A2 S2017

1. We use a modified BFS. The algorithm is the same as BFS, but we add the number of dollars we have left to our state. When we check the neighbours of each node, we do not travel private roads if we have no money.

```
function shortestPath(graph, K, s, t) {
    init queue, visited
    queue.push(s, K, 0)    // (node, dollarsLeft, pathLength)

    while (!queue.empty())
        node, dollarsLeft, pathLength = queue.pop()

        for neighbour of node    // cost = 0 for public, cost = 1 for private
            if (neighbour.cost <= dollarsLeft)
                if (neighbour == t)
                    return pathLength+1
                remainingDollars = dollarsLeft - neighbour.cost
                if (neighbour not in visited or visited[neighbour] < remainingDollars)
                    visited[neighbour] = remainingDollars
                    queue.push(neighbour, remainingDollars, pathLength+1)

    return -1   // impossible to reach
}
```

[To print the actual path, keep track of each node's parent and print them in reverse order]

Runtime: Since we run BFS once, the runtime of this algorithm is O(m+n).

2. [A2 problem]
We find the number of shortest s-t paths. If this number is 1, we return true. Otherwise we return false.

We use a modified BFS algorithm to return the number of shortest s-t paths. We initialize an array numpaths, where numpaths[v] is the number of shortest paths from s to v. We initialize this array to 0, except numpaths[s] = 1. We also initialize an array mindist, where mindist[v] is the minimum path length from v to s. In each iteration of the BFS, we update numpaths as follows. Let v be the current vertex we are traversing in the BFS. For each of its neighbours u, we:

If u has not been visited, we set numpaths[u] = numpaths[v], and mindist[u] = mindist[v] + 1.

Otherwise, if mindist[u] = mindist[v] + 1, we update numpaths[u] to numpaths[u] + numpaths[v].

At the end of the BFS, we return numpaths[t].

Correctness: We prove correctness of this algorithm by induction on k, the length of the shortest path from s to any vertex t.

Base case: k=0. Then s=t, and there is one shortest path from s to itself. We initialize numpaths[s] = 1, so we return the correct solution.

Inductive step: Assume the algorithm computes the correct number of shortest paths for all paths of length k. Every path of length k+1 from s to t must pass through a neighbor of t with distance k from s. By the inductive hypothesis, our algorithm computes the correct number of k-length shortest paths. Hence our algorithm computes the correct number of k+1-length shortest paths.

3. We assume that G is connected. We claim that if a directed acyclic graph contains a vertex from which all other vertices are reachable, then the vertex with the largest finish time in a DFS must be one of them. We prove this statement. Suppose the graph contains a vertex from which all other vertices are reachable, u. Suppose for sake of contradiction that the vertex with largest finish time is not u, but instead some vertex v where not all other vertices are reachable. If start[u] < start[v], then v must be a descendant of u, since all other vertices are reachable from u. Then by the parenthesis property, finish[u] > finish[v], which is a contradiction. If start[v] < start[u], then v was visited before u. If u is a descendant of v, then all other vertices must be reachable from v, since they are from u, which is a contradiction. If v is not a descendant of u, then we must have finish[v] < finish[u] by the parenthesis property, which is a contradiction. Hence we must have that if the graph contains a vertex from which all other vertices are reachable, then the vertex with the largest finish time in a DFS must be one of them.

Our algorithm then works as follows. We first find all the strongly connected components of G, and contract each into a single vertex, so we have a directed acyclic graph of SCCs. We then run DFS on this DAG, and find the vertex with the largest finish time. We run BFS/DFS starting from this vertex to check if all other vertices are reachable from it. If so, we return any of the vertices in this SCC. Otherwise we return false.

Runtime: It takes O(m+n) time to find the SCCs of G. We then run two iterations of DFS, which take O(m+n) time each. Hence our algorithm runs in O(m+n) time.

4. [A2 problem]
We have that a graph G has an odd directed cycle if and only if one of its strongly connected components is non-bipartite (as an undirected graph). Hence our algorithm works as follows.

We first run DFS to find its strongly connected components. For each strongly connected component, we check if its undirected version is bipartite, by converting it to an undirected graph then running BFS to check for bipartite-ness. If one of the SCCs is non-bipartite, the graph contains an odd directed cycle and we return true. Otherwise we return false.