// A Java program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

**import** java.util.\*;

**import** java.lang.\*;

**import** java.io.\*;

**class** Dijkstra

{

// A utility function to find the vertex with minimum distance value,

// from the set of vertices not yet included in shortest path tree

**public** **static** **final** **int** ***V***=5;

**public** **int** minDistance(**int** dist[], Boolean sptSet[])

{

// Initialize min value

**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;

}

// A utility function to print the constructed distance array

**public** **void** printSolution(**int** dist[], **int** n)

{

System.***out***.println("Vertex Distance from Source");

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

System.***out***.println("Node is "+i+" distance is "+dist[i]);

}

// Funtion that implements Dijkstra's single source shortest path

// algorithm for a graph represented using adjacency matrix

// representation

**public** **void** dijkstra\_algo(**int** graph[][], **int** src)

{

**int** dist[] = **new** **int**[***V***]; // The output array. dist[i] will hold

// the shortest distance from src to i

// sptSet[i] will true if vertex i is included in shortest

// path tree or shortest distance from src to i is finalized

Boolean sptSet[] = **new** Boolean[***V***];

// Initialize all distances as INFINITE and stpSet[] as false

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

{

dist[i] = Integer.***MAX\_VALUE***;

sptSet[i] = **false**;

}

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

**for** (**int** count = 0; count < ***V***-1; count++)

{

// Choose the minimum distance vertex from the set of vertices

// not yet processed. u is always equal to src in first

// iteration.

**int** u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = **true**;

// Update dist value of the adjacent vertices of the

// picked vertex.

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

// Update dist[v] only if is not in sptSet, there is an

// edge from u to v, and total weight of path from src to

// v through u is smaller than current value of dist[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];

}

// print the constructed distance array

printSolution(dist, ***V***);

}

// Driver method

**public** **static** **void** main(String[] args)

{

/\* Let us create the example graph discussed above \*/

**int** graph[][] = {{0, 10, 5, 0, 0},{0, 0, 2, 1, 0},{0, 3, 0, 9, 2},{0, 0, 0, 0, 4},{7, 0, 0, 6, 0}};

System.***out***.println("Starting with S");

Dijkstra q = **new** Dijkstra();

q.dijkstra\_algo(graph, 0);

System.***out***.println();

System.***out***.println("Starting with Z");

Dijkstra w = **new** Dijkstra();

w.dijkstra\_algo(graph, 4);

}

}