```cpp
    /// <summary>
    /// Bellman-Ford algorithm to find the shortest path from a source vertex to
    /// all other vertices in the graph and detect negative weight cycles that
    /// are reachable from the source
    /// </summary>
    /// <param name="s">the source vertex</param>
    /// <returns>Tuple containing 3 elements:
    ///             - a vector of distances from the source to each vertex
    ///             - a vector of predecessors for each vertex which helps to recreate the path
    ///             - a boolean indicating whether a negative cycle was detected
    /// </returns>
    std::tuple<std::vector<int>, std::vector<int>, bool> bellman_ford(int s)
    {
        // vector to hold the shortest distances from the source to all vertices
        // vertices are initialized with infinity(2e9) except for the source vertex
        // which is initialized with a distance of 0
        std::vector<int> distance(get_nr_of_vertices(), 2e9);
        distance[s] = 0;

        // vector to hold the predecessor for each vertex which helps us to
        // recreate the shortest path, beacuse it tells us the previous vertex
        // of current one
        // It is initilized with -1 because, initially, we don't know the predecessor
        std::vector<int> predecessor(get_nr_of_vertices(), -1);

```

```cpp
        // Relaxing the edges
        // Iterate V - 1 times where V is the number of vertices
        // Since the shortest path in a graph with V vertices can have at most V - 1 edges
        for (int k = 1; k <= get_nr_of_vertices() - 1; ++k)
        {
            for (auto p : get_edges())
                // Check every edge in the graph to see if the distance to the vertex
                // at the end of the edge can be improved by going through the vertex
                // at the start of the edge
                if (distance[p.second] > distance[p.first] + get_cost(p.first, p.second))
                {
                    distance[p.second] = distance[p.first] + get_cost(p.first, p.second);
                    predecessor[p.second] = p.first;
                }
        }

        // Check for negative weight cycles
        // One more pass over all the edges to check for negative weight cycles
        for (auto p : get_edges())
            // If a negative cycle is detected, return True indicating the cycle's presence
            if (distance[p.second] > distance[p.first] + get_cost(p.first, p.second))
                return std::make_tuple(distance, predecessor, true);

        // Return the distances, the predecessor for each vertex and False
        // because there is no cycle
        return std::make_tuple(distance, predecessor, false);
    }
```