HW4.

1.

Consider the case of a strongly connected graph first. The case of a general graph can be handled by breaking it into its strongly connected components, since a cycle can only be present in a single SCC. We proceed by coloring alternate levels of the DFS tree as red and blue. We claim that the graph has an odd cycle if and only if there is an edge between two vertices of the same color (which can be checked in linear time).

If there is an odd cycle, it cannot be two colored and hence there must be a monochromatic edge. For the other direction, let u and v be two vertices having the same color and let (u, v) be an edge. Also, let w be their lowest common ancestor in the tree. Since u and v have the same color, the distances from w to u and v are either both odd or both even. This gives two paths p_1 and p_2 from w to v, one through u and one not passing through u, one of which is odd and the other is even.

Since the graph is strongly connected, there must also be a path q from v to w. Since the length of this path is either odd or even, q along with one of p_1 and p_2 will give an odd length tour (a cycle which might visit a vertex multiple times) passing through both v and w. Starting from v, we progressively break the tour into cycles whenever it intersects itself. Since the length of the tour is odd, one of these cycles must have odd length (as the sum of their lengths is the length of the tour).

2.

Let us call a vertex from which all other vertices are reachable, a *vista* vertex. If the graph has a vista vertex, then it must have only one source SCC (since two source SCCs are not reachable from each other), which must contain the vista vertex (if it's in any other SCC, there is no path from the vista vertex to the source SCC). Moreover, in this case *every* vertex in the source SCC will be a vista vertex.

The algorithm is then simply to a DFS starting from any node and mark the vertex with the highest `post` value. This must be in a source SCC. We now again run a DFS from this vertex to check if we can reach all nodes. Since the algorithm just uses decomposition into SCCs and DFS, the running time is linear.

3

We start by linearizing the DAG. Any path from s to t can only pass through vertices between s and t in the linearized order and hence we can ignore the other vertices.

Let $s = v_0, v_1, \ldots, v_k = t$ be the vertices from s to t in the linearized order. For each i, we count the number of paths from s to v_i as n_i. Each path to a vertex i and an edge (i, j), gives a path the vertex j and hence

$$n_j = \sum_{(i,j)\in E} n_i$$

Since $i < j$ for all $(i, j) \in E$, this can be computed in increasing order of j. The required answer is n_k.

4.

Start by linearizing the DAG. Since the edges can only go in the increasing direction in the linearized order, and the required path must touch all the vertices, we simply check if the DAG has an edge $(i, i+1)$ for every pair of consecutive vertices labelled i and $i + 1$ in the linearized order. Both, linearization and checking outgoing edges from every vertex, take linear time and hence the total running time is linear.

HW5.

1.

Perform k rounds of the `update` procedure on all edges.

2.

Let u and v be the end vertices of e. Using Dijkstra's algorithm, compute shortest path lengths from u and v to all other vertices in $G - e$, the graph obtained removing edge e from G. Let $d_u(x)$ and $d_v(x)$ respectively denote the shortest path length from u and v to node x in this graph. For any node x, there exists then a cycle of size at most $d_u(x) + d_v(x) + 1$ containing e in G: this cycle consists of the non-overlapping parts of the shortest paths from u and v to x and of edge e. Moreover, if x belongs to the shortest cycle C containing e, $d_u(x) + d_v(x) + 1$ must be the length of C or a shorter cycle will exist. This shows that the length of C is the minimum over all x of $d_u(x) + d_v(x) + 1$, which can be calculated in time $O(|V|)$. Hence, the overall running time is $O(|V|^2)$ given by the two initial runs of Dijkstra's algorithm.

3.

Suppose the graph has two different MSTs T_1 and T_2. Let e be the lightest edge which is present in exactly one of the trees (there must be some such edge since the trees must differ in at least one edge). Without loss of generality, say $e \in T_1$. Then adding e to T_2 gives a cycle. Moreover, this cycle must contain an edge e' which is (strictly) heavier that e, since all lighter edges are also present in T_1, where e does not induce a cycle. Then adding e to T_2 and removing e' gives a (strictly) better spanning tree than T_2 which is a contradiction.

4.

Multiply the weights of all the edges by -1. Since both Kruskal's and Prim's algorithms work for positive as well as negative weights, we can find the minimum spanning tree of the new graph. This is the same as the maximum spanning tree of the original graph.