```
Dijkstra(graph, start):
    // Create a set of unvisited vertices
    unvisitedVertices = set of all vertices in the graph

    // Create a dictionary to store the shortest distances from the start vertex
    distances = {}
    for each vertex in graph:
        distances[vertex] = infinity
    distances[start] = 0

    while unvisitedVertices is not empty:
        // Find the vertex with the smallest distance
        currentVertex = vertex in unvisitedVertices with the smallest distance

        // Remove currentVertex from unvisitedVertices
        remove currentVertex from unvisitedVertices

        // Update distances to neighboring vertices through currentVertex
        for each neighbor of currentVertex:
            // Calculate the tentative distance
            tentativeDistance = distances[currentVertex] + distance between currentVertex and neighbor

            // If the tentative distance is less than the current distance, update it
            if tentativeDistance < distances[neighbor]:
                distances[neighbor] = tentativeDistance

    return distances
```

BellmanFord:

```
BellmanFord(graph, start):
    // Create a list to store the shortest distances from the start vertex
    distances = []
    for each vertex in graph:
        distances[vertex] = infinity
    distances[start] = 0

    // Relax edges repeatedly (V-1) times
    for i from 1 to |V| - 1:
        for each edge (u, v) in graph:
            if distances[u] + weight(u, v) < distances[v]:
                distances[v] = distances[u] + weight(u, v)

    // Check for negative weight cycles
    for each edge (u, v) in graph:
        if distances[u] + weight(u, v) < distances[v]:
            // A negative weight cycle exists in the graph

    return distances
```

FloyedWarshal:

```
FloydWarshall(graph):
    // Initialize the distance matrix with infinity for all pairs of vertices
    n = number of vertices in graph
    distances = create a 2D array of size n x n
    for each vertex u in graph:
        for each vertex v in graph:
            if u == v:
                distances[u][v] = 0
            else if edge (u, v) exists:
                distances[u][v] = weight(u, v)
            else:
                distances[u][v] = infinity

    // Calculate shortest paths
    for each vertex k in graph:
        for each vertex i in graph:
            for each vertex j in graph:
                if distances[i][k] + distances[k][j] < distances[i][j]:
                    distances[i][j] = distances[i][k] + distances[k][j]

    return distances
```