Due Dates: One numbered question of your choice is due by **11:59PM Sunday, 23 October** and will be graded on completion only (1 point). Your full answers to all questions are due by **11:59PM Wednesday, 26 October**. Each solution will be graded for correctness and clarity (4 points per question).

At the top of your write-up for each problem, estimate the amount of time you spent on the problem, list your collaborators, and briefly describe the nature of your collaboration.

1. **Solving a recurrence**

   Suppose an algorithm's running time $T(n)$ satisfies

   $$T(n) \leq 4T(n/4) + cn$$

   $$T(4) = c$$

   for some $c > 0$.

   (a) Identify the tightest O-bound you can for T.

   (b) Prove your proposed bound directly using mathematical induction.

   Before embarking on your writeup for part (b), please refresh your memory of the conventional structure of induction proofs. Also, bear in mind my recommendation that most proofs should flow from assumptions towards conclusions and not the other way around.

2. **Minimum spanning trees** These questions are my effort to get you to take a good look at the proof of Kruskal's algorithm. Will it work? You can let me know.

   (a) Provide an example of a weighted graph G that has more than one MST.

   (b) Let G be any weighted graph that has two distinct MSTs T_1 and T_2. Let $e = \{u, v\}$ be an edge in T_1 that is not in T_2. Does the existence of this edge violate the *Cut Property* (claim 4.17 on page 145 of the textbook)? Explain why or why not.

   (c) You can view Kruskal's algorithm as starting with a graph $T = (V, E' = \emptyset)$ and gradually adding to the edge set E', one edge per iteration. During this process, the number of connected components can change from iteration to iteration. Does the number of connected components always increase, always decrease, or neither? Explain.

   (d) Imagine you are using Kruskal's algorithm to build T as in the previous question, and that you are considering edge $e = (u, v)$ for addition to E'. Show that e will create a cycle if and only if u and v are in the same connected component.

3. **Network routing and Dijkstra's algorithm**

   On a local computer network like the wifi at my house, the router has a pretty straightforward job. If one of the devices on the network wants to send a data packet to another device on the network, the router just accepts the packet from the originator and hands it to the destination. This happens, for example, whenever I want to print something–my laptop sends data to the wifi router, which passes the data along to the printer. If I want to go outside my local network (e.g., send a request for the Wikipedia page about goats or minimum spanning trees), my laptop creates a query packet, hands it to my router, and the router hands the packet to the cable modem in the basement, after which the router can forget about that packet. In

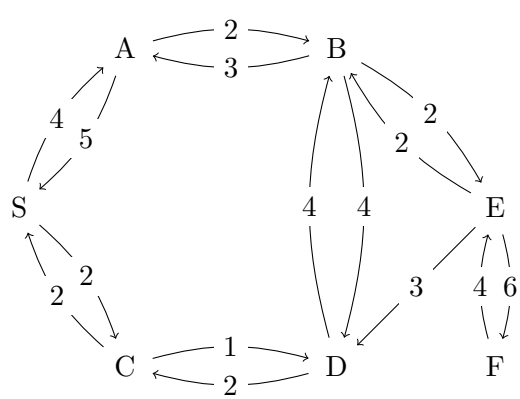
either situation, the wifi router is making a *routing decision*: given a packet's destination IP address, to which of my neighbor devices should I hand the packet? For the wifi router in my kitchen, the decision is easy–if the destination is one of the devices currently connected to the router, hand the packet directly to its intended recipient; otherwise hand it to the cable modem.

The high-capacity routers and switches that make up the regional, national, and international subnetworks of the internet are charged with moving a huge number of data packets toward their destinations. Unlike local networks, the routers on these more central networks have more complicated routing decisions to make, and they use more complicated routing algorithms to help them quickly send the packets in the right direction.

One category of routing algorithms is called link state algorithms/protocols. The idea of a link-state algorithm is that every node/router in the network contains a complete map of the current state of the entire network. This map is a weighted, directed graph. The weights typically represent *round-trip times* (also known as *ping times*). Any given node u can determine the weight of the edge connecting u to its neighbor v by sending a small packet (or *ping*) to v requesting that v just send the same packet back–the weight of the edge (u, v) is then recorded as the number of milliseconds that passed between u sending the ping and u receiving v's reply.

Once the router u has an up-to-date graph of the network, it can run Dijkstra's algorithm to determine all the shortest paths from u to each other node on the network. From this information, u can assemble a fast lookup table to answer the routing decision question: if this packet is bound for node x, to which of my neighbors should I hand the packet to get it to x as fast as possible?

Here's an example of a small network and the routing table maintained by the node S.



Destination	Give to	Distance
S	S	0
A	A	4
B	A	6
C	C	2
D	C	3
E	A	8
F	A	14

But wait, what happens when the network changes? Maybe there's a lot of traffic on one edge, so the round-trip time has gone up. Maybe somebody added a new node to the network. Maybe a backhoe cut the cable connecting two nodes so an edge disappears.

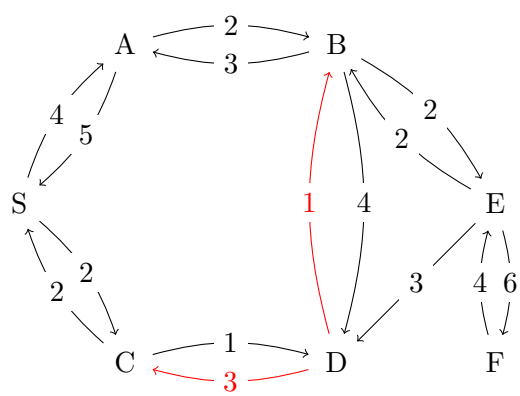
To handle the problem of a changing network, link-state protocols allow any node to send a *link-state update* to all the other nodes in the network. A link-state update from node D in the example shown above would consist of a list of the neighbors to which D has an outgoing edge, plus the new weights for those outgoing edges. (Note that D does not know

from personal observation what the weights on the incoming edges are, so D only reports on its outgoing edges.)

Suppose, for example, S in the graph above receives a link-state update from D that looks like this:

$$Update = (D, [B, C], [1, 3])$$

That is, D is saying that its outgoing neighbors are B and C, and the edges $D \to B$ and $D \to C$ now have weights 1 and 3 respectively. In response to this new information, S needs to change its graph and its routing table to adapt to the new conditions over at node D. As a result, the graph and routing table look like this:



Destination	Give to	Distance
S	S	0
A	A	4
B	C	4
C	C	2
D	C	3
E	C	6
F	C	12

What you're going to do

For this exercise, your job will be to develop an algorithm that a node can use to update its map of the network whenever it receives a link-state update.

Givens

- $G = (V, E)$, a directed graph, with weights $W : E \to \mathbb{R}^+$. This represents the current state of the computer network.
- Node $s \in V$. This represents the router we're focusing on.
- Tree $T_s = (V, P_s)$, where P_s consists of the directed edges computed by Dijkstra's algorithm starting at s.
- Link-state update $U = (v, B_v, W_v)$, where $v \in V$, $B_v = \{u \in V | (v, u) \in E\}$, and $W_v : B_v \to \mathbb{R}^+$. That is, this update is sent by v to inform s of the updated weights on all the edges exiting from v.

Goal

Describe an algorithm that updates G, W, and T_s based on the link-state update U. Present an argument that your algorithm produces a correct shortest-path-from-s tree after each update. Analyze the running time of your algorithm.

Some things to consider:

- Start from a baseline algorithm like this: use the information in U to modify E and W and then just rerun Dijkstra's algorithm from scratch. The question is whether (or under what conditions) you can do better than that.
- Your algorithm may or may not include adding new data structures to go with G and T_s if they will help make your updating algorithm more efficient. If you add new structures, you'll need to also talk about how they will be modified for each link-state update.
- If you want an extra challenge, you can remove the assumption that $v \in V$–that is, you can allow a link-state update to add nodes to the network.