UNIVERSITY VISVESVARAYA COLLEGE OF ENGINEERING
(UVCE)

K R Circle, Dr Ambedkar Veedhi, Bengaluru, Karnataka 560001

Laboratory Manual

For

Design and Analysis of Algorithm

(SUBCODE:21CCPC405L)

(4th semester B.tech)

Prepred by

Anjana

V

USN: U03NM21T029004

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Acknowledgement

I, Anjana V like to express my sincere appreciation to Neetha who contributed to the development of this lab manual. The dedication and support were invaluable in creating a resource that enhances the learning experience.

Thank you Neetha, for the expertise and guidance in merge sort,quick sort, binary search tree, BFS, topological order and so on DAA lab. The insights and feedback greatly enriched the content of this manual.

Thank you Neetha for the assistance in designing the experiments and providing valuable input during the manual's development.

I would like to acknowledge for providing valuable insights that contributed to its refinement.

BANGALORE UNIVERSITY

UNIVERSITY VISVESVARAYA COLLEGE OF ENGINEERING

K. R. CIRCLE, BENGALURU – 560001

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that, Anjana V(U03NM21T029004), has satisfactorily completed the course of Experiments in Practical Design And Analysis Of Algorithm, prescribed by Department of CSE, Bangalore University, during the academic year 2022- 23.

Signature of the Teacher Signature of the Examiner

_________________________ _________________________

Date: ____________

Name of the Candidate: Anjana V

Reg No.: U03NM21T029004

Examination Centre: UVCE

INDEX

INTRODUCTION

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks.It is an efficient method that can be expressed within finite amount of time and space. Design and Analysis of Algorithm is very important for designing algorithm to solve different types of problems in the branch of computer science and information technology.

Algorithm Design

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space. To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Characteristics of Algorithms

The main characteristics of algorithms are as follows –

- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point.

Design and Analysis of Algorithm

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term "**analysis of algorithms**" was coined by Donald Knuth. Donald Knuth once said “A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyse them.” Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

The Need for Analysis

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm. Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm. Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis

- **Worst-case** – The maximum number of steps taken on any instance of size **a**.
- **Best-case** – The minimum number of steps taken on any instance of size **a**.
- **Average case** – An average number of steps taken on any instance of size **a**.

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**.

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.Arrays;
import java.util.Scanner;
public class MergeSort {
 public static void mergeSort(int[] arr, int left, int right) {
 if (left < right) {
int mid = left + (right - left) / 2;
```

```
 mergeSort(arr, left, mid);
mergeSort(arr, mid + 1, right);
merge(arr, left, mid, right);
 }
 }
public static void merge(int[] arr, int left, int mid, int right)
{
int n1 = mid - left + 1; int n2 = right - mid;
int[] leftArray = new int[n1];
 int[] rightArray = new int[n2];
 for (int i = 0; i < n1; i++) {
leftArray[i] = arr[left + i];
 }
 for (int j = 0; j < n2; j++) {
 rightArray[j] = arr[mid + 1 + j];
}
 int i = 0, j = 0, k = left;
 while (i < n1 && j < n2) {
if (leftArray[i] <= rightArray[j]) {
 arr[k++] = leftArray[i++];
} else {
arr[k++] = rightArray[j++];
}
 }
 while (i < n1) {
 arr[k++] = leftArray[i++];
 }
while (j < n2) {
arr[k++] = rightArray[j++];
}
}
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.println("Original array: " + Arrays.toString(arr));
 long startTime = System.nanoTime();
 mergeSort(arr, 0, arr.length - 1);
 long endTime = System.nanoTime();
System.out.println("Sorted array: " + Arrays.toString(arr));
 double timeElapsed = (endTime - startTime) / 1e6;
System.out.println("Time  complexity:  "  +  timeElapsed  +  "
milliseconds");
scanner.close();
}
 }
```

Output

```
  Enter the number of elements: 5
  Enter the elements:

  100
  55
  33
  888
  11
  Original array: [100, 55, 33, 888, 11]
  Sorted array: [11, 33, 55, 100, 888]
  Time complexity: 0.0236 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 theem according to their value.

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 + 1
  swap A[i] with A[j]
   }}
  swap A[i+1] with A[end]
  return i+1
}
```

Program:
```
import java.util.Arrays;
import java.util.Scanner;
import java.util.Random;
public class QuickSort {
     public static void quickSort(int[] arr, int low, int high)
{
           if (low < high) {
           int partitionIndex = partition(arr, low, high);
           quickSort(arr, low, partitionIndex - 1);
           quickSort(arr, partitionIndex + 1, high);
           }
           }
           public static int partition(int[] arr, int low, int
high) {
           int pivot = arr[high];
           int i = low - 1;
           for (int j = low; j < high; j++) {
           if (arr[j] < pivot) {
           i++;
           int temp = arr[i];
           arr[i] = arr[j];
           arr[j] = temp;
           }
           }
           int temp = arr[i + 1];
           arr[i + 1] = arr[high];
           arr[high] = temp;
           return i + 1;
     }
     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.println("Original array: " +
Arrays.toString(arr));
           long startTime = System.nanoTime();
           quickSort(arr, 0, arr.length - 1);
           long endTime = System.nanoTime();
```

```
            System.out.println("Sorted array: " +
Arrays.toString(arr));
            double timeElapsed = (endTime - startTime) / 1e6; //
Convert nanoseconds to
            System.out.println("Time complexity: " + timeElapsed
+ " milliseconds");
            scanner.close();
      }

}
```

Output

```
Enter the number of elements: 5
Enter the elements:

55
33
77
22
88
Original array: [55, 33, 77, 22, 88]
Sorted array: [22, 33, 55, 77, 88]
Time complexity: 0.0148 milliseconds
```

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;
} // Insert a key into the BST
    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 {
 // Node with only one child or no child
   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(50);
     tree.insert(30);
     tree.insert(20);
     tree.insert(40);
     tree.insert(70);
     tree.insert(60);
     tree.insert(80);
   System.out.println("Inorder traversal:");
     tree.inorder();
System.out.println(); // Delete elements
System.out.println("Delete 20:");
     tree.delete(20);
     tree.inorder();
 System.out.println();
System.out.println("Delete 30:");
      tree.delete(30);
     tree.inorder();
   System.out.println();
    }
}
```

Output

```
Inorder traversal:
20 30 40 50 60 70 80
Delete 20:
30 40 50 60 70 80
Delete 30:
40 50 60 70 80
```

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);
    }

}
```

Ouput

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

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 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:
5 2 0 6
4 0 1 2
2 3 1 5
5 4 3 2
Topological Sort:
3 1 2 0
```

6. b) 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 ≤ q ≤ 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
```

7. a) Check whether a given graph is connected or not using DFS method.

we can start a DFS (Depth First Search) from any of the vertices and mark the visited vertices as True in the visited [] array. After completion of DFS, we can check if all the vertices in the visited [] array are marked as True. If yes, then the graph is connected; otherwise, the graph is not connected or disconnected

```
import java.util.Scanner;
public class GraphDFS {
private int vertices;
private int[][] adjacencyMatrix;
public GraphDFS(int v) {
vertices = v; adjacencyMatrix = new int[v][v];
}
public void addEdge(int start, int end) {
adjacencyMatrix[start][end] = 1;
adjacencyMatrix[end][start] = 1;
}
public void dfs(int startVertex, boolean[] visited) {
System.out.print(startVertex + " "); visited[startVertex] =
true;
for (int i = 0; i < vertices; i++) {
if(adjacencyMatrix[startVertex][i] == 1 && !visited[i]) {
      dfs(i, visited);
}
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of vertices: ");
int v = scanner.nextInt();
GraphDFS graph = new GraphDFS(v);
System.out.println("Enter the adjacency matrix:");
for (int i = 0; i < v; i++) {
for (int j = 0; j < v; j++) {
graph.adjacencyMatrix[i][j] = scanner.nextInt();
 }
 }
System.out.print("Enter the starting vertex for DFS: ");
int startVertex = scanner.nextInt();
boolean[] visited = new boolean[v];
System.out.print("DFS  traversal  starting  from  vertex  "  +
startVertex + ": ");
graph.dfs(startVertex, visited);
scanner.close();
 }
}
```

Output

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

0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0
Enter the starting vertex for DFS: 2
DFS traversal starting from vertex 2: 2 0 1 3
```

```
7.  b) Implement Floyd's algorithm for the All-Pairs-
Shortest-Paths problem
```

Floyd's algorithm is used to find the shortest path between all pairs of vertices in a weighted graph. It is an extension of Dijkstra's algorithm, which finds the shortest path between a single source and all other vertices in a graph

```
class FloydWarshall {
final static int INF = Integer.MAX_VALUE;

     public void FloydWarshall() {
     }
int[][] floydWarshall(int A[][]) {
int n = A.length;
int D[][] = createDistanceMatrix(A, n);
for (int k = 0; k < n; k++) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
if (D[i][k] < INF && D[k][j] < INF)
D[i][j] = Math.min(D[i][j], D[i][k] + D[k][j]);
}
}
}
 return D;
}
 int[][] createDistanceMatrix(int A[][], int n) {
 int D[][] = new int[n][n];
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
 if (i == j)
  D[i][j] = 0;
 else if (A[i][j] == 0)
   D[i][j] = INF;
  else
  D[i][j] = A[i][j];
}
}
return D;
}
```

```
   public static void main(String[] args) {
       FloydWarshall fw = new FloydWarshall();
       int[][] adjMatrix = {
           {0, 3, INF, 5},
           {2, 0, INF, 4},
           {INF, 1, 0, INF},
           {INF, INF, 2, 0}
       };
       int[][] shortestPaths = fw.floydWarshall(adjMatrix);
       for (int i = 0; i < shortestPaths.length; i++) {
           for (int j = 0; j < shortestPaths.length; j++) {
               System.out.printf("%d ", shortestPaths[i][j]);

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

Output

```
0 3 7 5

2 0 6 4

3 1 0 5

5 3 2 0
```

8. 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

```
class HeapSort
{
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of he
ap. */
static void heapify(int a[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
static void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        heapify(a, i, 0);
    }
}
/* function to print the array elements */
static void printArr(int a[], int n)
{
    for (int i = 0; i < n; ++i)
        System.out.print(a[i] + " ");
}
public static void main(String args[])
{
    int a[] = {45, 7, 20, 40, 25, 23, -2};
    int n = a.length;
    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);
}
}
```

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
```

9. `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
```

10. 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
```

11. 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, flag=0;
int sol[]=new int[10];
 System.out.println("Enter the number of vertices");
Scanner sc=new Scanner(System.in);
 n=sc.nextInt();
 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;
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
if(sol[i]==1&&sol[j]==0)
 if(i!=j&&min>w[i][j]) {
 min=w[i][j]; u=i; v=j;
 }
 sol[v]=1;
 sum=sum+min;
k++;
System.out.println(u+"->"+v+"="+min);
}
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
10 20 30
20 30 40
```

```
30 40 50
Enter the source vertex
1
1->2=20
1->3=30
The cost of minimum spanning tree is 50
```

12. 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) {
int p=m;
 while(parent[p]!=0)
 p=parent[p];
return p;
 }
 void union(int i,int j) {
 if(i<=n;i++)
for(j=1;j<=n;j++)
if(a[i][j]<min&&i!=j) {
 min=a[i][j]; u=i; v=j;
 }
 i=find(u);
 j=find(v);
 if(i!=j) {
 union(i,j);
 System.out.println("("+u+","+v+")"+"="+a[u][v]); sum=sum+a[u][v]; k++;
}
 a[u][v]=a[v][u]=99;
 }
 System.out.println("The cost of minimum spanning tree = "+sum);
}
 public static void main(String[] args) {
 int a[][]=new int[10][10], i , j;
 System.out.println("Enter the number of vertices of the graph");
Scanner sc=new Scanner(System.in);
int n=sc.nextInt();
 System.out.println("Enter the wieghted matrix");
 for(i=1;i<=n;i++)
 for(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 12 10
  99 22 43 12
  23 24 25 28
  73 76 46 29
  (1,4)=10
  (1,3)=12
  (2,4)=12
The cost of minimum spanning tree=34
```

13. 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 Dijkstra {

 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,i,j,min;

for(v=0;v<min&& visited[j]==0) {

 min=d[j]; u=j;

 }

}

 visited[u]=1;

for(v=0;v<d[v])&&(u!=v)&&visited[v]==0) {

 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){
 int i;
for(i=0;i<n;i++}{
if(i!=s){
System.out.print(s+" ");
path(i,s);
}
if(i!=s)
System.out.print("="+d[i]+" "); System.out.println();
}
}
public static void main(String[] args) { int a[][]=new int[10][10];
int i,j,n,s;
System.out.println("enter the number of vertices");
Scanner sc = new Scanner(System.in);
n=sc.nextInt();
System.out.println("enter the weighted 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();
Dijkstra tr=new Dijkstra();
```

```
tr.dijk(a,s,n);
System.out.println("the shortest path between source"+s+"to remaining vertices are");
tr.display(s,n);
sc.close();
}
}
```

Output

```
Enter the number of vertices
4
Enter the weighted matrix
12 34 56 78
89 90 11 13
14 15 16 17
21 22 23 24
Enter the source vertex
3
The shortest path between source3 to remaining vertices are-
3->0=21
3->1=22
3->2=23
```

14. 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, 0, 0, 0, 0 }, { 0, 0, 10, 15, 20
},{0, 10, 0, 25, 25 }, { 0, 15, 25, 0, 30 },{ 0, 20, 25, 30,
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; // result of this sub-problem

          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);
     }
}
```

Output

```
The cost of most efficient tour=80
```

15. 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 . .
```

16. 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]);
/*Java program that reads inputs from the keyboard to solve the Subset Sum
problem using dynamic programming and displays the time complexity, including
finding and displaying the actual subsets that contribute to the sum*/
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
```