Assignment 16

Alex Clemmer
Student number: u0458675

1

This algorithm has 3 basic steps: (1) linearize the DAG; (2) use modified Dijkstra's to find the "cost" of each node from the roots down.

The first step, linearizing the DAG is extremely straightforward. This algorithm is outlined pretty directly in Leiserson, Cormen, *et al* text, but with a few simple modifications, we can adjust it for some specific purposes we have:

```
S = list of sorted nodes
N = list of nodes with no incoming edges.

foreach node n in N:
    visit(n)

def visit(Node n):
    if n not visited:
        mark n as visited
        foreach node n1 in n's child nodes:
            visit(n1)
        add n to N
```

For an adjacency matrix, this will obviously run in $O(|V|^2)$, because there are $|V|^2$ total nodes. The adjacency list is roughly linear or $O(|V| + |E|)$, which ever is longer. This gives us the basis of representation for the tree that will make up our course list.

From here, we only need to find the cost of every node. Since cost is defined as the longest possible path from a node to a root, we need only do a depth-first search. The one major sticking point here is that when branches collide, the cost of the node becomes the larger of the costs, since the larger of the costs would be the minimum required prerequisite path.

```
S = list of sorted nodes

visit(S[0])

def visit(n, cost):
    if (n.visited == true) and n.cost < cost:
        n.cost = cost
    foreach child node t in n:
        if t.visited == false: visit(t, cost + 1)
```

The cost of this algorithm is $O(n)$, because we visit only those nodes that have not been visited. This adds up to a total of $O(|V|^2)$ for the matrix, because the polynomial dominates, or $O(|V| + |E|)$ for the list. Note that it doesn't matter in what order we encounter the nodes, because we will only update the `cost` in the event that the current cost is greater than the given one.