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.

Solution: 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

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

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***. ■

2. 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*.

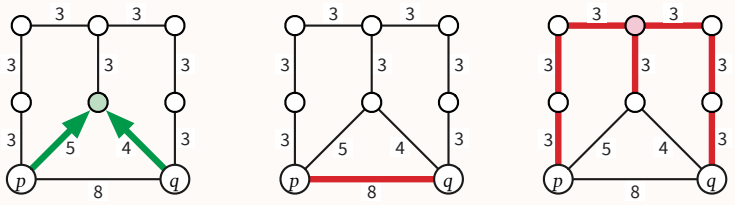
Solution: Let $dist(x, y)$ denote the shortest-path distance from vertex x to vertex y. If my friend and I decide to meet at some vertex t, and we leave our respective vertices at time 0, then we will meet at time $\max\{dist(p, t), dist(q, t)\}$. We can compute this meeting time for *all* vertices t by running Dijkstra's algorithm twice—once from p and once from q—and then find the earliest meeting time using a brute-force scan 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 Jeff Erickson last gave this problem on an exam, several students proposed each of the following alternate solutions. None of these are correct.

- ***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.


3 3
3 3 3
3 5 4 3
p 8 q
3 3
3 3 3
3 5 4 3
p 8 q
3 3
3 3 3
3 5 4 3
p 8 q

Harder problems 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:

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

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

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

   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***. ■