1. DPV problem 3.9
The algorithm goes like the following:

```
getTwoDegree ($G(V, E)$)
    for each vertex $u \in V$
        degree[$u$]                    …(1)
    twoDegree[] = null # array declaration
    for each vertex $u \in V$
        for each vertex $v$ adjacent to $u$
            twoDegree[u] = twoDegree[u] + degree[v]          …(2)
```

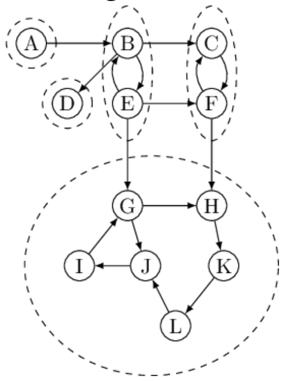
(1) goes around every vertex in graph G and calculates the degree of each node. And, we save this value in the degree[] array. Since there are $|E|$ number of nodes in our adjacency list, the cost of computation is $O(|E|)$.
Before the execution of the nested for-each loops, we declare an array called twoDegree[] which will be used to store the twodegree value of each $u \in V$. Let v be the neighbor(s) of u. For each u, we go through the adjacency list and add up all the degrees of v …(2). This operation returns the desired twoDegree[u] with $O(|E|)$ to go over all the edges again and sum up the degrees of v's. Therefore, our total complexity is linear.

2. DPV Problem 3.15

(a) We can view the intersections as vertices of a graph and the streets as directed edges. Then the question becomes whether a given graph is strongly connected. We can easily check this using Kosaraju's DFS algorithm. This algorithm incurs a linear time computation, using DFS twice. If the algorithm returns true, then the major's claim is also true that any pair of vertices(intersections) are connected.

(b) scc: strongly connected component (circled components)

The claim says that starting from an arbitrary vertex (city hall) in a graph, you can find a path that enters an scc and comes back out of it to return to the original vertex.



For example, we will construct a new graph G' from the graph above that contains scc's as its vertices. In our case, G' will have 5 vertices {A}, {D}, {B,E}, {C,F}, {G,H,I,J,K,L}. If any of the five vertices is a sink node, then we can claim that there is no path from that sink node back to the town hall. We can use BFS/DFS to check if there exists a sink node i.e., there is no outward edge from any of the nodes. Using BFS/DFS incurs linear time computation, therefore the major's claim can be checked in linear time.

3. DPV Problem 3.22

The construction of this problem will be similar to the one above (DPV 3.15)

Let the given graph be called $G(V, E)$. Then, we will construct a new graph $G'(V', E')$ which has the strongly connected components of G as its vertices. Also E' will be the flipped directed edges of E.

The target vertex $s \in V$ from which all other vertices are reachable be called the *target vertex*. Find where s is located in V'. Then, we run DFS/BFS on G' from v and mark all the visit-able vertices. After running the algorithm, if there is any vertices that had not been marked as visited, then this means that there is no path from the *Town Hall* to every other vertices in G' (or in other words no path from any vertex to the town hall).
If path fails, return false. Otherwise, return true.

The algorithm implements DFS/BFS and therefore has a linear running time.

4. DPV Problem 3.23

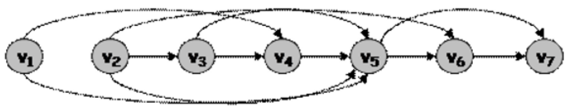
First do topological sorting of G. Then, remove all the vertices that come before s and after t (since a path cannot take a place before s or after t). Now, flip around all the edges. If $(a, b) \in E$, then $(b, a) \in E'$. We will call this new graph $G'(V', E')$.

We will run a BFS with G' but the only difference is that instead of the BFS having to returning *dist()*, we will make it return strings that show the path from t to s (ex. t -> a -> b -> s , t -> c -> b -> s , etc). This will allow us to count how many different paths are possible. Also, using BFS gives us linear time complexity which makes it an efficient algorithm.

5. DPV Problem 3.24

Use DFS to find the topological ordering of G. Note that G was a directed acyclic graph. We will call this new *linear* topological ordering G'.

Let n be the number of vertices in graph G'. We will number all the vertices again, starting from the leftmost vertex as v_1 all the way to v_n.



For i in range(n-1), if there is a directed edge between each ordered pair v_i and v_{i+1}, then G' contains a directed linear path from the start to the end that touches every vertex

The algorithm has a linear time of using DFS & traversing through G' (both operations are linear).