2. Sort a given set of elements using Merge sort and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot graph of the time taken versus number of elements. The elements can be read from file or generated using random number generator.

Merge sort is the sorting technique that follows the divide and conquer approach. The important part of the merge sort is the **MERGE** function. This function performs the merging of two sorted sub-arrays that are **A[beg…mid]** and **A[mid+1…end]**, to build one sorted array **A[beg…end]**. So, the inputs of the **MERGE** function are **A[], beg, mid,** and **end**.

Time Complexity: O(n log (n))

```
Algorithm Merge_Sort(arr,beg,end)
If beg<end
Set mid==(beg+ end)/2 Merge_Sort(arr,beg,mid) Merge_Sort(arr, mid+1,end)
Merge(arr, beg, mid, end)
End if
END Merge_Sort
Program
import java.util.Scanner;
import java.util.Arrays;
import java.util.Random;
public class PracMSort {

        public static void mergesort(int[] arr, int l, int r) {

                if(l<r) {
                        int m = l+(r-l)/2;
                        mergesort(arr,l,m);
                        mergesort(arr,m+1,r);
                        merge(arr,l,m,r);
                }

        }
```

```
public static void merge(int[] arr, int l,int m, int r) {
                int n1 = m-l+1;
                int n2 = r-m;
                int[]la=new int [n1];
                int[]ra=new int [n2];
for(int i=0; i<n1;i++)
                        la[i]=arr[l+i];
```

```
        for(int i=0; i<n2;i++)
                ra[i]=arr[(m+1)+i];

        int i=0,j=0,k=l;

        while(i<n1 && j<n2) {

                if(la[i]<ra[j]) {
                        arr[k++]=la[i++];
                }
                else {
                        arr[k++]=ra[j++];
                }
        }
        while(i<n1)
        {
                arr[k++]=la[i++];
        }

        while(j<n2)
        {
                arr[k++]=ra[j++];
        }
    }
    public static void main(String[] args) {
        int n;
        Scanner sc=new Scanner(System.in);
        Random r= new Random();
        System.out.println("Enter the number of elements: ");
        n=sc.nextInt();
        sc.close();
        int[] arr=new int[n];

        for(int i=0;i<n;i++) {
                arr[i]=r.nextInt(100);
        }

System.out.println("Original array: "+Arrays.toString(arr));
long st=System.nanoTime();
mergesort(arr,0,n-1);
long et=System.nanoTime();
```

```
System.out.println("Sorted array: "+Arrays.toString(arr)+ '\n'+"Time Complexity: "+ ((et-st)/1e6)+" milliseconds");

        }
}
```

Output

Enter the number of elements:

5

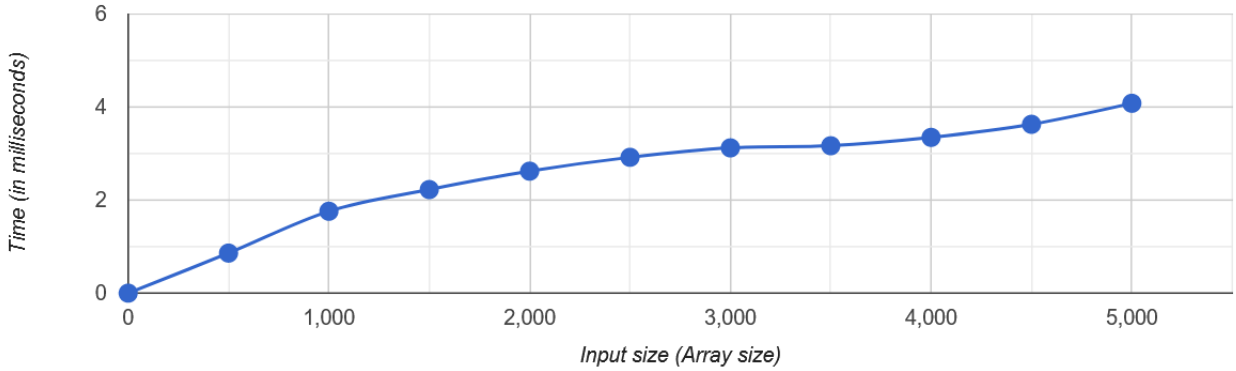
Original array: [73, 44, 54, 70, 25]

Sorted array: [25, 44, 54, 70, 73]

Time Complexity: 0.0131 milliseconds

Graph

Enter the number of elements: 500 Time Complexity: 0.8605 milliseconds
Enter the number of elements: 1000 Time Complexity: 1.7600 milliseconds
Enter the number of elements: 1500 Time Complexity: 2.2260 milliseconds
Enter the number of elements: 2000 Time Complexity: 2.6214 milliseconds
Enter the number of elements: 2500 Time Complexity: 2.9194 milliseconds
Enter the number of elements: 3000 Time Complexity: 3.1217 milliseconds
Enter the number of elements: 3500 Time Complexity: 3.1676 milliseconds
Enter the number of elements: 4000 Time Complexity: 3.3472 milliseconds
Enter the number of elements: 4500 Time Complexity: 3.6285 milliseconds
Enter the number of elements: 5000 Time Complexity: 4.0799 milliseconds



3. Sort a given set of elements using Quick sort and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot graph of the time taken versus number of elements. The elements can be read from file or generated using random number generator.

   Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value.

   Time Complexity: O(n log (n))

```
Algorithm:
QUICKSORT (array A, start, end)
{
if (start < end) {
p = partition(A, start, end) QUICKSORT (A, start, p - 1) QUICKSORT (A, p + 1, end)
}
}
Partition Algorithm:
PARTITION (array A, start, end) { pivot ? A[end] i ? start-1
for j ? start to end -1 { do if (A[j] < pivot) { then i ? i + 1swap A[i] with A[j]
}}
swap A[i+1] with A[end]
return i+1
}
```

```
Program:
import java.util.Scanner;
import java.util.Arrays;
import java.util.Random;
public class PracQSort {
        public static void quicksort(int[] a, int l, int h) {
                if(l<h) {
                        int pi=part(a,l,h);
                        quicksort(a,l,pi-1);
                        quicksort(a,pi+1,h);
                }
        }
        public static int part(int[] a,int l, int h) {
                int p = a[h];
                int i = l-1;//since we add one later
```

```
                for(int j=l;j<h;j++) {
                        if(a[j]<p) {
                                i++;
                                int t = a[i];
                                a[i]=a[j];
                                a[j]=t;
                        }
                }
                int t = a[i+1];
                a[i+1]=a[h];
                a[h]=t;
                return i+1;
        }
        public static void main(String[] args) {
                int n;
                Scanner sc=new Scanner(System.in);
                Random r= new Random();
System.out.println("Enter the number of elements:");
                n=sc.nextInt();
                sc.close();
                int[] arr=new int[n];
                for(int i=0;i<n;i++) {
                        arr[i]=r.nextInt(10000);
                }

System.out.println("Original array: "+Arrays.toString(arr));
long st=System.nanoTime();
quicksort(arr,0,n-1);
long et=System.nanoTime();
System.out.println("Sorted array: "+Arrays.toString(arr));
System.out.println("Time Complexity: "+(et-st)/1e6+" milliseconds");
        }
}
```

Output

Enter the number of elements:

5

Original array: [3709, 2416, 2447, 7986, 3813]

Sorted array: [2416, 2447, 3709, 3813, 7986]

4. Write a program to perform insert and delete operations in Binary Search Tree. Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

   Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match

```
class BinarySearchTree { static class Node {
        int key;
        Node left, right;
        public Node(int item) {
        key = item; left = right = null;
    }
} // Root of the BST Node root;
 Node root;
BinarySearchTree()
{ root = null;}

 void insert(int key) {
        root = insertRec(root, key);
}
Node insertRec(Node root, int key) {
         if (root == null) {
        root = new Node(key); return root;
}
        if (key < root.key) {
        root.left = insertRec(root.left, key);}
        else if (key > root.key) {
        root.right = insertRec(root.right, key);
}
return root;
} // Delete a key from the BST
void delete(int key) {
root = deleteRec(root, key);
}
```

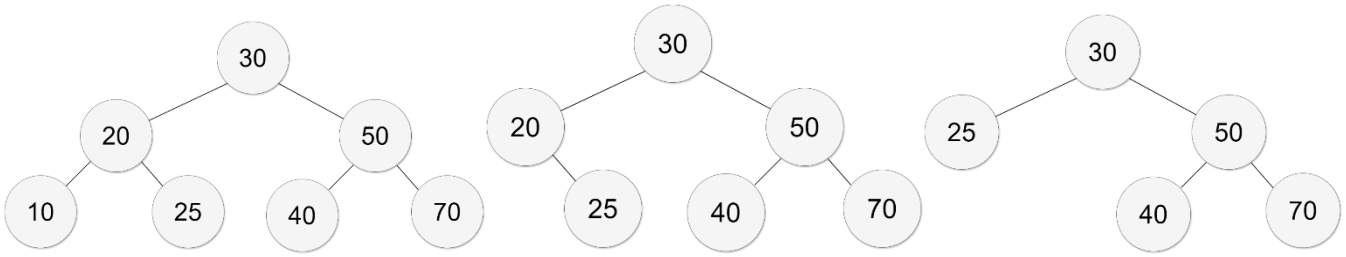
```
Node deleteRec(Node root, int key)
{
  if (root == null) {
  return root;
}
if (key < root.key) {
        root.left = deleteRec(root.left, key);
}
else if (key > root.key) {
        root.right = deleteRec(root.right, key);
}
else {
        if (root.left == null) {
        return root.right;
}
else if (root.right == null) {
        return root.left;
} // Node with two children
        root.key = minValue(root.right); // Delete the in-order successor
        root.right = deleteRec( root.right, root.key);
}
return root;
}
int minValue(Node root)
{
         int minValue = root.key;
        while (root.left != null) {
         minValue = root.left.key;
         root = root.left;
}
return minValue;
} // Print the inorder traversal of the tree
void inorder() {
        inorderRec(root);
}
void inorderRec(Node root) {
        if (root != null) {
         inorderRec(root.left);
```

```
        System.out.print(root.key + " ");
        inorderRec(root.right);
}
} // Main method for testing the BST operations
public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree(); // Insert elements
        tree.insert(30); tree.insert(20); tree.insert(10); tree.insert(25); tree.insert(50);
tree.insert(40); tree.insert(70);
        System.out.println("Inorder traversal:");
        tree.inorder();
        System.out.println(); // Delete elements
        System.out.println("Delete 10:");
        tree.delete(10);
        tree.inorder();
        System.out.println(); System.out.println("Delete 20:");
        tree.delete(20);
        tree.inorder();
        System.out.println();
}
}
```

Output

Inorder traversal:

10 20 25 30 40 50 70
Delete 10:
20 25 30 40 50 70
Delete 20:
25 30 40 50 70



5. Print all the nodes reachable from a given starting node in a digraph using BFS method.

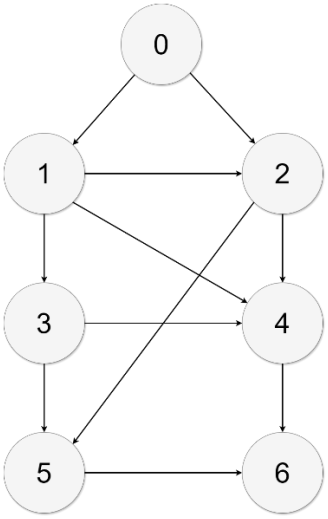
BFS (breadth-first search) is an algorithm that is used for traversing or searching a graph or *tree data* structure. It starts at the *root node* (or any *arbitrary node*) and explores all the nodes at the current depth level before moving on to the nodes at the *next depth level*.

```
import java.util.*;
class Graph {
        private int vertices;
        private LinkedList<Integer>[] adjacencyList;
        public Graph(int vertices) {
                        this.vertices = vertices;
                        adjacencyList = new LinkedList[vertices];
                        for (int i = 0; i < vertices; ++i) {
                                adjacencyList[i] = new LinkedList<>();
                         }
                }
public void addEdge(int v, int w) {
           adjacencyList[v].add(w);
}
public void printReachableNodes(int startNode) {
           boolean[] visited = new boolean[vertices];
           LinkedList<Integer> queue = new LinkedList<>();
           visited[startNode] = true;
           queue.add(startNode);
           System.out.println("Nodes reachable from node " + startNode + " are:");
           while (!queue.isEmpty()) {
                        startNode = queue.poll();
                        System.out.print(startNode + " ");
                        Iterator<Integer> iterator = adjacencyList[startNode].listIterator();
                        while (iterator.hasNext()){
                         int nextNode = iterator.next();
                         if (!visited[nextNode]) {
                           visited[nextNode] = true;
                            queue.add(nextNode);    }
                       }
            }
}
public static void main(String[] args) {
                    Graph graph = new Graph(7);
```

```
                graph.addEdge(0, 1);
                graph.addEdge(0, 2);
                graph.addEdge(1, 3);
                graph.addEdge(1, 4);
                graph.addEdge(2, 5);
                graph.addEdge(2, 6);
                int startNode = 0;
                graph.printReachableNodes(startNode);
        }
}
```

Output

Nodes reachable from node 0 are:
0 1 2 3 4 5 6



Viva Questions

1. What is Breadth-First Search (BFS)?

   BFS is a graph traversal algorithm that explores the graph level by level, starting from a given node. It visits all the neighbors of the starting node before moving on to the neighbors' neighbors.

2. How does BFS work in a directed graph (digraph)?

   In a digraph, BFS starts at a given starting node and explores all reachable nodes by following directed edges. It uses a queue to explore each node's neighbors in a level-order manner.

3. What are the applications of BFST?
   Dynamic data structures like symbol tables in compilers. Indexing in database systems. Maintaining hierarchical structures (e.g., file systems). Implementing associative arrays for memory-efficient storage.

6.a) Obtain the topological ordering of vertices in a given digraph.

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices in which u occurs before v in the ordering for every directed edge uv from vertex u to vertex v. For example, the graph's vertices could represent jobs to be completed, and the edges could reflect requirements that one work must be completed before another.

```
import java.util.*;
public class TopologicalSort {
        private int V; // Number of vertices
        private List<Integer> adjList[];
        public TopologicalSort(int v) {
                V = v; adjList = new LinkedList[v];
                for (int i = 0; i < v; ++i)
                        adjList[i] = new LinkedList<>();
        }
        private void addEdge(int v, int w) {
            adjList[v].add(w);
        }
        private void topologicalSortUtil(int v, boolean visited[], Stack stack) {
                visited[v] = true;
                for (Integer neighbor : adjList[v]) {
                        if (!visited[neighbor])
                        topologicalSortUtil(neighbor, visited, stack);
                }
                stack.push(v);
         }
                private void topologicalSort() {
                        Stack stack = new Stack<>();
                        boolean visited[] = new boolean[V];
                        Arrays.fill(visited, false);
                        for (int i = 0; i < V; ++i) {
                                if (!visited[i])
                                topologicalSortUtil(i, visited, stack);
                        }
                        System.out.println("Topological Sort:");
                        while (!stack.isEmpty())
                                System.out.print(stack.pop() + " ");
                 }
        public static void main(String args[]) {
```

```
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        int V = scanner.nextInt();
        TopologicalSort g = new TopologicalSort(V);
        System.out.println("Enter the adjacency matrix:");
        for (int i = 0; i < V; i++) {
                for (int j = 0; j < V; j++) {
        if (scanner.nextInt() == 1) {
        g.addEdge(i, j);    }
                }
        }
        g.topologicalSort(); scanner.close();
    }
}
```

Output

Enter the number of vertices: 4 Enter the adjacency matrix:

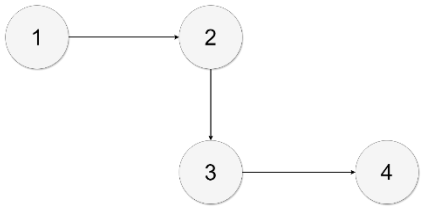
0 1 0 0

0 0 1 0

0 0 0 1

0 1 0 0

Topological Sort: 1 2 3 4



6b) Compute the transitive closure of a given directed graph using Warshall's algorithm

Warshall's algorithm is used to determine the transitive closure of a directed graph or all paths in a directed graph by using the adjacency matrix. It generates a sequence of n matrices, where n is the number of vertices. The kth matrix (R(k)) contains the definition of the element at the with row and jth column, which will be one if it contains a path from v_i to v_j. For all intermediate vertices, w_q is among the first k vertices that mean $1 \leq q \leq k$. The R(0) matrix is used to describe the path without any intermediate vertices. So we can say that it is an adjacency matrix. The R(n) matrix will contain ones if it contains a path between vertices with intermediate vertices from any of the n vertices of a graph. So we can say that it is a transitive closure.

```
import java.util.Scanner;
public class WarshallsAlgorithm {
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
// Get the number of vertices
System.out.print("Enter the number of vertices: ");
int vertices = scanner.nextInt();
// Initialize the adjacency matrix
int[][] graph = new int[vertices][vertices];
// Get the adjacency matrix from the user
System.out.println("Enter the adjacency matrix (0 for no edge, 1 for edge):");
for (int i = 0; i < vertices; i++) {
for (int j = 0; j < vertices; j++) { graph[i][j] = scanner.nextInt();
}
}
// Find the transitive closure using Warshall's Algorithm

for (int k = 0; k < vertices; k++) {
for (int i = 0; i < vertices; i++) {
for (int j = 0; j < vertices; j++) {
graph[i][j] = graph[i][j] | (graph[i][k] & graph[k][j]);
}
}
}
System.out.println("Transitive Closure:");
for (int i = 0; i < vertices; i++) {
for (int j = 0; j < vertices; j++)
{ System.out.print(graph[i][j] + " ");
}
System.out.println();
}
```

```
scanner.close();
}
}
```

Output

Enter the number of vertices: 4
Enter the adjacency matrix (0 for no edge, 1 for edge):

0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0

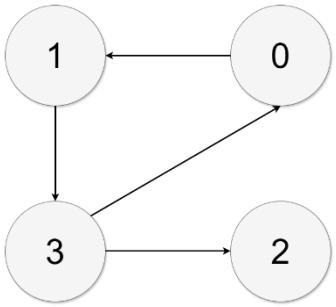
Transitive Closure:

1 1 1 1

1 1 1 1

0 0 0 0

1 1 1 1



Viva Questions

1. What is a topological ordering of a graph?

   Topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge from vertex u to vertex v, u comes before v in the ordering. This ordering is only possible for Directed Acyclic Graphs (DAGs).

2. What is the transitive closure of a graph?

   The transitive closure of a directed graph is a matrix that indicates whether there is a path between any pair of vertices. In this matrix, an entry (i, j) is true (or 1) if there is a path from vertex i to vertex j, and false (or 0) otherwise.

3. How does Warshall's algorithm update the adjacency matrix of a graph?

   Warshall's algorithm updates the adjacency matrix by considering each vertex k as an intermediate node. For every pair of vertices i and j, it checks if there is a path from i to j via k. If such a path exists, it updates the matrix entry at [i][j] to 1

4. What are the practical applications of computing the transitive closure of a graph?

   Applications
   1. Analyzing connectivity in social networks.
   2. Finding reachability in a web of hyperlinked documents.
   3. Traffic network analysis to find accessible nodes.
   4. Determining inheritance relationships in object-oriented programming.

7. Sort a given set of elements using the Heap sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus number of elements.

Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure. It is similar to selection sort, where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements

```
import java.util.Random;
import java.util.Scanner;
class HeapSort {
   static void heapify(int a[], int n, int i) {
      int largest = i;
      int left = 2 * i + 1;
      int right = 2 * i + 2;
      if (left < n && a[left] > a[largest])
         largest = left;
      if (right < n && a[right] > a[largest])
         largest = right;
      if (largest != i) {
         int temp = a[i];
         a[i] = a[largest];
         a[largest] = temp;
         heapify(a, n, largest);
      }
   }
   static void heapSort(int a[], int n) {
      for (int i = n / 2 - 1; i >= 0; i--)
         heapify(a, n, i);
      for (int i = n - 1; i >= 0; i--) {
         int temp = a[0];
         a[0] = a[i];
         a[i] = temp;
         heapify(a, i, 0);
      }
   }
   static void printArr(int a[], int n) {
      for (int i = 0; i < n; ++i)
         System.out.print(a[i] + " ");
      System.out.println();
   }
```

```
    public static void main(String args[]) {
        Random rand = new Random();
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the size of array: ");
        int n = sc.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) {
            a[i] = rand.nextInt(100);
        }
        System.out.print("Before sorting array elements are - \n");
        printArr(a, n);
        heapSort(a, n);
        System.out.print("\nAfter sorting array elements are - \n");
        printArr(a, n);
        sc.close();
    }}
```

Output

Before sorting array elements are:

45 7 20 40 25 23 2
After sorting array elements are:

2 7 20 23 25 40 45

Output

Enter the number of elements: 1000 Time Complexity: 1.463 milliseconds

Enter the number of elements: 2000 Time Complexity: 1.253 milliseconds

Enter the number of elements: 3000 Time Complexity: 2.1698 milliseconds

Enter the number of elements: 4000 Time Complexity: 2.2928 milliseconds

Enter the number of elements: 5000 Time Complexity: 3.042 milliseconds

Enter the number of elements: 6000 Time Complexity: 4.4061 milliseconds

Enter the number of elements: 7000 Time Complexity: 5.6577 milliseconds

Enter the number of elements: 8000 Time Complexity: 3.7093 milliseconds

Enter the number of elements: 9000 Time Complexity: 4.1508 milliseconds

Enter the number of elements: 10000 Time Complexity: 4.9808 milliseconds

8. Search for a pattern string in a given text using Horspool String Matching algorithm.

Horspool's algorithm is a string matching algorithm that compares characters from the end of the pattern to its beginning. When characters do not match, searching jumps to the next matching position in the pattern.

```
class AWQ{
static int NO_OF_CHARS = 256;
static int max (int a, int b) { return (a > b)? a: b; } static void badCharHeuristic( char []str,
int size,int
badchar[])
{
for (int i = 0; i < NO_OF_CHARS; i++) badchar[i] = -1;
for (int i = 0; i < size; i++) badchar[(int) str[i]] = I;
}
static void search( char txt[], char pat[])
{
int m = pat.length; int n = txt.length;
int badchar[] = new int[NO_OF_CHARS]; badCharHeuristic(pat, m, badchar); int s = 0;
while(s <= (n - m))
{
int j = m-1;
while(j >= 0 && pat[j] == txt[s+j]) j--;
if (j < 0)
{
System.out.println("Patterns occur at shift = " + s); s += (s+m < n)? m-badchar[txt[s+m]]
: 1;
}else
s += max(1, j - badchar[txt[s+j]]);
}
}
public static void main(String []args) { char txt[] = "ABAAABCD".toCharArray(); char
pat[] = "ABC".toCharArray(); search(txt, pat);
}
}
```

Output

Pattern occur at shift=4

9. Implement 0/1 Knapsack problem using dynamic programming.

The 0/1 Knapsack problem is a classic optimization problem in computer science. It is a problem of combinatorial optimization, where we have a set of items, each with a weight and a value, and we want to determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible

```
class Knapsack {
static int max(int a, int b) { return (a > b) ? a : b;
}
static int knapSack(int W, int wt[], int val[], int n)
{
if (n == 0 || W == 0) return 0;
if (wt[n - 1] > W)
return knapSack(W, wt, val, n - 1);
else
return max(val[n - 1] + knapSack(W - wt[n -1], wt,val, n - 1), knapSack(W, wt, val, n -
1));
}
public static void main(String args[])
{
int profit[] = new int[] { 60, 100, 120 }; int weight[] = new int[] { 10, 20, 30 }; int W =
50;
int n = profit.length; System.out.println(knapSack(W, weight, profit, n));
}
}
```

Output

220

Viva Questions

1. What is the 0/1 Knapsack problem?

The 0/1 Knapsack problem is a combinatorial optimization problem where given a set of items, each with a weight and value, the goal is to determine the maximum value that can be carried in a knapsack of fixed capacity, with the constraint that each item can either be included (1) or excluded (0).

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

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. It starts with an arbitrary vertex and adds the minimum weight edge to the tree at each step until all vertices are included in the tree. The algorithm maintains two sets of vertices: one set contains the vertices already included in the minimum spanning tree, and the other set contains the vertices not yet included. At each step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing the minimum spanning tree. The algorithm stops when all vertices are included in the minimum spanning tree

```
import java.util.Scanner;

public class Prims {
   public static void main(String[] args) {
      int w[][] = new int[10][10];
      int n, i, j, s, k = 0, min, sum = 0, u = 0, v = 0;
   int flag = 0;
      int sol[] = new int[10];

      Scanner sc = new Scanner(System.in);

      System.out.println("Enter the number of vertices:");
      n = sc.nextInt();

      // Initialize the solution array
      for (i = 1; i <= n; i++) sol[i] = 0;

      System.out.println("Enter the weighted graph:");
      for (i = 1; i <= n; i++) {
         for (j = 1; j <= n; j++) {
            w[i][j] = sc.nextInt();
         }
      }

      System.out.println("Enter the source vertex:");
      s = sc.nextInt();
      sol[s] = 1;

      k = 1;
      while (k <= n - 1) {
         min = 99; // Set min to a high value for comparison
```

```
        // Find the smallest edge from the current tree to the next vertex
        for (i = 1; i <= n; i++) {
            for (j = 1; j <= n; j++) {
                if (sol[i] == 1 && sol[j] == 0 && i != j) {
                    if (min > w[i][j]) {
                        min = w[i][j];
                        u = i;
                        v = j;
                    }
                }
            }
        }

        sol[v] = 1; // Include vertex v in the solution
        sum += min; // Add the weight of the selected edge to the total cost
        k++; // Increment the edge count
        System.out.println(u + " -> " + v + " = " + min);
    }

    // Check if all vertices are included in the solution
    for (i = 1; i <= n; i++) {
        if (sol[i] == 0) {
            flag = 1;
        }
    }

    if (flag == 1) {
        System.out.println("No spanning tree");
    } else {
        System.out.println("The cost of minimum spanning tree is " + sum);
    }

    sc.close();
  }
}
```

Output

Enter the number of vertices 3

Enter the weighted graph
0 20 30

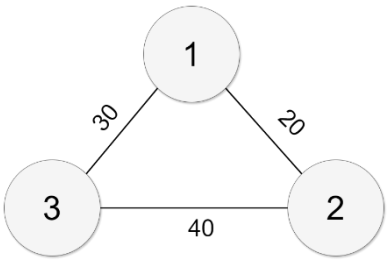
20 0 40

30 40 0

Enter the source vertex 1
1->2=20

1->3=30

The cost of minimum spanning tree is 50



Viva Questions

1. What is Prim's algorithm?

   Prim's algorithm is a greedy algorithm used to find the Minimum Cost Spanning Tree (MST) of a connected, weighted graph. It starts with a single vertex and grows the MST by adding the smallest edge that connects a vertex in the tree to a vertex outside the tree until all vertices are included.

2. What is the time complexity of Prim's algorithm?

   The time complexity of Prim's algorithm depends on the data structures used:

   1. Using an adjacency matrix and simple arrays: $O(V^2)$, where V is the number of vertices.
   2. Using a priority queue with a binary heap: O(E log V), where E is the number of edges.

3. How does Prim's algorithm ensure that the spanning tree remains acyclic?

   Prim's algorithm adds edges in a way that always connects a vertex in the MST to a vertex outside the MST. By only adding the minimum edge that connects the two sets, it ensures that no cycles are formed, thus maintaining the acyclic nature of the spanning tree.

4. What are some real-time applications of Prim's algorithm?

   1. **Network Design:** Optimizing the layout of network cables or telecommunications to minimize costs.
   2. **Infrastructure Development**: Designing water supply networks or electrical grids to ensure minimal piping or wiring costs.
   3. **Transportation Networks:** Creating efficient road or rail networks to connect multiple cities with the least total construction cost.
   4. **Cluster Analysis:** In data mining, forming clusters of data points that minimize the total distance between points, akin to building a minimum spanning tree.

11. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. It starts by sorting all the edges in non-decreasing order of their weight. It then adds the smallest edge to the minimum spanning tree and checks if it forms a cycle with the edges already in the tree. If it does not form a cycle, the edge is included in the minimum spanning tree. If it forms a cycle, the edge is discarded. This process is repeated until all vertices are included in the minimum spanning tree

```
import java.util.Scanner;
public class Kruskal {
        int parent[] = new int[10];
        int find(int m) {
        while (parent[m] != 0) {
        m = parent[m];
        }
        return m;
        }
        void union(int i, int j) {
        parent[j] = i;
        }
        void krkl(int a[][], int n) {
        int u = 0, v = 0, min, sum = 0, k = 0;
        while (k < n - 1) {
        min = 99;
        for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                if (a[i][j] < min && i != j) {
                min = a[i][j];
                u = i;
                v = j;
                }
                }
        }
        int i = find(u);
        int j = find(v);
        if (i != j) {
                union(i, j);
                System.out.println("(" + u + "," + v + ") = " + a[u][v]);
                sum += a[u][v];
                k++;
        }
```

```
a[u][v] = a[v][u] = 99;
}
System.out.println("The cost of the minimum spanning tree = " + sum);
}
public static void main(String[] args) {
int a[][] = new int[10][10];
Scanner sc = new Scanner(System.in);
System.out.println("Enter the number of vertices of the graph");
int n = sc.nextInt();
System.out.println("Enter the weighted adjacency matrix");
for (int i = 1; i <= n; i++) {
for (int j = 1; j <= n; j++) {
        a[i][j] = sc.nextInt();
}
}
Kruskal k = new Kruskal();
k.krkl(a, n);
sc.close();
}}
```

Output

Enter the number of vertices of the graph 4

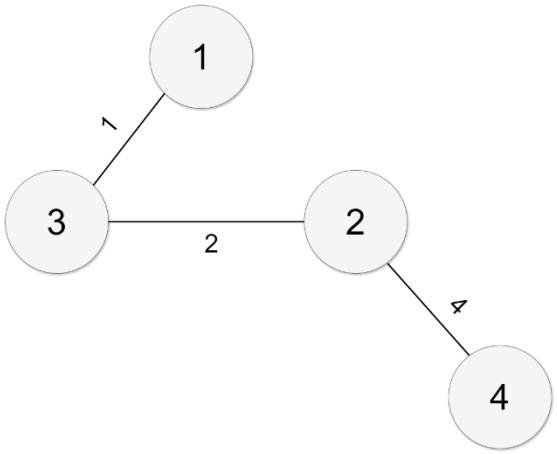
Enter the weighted matrix

0 99 1 99

99 0 2 99

1 2 0 99

99 4 99 0



(1,3)=1

(3,2)=2

(2,4)=4

The cost of minimum spanning tree=7

12. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm

Dijkstra’s algorithm is a popular algorithm used to find the shortest path from a given vertex to all other vertices in a weighted connected graph. The algorithm works by maintaining a set of vertices whose shortest distance from the source vertex is already known. Initially, the distance of the source vertex is set to 0, and the distance of all other vertices is set to infinity. At each step, the algorithm selects the vertex with the minimum distance from the source vertex that is not yet included in the set of vertices whose shortest distance is already known. It then updates the distances of all adjacent vertices of the selected vertex. The algorithm repeats this process until all vertices are included in the set of vertices whose shortest distance is already known

```
import java.util.Scanner;
public class Djikstra {
        int d[] = new int[10];
        int p[] = new int[10];
        int visited[] = new int[10];
        public void dijk(int[][] a, int s, int n) {
        int u = -1, v, min, i;
        for (i = 0; i < n; i++) {
        d[i] = Integer.MAX_VALUE;
        p[i] = -1;
        visited[i] = 0;
        }
        d[s] = 0;
        for (i = 0; i < n; i++) {
        min = Integer.MAX_VALUE;
        for (v = 0; v < n; v++) {
                if (visited[v] == 0 && d[v] < min) {
                min = d[v];
                u = v;
                }
        }
        if (u == -1) return;
        visited[u] = 1;
        for (v = 0; v < n; v++) {
                if (a[u][v] != 0 && visited[v] == 0 && d[u] + a[u][v] < d[v]) {
                d[v] = d[u] + a[u][v];
                p[v] = u;
                }
        }
        }
```

```
        }

        void path(int v, int s) {
        if (p[v] != -1) path(p[v], s);
        if (v != s) System.out.print("->" + v + " ");
        }

        void display(int s, int n) {
        for (int i = 0; i < n; i++) {
        if (i != s) {
                System.out.print(s + " ");
                path(i, s);
                System.out.println("=" + d[i] + " ");
        }
        }
        }
        public static void main(String[] args) {
        int a[][] = new int[10][10];
        int i, j, n, s;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of vertices:");
        n = sc.nextInt();
        System.out.println("Enter the weighted adjacency matrix:");
        for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
                a[i][j] = sc.nextInt();
        }
        }
        System.out.println("Enter the source vertex:");
        s = sc.nextInt();
        Djikstra tr = new Djikstra();
        tr.dijk(a, s, n);
        System.out.println("The shortest path between source " + s + " to remaining
vertices is:");
        tr.display(s, n);
        sc.close();
        }
}
```

Output

Enter the number of vertices 4

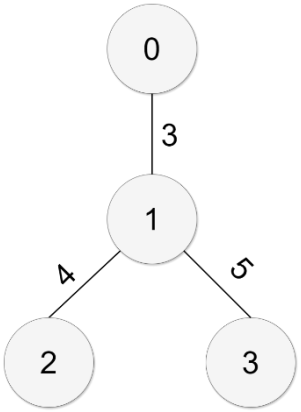
Enter the weighted matrix

0 3 99 99

3 0 4 5

99 4 0 99

99 5 99 0



Enter the source vertex 1

The shortest path between source 1 to remaining vertices are

1->0=3

1->2=4

1->3=5

Viva Questions

1. What is Dijkstra's algorithm?

Dijkstra's algorithm is a greedy algorithm that finds the Minimum Spanning Tree for a connected, weighted graph by sorting all the edges and adding them one by one, provided they do not form a cycle, until all vertices are included.

2. What is the time complexity of Dijkstra's algorithm?

The time complexity is O(V^2) or O((V+E) log V), where E is the number of edges and V is the number of vertices, primarily due to sorting the edges.

3. How does Dijkstra's algorithm ensure no cycles are formed in the MST?

By using the Disjoint Set data structure, Dijkstra's algorithm checks whether two vertices are in the same set before adding an edge. If they are, adding the edge would form a cycle, so it is skipped.

4. What are some real-world applications of Dijkstra's algorithm?

1. **Network Design:** Minimizing the cost of connecting different network nodes, such as in telecommunications.
2. **Infrastructure Projects:** Designing road networks or water supply systems to minimize construction costs.
3. **Cluster Analysis:** Grouping data points in machine learning to minimize the distance within clusters.
4. **Computer Graphics:** Creating minimum spanning trees for efficient rendering of connected structures.

13. Write a program to solve Travelling Sales Person problem using dynamic programming approach

The Travelling Salesman Problem (TSP) is a classic optimization problem in computer science. It is a problem of combinatorial optimization, where we have a set of cities and the distance between every pair of cities, and we want to find the shortest possible route that visits every city exactly once and returns to the starting point. The dynamic programming approach is one of the most efficient ways to solve this problem.

```
import java.io.*;
import java.util.*;
public class TSE {
        static int n = 4;
        static int MAX = 1000000;
   static int[][] dist = {
        {0,60, 20, 30},
        {60, 0, 40, 10},
        {20, 40, 0, 50},
        {30, 10, 50, 0}
        };
        static int[][] memo = new int[n + 1][1 << (n + 1)];
        static int fun(int i, int mask) {
        if (mask == ((1 << i) | 3)) return dist[1][i];
        if (memo[i][mask] != 0) return memo[i][mask];
        int res = MAX;
        for (int j = 1; j <= n; j++) {
        if ((mask & (1 << j)) != 0 && j != i && j != 1) {
                res = Math.min(res, fun(j, mask & (~(1 << i))) + dist[j][i]);
        }
        }
        return memo[i][mask] = res;
        }
        public static void main(String[] args) {
        int ans = MAX;
        for (int i = 1; i <= n; i++) {
        ans = Math.min(ans, fun(i, (1 << (n + 1)) - 1) + dist[i][1]);
        }
        System.out.println("The cost of most efficient tour = " + ans);
        }
}
```

14. Implement N Queen's problem using Back Tracking.

The N Queen's problem is a classic problem of placing N chess queens on an NxN chessboard so that no two queens attack each other. This problem can be solved using backtracking.

```
public class NQueenProblem {
        final int N = 4;
        void printSolution(int board[][]) {
        for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
                if (board[i][j] == 1) System.out.print("Q ");
                else System.out.print(". ");
        }
        System.out.println();
        }
        }
        boolean isSafe(int board[][], int row, int col) {
        int i, j;
        for (i = 0; i < col; i++) if (board[row][i] == 1) return false;
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--) if (board[i][j] == 1) return false;
        for (i = row, j = col; j >= 0 && i < N; i++, j--) if (board[i][j] == 1) return false;
        return true;
        }
        boolean solveNQUtil(int board[][], int col) {
        if (col >= N) return true;
        for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
                board[i][col] = 1;
                if (solveNQUtil(board, col + 1) == true) return true;
                board[i][col] = 0; // BACKTRACK
        }
        }
        return false;
        }
        boolean solveNQ() {
        int board[][] = { { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 } };
        if (solveNQUtil(board, 0) == false) {
        System.out.print("Solution does not exist");
        return false;
        }
        printSolution(board);
        return true;
```

```
        }
        public static void main(String args[]) {
        NQueenProblem Queen = new NQueenProblem();
        Queen.solveNQ();
        }
}
```

Output

. . Q .

Q . . .

. . . Q

. Q . .

Viva Questions

1. What is the N-Queens problem?

The N-Queens problem involves placing N queens on an N×N chessboard so that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal.

2. How does the backtracking approach solve the N-Queens problem?

Backtracking explores all possible configurations of placing queens. It places a queen in a valid position and recursively attempts to place the next queen. If a conflict arises, it backtracks by removing the last placed queen and trying the next possible position.

3. What are the key constraints to check when placing a queen?

Check that no other queen is in the same row, same column, same diagonal.

4. What are some real-world applications of the N-Queens problem?

1. **Constraint Satisfaction Problems:** Solving scheduling or resource allocation problems where certain constraints must be met.
2. **Game Development:** Designing AI for games involving board configurations, such as chess.
3. **Parallel Processing:** Optimizing the placement of tasks on processors in computing environments.
4. **Combinatorial Optimization:** Exploring configuration spaces in engineering and design problems.

15. Find a subset of a given set S={S1,S2,…,Sn} of n positive integers whose SUM is equal to a given positive integer d.

    This is a classic problem in computer science known as the **Subset Sum Problem**. Given a set of non-negative integers and a value sum, the task is to check if there is a subset of the given set whose sum is equal to the given sum.

    One way to solve this problem is to use **dynamic programming**. The time complexity of this approach is O(n*sum) .

```
import java.util.ArrayList;
import java.util.Scanner;

public class SubsetSumDP {
        public static boolean subsetSum(int[] arr, int sum, ArrayList subset) {
        int n = arr.length;
        boolean[][] dp = new boolean[n + 1][sum + 1];
        for (int i = 0; i <= n; i++) {
        dp[i][0] = true;
        }
        for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
                if (j >= arr[i - 1]) {
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - arr[i - 1]];
                } else {
                dp[i][j] = dp[i - 1][j];
                }
        }
        }
        if (!dp[n][sum]) {
        return false;
        }
        int i = n, j = sum;
        while (i > 0 && j > 0) {
        if (dp[i][j] != dp[i - 1][j]) {
                subset.add(arr[i - 1]);
                j -= arr[i - 1];
        }
        i--;
        }
        return true;
        }
        public static void main(String[] args) {
```

```
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of elements: ");
        int n = scanner.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter the elements:");
        for (int i = 0; i < n; i++) {
        arr[i] = scanner.nextInt();
        }
        System.out.print("Enter the target sum: ");
        int sum = scanner.nextInt();
        ArrayList subset = new ArrayList<>();
        long startTime = System.nanoTime();
        boolean hasSubsetSum = subsetSum(arr, sum, subset);
        long endTime = System.nanoTime();
        System.out.println("Subset sum exists: " + hasSubsetSum);
        if (hasSubsetSum) {
        System.out.println("Subset contributing to the sum: " + subset);
        }
        double timeElapsed = (endTime - startTime) / 1e6;
        System.out.println("Time complexity: " + timeElapsed + " milliseconds");
        scanner.close();
        }
}
```

Output

Enter the number of elements: 4

Enter the elements:5 6 7 8

Enter the target sum:15 Subset sum exists: true

Subset contributing to the sum: [8,7]

Time complexity: 0.1025 milliseconds