DAA LAB MANUAL 2024-2025

Design and Analysis of Algorithm

BLDEA's Comm BHS Arts and TGP Science College
Jamkhandi
BCA V Sem

Sl No	Title of the Program	Page No
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

PROGRAM 1: TO SORT A LIST OF N ELEMENTS USING SELECTION SORT TECHNIQUE.

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n − 1 elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the ith pass through the list, which we number from 0 to n − 2, the algorithm searches for the smallest item among the last n − i elements and swaps it with Ai: After n − 1 passes, the list is sorted. Here is pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array:

```
ALGORITHM SelectionSort(A[0..n − 1])
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order for i ← 0 to n − 2 do
min ← i
for j ← i + 1 to n − 1 do
if A[j ] < A[min] min ← j swap A[i] and A[min]
```

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

```
| 89   45   68   90   29   34   17
  17 | 45   68   90   29   34   89
  17   29 | 68   90   45   34   89
  17   29   34 | 90   45   68   89
  17   29   34   45 | 90   68   89
  17   29   34   45   68 | 90   89
  17   29   34   45   68   89 | 90
```

```
public class Lab1
{
        public static void selectionSort(int[] arr)
        {
                for (int i = 0; i < arr.length - 1; i++)
                {
                        int index = i;
                        for (int j = i + 1; j<arr.length; j++)
                        {
                                if (arr[j] < arr[index])
                                {
                                        index = j;
                                }
                        }
                        int smallerNumber = arr[index];
                        arr[index] = arr[i];
                        arr[i] = smallerNumber;
                }
        }
        public static void main(String a[])
        {
                int[] arr1 = {9,14,3,2,43,11,58,22};
                System.out.println("Before Selection Sort");
                for(int i:arr1)
                {
                        System.out.print(i+" ");
                }
                System.out.println();
                selectionSort(arr1);
                System.out.println("After Selection Sort");
                for(int i:arr1)
                {
                        System.out.print(i+" ");
                }
        }
}
```

Output:

```
Before Selection Sort
9 14 3 2 43 11 58 22
After Selection Sort
2 3 9 11 14 22 43 58
```

PROGRAM 2: TO PERFORM TRAVELING SALESMAN PROBLEM

The Traveling Salesman Problem (TSP) can be formally defined as follows: Given a set of cities and the distances between each city, find the shortest possible route that visits each city exactly once and returns to the starting city.

The input to the TSP consists of a set of n cities, where n is an integer greater than 2. Each city is represented by a unique identifier, and the distances between each city are given in a distance matrix, where the ij-th element of the matrix represents the distance between city i and city j.

The output of the TSP is a permutation of the cities, representing the order in which the cities should be visited. The length of the route is calculated as the sum of the distances between each consecutive pair of cities in the permutation. The goal is to find the permutation that minimizes the length of the route.

For the purposes of this article, we will use a simple example with four cities, A, B, C, and D. The distance matrix for this example is given below:

	A	B	C	D
A	0	4	8	7
B	4	0	2	3
C	8	2	0	6
D	7	3	6	0

In this example, the TSP requires finding the shortest route to visit each city exactly once and return to the starting city. One possible route could be A -> D -> B -> C -> A with a total distance of 7 + 3 + 2 + 8 = 20. This means that starting from city A, we visit city D, then city B, then city C, and finally return to city A.

```
import java.util.*;
import java.io.*;
import java.util.Scanner;
class Lab12
{
        static int findHamiltonianCycle(int[][] distance, boolean[] visitCity, int currPos,
        int cities, int count, int cost, int hamiltonianCycle)
        {
                if(count == cities && distance[currPos][0] > 0)
                {
                        hamiltonianCycle    =    Math.min(hamiltonianCycle,    cost    +
                        distance[currPos][0]);
                        return hamiltonianCycle;
                }
                for (int i = 0; i < cities; i++)
                {
                        if (visitCity[i] == false && distance[currPos][i] > 0)
                        {
                                visitCity[i] = true;
                                hamiltonianCycle     =     findHamiltonianCycle(distance,
                                visitCity, i, cities,
                                count + 1, cost + distance[currPos][i], hamiltonianCycle);
                                visitCity[i] = false;
                        }
                }
                return hamiltonianCycle;
        }
        public static void main(String[] args)
        {
                int cities;
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter total number of cities");
                cities = sc.nextInt();
                int distance[][] = new int[cities][cities];
                for(int i = 0; i < cities; i++)
                {
                        for(int j = 0; j < cities; j++)
                        {
                                System.out.println("Distance from city"+ (i+1) +" to city "+
                                        (j+1) +":");
                                distance[i][j] = sc.nextInt();
```

```
                    }
                }
                boolean[] visitCity = new boolean[cities];
                visitCity[0] = true;
                int hamiltonianCycle = Integer.MAX_VALUE;
                hamiltonianCycle = findHamiltonianCycle(distance,visitCity, 0, cities, 1, 0,
                hamiltonianCycle);
                System.out.println(“Minimum Distance : ”+hamiltonianCycle);
        }
}
```

Output:

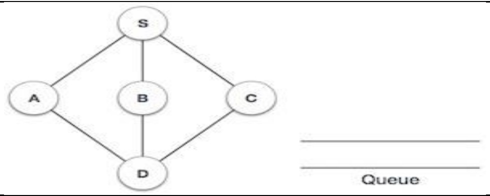
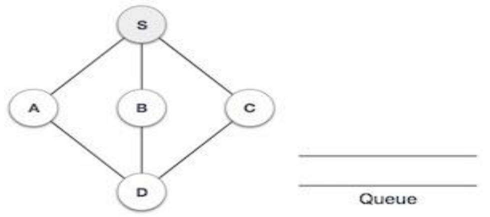
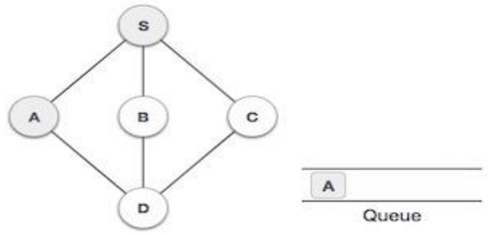
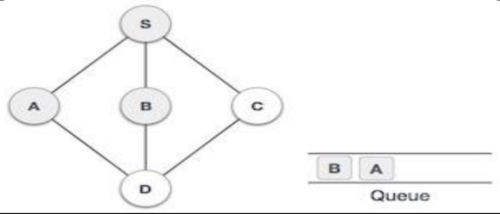
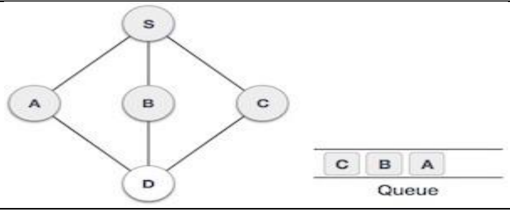
```
Enter total number of cities
4
Distance from city1 to city 1:
0
Distance from city1 to city 2:
3
Distance from city1 to city 3:
5
Distance from city1 to city 4:
2
Distance from city2 to city 1:
3
Distance from city2 to city 2:
0
Distance from city2 to city 3:
4
Distance from city2 to city 4:
6
Distance from city3 to city 1:
5
Distance from city3 to city 2:
4
Distance from city3 to city 3:
0
Distance from city3 to city 4:
5
Distance from city4 to city 1:
2
Distance from city4 to city 2:
2
Distance from city4 to city 3:
5
Distance from city4 to city 4:
0
Minimum Distance : 13
```

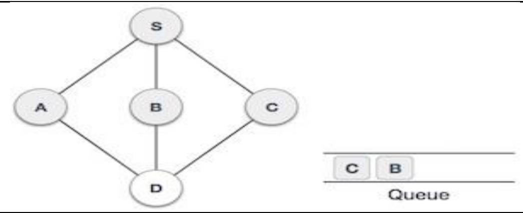
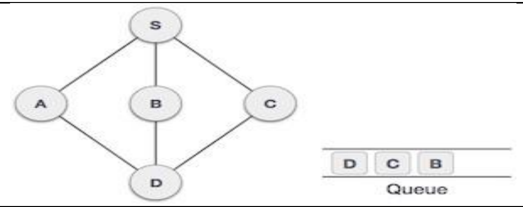
PROGRAM 3: TO IMPEMENT THE BREADTH FIRST SEARCH (BFS) ALGORITHM FOR A GRAPH.

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion to search a graph data structure for a node that meets a set of criteria. It uses a queue to remember the next vertex to start a search, when a dead end occurs in any iteration.

Breadth First Search (BFS) algorithm starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.

- Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.
- Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.

6		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7		From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

```
import java.io.*;
import java.util.*;
class Graph
{
        private int Ver;
        private LinkedList<Integer> adj[];
        private Queue<Integer> queue;
        Graph(int v)
        {
                Ver = v;
                adj = new LinkedList[v];
                for (int i=0; i<v; i++)
                {
                        adj[i] = new LinkedList<>();
                }
                queue = new LinkedList<Integer>();
        }
        void addEdge(int v,int w)
        {
                adj[v].add(w);
        }
        void BFS(int n)
        {
                boolean nodes[] = new boolean[Ver];
                int a = 0;
                nodes[n]=true;
                queue.add(n);
                while (queue.size() != 0)
                {
                        n = queue.poll();
                        System.out.print(n+" ");
                        for (int i = 0; i < adj[n].size(); i++)
                        {
                                a = adj[n].get(i);
                                if (!nodes[a])
                                {
                                        nodes[a] = true;
                                        queue.add(a);
                                }
                        }
                }
        }
        public static void main(String args[])
        {
                Graph graph = new Graph(6);
                graph.addEdge(0, 1);
                graph.addEdge(0, 3);
                graph.addEdge(0, 4);
                graph.addEdge(4, 5);
                graph.addEdge(3, 5);
                graph.addEdge(1, 2);
                graph.addEdge(1, 0);
                graph.addEdge(2, 1);
                graph.addEdge(4, 1);
                graph.addEdge(3, 1);
                graph.addEdge(5, 4);
                graph.addEdge(5, 3);
                System.out.println("The Breadth First Traversal of the graph is as follows
                :");
                graph.BFS(0);
        }
}
```

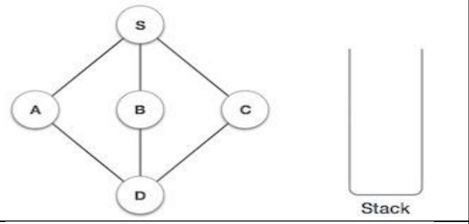
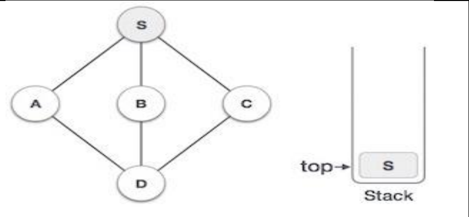
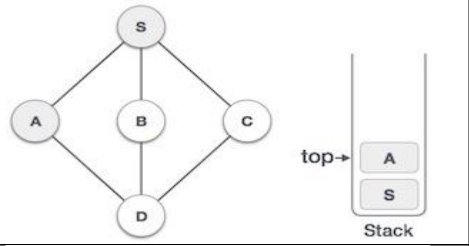
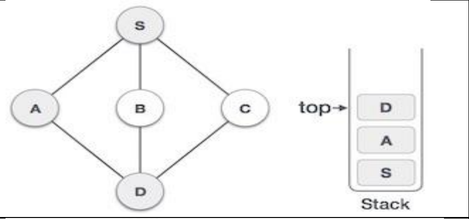
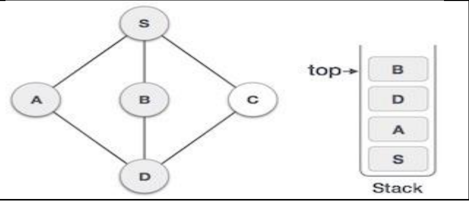
Output:

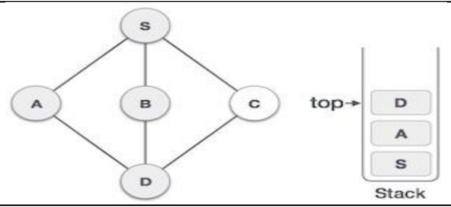
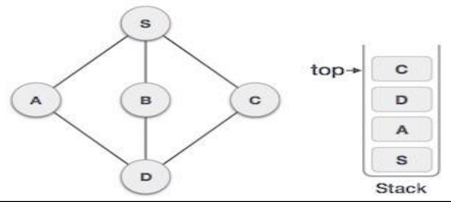
```
The Breadth First Traversal of the graph is as follows :
0 1 3 4 2 5
```

PROGRAM 4: TO IMPLEMENT DEPTH FIRST TRAVERSAL (DFS)

Depth First Search (DFS) algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure. This algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

- Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.
4		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.

6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7		Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.

As C does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

```
import java.util.*;
class Graph2
{
        int Ver;
        LinkedList<Integer>[] adj;
        Graph2(int Ver)
        {
                this.Ver = Ver;
                adj = new LinkedList[Ver];
                for (int i = 0; i < adj.length; i++)
                adj[i] = new LinkedList<Integer>();
        }
        void addEdge(int v, int w)
        {
                adj[v].add(w);
        }
        void DFS(int n)
        {
                boolean nodes[] = new boolean[Ver];
                Stack<Integer> stack = new Stack<>();
                stack.push(n);
                int a = 0;
                while(!stack.empty())
                {
                        n = stack.peek();
                        stack.pop();
                        if(nodes[n] == false)
                        {
                                System.out.print(n + " ");
                                nodes[n] = true;
                        }
                        for (int i = 0; i < adj[n].size(); i++)
                        {
                                a = adj[n].get(i);
                                if (!nodes[a])
                                {
                                        stack.push(a);
                                }
                        }
                }
        }
        public static void main(String[] args)
        {
                Graph2 g = new Graph2(6);
                g.addEdge(0, 1);
                g.addEdge(0, 2);
                g.addEdge(1, 0);
                g.addEdge(1, 3);
                g.addEdge(2, 0);
                g.addEdge(2, 3);
                g.addEdge(3, 4);
                g.addEdge(3, 5);
                g.addEdge(4, 3);
                g.addEdge(5, 3);
                System.out.println("Following is the Depth First Traversal");
                g.DFS(0);
        }
}
```

Output:

```
Following is the Depth First Traversal
0 2 3 5 4 1
```

PROGRAM 5: TO FIND MINIMUM AND MAXIMUM VALUE IN AN ARRAY USING DIVIDE AND CONQURE.

Divide and Conquer (DAC) approach has three steps at each level of recursion:

1. Divide the problem into number of smaller units called sub-problems.
2. Conquer (Solve) the sub-problems recursively.
3. Combine the solutions of all the sub-problems into a solution for the original problem.

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is y−x+1, where y is greater than or equal to x.

Max−Min(x,y) will return the maximum and minimum values of an array numbers[x...y]

```
Algorithm: Max - Min(x, y)
{
        if y – x ≤ 1 then
                return (max(numbers[x],numbers[y]),min((numbers[x],numbers[y]))
        else
                (max1, min1):= maxmin(x, ⌊ ((x + y)/2)⌋ )
                (max2, min2):= maxmin(⌊ ((x + y)/2) + 1)⌋ ,y)
        return (max(max1, max2), min(min1, min2))
}
```

```
class Pair
{
        public int max, min;
        public Pair(int max, int min)
        {
                this.max = max;
                this.min = min;
        }
}
class Main
{
        public static void findMinAndMax(int[] nums, int left, int right, Pair p)
        {
                if (left == right)
                {
                        if (p.max < nums[left])
                        {
                                p.max = nums[left];
                        }
                        if (p.min > nums[right])
                        {
                                p.min = nums[right];
                        }
                        return;
                }
                if (right - left == 1)
                {
                        if (nums[left] < nums[right])
                        {
                                if (p.min > nums[left])
                                {
                                        p.min = nums[left];
                                }
                                if (p.max < nums[right])
                                {
                                        p.max = nums[right];
                                }
                        }
                        else
                        {
                                if (p.min > nums[right])
                                {
                                        p.min = nums[right];
                                }
                                if (p.max < nums[left])
                                {
                                        p.max = nums[left];
                                }
                        }
                        return;
                }
                int mid = (left + right) / 2;
                findMinAndMax(nums, left, mid, p);
                findMinAndMax(nums, mid + 1, right, p);
        }
        public static void main(String[] args)
        {
                int[] nums = { 7,2,4,9 };
                Pair p = new Pair(Integer.MIN_VALUE, Integer.MAX_VALUE);
                findMinAndMax(nums, 0, nums.length - 1, p);
                System.out.println("The minimum array element is " + p.min);
                System.out.println("The maximum array element is " + p.max);
        }
}
```

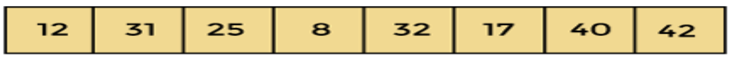
Output:

```
The minimum array element is 2
The maximum array element is 9
```

PROGRAM 6: TO IMPLEMENT MERGE SORT ALGORITHM FOR SORTING A LIST OF INTEGERS IN ASCENDING ORDER.

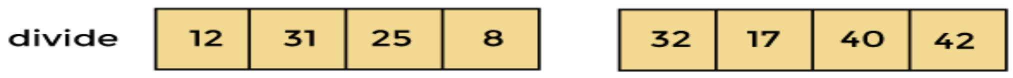
Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller sub arrays and sorting those sub arrays then merging them back together to obtain the sorted array.
In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Working of Merge sort Algorithm

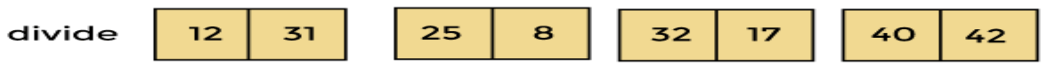


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

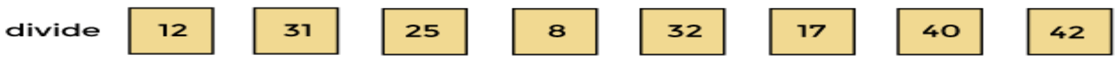
As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



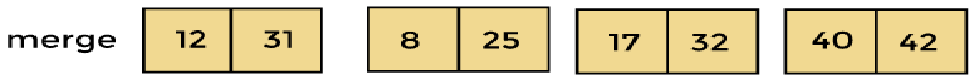
Now, again divide these arrays to get the atomic value that cannot be further divided.



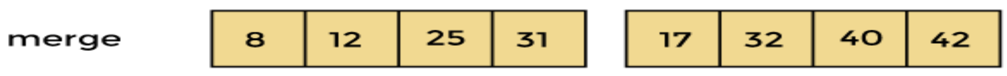
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

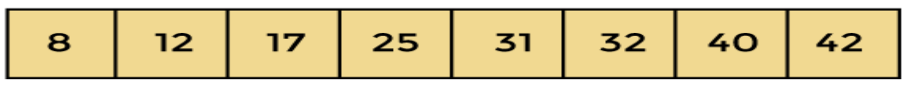
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

```
class Merge
{
        void merge(int a[], int l, int mid, int h)
        {
                int i, j, k;
                int n1 = mid - l + 1;    //1
                int n2 = h - mid;    // 0
                int LeftArray[] = new int[n1];
                int RightArray[] = new int[n2];

                for (i = 0; i < n1; i++)
                {
                        LeftArray[i] = a[l + i];
                }
                for (j = 0; j < n2; j++)
                {
                        RightArray[j] = a[mid + 1 + j];
                }
                i = 0;
                j = 0;
                k = l;

                while (i < n1 && j < n2)
                {
                        if(LeftArray[i] <= RightArray[j])
                        {
                                a[k] = LeftArray[i];
                                i++;
                        }
                        else
                        {
                                a[k] = RightArray[j];
                                j++;
                        }
                        k++;
                }
                while (i<n1)
                {
                        a[k] = LeftArray[i];
                        i++;
                        k++;
                }
                while (j<n2)
                {
                        a[k] = RightArray[j];
                        j++;
                        k++;
                }
        }
        void mergeSort(int a[], int l, int h)
        {
                if (l < h)
                {
                        int mid = (l + h) / 2;
                        mergeSort(a, l, mid);
                        mergeSort(a, mid + 1, h);
                        merge(a, l, mid, h);
                }
        }
        void printArray(int a[], int n)
        {
                int i;
                for (i = 0; i < n; i++)
                {
                        System.out.print(a[i] + " ");
                }
        }

        public static void main(String args[])
        {
                int a[] = { 11, 30, 24, 7, 31, 16, 39, 41 };
                int n = a.length;
                Merge m1 = new Merge();
                System.out.println("\nBefore sorting array elements are - ");
                m1.printArray(a, n);
                m1.mergeSort(a, 0, n - 1);
                System.out.println("\nAfter sorting array elements are - ");
                m1.printArray(a, n);
                System.out.println("");
        }
}
```

Output:

```
Before sorting array elements are -
11 30 24 7 31 16 39 41
After sorting array elements are -
7 11 16 24 30 31 39 41
```

PROGRAM 7: TO IMPLEMENT DIVIDE AND CONQUER STRATEGY FOR QUICK SORT ALGORITHM.

- Quick Sort is a famous sorting algorithm.
- It sorts the given data items in ascending order.
- It uses the idea of divide and conquer approach.
- It follows a recursive algorithm.

How Does Quick Sort Works?

- Quick Sort follows a recursive algorithm.
- It divides the given array into two sections using a partitioning element called as pivot.

The division performed is such that-

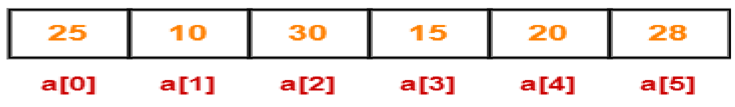
- All the elements to the left side of pivot are smaller than pivot.
- All the elements to the right side of pivot are greater than pivot.

After dividing the array into two sections, the pivot is set at its correct position.

Then, sub arrays are sorted separately by applying quick sort algorithm recursively.

Quick Sort Example-

Consider the following array has to be sorted in ascending order using quick sort algorithm-



25	10	30	15	20	28
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

Quick Sort Example

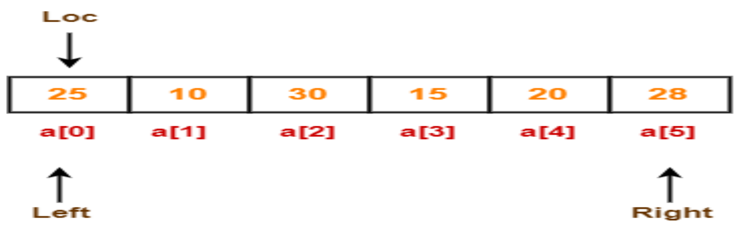
Quick Sort Algorithm works in the following steps-

Step-01:

Initially-

- **Left** and **Loc** (pivot) points to the first element of the array.
- **Right** points to the last element of the array.

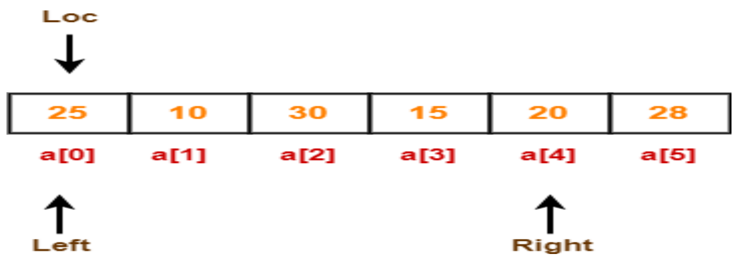
So to begin with, we set **loc** = 0, **left** = 0 and **right** = 5 as-



Step-02:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

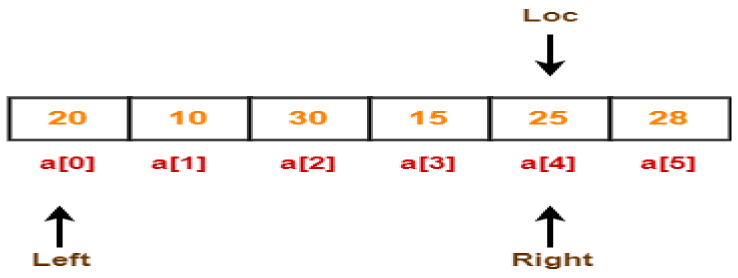


Now, **loc** = 0, **left** = 0 and **right** = 4.

Step-03:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-

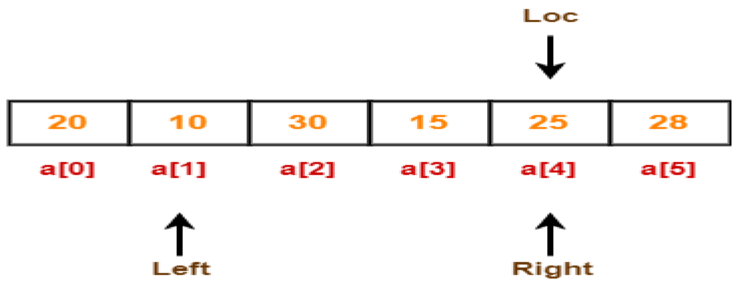


Now, **loc** = 4, **left** = 0 and **right** = 4.

Step-04:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

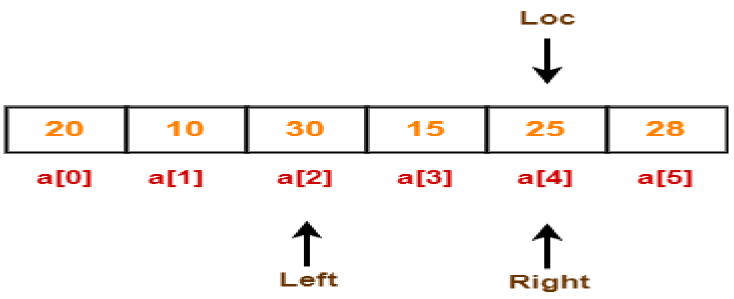


Now, **loc** = 4, **left** = 1 and **right** = 4.

Step-05:

Since **loc** points at right, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

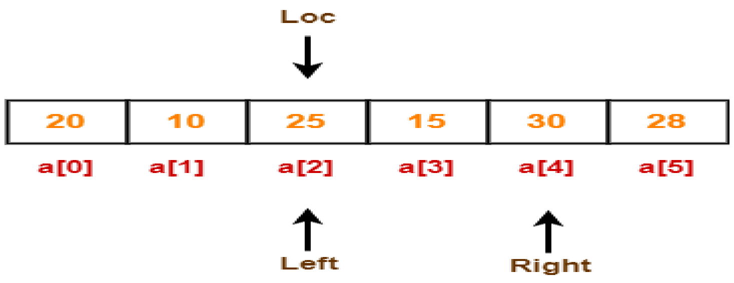


Now, **loc** = 4, **left** = 2 and **right** = 4.

Step-06:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] < a[left], so we algorithm swaps a[loc] and a[left] and **loc** points at **left** as-

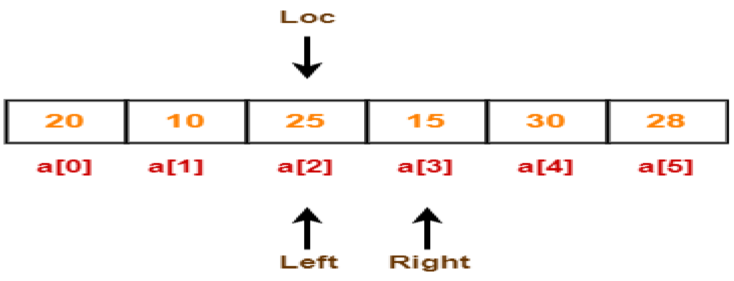


Now, **loc** = 2, **left** = 2 and **right** = 4.

Step-07:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

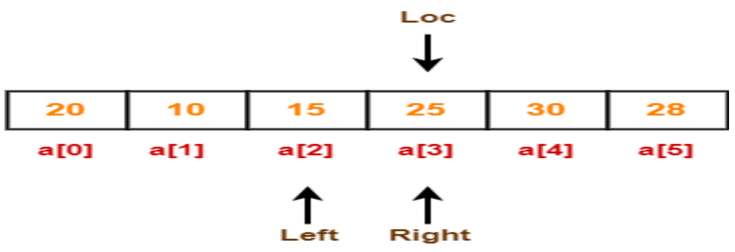


Now, **loc** = 2, **left** = 2 and **right** = 3.

Step-08:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

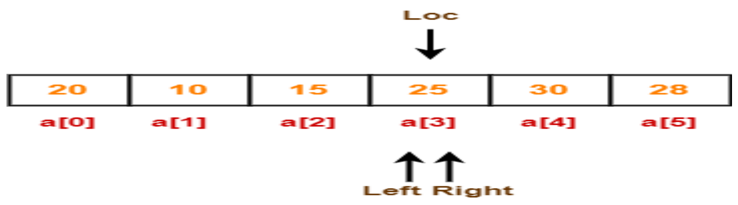
As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-



Now, **loc** = 3, **left** = 2 and **right** = 3.

Step-09:

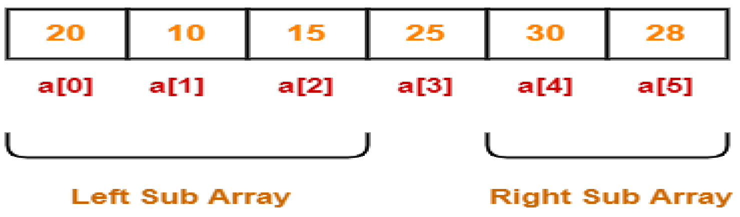
Since **loc** points at **right**, so algorithm starts from **left** and move towards right.
As a[loc] > a[left], so algorithm moves **left** one position towards right as-



Now, **loc** = 3, **left** = 3 and **right** = 3.

Now,

- **loc**, **left** and **right** points at the same element.
- This indicates the termination of procedure.
- The pivot element 25 is placed in its final position.
- All elements to the right side of element 25 are greater than it.
- All elements to the left side of element 25 are smaller than it.



Now, quick sort algorithm is applied on the left and right sub arrays separately in the similar manner.

```
public class QuickSort
{

  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);
    }
  }

  public static int partition(int[] arr, int low, int high)
       {
    int pivot = arr[low];
    int i = low + 1;
    int j = high;

    while (i <= j)
              {
      while (i <= high && arr[i] <= pivot)
                     {
        i++;
      }

      while (arr[j] > pivot)
                     {
        j--;
      }
      if (i < j)
                     {
        swap(arr, i, j);
      }
    }
    swap(arr, low, j);
    return j;
  }

  public static void swap(int[] arr, int i, int j)
       {
    int temp = arr[i];
    arr[i] = arr[j];
```

```
        arr[j] = temp;
    }

    public static void printArray(int[] arr)
            {
                         for (int i = 0; i < arr.length; i++)
                        {
                                    System.out.print(arr[i] + " ");
                         }
                        System.out.println();
            }

    public static void main(String[] args)
            {
                        int[] arr = {10, 7, 8,6,14,2,9};
                          int n = arr.length;

                        System.out.println("Original array:");
                        printArray(arr);
                         quickSort(arr, 0, n - 1);
                        System.out.println("Sorted array:");
                        printArray(arr);
             }
}
```

Output:

```
Original array:
10 7 8 6 14 2 9
Sorted array:
2 6 7 8 9 10 14
```

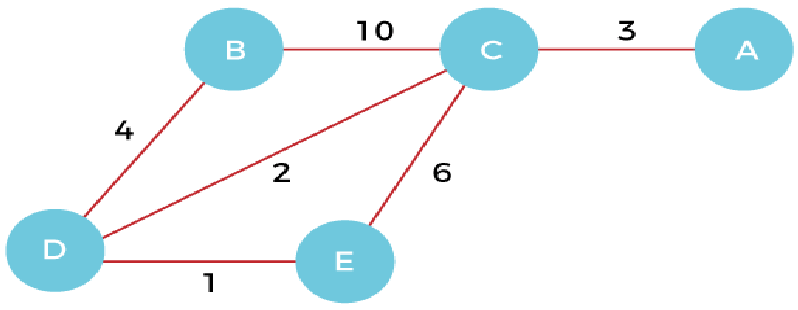
PROGRAM 8: TO IMPLEMENTS PRIMS ALGORITHM TO GENERATE MINIMUM COST SPANNING TREE.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

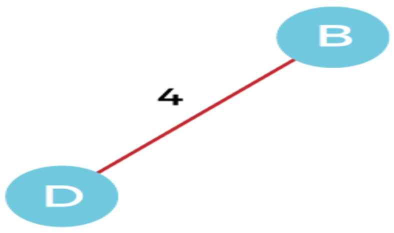
Now, let's see the working of prim's algorithm using an example.

Suppose, a weighted graph is -

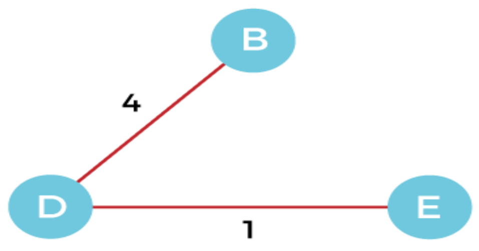


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

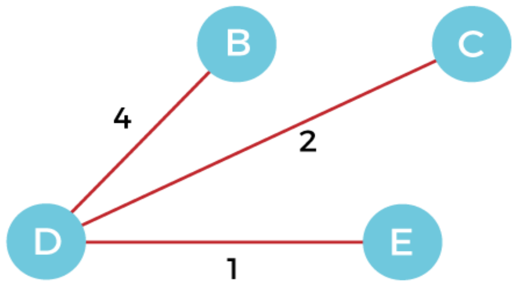
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



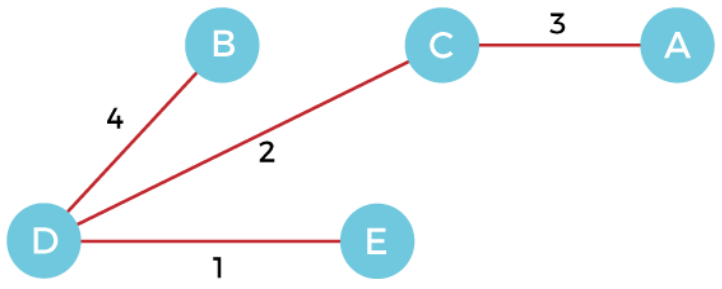
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Algorithm

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has mi nimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

```
import java.util.Scanner;
public class Prims1
{
        static int mincost=0,n,i,j,ne,a=0,b=0,min,u = 0,v=0;
        public void prim(int n,int[][] cost)
        {
                int[] visited = new int[n];
                for(i=1;i<n;i++)
                        visited[i]=0;
                visited[0]=1;
                ne=1;
                while(ne<n)
                {
                        min=999;
                        for(i=0;i<n;i++)
                        {
                                for(j=0;j<n;j++)
                                {
                                        if(cost[i][j]<min)
                                        {
                                                if(visited[i]==0)
                                                        continue;
                                                else
                                                {
                                                        min=cost[i][j];
                                                        a=u=i;
                                                        b=v=j;
                                                }
                                        }
                                }
                        }
                        if(visited[u]==0||visited[v]==0)
                        {
                                System.out.println((ne)+" edge("+a+","+b+") = "+min);
                                ne=ne+1;

                                mincost=mincost+min;

                                visited[v]=1;

                        }
                        cost[a][b]=cost[b][a]=999;
                }
                System.out.println("The minimum cost of spanning tree is "+mincost);
        }
        public static void main(String[ ] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter the number of vertices\n");
                n=sc.nextInt();
                int cost[][]= new int [n][n];
                System.out.println("Enter the cost matrix\n");
                for(i=0;i<n;i++)
                {
                        for(j=0;j<n;j++)
                        {
                                cost[i][j]=sc.nextInt();
                                if(cost[i][j]==0)
                                        cost[i][j]=999;
                        }
                }
                Prims1 p = new Prims1();
                p.prim(n,cost);
        }
}
```

Output:

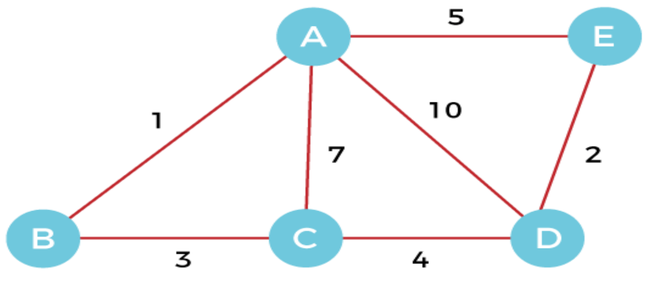
```
Enter the number of vertices
3
Enter the cost matrix

0 10 20
12 0 15
16 18 0
1 edge(0,1) = 10
2 edge(1,2) = 15
The minimum cost of spanning tree is 25
```

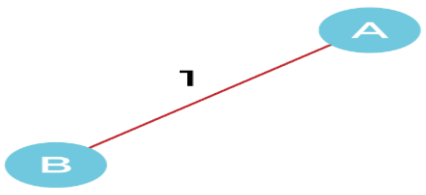
PROGRAM 9: TO IMPLEMENTS KRUSKALS ALGORITHM TO GENERATE MINIMUM COST SPANNING TREE.

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.
Now, let's see the working of Kruskal's algorithm using an example.

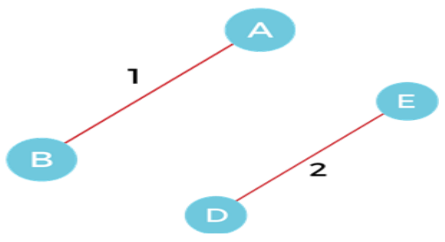
Suppose a weighted graph is -



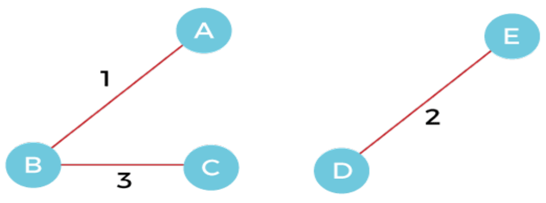
Step 1 - First, add the edge **AB** with weight **1** to the MST.



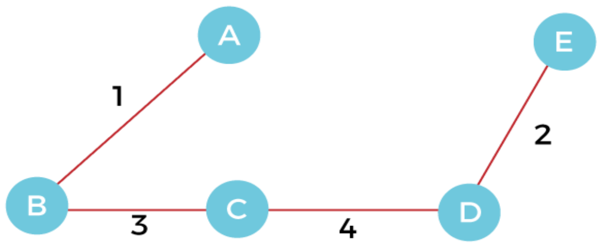
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

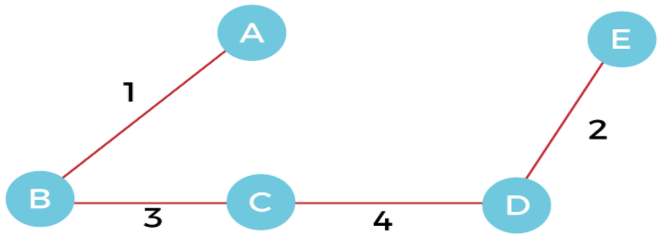


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

```
import java.util.Scanner;
public class KruskalsDemo
{
        final static int MAX = 20;
        static int n;
        static int cost[ ][ ];
        static Scanner scan = new Scanner(System.in);
        public static void main(String[ ] args)
        {
                ReadMatrix();
                Kruskals();
        }
        static void ReadMatrix()
        {
                int i, j;
                cost = new int[MAX][MAX];
                System.out.println("Implementation of Kruskal's algorithm");
                System.out.println("Enter the no. of vertices");
                n = scan.nextInt();
                System.out.println("Enter the cost adjacency matrix");
                for (i = 1; i <= n; i++)
                {
                        for (j = 1; j <= n; j++)
                        {
                                cost[i][j] = scan.nextInt();
                                if (cost[i][j] == 0)
                                        cost[i][j] = 999;
                        }
                }
        }
        static void Kruskals()
        {
                int a = 0, b = 0, u = 0, v = 0, i, j, ne = 1, min, mincost = 0;
                System.out.println("The edges of Minimum Cost Spanning Tree are");
                while (ne < n)
                {
                        for (i = 1, min = 999; i <= n; i++)
                        {
                                for (j = 1; j <= n; j++)
                                {
                                        if (cost[i][j] < min)
                                        {
```

```
                                        min = cost[i][j];
                                        a = u = i;
                                        b = v = j;
                                    }
                                }
                            }
                            u = find(u);
                            v = find(v);
                            if (u != v)
                            {
                                uni(u, v);
                                System.out.println(ne++ + "edge (" + a + "," + b + ") =" +
                                min);
                                mincost += min;
                            }
                            cost[a][b] = cost[b][a] = 999;
                        }
                        System.out.println("Minimum cost :" + mincost);
                    }
                    static int find(int i)
                    {
                        int parent[] = new int[9];
                        while (parent[i] == 1)
                            i = parent[i];
                        return i;
                    }
                    static void uni(int i, int j)
                    {
                        int parent[] = new int[9];
                        parent[j] = i;
                    }
                }
```

Output:

```
Enter the no. of vertices
3
Enter the cost adjacency matrix
0 10 20
12 0 15
16 18 0
The edges of Minimum Cost Spanning Tree are
1edge (1,2) =10
2edge (2,3) =15
Minimum cost :25
```

PROGRAM 10: TO PERFORM KNAPSACK PROBLEM USING GREEDY SOLUTION

The 0/1 Knapsack Problem cannot be solved by a greedy algorithm because it does not fulfill the greedy choice property, and the optimal substructure property, as mentioned earlier.

The 0/1 Knapsack Problem

Rules:

1. Every item has a weight and value.
2. Your knapsack has a weight limit.
3. Choose which items you want to bring with you in the knapsack.
4. You can either take an item or not, you cannot take half of an item for example.

Goal: Maximize the total value of the items in the knapsack.

```
public class Knapsak
{
  static int n = 5;
  static int p[] = {3, 3, 2, 5, 1};
  static int w[] = {10, 15, 10, 12, 8};
  static int W = 10;
  public static void main(String args[])
  {
    int cur_w;
    float tot_v = 0;
    int i, maxi;
    int used[] = new int[10];
    for (i = 0; i < n; ++i)
      used[i] = 0;
    cur_w = W;
    while (cur_w > 0)
          {
      maxi = -1;
      for (i = 0; i < n; ++i)
        if ((used[i] == 0) && ((maxi == -1) || ((float)w[i]/p[i] > (float)w[maxi]/p[maxi])))
          maxi = i;
      used[maxi] = 1;
      cur_w -= p[maxi];
      tot_v += w[maxi];
      if (cur_w >= 0)
                    System.out.println("Added object " + maxi + 1 + " (" + w[maxi] + "," +
                    p[maxi] + ") completely in the bag. Space left: " + cur_w);
      else
         {
                    System.out.println("Added " + ((int)((1 + (float)cur_w/p[maxi]) * 100)) +
                    "% (" + w[maxi] + "," + p[maxi] + ") of object " + (maxi + 1) + " in the
                    bag.");
                     tot_v -= w[maxi];
                    tot_v += (1 + (float)cur_w/p[maxi]) * w[maxi];
         }
    }
    System.out.println("Filled the bag with objects worth " + tot_v);
  }
}
```

Output:

```
Added object 41 (8,1) completely in the bag. Space left: 9
Added object 11 (15,3) completely in the bag. Space left: 6
Added object 21 (10,2) completely in the bag. Space left: 4
Added object 01 (10,3) completely in the bag. Space left: 1
Added 19% (12,5) of object 4 in the bag.
Filled the bag with objects worth 45.4
```