Name : Jagtap Nachiket Nitin
En.No.: 21221079

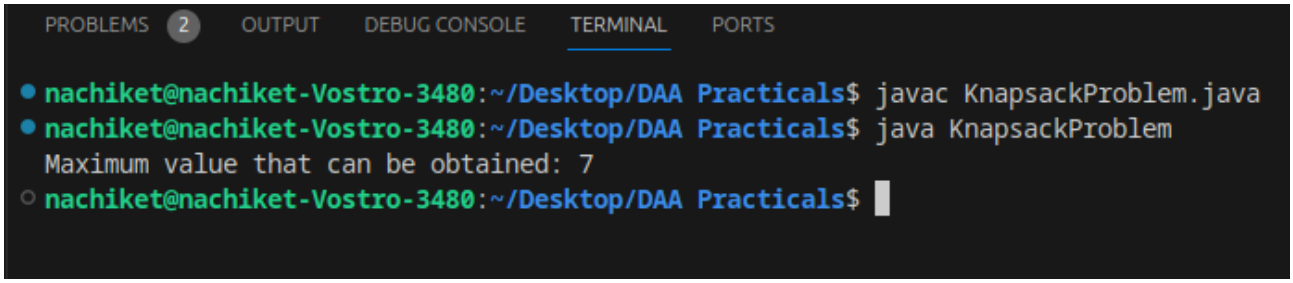
Assignment 4

Backtracking and Branch-n-Bound

e) Knapsack problem,
Java Code :

```
public class KnapsackProblem {
// Function to solve 0/1 Knapsack problem using dynamic programming
public static int knapsack(int[] weights, int[] values, int totalWeight) {
int n = weights.length;
int[][] dp = new int[n + 1][totalWeight + 1];

// Build dp table
for (int i = 1; i <= n; i++) {
for (int w = 1; w <= totalWeight; w++) {
if (weights[i - 1] <= w) {
dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
} else {
dp[i][w] = dp[i - 1][w];
}
}
}

// The maximum value that can be obtained
return dp[n][totalWeight];
}

// Main method to test the knapsack function
public static void main(String[] args) {
int[] weights = {2, 3, 4, 5};
int[] values = {3, 4, 5, 6};
int totalWeight = 5;

int maxValue = knapsack(weights, values, totalWeight);
System.out.println("Maximum value that can be obtained: " + maxValue);
}
}
```

Output :



f) Travelling Salesman Problem

Java Code:

```
import java.util.Arrays;

public class TravelingSalesmanProblem {

// Number of vertices in the graph
static int V = 4;

// Memoization table to store the results of subproblems
static int[][] dp;

// Adjacency matrix representing the graph
static int[][] graph = {
{0, 10, 15, 20},
{10, 0, 35, 25},
{15, 35, 0, 30},
{20, 25, 30, 0}
};

// Function to solve the Traveling Salesman Problem using dynamic programming
static int tsp(int mask, int pos) {
// If all cities have been visited, return the cost from the current city to the starting city
if (mask == (1 << V) - 1) {
return graph[pos][0];
}

// If the subproblem has already been solved, return the stored result
if (dp[mask][pos] != -1) {
return dp[mask][pos];
}

// Initialize the result to a large value
int minCost = Integer.MAX_VALUE;

// Try to visit all cities
for (int city = 0; city < V; city++) {
// If the city is not visited yet and there is a direct edge from the current city to this city
if ((mask & (1 << city)) == 0 && graph[pos][city] > 0) {
```

```
int newMask = mask | (1 << city);
int newCost = graph[pos][city] + tsp(newMask, city);
minCost = Math.min(minCost, newCost);
}
}

// Store the result of the subproblem in the memoization table
dp[mask][pos] = minCost;
return minCost;
}

public static void main(String[] args) {
// Initialize the memoization table with -1
dp = new int[1 << V][V];
for (int[] row : dp) {
Arrays.fill(row, -1);
}

// Start the TSP from city 0 and consider all other cities as unvisited (mask = 1)
int mask = 1;
int minCost = tsp(mask, 0);

System.out.println("Minimum cost of visiting all cities: " + minCost);
}
}
```

Output: