Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

4. Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges—what's being bought—are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them—for example, they may want to know if the four events

```
buy Yahoo, buy eBay, buy Yahoo, buy Oracle
```

occur in this sequence S, in order but not necessarily consecutively.

They begin with a collection of possible *events* (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., `Yahoo` stock may be bought many times in a single sequence S). We will say that a sequence S' is a *subsequence* of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence

```
buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo,
buy Oracle
```

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events—S' of length m and S of length n, each possibly containing an event more than once—and decides in time $O(m + n)$ whether S' is a subsequence of S.

5. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

The Problem. Given a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, determine whether there exists a valid schedule.

Example. Suppose we have $n = 3$ streams, with

$$(b_1, t_1) = (2000, 1), \quad (b_2, t_2) = (6000, 2), \quad (b_3, t_3) = (2000, 1),$$

and suppose the link's parameter is $r = 5000$. Then the schedule that runs the streams in the order $1, 2, 3$, is valid, since the constraint $(*)$ is satisfied:

$t = 1$: the whole first stream has been sent, and $2000 < 5000 \cdot 1$
$t = 2$: half of the second stream has also been sent,
and $2000 + 3000 < 5000 \cdot 2$
Similar calculations hold for $t = 3$ and $t = 4$.

(a) Consider the following claim:

Claim: There exists a valid schedule if and only if each stream i satisfies $b_i \leq rt_i$.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

(b) Give an algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, and determines whether there exists a valid schedule. The running time of your algorithm should be polynomial in n.

13. A small business—say, a photocopying service with a single large machine—faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer i's job will take t_i time to complete. Given a schedule (i.e., an ordering of the jobs), let C_i denote the finishing time of job i. For example, if job j is the first to be done, we would have $C_j = t_j$; and if job j is done right after job i, we would have $C_j = C_i + t_j$. Each customer i also has a given weight w_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i's job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Example. Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$, while the second job takes time $t_2 = 3$ and has weight

$w_2 = 2$. Then doing job 1 first would yield a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the second job first would yield the larger weighted completion time of $10 \cdot 4 + 2 \cdot 3 = 46$.

14. You're working with a group of security consultants who are helping to monitor a large computer system. There's particular interest in keeping track of processes that are labeled "sensitive." Each such process has a designated start time and finish time, and it runs continuously between these times; the consultants have a list of the planned start and finish times of all sensitive processes that will be run that day.

As a simple first step, they've written a program called `status_check` that, when invoked, runs for a few seconds and records various pieces of logging information about all the sensitive processes running on the system at that moment. (We'll model each invocation of `status_check` as lasting for only this single point in time.) What they'd like to do is to run `status_check` as few times as possible during the day, but enough that for each sensitive process P, `status_check` is invoked at least once during the execution of process P.

(a) Give an efficient algorithm that, given the start and finish times of all the sensitive processes, finds as small a set of times as possible at which to invoke `status_check`, subject to the requirement that `status_check` is invoked at least once during each sensitive process P.

(b) While you were designing your algorithm, the security consultants were engaging in a little back-of-the-envelope reasoning. "Suppose we can find a set of k sensitive processes with the property that no two are ever running at the same time. Then clearly your algorithm will need to invoke `status_check` at least k times: no one invocation of `status_check` can handle more than one of these processes."

This is true, of course, and after some further discussion, you all begin wondering whether something stronger is true as well, a kind of converse to the above argument. Suppose that k^* is the largest value of k such that one can find a set of k sensitive processes with no two ever running at the same time. Is it the case that there must be a set of k^* times at which you can run `status_check` so that some invocation occurs during the execution of each sensitive process? (In other words, the kind of argument in the previous paragraph is really the only thing forcing you to need a lot of invocations of `status_check`.) Decide whether you think this claim is true or false, and give a proof or a counterexample.

edge $e = (v, w)$ connecting two sites v and w, and given a proposed starting time t from location v, the site will return a value $f_e(t)$, the predicted arrival time at w. The Web site guarantees that $f_e(t) \geq t$ for all edges e and all times t (you can't travel backward in time), and that $f_e(t)$ is a monotone increasing function of t (that is, you do not arrive earlier by starting later). Other than that, the functions $f_e(t)$ may be arbitrary. For example, in areas where the travel time does not vary with the season, we would have $f_e(t) = t + \ell_e$, where ℓ_e is the time needed to travel from the beginning to the end of edge e.

Your friends want to use the Web site to determine the fastest way to travel through the directed graph from their starting point to their intended destination. (You should assume that they start at time 0, and that all predictions made by the Web site are completely correct.) Give a polynomial-time algorithm to do this, where we treat a single query to the Web site (based on a specific edge e and a time t) as taking a single computational step.

19. A group of network designers at the communications company CluNet find themselves facing the following problem. They have a connected graph $G = (V, E)$, in which the nodes represent sites that want to communicate. Each edge e is a communication link, with a given available bandwidth b_e.

For each pair of nodes $u, v \in V$, they want to select a single u-v path P on which this pair will communicate. The *bottleneck rate* $b(P)$ of this path P is the minimum bandwidth of any edge it contains; that is, $b(P) = \min_{e \in P} b_e$. The *best achievable bottleneck rate* for the pair u, v in G is simply the maximum, over all u-v paths P in G, of the value $b(P)$.

It's getting to be very complicated to keep track of a path for each pair of nodes, and so one of the network designers makes a bold suggestion: Maybe one can find a spanning tree T of G so that for *every* pair of nodes u, v, the unique u-v path in the tree actually attains the best achievable bottleneck rate for u, v in G. (In other words, even if you could choose any u-v path in the whole graph, you couldn't do better than the u-v path in T.)

This idea is roundly heckled in the offices of CluNet for a few days, and there's a natural reason for the skepticism: each pair of nodes might want a very different-looking path to maximize its bottleneck rate; why should there be a single tree that simultaneously makes everybody happy? But after some failed attempts to rule out the idea, people begin to suspect it could be possible.

Show that such a tree exists, and give an efficient algorithm to find one. That is, give an algorithm constructing a spanning tree T in which, for each $u, v \in V$, the bottleneck rate of the u-v path in T is equal to the best achievable bottleneck rate for the pair u, v in G.

20. Every September, somewhere in a far-away mountainous part of the world, the county highway crews get together and decide which roads to keep clear through the coming winter. There are n towns in this county, and the road system can be viewed as a (connected) graph $G = (V, E)$ on this set of towns, each edge representing a road joining two of them. In the winter, people are high enough up in the mountains that they stop worrying about the *length* of roads and start worrying about their *altitude*—this is really what determines how difficult the trip will be.

So each road—each edge e in the graph—is annotated with a number a_e that gives the altitude of the highest point on the road. We'll assume that no two edges have exactly the same altitude value a_e. The *height* of a path P in the graph is then the maximum of a_e over all edges e on P. Finally, a path between towns i and j is declared to be *winter-optimal* if it achieves the minimum possible height over all paths from i to j.

The highway crews are going to select a set $E' \subseteq E$ of the roads to keep clear through the winter; the rest will be left unmaintained and kept off limits to travelers. They all agree that whichever subset of roads E' they decide to keep clear, it should have the property that (V, E') is a connected subgraph; and more strongly, for every pair of towns i and j, the height of the winter-optimal path in (V, E') should be no greater than it is in the full graph $G = (V, E)$. We'll say that (V, E') is a *minimum-altitude connected subgraph* if it has this property.

Given that they're going to maintain this key property, however, they otherwise want to keep as few roads clear as possible. One year, they hit upon the following conjecture:

The minimum spanning tree of G, with respect to the edge weights a_e, is a minimum-altitude connected subgraph.

(In an earlier problem, we claimed that there is a unique minimum spanning tree when the edge weights are distinct. Thus, thanks to the assumption that all a_e are distinct, it is okay for us to speak of *the* minimum spanning tree.)

Initially, this conjecture is somewhat counterintuitive, since the minimum spanning tree is trying to minimize the *sum* of the values a_e, while the goal of minimizing altitude seems to be asking for a fairly different thing. But lacking an argument to the contrary, they begin considering an even bolder second conjecture: