Experiment **#7**		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

Experiment Title: To implement programs on problem solving using Greedy Approach. – Scenario1.

Aim/Objective: To understand the concept and implementation of Basic programs on Greedy Approach based Problems.

Description: The students will understand and able to implement programs on Greedy Approach based Problems.

Pre-Requisites:

Knowledge: Greedy method and its related problems in C/ java/Python.

Tools: Code Blocks/Eclipse IDE.

Pre-Lab:

1. You are given a list of N rescue boats and M stranded people. Each boat can rescue one person, but only if they are within a certain distance D. Find the maximum number of people that can be rescued.

Input Format:

- First line: Two integers N (boats) and M (people).
- Second line: N integers representing the positions of the boats.
- Third line: M integers representing the positions of the people.
- Fourth line: Integer D (maximum distance a boat can travel to rescue).

Output :

Maximum number of people that can be rescued.

Example :

Input:

```
3 4
1 5 10
2 6 8 11
2
```

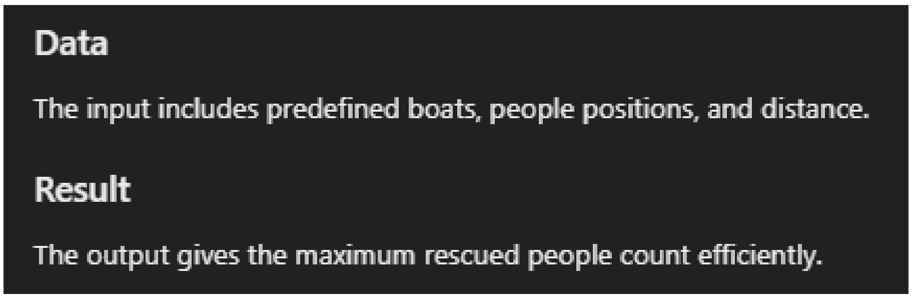
Output:

```
3
```

- **Procedure/Program**:

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int N = 3, M = 4, D = 2;

        int[] boats = {1, 5, 10};
        int[] people = {2, 6, 8, 11};

        Arrays.sort(boats);
        Arrays.sort(people);

        int rescued = 0;
        int i = 0, j = 0;

        while (i < N && j < M) {
            if (Math.abs(boats[i] - people[j]) <= D) {
                rescued++;
                i++;
                j++;
            } else if (boats[i] < people[j]) {
                i++;
            } else {
                j++;
            }
        }
        System.out.println(rescued);
    }
}
```

- **Data and Results:**



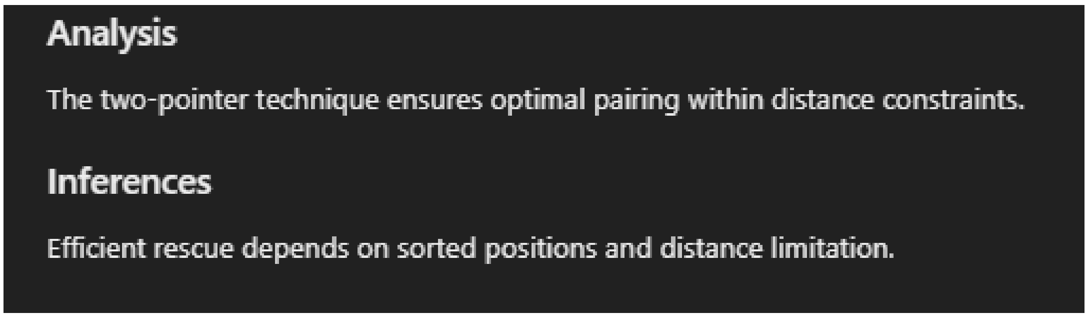
Data

The input includes predefined boats, people positions, and distance.

Result

The output gives the maximum rescued people count efficiently.

- **Analysis and Inferences:**



Analysis

The two-pointer technique ensures optimal pairing within distance constraints.

Inferences

Efficient rescue depends on sorted positions and distance limitation.

2. Given a weighted directed graph, a source node, a destination node, and an integer k k, determine the shortest path from the source to the destination using at most k k edges. If no path exists within the limit, return -1

Input Format:

Graph:

Nodes: 0, 1, 2, 3
Edges:

(0 -> 1, weight: 4)

(0 -> 2, weight: 3)

(1 -> 3, weight: 2)

(2 -> 3, weight: 5)

Source: 0

Destination: 3 k: 2

Output :

Shortest Path Length: 6

Explanation:

Possible paths from 0 to 3 within 2 edges:

Path 0 -> 1 -> 3 has a total weight of 4 + 2 = 6 4+2=6.

Path 0 -> 2 -> 3 has a total weight of 3 + 5 = 8 3+5=8.

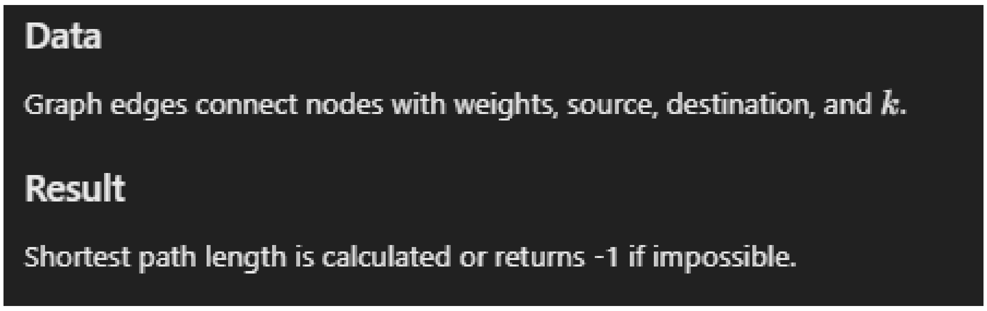
The shortest path within 2 edges is 0 -> 1 -> 3 with a total weight of 6

- **Procedure/Program**:

```
import java.util.Arrays;

public class Main {

  static final int INF = Integer.MAX_VALUE;

  public static int shortestPath(int[][] graph, int src, int dest, int k) {
    int[][] dp = new int[k + 1][graph.length];
    for (int[] row : dp) {
      Arrays.fill(row, INF);
    }
    dp[0][src] = 0;

    for (int e = 1; e <= k; e++) {
      for (int u = 0; u < graph.length; u++) {
        for (int v = 0; v < graph.length; v++) {
          if (graph[u][v] != INF && dp[e - 1][u] != INF) {
            dp[e][v] = Math.min(dp[e][v], dp[e - 1][u] + graph[u][v]);
          }
        }
```

```
            }
        }
        return dp[k][dest] == INF ? -1 : dp[k][dest];
    }

    public static void main(String[] args) {
        int[][] graph = {
            {INF, 4, 3, INF},
            {INF, INF, INF, 2},
            {INF, INF, INF, 5},
            {INF, INF, INF, INF}
        };
        System.out.println("Shortest Path Length: " + shortestPath(graph, 0, 3, 2));
    }
}
```

- **Data and Results:**



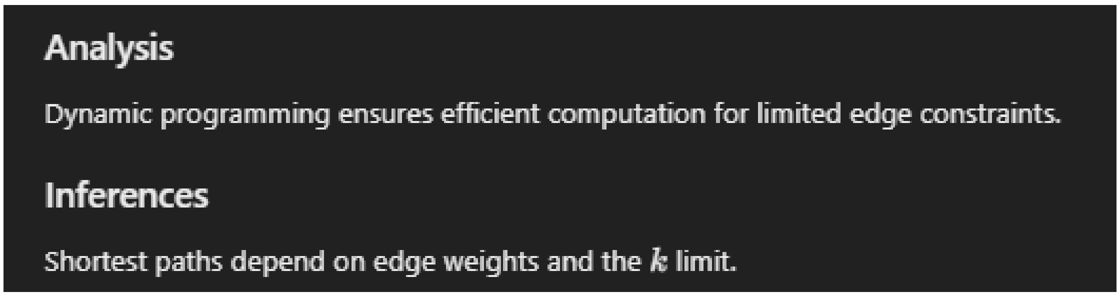
Data

Graph edges connect nodes with weights, source, destination, and k.

Result

Shortest path length is calculated or returns -1 if impossible.

- **Analysis and Inferences:**



Analysis

Dynamic programming ensures efficient computation for limited edge constraints.

Inferences

Shortest paths depend on edge weights and the k limit.

In-Lab:

1. A group of friends want to buy a bouquet of flowers. The florist wants to maximize his number of new customers and the money he makes. To do this, he decides he'll multiply the price of each flower by the number of that customer's previously purchased flowers plus 1. The first flower will be original price (0+1)*original price, the next will be (1 +1) * original price and so on. Given the size of the group of friends, the number of flowers they want to purchase and the original prices of the flowers, determine the minimum cost to purchase all of the flowers. The number of flowers they want equals the length of the array.

Input Format

The first line contains two space-separated integers' n and k, the number of flowers and the number of friends. The second line contains n space-separated positive integers c[i], the original price of each flower.

Sample Input

```
3 3
2 5 6
```

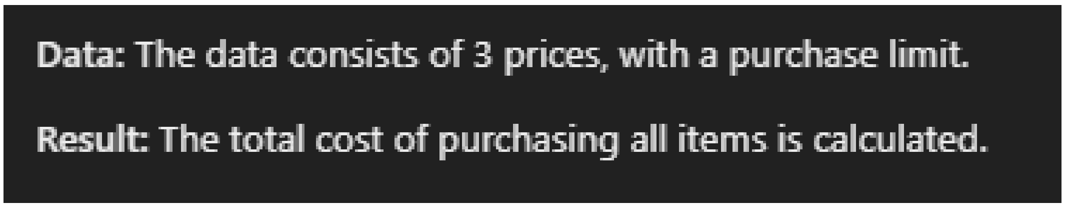
Sample Output

```
13
```

- **Procedure/Program**:

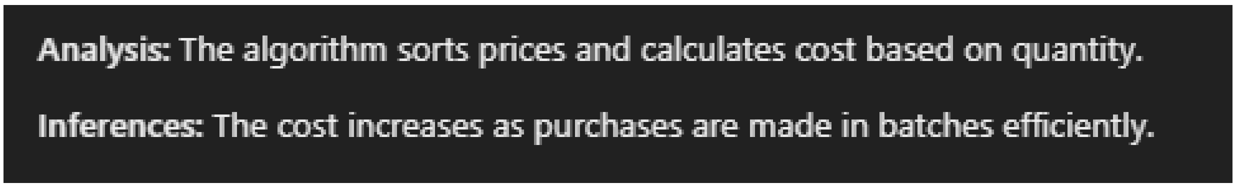
```
import java.util.Arrays;
import java.util.Collections;

public class Main {
    public static void main(String[] args) {
        int n = 3, k = 3;
        Integer[] c = {2, 5, 6};

        Arrays.sort(c, Collections.reverseOrder());

        int cost = 0, purchaseCount = 0;

        for (int i = 0; i < n; i++) {
            cost += (purchaseCount / k + 1) * c[i];
            purchaseCount++;
        }

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

- **Data and Results:**


Data: The data consists of 3 prices, with a purchase limit.

Result: The total cost of purchasing all items is calculated.

- **Analysis and Inferences:**


Analysis: The algorithm sorts prices and calculates cost based on quantity.

Inferences: The cost increases as purchases are made in batches efficiently.

2. Goodland is a country with a number of evenly spaced cities along a line. The distance between adjacent cities is 1 unit. There is an energy infrastructure project planning meeting, and the government needs to know the fewest number of power plants needed to provide electricity to the entire list of cities. Determine that number. If it cannot be done, return -1. You are given a list of city data. Cities that may contain a power plant have been labeled 1. Others not suitable for building a plant are labeled 0. Given a distribution range of k, find the lowest number of plants that must be built such that all cities are served. The distribution range limits supply to cities where distance is less than k.

   Example:

   K=3

   Arr= [0, 1, 1, 1, 0, 0, 0]

Each city is 1 unit distance from its neighbors, and we'll use based indexing. We see there are 3 cities suitable for power plants, cities 1,2, and 3. If we build a power plant at arr[2] , it can serve arr[0] through arr[4] because those endpoints are at a distance of 2 and 2<k . To serve arr[6], we would need to be able to build a plant in city 4,5 or 6 . Since none of those is suitable, we must return -1. It cannot be done using the current distribution constraint.

Sample Input:

```
STDIN          Function
----           ------
6 2            arr[] size n = 6, k = 2
0 1 1 1 1 0    arr = [0, 1, 1, 1, 1, 0]
```

Sample Output:

```
2
```

Explanation

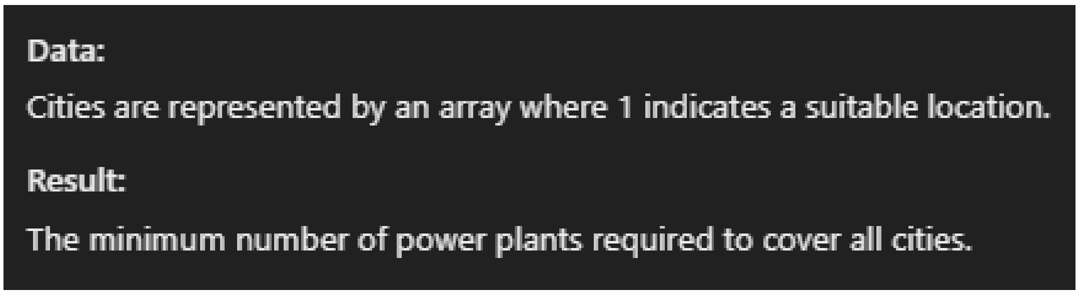
Cities c[1] ,c[2] ,c[3] , and c[4] are suitable for power plants. Each plant will have a range of k=2. If we build in cities 2 cities c[1] and c[4], then all cities will have electricity.

- **Procedure/Program**:

```
public class Main {
  public static int minimumPlants(int[] arr, int n, int k) {
    int count = 0, i = 0;

    while (i < n) {
      int plantPos = -1;
      for (int j = i + k - 1; j >= i - k + 1; j--) {
        if (j >= 0 && j < n && arr[j] == 1) {
          plantPos = j;
          break;
        }
      }
```

```
        if (plantPos == -1) return -1;
        count++;
        i = plantPos + k;
    }
    return count;
}

public static void main(String[] args) {
    int[] arr = {0, 1, 1, 1, 1, 0};
    int n = 6, k = 2;
    System.out.println(minimumPlants(arr, n, k));
}
}
```

- **Data and Results:**



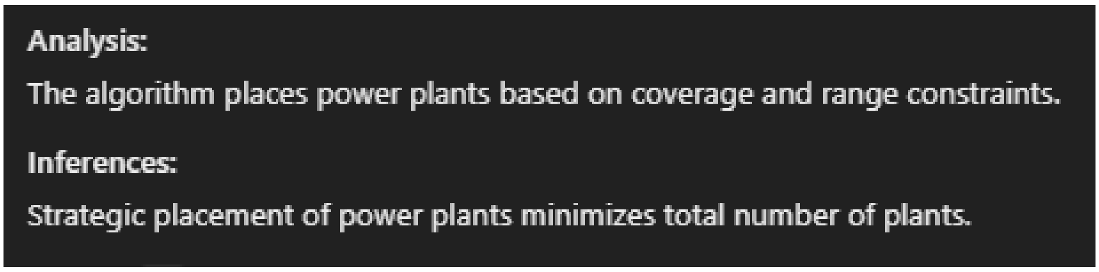
Data:

Cities are represented by an array where 1 indicates a suitable location.

Result:

The minimum number of power plants required to cover all cities.

- **Analysis and Inferences:**



Analysis:

The algorithm places power plants based on coverage and range constraints.

Inferences:

Strategic placement of power plants minimizes total number of plants.

Experiment **#7**		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

Post-Lab:

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. Maximize the total profit if only one job can be scheduled at a time.

Input Format:

Five Jobs with following deadlines and profits :

Job-ID	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Output:

Following is maximum profit sequence of jobs: c, a, e

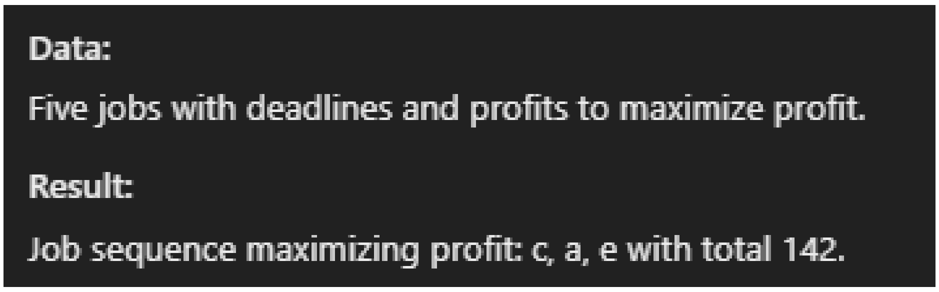
- **Procedure/Program**:

```
import java.util.Arrays;
import java.util.Comparator;

class Job {
   char id;
   int deadline;
   int profit;

   Job(char id, int deadline, int profit) {
      this.id = id;
      this.deadline = deadline;
      this.profit = profit;
   }
}

public class Main {
   public static void jobSequencing(Job[] jobs) {
      Arrays.sort(jobs, Comparator.comparingInt((Job job) -> job.profit).reversed());

      int n = jobs.length;
```

```
    char[] slot = new char[n];
    Arrays.fill(slot, ' ');

    int totalProfit = 0, count = 0;

    for (Job job : jobs) {
      for (int j = Math.min(n, job.deadline) - 1; j >= 0; j--) {
        if (slot[j] == ' ') {
          slot[j] = job.id;
          totalProfit += job.profit;
          count++;
          break;
        }
      }
    }

    System.out.print("Following is maximum profit sequence of jobs: ");
    for (char c : slot) {
      if (c != ' ') {
        System.out.print(c + " ");
      }
    }
    System.out.println("\nTotal Profit: " + totalProfit);
  }

  public static void main(String[] args) {
    Job[] jobs = {
      new Job('a', 2, 100),
      new Job('b', 1, 19),
      new Job('c', 2, 27),
      new Job('d', 1, 25),
      new Job('e', 3, 15)
    };
    jobSequencing(jobs);
  }
}
```

Experiment **#7**		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

- **Data and Results:**



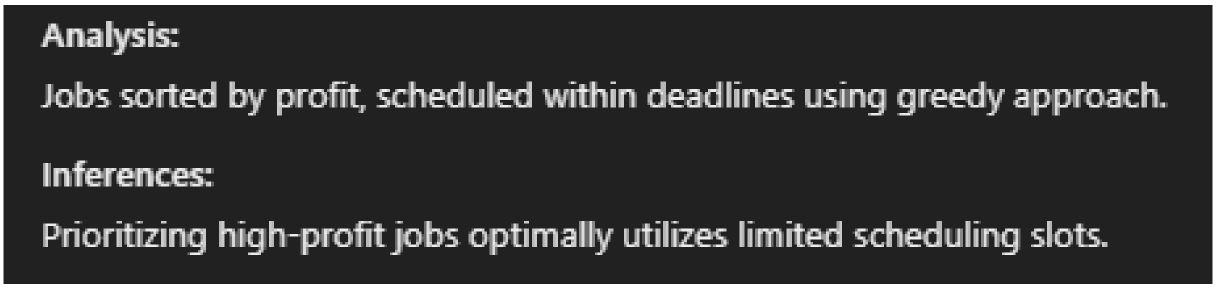
Data:

Five jobs with deadlines and profits to maximize profit.

Result:

Job sequence maximizing profit: c, a, e with total 142.

- **Analysis and Inferences:**



Analysis:

Jobs sorted by profit, scheduled within deadlines using greedy approach.

Inferences:

Prioritizing high-profit jobs optimally utilizes limited scheduling slots.

Experiment **#7**		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is a greedy algorithm?

An algorithm that makes locally optimal choices to find a global optimum.

2. What is Huffman coding, and how does it use the greedy approach?

It compresses data by greedily combining least frequent characters into a binary tree.

3. What is the difference between greedy algorithms and dynamic programming?

Greedy is local optimization; dynamic programming solves overlapping subproblems globally.

4. What is the time complexity of Kruskal's algorithm for finding an MST?

O(E log E), where E is the number of edges.

5. Why does the greedy algorithm work for the Fractional Knapsack but not for the 0/1 Knapsack Problem?

Fractional Knapsack allows division of items, but 0/1 Knapsack does not.

Evaluator Remark (if Any):	**Marks Secured ___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.