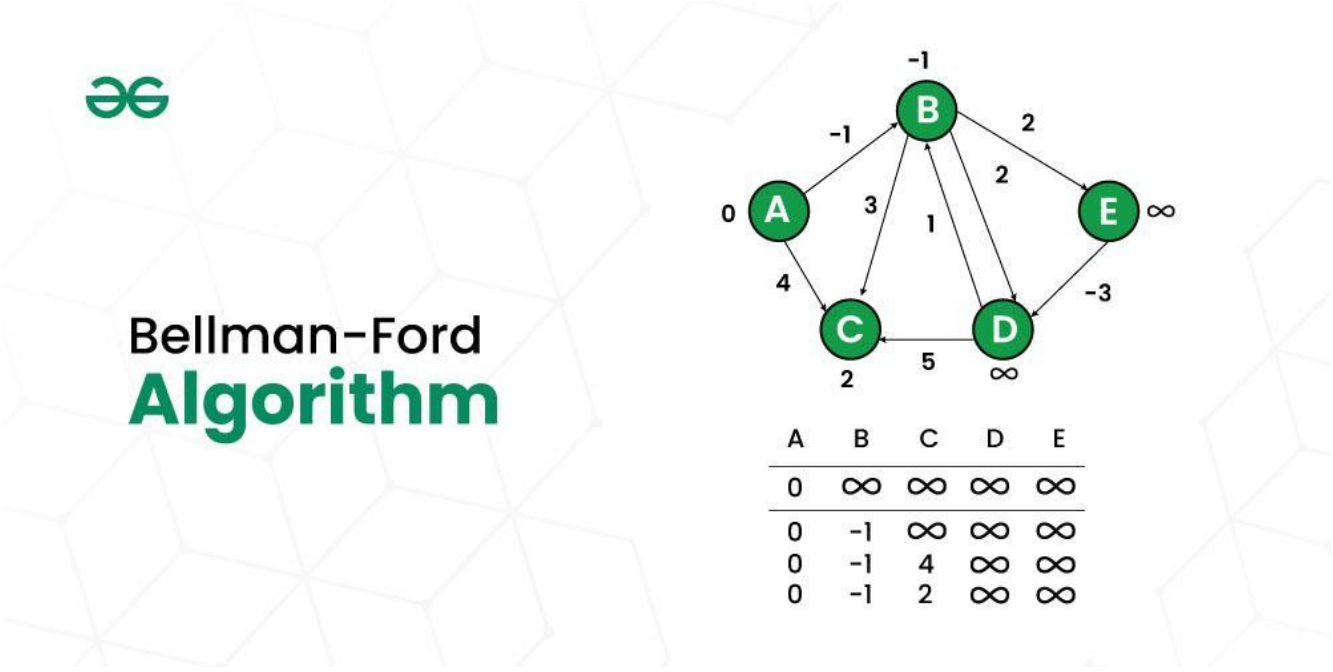
Bellman-Ford is a **single source** shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph. This algorithm can be used on both **weighted** and **unweighted** graphs.



Algorithm to Find Negative Cycle in a Directed Weighted Graph Using Bellman-Ford:

- Initialize distance array dist[] for each vertex '**v**' as **dist[v] = INFINITY**.
- Assume any vertex (let's say '0') as source and assign **dist = 0**.
- Relax all the **edges(u,v,weight) N-1** times as per the below condition:
    - **dist[v] = minimum(dist[v], distance[u] + weight)**
- Now, Relax all the edges one more time i.e. the **Nth** time and based on the below two cases we can detect the negative cycle:
    - Case 1 (Negative cycle exists): For any **edge(u, v, weight), if dist[u] + weight < dist[v]**
    - Case 2 (No Negative cycle) : case 1 fails for all the edges.

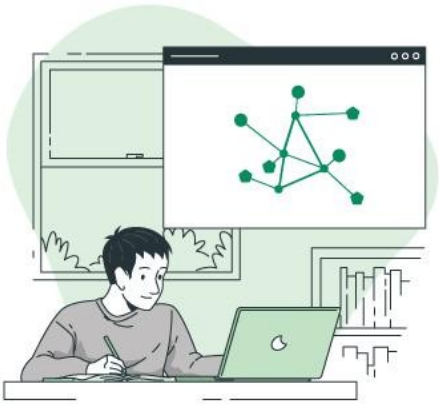
Handling Disconnected Graphs in the Algorithm:

The above algorithm and program might not work if the given graph is disconnected. It works when all vertices are reachable from source vertex **0**.
To handle disconnected graphs, we can repeat the above algorithm for vertices having **distance = INFINITY,** or simply for the vertices that are not visited.

- **Time Complexity when graph is connected:**
- **Best Case:** O(E), when distance array after 1st and 2nd relaxation are same , we can simply stop further processing
    - **Average Case:** O(V*E)

- **Worst Case:** O(V*E)

Dijkstra Algorithm



The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

function Dijkstra(Graph, source):

```
  // Initialize distances to all nodes as infinity, and to the source node as 0.
distances = map(all nodes -> infinity)
distances = 0
  // Initialize an empty set of visited nodes and a priority queue to keep track of the nodes to visit.
  visited = empty set
  queue = new PriorityQueue()
  queue.enqueue(source, 0)
  // Loop until all nodes have been visited.
  while queue is not empty:
    // Dequeue the node with the smallest distance from the priority queue.
    current = queue.dequeue()
    // If the node has already been visited, skip it.
    if current in visited:
      continue
    // Mark the node as visited.
    visited.add(current)
```

```
    // Check all neighboring nodes to see if their distances need to be updated.
    for neighbor in Graph.neighbors(current):
        // Calculate the tentative distance to the neighbor through the current node.
        tentative_distance = distances[current] + Graph.distance(current, neighbor)

        // If the tentative distance is smaller than the current distance to the neighbor, update the distance.
        if tentative_distance < distances[neighbor]:
            distances[neighbor] = tentative_distance

            // Enqueue the neighbor with its new distance to be considered for visitation in the future.
            queue.enqueue(neighbor, distances[neighbor])

    // Return the calculated distances from the source to all other nodes in the graph.
    return distances
```

Algorithm for Dijkstra's Algorithm:

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

Complexity Analysis of Dijkstra Algorithm:

- **Time complexity:** O((V + E) log V)**,** where V is the number of vertices and E is the number of edges.
- **Auxiliary Space:** O(V), where V is the number of vertices and E is the number of edges in the graph.

Kruskal's Minimum Spanning Tree Algorithm:

This is one of the popular algorithms for finding the minimum spanning tree from a connected, undirected graph. This is a greedy algorithm. The algorithm workflow is as below:

- First, it sorts all the edges of the graph by their weights,
- Then starts the iterations of finding the spanning tree.
- At each iteration, the algorithm adds the next lowest-weight edge one by one, such that the edges picked until now does not form a cycle.

Prim's Minimum Spanning Tree Algorithm:

This is also a greedy algorithm. This algorithm has the following workflow:

- It starts by selecting an arbitrary vertex and then adding it to the MST.
- Then, it repeatedly checks for the minimum edge weight that connects one vertex of MST to another vertex that is not yet in the MST.
- This process is continued until all the vertices are included in the MST.

Complexity O(E logV)