DESIGN AND ANALYSIS OF ALGORITHMS 18CSI401L

PRACTICAL RECORD

SUBMITTED BY

YAMASANI MAHENDRA REDDY

20BTRCO011

Department of Computer Science and Engineering

School of Engineering and Technology

JAIN University

Jakkasnadra post, Kanakapura – 562112

Academic Year: February – June (2021-2022)

Jain Global Campus, Jakkasandra Post, Kanakpura Taluk, Ramanagar District - 562112

Laboratory Certificate

This is to certify that Mr. /Ms. ....

.... has satisfactorily completed the course of experiments

in practical .... prescribed

by the Jain (Deemed-To-Be-University) .... Semester

Course in the Laboratory of this college in year 20 **21** - 20 **22** ....

Date :

Name of the Candidate :

....

Reg. No

....

Date of Practical Examination ....

REMARKS

VALUED
Examiner 1..
Examiner 2..

Signature of the
Teacher Incharge of the
Batch

Head of Department

INDEX:

Experiment -1

A. Create a Java class called Student with the following details as variables within it. (i) USN (ii) Name (iii) Branch (iv) Phone no. Write a Java program to create n student objects and print the USN, Name, Branch, and Phone no. of these objects with suitable headings.

AIM: The aim of this program is to create a class and make objects of that class to print the details of a student.

DESCRIPTION:

- Create a class named “student”.
- Declare variables within it containing the details like name, USN, Branch, Phone no.
- Create a constructor to initialize these variables.
- Create a function that prints these details like usn, name, branch and phone no.

ALGORITHM:

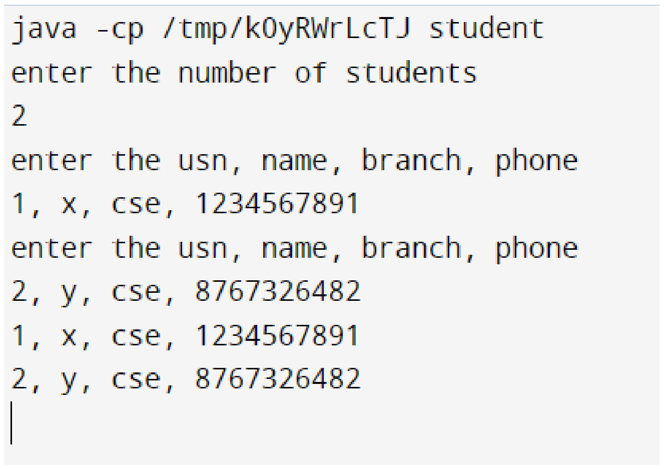
- //Input: Values for USN, name, branch and phone no.
- //Output: Displaying the details of n student objects.
- //Steps:
- class “student” is created.
- Declare variables USN, Name, Branch, and Phone no.
- a constructor of Student class is created to initialize these variables.
- a function “display_details” is created that prints these details like usn, name, branch and phone no.
- Multiple objects of “student” class calls the function “display_details” to print the details contained in student class.

PROGRAM:

```
import java.util.Scanner;

public class student {

String USN;

String Name;

String branch;

String phone;

void insertRecord(String reg, String name, String brnch, String ph) {

USN = reg;

Name = name;

branch = brnch;

phone = ph;

}

void displayRecord() {
```

```
System.out.println(USN + " " + Name + " " + branch + " " + phone);
}
public static void main(String args[]) {
student s[] = new student[100];
Scanner sc = new Scanner(System.in);
System.out.println("enter the number of students");
int n = sc.nextInt();
for (int i = 0; i < n; i++)
s[i] = new student();
for (int j = 0; j < n; j++) {
System.out.println("enter the usn, name, branch, phone");
String USN = sc.next();
String Name = sc.next();
String branch = sc.next();
String phone = sc.next();
s[j].insertRecord(USN, Name, branch, phone);
}
for (int m = 0; m < n; m++) {
s[m].displayRecord();
} } }
```

OUTPUT:



```
java -cp /tmp/kOyRWrLcTJ student
enter the number of students
2
enter the usn, name, branch, phone
1, x, cse, 1234567891
enter the usn, name, branch, phone
2, y, cse, 8767326482
1, x, cse, 1234567891
2, y, cse, 8767326482
```

B Write a Java program to implement the Stack using arrays. Write push(), pop(), and display() methods to demonstrate its working.

AIM:

The aim of this program is to create stack using arrays and perform all the stack related functions like pushing elements, popping elements and displaying the contents of stack. Stack is abstract data type which demonstrates Last in first out (LIFO) behavior. We will implement same behavior using Array.

DESCRIPTION:

- A class is created that contains the defined array as well as all the variables defined.
- Constructor initialize those variables and array.
- Function are created for pushing he elements into stack.
- Function are created for popping the elements from stack.
- Function are created for displaying the contents of stack.

ALGORITHM:

- // Input : Elements to be pushed or popped from the Stack.
- // Output : pushed elements, popped elements, contents of stack.
- Steps:
- A class created and variables are defined like size, array[], top.
- A customized constructor of same class is used for initializing size, top variables and the array[].
- A function created for pushing the elements into stack :

```
push(int pushedElemnet)
{
if(stack is not full)
{
Top++;
Array[top]=pushedElement;
}
else
{
Stack is full
} }
```

- A function created for popping the elements from stack :

```
Pop()
{
if(stack is not empty)
{
A=top; Top--;
}
else
{
Stack is empty
} }
```

- A function is created for displaying the elements in the stack:

```
printElemnts()
{
if(top>=0)
{
for(i=0;i<=top;i++)
{
Print all elements of array
} }
```

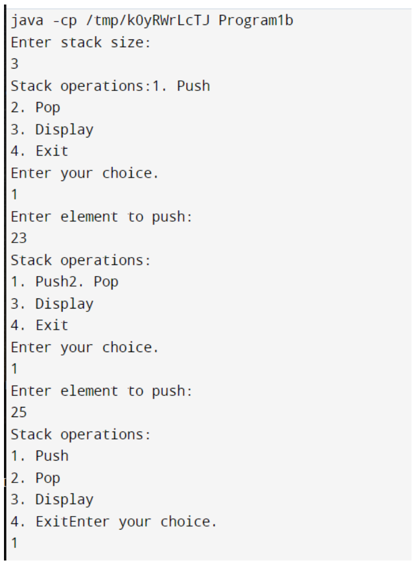
- A boolen function is created to cheak whether stack is empty or full :

```
Boolen isFull ()
{
return (size-1==top)
}
Boolen isEmpty()
{
return (top==-1)
}
```

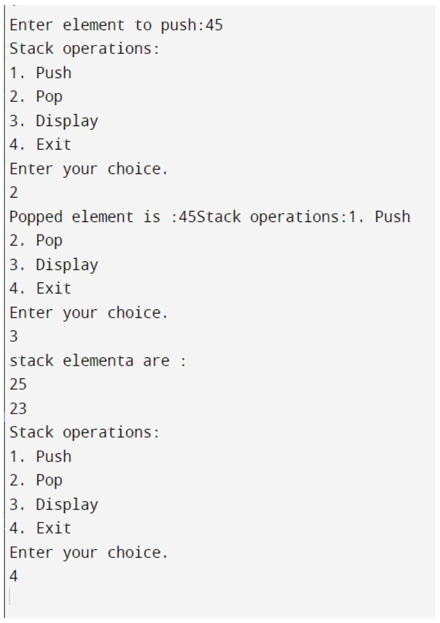
PROGRAM:

```
import java.util.Scanner;
public class Program1b {
static int[] integerStack;
static int top = -1;
public static void main(String[] args) {
 System.out.println("Enter stack size:");
 Scanner scanner = new Scanner(System.in);
 int size = scanner.nextInt();
 integerStack = new int[size];
 System.out.println("Stack operations:");
 System.out.println("1. Push");
 System.out.println("2. Pop");
 System.out.println("3. Display");
 System.out.println("4. Exit");
 System.out.println("Enter your choice.");
 int choice = scanner.nextInt();
 while (choice != 4) {
 if (choice == 1) {
 System.out.println("Enter element to push:");
 int element = scanner.nextInt();
 if (top == size - 1)
 System.out.println("stack is full");
 else {
 top = top + 1;
 integerStack[top] = element;
 }  }
else if (choice == 2) {
 if (top == -1) {
 System.out.println("stack is empty.");
 } else {
 System.out.println("Popped element is :" + integerStack[top]);
 top = top - 1;
 }  }
else if (choice == 3) {
 if (top == -1)
 System.out.println("stack is empty");
 else {
 System.out.println("stack elementa are :");
 for (int i = top; i >= 0; i--)
 System.out.println(integerStack[i]);
 }  }
else
 System.out.println("Enter correct choice.");
 System.out.println("Stack operations:");
 System.out.println("1. Push");
 System.out.println("2. Pop");
 System.out.println("3. Display");
 System.out.println("4. Exit");
 System.out.println("Enter your choice.");
 choice = scanner.nextInt();
 } } }
```

OUTPUT:



```
java -cp /tmp/kOyRWrLcTJ Program1b
Enter stack size:
3
Stack operations:1. Push
2. Pop
3. Display
4. Exit
Enter your choice.
1
Enter element to push:
23
Stack operations:
1. Push2. Pop
3. Display
4. Exit
Enter your choice.
1
Enter element to push:
25
Stack operations:
1. Push
2. Pop
3. Display
4. ExitEnter your choice.
1
```



```
Enter element to push:45
Stack operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice.
2
Popped element is :45Stack operations:1. Push
2. Pop
3. Display
4. Exit
Enter your choice.
3
stack elementa are :
25
23
Stack operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice.
4
```

Experiment -2

A. Write a Java program to read two integers a and b. Compute a/b and print, when b is not zero. Raise an exception when b is equal to zero.

AIM:

Understanding the concepts of exception handling in java.

DESCRIPTION:

- A class is created containing the main method.
- Inside the main method the calculation is done of division using two operands.
- Try and Catch block is implemented for handling the arithmetic exception raised when division is done by zero.
- Else the final output is printed.

ALGORITHM:

- // Input: values of two operand i.e a and b.
- // Output: a) answer displayed when b != 0.

Arithmetic exception raised and error message displayed when b = 0.

Steps :

- A class is created containing the main method.
- Two variables are declared i.e. a and b.
- Input is obtained from console

```
Scanner sc = new Scanner(System.in);
a = sc.nextInt();
b = sc.nextInt();
```

1) The code to calculate division is kept under try block try

```
{
System.out.println(a/b);
}
```

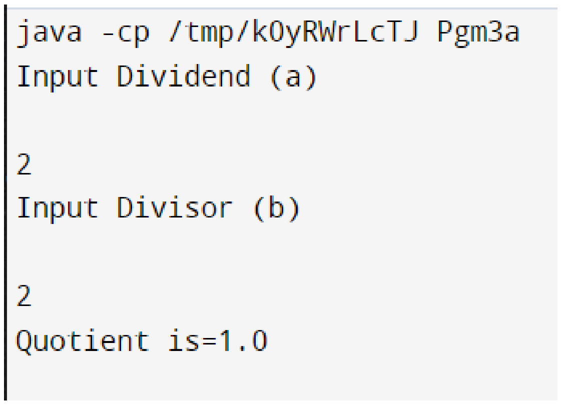
2) The arethemetic exception raised when b=0 is handled in catch block that follows try block

```
catch(ArithmeticException e)
{
e.printStackTrace();
}
```

PROGRAM:

```
import java.util.Scanner;
public class Pgm3a {
public static void main(String[] args) {
 int a, b;
 float res;
 try {
 Scanner inn = new Scanner(System.in);
 System.out.println("Input Dividend (a) \n");
 a = inn.nextInt();
 System.out.println("Input Divisor (b) \n");
 b = inn.nextInt();
 res = a / b;
 System.out.println("Quotient is=" + res);
 } catch (ArithmeticException e) {
 System.out.println("Divide by zero error");
 } } }
```

OUTPUT:



```
java -cp /tmp/kOyRWrLcTJ Pgm3a
Input Dividend (a)

2
Input Divisor (b)

2
Quotient is=1.0
```

B. Write a Java program that implements a multi-thread application that has three threads. First thread generates a random integer for every 1 second; second thread computes the square of the number and prints; third thread will print the value of cube of the number.

AIM:

To understand the concepts of multithreading by creating three threads that perform different tasks when one thread is suspended for some time duration.

DESCRIPTION:

- Create three class, one for each thread to work.
- First class is to generate a random number for every 1 second, second class computes the square of the number and the last class generates cube of the number. All the classes prints the number after generation.

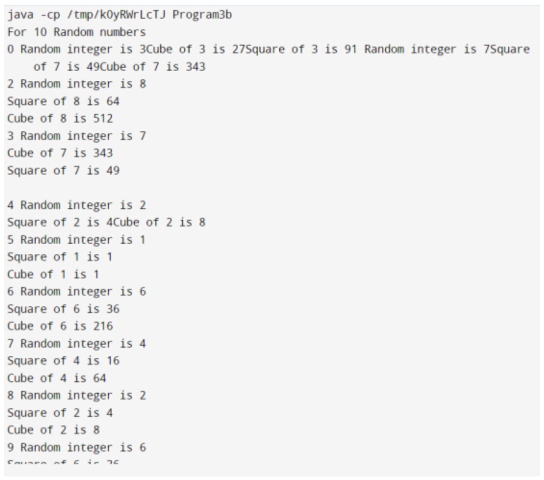
ALGORITHM:

- // Input: Random number.
- //Output: square and cube of the number.
- Steps: Three threads are created.
- Three classes RandomNumber, SquareGenerator and CubeGenerator are created.
- Class RandomNumber generates an integer using random number generator and prints the integer with thread t1.
- Next class SquareGenerator is called to generate square of the number and print it with thread t2.
- At last class CubeGenerator is called to generate cube of the number and print it with thread t3.

PROGRAM:

```
import java.util.Random;
//using Thread Class
public class Program3b {
static int randomInteger;
public static void main(String[] args) {
 System.out.println("For 10 Random numbers");
 MyThread1 thread1 = new MyThread1();
 thread1.start();// start thread1
 } }
//Thread1
class MyThread1 extends Thread {
public void run() {
 int i = 0;
 try {
 while (i < 10) {
 Random random = new Random();
 Program3b.randomInteger = random.nextInt(10);
 System.out.println(i + " Random integer is " + Program3b.randomInteger);
 new MyThread2().start();// start thread2
 new MyThread3().start();// start thread3
 Thread.sleep(1000 * 1);// delay for synchronization
 System.out.println("\n\n");
 i++;
 } }
catch (InterruptedException exception) {
 exception.printStackTrace();
 } } }
class MyThread2 extends Thread {
public void run() {
 System.out.println("Square of " + Program3b.randomInteger + " is " +
Program3b.randomInteger * Program3b.randomInteger);
 } }
class MyThread3 extends Thread {
public void run() {
 System.out.println("Cube of " + Program3b.randomInteger + " is "
 + Program3b.randomInteger * Program3b.randomInteger *
Program3b.randomInteger);
 }}
```

OUTPUT:



```
java -cp /tmp/kOyRWrLcTJ Program3b
For 10 Random numbers
0 Random integer is 3Cube of 3 is 27Square of 3 is 91 Random integer is 7Square
    of 7 is 49Cube of 7 is 343
2 Random integer is 8
Square of 8 is 64
Cube of 8 is 512
3 Random integer is 7
Cube of 7 is 343
Square of 7 is 49

4 Random integer is 2
Square of 2 is 4Cube of 2 is 8
5 Random integer is 1
Square of 1 is 1
Cube of 1 is 1
6 Random integer is 6
Square of 6 is 36
Cube of 6 is 216
7 Random integer is 4
Square of 4 is 16
Cube of 4 is 64
8 Random integer is 2
Square of 2 is 4
Cube of 2 is 8
9 Random integer is 6
Square of 6 is 36
```

Experiment -3

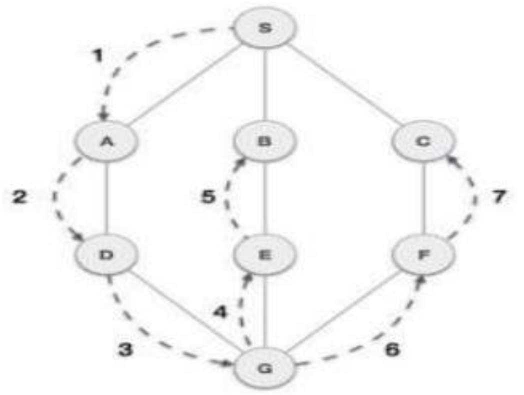
Print all the nodes reachable from a given starting node in a digraph using BFS and DFS method

AIM:

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

DESCRIPTION:

Depth First Search (DFS) 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.



As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

• 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.

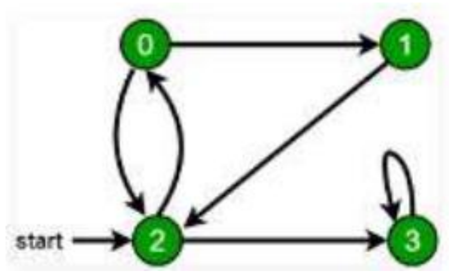
Breadth First Search or BFS for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.

Following are the implementations of simple Breadth First Traversal from a given source.

The implementation uses adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.



ALGORITHM:

A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

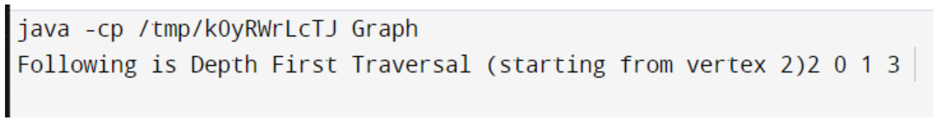
The DFS algorithm works as follows:

- Start by putting any one of the graph's vertices on top of a stack.
- Take the top item of the stack and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.
- Keep repeating steps 2 and 3 until the stack is empty.

PROGRAM:

```
// Java program to print DFS traversal from a given graph
import java.io.*;
import java.util.*;
class Graph
{
private int V; // No. of vertices
private LinkedList<Integer> adj[];
Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }
void addEdge(int v, int w)
 {
 adj[v].add(w); // Add w to v's list.
 }
void DFSUtil(int v,boolean visited[])
 {
 visited[v] = true;
 System.out.print(v+" ");
Iterator<Integer> i = adj[v].listIterator();
 while (i.hasNext())
 {
 int n = i.next();
 if (!visited[n])
 DFSUtil(n, visited);
 } }
void DFS(int v)
 {
 boolean visited[] = new boolean[V];
 DFSUtil(v, visited);
 }
public static void main(String args[])
 {
 Graph g = new Graph(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);
 System.out.println("Following is Depth First Traversal "+
 "(starting from vertex 2)");
 g.DFS(2);
 } }
```

OUTPUT:



```
java -cp /tmp/kOyRWrLcTJ Graph
Following is Depth First Traversal (starting from vertex 2)2 0 1 3
```

Experiment -4

Sort a given set of elements using the quick 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 1st to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

AIM:

The aim of this program is to sort "n" randomly generated elements using Quick sort and plotting the graph of the time taken to sort n elements versus n.

DESCRIPTION:

- Call a function to generate list of random numbers (integers)
- Record clock time
- Call Quick sort function to sort n randomly generated elements.
- Record clock time.
- Measure difference in clock time to get elapse time to sort n elements using Quick sort
- Print the Sorted „ n‟ elements and time taken to sort.
- Repeat the above steps for different values of n as well as to demonstrate worst, best and average case complexity.

ALGORITHM:

- Declare time variables.
- call function to record the start time before sorting.
- Generate "n" elements randomly using random number generator.
- Call Quick sort function to sort n elements.
- call function to record the end time after sorting.
- Calculate the time required to sort n elements using Quick sort i.e elapse time
- elapse_time = (endtime - starttime);
- Print "elapse_time".

ALGORITHM

Quick sort (A[l….r]) 30

- // Sorts a sub array by recursive quick sort
- //Input : A sub array A[l..r] of A[0..n-1] ,defined by its left and right indices l
- //and r
- // Output : The sub array A[l..r] sorted in non decreasing order

Steps:

if l < r s = Partition (A[l..r])

//s is a split position Quick sort (A[l …s-1])

Quick sort (A[s+1…r])

ALGORITHM

Partition (A[l…r])

- //Partition function divides an array into sub arrays by using its first element as pivot

- // Input : A sub array A[l…r] of A[0…n-1] defined by its left and right indices l and r (l < r)
- // Output : A partition of A[l…r], with the split position returned as this function‟s value

Steps:

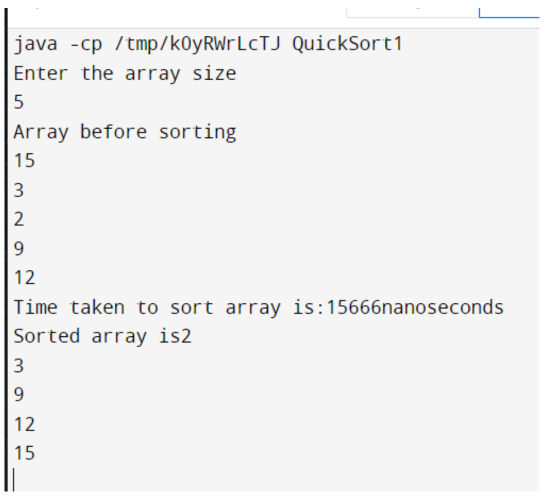
```
p=A[l] i=l;
j=r+1;
repeat
repeat i= i+1 until A[i] >= p
repeat j=j-1 until A[J] <= p
Swap (A[i],A[j]) until
i >=j
Swap (A[i],A[j]) // Undo last
Swap when i>= j Swap (A[l],A[j])
return j
```

PROGRAM:

```
import java.util.Random;
import java.util.Scanner;
public class QuickSort1 {
static int max = 2000;
int partition(int[] a, int low, int high) {
 int p, i, j, temp;
 p = a[low];
 i = low + 1;
 j = high;
 while (low < high) {
 while (a[i] <= p && i < high)
 i++;
 while (a[j] > p)
 j--;
 if (i < j) {
 temp = a[i];
 a[i] = a[j];
 a[j] = temp;
 } else {
 temp = a[low];
 a[low] = a[j];
 a[j] = temp;
 return j;
 }  }
 return j;
 }
void sort(int[] a, int low, int high) {
 if (low < high) {
 int s = partition(a, low, high);
 sort(a, low, s - 1);
 sort(a, s + 1, high);
 } }
public static void main(String[] args) {
int[] a;
int i;
System.out.println("Enter the array size");
 Scanner sc = new Scanner(System.in);
 int n = sc.nextInt();
 a = new int[max];
 Random generator = new Random();
 for (i = 0; i < n; i++)
 a[i] = generator.nextInt(20);
 System.out.println("Array before sorting");
 for (i = 0; i < n; i++)
 System.out.println(a[i] + " ");
 long startTime = System.nanoTime();
 QuickSort1 m = new QuickSort1();
 m.sort(a, 0, n - 1);
 long stopTime = System.nanoTime();
 long elapseTime = (stopTime - startTime);
 System.out.println("Time taken to sort array is:" + elapseTime + "nanoseconds");
 System.out.println("Sorted array is");
 for (i = 0; i < n; i++)
```

```
System.out.println(a[i]);
} }
```

OUTPUT:



```
java -cp /tmp/kOyRWrLcTJ QuickSort1
Enter the array size
5
Array before sorting
15
3
2
9
12
Time taken to sort array is:15666nanoseconds
Sorted array is2
3
9
12
15
```

Experiment -5

Implement merge sort algorithm to sort a given set of elements 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 n. The elements can be read from a file or can be generated using the random number generator.

AIM:

The aim of this program is to sort "n" randomly generated elements using Merge Sort and plotting the graph of the time taken to sort n elements versus n.

DESCRIPTION:

- Call a function to generate list of random numbers (integers).
- Record clock time.
- Call Merge sort function to sort n randomly generated elements.
- Record clock time.
- Measure difference in clock time to get elapse time to sort n elements using Merge sort.
- Print the sorted „ n‟ elements and time taken to sort.
- Repeat the above steps for different values of n as well as to demonstrate worst, best and average case complexity.

ALGORITHM:

- Declare time variables.
- Generate "n" elements randomly using rand () function.
- Call function to record the start time before sorting.
- Call Merge sort function to sort n elements.
- Call function to record the end time after sorting.

- Calculate the time interms of seconds required to sort n elements using Merge sort i.e elapse time elapse_time = (endtime - starttime); print "elapse_time”.

ALGORITHM

Merge sort (A[0…n-1]

- // Sorts array A[0..n-1] by Recursive merge sort
- // Input : An array A[0..n-1] elements
- // Output : Array A[0..n-1] sorted in non decreasing order

```
If n > 1
Copy A[0…(n/2)-1] to B[0…(n/2)-1] Copy
A[n/2…n-1] to C[0…(n/2)-1] Mergesort
(B[0…(n/2)-1])
Mergesort (C[0…(n/2)-1])
Merge(B,C,A)
```

ALGORITHM

Merge (B[0…p-1], C[0…q-1],A[0….p+q-1])

- // merges two sorted arrays into one sorted array
- // Input : Arrays B[0..p-1] and C[0…q-1] both sorted
- // Output : Sorted array A[0…. p+q-1] of the elements of B and C i = 0;

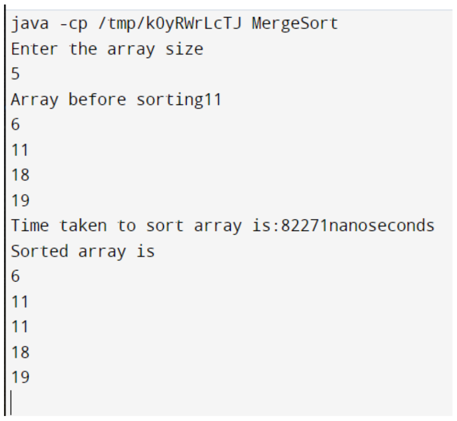
```
j = 0;
k= 0;
while i < p and j < q do
if B[i] <= C[j] A[k]= B[i];
i = i+1;
else
A[k] = C[j];
j = j+1;
k=k+1;
if i = = p Copy C[ j..q-1] to A[k….p+q-1]
else
Copy B[i … p-1] to A[k …p+q-1]
```

PROGRAM:

```
import java.util.Random;
import java.util.Scanner;
public class MergeSort {
static int max = 10000;
```

```
void merge(int[] array, int low, int mid, int high) {
 int i = low;
 int j = mid + 1;
 int k = low;
 int[] resarray;
 resarray = new int[max];
 while (i <= mid && j <= high) {
 if (array[i] < array[j]) {
 resarray[k] = array[i];
 i++;
 k++;
 } else {
 resarray[k] = array[j];
 j++;
 k++;
 } }
 while (i <= mid)
 resarray[k++] = array[i++];
 while (j <= high)
 resarray[k++] = array[j++];
 for (int m = low; m <= high; m++)
 array[m] = resarray[m];
 }
void sort(int[] array, int low, int high) {
 if (low < high) {
 int mid = (low + high) / 2;
 sort(array, low, mid);
 sort(array, mid + 1, high);
 merge(array, low, mid, high);
 } }
public static void main(String[] args) {
 int[] array;
 int i;
System.out.println("Enter the array size");
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
array = new int[max];
Random generator = new Random();
for (i = 0; i < n; i++)
array[i] = generator.nextInt(20);
System.out.println("Array before sorting");
for (i = 0; i < n; i++)
System.out.println(array[i] + " ");
long startTime = System.nanoTime();
MergeSort m = new MergeSort();
m.sort(array, 0, n - 1);
long stopTime = System.nanoTime();
long elapseTime = (stopTime - startTime);
System.out.println("Time taken to sort array is:" + elapseTime + "nanoseconds");
System.out.println("Sorted array is");
for (i = 0; i < n; i++)
System.out.println(array[i]);
}}
```

OUTPUT:



```
java -cp /tmp/k0yRWrLcTJ MergeSort
Enter the array size
5
Array before sorting11
6
11
18
19
Time taken to sort array is:82271nanoseconds
Sorted array is
6
11
11
18
19
```

Experiment -6

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

AIM:

Kruskal's Algorithm for computing the minimum spanning tree is directly based on the generic MST algorithm. It builds the MST in forest. Prim's algorithm is based on a generic MST algorithm. It uses greedy approach.

DESCRIPTION:

(A). Kruskal's Algorithm

Start with an empty set A, and select at every stage the shortest edge that has not been chosen or rejected, regardless of where this edge is situated in graph.

- Initially, each vertex is in its own tree in forest.
- Then, algorithm consider each edge in turn, order by increasing weight.
- If an edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST, and two trees connected by an edge (u, v) are merged into a single tree.
- On the other hand, if an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded.

(B). Prim's Algorithm

Choose a node and build a tree from there selecting at every stage the shortest available edge that can extend the tree to an additional node.

- Prim's algorithm has the property that the edges in the set A always form a single tree.
- We begin with some vertex v in a given graph G =(V, E), defining the initial set of vertices A.
- In each iteration, we choose a minimum-weight edge (u, v), connecting a vertex v in the set A to the vertex u outside of set A.
- The vertex u is brought in to A. This process is repeated until a spanning tree is formed.
- Like Kruskal's algorithm, here too, the important fact about MSTs is we always choose the smallest-weight edge joining a vertex inside set A to the one outside the set A.
- The implication of this fact is that it adds only edges that are safe for A; therefore when the algorithm terminates, the edges in set A form a MST

PROGRAM:

(a) Kruskal's algorithm

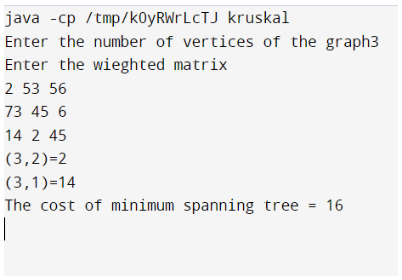
```
import java.util.Scanner;

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 < j)
 parent[i] = j;
 else
 parent[j] = i;
 }
void krkl(int[][] a, int n) {
 int u = 0, v = 0, min, k = 0, i, j, sum = 0;
 while (k < n - 1) {
 min = 99;
 for (i = 1; 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];
int i, j;
System.out.println("Enter the number of vertices of the graph");
Scanner sc = new Scanner(System.in);
int n;
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();
} }
```

Prim's algorithm

```
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;
 int min;
 int sum = 0;
 int u = 0, v = 0;
 int 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();
} }
```

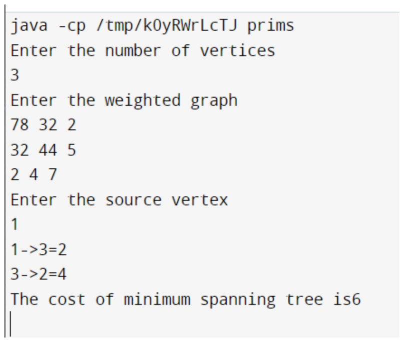
OUTPUT:

Kruskal's algorithm



```
java -cp /tmp/kOyRWrLcTJ kruskal
Enter the number of vertices of the graph3
Enter the wieghted matrix
2 53 56
73 45 6
14 2 45
(3,2)=2
(3,1)=14
The cost of minimum spanning tree = 16
```

Prim's algorithm



```
java -cp /tmp/kOyRWrLcTJ prims
Enter the number of vertices
3
Enter the weighted graph
78 32 2
32 44 5
2 4 7
Enter the source vertex
1
1->3=2
3->2=4
The cost of minimum spanning tree is6
```

Experiment -7

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

AIM:

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

ALGORITHM:

- // Input: A weighted connected graph G ={V,E}, source s
- // Output dv: the distance-vertex matrix
  - Read number of vertices of graph G
  - Read weighted graph G o Print weighted graph
  - Initialize distance from source for all vertices as weight between source node and other vertices, i, and none in tree
- // initial condition
  - For all other vertices, dv[i] = wt_graph[s,i], TV[i]=0, prev[i]=0 dv[s] = 0,

prev[s] = s // source vertex

- Repeat for y = 1 to n v = next vertex with minimum dv value, by calling FindNextNear() Add v to tree.

```
For all the adjacent u of v and u is not added to the tree,
if dv[u]> dv[v] + wt_graph[v,u]
then dv[u]= dv[v] + wt_graph[v,u] and prev[u]=v.
• findNextNear
//Input: graph, dv matrix
//Output: j the next nearest vertex minm = 9999
For k =1 to n
if k vertex is not selected in tree and
if dv[k] < minm { minm = dv [ k] j=k
}
```

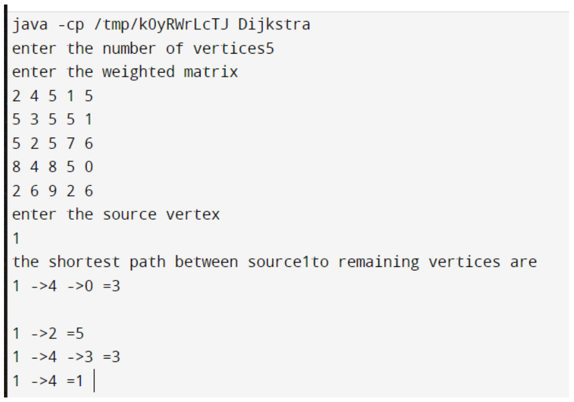
PROGRAM:

```
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 < n; v++) {
 d[v] = 99;
 p[v] = -1;
 }
 d[s] = 0;
 for (i = 0; i < n; i++) {
 min = 99;
 for (j = 0; j < n; j++) {
 if (d[j] < min && visited[j] == 0) {
 min = d[j];
 u = j;
 } }
 visited[u] = 1;
 for (v = 0; v < n; v++) {
```

```
 if ((d[u] + a[u][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:



```
java -cp /tmp/kOyRWrLcTJ Dijkstra
enter the number of vertices5
enter the weighted matrix
2 4 5 1 5
5 3 5 5 1
5 2 5 7 6
8 4 8 5 0
2 6 9 2 6
enter the source vertex
1
the shortest path between source1to remaining vertices are
1 ->4 ->0 =3

1 ->2 =5
1 ->4 ->3 =3
1 ->4 =1
```

Experiment -8

Implement in Java, the 0/1 Knapsack problem using Greedy method and Dynamic Programming method

AIM:

We are given a set of n items from which we are to select some number of items to be carried in a knapsack(BAG). Each item has both a weight and a profit. The objective is to choose the set of items that fits in the knapsack and maximizes the profit.

Given a knapsack with maximum capacity W, and a set S consisting of n items , Each item i has some weight wi and benefit value bi (all wi , bi and W are integer values).

Problem: How to pack the knapsack to achieve maximum total value of packed items?

ALGORITHM:

(A).USING : Dynamic programming

It gives us a way to design custom algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).

ALGORITHM

- //Input: (n items, W weight of sack) Input: n, wi,,, vi and W – all integers
- //Output: V(n,W)

Steps:

- // Initialization of first column and first row elements
    - Repeat for i = 0 to n

set V(i,0) = 0

- Repeat for j = 0 to W Set V(0,j) = 0 //complete remaining entries row by row
- Repeat for i = 1 to n

repeat for j = 1 to W

if (wi <= j) V(i,j)) = max{ V(i-1,j), V(i-1,j-wi) + vi }

if (wi > j) V(i,j) = V(i-1,j)

- Print V(n,W)

Greedy Method Algorithms:

- General design technique o Used for optimization problems
- Simply choose best option at each step
- Solve remaining sub-problems after making greedy step
- Fractional Knapsack: (using greedy method)
- N items (can be the same or different)
- Can take fractional part of each item (eg bags of gold dust)

Algorithm:

- Assume knapsack holds weight W and items have value vi and weight wi
- Rank items by value/weight ratio:

vi / wi Thus: vi / wi ≥ vj / wj, for all i ≤ j

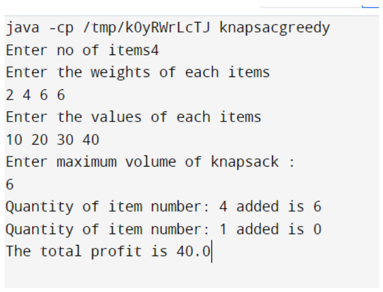
- Consider items in order of decreasing ratio
- Take as much of each item as possible based on knapsack‟s capacity

PROGRAM:

(b)Greedy method

```
import java.util.Scanner;
public class knapsacgreedy {
public static void main(String[] args) {
 int i, j = 0, max_qty, m, n;
 float sum = 0, max;
 Scanner sc = new Scanner(System.in);
 int array[][] = new int[2][20];
 System.out.println("Enter no of items");
 n = sc.nextInt();
 System.out.println("Enter the weights of each items");
 for (i = 0; i < n; i++)
 array[0][i] = sc.nextInt();
 System.out.println("Enter the values of each items");
for (i = 0; i < n; i++)
array[1][i] = sc.nextInt();
System.out.println("Enter maximum volume of knapsack :");
max_qty = sc.nextInt();
m = max_qty;
while (m >= 0) {
max = 0;
for (i = 0; i < n; i++) {
if (((float) array[1][i]) / ((float) array[0][i]) > max) {
max = ((float) array[1][i]) / ((float) array[0][i]);
j = i;
} }
if (array[0][j] > m) {
System.out.println("Quantity of item number: " + (j + 1) + " added is " + m);
sum += m * max;
m = -1;
} else {
System.out.println("Quantity of item number: " + (j + 1) + " added is " +
array[0][j]);
m -= array[0][j];
sum += (float) array[1][j];
array[1][j] = 0;
} }
System.out.println("The total profit is " + sum);
sc.close();
}}
```

OUTPUT:



```
java -cp /tmp/kOyRWrLcTJ knapsacgreedy
Enter no of items4
Enter the weights of each items
2 4 6 6
Enter the values of each items
10 20 30 40
Enter maximum volume of knapsack :
6
Quantity of item number: 4 added is 6
Quantity of item number: 1 added is 0
The total profit is 40.0
```

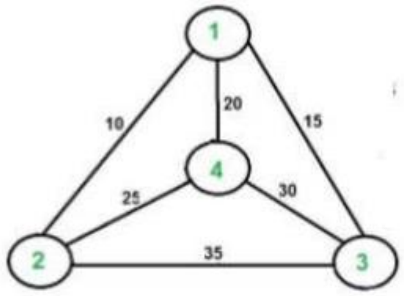
Experiment -9

Write Java programs to Implement Travelling Salesperson problem using Dynamic programming

AIM:

Travelling Sales Person problem: Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in above figure. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.



Solution using Dynamic Programming:

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i.

We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on.

Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i}, C(S, i) = dist(1,i)

Else if size of S is greater than 2.

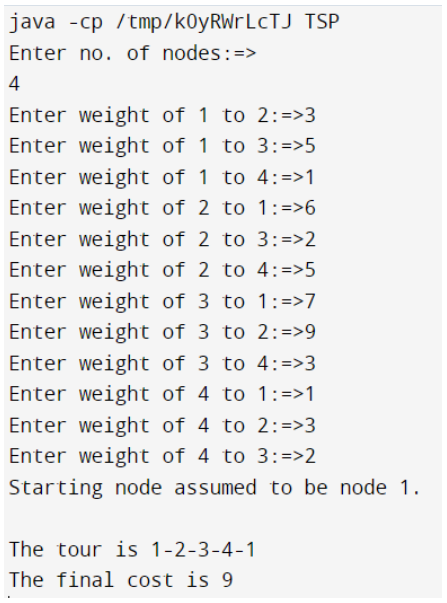
C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j!=1.

PROGRAM:

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

```
class TSPExp {
int weight[][], n, tour[], finalCost;
final int INF = 1000;
 TSPExp() {
 Scanner s = new Scanner(System.in);
 System.out.println("Enter no. of nodes:=>");
 n = s.nextInt();
 weight = new int[n][n];
 tour = new int[n - 1];
 for (int i = 0; i < n; i++) {
 for (int j = 0; j < n; j++) {
 if (i != j) {
 System.out.print("Enter weight of " + (i + 1) + " to " + (j + 1) + ":=>");
 weight[i][j] = s.nextInt();
 } } }
 System.out.println();
 System.out.println("Starting node assumed to be node 1.");
 eval();
 }
public int COST(int currentNode, int inputSet[], int setSize) {
 if (setSize == 0)
 return weight[currentNode][0];
 int min = INF;
 int setToBePassedOnToNextCallOfCOST[] = new int[n - 1];
 for (int i = 0; i < setSize; i++) {
 int k = 0;// initialise new set
 for (int j = 0; j < setSize; j++) {
 if (inputSet[i] != inputSet[j])
 setToBePassedOnToNextCallOfCOST[k++] = inputSet[j];
 }
 int temp = COST(inputSet[i], setToBePassedOnToNextCallOfCOST, setSize - 1);
 if ((weight[currentNode][inputSet[i]] + temp) < min) {
 min = weight[currentNode][inputSet[i]] + temp;
 } }
 return min;
 }
public int MIN(int currentNode, int inputSet[], int setSize) {
 if (setSize == 0)
 return weight[currentNode][0];
 int min = INF, minindex = 0;
 int setToBePassedOnToNextCallOfCOST[] = new int[n - 1];
 for (int i = 0; i < setSize; i++)// considers each node of inputSet
 {
 int k = 0;
 for (int j = 0; j < setSize; j++) {
 if (inputSet[i] != inputSet[j])
 setToBePassedOnToNextCallOfCOST[k++] = inputSet[j];
 }
 int temp = COST(inputSet[i], setToBePassedOnToNextCallOfCOST, setSize - 1);
 if ((weight[currentNode][inputSet[i]] + temp) < min) {
 min = weight[currentNode][inputSet[i]] + temp;
 minindex = inputSet[i];
 } }
 return minindex;
 }
public void eval() {
 int dummySet[] = new int[n - 1];
 for (int i = 1; i < n; i++)
 dummySet[i - 1] = i;
 finalCost = COST(0, dummySet, n - 1);
 constructTour();
 }
public void constructTour() {
 int previousSet[] = new int[n - 1];
 int nextSet[] = new int[n - 2];
 for (int i = 1; i < n; i++)
 previousSet[i - 1] = i;
 int setSize = n - 1;
 tour[0] = MIN(0, previousSet, setSize);
 for (int i = 1; i < n - 1; i++) {
 int k = 0;
 for (int j = 0; j < setSize; j++) {
 if (tour[i - 1] != previousSet[j])
 nextSet[k++] = previousSet[j];
 }
 --setSize;
 tour[i] = MIN(tour[i - 1], nextSet, setSize);
 for (int j = 0; j < setSize; j++)
 previousSet[j] = nextSet[j];
 }
 display();
 }
public void display() {
 System.out.println();
 System.out.print("The tour is 1-");
 for (int i = 0; i < n - 1; i++)
 System.out.print((tour[i] + 1) + "-");
 System.out.print("1");
 System.out.println();
 System.out.println("The final cost is " + finalCost);
 } }
class TSP {
public static void main(String args[]) {
 TSPExp obj = new TSPExp();
 } }
```

OUTPUT:



```
java -cp /tmp/kOyRWrLcTJ TSP
Enter no. of nodes:=>
4
Enter weight of 1 to 2:=>3
Enter weight of 1 to 3:=>5
Enter weight of 1 to 4:=>1
Enter weight of 2 to 1:=>6
Enter weight of 2 to 3:=>2
Enter weight of 2 to 4:=>5
Enter weight of 3 to 1:=>7
Enter weight of 3 to 2:=>9
Enter weight of 3 to 4:=>3
Enter weight of 4 to 1:=>1
Enter weight of 4 to 2:=>3
Enter weight of 4 to 3:=>2
Starting node assumed to be node 1.

The tour is 1-2-3-4-1
The final cost is 9
```

Experiment -10

Write a Java program to Implement All-Pairs Shortest Paths problem using Floyd's algorithm.

AIM:

The Floyd–Warshall algorithm (sometimes known as the WFI Algorithm or Roy–Floyd algorithm) is a graph analysis algorithm for finding shortest paths in a weighted graph (with positive or negative edge weights). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices though it does not return details of the paths themselves. The algorithm is an example of dynamic programming.

ALGORITHM:

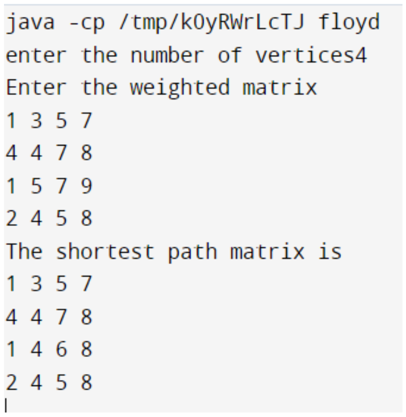
Floyd's Algorithm

- Accept no .of vertices
- Call graph function to read weighted graph // w(i,j)
- Set D[ ] <- weighted graph matrix // get D {d(i,j)} for k=0
- // If there is a cycle in graph, abort. How to find?
- Repeat for k = 1 to n
- Repeat for i = 1 to n
- Repeat for j = 1 to n D[i,j] = min {D[i,j], D[i,k] + D[k,j]}
- Print D

PROGRAM:

```
import java.util.Scanner;
public class floyd {
void flyd(int[][] w, int n) {
 int i, j, k;
 for (k = 1; k <= n; k++)
 for (i = 1; i <= n; i++)
 for (j = 1; j <= n; j++)
 w[i][j] = Math.min(w[i][j], w[i][k] + w[k][j]);
 }
public static void main(String[] args) {
 int a[][] = new int[10][10];
 int n, i, j;
 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 = 1; i <= n; i++)
 for (j = 1; j <= n; j++)
 a[i][j] = sc.nextInt();
 floyd f = new floyd();
 f.flyd(a, n);
 System.out.println("The shortest path matrix is");
 for (i = 1; i <= n; i++) {
 for (j = 1; j <= n; j++) {
 System.out.print(a[i][j] + " ");
 }
 System.out.println();
 }
 sc.close();
 } }
```

OUTPUT:


```
java -cp /tmp/kOyRWrLcTJ floyd
enter the number of vertices4
Enter the weighted matrix
1 3 5 7
4 4 7 8
1 5 7 9
2 4 5 8
The shortest path matrix is
1 3 5 7
4 4 7 8
1 4 6 8
2 4 5 8
```

<u>Experiment -11</u>

Design and implement in Java to find a subset of a given set S = {Sl, S2,.....,Sn} of n positive integers whose SUM is equal to a given positive integer d. For example, if S ={1, 2, 5, 6, 8} and d= 9, there are two solutions {1,2,6}and {1,8}. Display a suitable message, if the given problem instance doesn't have a solution.

AIM:

An instance of the Subset Sum problem is a pair (S, t), where S = {x1, x2, ..., xn} is a set of positive integers and t (the target) is a positive integer. The decision problem asks for a subset of S whose sum is as large as possible, but not larger than t.

This problem is NP-complete. This problem arises in practical applications. Similar to the knapsack problem we may have a truck that can carry at most t pounds and we have n different boxes to ship and the with box weighs xi pounds. The naive approach of computing the sum of the elements of every subset of S and then selecting the best requires exponential time. Below we present an exponential time exact algorithm.

ALGORITHM:

- accept n: no of items in set o accept their values, sk in increasing order
- accept d: sum of subset desired o initialise x[i] = -1 for all i
- check if solution possible or not
- if possible then call SumOfSub(0,1,sum of all elements)
- SumOfSub (s, k, r)
- //Values of x[j], 1 <= j < k, have been determined
- //Node creation at level k taking place: also call for creation at level K+1 if possible
- //s= sum of 1 to k-1 elements and r is sum of k to n elements
- //generating left child that means including k in solution
- Set x[k] = 1
- If (s + s[k] = d) then subset found, print solution
- If (s + s[k] + s[k+1] <=d)

then SumOfSum (s + s[k], k+1, r – s[k])

//Generate right child i.e. element k absent

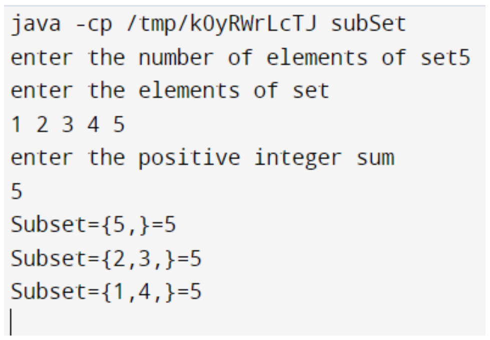
- If (s + r - s[k] >=d) AND (s + s[k+1])<=d

```
THEN
{
x[k]=0;
SumOfSub(s, k+1, r – s[k])
}
```

PROGRAM:

```
import java.util.Scanner;
public class subSet {
void subset(int num, int n, int x[]) {
 int i;
 for (i = 1; i <= n; i++)
 x[i] = 0;
 for (i = n; num != 0; i--) {
 x[i] = num % 2;
 num = num / 2;
 } }
public static void main(String[] args) {
int a[] = new int[10];
 int x[] = new int[10];
 int n, d, sum, present = 0;
 int j;
 System.out.println("enter the number of elements of set");
 Scanner sc = new Scanner(System.in);
 n = sc.nextInt();
 System.out.println("enter the elements of set");
 for (int i = 1; i <= n; i++)
 a[i] = sc.nextInt();
 System.out.println("enter the positive integer sum");
 d = sc.nextInt();
 if (d > 0) {
 for (int i = 1; i <= Math.pow(2, n) - 1; i++) {
 subSet s = new subSet();
 s.subset(i, n, x);
sum = 0;
for (j = 1; j <= n; j++)
if (x[j] == 1)
sum = sum + a[j];
if (d == sum) {
System.out.print("Subset={");
present = 1;
for (j = 1; j <= n; j++)
if (x[j] == 1)
System.out.print(a[j] + ",");
System.out.print("}=" + d);
System.out.println();
} } }
if (present == 0)
System.out.println("Solution does not exists");
} }
```

OUTPUT:



```
java -cp /tmp/k0yRWrLcTJ subSet
enter the number of elements of set5
enter the elements of set
1 2 3 4 5
enter the positive integer sum
5
Subset={5,}=5
Subset={2,3,}=5
Subset={1,4,}=5
```

Experiment -12

Design and implement in Java to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle.

AIM:

Design and implement in Java to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle.

DESCRIPTION:

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

Input:

A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

Output:

An array path[V] that should contain the Hamiltonian Path. path[i] should represent the vertex i in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

ALGORITHM:

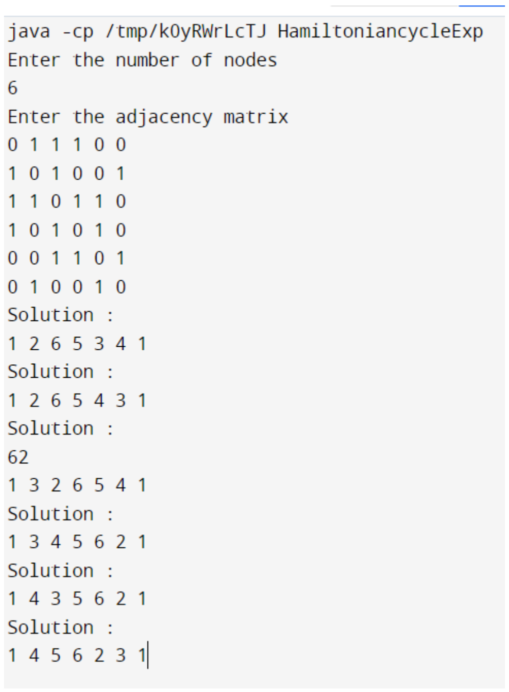
```
hamiltonian(p,index)
if index>n then
path is complete display the values in p
else
for each node v in G
if v can be added to the path then add v to path p and call hamiltonian(p,index+1)
end Hamiltonian
```

PROGRAM:

```
import java.util.*;
class Hamiltoniancycle {
private int adj[][], x[], n;
public Hamiltoniancycle() {
 Scanner src = new Scanner(System.in);
 System.out.println("Enter the number of nodes");
 n = src.nextInt();
 x = new int[n];
 x[0] = 0;
 for (int i = 1; i < n; i++)
 x[i] = -1;
 adj = new int[n][n];
 System.out.println("Enter the adjacency matrix");
 for (int i = 0; i < n; i++)
 for (int j = 0; j < n; j++)
 adj[i][j] = src.nextInt();
 }
public void nextValue(int k) {
 int i = 0;
 while (true) {
 x[k] = x[k] + 1;
 if (x[k] == n)
 x[k] = -1;
 if (x[k] == -1)
 return;
 if (adj[x[k - 1]][x[k]] == 1)
 for (i = 0; i < k; i++)
 if (x[i] == x[k])
 break;
 if (i == k)
 if (k < n - 1 || k == n - 1 && adj[x[n - 1]][0] == 1)
 return;
 }  }
public void getHCycle(int k) {
 while (true) {
 nextValue(k);
 if (x[k] == -1)
 return;
 if (k == n - 1) {
 System.out.println("\nSolution : ");
 for (int i = 0; i < n; i++)
 System.out.print((x[i] + 1) + " ");
 System.out.println(1);
 } else
 getHCycle(k + 1);
 } } }
```

```
class HamiltoniancycleExp {
public static void main(String args[]) {
 Hamiltoniancycle obj = new Hamiltoniancycle();
 obj.getHCycle(1);
 } }
```

OUTPUT:



```
java -cp /tmp/kOyRWrLcTJ HamiltoniancycleExp
Enter the number of nodes
6
Enter the adjacency matrix
0 1 1 1 0 0
1 0 1 0 0 1
1 1 0 1 1 0
1 0 1 0 1 0
0 0 1 1 0 1
0 1 0 0 1 0
Solution :
1 2 6 5 3 4 1
Solution :
1 2 6 5 4 3 1
Solution :
62
1 3 2 6 5 4 1
Solution :
1 3 4 5 6 2 1
Solution :
1 4 3 5 6 2 1
Solution :
1 4 5 6 2 3 1
```