1. After moving to a new city, you decide to choose a walking route from your home to your new office. Your route must consist of an uphill path (for exercise) followed by a downhill path (to cool down), or just an uphill path, or just a downhill path. But you also want the *shortest* path that satisfies these conditions, so that you actually get to work on time.

   Your input consists of an undirected graph G, whose vertices represent intersections and whose edges represent road segments, along with a start vertex s and a target vertex t. Every vertex v has a value $h(v)$, which is the height of that intersection above sea level, and each edge uv has a value $\ell(uv)$, which is the length of that road segment.

   (a) Describe and analyze an algorithm to find the shortest uphill–downhill walk from s to t. Assume all vertex heights are distinct.

Solution: We define a new directed graph $G' = (V', E')$ as follows:

- $V' = \{v^\uparrow, v^\downarrow \mid V \in V\}$. Vertex $v^\uparrow$ indicates that we are at intersection v moving uphill, and vertex $v^\downarrow$ indicates that we are at intersection v moving downhill.
- E' is the union of three sets:
  - Uphill edges: $\{u^\uparrow \to v^\uparrow \mid uv \in E \text{ and } h(u) < h(v)\}$. Each uphill edge $u^\uparrow \to v^\uparrow$ has weight $\ell(uv)$.
  - Downhill edges: $\{u^\downarrow \to v^\downarrow \mid uv \in E \text{ and } h(u) > h(v)\}$. Each downhill edge $u^\downarrow \to v^\downarrow$ has weight $\ell(uv)$.
  - Switch edges: $\{v^\uparrow \to v^\downarrow \mid v \in V\}$; each switch edge has weight 0.

We need to compute three shortest paths in this graph:

- The shortest path from $s^\uparrow$ to $t^\downarrow$ gives us the best uphill-then-downhill route.
- The shortest path from $s^\uparrow$ to $t^\uparrow$ gives us the best uphill-only route.
- The shortest path from $s^\downarrow$ to $t^\downarrow$ gives us the best downhill-only route.

G' is a directed ***acyclic*** graph; we can get a topological ordering by listing the up vertices $v^\uparrow$, sorted by increasing height, followed by the down vertices $v^\downarrow$, sorted by decreasing height. Thus, we can compute the shortest path in G' from any vertex to any other in $O(V' + E') = O(V + E)$ time by dynamic programming. (The algorithm is the same as the longest-path algorithm in the notes, except we use "min" instead of "max" in the recurrence, and define $\min \varnothing = \infty$.)

Our overall algorithm runs in $\boldsymbol{O(V + E)}$ ***time***. ■

Rubric: 5 points = 1 for vertices + 1 for edges + 1 for arguing G' is a dag + 1 for algorithm + 1 for running time. This is not the only correct solution. Max 4 points for a correct reduction to Dijkstra's algorithm that runs in $O(E \log V)$ time.

(b) Suppose you discover that there is no path from s to t with the structure you want. Describe an algorithm to find a path from s to t that alternates between "uphill" and "downhill" subpaths as few times as possible, and has minimum length among all such paths. (There may be even shorter paths with more alternations, but you don't care about them.) Again, assume all vertex heights are distinct.

Solution (Dijkstra, 5/5): Let $L = 1 + \sum_{u \to v} \ell(u \to v)$. Define a new graph $G' = (V', E')$ as follows:

- $V' = \{v^\uparrow, v^\downarrow \mid V \in V\} \cup \{s, t\}$. Vertex $v^\uparrow$ indicates that we are at intersection v moving uphill, and vertex $v^\downarrow$ indicates that we are at intersection v moving downhill.
- E' contains four types of edges:
  - Uphill edges: $\{u^\uparrow \to v^\uparrow \mid uv \in E \text{ and } h(u) < h(v)\}$. Each uphill edge $u^\uparrow \to v^\uparrow$ has weight $\ell(uv)$.
  - Downhill edges: $\{u^\downarrow \to v^\downarrow \mid uv \in E \text{ and } h(u) > h(v)\}$. Each downhill edge $u^\downarrow \to v^\downarrow$ has weight $\ell(uv)$.
  - Switch edges: $\{v^\uparrow \to v^\downarrow \mid v \in V\} \cup \{v^\downarrow \to v^\uparrow \mid v \in V\}$. Each switch edge has weight L.
  - Start and end edges $s \to s^\uparrow$, $s \to s^\downarrow$, $t^\uparrow \to t$, and $t^\downarrow \to t$, each with weight 0,

We need to compute the shortest path from s to t in G'; the large weight L on the switch edges guarantees that this path with have the minimum number of switches, and the minimum length among all paths with that number of switches. Dijkstra's algorithm finds this shortest path in $O(E' \log V') = \boldsymbol{O(E \log V)}$ ***time***.

(Because G' includes switch edges in both directions, G' is not a dag, so we can't use dynamic programming directly.) ■

Rubric: 5 points, standard graph-reduction rubric. This is not the only correct solution with running time $O(E \log V)$.

Solution (clever, +5 extra credit): Our algorithm works in two phases: First we determine the minimum number of switches required to reach every vertex, and then we compute the shortest path from s to t with the minimum number of switches. The first phase is can be solved in $O(V+E)$ time by a modification of breadth-first search; the second by computing shortest paths in a dag.

For the first phase, we define a new graph $G' = (V', E')$ as follows:

- $V' = \{v^\uparrow, v^\downarrow \mid V \in V\} \cup \{s, t\}$. Vertex $v^\uparrow$ indicates that we are at intersection v moving uphill, and vertex $v^\downarrow$ indicates that we are at intersection v moving downhill.
- E' contains four types of edges:
  - Uphill edges: $\{u^\uparrow \to v^\uparrow \mid uv \in E \text{ and } h(u) < h(v)\}$. Each uphill edge has weight 0.
  - Downhill edges: $\{u^\downarrow \to v^\downarrow \mid uv \in E \text{ and } h(u) > h(v)\}$. Each downhill edge has weight 0.
  - Switch edges: $\{v^\uparrow \to v^\downarrow \mid v \in V\} \cup \{v^\downarrow \to v^\uparrow \mid v \in V\}$. Each switch edge has weight 1.
  - Start and end edges $s \to s^\uparrow$, $s \to s^\downarrow$, $t^\uparrow \to t$, and $t^\downarrow \to t$, each with weight 0.

Now we compute the shortest path distance from s to every other vertex in G'. We could use Dijkstra's algorithm in $O(E \log V)$ time, but the structure of the graph supports a faster algorithm.

Intuitively, we break the shortest-path computation into phases, where in the kth phase, we mark all vertices at distance k from the source vertex s. During the kth phase, we amy also discover vertices at distance $k+1$, but no further. So instead of using a binary heap for the priority queue, it suffices to use two bags: one for vertices at distance k, and one for vertices at distance $k+1$.

```
ZeroOneDijkstra(G, ℓ, s):
  s.dist ← 0
  for all vertices v ≠ s
      v.dist ← ∞
  curr ← new empty bag
  add s to curr
  for k ← 0 to V
      next ← new empty bag
      while curr is not empty
          take v from curr          ⟨⟨v.dist = k⟩⟩
          for all edges v→w
              if w.dist > v.dist + ℓ(v→w)
                  w.dist ← v.dist + ℓ(v→w)
                  if ℓ(v→w) = 0
                      add w to curr
                  else ⟨⟨if ℓ(v→w) = 1⟩⟩
                      add w to next
      curr ← next
```

This phase of the algorithm runs in $O(V' + E') = O(V + E)$ time.

Once we have computed distances in G', we construct a second graph $G'' = (V', E'')$ with the same vertices as G', but only a subset of the edges:

$$E'' = \left\{ u' \to v' \in E' \;\middle|\; u'.dist + \ell(u' \to v') = v'.dist \right\}$$

Equivalently, an edge $u' \to v'$ belongs to E'' if and only if that edge is part of at least one shortest path in G' from s to another vertex. It follows (by induction, of course), that every path in G'' from s to another vertex v' is a shortest path in G', and therefore a minimum-switch path in G.

We also reassign the edge weights in G''. Specifically, we assign each uphill edge $u^\uparrow \to v^\uparrow$ and downhill edge $u^\downarrow \to v^\downarrow$ in G'' weight $\ell(uv)$, and we assign every switch edge, start edge, and end edge weight 0. **Now we need to compute the shortest path from s to t in G'', with respect to these new edge weights.**

We can expand the definition of E'' in terms of the original input graph as follows:

$$\begin{aligned} E'' = &\left\{ u^\uparrow \to v^\uparrow \;\middle|\; uv \in E \text{ and } h(u) < h(v) \text{ and } u^\uparrow.dist = v^\uparrow.dist \right\} \\ &\cup \left\{ u^\downarrow \to v^\downarrow \;\middle|\; uv \in E \text{ and } h(u) > h(v) \text{ and } u^\downarrow.dist = v^\downarrow.dist \right\} \\ &\cup \left\{ v^\uparrow \to v^\downarrow \;\middle|\; v \in V \text{ and } v^\uparrow.dist < v^\downarrow.dist \right\} \\ &\cup \left\{ v^\downarrow \to v^\uparrow \;\middle|\; v \in V \text{ and } v^\downarrow.dist < v^\uparrow.dist \right\} \end{aligned}$$

We can topologically sort G'' by first sorting the vertices by increasing $v'.dist$, and then within each subset of vertices with equal $v'.dist$, listing the up-vertices by increasing height, followed by the down vertices by decreasing height. It follows that G'' is a dag! Thus, we can compute shortest paths in G'' in $O(V'' + E'') = O(V + E)$ time, using the same dynamic programming algorithm that we used in part (a).

The overall algorithm runs in $\boldsymbol{O(V+E)}$ ***time***. ■

Rubric: max 10 points =

- 5 for computing minimum-switch paths = 1 for vertices + 1 for edges (including weights) + 2 for 0/1 shortest path algorithm + 1 for running time.
- 5 for computing shortest minimum-switch paths = 1 for vertices + 1 for edges (including weights) + 1 for proving dag + 1 for dynamic programming algorithm + 1 for running time

If total score on problem 1 is greater than 10, record excess as extra credit.

2. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero.

   (a) How could we delete an arbitrary vertex v from this graph, without changing the shortest-path distance between any other pair of vertices? Describe an algorithm that constructs a directed graph $G' = (V', E')$ with weighted edges, where $V' = V \setminus \{v\}$, and the shortest-path distance between any two nodes in G' is equal to the shortest-path distance between the same two nodes in G, in $O(V^2)$ time.

Solution: To transform G into G', we replace every path $u \to v \to x$ with a single edge $u \to x$ of the same length, unless $u \to x$ is already shorter than $u \to v \to x$. We make three assumptions to simplify our formal presentation:

- The input graph G has no negative cycles.
- The input to our algorithm is a matrix $w[1..V, 1..V]$ of edge weights, with $w[i, i] = 0$ for all i and $w[i, j] = \infty$ for any other missing edge $i \to j$. If G is given as an adjacency list, we can can easily construct this matrix in $O(V^2)$ time.
- We are deleting the vertex v represented by the last row and column of the weight matrix; otherwise, we can swap two rows and two columns in $O(V)$ time.

The following algorithm clearly runs in $O(V^2)$ time.

```
RemoveLastVertex(w[1..V, 1..V]):
  for i ← 1 to V − 1
      for j ← 1 to V − 1
          w'[i, j] ← min{w[i, j], w[i, V] + w[V, j]}
  return w'[1..V − 1, 1..V − 1]
```

For any path p, let $w(p)$ denote its length in the input graph G, and let $w'(p)$ denote its length in the output graph G'. For any path p in G, let p' be the path in G' obtained by replacing any subpath $u \to v \to x$ with the edge $u \to x$; in particular, if p does not pass through v, then $p' = p$.

The identity $w'(u \to x) = \min\{w(u \to v \to x), w(u \to x)\}$ immediately implies that $w'(p') = \min\{w(p), w(p')\}$ for every path p in G. This identity has two consequences:

- Every path in G' has the same length as some path in G with the same endpoints. Thus, distances in G' cannot be smaller than distances in G.
- If p is a *shortest* path in G, then $w(p) \le w(p')$, so $w'(p') = w(p)$. Thus, distances in G' cannot be larger than distances in G.

We conclude that distances in G and G' are equal, as required. ■

Rubric: 4 points = 1 for main idea + 3 for algorithm details + 1 for time analysis. The proof of correctness is not required for full credit.

(b) Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances in the original graph G from v to every other vertex, and from every other vertex to v, all in $O(V^2)$ time.

Solution (brute force): For all vertices $x \neq v$, the shortest-path distances between s and v satisfy the following identities:

$$dist(x,v) = \min_{u \neq v} \big(dist(x,u) + w(u \to v) \big)$$

$$dist(v,x) = \min_{u \neq v} \big(dist(u,x) + w(v \to u) \big)$$

Both $dist(s,u)$ and $dist(u,x)$ are distances in G', which we've already computed, and the edge weights $w(u \to v)$ and $w(u \to v)$ are part of the input. Thus, we can simply implement these equations directly, with no further recursive calls.

Again, we assume that the graph G is specified by a matrix of edge weights, whose last row and column correspond to v. We are also given the matrix $dist[1..V, 1..V]$ of shortest path distances in G'; our task is to fill in the last row and column of this matrix.

The following algorithm clearly runs in $\boldsymbol{O(V^2)}$ ***time***:

```
RestoreLastVertex(w[1..V, 1..V], dist[1..V, 1..V]):
  for i ← 1 to V − 1
      dist[i, V] ← ∞
      for j ← 1 to V − 1
          dist[i, V] ← min{dist[i, V], dist[i, j] + w[j, V]}
      dist[V, i] ← ∞
      for h ← 1 to V − 1
          dist[V, i] ← min{dist[V, i], w(V, h) + dist[h, i]}
```

■

Rubric: 4 points = 1 for main idea + 2 for algorithm details + 1 for time analysis.

(c) Combine parts (a) and (b) into another all-pairs shortest path algorithm that runs in $O(V^3)$ time. (The resulting algorithm is *almost* the same as Floyd-Warshall!)

Solution: If the graph has only one vertex, return a 1×1 array containing the number 0. Otherwise, call DELETELASTVERTEX to delete some vertex v, recursively compute shortest paths in the smaller graph, and finally call RESTORELASTVERTEX to compute all distances to v and from v. The running time satisfies the recurrence $T(V) = T(V-1) + O(V^2)$, so the algorithm clearly runs in ***O(V³) time***, as required.

Unrolling all the recursive calls gives us the following iterative algorithm:

```
KLEENEAPSP(w[1..V, 1..V]):
  for i ← 1 to V
      for j ← 1 to V
          dist[i, j] ← w[i, j]

  ⟨⟨Tear down the graph⟩⟩
  for i ← V down to 1
      for j ← 1 to i − 1
          for k ← 1 to i − 1
              dist[j, k] ← min{dist[j, k], dist[j, i] + dist[i, k]}

  ⟨⟨Build the graph back up⟩⟩
  for i ← 1 to V
      for k ← 1 to i − 1
          for j ← 1 to i − 1
              dist[i, k] ← min{dist[i, k], dist[i, j] + dist[j, k]}
              dist[k, i] ← min{dist[k, i], dist[k, j] + dist[j, i]}

  return dist[1..V, 1..V]
```

■

Rubric: 2 points = 1 for algorithm + 1 for time analysis. Full credit for correctly invoking parts (a) and (b) even if those parts are not solved correctly (or at all). It is not necessary to give the unrolled iterative pseudocode.

3. The first morning after returning from a glorious spring break, Alice wakes to discover that her car won't start, so she has to get to her classes at Sham-Poobanana University by public transit. She has a complete transit schedule for Poobanana County. The bus routes are represented in the schedule by a directed graph G, whose vertices represent bus stops and whose edges represent bus routes between those stops. For each edge $u \to v$, the schedule records three positive real numbers:

   - $\ell(u \to v)$ is the length of the bus ride from stop u to stop v (in minutes)
   - $f(u \to v)$ is the first time (in minutes past 12am) that a bus leaves stop u for stop v.
   - $\Delta(u \to v)$ is the time between successive departures from stop u to stop v (in minutes).

   Alice wants to leaves from stop s (her home) at a certain time and arrive at stop t (The See-Bull Center for Fake News Detection) as quickly as possible.

   Describe and analyze an algorithm to find the earliest time Alice can reach her destination. Your input consists of the directed graph $G = (V, E)$, the vertices s and t, the values $\ell(e), f(e), \Delta(e)$ for each edge $e \in E$, and Alice's starting time (in minutes past 12am).

Solution: First, for any edge $u \to v$ and any time T, define

$$next(u \to v, T) := f(u \to v) + \left\lceil \frac{T - f(u \to v)}{\Delta(u \to v)} \right\rceil \cdot \Delta(u \to v) + \ell(u \to v);$$

this is the earliest time that a rider who starts at stop u at time T can reach stop v along the edge $u \to v$.

Now we modify Dijkstra's algorithm by using earliest arrival times instead of shortest-path distances everywhere. Specifically:

- Each vertex v stores a value $v.when$, which stores our current estimate of the earliest time that Alice can reach v. Initially, we set $s.when$ to Alice's starting time and $v.when \leftarrow \infty$ for all $v \neq s$.
- An edge $u \to v$ is tense if $v.when > next(u \to v, u.when)$. To relax $u \to v$, we set $v.when \leftarrow next(u \to v, u.when)$ and then insert v into the priority queue with priority $v.when$ (or decrease v's priority if v is already in the priority queue).

Otherwise the algorithm is unchanged. Because we can compute $next(u \to v, u.when)$ in $O(1)$ time, the analysis is also unchanged; the algorithm runs in $\boldsymbol{O(E \log V)}$ ***time***.

When all edge weights are positive, the proof of correctness for Dijkstra's (standard) algorithm ultimate relies only on the following fact: *For any path P starting at s, every proper prefix of P is shorter than P.* Our modified "shortest" path problem also satisfies this property, where "shorter" means "ends earlier". Thus, exactly the same proof implies that that our modification of Dijkstra discovers vertices in order of increasing minimum arrival time. ■

Rubric: 10 points = 3 points for *next* + 5 points for Dijkstra modifications + 2 points for running time. The proof of correctness is not required.