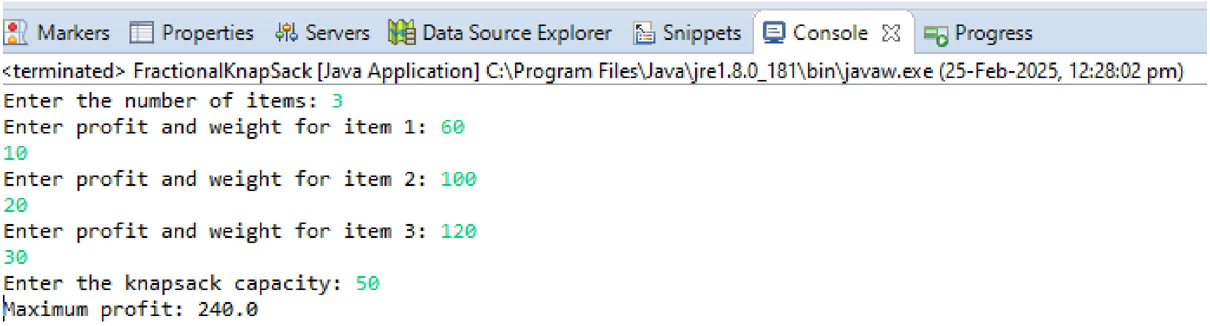
Unit -3 Greedy Technique
Lab Notes

Program-11 Write a program to implement the Knapsack problem using the greedy method

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

// Greedy approach for Fractional Knapsack
public class FractionalKnapSack {

    // ItemValue class represents an item with profit and weight.
    static class ItemValue {
        int profit, weight;

        public ItemValue(int profit, int weight) {
            this.profit = profit;
            this.weight = weight;
        }
    }

    // Function to calculate maximum value for the fractional knapsack.
    private static double getMaxValue(ItemValue[] arr, int capacity) {
        // Sorting items by profit/weight ratio in descending order.
        Arrays.sort(arr, new Comparator<ItemValue>() {
            @Override
            public int compare(ItemValue item1, ItemValue item2) {
                double ratio1 = (double) item1.profit / item1.weight;
                double ratio2 = (double) item2.profit / item2.weight;
                return ratio1 < ratio2 ? 1 : -1;
            }
        });

        double totalValue = 0d;

        // Iterate over the sorted items
        for (ItemValue item : arr) {
            int curWt = item.weight;
            int curVal = item.profit;

            if (capacity - curWt >= 0) {
                // If the entire item can be taken, subtract its weight and add its profit.
                capacity -= curWt;
                totalValue += curVal;
            } else {
```

```
                // Only a fraction of the item can be taken.
                double fraction = (double) capacity / curWt;
                totalValue += (curVal * fraction);
                capacity = 0; // Knapsack is now full.
                break;
            }
        }

        return totalValue;
    }

    // Main method that takes input from the user.
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input number of items.
        System.out.print("Enter the number of items: ");
        int n = sc.nextInt();

        // Create an array to hold the items.
        ItemValue[] items = new ItemValue[n];

        // Read each item's profit and weight.
        for (int i = 0; i < n; i++) {
            System.out.print("Enter profit and weight for item " + (i + 1) + ": ");
            int profit = sc.nextInt();
            int weight = sc.nextInt();
            items[i] = new ItemValue(profit, weight);
        }

        // Input the knapsack capacity.
        System.out.print("Enter the knapsack capacity: ");
        int capacity = sc.nextInt();

        // Calculate and display the maximum profit.
        double maxValue = getMaxValue(items, capacity);
        System.out.println("Maximum profit: " + maxValue);

        sc.close();
    }
}
```



Program-12 By applying Greedy Technique, write a program to implement a Minimum cost-spanning tree using Prims and Kruskal.

```
import java.util.*;

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

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

        // Sorting edges based on weight (for PriorityQueue)
        public int compareTo(Edge other) {
            return this.weight - other.weight;
        }
    }

    static int findParent(int[] parent, int node) {
        if (parent[node] != node)
            parent[node] = findParent(parent, parent[node]); // Path Compression
        return parent[node];
    }

    static void union(int[] parent, int[] rank, int u, int v) {
        int rootU = findParent(parent, u);
        int rootV = findParent(parent, v);

        if (rootU != rootV) {
            if (rank[rootU] > rank[rootV])
                parent[rootV] = rootU;
            else if (rank[rootU] < rank[rootV])
                parent[rootU] = rootV;
```

```
        else {
            parent[rootV] = rootU;
            rank[rootU]++;
        }
    }
}

static void kruskalMST(int V, List<Edge> edges) {
    PriorityQueue<Edge> pq = new PriorityQueue<>(edges); // Min-heap of edges
    int[] parent = new int[V];
    int[] rank = new int[V];

    // Initialize disjoint set (each node is its own parent)
    for (int i = 0; i < V; i++) {
        parent[i] = i;
        rank[i] = 0;
    }

    List<Edge> mst = new ArrayList<>();
    int totalCost = 0;

    while (!pq.isEmpty() && mst.size() < V - 1) {
        Edge edge = pq.poll(); // Get the edge with minimum weight
        int rootU = findParent(parent, edge.src);
        int rootV = findParent(parent, edge.dest);

        if (rootU != rootV) { // If adding this edge does NOT form a cycle
            mst.add(edge);
            totalCost += edge.weight;
            union(parent, rank, rootU, rootV);
        }
    }

    // Print the MST
    System.out.println("Minimum Spanning Tree (MST) Edges:");
    for (Edge e : mst) {
        System.out.println(e.src + " - " + e.dest + " (Weight: " + e.weight + ")");
    }
    System.out.println("Total Minimum Cost: " + totalCost);
}

public static void main(String[] args) {
    int V = 5; // Number of vertices
    List<Edge> edges = Arrays.asList(
        new Edge(0, 1, 2),
        new Edge(0, 3, 6),
        new Edge(1, 2, 3),
        new Edge(1, 3, 8),
        new Edge(1, 4, 5),
        new Edge(2, 4, 7),
```

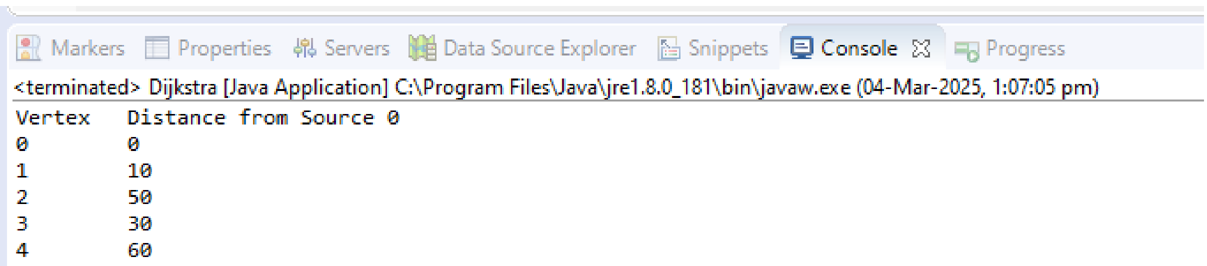
```
            new Edge(3, 4, 9)
        );

        kruskalMST(V, edges);
    }

}
```

```
Markers   Properties   Servers   Data Source Explorer   Snippets   Console   Progress
<terminated> KruskalSimple [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (04-Mar-2025, 9:36:53 am)
Minimum Spanning Tree (MST) Edges:
0 - 1 (Weight: 2)
1 - 2 (Weight: 3)
1 - 4 (Weight: 5)
0 - 3 (Weight: 6)
Total Minimum Cost: 16
```

Program 13 Write a program to implement a Single source shortest path (Dijkstra's algorithm) using the greedy method

```
import java.util.*;

public class Dijkstra {
    static final int INF = Integer.MAX_VALUE; // Represents infinity

    // Function to find the vertex with the minimum distance
    static int minDistance(int[] dist, boolean[] visited, int V) {
        int min = INF, minIndex = -1;
        for (int v = 0; v < V; v++) {
            if (!visited[v] && dist[v] < min) {
                min = dist[v];
                minIndex = v;
            }
        }
        return minIndex;
    }

    // Dijkstra's algorithm using adjacency matrix
    static void dijkstra(int[][] graph, int src) {
        int V = graph.length;
        int[] dist = new int[V]; // Stores the shortest distance from source
        boolean[] visited = new boolean[V]; // Marks processed nodes

        // Initialize distances with INF and source distance as 0
        Arrays.fill(dist, INF);
        dist[src] = 0;

        // Find the shortest path for all vertices
        for (int count = 0; count < V - 1; count++) {
```

```
            int u = minDistance(dist, visited, V); // Pick the min distance node
            visited[u] = true; // Mark as processed

            // Update distances for adjacent vertices
            for (int v = 0; v < V; v++) {
                if (!visited[v] && graph[u][v] != 0 && dist[u] != INF &&
                        dist[u] + graph[u][v] < dist[v]) {
                    dist[v] = dist[u] + graph[u][v];
                }
            }
        }

        // Print the shortest path distances
        printSolution(dist, src);
    }

    // Function to print the shortest distances
    static void printSolution(int[] dist, int src) {
        System.out.println("Vertex \t Distance from Source " + src);
        for (int i = 0; i < dist.length; i++) {
            System.out.println(i + " \t " + (dist[i] == INF ? "INF" : dist[i]));
        }
    }

    public static void main(String[] args) {
        int[][] graph = {
            {0, 10, 0, 30, 100},
            {10, 0, 50, 0, 0},
            {0, 50, 0, 20, 10},
            {30, 0, 20, 0, 60},
            {100, 0, 10, 60, 0}
        };
        int source = 0; // Starting node
        dijkstra(graph, source);
    }
}
```



Working:

1 Graph Representation (Adjacency Matrix)

```
int[][] graph = {
    {0, 10, 0, 30, 100},
    {10, 0, 50, 0, 0},
    {0, 50, 0, 20, 10},
    {30, 0, 20, 0, 60},
    {100, 0, 10, 60, 0}
};
```

- The graph is represented as a **2D matrix (`graph[V][V]`)**.
- If `graph[i][j] = weight`, then there is an **edge** from vertex `i` to `j` with that weight.
- If `graph[i][j] = 0`, **no direct edge** exists.

2 Initializing the Algorithm

```
int[] dist = new int[V]; // Stores shortest distances
boolean[] visited = new boolean[V]; // Marks visited nodes
Arrays.fill(dist, INF); // Set all distances to Infinity
dist[src] = 0; // Distance to the source is 0
```

- `dist[]` holds the **shortest distance from the source vertex** to every other vertex.
- `visited[]` tracks which nodes have been **processed**.
- **All distances are initially set to Infinity (`INF`)**, except for the source node (`dist[src] = 0`).

3 Finding the Minimum Distance Node (`minDistance` function)

```
static int minDistance(int[] dist, boolean[] visited, int V) {
    int min = INF, minIndex = -1;
    for (int v = 0; v < V; v++) {
        if (!visited[v] && dist[v] < min) {
            min = dist[v];
            minIndex = v;
        }
    }
    return minIndex;
}
```

- Finds the **unvisited vertex with the smallest distance**.
- This ensures that we **always process the closest vertex first** (Greedy approach).

4 Processing Each Vertex

```
for (int count = 0; count < V - 1; count++) {
```

```
    int u = minDistance(dist, visited, V); // Pick the closest vertex
    visited[u] = true; // Mark it as visited
```

- Selects the **closest unvisited vertex** (u).
- Marks it as **visited**, so we don't process it again.

5□ Updating Adjacent Nodes

```
for (int v = 0; v < V; v++) {
    if (!visited[v] && graph[u][v] != 0 && dist[u] != INF &&
            dist[u] + graph[u][v] < dist[v]) {
        dist[v] = dist[u] + graph[u][v];
    }
}
```

- **Conditions for updating a vertex v:**
    1. v **is not visited**.
    2. There is an **edge from u to v** (graph[u][v] ≠ 0).
    3. The distance to u **is not INF** (so it's reachable).
    4. The **new distance (dist[u] + graph[u][v]) is smaller** than the current dist[v], so we update it.

6□ Printing the Shortest Distances

```
static void printSolution(int[] dist, int src) {
    System.out.println("Vertex \t Distance from Source " + src);
    for (int i = 0; i < dist.length; i++) {
        System.out.println(i + " \t " + (dist[i] == INF ? "INF" :
dist[i]));
    }
}
```

- Prints the **shortest distances** from the **source vertex** to all other vertices