a.

The Algorithm

This is just the algorithm taught in class for minimum path cost between two nodes in a graph without negative cycles. The reason we imposed that we use this algorithm on a graph without negative cycles is that the minimum cost then might be negative infinity because you could just loop over the negative cycle infinitely many times to get decreasing costs each time, there's no limit to the edge count. However, putting an edge limit on it would still make it work – it just gets the minimum cost within the allotted k edges. The algorithm stays the same, just takes in an extra parameter k and runs for k iterations instead of the original n-1 iterations:

```
Shortest-Path(G,s,t,k)
    Array M[0 ... k, V]
    Define M[0,t] = 0 and M[0,v] = ∞ for all over v ∈ V
    For i = 1, ..., k
        For v ∈ V in any order
            node w = the node that minimizes M[i-1,w] + edge_weight(v,w)
            M[i,v] = min(M[i-1,v], M[i-1,w] + edge_weight(v,w))
        Endfor
    Endfor
    Return M[k,s]
```

Runtime Analysis

Iterating through M is O(kn) because there's k rows for the k edges we're allowed to traverse and n columns for the n nodes. Updating each cell in M is O(n) because there's at most n nodes $w \in V$ we have to consider. So altogether, the complexity is $\boldsymbol{O(kn^2)}$

b.

The Algorithm

Let's consider the first part for now. Finding the minimum cost path from s to t if G has no negative cycles is just having $n-1$ columns in our table M and returning $M[n-1, s]$ because, as proved in class and stated in the textbook, "If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most n − 1 edges".

Now let's consider detecting a negative cycle. To return the negative cycle, we need two things, to know that there IS a negative cycle, and the actual path (i.e. node sequence) – not just the weight – from each node to t, so that we could return the negative cycle.

To address the former, section 6.10 of the textbook proves that if we let the algorithm above run for one more iteration (have a row n), and see if any of the cells in that row are different from their corresponding cells in the same column but prior row (row $n-1$). If anything differs, then we have a negative cycle (according to Piazza post @101, we don't have to prove what's in the textbook already).

To address the latter, we would need to store the path as well, not just the path weight. The three cases we have to consider when storing paths is M[i-1,v] less than M[i-1,w] + edge_weight(v,w), greater than, and equal two. The former two cases make the running optimal solution unique because they're different values. The last case means there's two or more solutions We would maintain an extra table S that stores all the minimum paths. The algorithm is as follows:

```
Shortest-Path(G,s,t)
    Array M[0 ... n, V]
    Array S[0 ... n, V]
    Define M[0,t] = 0 and M[0,v] = ∞ for all over v ∈ V
    Define S[0,t] = [[t]] and S[0,v] = [[]] for all over v ∈ V
    For i = 1, ..., n
        For v ∈ V in any order
            nodes ws = a list of nodes that equally minimizes M[i-1,w] + edge_weight(v,w),
                , w being any element from ws
            if (M[i-1,w] + edge_weight(v,w) < M[i-1,v]) then
                M[i,v] = M[i-1,w] + edge_weight(v,w)

                For w ∈ ws
                    S[i,v] = v prepended to each of S[i-1,w]'s lists
                Endfor
            else if (M[i-1,w] + edge_weight(v,w) > M[i-1,v])
                M[i,v] = M[i-1,v]

                S[i,v] = S[i,v]
            else
                M[i,v] = M[i-1,v]
                S[i,v] = join(S[i-1,v], v prepended to each of S[i-1,w]'s lists)
            endif
        Endfor
    Endfor

    For v ∈ V in any order
        if M[n,v] != M[n-1,v] then
            return Get-Negative-Cycle(G,v)
    Endfor
    Return S[n-1,s]
```

Runtime Analysis

There are $n(n+1)$ cells in tables M and S, so it's $O\big(2n(n+1)\big) = O(n^2)$ to traverse them. Defining each cell takes $O(n)$ because there's n possible nodes for w and we need to prepend v

to each of their running solutions. Getting a negative cycle is $O(n^2 + n) = O(n^2)$ because it runs the same algorithm as Shortest-Path and iterates through a solution to find a cycle. Altogether, it is $\boldsymbol{O(n^3)}$**.**

c. Our algorithm in part b stores all paths with minimum cost (each cell in table S is a list of lists of nodes), so we just need to see the length of S[k,t] in a and S[n-1,t] in b. If it's 1, then it's unique, it it's greater than 1, then it's not unique.