**Analysis of Algorithm**

**Practical no 9**

**Floyd-Warshall Algorithm**

**Code :**

public class FloydWarshalls {

    static void floydWarshall(int[][] graph) {

        int V = graph.length;

        int k,i,j;

        if(V < 2)

            return;

        // Add all vertices one by one to

        // the set of intermediate vertices.

        for (k = 0; k < V; k++) {

            // Pick all vertices as source one by one

            for (i = 0; i < V; i++) {

                // Pick all vertices as destination

                // for the above picked source

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

                    // If vertex k is on the shortest path from

                    // i to j, then update the value of graph[i][j]

                    if ((graph[i][j] == -1 || graph[i][j] > (graph[i][k] + graph[k][j])) && (graph[k][j] != -1 && graph[i][k] != -1))

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

                }

            }

            System.out.print("\n Iteration "+ k + " : \n");

            for (int a = 0; a < graph.length; a++) {

                for (int b = 0; b < graph.length; b++) {

                    System.out.print(graph[a][b] + " ");

                }

                System.out.println();

            }

        }

    }

    public static void main(String[] args) {

        int[][] graph = {

            {0,4,11},

            {6,0,2},

            {3, Integer.MAX\_VALUE,0}

        };

        floydWarshall(graph);

        System.out.println("\n Final Output : ");

        for (int i = 0; i < graph.length; i++) {

            for (int j = 0; j < graph.length; j++) {

                System.out.print(graph[i][j] + " ");

            }

            System.out.println();

        }

    }

}

**Output :**

A screenshot of a computer

AI-generated content may be incorrect.

**Analysis :**

The **Floyd-Warshall algorithm** is a well-known algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. In this case, you provided a Java implementation of the algorithm that works with a graph where:

* graph[i][j] represents the weight of the edge from vertex i to vertex j.
* If there is no direct edge between i and j, the value is represented as -1 (which is treated as infinity in the algorithm).
* The algorithm assumes that the graph can have at most V vertices, where V is the size of the graph (i.e., graph.length).

 **Input Graph**: The graph is represented as a 2D array where:

* graph[i][j] gives the weight of the edge from vertex i to vertex j.
* If there is no edge between i and j, we store -1 to represent "infinity" (or the absence of an edge).

 **Algorithm Logic**:

* We iterate through all possible intermediate vertices k, and for each pair of source (i) and destination (j), we check if the path through vertex k is shorter than the current known path from i to j.
* If the path through k is shorter, we update graph[i][j].

 **Condition Check**:

* graph[i][j] == -1 checks if there's no existing path between i and j (in which case it's treated as infinity).
* graph[i][j] > (graph[i][k] + graph[k][j]) checks if the path through vertex k is shorter than the current known distance.
* (graph[k][j] != -1 && graph[i][k] != -1) ensures that both paths (from i to k and from k to j) are valid (i.e., not -1).

 **Graph Update**:

* If the condition holds true, we update graph[i][j] to the shorter path found via k.

 **Time Complexity**: The algorithm always runs in **O(V³)** time because of the three nested loops, each running V times.

 **Space Complexity**: The space complexity is **O(V²)** due to the storage required for the graph.