Instructions: Please note that handwritten assignments **will not be graded**. Use the provided LaTeX template to complete your homework. Please do not alter the order or spacing of questions (keep each question on its own page). When you submit to Gradescope, you must mark which page(s) correspond to each question. **You may not receive credit for unmarked questions**.
When including graphical figures, we encourage the use of tools such as graphviz or packages like tikz for simple and complex figures. However, these may be handwritten only if they are neat and legible (as defined by the grader).

List any collaborators (besides TAs or professors) here:

1. (35 points) [W4, ★★★★★] Alternative Topological Sorting

   When topologically sorting a directed-acyclic graph (DAG), let any edge $A \to B$ denote that A must come before B in any valid topological ordering.

   From the last homework, you know that a topological order must start with a vertex that has no incoming edges (in-degree of zero).

   (a) (10 points) Prove that a directed-acyclic graph has at least one vertex with in-degree zero and one vertex with out-degree zero. (Hint: prove in-degree by contradiction, and then reference that proof for out-degree).

   Proof by contradiction: Assume that a DAG has no vertex with in-degree zero. Then, there is no vertex that can be the first vertex in a topological ordering. This contradicts the definition of a topological ordering. Therefore, a DAG must have at least one vertex with in-degree zero.

   For the out-degree, the proof is the same as above but with the roles of in-degree and out-degree reversed.

   (b) (15 points) Based on your previous answer, describe an algorithm for topological sorting that uses in-degrees to process vertices. The complexity of your algorithm must be no worse than the DFS-based toposort algorithm discussed in class.

   1. Create an array to store the in-degree of each vertex.
   2. Enqueue all vertices with in-degree zero.
   3. While the queue is not empty, do the following:
      - Dequeue a vertex v from the queue.
      - Add v to the topological ordering.
      - For each neighbor u of v, decrement the in-degree of u by one.
      - If the in-degree of u is zero, enqueue u.
   4. If the topological ordering contains all vertices, return the topological ordering. Otherwise, return that the graph has a cycle.

   (c) (10 points) While running this algorithm, it reaches a point where there are still vertices left that have not been processed, yet none of these vertices have an in-degree of zero. What does this indicate about the graph?

   This indicates that the graph contains a cycle.

2. (10 points) [W5, ⋆⋆] MST.

   (a) (2 points) Select whether the statement is true or false, and write a *brief* explanation of your reasoning.

   If each edge of a graph has a unique weight, the MST of that graph will be unique.

   ■ True □ False **No explanation necessary for part (a).**

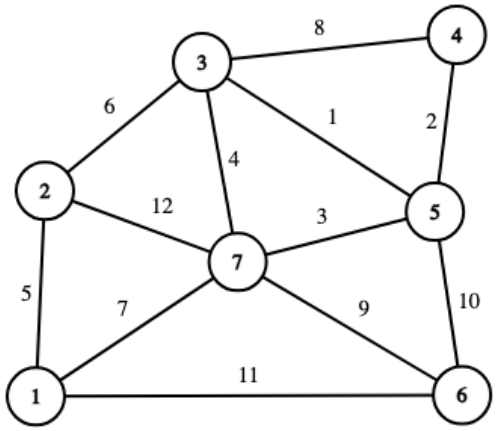
   (b) (4 points) Describe how you would modify Prim's algorithm to compute the **maximum** spanning tree. (That is, the tree with the highest cost.)

   Simply change the edge selection criteria from the minimum to the maximum.

   (c) (4 points) Describe how you would modify Kruskal's algorithm to compute the **maximum** spanning tree. (That is, the tree with the highest cost.)

   Change the sorting order of the edges from ascending to descending.

3. (10 points) [W5, ⋆] Disjoint Set Data Structure.



Run the Kruskal's algorithm on the graph above and show the state of the disjoint set array each time after an edge is added to the MST. Do not use path compression and attach the smaller component to the larger component during merge. **When merging components of same sizes, attach the component with the smaller root node ID to the larger one. See I1 below.** You can track component sizes manually (no need to show sizes in your solution).

I0:

1	2	3	4	5	6	7
0	0	0	0	0	0	0

I1:

1	2	3	4	5	6	7
0	0	5	0	0	0	0

I2:

1	2	3	4	5	6	7
0	0	5	5	0	0	0

I3:

1	2	3	4	5	6	7
0	0	5	5	0	0	5

I4:

1	2	3	4	5	6	7
2	0	5	5	0	0	5

I5:

1	2	3	4	5	6	7
2	3	5	5	0	0	5

I6:

1	2	3	4	5	6	7
2	3	5	5	0	7	5

4. (25 points) [W6, ★★★★] Let G be a connected (strongly connected, if directed) weighted graph, where the weights are integers (i.e., weights can be positive, negative or zero), that does not contain any negative weighted cycles. Prove that there exists a shortest path between any two distinct vertices s and t such that no vertex on the path is repeated (i.e., the path does not include a cycle).

   Proof by contradiction:

   Assume all of the paths from s to t contain cycles (at least one repeated node).

   The cycle in each path has a total positive weight or zero.

   Because the cycle weight is positive or zero, removing the cycle would result in the same total length (if cycle weight is 0) or a shorter path weight (if cycle weight is positive).

   i.e. a shortest path is possible without repeating nodes.

5. (20 points) [W6, ⋆⋆] Shortest Paths. For the following questions, select the true statement(s). **No explanation is necessary for these questions.**

   (a) (5 points) Djikstra's algorithm can also be thought of as a search algorithm. When thought of this way, we are searching for the shortest path from node A to node B. Under this interpretation, focusing on the data structures used, choose the sequence of data structures which matches up respectively to the following sequence: BFS, DFS, Djikstra's.

   □ stack, queue, priority queue
   □ priority queue, queue, stack
   ■ queue, stack, priority queue
   □ stack, priority queue, queue

   (b) (5 points) Djikstra's algorithm in full finds the shortest path from a source node to all other nodes in the graph. Now think again of the search case. That is, we are only interested in the shortest path between the source node A and one other node B in the graph. Which of the statements below is correct?

   ■ We can stop Djikstra's algorithm once we have first overwritten the distance of node B.
   □ We can stop the search once we have processed node B.

   (c) (10 points) Select the correct statements:

   □ Dijkstra's algorithm can correctly find shortest paths in graphs that contain negative weight edges, but not negative weight cycles.
   ■ The Bellman-Ford algorithm can correctly find shortest paths in graphs that contain negative weight edges, but not negative weight cycles.
   □ Dijkstra's algorithm can correctly find shortest paths in graphs that contain negative weight cycles.
   □ The Bellman-Ford algorithm can correctly find shortest paths in graphs that contain negative weight cycles.
   ■ The complexity of Djikstra's algorithm when using a Fibonacci heap is $\mathcal{O}(n \log n + m)$.
   ■ The Bellman-Ford algorithm is $\mathcal{O}(nm)$.
   ■ The first `for` loop in the Bellman-Ford algorithm determines whether the graph contains a negative weight cycle.
   ■ The first `for` loop of the Bellman-Ford algorithm runs $n - 1$ times.