Q6. Write the algorithm to find the longest path in a Graph and Estimate the complexity of the algorithm.

We assume that the given graph is a weighted directed acyclic graph with no multiple edges and self loops. Also, we are assuming that the weight of every edge of the graph is non-negative.

Since, a cyclic graph will have an "infinite longest path" as we can go through the cycle any number of times. Also, for the same reason, a graph with self loops will suffer from the same problem.

The main steps in this algorithm are as follows.

1. Find the topological order of the vertices.
2. Process the vertices in topological order and find the maximum cost to go to next level in topological order.
3. At the last level of topological ordering, the maximum cost will represent the maximum cost among any path.

A similar procedure can be used for unweighted directed graph. In such case, we can just assume that the cost of each edge is 1.

The detailed algorithm to find the longest path in a weighted directed acyclic graph is as follows. We are using adjacency list representation of the graph here.

```
# adjList: adjacency list of the given graph. each cell of the list has 3 values.
u, v and w which represents a edge from u to v with weight w

# n: number of nodes in the graph

# DFS function
FUNCTION DFS(node, visited, st):
    visited[node] = True
    FOR neighbour in adjList[node]:
        IF visited[neighbour] == False:
            DFS(neighbour, visited, st)
    st.push(node)

# Initialize
st = Create new Stack
visited = Create array of size n
FOR i = 1 to n:
    visited[i] = False

# Apply DFS
FOR i = 1 to n:
    IF visited[i] == False:
        DFS(i, visited, st)

# Store the topological order into an array
topOrder = Create array of size n
FOR i = 0 to n:
    topOrder[i] = st.pop()
```

The time complexity of topological sorting is O(V + E).

Now, after the topological sorting, the vertices are in required order. Now, we will process each node in this order and will find the max cost to reach to the next level of topological order from the set of nodes in the current level of topological order. We will maintain a "distance" array. Its ith cell will represent the max distance/cost to reach to the ith node of the graph from any of the nodes that come before ith node in topological order. Also another array called "parent" will be used. Its ith cell will represent the parent of the ith node. It will help in finding the actual path of the longest/maximum cost path.

Also, two extra variable will be maintained. One called "maxCost" to store the max cost to reach to any vertex and other called "maxCostNode" which will keep track of that maximum cost node.

Its pseudo code is as follows.

```
# adjList: adjacency list of the given graph. each cell of the list has 3 values.
u, v and w which represents a edge from u to v with weight w
# n: number of nodes in the graph
# distance: an array to store the distance to reach to all nodes
# parent: an array to store the parent nodes of all nodes
# maxCost: a variable to store the max cost/distance
# maxCostNode: a variable to store the node with max cost/distance

# initialize
FOR i = 1 to n:
    distance[i] = INF

distance[1] = 0
maxCost = 0
maxCostNode = -1
parent[1] = 1

# find the longest path
FOR i = 1 to n:
    FOR to, weight in adjList[i]:
        IF distance[to] < distance[i] + weight:
            distance[to] = distance[i] + weight
                parent[to] = i
                IF maxCost < distance[to]:
                    maxCost = max(maxCost, distance[to])
                    maxCostNode = to

# print the longest path in reverse order
node = maxCostNode
while parent[node] != node:
    PRINT node
    node = parent[node]
```

The above code prints the longest path in reverse order.

The time complexity of above procedure is O(V + E) as algorithm processes every node of the graph and then finds all of the adjacent nodes as well which is O(E) in worst case.

Hence, the overall time complexity is O(V + E).