In the reading, we saw *Dijkstra's (Single-Source Shortest Path) Algorithm*. This takes as input a simple graph (which may be directed or undirected) that has *positive edge weights* and a designated vertex s, which is referred to as the start vertex. The output is a tree, consisting of every vertex in the original graph that is reachable from s and the set of edges such that, for every vertex v in the tree, the unique path from s to v in the tree is the shortest path between the two in the original graph.

Two key operations in Dijkstra's Algorithm were when we found a new short**er** path to a vertex v and when we wanted to know the shortest path we had yet to explore. Each of these can be implemented as a *priority queue operation*; if we use a binary heap for our priority queue and our input graph has n vertices and m edges, the total running time of the algorithm is $\mathcal{O}((m+n)\log n)$. A good exercise would be to look at the pseudo-code on the extra practice page (attached) and think about how you would translate this into a computer program, in particular where the priority queue operations go.

Question 1. In the reading, we saw the Dijkstra's Algorithm produces a correct single-source shortest path tree when the input edge weights are all non-negative. Suppose I have a graph $G = (V, E)$ where one or more edge weights are negative and I want to compute a single-source shortest path tree from a designated vertex s. Fortunately, the graph *does not* have any negative-cost cycles. I read the edges and discover the largest magnitude negative-cost edge has cost $-C$, for some positive number C. I observe that adding C to every edge's cost will make every edge cost non-negative. If I do that, and then call Dijkstra's Algorithm, will I get the correct single-source shortest path tree for the original input graph? Either explain why it will or give a counter-example to show it will not.

Question 2. Suppose I have a weighted directed graph $G = (V, E)$ and I wish to compute the shortest path from some vertex s to a designated target vertex t. However, *exactly one* edge in the graph has a negative weight. Fortunately, this does not cause the graph to have a *negative cost cycle*. Describe how I can still use Dijkstra's Algorithm to correctly find the shortest path in the original graph, from s to t, in the same asymptotic time that a single call to Dijkstra's Algorithm will take.

Hint: the correct shortest path either does or does not go through the edge with negative cost.

Question 3. ESCAPE! I am trapped on an island with velociraptors. I am currently at a safe house S and I want to get to a boat T that will allow me to get safely to land. The island has many paths that I can travel along, represented as a directed graph $G = (V, E)$. Unfortunately, each edge e in the graph has some probability $p(e)$ that a traveler on this edge will be eaten by a raptor. We assume these probabilities are independent of one another. If we select some path P from S to T on which to travel, I will arrive safely with probability $\Pi_{e \in P}(1 - p(e))$. I would like to select a path P which maximizes the probability of arriving safely.

Fortunately, someone has already implemented and tested Dijkstra, and we'd like to re-use that code. Unfortunately, we can't access the source code, so we can't just modify the algorithm to multiply edge weights instead of adding them. Instead, let's create a new set of edge weights w_e such that, if the original graph and these edge weights (instead of the given probabilities) are passed to the implemented Dijkstra's algortihm, the path it returns solves the problem here.

Give a linear-time algorithm that, given the graph and set of probabilities, will compute a set of edge weights that does so.

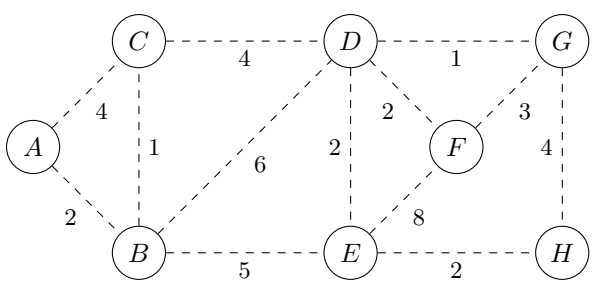
Tempting as it might be, please do not feed your teacher to hungry dinosaurs.

```
Dijkstra's (Single-Source, Shortest Path Tree) Algorithm
  for each vertex v do
    intree(v) = false
    parent(v) = N/A
    dist(v) = ∞
  end for
  dist(s) = 0
  while ∃ vertex u with intree(u) = false do
    u ← vertex with intree(u) = false and smallest dist(u)
    intree(u) = true
    for each vertex v ∈ adj[u] do
      if dist(v) > dist(u) + w(u, v) then
        dist(v) = dist(u) + w(u, v)
        parent(v) = u
      end if
    end for
  end while
```

This algorithm keeps track of three pieces of information for each vertex: whether or not it's in the shortest path tree so far, which vertex either is its parent in that tree, or would be if it were added right now (using the best path found so far), and what the distance from the start vertex to it is (along edges viewed so far).

If you wish to practice Dijkstra's Algorithm on some more graphs, here are a few you can use. The first one has a table that may help you keep track of the information kept for each node, but its use is not required. That table is not how a typical implementation of Dijkstra's Algorithm on a computer would proceed, but it is convenient for a person walking through the algorithm. The table is setup to use vertex A as a starting point, although you could trace the algorithm for other starting points as well.

Dijkstra's Algorithm is defined for both directed and undirected graphs; for more practice, if desired, you could direct the edges to create a directed graph.



v	intree(v)	parent(v)	dist(v)
A		N/A	0
B			
C			
D			
E			
F			
G			
H