1. **(2 points each question) True or false?** Circle the correct answer. No explanation required. No points will be subtracted for incorrect answers, so guess all you want.

T F In an undirected graph, the shortest path between two nodes always lies on some minimum spanning tree.

False, think of a graph with three nodes, a, b, c with edge weights: $(a, b) = 4$, $(b, c) = 5$, $(c, a) = 6$. The MST will consist of edges (a, b) and (b, c), but the shortest path from a to c is not on the MST.

T F If all edges of a graph have different weights the minimum spanning tree is unique.

True, you proved this on your hw.

T F If all edges of a graph have different weights the second-best minimum spanning tree is unique.

False, consider a graph consisting of two triangles sharing one vertex. The first triangle has edge weights 1,2,3 and the second, 4,5,6. Then we obtain second-best MSTs by replacing 2 with 3 OR by replacing 5 with 6.

T F If all edges of the graph have different weights then the highest weight spanning tree is unique.

True, just multiply all the edge weights by -1 and you convert this to an MST problem.

T F Prim's algorithm for computing minimum spanning trees runs in $O(|V^2|)$ steps.

True, Prim has the same running time as Dijkstra which, implemented with a linked list, runs in $O(|V^2|)$ steps.

T F In a depth first search on a directed acyclic graph, the vertex with the highest postorder number is necessarily a source.

True.

T F In Huffman coding, the item with the second-lowest probability is always at the leaf that is furthest from the root.

True.

T F In Huffman coding, the item with the highest probability is always at the leaf that is closest to the root.

True.

T F In Huffman coding, the item with the highest probability is always at a leaf that is the child of the root.

False. The children of the root aren't necessarily leaves.

T F If a set of Horn clauses contains no clause with a single literal, then it is satisfiable.

True.

T F A set of Horn clauses can be tested for satisfiability in linear time.

True.

T F In a depth-first search of a directed graph, it is possible to have an edge between two vertices u and v with previsit and postvisit numbers: $(10, 40)$ for u and $(30, 50)$ for v.

False.

T F In a depth-first search of an undirected graph, it is possible to have an edge between two vertices u and v with previsit and postvisit numbers: $(5, 20)$ for u and $(30, 50)$ for v.

False, there are no cross edges in an undirected graph.

T F In a depth-first search of a directed graph, it is possible to have an edge between two vertices u and v with previsit and postvisit numbers: $(5, 20)$ for u and $(30, 50)$ for v.

True.

T F In a depth-first search of a directed graph, if u has pre and postvisit numbers $(10, 20)$ and v, $(15, 18)$, then (v, u) must be a back edge.

True.

- **(15 points total)** In the weighted graph shown below,
  - **(4 points)** In running Dijkstra's algorithm for shortest distances from node A, which nodes will be inserted into the heap more than once? How many times?

    B gets inserted 3 times.
  - **(4 points)** In running Prim's algorithm for minimum spanning tree from node A, what is the sequence of edges added to the minimum spanning tree?

    $(A, D), (D, E), (E, B), (B, C), (C, F)$
  - **(7 points)** In running Kruskal's algorithm for minimum spanning tree, what is the union-find tree (*with* path compression) after the end of the algorithm? Without path compression? (Recall that Kruskal's algorithm examines all edges of the graph.)

    Without path compression: A, B, D point at E, C, E point at F.

    With path compression: Everything points at F except for D which points at E.

- **(20 points total)** The police department in the city of Computopia has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. Furthermore, the city elections are coming up soon, and there is just enough time to run a *linear-time algorithm.*
  - Formulate this problem as a graph-theoretic problem, and explain why this problem can indeed be solved by a linear-time algorithm.

    Solution: Create a directed graph with the vertices being the intersections. Moreover create an edge from A to B if there is a one-way street from A to B. Then checking whether it is possible to drive from any intersection to any other intersection in the city is the same as checking whether the directed graph we created consists of one strongly connected component. We can check this by running the strongly connected component algorithm presented in class. (Run DFS on the reverse graph and then on the original graph starting with the node with highest post-order number.) This algorithm is linear. If it returns one strongly connected component the mayor is right, otherwise the opposition is right.
  - Suppose now that the mayor needs to show that, even though her original claim was false, something weaker does hold: If you start navigating one-way streets from the townhall, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how this problem too can be solved by a linear-time algorithm.

    Solution: Constructing a graph in the same way as in the first part of the question this problem is equivalent to asking, whether the townhall is a node in a sink strongly connected component: If the townhall is in a sink scc, then all the nodes reachable from the townhall are in the same scc as the townhall, so by the definition of strongly conncted components there is a path back to the townhall from all the nodes reachable from the townhall. If the townhall is not in a sink scc, then we can reach a vertex w from the townhall that is not in the same scc as the townhall, and then we will not find a path from w back to the townhall. The following algorithm checks whether the townhall is in a sink scc: Run the algorithm that finds the strongly connected components of the graph. Mark all the nodes that are in the same scc as the townhall. Then run DFS starting from the townhall until you get stuck. Check whether all the nodes reachable from the townhall are in the same scc as the townhall. If that is the case the mayor is right. If the set of nodes reachable from the townhall contains a node which is not in the same scc as the townhall the opposition is right. Since DFS is linear, this algorithm is linear.

- **(30 points total)** We are given a strongly connected directed graph $G = (V, E)$ with positive edge weights, and let $v_0 \in V$. We want to find the shortest paths between *all pairs of nodes*; however, we are only interested in paths that go through the particular node v_0.
  - **(20 points)** Give an $O(|V|^2)$ algorithm for solving this problem.

    Brief description or pseudocode:

    Run Dijkstra once with v_0 as the source. Reverse the graph. Run Dijkstra again from v_0. For each ordered pair (a, b), add the shortest path from v_0 to b and the shortest path from v_0 to a in the reverse graph.

    Justification of correctness and running time:

    The shortest path from a to b through v_0 is the union of the shortest path from a to v_0 and the shortest path from v_0 to b. The shortest path from v_0 to b is the shortest path from b to v_0 in the reverse graph.
    Dijkstra, implemented with a linked list, is $O(V^2)$. This algorithm involves running Dijkstra twice, reversing a graph which is linear time, and adding V^2 numbers. The total time is still $O(V^2)$.
  - **(5 points)** Do you think that there is a faster algorithm for this problem? Briefly support your answer.
    There is no faster algorithm because the output is size V^2.
  - **(5 points)** Prove or give a counterexample: The shortest path between nodes a and b through node v_0 in G, where $v_0 \neq a, b$ has no repeated edges.
    This is false. It's easy to come up with counterexamples.

  Comments: The most common mistake was running Dijkstra for every node; this yields an $\Omega(V^2)$ algorithm. Also, many students assumed a and b were inputs to the algorithm and solved an easier problem. The professors decided that you got at most 5 points of partial credit for this.

  Many students were confused about the running time for Dijkstra. Remember that using heap is not desirable if the graph is dense. The data structure yielding the optimal running time depends on the input graph.

- **Dynamic Programming (25 points)** We are given two strings x and y of length m and n, respectively. We are asked to find the *edit distance* between these two strings, that is, the minimum number of operations needed to transform x to y, when these kinds of operations are allowed: (i) insert a character in any position; (ii) change one character into another; (iii) delete *a whole consecutive block of characters of* x. Each of these three operations counts as one step. Find a dynamic programming algorithm that solves this problem, as follows:

  Define, for $i = 0, \ldots, m$ and $j = 0, \ldots, n$ $ED[i, j]$ to be the edit distance between the i characters of x and the first j characters of y.

  Initially, $ED[0, j] = j$ (delete the characters one at a time).
  and $ED[i, 0] = 1$ (delete the characters in one go).

  Recurrence equation: for $i, j > 0$,

$$ED[i,j] = \min \begin{cases} ED[i-1, j-1] & \text{if } x_i = y_j \\ ED[i, j-1] + 1 & \text{(insert into x)} \\ ED[i-1, j-1] + 1 & \text{(change character)} \\ \min_{1 < k \leq i} ED[i-k, j] + 1 & \text{(delete block from x)} \end{cases}$$

  Running time and justification.

  Setting each element $ED[i, j]$ takes $O(m)$ time, and there are mn subproblems overall, so the total running time is $O(m^2 n)$.

  (Extra Credit:) **(10 points)** Can you devise an $O(m \cdot n)$ algorithm for this problem?

  Think of the two-dimensional table $ED[i, j]$ in terms of its rows (i) and columns (j). Process the table according to the recurrence above, moving down one column at a time, starting at the left ($j = 0$).
  As you move down each column j (let's say you're at row i), keep track of the quantity $\min_{0 \leq k \leq i} ED[k, j]$. This requires only a constant amount of extra overhead per table entry, and enables each entry $ED[i, j]$ to be computed in constant time.