EXPERIMENT NO: 01

NAME OF THE EXPERIMENT: Quick sort algorithm.

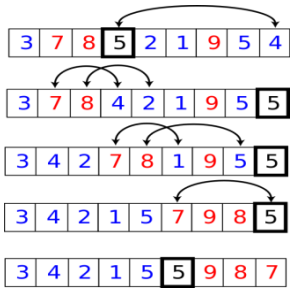
AIM: Implement Quick sort algorithm for sorting a list of integers in ascending order

THEORY:

Quick sort or partition-exchange sort is a fast sorting algorithm, which is using divide and conquer algorithm. Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

Steps to implement Quick sort:

1) Choose an element, called pivot, from the list. Generally pivot can be the middle index element.
2) Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3) Recursively applies the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.



Source Code:-

```
import java.util.*;

public class QuickSort{

   public static void main(String args[]){

        Scanner s=new Scanner(System.in);

        System.out.print("Enter the size of Array :");

        int size=s.nextInt();

        int []numbers=new int[size];
```

```
   for(int k=0;k<size;k++){
      numbers[k]=s.nextInt();
   }
System.out.print("Before sorting, numbers are ");
for(int i = 0; i < numbers.length; i++){
System.out.print(numbers[i]+" ");
}
System.out.println();
quickSortInAscendingOrder(numbers,0,numbers.length-1);
System.out.print("After sorting, numbers are ");
for(int i = 0; i < numbers.length; i++){
System.out.print(numbers[i]+" ");
}
}
static void quickSortInAscendingOrder (int[] numbers, int low, int high){
int i=low; int j=high; int temp;
int middle=numbers[(low+high)/2];
while (i<j){
while (numbers[i]<middle){
i++;
}
while (numbers[j]>middle){
j--;
}
if (i<=j){
temp=numbers[i]; numbers[i]=numbers[j]; numbers[j]=temp;
i++; j--;
}
```

```
        }
        if (low<j){
        quickSortInAscendingOrder(numbers, low, j);
        }
        if (i<high){
        quickSortInAscendingOrder(numbers, i, high);
        }
    }
}
```

OUTPUT:-

Enter the size of Array :5

57

61

51

47

21

Before sorting, numbers are 57 61 51 47 21

After sorting, numbers are 21 47 51 57 61

PS C:\Users\TAUSIF\OneDrive\Desktop\NodeJs\Node.Code>

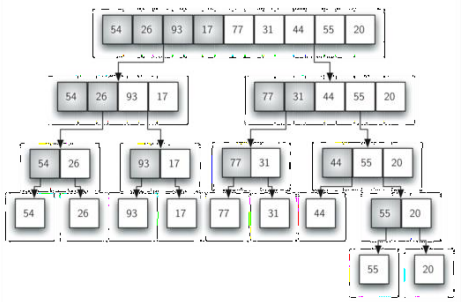
EXPERIMENT No.02

NAME OF THE EXPERIMENT: Merge sort.

AIM: Implement Merge sort algorithm for sorting a list of integers in ascending order**.**

THEORY:

Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a **merge**, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list. Fig1 shows our familiar example list as it is being split by mergeSort.Fig2 shows the simple lists, now sorted, as they are merged back together.



SOURCE CODE:

```
import java.util.*;

public class MergeSort{

   static int max=1000;

   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]+" ");
        MergeSort m=new MergeSort();
        m.sort(array,0,n-1);
        System.out.println("Sorted array is");
        for(i=0;i<n;i++)
           System.out.println(array[i]);
        }
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);
        }
    }
}
```

OUTPUT:-

Enter the array size

10

Array before sorting

17 5 1 12 5 16 11 5 12 2

Sorted array is

1 2 5 5 5 11 12 12 16 17

PS C:\Users\TAUSIF\OneDrive\Desktop\NodeJs\Node.Code>

EXPERIMENT NO: 03

NAME OF THE EXPERIMENT: 0/1 Knapsack problem

AIM: Implement Dynamic Programming algorithm for the 0/1 Knapsack problem.

THEORY: In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same.
0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

SOURCE CODE:

```
import java.util.*;

public 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){

   int i, w;

   int K[][] = new int[n+1][W+1];

   for (i = 0; i <= n; i++){

      for (w = 0; w <= W; w++){

         if (i==0 || w==0)
```

```
            K[i][w] = 0;
            else if (wt[i-1] <= w)
            K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
            else
            K[i][w] = K[i-1][w];
        }
    }
    return K[n][W];
    }
    public static void main(String args[]){
        Scanner sc=new Scanner(System.in);
    System.out.println("Enter the size of Array n:");
    int n=sc.nextInt();
    int val[] = new int[n];
    int  wt[] = new int[n];
    System.out.println("Enter the value and its weight");
    for(int i=0;i<n;i++){
        val[i]=sc.nextInt();
        wt[i]=sc.nextInt();
    }
    System.out.println("Enter the weight w:");
    int W =sc.nextInt();
    System.out.println(knapSack(W, wt, val, n));
    }
}
```

OUTPUT:-

Enter the size of Array n:

3

Enter the value and its weight

60 10

100 20

120 30

Enter the weight w:

50

220

PS C:\Users\TAUSIF\OneDrive\Desktop\NodeJs\Node.Code>

EXPERIMENT NO: 04

NAME OF THE EXPERIMENT: Single source shortest path using Dijkstra's algorithm.

AIM: Implement Dijkstra's algorithm for the Single source shortest path problem

THEORY: Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph $G = (V, E)$, where all the edges are non-negative (i.e., $w(u, v) \geq 0$ for each edge $(u, v) \in E$).

In the following algorithm, we will use one function ***Extract-Min()***, which extracts the node with the smallest key.

SOURCE CODE:

```
import java.util.*;

public class DijKstra{

   int minDistance(int dist[], Boolean sptSet[],int V){

   int min = Integer.MAX_VALUE, min_index=-1;

   for (int v = 0; v < V; v++)

   if (sptSet[v] == false && dist[v] <= min){

   min = dist[v];

   min_index = v;

   }

   return min_index;
```

```
}
void printSolution(int dist[], int V){
System.out.println("Vertex Distance from Source");
for (int i = 0; i < V; i++)
System.out.println(i+" tt "+dist[i]);
}
void dijkstra(int graph[][], int src,int V){
int dist[] = new int[V];
Boolean sptSet[] = new Boolean[V];
for (int i = 0; i < V; i++){
dist[i] = Integer.MAX_VALUE; sptSet[i] = false;
}
dist[src] = 0;
for (int count = 0; count < V-1; count++) {
int u = minDistance(dist, sptSet,V);
sptSet[u] = true;
for (int v = 0; v < V; v++)
if (!sptSet[v] && graph[u][v]!=0 &&
dist[u] != Integer.MAX_VALUE && dist[u]+graph[u][v] < dist[v])
dist[v] = dist[u] + graph[u][v];
}
printSolution(dist, V);
}
public static void main (String[] args){
   Scanner s=new Scanner(System.in);
   System.out.println("enter the matrix size :");
   int n=s.nextInt();
int graph[][] = new int[n][n];
```

```
    System.out.println("Enter the value of vertex :");
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            graph[i][j]=s.nextInt();
        }
    }
    DAALAB t = new DAALAB();
        t.dijkstra(graph, 0, n);
    }
}
```

OUTPUT:-

enter the matrix size :

9

Enter the value of vertex :

0 4 0 0 0 0 0 8 0

4 0 8 0 0 0 0 11 0

0 8 0 7 0 4 0 0 2

0 0 7 0 9 14 0 0 0

0 0 0 9 0 10 0 0 0

0 0 4 14 10 0 2 0 0

0 0 0 0 0 2 0 1 6

8 11 0 0 0 0 1 0 7

0 0 2 0 0 0 6 7 0

Vertex Distance from Source

0 tt 0

1 tt 4

2 tt 12

3 tt 19

4 tt 21

5 tt 11

6 tt 9

7 tt 8

8 tt 14

PS C:\Users\TAUSIF\OneDrive\Desktop\NodeJs\Node.Code>

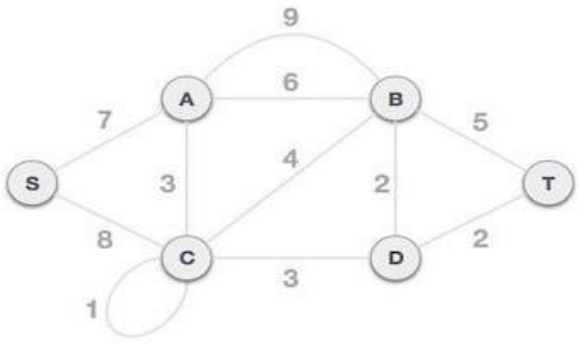
EXPERIMENT NO: 05

NAME OF THE EXPERIMENT: Minimum Cost Spanning Tree using Kruskal's.

AIM: Implements Kruskal's algorithm to generate minimum cost Spanning tree.

THEORY: Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −



SOURCE CODE:

```
import java.util.*;
```

```
import java.lang.*;
import java.io.*;
class Graph{
   class Edge implements Comparable<Edge>{
   int src, dest, weight;
   public int compareTo(Edge compareEdge){
   return this.weight-compareEdge.weight;
   }
   };
   class subset{
   int parent, rank;
   };
   int V, E;
   Edge edge[];
   Graph(int v,int e){
   V = v; E = e;
   edge = new Edge[E];
   for (int i=0; i<e; ++i)
   edge[i] = new Edge();
   }
   int find(subset subsets[],int i){
      if(subsets[i].parent!=i)
      subsets[i].parent = find(subsets, subsets[i].parent);
   return subsets[i].parent;
   }
   void Union(subset subsets[], int x, int y){
   int xroot = find(subsets, x);
   int yroot = find(subsets, y);
```

```
if (subsets[xroot].rank < subsets[yroot].rank)
subsets[xroot].parent = yroot;
else if (subsets[xroot].rank > subsets[yroot].rank)
subsets[yroot].parent = xroot;
else{
subsets[yroot].parent = xroot;
subsets[xroot].rank++;
}
}
void KruskalMST(){
Edge result[] = new Edge[V];
int e=0;
int i = 0;
for(;i<V;i++)
result[i] = new Edge();
Arrays.sort(edge);
subset subsets[]=new subset[V];
for(i=0; i<V; ++i)
subsets[i]=new subset();
for(int v=0;v<V;v++){
subsets[v].parent = v;
subsets[v].rank = 0;
}
i = 0;
while(e<V-1){
   Edge next_edge=new Edge();
   next_edge=edge[i++];
   int x = find(subsets, next_edge.src);
```

```
   int y = find(subsets, next_edge.dest);
if(x!=y){
result[e++] = next_edge;
Union(subsets, x, y);
}
}
System.out.println("Following are the edges in " + "the constructed MST");
for (i = 0; i < e; ++i)
System.out.println (result[i].src+" -- " + result[i].dest+" == " + result[i].weight);
}
public static void main (String [] args)
{
int V = 4; // Number of vertices in graph
int E = 5;
Graph graph=new Graph(V,E);
graph.edge[0].dest = 0;
graph.edge[0].dest = 1;
graph.edge[0].weight = 10;
graph.edge[1].dest = 0;
graph.edge[1].dest = 2;
graph.edge[1].weight = 6;
graph.edge[2].dest = 0;
graph.edge[2].dest = 3;
graph.edge[2].weight = 5;
graph.edge[3].dest = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 15;
graph.edge[4].dest = 2;
```

```
    graph.edge[4].dest = 3;
    graph.edge[4].weight = 4;

    graph.KruskalMST();
    }
}
```

OUTPUT:-

Following are the edges in the constructed MST

0 -- 3 == 4

0 -- 2 == 6

0 -- 1 == 10

C:\Users\TAUSIF\OneDrive\Desktop\JAVA PROGRAM\20BTCS047HY>

EXPERIMENT NO: 06

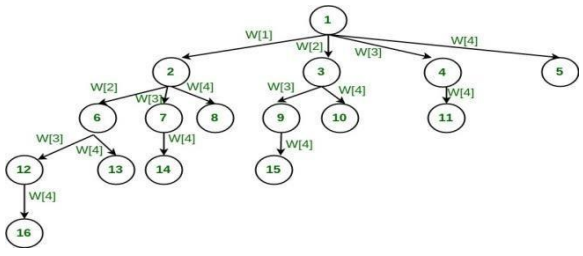
NAME OF THE EXPERIMENT: Sum of Sub Sets Problem.

AIM: Implement the backtracking algorithm for the sum of subsets problem.

THEORY: Backtracking Algorithm for Subset Sum

Using exhaustive search we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Assume given set of 4 elements, say w[1] … w[4]. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts approach of generating variable sized tuple.



SOURCE CODE:

```
import java.util.*;

public class SubSetSum {

public static void main(String[] args) {

Scanner s=new Scanner(System.in);

System.out.println("Enter th array size:");

int n=s.nextInt();

int [] input=new int[n];

System.out.println("Enter the Array Value:");

for(int i=0;i<n;i++)

input[i]=s.nextInt();

System.out.println("Enter the  value of targetSum :");

int targetSum=s.nextInt();

//int[] input = { 2, 3, 4, 5 };

//int targetSum = 7;

SubSetSum subSetSum = new SubSetSum();

subSetSum.findSubSets(input, targetSum);

}

private int[] set;

private int[] selectedElements;

private int targetSum;

private int numOfElements;

public void findSubSets(int[] set, int targetSum) {

this.set = set;

this.numOfElements = set.length; this.targetSum = targetSum;

selectedElements = new int[numOfElements]; quicksort(set, 0, numOfElements-1);

int sumOfAllElements = 0; for(int element : set){

sumOfAllElements += element;
```

```
}
findSubSets(0, 0, sumOfAllElements);
}
private void findSubSets(int sumTillNow, int index, int sumOfRemaining) {
selectedElements[index] = 1;
if (targetSum == set[index] + sumTillNow) {
print();
}
if ((index + 1 < numOfElements) && (sumTillNow + set[index] + set[index + 1] <= targetSum)) {
findSubSets(sumTillNow + set[index], index + 1, sumOfRemaining - set[index]);
}
selectedElements[index] = 0;
if ((index + 1 < numOfElements) && (sumTillNow + set[index + 1] <= targetSum) && (sumTillNow +
sumOfRemaining - set[index] >= targetSum)) {
findSubSets(sumTillNow, index + 1, sumOfRemaining - set[index]);
}
}
private void print() {
for (int i = 0; i < numOfElements; i++) { if (selectedElements[i] == 1) {
System.out.print(set[i]+" ");
}
}
System.out.println();
}
private void quicksort(int[] arr, int start, int end) {
if (start < end) {
swap(arr, (start + (end - start) / 2), end);
int pIndex = partition(arr, start, end);
```

```
quicksort(arr, start, pIndex - 1);
quicksort(arr, pIndex + 1, end);
}
}
private int partition(int[] arr, int start, int end) {
int pIndex = start, pivot = arr[end];
for (int i = start; i < end; i++) {
if (arr[i] < pivot) {
swap(arr,pIndex,i);
pIndex++;
}
}
swap(arr,pIndex,end);
return pIndex;
}
private void swap(int[] arr, int index1, int index2) {
int temp = arr[index1];
arr[index1] = arr[index2];
arr[index2] = temp;
}
}
```

OUTPUT:-

Enter the array size:

4

Enter the Array Value:

1

3

4

6

Enter the value of targetSum :

4

1 3

4

C:\Users\TAUSIF\OneDrive\Desktop\JAVA PROGRAM\20BTCS047HY>

EXPERIMENT NO: 07

NAME OF THE EXPERIMENT: Minimum cost Spanning Tree Using Prim's Algorithm.

AIM: Implements Prim's algorithm to generate minimum cost spanning tree.

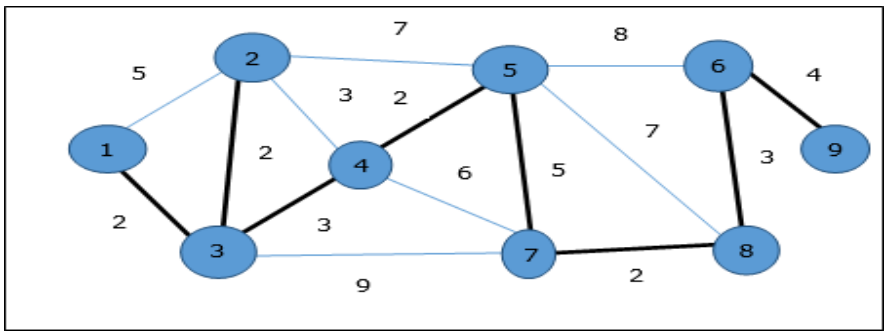
THEORY: Using Prim's algorithm, we can start from any vertex, let us start from vertex **1**.

Vertex **3** is connected to vertex **1** with minimum edge cost, hence edge **(1, 2)** is added to the spanning tree.

Next, edge **(2, 3)** is considered as this is the minimum among edges **{(1, 2), (2, 3), (3, 4), (3, 7)}**.

In the next step, we get edge **(3, 4)** and **(2, 4)** with minimum cost. Edge **(3, 4)** is selected at random.

In a similar way, edges **(4, 5), (5, 7), (7, 8), (6, 8)** and **(6, 9)** are selected. As all the vertices are visited, now the algorithm stops.



SOURCE CODE:

```
import java.util.*;

import java.lang.*;

import java.io.*;

class MST{

int minKey(int key[], Boolean mstSet[],int V){

int min = Integer.MAX_VALUE, min_index=-1;

for (int v = 0; v < V; v++)

if (mstSet[v] == false && key[v] < min){

 min = key[v];

 min_index = v;

}

return min_index;

}

void printMST(int parent[], int V, int graph[][]){
```

```
System.out.println("Edge Weight");
for (int i = 1; i < V; i++)
System.out.println(parent[i]+" - "+ i+" "+graph[i][parent[i]]);
}
void primMST(int graph[][],int V){
int parent[] = new int[V];
int key[] = new int [V];
Boolean mstSet[] = new Boolean[V];
for (int i = 0; i < V; i++){
key[i] = Integer.MAX_VALUE;
mstSet[i] = false;
}
key[0] = 0;
parent[0] = -1;
for (int count = 0; count < V-1; count++){
int u = minKey(key, mstSet,V);
mstSet[u] = true;
for (int v = 0; v < V; v++)
if (graph[u][v]!=0 && mstSet[v] == false &&graph[u][v] < key[v]){
parent[v] = u;
key[v] = graph[u][v];
}
}
printMST(parent, V, graph);
}
public static void main (String[] args){
MST t = new MST();
Scanner s=new Scanner(System.in);
```

```
System.out.println("Enter the Graph Size:");
int n=s.nextInt();
int graph[][] = new int[n][n];
System.out.println("Enter the value of vertices:");
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
graph[i][j]=s.nextInt();
t.primMST(graph,n);
}
}
```

OUTPUT:-

Enter the Graph Size:

3

Enter the value of vertices:

0 2 0

2 0 3

0 3 0

Edge Weight

0 - 1 2

1 - 2 3

C:\Users\TAUSIF\OneDrive\Desktop\JAVA PROGRAM>

EXPERIMENT NO: 08

NAME OF THE EXPERIMENT: All Pairs Shortest Path

AIM: Implement Floyd's algorithm for the all pairs shortest path problem.

THEORY: We initialize the solution matrix same as the input graph matrix as a first step.

Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

SOURCE CODE:

```
import java.util.*;

import java.lang.*;

import java.io.*;

class AllPairShortestPath{

final static int INF = 99999;

void floydWarshall(int graph[][],int V){

int dist[][] = new int[V][V]; int i, j, k;

for (i = 0; i < V; i++)

for (j = 0; j < V; j++)

dist[i][j] = graph[i][j];

for (k = 0; k < V; k++){

for (i = 0; i < V; i++){

for (j = 0; j < V; j++){

if (dist[i][k] + dist[k][j] < dist[i][j])

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}
```

```
printSolution(dist,V);
}
void printSolution(int dist[][],int V){
System.out.println("Following matrix shows the shortest "+"distances between every pair of vertices");
for (int i=0; i<V; ++i){
for (int j=0; j<V; ++j){
if (dist[i][j]==INF)
System.out.print("INF ");
else
System.out.print(dist[i][j]+" ");
}
System.out.println();
}
}
public static void main (String[] args){
Scanner s=new Scanner(System.in);
System.out.println("Enter the size of Graph :");
int n=s.nextInt();
int graph[][]=new int [n][n];
System.out.println("Enter the value of vertices:");
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
graph[i][j]=s.nextInt();
/*int graph[][] = { {0, 5, INF, 10},
{INF, 0, 3, INF},
{INF, INF, 0, 1},
{INF, INF, INF, 0}
};*/
```

```
AllPairShortestPath a = new AllPairShortestPath();
a.floydWarshall(graph,n);
}
}
```

OUTPUT:-

Enter the size of Graph :

3

Enter the value of vertices:

0 5 99999

99999 0 3

99999 99999 0

Following matrix shows the shortest distances between every pair of vertices

0 5 8

INF 0 3

INF INF 0

C:\Users\TAUSIF\OneDrive\Desktop\JAVA PROGRAM\20BTCS047HY>

EXPERIMENT NO: 09

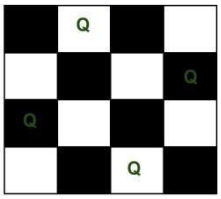
NAME OF THE EXPERIMENT: N-queens Problem using Backtracking**.**

AIM: Implement backtracking algorithm for the N-queens Problem.

THEORY: Backtracking | Set 3 (N Queen Problem).

We have discussed Knight's tour and Rat in Maze problems in Set 1 and Set 2 respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen Problem.



SOURCE CODE:

```
package backtracking;

import java.util.*;

public class NQueens2DArray {

public static void main(String[] args) {

Scanner s=new Scanner(System.in);

System.out.println("Enter the size of 2D Array:");

int size=s.nextInt();

placeQueens(size);

}

private static void placeQueens(int gridSize){

if(gridSize<4)

System.out.println("No Solution available");

else{

int[][] board = new int[gridSize][gridSize];
```

```
placeAllQueens(board, 0);
printBoard(board);
}
}
private static boolean placeAllQueens(int board[][], int row){
if(row>=board.length)
return true;
boolean isAllQueensPlaced = false;
for (int j = 0; j < board.length; j++) {
if(isSafe(board, row, j)){
board[row][j] = 1;
isAllQueensPlaced = placeAllQueens(board, row+1);
}
if(isAllQueensPlaced)
break;
else
board[row][j] = 0;
}
return isAllQueensPlaced;
}
private static boolean isSafe(int board[][], int row, int col){
for (int i = row-1, j = col-1; i >= 0 && j >= 0; i--, j--) {
if(board[i][j] == 1)
return false;
}
for (int i = row-1, j = col+1; i >= 0 && j < board.length; i--, j++) {
if(board[i][j] == 1)
return false;
```

```
}
for (int i = row-1; i >= 0; i--) {
if(board[i][col] == 1)
return false;
}
return true;
}
private static void printBoard(int[][] board){
for (int row = 0; row < board.length; row++) {
 for (int col = 0; col < board.length; col++) {
 if(board[row][col] == 1)
System.out.print("Q ");
else
System.out.print("_ ");
}
System.out.println();
}
System.out.println("");
}
}
```

OUTPUT:-

Enter the size of 2D Array:

4

_ Q _ _

_ _ _ Q

Q _ _ _

_ _ Q _

EXPERIMENT NO: 10

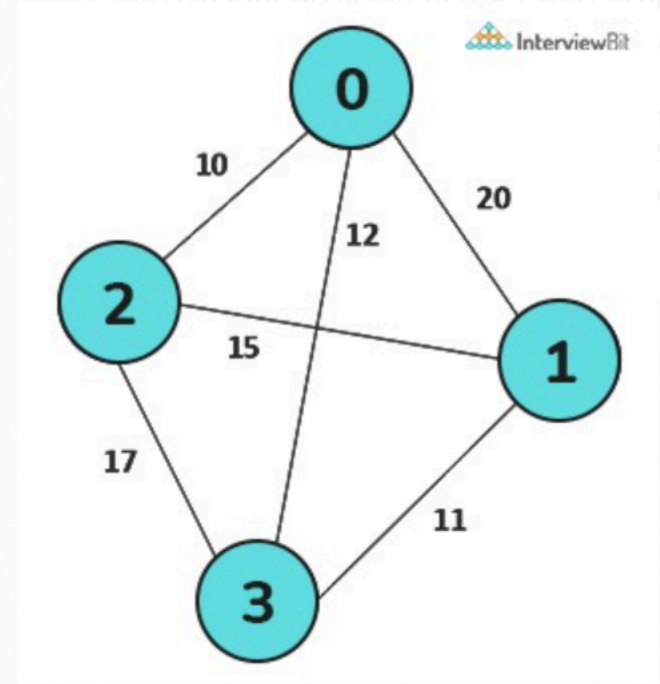
NAME OF THE EXPERIMENT: Travelling Salesperson Problem**.**

AIM: Implement the travelling salesperson problem (TSP)using dynamic programming.

THEORY:

In travelling salesman problem algorithm, we take a subset N of the required cities that need to be visited, the distance among the cities dist, and starting city s as inputs. Each city is identified by a unique city id which we say like 1,2,3,4,5……n.

There are at most O(n2^n) subproblems, and each one takes linear time to solve. The total running time is, therefore, O(n^22^n). The time complexity is much less than O(n!) but still exponential. The space required is also exponential.



Source code:-

```
import java.util.*;

class TSPbyDP{

static int travllingSalesmanProblem(int graph[][],int s,int V){

   ArrayList<Integer> vertex = new ArrayList<Integer>();
```

```
    for (int i = 0; i < V; i++)
            if (i != s)
               vertex.add(i);
    int min_path = Integer.MAX_VALUE;
    do{
            int current_pathweight = 0;
            int k = s;
          for (int i = 0; i < vertex.size(); i++){
               current_pathweight += graph[k][vertex.get(i)];
               k = vertex.get(i);
            }
            current_pathweight += graph[k][s];
            min_path = Math.min(min_path, current_pathweight);
          } while (findNextPermutation(vertex));
    return min_path;
}
public static ArrayList<Integer> swap(ArrayList<Integer> data,int left, int right){
    int temp = data.get(left);
    data.set(left, data.get(right));
    data.set(right, temp);
    return data;
}
public static ArrayList<Integer> reverse(ArrayList<Integer> data, int left, int right){
    while (left < right){
            int temp = data.get(left);
            data.set(left++, data.get(right));
            data.set(right--, temp);
    }
```

```
    return data;
}
public static boolean findNextPermutation(ArrayList<Integer> data){
    if (data.size() <= 1)
            return false;
    int last = data.size() - 2;
    while (last >= 0){
            if (data.get(last) < data.get(last + 1))
                break;
            last--;
    }
    if (last < 0)
            return false;
    int nextGreater = data.size() - 1;
    for (int i = data.size() - 1; i > last; i--) {
            if (data.get(i) > data.get(last)){
                nextGreater = i;
                break;
            }
    }
    data = swap(data,nextGreater, last);
    data = reverse(data, last + 1, data.size() - 1);
    return true;
}
public static void main(String args[]){
Scanner s=new Scanner(System.in);
System.out.println("Enter the size of Graph :");
int n=s.nextInt();
```

```
int graph[][]=new int[n][n];
System.out.println("Enter the value of vertices :");
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
graph[i][j]=s.nextInt();
  /* int graph[][] = {{0, 10, 15, 20},
                      {10, 0, 35, 25},
                        {15, 35, 0, 30},
                        {20, 25, 30, 0}};*/
   System.out.println("Enter the value of S :");
int S=s.nextInt();
   System.out.println(travllingSalesmanProblem(graph, S,n));
   }
}
```

OUTPUT:-

Enter the size of Graph :

3

Enter the value of vertices :

1 2 3 4 5 6 7 8 9

Enter the value of S :

2

15

C:\Users\TAUSIF\OneDrive\Desktop\JAVA PROGRAM\20BTCS047HY>