackItem(120, 30)
        };

        int capacity = 50; // Knapsack capacity

        int maxValue = knapsackGreedy(items, capacity);
        System.out.println("Maximum value obtained: " + maxValue);
    }
}
```

Output:

- The total value of the items included in the knapsack is returned.

Sample Output

For the input items:

- Value = 60, Weight = 10
- Value = 100, Weight = 20
- Value=120,Weight=30
  knapsack capacity = 50.

Maximum value obtained: 220

How It Works

- The value-to-weight ratios are calculated as:
    - Item 1: 60/10=660 / 10 = 660/10=6
    - Item 2: 100/20=5100 / 20 = 5100/20=5
    - Item 3: 120/30=4120 / 30 = 4120/30=4
- The items are sorted in descending order by their ratios.
- The algorithm selects:
    - Item 1 (entirely) → Remaining capacity: 40 → Total value: 60
    - Item 2 (entirely) → Remaining capacity: 20 → Total value: 160
    - Item 3 (partially cannot fit) → Remaining capacity: 0 → Final value: 220.

KnapsackGreedy

How to Run

- Save the code as *KnapsackGreedy*`.java`
- Compile the program: javac *KnapsackGreedy.java*
- Execute the program: java *KnapsackGreedy*

7. Implement in Java, the 0/1 Knapsack problem using Dynamic Programming method.

The **0/1 Knapsack problem** using Dynamic Programming solves the problem by creating a 2D table where the value at dp[i][w] represents the maximum value that can be obtained using the first i items with a weight capacity w.

Explanation

1. **Input:**
    - values[]: Array of values for each item.
    - weights[]: Array of weights for each item.
    - capacity: The maximum weight the knapsack can carry.
2. **Dynamic Programming Table:**
    - dp[i][w] stores the maximum value that can be obtained using the first i items with weight w.
    - If the weight of the current item (weights[i-1]) is less than or equal to w, two options are considered:
        - Include the item: values[i-1] + dp[i-1][w - weights[i-1]]
        - Exclude the item: dp[i-1][w]
    - Otherwise, the item is excluded.

Java program

```
public class KnapsackDP {

   // Method to solve 0/1 Knapsack using Dynamic Programming

   static int knapsackDP(int[] values, int[] weights, int capacity) {

      int n = values.length;

      int[][] dp = new int[n + 1][capacity + 1];


      // Build DP table

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

         for (int w = 1; w <= capacity; w++) {
```

```
            if (weights[i - 1] <= w) {
                // Include the current item
                dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
            } else {
                // Exclude the current item
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][capacity];
}

public static void main(String[] args) {
    int[] values = {60, 100, 120};  // Values of the items
    int[] weights = {10, 20, 30};  // Weights of the items
    int capacity = 50;  // Knapsack capacity

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

Result:

- The final result is stored in dp[n][capacity], where n is the total number of items.

Sample Output

For the input:

- values[] = {60, 100, 120}
- weights[] = {10, 20, 30}
- capacity = 50

The output will be:

Maximum value in Knapsack: 220

How it works:

Step-by-Step Explanation

1. **Initialize Table:**
    - dp[0][w] = 0 for all w, as no items can be included.
    - dp[i][0] = 0 for all i, as the capacity is 0.
2. **Fill Table:**
    - For i = 1, w = 10: Include the first item (value = 60, weight = 10).
    - For i = 2, w = 20: Include the second item (value = 100, weight = 20).
    - For i = 3, w = 50: Include both the second and third items, giving a maximum value of 220.
3. **Final Result:**
    - The value in dp[3][50] is 220.

DP Table Construction

Each cell `dp[i][w]` represents the maximum value that can be obtained using the first `i` items with weight capacity `w`.

Item/Weight w	0	10	20	30	40	50
0 items	0	0	0	0	0	0
1 item (60, 10)	0	60	60	60	60	60
2 items (100, 20)	0	60	100	160	160	160
3 items (120, 30)	0	60	100	160	180	220

Explanation of the Table

1. **Row 0 (No Items):**
    - dp[0][w] = 0 for all www, as no items are included.
2. **Row 1 (Item 1: Value = 60, Weight = 10):**
    - For w<10w < 10w<10, item 1 can't be included: dp[1][w] = 0.
    - For w≥10w \geq 10w≥10, include item 1: dp[1][w] = 60.
3. **Row 2 (Item 2: Value = 100, Weight = 20):**
    - For w<20w < 20w<20, item 2 can't be included: dp[2][w] = dp[1][w].
    - For w≥20w \geq 20w≥20, consider:
        - Exclude item 2: dp[1][w].
        - Include item 2: value[2] + dp[1][w - weight[2]].
    - Example: At w=30w = 30w=30, dp[2][30] = max(60, 100 + dp[1][10]) = 160.
4. **Row 3 (Item 3: Value = 120, Weight = 30):**
    - For w<30w < 30w<30, item 3 can't be included: dp[3][w] = dp[2][w].
    - For w≥30w \geq 30w≥30, consider:
        - Exclude item 3: dp[2][w].
        - Include item 3: value[3] + dp[2][w - weight[3]].

- Example: At w=50w = 50w=50, dp[3][50] = max(160, 120 + dp[2][20]) = 220.

How to Run

- Save the code as KnapsackDP`.java`
- Compile the program: javac KnapsackDP.*java*
- Execute the program: java KnapsackDP

8.Write Java programs to (a) Compute the Transitive Closure of a given directed graph using Warshall's Algorithm. (b) Implement All-Pairs Shortest Paths problem using Floyd's algorithm.

(a) Transitive Closure Using Warshall's Algorithm

Warshall's Algorithm is used to compute the transitive closure of a directed graph, which identifies if there exists a path between any two vertices.

Explanation

1. Input Graph Representation:
    - The graph is represented as an adjacency matrix.
    - If graph[i][j] = 1, there is a direct edge from vertex i to vertex j.
2. Warshall's Algorithm Logic:
    - Update reach[i][j] to 1 if there is a path from i to j through an intermediate vertex k.

Java Program

```
import java.util.*;


public class TransitiveClosure {
   // Method to compute the transitive closure using Warshall's Algorithm
   static void warshall(int[][] graph) {
      int vertices = graph.length;
      int[][] reach = new int[vertices][vertices];

      // Initialize reachability matrix with input graph
      for (int i = 0; i < vertices; i++) {
         for (int j = 0; j < vertices; j++) {
            reach[i][j] = graph[i][j];
         }
```

```
    }

    // Warshall's Algorithm
    for (int k = 0; k < vertices; k++) {
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                reach[i][j] = reach[i][j] | (reach[i][k] & reach[k][j]);
            }
        }
    }

    // Print the transitive closure matrix
    System.out.println("Transitive Closure:");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            System.out.print(reach[i][j] + " ");
        }
        System.out.println();
    }
}

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

```
            {0, 0, 1}
        };


        warshall(graph);
    }
}
```

Output:

- The final matrix shows whether a path exists (1) or does not exist (0) between each pair of vertices.

Sample Output

For the input graph:

1 1 0

0 1 1

0 0 1

The output will be:

Transitive Closure:

1 1 1

0 1 1

0 0 1

How to Run

- Save the code as TransitiveClosure.`java`
- Compile the program: javac TransitiveClosure.*java*
- Execute the program: java TransitiveClosure

b) Java Program for Floyd's Algorithm

Explanation

1. **Input Graph Representation:**
   - graph[i][j] holds the weight of the edge between vertex i and j.
   - If there's no edge, the value is set to INF (a large value like 99999).
2. **Floyd's Algorithm Logic:**
   - For each intermediate vertex k, update the shortest path between every pair of vertices i and j:
     - dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j]).

Java Program

```
import java.util.*;



public class FloydAlgorithm {

   final static int INF = 99999;



   // Method to compute shortest paths using Floyd's Algorithm

   static void floyd(int[][] graph) {

      int vertices = graph.length;

      int[][] dist = new int[vertices][vertices];



      // Initialize the distance matrix with input graph

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

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

            dist[i][j] = graph[i][j];

         }
```

```
}

// Floyd's Algorithm
for (int k = 0; k < vertices; k++) {
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}

// Print the shortest path matrix
System.out.println("Shortest Path Matrix:");
for (int i = 0; i < vertices; i++) {
    for (int j = 0; j < vertices; j++) {
        if (dist[i][j] == INF) {
            System.out.print("INF ");
        } else {
            System.out.print(dist[i][j] + " ");
        }
    }
```

```
            System.out.println();
        }
    }

    public static void main(String[] args) {
        int[][] graph = {
            {0, 3, INF, 5},
            {2, 0, INF, 4},
            {INF, 1, 0, INF},
            {INF, INF, 2, 0}
        };

        floyd(graph);
    }
}
```

Output:

- The final matrix contains the shortest path distances between every pair of vertices.

Sample Output

For the input graph:

0	3	INF	5
2	0	INF	4
INF	1	0	INF
INF	INF	2	0

The output will be:
Shortest Path Matrix:

0	3	7	5
2	0	6	4
3	1	0	5
5	3	2	0

How to Run

- Save the code as FloydAlgorithm.`java`
- Compile the program: javac FloydAlgorithm.*java*
- Execute the program: java FloydAlgorithm

9.Implement Travelling Sales Person problem using Dynamic programming.

The **Travelling Salesperson Problem (TSP)** is a well-known combinatorial optimization problem. It involves finding the shortest possible route that visits a set of cities exactly once and returns to the starting city.

Explanation

Dynamic Programming Approach

We solve TSP using the **Dynamic Programming** approach with bitmasking. The state is represented as:

- dp[mask][i]: The minimum cost to visit all cities in mask and end at city i.

Here, mask is a bitmask representing the set of visited cities.

- **Input:**
  - dist[][]: Adjacency matrix where dist[i][j] is the distance between city i and city j.
- **Dynamic Programming Table:**
  - dp[mask][pos]: Stores the minimum cost to visit cities in mask and end at city pos.
- **Recursive Function (tspUtil):**

- Base Case: If all cities are visited (mask == VISITED_ALL), return the cost to go back to the starting city.
- Recursive Case: Explore all unvisited cities and calculate the minimum cost.

🎬 **Bitmask Representation:**

- mask: Represents the set of visited cities.
- (mask & (1 << city)) == 0: Checks if a city is not visited.
- (mask | (1 << city)): Marks a city as visited.

```
import java.util.Arrays;


public class TSPDynamicProgramming {
    static final int INF = Integer.MAX_VALUE;


    // TSP solution using Dynamic Programming
    static int tsp(int[][] dist, int n) {
        int VISITED_ALL = (1 << n) - 1; // All cities visited
        int[][] dp = new int[1 << n][n];


        // Initialize DP table
        for (int[] row : dp) {
            Arrays.fill(row, -1);
        }


        // Start the recursion with mask=1 (starting city visited) and current city=0
        return tspUtil(dist, dp, 1, 0, n, VISITED_ALL);
    }
```

```
    static int tspUtil(int[][] dist, int[][] dp, int mask, int pos, int n, int VISITED_ALL) {
        if (mask == VISITED_ALL) {
            // All cities visited, return cost to start city
            return dist[pos][0];
        }

        if (dp[mask][pos] != -1) {
            // Return memoized result
            return dp[mask][pos];
        }

        int minCost = INF;

        // Visit all unvisited cities
        for (int city = 0; city < n; city++) {
            if ((mask & (1 << city)) == 0) {
                int newCost = dist[pos][city] + tspUtil(dist, dp, mask | (1 << city), city, n,
VISITED_ALL);
                minCost = Math.min(minCost, newCost);
            }
        }

        // Memoize and return result
        return dp[mask][pos] = minCost;
```

```
    }

    public static void main(String[] args) {
        int[][] dist = {
            {0, 10, 15, 20},
            {10, 0, 35, 25},
            {15, 35, 0, 30},
            {20, 25, 30, 0}
        };
        int n = dist.length;

        int result = tsp(dist, n);
        System.out.println("The minimum cost to complete the tour is: " + result);
    }
}
```

Sample Output

For the given input distance matrix:

0 10 15 20

10 0 35 25

15 35 0 30

20 25 30 0

The output will be:

The minimum cost to complete the tour is: 80

How it works

Step-by-Step Explanation

1. Matrix Representation:
    - The matrix represents the distances between cities.
    - Example: Distance from city 0 to city 1 is 10.
2. Masking and DP States:
    - dp[mask][pos] is initialized to -1.
    - Starting city is 0, and the recursive function computes the minimum cost for all possible tours.
3. Optimal Tour:
    - For the input, the optimal tour is:
        - Start at city 0 → city 1 → city 3 → city 2 → back to city 0.
    - The cost of this tour is 80.

How to Run

Save the code as TSPDynamicProgramming.`java`

Compile the program: javac TSPDynamicProgramming.*java*

Execute the program: java TSPDynamicProgramming

10.Design and implement in Java 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.

The **Subset Sum Problem** involves finding subsets of a given set of positive integers SSS whose sum equals a target sum ddd. The goal is to generate all possible subsets and check if their sum equals ddd.

Explanation

1. **Input:**
    - set[]: Array of positive integers.
    - targetSum: The target sum ddd.
2. **Recursive Backtracking:**
    - Start with an empty subset and iterate through the array.
    - At each step:
        - Include the current element in the subset and reduce the target sum.
        - Exclude the current element and proceed.
    - If the target sum reaches 0, the subset is a valid solution.
    - If the index exceeds the array length or the target sum becomes negative, backtrack.
3. **Backtracking Mechanism:**
    - Add the current element to the subset.
    - Explore further subsets.
    - Remove the element (backtrack) and explore other possibilities.

Java Implementation

```
import java.util.ArrayList;

import java.util.List;
```

```
public class SubsetSum {

    // Method to find subsets with a given sum

    static void findSubsets(int[] set, int index, int targetSum, List<Integer> currentSubset,
List<List<Integer>> allSolutions) {

        // Base case: if targetSum is 0, add the current subset to the solutions

        if (targetSum == 0) {

            allSolutions.add(new ArrayList<>(currentSubset));

            return;

        }


        // If index exceeds the array length or targetSum becomes negative, return

        if (index >= set.length || targetSum < 0) {

            return;

        }


        // Include the current element in the subset

        currentSubset.add(set[index]);

        findSubsets(set, index + 1, targetSum - set[index], currentSubset, allSolutions);


        // Backtrack: exclude the current element from the subset

        currentSubset.remove(currentSubset.size() - 1);

        findSubsets(set, index + 1, targetSum, currentSubset, allSolutions);

    }


    public static void main(String[] args) {
```

```
        int[] set = {1, 2, 5, 6, 8}; // Input set
        int targetSum = 9;           // Target sum

        List<List<Integer>> allSolutions = new ArrayList<>();
        findSubsets(set, 0, targetSum, new ArrayList<>(), allSolutions);

        if (allSolutions.isEmpty()) {
            System.out.println("No subset with the given sum exists.");
        } else {
            System.out.println("Subsets with the given sum are:");
            for (List<Integer> solution : allSolutions) {
                System.out.println(solution);
            }
        }
    }
}
```

Output:

- If no solution exists, print a suitable message.
- Otherwise, print all subsets whose sum equals ddd.

Sample Output

For the input:

- S={1,2,5,6,8}S = \{1, 2, 5, 6, 8\}S={1,2,5,6,8}
- d=9d = 9d=9

The output will be:

Subsets with the given sum are:

[1, 2, 6]

[1, 8]

How it works

Step-by-Step Execution

1. Start with an empty subset and target sum 999.
2. Include or exclude each element in the array:
    - Include 111, target becomes 9−1=89 - 1 = 89−1=8.
    - Include 222, target becomes 8−2=68 - 2 = 68−2=6.
    - Include 666, target becomes 6−6=06 - 6 = 06−6=0. Add subset [1,2,6][1, 2, 6][1,2,6] to solutions.
3. Backtrack and explore other subsets:
    - Include 888, target becomes 9−8=09 - 8 = 09−8=0. Add subset [1,8][1, 8][1,8] to solutions.
4. Repeat until all possibilities are explored.

Complexity

- **Time Complexity:** O(2n)O(2^n)O(2n), where nnn is the size of the input array. Each element can either be included or excluded.
- **Space Complexity:** O(n)O(n)O(n), due to the recursive call stack and the storage of the current subset.

How to Run

Save the code as SubsetSum`.java`

Compile the program: javac SubsetSum*.java*

Execute the program: java SubsetSum