Algorithms: Floyd-Warshall

1 *Model 1: Adjacency Matrix*

	v_1	v_2	v_3	v_4
v_1	-	8	7	2
v_2	3	-	3	6
v_3	8	3	-	2
v_4	6	1	5	-

Figure 1: An adjacency matrix for a weighted directed graph G. The value at $M[i, j]$ corresponds to the weight of edge (v_i, v_j). In other words, nodes on the left are originating vertices. Nodes on the top are destination vertices.

(page intentionally left blank for scratch work)

2 Consider the *adjacency matrix* M in Model 1: it describes a weighted graph. In other words, the entry $M[i, j]$ stores the weight of the edge $v_i \to v_j$. Note that the matrix is not symmetric; this is because the graph that M reprents is a *directed* weighted graph. In the space below the model, draw the graph that M depicts.

Learning objective: Students will derive an efficient algorithm for finding the shortest paths between all pairs of vertices in a directed, weighted graph.

3 Examining the graph, list all pairs of vertices for which there is a path through an intermediate vertex that is cheaper than tracing the edge from the first to the second vertex. For example, it is cheaper to travel from v_1 to v_2 through intermediate vertex v_4 (total length: 2 ($v1$ to v_4) + 1 (v_4 to v_2) = 3) than directly from v_1 to v_2 (edge weight: 6).

Note again that the graph is not symmetric; the shortest path from v_1 to v_2 need not be the same as the shortest path from v_2 to v_1.

Be sure to list the pair, the intermediate vertex, and the total cost of the path with the intermediate vertex. You should find a total of five pairs.

4 Find an instance where taking a path that goes through a second intermediate vertex is cheaper than both (a) the path going through just one intermediate vertex and (b) tracing the single edge that connects the starting and ending vertices. There is only one such instance.

5 The partially completed algorithm below finds the shortest path distance between any pair of vertices for a graph with n vertices. Here are some notes about the algorithm:

- The parameter `g` refers to the graph being explored,
- `g.edge_weight(i, j)` returns the weight of the edge that connects v_i to v_j in graph g.
- The `start` and `end` parameters represent the indices of the vertices between which we seek the shortest path: $v_{\text{start}} \rightsquigarrow v_{\text{end}}$
- The k parameter represents the maximum vertex index under consideration as an intermediate node. For example:
  - When $k = 0$, no intermediate nodes will be considered; only the edge weight from `start` to `end` can be used.
  - When $k = 1$, v_1 can be considered.
  - When $k = 2$, v_1 and v_2 can be considered,
  - ...

Inspired by your answers to Questions 3 and 4, fill in the base case and recursive cases.

```
shortest_path_distance(Graph g, int start, int end, int k)
  if k == 0
    return (                                               )
  else
    let distance_ignoring_vertex_k =
       shortest_path_distance(g,     ,     ,    )
    let distance_using_vertex_k =
       shortest_path_distance(g,     ,     ,    )
       + shortest_path_distance(g,     ,     ,    )
    if (                                                )
      return distance_ignoring_vertex_k
    else
      return distance_using_vertex_k
```

6 List every distinct base-case recursive call encountered when calling the algorithm on Model 1 with the call `shortest_path_distance(model_1, 1, 4, 4)`. For example, one base case is: `start=1, end=4, k=0`.

7 To prove the algorithm from Question 5 correctly finds the shortest path from `start` to `end`, we need to use *strong induction*[1] due to its multiple distinct recursive calls. What would all the base cases be for a correctness proof by strong induction?

[1] In weak induction, we assume that $P(k)$ is true in order to prove $P(k+1)$. In strong induction, we assume that our inductive hypothesis holds for *all* values preceding k. Said differently, we assume that each $P(i)$—from our base case up until $P(k)$—is true (e.g., $P(1), P(2), \ldots, P(k)$ all hold) in order to prove that $P(k+1)$ is true.

8 How many base cases would there be?

9 What would be the inductive hypothesis?

10 What would we need to prove in the inductive step?

11 Complete the proof by strong induction that this algorithm finds the shortest path from `start` to `end`.

12 Write a recurrence for the asymptotic time complexity of the algorithm you wrote in Question 5. Remember, the recurrence should capture: the number of recursive calls, the size of the subproblems, and the total nonrecursive work in each call.

13 Estimate the asymptotic time complexity of your algorithm based on the recurrence from Question 12.

14 How might the algorithm (or adjacency matrix) from Question 5 be modified to handle missing edges?

15 If we were to rewrite the algorithm from Question 5 as a bottom-up dynamic programming algorithm (e.g., left-to-right, row-major order, etc.), how many dimensions would the table need to have? What would each dimension represent?

16 Write pseudocode for a bottom-up dynamic programming version of the algorithm. (This is known as the Floyd-Warshall algorithm.)

17 What is the asymptotic time complexity of the algorithm from Question 16? (You may want to think about the number cells in your memoziation data structure and the cost of filling in each cell. Is there any preprocessing?)