Algorithm :-

Input : A connected, weighted, undirected graph $G = (V, E)$ with weight function $w(u, v)$,

Output : A minimum spanning Tree (MST).

1. Initialize :
   - Select an arbitary vertex v_0 as the starting node.
   - Set $T = \{v_0\}$ (initial MST with one vertex).
   - Initialize a priority queue PQ with edges connected to v_0, prioritized by weight.
2. Repeat until all vertices are included in T (i.e., $|T| = |V|$):
   - Extract the edge (u, v) with the minimum weight from PQ.
   - If v is not already in T :
     - Add v to T.
     - Add all edges of v (not already in T) to PQ.
3. Output the edges of the minimum spanning Tree.

Time Complexity Analysis.

- Using an Adjacency matrix & simple search : $O(V^2)$

Date :

Exp. No. 05. a.

Exp. Title Implement Prim's algorithm and Analyze its time complexity.

Page No. 11

Program :

```
import java.util.Scanner;
public class primAlgorithm1 {
    static final int INF = 9999;
    static final int MAX = 20;
    static int[][] G = new int[MAX][MAX];
    static int[][] spanning = new int[MAX][MAX];
    static int n;

    public static void main(String[] args)
    {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices:");
        n = scanner.nextInt();
        System.out.println("\n Enter the adjacency :");
        for (int i=0; i<n; i++)
        {
            for (int j=0; j<n; j++)
            {
                G[i][j] = scanner.nextInt();
            }
        }
        int totalCost = prims();
        System.out.println("\n Spanning tree matrix:");
        for (int i=0; i<n; i++)
        {
            for (int j=0; j<n; j++)
            {
                System.out.print(spanning[i][j] + "\t");
            }
```

Date :
Exp. No.

Exp. Title


```
        System.out.println();
    }
    System.out.println("\nTotal cost of the spanning tree =" + totalCost);
}
Syste Static int prims(){
        int[][] cost = new int[MAX][MAX];
        int[] distance = new int[MAX];
        int[] from = new int[MAX];
        int[] visited = new int[MAX];
        int minCost = 0;
        for (int i=0; i<n; i++)
        {
          for (int j=0; j<n; j++)
          {
            if(a[i][j]==0)
            {
              Cost[i][j]=INF;
            }
            else{
              cost[i][j] = a[i][j];
            }
            spanning[i][j]=0;
          }
        }
        distance[0]=0;
        visited[0]=1;
        for (int i=1; i<n; i++)
        {
          distance[i] = cost[0][i];
          from[i]=0;
```

Output:

```
Enter the number of vertices : 3
Enter the adjacency matrix:
  0  1  1
  1  0  1
  1  1  0
Spanning tree matrix:
  0  1  1
  1  0  0
  1  0  0

Total cost of the spanning tree=2
```

Date : Exp. No.	Exp. Title	Page No. 13

```
        visited[i]=0;
    }
    int noOfEdges = n-1;
    while (noOfEdges >0)
    {
        int minDistance = INF, v= -1;
        for (int i=1; i<n; i++)
        {
            if (visited[i]==0 && distance[i] < minDistance)
            {
                v=i;
                minDistance = distance[i];
            }
        }
        int u= from[v];
        spanning[u][v] = distance[v];
        spanning[v][u]= distance[v];
        noOfEdges --;
        visited[v] =1;
        for (int i=1; i<n; i++)
        {
            if (visited[i]==0 && cost[i][v] < distance[i])
            {
                distance[i]= cost[i][v];
                from[i]=v;
            }
        }
        mincost += cost[u][v];
    }
    return mincost;
}
```

Algorithm :

Input : A weighted, directed graph $G = (V, E)$ with non-negative edge weight $w(u, v)$, and a source vertex s.

Output : The shortest path distances from s to all vertices in G.

1. Initialize :
   - Set the distance to the source : $d(s) = 0$
   - Set all other distances : $d(v) = \infty$ for $v \neq s$.
   - Create a priority queue (PQ) & insert (s, 0) (source with distance 0).

2. While the priority queue is not empty :
   - Extract the vertex u with the minimum distance from PQ.
   - For each neighbor v of u :
     - If $d(u) + w(u, v) < d(v)$:
       - update $d(v)$ to $d(u) + w(u, v)$.
       - Insert/update $(v, d(v))$ in PQ.

3. Output : The shortest path distances from s to all other vertices.

Time Complexity Analysis.

- Using an Adjacency matrix with simple search : $O(V^2)$

Date :	Exp. Title Dijkstra's algorithm and Analyze its Time complexity.	Page No.
Exp. No. 05 b.		14

Program :

```
Public class DijkstraAlgorithm
{
  Public void dijkstraAlgorithm (int[][] graph, int
source)
  {
    int nodes = graph.length;
    boolean[] visited_vertex = new boolean[nodes];
    int[] dist = new int[nodes];
    for (int i=0; i<nodes; i++)
    {
      visited_vertex[i] = false;
      dist[i] = Integer.MAX_VALUE;
    }
    dist[source] = 0;
    for(int i=0; i<nodes; i++)
    {
      int u = find_min_distance (dist, visited_vertex);
      visited_vertex[u] = true;
      for (int v=0; v< nodes; v++)
      {
        if (!visited_vertex[v] && graph[u][v]!
=0 && (dist[u] + graph[u][v] < dist[v]))
        {
          dist[v] = dist[u] + graph[u][v];
        }
      }
    }
    for (int i=0; i< dist.length; i++)
    {
```

```
System.out.Println(String.format("Distance from
vertex %S to vertex %S" is %S, source, i, dist[i]));
    }
  }
  Private static int find-min-distance(int[] dist,
boolean[] visited_vertex)
  {
    int minimum_distance = Integer.MAX_VALUE;
    int minimum_distance_vertex = -1;
    for (int i=0; i< dist.length; i++)
    {
      if (!visited_vertex[i] && dist[i] <minimum
-distance)
      {
        minimum_distance = dist[i];
        minimum_distance_vertex = i;
      }
    }
    return minimum_distance_vertex;
  }
  Public static void main(String[] args)
  {
    int graph[][] = new int[][]
    {
      {0, 1, 1, 2, 0, 0, 0},
      {0, 0, 2, 0, 0, 3, 0},
      {1, 2, 0, 1, 3, 0, 0},
      {2, 0, 1, 0, 2, 0, 1},
      {0, 0, 3, 0, 0, 2, 0},
      {0, 3, 0, 0, 2, 0, 1},
      {0, 2, 0, 1, 0, 1, 0},
```

Output :

Distance from vertex 0 to vertex 0 is 0
Distance from vertex 0 to vertex 1 is 1
Distance from vertex 0 to vertex 2 is 1
Distance from vertex 0 to vertex 3 is 2
Distance from vertex 0 to vertex 4 is 4
Distance from vertex 0 to vertex 5 is 4
Distance from vertex 0 to vertex 6 is 3.

Date :	Exp. Title	Page No.
Exp. No.		16

```
};
DijkstraAlgorithm test = new DijkstraAlgorithm();
test.dijkstraAlgorithm(graph, 0);
}
}
```

Algorithm : Warshall's Algorithm.

Input : A directed graph represented as an adjacency matrix R(0), where :

- R(0)[i][j] = 1 if there is a direct edge from i to j, else 0.

Output : The transitive closure matrix R(n), where R(n)[i][j] = 1 if there exists a path from i to j, otherwise 0.

Steps :

1. Initialize R(0) as the adjacency matrix
2. Iterate over all vertices k as intermediate nodes :
   - For each pair of vertices (i, j) :
     - Update the matrix :

       R(k)[i][j] = R(k-1)[i][j] OR (R(k-1)[i][k] AND R[k-1][k][j])
3. Final matrix R(n) gives the transitive close of the graph.

Time Complexity of Warshall's Algorithm.

- The algorithm runs three nested loops over V vertices $O(V^3)$.

Date :	Exp. Title Implement Warshall and Floyd's algorithm.	Page No. 17
Exp. No. 06.a&b.		

Program :

```
import java.lang.*;
Public class AllPairsShortestPath
{
final static int INF = 99999, V=4;
void floydwarshall (int dist[][])
{
int i, j, k;
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);
}
void printSolution (int dist[][])
{
System.out.println("The following matrix shows
the shortest" + "distance btw 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");
```

Algorithm : Floyd - Warshall Algorithm.

Input: A weighted graph represented as a distance matrix D(0), where:

- D(0)[i][j] is the weight of edge from i to j (or ∞ if no direct edge exists).
- D(0)[i][i] = 0 for all i.

Output: A matrix D(n) where D(n)[i][j] gives the shortest distance between i and j.

Steps:

1. Initialize D(0) as the weight matrix.
2. Iterate over all vertices k as intermediate nodes:
   - For each pair (i, j):
   - Update the matrix using:

$$D(k)[i][j] = \min(D(k-1)[i][j], D(k-1)[i][k] + D(k-1)[k][j])$$

3. Final matrix D(n) gives shortest paths between all pairs.

Time Complexity of Floyd - Warshall Algorithm

- It has three nested loops, iterating over all pairs of vertices → $O(V^3)$.

Output: The following matrix shows the shortest distance blw every pair of vertices.

0	5	8	9
INF	0	3	4
INF	INF	0	1
INF	INF	INF	0

```
        else
            System.out.Print(dist[i][j]+"   ");
        }
        System.out.println();
    }
}
Public static void main (String[] args)
{
    int graph[][] = {{0, 5, INF, 10},
                     {INF, 0, 3, INF},
                     {INF, INF, 0, 1},
                     {INF, INF, INF, 0}};
    AllPairsShortestPath a = new AllPairsShortestPath();
}
}
```

Algorithm :- Backtracking Approach for Hamiltonian cycle.

This algorithm checks whether a Hamiltonian cycle exists in a given undirected graph using backtracking.

Input:
- A graph $G = (V, E)$ represented by an adjacency matrix adj[][].
- A starting vertex V_0.

Output:
- A Hamiltonian cycle (if one exists) or a message indicating that no cycle exists.

Steps.
1. Initialize an array Path[] to store the cycle, setting path[0] = V-0.
2. Recursively try to construct the cycle by adding one vertex at a time:
   - Choose the next vertex V such that:
     - It is adjacent to the previous vertex in Path[].
     - It has not been visited before.
   - If a valid vertex is found, mark it as visited and move to the next position.
3. If all vertices are in Path[] and the last vertex connects back to V-0, return success.
4. Backtrack if no valid vertex can be found at a position.
5. If all possibilities are exhausted, return that no Hamiltonian cycle exists.

Date :	Exp. Title Implement Hamiltonian cycle -s algorithm.	Page No.
Exp. No. 07		19

Program.

```
Public class HamiltonianCycle
{
    final int V=5;
    int Path[];
    {
        for (int i=0; i<pos; i++)
            if (path[i] == v)
                return false;
        return true;
    }
boolean hamCycleUtil (int graph[][], int path[],
                      int pos)
{
    if (pos == V)
    {
        if (graph[Path[pos-1]][Path[0]] == 1)
            return true;
        else
            return false;
    }
    for (int v=1; v<V; v++)
    {
        if (isSafe (v, graph, Path, pos))
        {
            Path[pos] = v;
            if (hamCycleUtil (graph, Path, pos+1) == true)
                return true;
            path[pos] = -1;
        }
```

Time Complexity Analysis.

- The algorithm explores all possible permutations of vertices to form a cycle
- Since each vertex can be chosen in at most V! ways, the worst-case time complexity is O(V!) (Factorial time complexity).
- In practice, pruning techniques (backtracking and branch-and-bound) reduce the number of cases checked, but the problem remain exponential in the worst case.

```
    }
    return false;
}
int hamCycle(int graph[][])
{
    Path = new int[V];
    for (int i=0; i<V; i++)
        Path[i] = -1;
    Path[0] = 0;
    if (hamCycleUtil(graph, Path, 1) == false)
    {
        System.out.println("\n Solution does not
Exist");
        return 0;
    }
    PrintSolution(Path);
    return 1;
}
void PrintSolution(int Path[])
{
    System.out.println("Solution exists: Following"+
    "is one Hamiltonian Cycle");
    for (int i=0; i<V; i++)
        System.out.print(" " + Path[i] + " ");
        System.out.println(" " + Path[0] + " ");
}
Public static void main(String args[])
{
    HamiltonianCycle hamiltonian = new HamiltonianCycle
                                    ();
```

Output:

Solution Exists: Following is one Hamiltonian Cycle

0 1 2 4 3 0

For Graph 2

Solution does not exist:

```
int graph1[][] = {{ 0, 1, 0, 1, 0},
                  { 1, 0, 1, 1, 1},
                  { 0, 1, 0, 0, 1},
                  { 1, 1, 0, 0, 1},
                  { 0, 1, 1, 1, 0},
                  };
hamiltonian.hamCycle(graph1);
System.out.println("For Graph 2:");
int graph2[][] = {{ 0, 1, 0, 1, 0},
                  { 1, 0, 1, 1, 1},
                  { 0, 1, 0, 0, 1},
                  { 1, 1, 0, 0, 0},
                  { 0, 1, 1, 0, 0},
                  };
hamiltonian.hamCycle(graph2);
}
}
```

Algorithm:

Steps:

1. Compute GCD(a,b) using Euclidean Algorithm.
2. Compute LCM(a,b) using the formula:

$$LCM(a,b) = \frac{|a \times b|}{GCD(a,b)}$$

Time Complexity Analysis.

- GCD calculation (Euclidean Algorithm):
  - Runs in $O(\log \min(a,b))$ time.
- LCM calculation:
  - Multiplication takes $O(1)$.
  - Division takes $O(1)$.
- Overall time complexity:

$$O(\log \min(a,b))$$

Output:

LCM of the array is: 300.

Date :	Exp. Title Implement lcm algorithm.	Page No. 22.
Exp. No. 08.		

Program.

```
Public class LCMCalculator
{
  Private static int gcd (int a, int b)
  {
     if(b==0)
        return a;
     return gcd (b, a%b);
  }
  Private static int lcm (int a, int b)
  {
     return (a*b) / gcd (a,b);
  }
  Public static int lcmArray (int[] arr) {
     int result = arr[0];
     for (int i=1; i< arr.length ; i++)
     {
        result = lcm (result, arr[i]);
     }
     return result;
  }
  Public static void main (String[] args)
  {
     int[] numbers = { 12, 15, 20, 25 };
     int result = lcmArray (numbers);
     System.out.println ("lcm of the array is : " + result);
  }
}.
```