1. Describe and analyze an algorithm to compute the shortest path from vertex s to vertex t in a directed graph with weighted edges, where exactly *one* edge $u \to v$ has negative weight. Assume the graph has no negative cycles. *[Hint: Modify the input graph and run Dijkstra's algorithm. Alternatively,* ***don't*** *modify the input graph, but run Dijkstra's algorithm anyway.]*

Solution (first hint): Let G denote the input graph, let $w(x \to y)$ denote the weight of edge $x \to y$, and let $u \to v$ denote the unique edge in G with negative weight. The shortest path in G from s to t either traverses the edge $u \to v$ or it doesn't; we consider each case separately.

Remove edge $u \to v$ from G and let G' denote the resulting graph. For any vertices x and y, let $dist(x, y)$ and $dist'(x, y)$ denote the distances from x to y in G and G', respectively. Then we have

$$dist(s,t) = \min \begin{Bmatrix} dist'(s,t) \\ dist'(s,u) + w(u \to v) + dist'(v,t) \end{Bmatrix}$$

Thus, we can compute $dist(s, t)$ by running Dijkstra *twice* in G': once starting at s to compute both $dist'(s, t)$ and $dist'(s, u)$, and once starting from v to compute $dist'(v, t)$. The algorithm runs in $\boldsymbol{O(E \log V)}$ ***time***. ■

Solution (second hint): We can compute the shortest path distance from s to t using a single invocation of the more general version of Dijkstra's algorithm, which reinserts vertices into the priority queue whenever an incoming edge is relaxed, on the original input graph. I claim that Dijkstra's algorithm runs in $O(E \log V)$ time in this setting.

As in the previous solution, let G denote the input graph, and let $u \to v$ denote the unique edge in G with negative weight.

Claim 1. *Vertex u is* Extracted *from the priority queue at most once.*

Proof: Following the exposition in the textbook, for any index i, let u_i denote the vertex return by the ith call to ExtractMin, and let d_i be the value of $dist(u_i)$ just after this Extraction.

Suppose vertex u (the tail of the only negative edge) is Extracted for the first time in the jth iteration of the main loop, so $u_j = u$ and $u_i \neq u$ for all $i \leq j$. Then edge $u \to v$ is not relaxed during the first $j-1$ iterations. Thus, the first $j-1$ iterations would be identical if edge $u \to v$ were removed from the graph. It follows that for all $i \leq j$, distance d_i is the length of the shortest path from s to u_i that does *not* include the edge $u \to v$. The true shortest path from s to u does not include the edge $u \to v$. Thus, when u is extracted for the first time, $dist(u)$ is already the true shortest-path distance from s to u. We conclude that u is never Extracted again. □

Claim 1 immediately implies that the negative edge $u \to v$ is relaxed at most once. Following the proof of Lemma 8.4 in the textbook, it follows that each vertex of G is Extracted from the priority queue at most *twice*: at most once before $u \to v$ is relaxed,

and at most once after $u \to v$ is relaxed. Thus, each edge is relaxed at most twice, so the number of priority queue operations is at most $O(E)$. We conclude that Dijkstra's algorithm still runs in $\boldsymbol{O(E \log V)}$ ***time*** in this setting, as claimed. ■

2. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cities that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where *exactly* should you meet?

   You are given a weighted graph $G = (V, E)$, where the vertices V represent cities and the edges E represent roads that directly connect cities. Each edge e has a weight $w(e)$ equal to the time required to travel between the two cities. You are also given a vertex p, representing your starting location, and a vertex q, representing your friend's starting location.

   Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as soon as possible, assuming both of you leave home *right now*.

Solution: Let $dist(x, y)$ denote the shortest-path distance from vertex x to vertex y. If my friend and I decide to meet at some vertex t, and we leave our respective vertices at time 0, then we will meet at time $\max\{dist(p, t), dist(q, t)\}$. We can compute this meeting time for *all* vertices t by running Dijkstra's algorithm twice—once from p and once from q—and then scanning through the vertices. The overall running time is $O(E \log V)$.

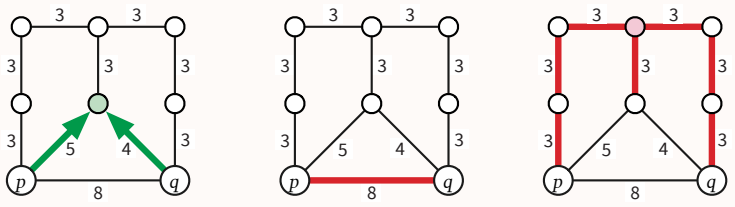
```
WhereToMeet(G, p, q):
  distp ← Dijkstra(G, p)      ⟨⟨distp(v) ← shortest-path distance from p to v⟩⟩
  distq ← Dijkstra(G, q)      ⟨⟨distq(v) ← shortest-path distance from q to v⟩⟩
  time ← ∞
  for all vertices t
      if time > max{distp(t), distq(t)}
          time ← max{distp(t), distq(t)}
          best ← t
  return best
```

■

Non-solution: When I last gave this problem on an exam, several students proposed the following alternate solutions, all of which are incorrect:

- ***Find the vertex t that minimizes $dist(p, t) + dist(q, t)$.*** This function is actually minimized for any vertex t on the shortest path from p to q—in particular, when $t = p$ or $t = q$.
- ***Find the midpoint of the shortest path from p to q.*** See the counterexample below. The best meeting point is the center vertex (at time 5), but the shortest path from p to q is a single edge; the midpoint of this path is not even a vertex.
- ***Find the vertex t that minimizes $|dist(p, t) - dist(q, t)|$.*** This function could be minimized on the moon if we can get there at *exactly* the same time. In the counterexample below, the difference in travel times is minimized at the top middle vertex (at time 9).
- ***Use the unique path from p to q in the minimum spanning tree.*** Shortest paths and minimum spanning trees don't have anything to do with each other.

The counterexample below shows the minimum spanning tree on the right.



To think about later:

3. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?

   Solution: $O(n \log n)$ — Dijkstra's algorithm runs in $O(E \log V)$ time, and a looped tree with n vertices has at most $2n$ edges. ■

   (b) Describe and analyze a faster algorithm.

   Solution: Let G be a looped tree, let T be the underlying binary tree, and let r be the root of T. Without loss of generality, assume we know which node is r (the only node with in-degree greater than 1) and which edges belong to T (the edges that don't lead to r).

   Our algorithm finds the shortest path from s to t by working backward from t. We first follow parent pointers, starting at t, until we reach either s or r.

   - If we reach s first, we've found the *only* path from s to t, so we're done.
   - If we reach r first, we must then find the shortest path from s to r. We compute this distance recursively as follows. For any node v, let $dist(v)$ denote the distance from v to r; this function obeys the following recurrence:

$$dist(v) = \begin{cases} w(v \to r) & \text{if } v \text{ is a leaf of } T \\ \min \left\{ \begin{array}{c} w(v \to left(v)) + dist(left(v)) \\ w(v \to right(v)) + dist(right(v)) \end{array} \right\} & \text{otherwise} \end{cases}$$

   Similarly, let $leaf(v)$ denote the unique leaf in T on the shortest path from v to r.

$$leaf(v) = \begin{cases} v & \text{if } v \text{ is a leaf of } T \\ leaf(left(v)) & \text{if } dist(v) = w(v \to left(v)) + dist(left(v)) \\ leaf(right(v)) & \text{otherwise} \end{cases}$$

   We can evaluate both of these functions at every vertex in $O(n)$ time using a postorder traversal of T. Then the shortest path from s to t consists of the unique path in T from s to $leaf(s)$ (which we can compute by following parent pointers upward from $leaf(s)$), followed by the edge $leaf(v) \to r$, followed by the unique path in T from r to t (which we've already computed).

   Altogether, the algorithm runs in ***O(n) time***. ■