(4.3) We can run a BFS for each node of the undirected graph. In each BFS, when updating the neighbor’s dist values for node \( u \), we also check whether \( \text{dist}(u) + \text{dist}(v) = 3 \), where \( v \) is a neighbor of \( u \). If the condition satisfies, we have a simple circle of length 4 in the graph. With adjacency list representation of the graph, the BFS has a complexity \( O(|V| + |E|) \). Therefore, the algorithm’s complexity is \( O(|V|^2 |V|) \). It is \( O(|V|^3) \), because \( |E| \) is \( O(|V|^2) \).

(4.7) We run Bellman-Ford algorithm on the tree \( T \). Since it is a tree, there is a single path from the specific node \( s \) to any other node. The paths can be found by depth first search with linear complexity. We then apply the update function to each tree edge and compute the tentative dist values for all the nodes. We need to check whether these dist values are the true ones. If they are, the tree \( T \) is the shortest path tree; otherwise it is not.

In fact, we simply need to test whether dist are stable. We run another pass of update for each edge of \( G \). If the dist changes for some edge target node, the dist assignment is not stable and otherwise it is. A stable dist assignment does not change no matter how many times we apply them. We then are sure that the dist values are the shortest distances from node \( s \) (did you see why?).

(4.9) Dijkstra’s algorithm is fine for such special negative weight graphs. If we add a large positive number \( L \) to these out-going edges from \( s \) and apply Dijkstra’s algorithm, the dist value for each node will be \( L + \text{dist}' \), where \( \text{dist}' \) is the original value for each node. This is apparent since each shortest path has to go through one of these neighbors of \( s \).

Subtracting \( L \) from each dist is equivalent to running the Dijkstra’s algorithm directly on the edges weights (including the negatives). It sure still works.

(4.10) Use Bellman-Ford shortest path algorithm. You need to show why \( k \) passes update is enough.

(4.11) To compute the shortest cycle in the graph, we need to find the shortest path between each pair of nodes. Since the graph is directed, \( d(i, j) \), the distance from node \( i \) to \( j \) may be different from \( j \) to \( i \). If \( D\{i, j\} = d(i, j) + d(j, i) < \infty \), there is a cycle that passes node \( i \) and node \( j \) (we are not interested in cycles that contain single nodes) and the sum equals the min cycle passing the two nodes. We just need to find the minimum \( D \).

Since all the edges are positive, we can run Dijkstra’s algorithm with each node as the starting point. However, we need to be careful about the implementation since we want to find a \( O(|V|^3) \) algorithm. Apparently, a simple binary heap is not good for the priority queue. We should use the more complex Fibonacci heap and the Dijkstra’s shortest path algorithm will have a complexity \( O(|V| \log |V| + |E|) \) (See textbook page 114). Thus the overall complexity is not worse than \( O(|V|^3) \).

(4.12) Denote \( e = (u, v) \). Run a Dijkstra’s algorithm with the starting node as \( v \). Use the Fibonacci heap to implement the priority queue.

(4.13) (1) Remove all the edges with length greater than \( L \) and then run a BFS or DFS.

(2) Remove all the edges with length greater than \( L \) and then run a Dijkstra’s algorithm.

(4.14) Run Dijkstra’s algorithm on \( G \) with \( v_0 \) as a starting node and get dist for each node. Then, run Dijkstra’s algorithm on \( G' \) (the reverse graph of \( G \)) with \( v_0 \) as a starting node and get dist' for each node. Here \( G' \) has every edge of \( G \) reversed. dist' tells us the shortest distance from each
node to \( v_0 \). We can use a double loop on all the nodes to combine \( dist \) and \( dist' \) to find out all the shortest distances between every pair of nodes and passing node \( v_0 \).

**5.3** If the number of the edges of \( G \) is greater than \(|V| - 1\), then there must be a circle and we can remove an edge and still keep \( G \) connected. However, \(|E|\) is \( O(|V|^2) \). So, counting all the edges will be a \( O(|V|^2) \) algorithm. In fact, we do not have to count all the edges; if the count is greater than \(|V| - 1\) we can stop. So, we never need to count more than \(|V|\). It is \( O(|V|) \).

**5.6** Use the cut property. For each cut, the minimum edge is on the MST. Since all the edges are distinct, the choice is unique for all the cuts. The MST must be unique.

**5.7** You can use a similar method to Kruskal’s algorithm or Prim’s algorithm to find the maximum spanning tree. In fact, the cut property still holds for the maximum spanning tree: the largest weight cutting edge must be in a maximum spanning tree. The proof is very similar to the MST cut property. If the edge is in a maximum spanning \( T \), we are done. If not, we show that we can build a \( T' \) that includes the edge and it is as good as \( T' \). The argument is that we can replace the connecting cut edge \( e \) in \( T \) with \( e' \) (the maximum cutting edge) and we still have a tree and since \( w(e') \geq w(e) \) the total weight is as big as \( T \). So, \( T' \) is also a maximum spanning tree. We finish the proof.

**5.8** For this question, it is impossible that the MST and the shortest path tree do not share edges. We take a special cut between set \( \{s\} \) and \( V - \{s\} \). Using the cut property, the smallest weight cutting edge must be in MST. This edge must also be in the shortest path tree.

**5.10** Still use the cut property. Each edge in \( T \), the MST of \( G \), is the smallest weight edge in each cut. If the edge is also in \( H \), it must be the smallest weight edge in a cut of \( H \) since \( H \) is a sub-graph of \( G \). Therefore, \( T \cap H \) is a MST of \( H \).