Due: Wednesday 9/26 before 9:15

1. Consider an undirected graph with two weights associated with each edge, w_1 and w_2. The first weight $w_1(e)$ represents the time it takes to construct the edge e. The second weight $w_2(e)$ represents the cost of the construction of e. The goal is to construct a network connecting all the vertices of the graph. Any number of edges can be constructed in parallel. The primary objective is to minimize the construction time (remember, any number of edges can be constructed in parallel). The secondary objective is to minimize the cost of the network. Describe an efficient algorithm that satisfies *both* objectives. In other words, give an algorithm that finds a spanning tree that can be constructed in minimum possible time such that it has the smallest weight among all trees that can be constructed in this minimum time. Prove that your algorithm is correct.

2. Suppose you are given a diagram of a telephone network, which is a graph G whose vertices represent switching centers, and whose edges represent communication lines between two centers. Each edge e is labeled by its bandwidth $b(e)$. The bandwidth of a path is defined as the bandwidth of its lowest-bandwidth edge. Give an efficient algorithm that, given a network diagram and two switching centers a and b, will output the maximum bandwidth of a path between a and b.

3. **(4.26)** One of the first things you learn in calculus is how to minimize a differentiable function such as $y = ax^2 + bx + c$, where $a > 0$. The minimum spanning tree problem, on the other hand, is a minimization problem of a very different flavor: there are now just a finite number of possibilities for how the minimum might be achieved—rather than a continuum of possibilities—and we are interested in how to perform the computation without having to exhaust this (huge) finite number of possibilities.

   One can ask what happens when these two minimization issues are brought together, and the following question is an example of this. Suppose we have a connected graph $G = (V, E)$. Each edge e now has a *time-varying edge cost* given by a function $f_e : \mathbb{R} \to \mathbb{R}$. Thus, at time t, it has cost $f_e(t)$. We'll assume that all these functions are positive over their entire range. Observe that the set of edges constituting the minimum spanning tree of G may change over time. Also, of course, the cost of the minimum spanning tree of G becomes a function of the time t; we'll denote this function $c_G(t)$. A natural problem then becomes: find a value of t at which $c_G(t)$ is minimized.

   Suppose each functio $f_e(t)$ is a polynomial of degree 2: $f_e(t) = a_e t^2 + b_e t + c_e$, where $a_e > 0$. Give an algorithm that takes the graph G and the values $\{(a_e, b_e, c_e) \mid e \in E\}$ and returns a value of the time t at which the minimum spanning tree has minimum cost. Your algorithm should run in time polynomial in the number of nodes and edges of the graph G. You may assume that arithmetic operations on the number $\{(a_e, b_e, c_e)\}$ can be done in constant time per operation.

4. **(4.28)** Suppose you're a consultant for the networking company CluNet, and they have the following problem. The network that they're currently working on is modeled by a

connected graph $G = (V, E)$ with n nodes. Each edge e is a fiber-optic cable that is owned by one of two companies—creatively named X and Y—and leased to CluNet.

CluNet's plan is to choose a spanning tree T of G and upgrade the links corresponding to the edges of T. Their business relations people have already concluded an arrangement with companies X and Y stipulating a number k such that in the tree T that is chosen, k of the edges will be owned by X and $n - k$ of the edges will be owned by Y.

CluNet management now faces the following problem. It is not at all clear to them whether there even *exists* a spanning tree T meeting these conditions, or how to find one if it does exist. So this is the problem they put to you: give a polynomial-time algorithm that takes G, with each edge labeled X or Y, and either **(1)** returns a spanning tree with exactly k edges labeled X, or **(2)** reports correctly that no such tree exists.

5. **[Implementation Question]** Implement Dijkstra's algorithm in python, using the library module `heapq` to handle the priority queue. More precisely, write a function `dijkstra` that takes as arguments a graph G and source vertex s, and outputs shortest paths from s to all other vertices of G.

   The graph G will be represented by a triple (n, E, c), where V is the number of vertices in G, E a list of adjacency lists representing the edges of G (so that, for example $E[1] = [3, 4, 5]$ means that there are directed edges in G from vertex 1 to exactly the three vertices 3, 4, and 5), and c a dictionary specifying the cost of each edge, so that, for example $c[(1, 3)]$ is the value of the edge from 1 to 3. A small example of a graph in this format will be given in the code directory of the course website.

   Your implementation should use the library module `heapq` to handle the priority queue. You should do this without defining any new classes (we'll deal with a more general problem later and there some class definitions will be necessary). Important facts in this will be **(1)** the heap module uses a standard comparison, and **(2)** tuples can be compared in python, and the comparison is defined to work: if (a, b) and (c, d) are two tuples, then $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$. While accessing the heap, you should only use the functions of the `heapq` module, even though the actual python type of the heap object is the ordinary list.

   The function `dijkstra` should return a list of $|V|$ lists; the i-th list will consist of the vertices on the shortest s-i path found by the algorithm. Example:

```
[ [2,  1,  4,  0], [2, 1], [2], [2,  1,  3], [2,  1,  4] ]
```

   One problem you'll have (since you cannot create new classes) is that there seems to be no way to implement a `decrease-key` operation. This is because the heap is just a list, and you cannot quickly locate a heap element by name to change its value. Hint: whenever the key of an element changes, add a new copy of the element to the heap. The smallest one will be extracted first, and the later ones can be safely ignored.

   A final request: when importing the functions from the `heapq` module write, for example,

```
from heapq import heapify, heappop, heappush
```