1 Stable Matching

Note: Exercises denoted with an asterisk ($*$) tend to be more difficult, or to rely on some of the more advanced material.

1. Gale and Shapley published their paper on the stable marriage problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

   Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals.

   The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

   We say that an assignment of students to hospitals is *stable* if neither of the following situations arises.

   - First type of instability: There are students s and s', and a hospital h, so that
     - s is assigned to h, and
     - s' is assigned to no hospital, and
     - h prefers s' to s.
   - Second type of instability: There are students s and s', and hospitals h and h', so that
     - s is assigned to h, and
     - s' is assigned to h', and
     - h prefers s' to s, and
     - s' prefers h to h'.

   So we basically have the stable marriage problem from class, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students.

   Show that there is always a stable assignment of students to hospitals, and give an efficient algorithm to find one. The input size is $\Theta(mn)$; ideally, you would like to find an algorithm with this running time.

   Solution. The algorithm is very similar to the one from class. At any point in time, a student is either "committed" to a hospital or "free." A hospital either has available positions, or it is "full." The algorithm is the following:

While some hospital h_i has available positions h_i offers a position to the next student s_j on its preference list if s_j is free then s_j accepts the offer else (s_j is already committed to a hospital h_k) if s_j prefers h_k to h_i then s_j remains committed to h_k else s_j becomes committed to h_i the number of available positions at h_k increases by one. the number of available positions at h_i decreases by one.

The algorithm terminates in $O(mn)$ steps because each hospital offers a positions to a student at most once, and in each iteration, some hospital offers a position to some student.

Suppose there are $p_i > 0$ positions available at hospital h_i. The algorithm terminates with an assignment in which all available positions are filled, because any hospital that did not fill all its positions must have offered one to every student; but then, all these students would be committed to some hospital, which contradicts our assumption that $\sum_{i=1}^{m} p_i < n$.

Finally, we want to argue that the assignment is stable. For the first kind of instability, suppose there are students s and s', and a hospital h as above. If h prefers s' to s, then h would have offered a position to s' before it offered one to s; from then on, s' would have a position at *some* hospital, and hence would not be free at the end — a contradiction.

For the second kind of instability, suppose that (h_i, s_j) is a pair that causes instability. Then h_i must have offered a position to s_j, for otherwise it has p_i residents all of whom it prefers to s_j. Moreover, s_j must have rejected h_i in favor of some h_k which he/she preferred; and s_j must therefore be committed to some h_ℓ (possibly different from h_k) which he/she also prefers to h_i.

2. We can think about a different generalization of the stable matching problem, in which certain man-woman pairs are explicitly *forbidden*. In the case of employers and applicants, picture that certain applicants simply lack the necessary qualifications or degree; and so they cannot be employed at certain companies, however desirable they may seem. Concretely, we have a set M of n men, a set W of n women, and a set $F \subseteq M \times W$ of pairs who are simply *not allowed* to get married. Each man m ranks all the women w for which $(m, w) \notin F$, and each woman w' ranks all the men m' for which $(m', w') \notin F$.

   In this more general setting, we say that a matching S is *stable* if it does not exhibit any of the following types of instability.

   (i) There are two pairs (m, w) and (m', w') in S with the property that m prefers w' to w, and w' prefers m to m'. *(The usual kind of instability.)*

   (ii) There is a pair $(m, w) \in S$, and a man m', so that m' is not part of any pair in the matching, $(m', w) \notin F$, and w prefers m' to m. *(A single man is more desirable and not forbidden.)*

(ii′) There is a pair $(m, w) \in S$, and a woman w', so that w' is not part of any pair in the matching, $(m, w') \notin F$, and m prefers w' to w. *(A single woman is more desirable and not forbidden.)*

(iii) There is a man m and a woman w, neither of which is part of any pair in the matching, so that $(m, w) \notin F$. *(There are two single people with nothing preventing them from getting married to each other.)*

Note that under these more general definitions, a stable matching need not be a perfect matching.

Now we can ask: for every set of preference lists and every set of forbidden pairs, is there always a stable matching? Resolve this question by doing one of the following two things: (a) Giving an algorithm that, for any set of preference lists and forbidden pairs, produces a stable matching; or (b) Giving an example of a set of preference lists and forbidden pairs for which there is no stable matching.

Solution. There is always a stable matching, even in the general model with forbidden pairs. The most direct way to prove this is through an algorithm that is almost the same as the one we used for the basic stable matching problem in class; indeed, the only change is that our condition for the *While* loop will say: "While there is a man m who is free and hasn't proposed to every woman w for which $(m, w) \notin F$." Here is the algorithm in full:

```
Initially all $m \in M$ and $w \in W$ are free
While there is a man $m$ who is free and hasn't proposed to every woman $w$ for which $(m, w) \notin F$
    Choose such a man $m$
    Let $w$ be the highest-ranked woman in $m$'s preference list to which $m$ has not yet proposed
    If $w$ is free then
        $(m, w)$ become engaged
    Else $w$ is currently engaged to $m'$
        If $w$ prefers $m'$ to $m$ then
            $m$ remains free
        Else $w$ prefers $m$ to $m'$
            $(m, w)$ become engaged
            $m'$ becomes free
        Endif
    Endif
Endwhile
Return the set $S$ of engaged pairs
```

Now, facts (1.1) - (1.3) from the book remain true; and we don't have to worry about establishing that the resulting matching S is perfect (indeed, it may not be). We also notice an additional pairs of facts. If m is a man who is not part of a pair in S, then m must have proposed to every non-forbidden woman; and if w is a woman who is not part of a pair in S, then it must be that no man ever proposed to w.

Finally, we need only show

- There is no instability with respect to the returned matching S.

Our general definition of instability has four parts; this means that we have to make sure none of the four bad things happens.

First, suppose that there is an instability of type (i), consisting of pairs (m, w) and (m', w') in S with the property that m prefers w' to w, and w' prefers m to m'. It follows that m must have proposed to w'; so w' rejected m, and thus she prefers her final partner to m — a contradiction. Next, suppose there is an instability of type (ii), consisting of a pair $(m, w) \in S$, and a man m', so that m' is not part of any pair in the matching, $(m', w) \notin F$, and w prefers m' to m. Then m' must have proposed to w and been rejected; again, it follows that w prefers her final partner to m' — a contradiction. Third, suppose there is an instability of type (ii′), consisting of a pair $(m, w) \in S$, and a woman w', so that w' is not part of any pair in the matching, $(m, w') \notin F$, and m prefers w' to w. Then no man proposed to w' at all; in particular, m never proposed to w', and so he must prefer w to w' — a contradiction. And finally, suppose there is an instability of type (iv), consisting of a man m and a woman w, neither of which is part of any pair in the matching, so that $(m, w) \notin F$. But for m to be single, he must have proposed to every non-forbidden woman; in particular, he must have proposed to w, whence she would not be single — a contradiction.

As in the proof of (1.3), there can be at most n^2 proposals; and using the implementation in Section 1.1, the whole algorithm can be made to run in time $O(n^2)$.

3. Consider a town with n men and n women seeking to get married to one another. Each man has a preference list that ranks all the women, and each woman has a preference list that ranks all the men.

   The set of all $2n$ people is divided into two categories: *good* people and *bad* people. Suppose that for some number k, $1 \leq k \leq n-1$, there are k good men and k good women; thus there are $n-k$ bad men and $n-k$ bad women.

   Everyone would rather marry any good person than any bad person. Formally, each preference list has the property that it ranks each good person of the opposite gender higher than each bad person of the opposite gender: its first k entries are the good people (of the opposite gender) in some order, and its next $n-k$ are the bad people (of the opposite gender) in some order.

   (a) Show that there **exists** a stable matching in which every good man is married to a good woman.

   (b) Show that in **every** stable matching, every good man is married to a good woman.

   Solution. **(a)** Here are two possible solutions.

   (i) We show that the stable matching produced by the proposal algorithm from class has this property. Suppose, by way of contradiction, that it does not — then there is a good man m who ends up married to a bad woman w. Since m ranks every good woman higher than w, he must have been rejected by every good woman in the execution of the algorithm. Consider a good woman w': at some moment, she rejects m, and we proved in class that her sequence of engaged partners gets better and better from this point onward. Thus w' ends up married to a man she ranks higher than m. Since only good men are ranked higher than m, w' ends up married to a good man. Since this holds for an arbitrary good woman, we conclude that each good woman ends up married to a good man. But this implies there are $k+1$ good men — one married to each good woman, plus m — and this is a contradiction.

(ii) We apply the stable matching algorithm to the good men and good women in isolation, obtaining a stable matching M'. We do the same for the bad men and bad women, obtaining a stable matching M''. Let $M = M' \cup M''$: we simply glue these two matchings together. M is a perfect matching, since each person is part of exactly one pair. If there were an instability (m, w), then one of m or w must be good and the other one must be bad — since M' and M'' by themselves are stable. Suppose that m is good — the case in which w is good is completely analogous. Then m prefers his current married partner to w, since his current married partner is good and w is bad — but this contradicts our assumption that (m, w) was an instability. Thus, M has no instabilities, so it is a stable matching in which every good person is married to another good person.

(b) Suppose that some stable matching does not have the desired property — there is a good man m married to a bad woman w. Since there are k good men and k good women, and one of the good men is married to a bad woman, it follows that at least one good woman w' is married to a bad man m'. Now consider the pair (m, w). Each is good, but is married to a bad partner. Thus, each prefers the other to their current partner, and hence they are an instability. This contradicts our assumption that the matching we considered was stable.

4. $(*)$ For this problem, we will explore the issue of *truthfulness* in the stable matching problem, and specifically in the Gale-Shapley algorithm. The basic question is: Can a man or a woman end up better off by lying about his or her preferences? More concretely, we suppose each participant has a true preference order. Now consider a woman w. Suppose w prefers man m to m', but both m and m' are low on her list of preferences. Can it be the case that by switching the order of m and m' on her list of preferences (i.e., by falsely claiming that she prefers m' to m) and running the algorithm with this false preference list, w will end up with a man m'' that she truly prefers to both m and m'? (We can ask the same question for men, but will focus on the case of women for purposes of this question.)

   Resolve this questions by doing one of the following two things:

   (a) Giving a proof that, for any set of preference lists, switching the order of a pair on the list cannot improve a woman's partner in the Gale-Shapley algorithm; or

   (b) Giving an example of a set of preference lists for which there is a switch that would improve the partner of a woman who switched preferences.

5. There are many other settings in which we can ask questions related to some type of "stability" principle. Here's one, involving competition between two enterprises.

   Suppose we have two television networks; let's call them AOL-Time-Warner-CNN and Disney-ABC-ESPN, or $\mathcal{A}$ and $\mathcal{D}$ for short. There are n prime-time programming slots, and each network has n TV shows. Each network wants to devise a *schedule* — an assignment of each show to a distinct slot — so as to attract as much market share as possible.

Here is the way we determine how well the two networks perform relative to each other, given their schedules. Each show has a fixed *Nielsen rating*, which is based on the number of people who watched it last year; we'll assume that no two shows have exactly the same rating. A network *wins* a given time slot if the show that it schedules for the time slot has a larger rating than the show the other network schedules for that time slot. The goal of each network is to *win* as many time slots as possible.

Suppose in the opening week of the fall season, Network $\mathcal{A}$ reveals a schedule S and Network $\mathcal{D}$ reveals a schedule T. On the basis of this pair of schedules, each network wins certain of the time slots, according to the rule above. We'll say that the pair of schedules (S, T) is *stable* if neither network can unilaterally change its own schedule and win more time slots. That is, there is no schedule S' so that Network $\mathcal{A}$ wins more slots with the pair (S', T) than it did with the pair (S, T); and symmetrically, there is no schedule T' so that Network $\mathcal{D}$ wins more slots with the pair (S, T') than it did with the pair (S, T).

The analogue of Gale and Shapley's question for this kind of stability is: For every set of TV shows and ratings, is there always a stable pair of schedules? Resolve this question by doing one of the following two things: (a) Giving an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or (b) Giving an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

Solution. There is not always a stable pair of schedules. Suppose Network $\mathcal{A}$ has two shows $\{a_1, a_2\}$ with ratings 20 and 40; and Network $\mathcal{D}$ has two shows $\{d_1, d_2\}$ with ratings 10 and 30.

Each network can reveal one of two schedules. If in the resulting pair, a_1 is paired against d_1, then Network $\mathcal{D}$ will want to switch the order of the shows in its schedule (so that it will win one slot rather than none). If in the resulting pair, a_1 is paired against d_2, then Network $\mathcal{A}$ will want to switch the order of the shows in its schedule (so that it will win two slots rather than one).

6. Peripatetic Shipping Lines, Inc., is a shipping company that owns n ships, and provides service to n ports. Each of its ships has a *schedule* which says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has m days, for some $m > n$.) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

   (†) No two ships can be in the same port on the same day.

   The company wants to perform maintenance on all the ships this month, via the following scheme. They want to *truncate* each ship's schedule: for each ship S_i, there will be some day when it arrives in its scheduled port and simply remains there for rest of the month (for maintenance). This means that S_i will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the *truncation* of S_i's schedule will simply consist of its original schedule up to a certain specified day on which it is in a port P; the remainder of the truncated schedule simply has it remain in port P.

Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (†) continues to hold: no two ships are ever in the same port on the same day.

Show that such a set of truncations can always be found, and give an efficient algorithm to find them.

Example: Suppose we have two ships and two ports, and the "month" has four days. Suppose the first ship's schedule is

port P_1; at sea; port P_2; at sea

and the second ship's schedule is

at sea; port P_1; at sea; port P_2

Then the (only) way to choose truncations would be to have the first ship remain in port P_2 starting on day 3, and have the second ship remain in port P_1 starting on day 2.

Solution. For each schedule, we have to choose a *stopping port*: the port in which the ship will spend the rest of the month. Implicitly, these stopping ports will define truncations of the schedules. We will say that an assignment of ships to stopping ports is *acceptable* if the resulting truncations satisfy the conditions of the problem — specifically, condition (†). (Note that because of condition (†), each ship must have a distinct stopping port in any acceptable assignment.)

We set up a stable marriange problem involving ships and ports. Each ship ranks each port in chronological order of its visits to them. Each port ranks each ship in reverse chronological order of their visits to it. Now we simply have to show:

Fact 1.1 *A stable matching between ships and ports defines an acceptable assignment of stopping ports.*

Proof. If the assignment is not acceptable, then it violates condition (†). That is, some ship S_i passes through port P_k after ship S_j has already stopped there. But in this case, under our preference relation above, ship S_i "prefers" P_k to its actual stopping port, and port P_k "prefers" ship S_i to ship S_j. This contradicts the assumption that we chose a stable matching between ships and ports. ■

7. Some of your friends are working for CluNet, a builder of large communication networks, and they are looking at algorithms for switching in a particular type of input/output crossbar.

Here is the set-up. There are n *input wires* and n *output wires*, each directed from a *source* to a *terminus*. Each input wire meets each output wire in exactly one distinct point, at a special piece of hardware called a *junction box*. Points on the wire are naturally ordered in the direction from source to terminus; for two distinct points x and y on the same wire, we say that x is *upstream* from y if x is closer to the source than y, and otherwise we say x is *downstream* from y. The order in which one input wire meets the output wires is not necessarily the same as the order in which another input wire meets the output wires. (And similarly for the orders in which output wires meet input wires.)

Now, here's the switching component of this situation. Each input wire is carrying a distinct data stream, and this data stream must be *switched* onto one of the output wires. If the stream of Input i is switched onto Output j, at junction box B, then this stream passes through all junction boxes *upstream* from B on Input i, then through B, then through all junction boxes *downstream* from B on Output j. It does not matter which input data stream gets switched onto which output wire, but each input data stream must be switched onto a *different* output wire. Furthermore — and this is the tricky constraint — no two data streams can pass through the same junction box following the switching operation.

Finally, here's the question. Show that for any specified pattern in which the input wires and output wires meet each other (each pair meeting exactly once), a valid switching of the data streams can always be found — one in which each input data stream is switched onto a different output, and no two of the resulting streams pass through the same junction box. Additionally, give an efficient algorithm to find such a valid switching. (The accompanying figure gives an example with its solution.)

Solution. A *switching* consists precisely of a perfect matching between input wires and output wires — we simply need to choose which input stream will be switched onto which output wire.

¿From the point of view of an input wire, it wants its data stream to be switched as early (close to the source) as possible: this minimizes the risk of running into another data stream, that has already been switched, at a junction box. ¿From the point of view of an output wire, it wants a data stream to be switched onto it as late (far from the source) as possible: this minimizes the risk of running into another data stream, that has not yet been switched, at a junction box.

Motivated by this intuition, we set up a stable marriage problem involving the input wires and output wires. Each input wire ranks the output wires in the order it encounters them from source to terminus; each output wire ranks the input wires in the reverse of the order it encounters them from source to terminus. Now we show:

- A stable matching between input and output wires defines a valid switching.

To prove this, suppose that this switching causes two data streams to cross at a junction box. Suppose that the junction box is at the meeting of Input i and Output j. Then

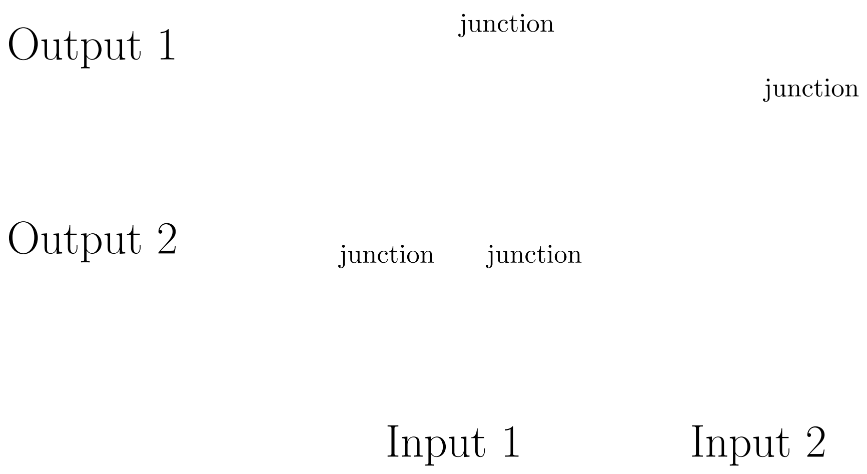


Figure 1: An example with two input wires and two output wires. Input 1 has its junction with Output 2 upstream from its junction with Output 1; Input 2 has its junction with Output 1 upstream from its junction with Output 2. A valid solution is to switch the data stream of Input 1 onto Output 2, and the data stream of Input 2 onto Output 1. On the other hand, if the stream of Input 1 were switched onto Output 1, and the stream of Input 2 were switched onto Output 2, then both streams would pass through the junction box at the meeting of Input 1 and Output 2 — and this is not allowed.

one stream must be the one that originates on Input i; the other stream must have switched from a different input wire, say Input k, onto Output j. But in this case, Output j prefers Input i to Input k (since j meets i downstream from k); and Input i prefers Output j to the output wire, say Output ℓ, onto which it is actually switched — since it meets Output j upstream from Output ℓ. This contradicts the assumption that we chose a stable matching between input wires and output wires.

Assuming the meetings of inputs and outputs are represented by lists containing the orders in which each wire meets the other wires, we can set up the preference lists for the stable matching problem in time $O(n^2)$. Computing the stable matching then takes an additional $O(n^2)$ time.

2 Algorithmic Primitives for Graphs

1. Inspired by the example of that great Cornellian, Vladimir Nabokov, some of your friends have become amateur lepidopterists. (They study butterflies.) Often when they return from a trip with specimens of butterflies, it is very difficult for them to

tell how many distinct species they've caught — thanks to the fact that many species look very similar to one another.

One day they return with n butterflies, and they believe that each belongs to one of two different species, which we'll call A and B for purposes of this discussion. They'd like to divide the n specimens into two groups — those that belong to A, and those that belong to B — but it's very hard for them to directly label any one specimen. So they decide to adopt the following approach.

For each pair of specimens i and j, they study them carefully side-by-side; and if they're confident enough in their judgment, then they label the pair (i, j) either "same" (meaning they believe them both to come from the same species) or "different" (meaning they believe them to come from opposite species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair *ambiguous*.

So now they have the collection of n specimens, as well as a collection of m judgments (either "same" or "different") for the pairs that were not declared to be ambiguous. They'd like to know if this data is consistent with the idea that each butterfly is from one of species A or B; so more concretely, we'll declare the m judgments to be *consistent* if it is possible to label each specimen either A or B in such a way that for each pair (i, j) labeled "same," it is the case that i and j have the same label; and for each pair (i, j) labeled "different," it is the case that i and j have opposite labels. They're in the middle of tediously working out whether their judgments are consistent, when one of them realizes that you probably have an algorithm that would answer this question right away.

Give an algorithm with running time $O(m + n)$ that determines whether the m judgments are consistent.

Solution. Given a set of n specimens and m judgements, we need to determine if the set of judgements are consistent. To be able to label each specimen either A or B, we construct an undirected graph $G = (V, E)$ as follows: Each specimen is a vertex. There is an edge between v_i and v_j if there is a judgement involving the corresponding specimens.

Once the graph is constructed, arbitrarily designate some vertex as the starting node s. Note that the graph G need not be connected in which case we will need starting nodes for each component. For each component G_i (with starting node s_i) of G label the specimen associated with s_i A. Now perform Breadth-First Search on G_i starting at s_i. For each node v_k that is visited from v_j, consider the judgement made on the specimens corresponding to (v_j, v_k). If the judgement was "same," label v_k the same as v_j and if the judgement was "different," label v_k the opposite of v_j. Note that there may be some specimens that are not associated with any judgements. These specimens maybe labeled arbitrarily, but we shall label them A. Once the labeling is complete we may go through the list of judgements to check for consistency. More precisely (Refer to pg. 39 of the text)

For each component C of G designate a starting node s and label it A Mark s as "Visited." Ititialize R=s. Define layer L(0)=s. For i=0,1,2,... For each node u in L(i) Consider each edge (u,v) incident to v If v is not marked "Visited" (then v is also not labeled) Mark v "Visited" If the judgement (u,v) was "same" then label v the same as u else (the judgement was "different") label v the opposite of u Endif Add v to the set R and ro layer L(i+1) Endif Endfor Endfor Endfor For each edge (u,v) (for each judgement (u,v)) If the judgement was "same" If u and v have different labels there is an inconsistency Endif Else (the judgement was "different") If u and v have the same labels there is an inconsistency Endif Endif Endfor

First note that the running time of this algorithm is $O(m+n)$: Constructing G takes $O(m+n)$ since it has n vertices and m edges. Performing BFS on G takes $O(m+n)$ and going through the list of judgements to check consistency takes $O(m)$. Thus the running time is $O(m+n)$.

It is easily shown that if the labeling produced by the BFS is inconsistent, then the set of judgements is inconsistent. Note that this BFS labeling uses a subset of the judgements (the edges of the resulting BFS tree). Further the BFS labeling is the only possible labeling with the exception of inverting the labeling in each component of G, i.e. switching A and B. Thus if an inconsistency is found in this labeling then surely the entire set of m judgements cannot be consistent. On the other hand if the labeling is consistent with respect to the m judgements, we are done.

2. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain the spanning tree T. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same spanning tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

   Solution. Suppose that G has an edge $e = \{a, b\}$ that does not belong to T. Since T is a depth-first search tree, one of the two ends must be an ancestor of the other — say a is an ancestor of b. Since T is a breadth-first search tree, the distance of the two nodes from u in T can differ by at most one.

   But if a is an ancestor of b, and the distance from u to b in T is at most one greater than the distance from u to a, then a must in fact be the direct parent of b in T. ¿From this it follows that $\{a, b\}$ is an edge of T, contradicting our initial assumption that $\{a, b\}$ did not belong to T.

3. When we discussed the problem of determining the cut-points in a graph, we mentioned that one can compute the values *earliest*(u) for all nodes u as part of the DFS computation — rather than computing the DFS tree first, and these values subsequently.

Give an algorithm that does this: show how to augment the recursive procedure $DFS(v)$ so that it still runs in $O(m+n)$, and it terminates with globally stored values for $earliest(u)$.

4. A number of recent stories in the press about the structure of the Internet and the Web have focused on some version of the following question: How far apart are typical nodes in these networks? If you read these stories carefully, you find that many of them are confused about the difference between the *diameter* of a network and the *average distance* in a network — they often jump back and forth between these concepts as though they're the same thing.

   As in the text, we say that the *distance* between two nodes u and v in a graph $G = (V, E)$ is the minimum number of edges in a path joining them; we'll denote this by $dist(u, v)$. We say that the *diameter* of G is the maximum distance between any pair of nodes; and we'll denote this quantity by $diam(G)$.

   Let's define a related quantity, which we'll call the *average pairwise distance* in G (denoted $apd(G)$). We define $apd(G)$ to be the average, over all $\binom{n}{2}$ sets of two distinct nodes u and v, of the distance between u and v. That is,

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u, v) \right] / \binom{n}{2}.$$

   Here's a simple example to convince yourself that there are graphs G for which $diam(G) \neq apd(G)$. Let G be a graph with three nodes u, v, w; and with the two edges $\{u, v\}$ and $\{v, w\}$. Then

$$diam(G) = dist(u, w) = 2,$$

   while

$$apd(G) = [dist(u, v) + dist(u, w) + dist(v, w)]/3 = 4/3.$$

   Of course, these two numbers aren't all *that* far apart in the case of this 3-node graph, and so it's natural to ask whether there's always a close relation between them. Here's a claim that tries to make this precise.

   > *Claim: There exists a positive natural number c so that for all graphs G, it is the case that*
   > $$\frac{diam(G)}{apd(G)} \leq c.$$

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

   Solution. The claim is false; we show that for every natural number c, there exists a graph G so that $diam(G)/apd(G) > c$. First, we fix a number k (the relation to c will be determined later), and consider the following graph. We take a path on

$k-1$ nodes $u_1, u_2, \ldots, u_{k-1}$ in this order. We then attach $n-k+1$ additional nodes $v_1, v_2, \ldots, v_{n-k+1}$, each by a single edge, to u_1; the number n will also be chosen below.

The diameter of G is equal to $dist(v_1, u_{k-1}) = k$. It is not difficult to work out the exact value of $apd(G)$; but we can get a simple upper bound as follows. There are at most kn 2-element sets with at least one element from $\{u_1, u_2, \ldots, u_{k-1}\}$. Each of these pairs is at most distance $\leq k$. The remaining pairs are all distance at most 2. Thus

$$apd(G) \leq \frac{2\binom{n}{2} + k^2 n}{\binom{n}{2}} \leq 2 + \frac{2k^2}{n-1}.$$

Now, if we choose $n-1 > 2k^2$, then we have $apd(G) < 3$. Finally, choosing $k > 3c$, we have $diam(G)/apd(G) > 3c/3 = c$.

5. Some friends of yours work on wireless networks, and they're currently studying the properties of a network of n mobile devices. As the devices move around (really, as their human owners move around), they define a graph at any point in time as follows: there is a node representing each of the n devices, and there is an edge between device i and device j if the physical locations of i and j are no more than 500 meters apart. (If so, we say that i and j are "in range" of each other.)

   They'd like it to be the case that the network of devices is connected at all times, and so they've constrained the motion of the devices to satisfy the following property: at all times, each device i is within 500 meters of at least $n/2$ of the other devices. (We'll assume n is an even number.) What they'd like to know is: Does this property by itself guarantee that the network will remain connected?

   Here's a concrete way to formulate the question as a claim about graphs:

   > *Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least $n/2$, then G is connected.*

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

   Solution. The claim is true; here is a proof. Let G be a graph with the given properties, and suppose by way of contradiction that it is not connected. Let S be the nodes in its smallest connected component. Since there are at least two connected components, we have $|S| \leq n/2$. Now, consider any node $u \in S$. Its neighbors must all lie in S, so its degree can be at most $|S| - 1 \leq n/2 - 1 < n/2$. This contradicts our assumption that every node has degree at least $n/2$.

3 Greedy Algorithms

1. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they

have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive to the New York station one-by-one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive — otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and want your opinion on whether the situation can be improved. Here is how they are thinking: maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allowing the next few trucks to be better packed.

Prove that 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.

2. 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 — e.g. 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 finite 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.

Solution. Let the sequence S consist of $s_1, \ldots, s_n$ and the sequence S' consist of $s'_1, \ldots, s'_m$. We give a greedy algorithm that finds the first event in S that is the same as s'_1, matches these two events, then finds the first event after this that is the same as s'_2, and so on. We will use $k_1, k_2, \ldots$ to denote the match have we found so far, i to denote the current position in S, and j the current position in S'.

> Initially $i = j = 1$ While $i \leq n$ and $j \leq m$ If s_i is the same as s'_j, then let $k_j = i$ let $i = i + 1$ and $j = j + 1$ otherwise let $i = i + 1$ EndWhile If $j = m + 1$ return the subsequence found: $k_1, \ldots, k_m$ Else return that "S' is not a subsequence of S"

The running time is $O(n)$: one iteration through the while look takes $O(1)$ time, and each iteration increments i, so there can be at most n iterations.

It is also clear that the algorithm finds a correct match if it finds anything. It is harder to show that if the algorithm fails to find a match, then no match exists. Assume that S' is the same as the subsequence $s_{l_1}, \ldots, s_{l_m}$ of S. We prove by induction that the algorithm will succeed in finding a match and will have $k_j \leq l_j$ for all $j = 1, \ldots, m$. This is analogous to the proof in class that the greedy algorithm finds the optimal solution for the interval scheduling problem: we prove that the greedy algorithm is always ahead.

- *For each $j = 1, \ldots, m$ the algorithm finds a match k_j and has $k_j \leq l_j$.*

Proof. The proof is by induction on j. First consider $j = 1$. The algorithm lets k_1 be the first event that is the same as s'_1, so we must have that $k_1 \leq l_1$.

Now consider a case when $j > 1$. Assume that $j - 1 < m$ and assume by the induction hypothesis that the algorithm found the match k_{j-1} and has $k_{j-1} \leq l_{j-1}$. The algorithm lets k_j be the first event after k_{j-1} that is the same as s'_j if such an event exists. We know that l_j is such an event and $l_j > l_{j-1} \geq k_{j-1}$. So $s_{l_j} = s'_j$, and $l_j > k_{j-1}$. The algorithm finds the first such index, so we get that $k_j \leq l_j$. ∎

3. 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 4 miles of one of the base stations.

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

   Solution. Here is a greedy algorithm for this problem. Start at the western end of the road and begin moving east until the first moment when there's a house h exactly four miles to the west. We place a base station at this point (if we went any farther east without placing a base station, we wouldn't cover h). We then delete all the houses covered by this base station, and iterate this process on the remaining houses.

Here's another way to view this algorithm. For any point on the road, define its *position* to be the number of miles it is from the western end. We place the first base station at the easternmost (i.e. largest) position s_1 with the property that all houses between 0 and s_1 will be covered by s_1. In general, having placed $\{s_1, \ldots, s_i\}$, we place base station $i+1$ at the largest position s_{i+1} with the property that all houses between s_i and s_{i+1} will be covered by s_i and s_{i+1}.

Let $S = \{s_1, \ldots, s_k\}$ denote the full set of base station positions that our greedy algorithm places, and let $T = \{t_1, \ldots, t_m\}$ denote the set of base station positions in an optimal solution, sorted in increasing order (i.e. from west to east). We must show that $k = m$.

We do this by showing a sense in which our greedy solution S "stays ahead" of the optimal solution T. Specifically, we claim that $s_i \geq t_i$ for each i, and prove this by induction. The claim is true for $i = 1$, since we go as far as possible to the east before placing the first base station. Assume now it is true for some value $i \geq 1$; this means that our algorithm's first i centers $\{s_1, \ldots, s_i\}$ cover all the houses covered by the first i centers $\{t_1, \ldots, t_i\}$. As a result, if we add t_{i+1} to $\{s_1, \ldots, s_i\}$, we will not leave any house between s_i and t_{i+1} uncovered. But the $(i+1)^{\text{st}}$ step of the greedy algorithm chooses s_{i+1} to be *as large as possible* subject to the condition of covering all houses between s_i and s_{i+1}; so we have $s_{i+1} \geq t_{i+1}$. This proves the claim by induction.

Finally, if $k > m$, then $\{s_1, \ldots, s_m\}$ fails to cover all houses. But $s_m \geq t_m$, and so $\{t_1, \ldots, t_m\} = T$ also fails to cover all houses, a contradiction.

4. Consider the following variation on the *Interval Scheduling Problem* from lecture. You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)

   Given a list of n such jobs, your goal is to accept *as many jobs as possible* (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n, the number of jobs. You may assume for simplicity that no two jobs have the same start or end times.

   Example: Consider the following four jobs, specified by *(start-time, end-time)* pairs.

   (6 pm, 6 am), (9 pm, 4 am), (3 am, 2 pm), (1 pm, 7 pm).

   The unique solution would be to pick the two jobs (9 pm, 4 am) and (1 pm, 7 pm), which can be scheduled without overlapping.

   Solution. Let $I_1, \ldots, I_n$ denote the n intervals. We say that an *I_j-restricted solution* is one that contains the interval I_j.

Here is an algorithm, for fixed j, to compute an I_j-restricted solution of maximum size. Let x be a point contained in I_j. First delete I_j and all intervals that overlap it. The remaining intervals do not contain the point x, so we can "cut" the time-line at x and produce an instance of the Interval Scheduling Problem from class. We solve this in $O(n)$ time, assuming that the intervals are ordered by ending time.

Now, the algorithm for the full problem is to compute an I_j-restricted solution of maximum size for each $j = 1, \ldots, n$. This takes a total time of $O(n^2)$. We then pick the largest of these solutions, and claim that it is an optimal solution. To see this, consider the optimal solution to the full problem, consisting of a set of intervals S. Since $n > 0$, there is some interval $I_j \in S$; but then S is an optimal I_j-restricted solution, and so our algorithm will produce a solution at least as large as S.

5. Consider the following scheduling problem. You have a n jobs, labeled $1, \ldots, n$, which must be run one at a time, on a single processor. Job j takes time t_j to be processed. We will assume that no two jobs have the same processing time; that is, there are no two distinct jobs i and j for which $t_i = t_j$.

   You must decide on a schedule: the order in which to run the jobs. Having fixed an order, each job j has a *completion time* under this order: this is the total amount of time that elapses (from the beginning of the schedule) before it is done being processed.

   For example, suppose you have a set of three jobs $\{1, 2, 3\}$ with

$$t_1 = 3, \quad t_2 = 1, \quad t_3 = 5,$$

   and you run them in this order. Then the completion time of job 1 will be 3, the completion of job 2 will be $3 + 1 = 4$, and the completion time of job 3 will be $3 + 1 + 5 = 9$.

   On the other hand, if you run the jobs in the reverse of the order in which they're listed (i.e. 3, 2, 1), then the completion time of job 3 will be 5, the completion of job 2 will be $5 + 1 = 6$, and the completion time of job 1 will be $5 + 1 + 3 = 9$.

   (a) Give an algorithm that takes the n processing times $t_1, \ldots, t_n$, and orders the jobs so that the *sum* of the completion times of all jobs is as small as possible. (Such an order will be called *optimal.*)

   The running time of your algorithm should be polynomial in n. You should give a complete proof of correctness of your algorithm, and also briefly analyze the running time. As above, you can assume that no two jobs have the same processing time.

   (b) Prove that if no two jobs have the same processing time, then the optimal order is *unique.* In other words, for any order other than the one produced by your algorithm in (a), the sum of the completion times of all jobs is not as small as possible.

   You may find it helpful to refer to parts of your analysis from (a).

Solution. We order the jobs in decreasing order of processing time; this is the schedule we return. The running time is $O(n \log n)$.

To see why this algorithm is correct, suppose by way of contradiction that some other algorithm is optimal. As in the argument for minimum lateness from class, this other schedule must have an *inversion*: it must place a job i just before a job j, with the property that $t_i > t_j$. Suppose in this schedule, the completion time of i is c_i; and so the completion time of j is $c_j = c_i + t_j$.

Now, we exchange: we consider the schedule obtained by swapping the order of i and j. The new completion time of i is $c'_i = c_j$. But the new completion time of j is $c'_j = c_i + (t_j - t_i) < c_i$. All other completion times remain the same, and so the total sum has gone down. Thus, our other schedule cannot be optimal, a contradiction.

(b) If we consider the proof from (a), it shows that any schedule with an inversion cannot be optimal. But the only schedule without an inversion is the one obtained by our algorithm in (a), and so this is the unique optimal solution.

6. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swin 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swins the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he/she's out and starts biking, a third contestant begins swimming ... and so on.)

   Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running. Your friend wants to decide on a *schedule* for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts.

   What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

   Solution. Let the contestants be numbered $1, \ldots, n$, and let s_i, b_i, r_i denote the swimming, biking, and running times of contestant i. Here is an algorithm to produce a schedule: arrange the contestants in order of decreasing $b_i + r_i$, and send them out in this order. We claim that this order minimizes the completion time.

   We prove this by an exchange argument. Consider any optimal solution, and suppose it does not use this order. Then the optimal solution must contain two contestants i

and j so that j is sent out directly after i, but $b_i + r_i < b_j + r_j$. We will call such a pair (i, j) an *inversion*. Consider the solution obtained by swapping the orders of i and j. In this swapped schedule, j completes earlier than he/she used to. Also, in the swapped schedule, i gets out of the pool when j previously got out of the pool; but since $b_i + r_i < b_j + r_j$, i finishes sooner in the swapped schedule than j finished in the previous schedule. Hence our swapped schedule does not have a greater completion time, and so it too is optimal.

Continuing in this way, we can eliminate all inversions without increasing the completion time. At the end of this process, we will have a schedule in the order produced by our algorithm, whose completion time is no greater than that of the original optimal order we considered. Thus the order produced by our algorithm must also be optimal.

7. The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it re-compiles its index. Fortunately, the company has at its disposal a single large super-computer together with an essentially unlimited supply of high-end PC's.

   They've broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, which can performed completely independently of each other. Each job consists of two stages: first it needs to be *pre-processed* on the super-computer, and then it needs to be *finished* on one of the PC's. Let's say that job J_i needs p_i seconds of time on the super-computer followed by f_i seconds of time on a PC.

   Since there are at least n PC's available on the premises, the finishing of the jobs can be performed fully in parallel — all the jobs can be processed at the same time. However, the super-computer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the super-computer. As soon as the first job in order is done on the super-computer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the super-computer; when the second job is done on the super-computer, it can proceed to a PC regardless of whether or not the first job is done or not (since the PC's work in parallel); and so on.

   Let's say that a *schedule* is an ordering of the jobs for the super-computer, and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing on the PC's. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

   Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

   Solution. It is clear that the working time *for the super-computer* does not depend on the ordering of jobs. Thus we can not change the time when the last job hands off to a PC. It is intuitively clear that the last job in our schedule should have the shortest *finishing* time.

   This informal reasoning suggests us that the following greedy schedule should be the optimal one.

Schedule G: Run jobs in the order of decreasing finishing time f_i.

Now we show that G is actually the optimal schedule. To prove this we will show that any other schedule can be optimized. More precisely, we will show that for any given schedule $S \neq G$, there is another schedule S' such that the competition time of S' is less than or equal to the competition time of S.

Consider any schedule S, and suppose it does not use this order. Then this schedule must contain two jobs J_k and J_l so that J_l runs directly after J_k, but the finishing time for the first job is less than the finishing time for the second one, i.e. $f_k < f_l$. We can optimize this schedule by swapping the order of these two jobs. Let S' be the schedule S where we swap only the order of J_k and J_l. It is clear that the finishing times for all jobs except J_k and J_l does not change. The job J_l now schedules earlier, thus this job will finish earlier than in the original schedule. The job J_k schedules later, but the super-computer hands off J_k to a PC in the new schedule S' at the same time as it would handed off J_l in the original schedule S. Since the finishing time for J_k is less than the finishing time for J_l, the job J_k will finish earlier in the new schedule than J_l would finish in the original one. Hence our swapped schedule does not have a greater completion time.

Using these swappings we can eventually get G (for example using the Bubble Sort) without increasing the completion time. Therefore the competition time for G is not greater than the competition time for any arbitrary schedule S. Thus G is optimal.

Grader's comments: Most people had the right algorithm. Some people worried about how to order the jobs with identical finish times f_i. The order of these jobs does not matter.

Common mistakes in the above exchange argument.

- The exchange argument should start with the optimal schedule $\mathcal{O}$ (or an arbitrary schedule S), and use exchanges to show that this schedule $\mathcal{O}$ can be turned into the schedule the algorithm produces without making the overall finishing time worse. Some people start with the algorithm's schedule G and argue that G cannot be improved by swapping two jobs. This proof shows only that a schedule obtained from G be a *single swap* is not better. There could be other schedules obtained by multiple swaps that are better.
- To make things work out in the argument you need to swap neighboring jobs in $\mathcal{O}$. This is important, as if you swap two jobs J_l and J_k that are not neighboring, than all the jobs between the two also change their finishing times.
- Many people claim that the proof is by contradiction. They start with an optimal schedule $\mathcal{O}$ and want to prove that assuming $\mathcal{O} \neq G$ leads to a contradiction. These proofs are somewhat wrong (even though we took at most 1 point off), as there are possibly many optimal schedules. The swaps of inverted jobs considered

above, do not necessarily make the schedule better, we only argue that they do not make it worse.

Many people tried to prove the correctness by induction, and almost everyone failed. There is a correct induction proof, but very few people had this. We can prove that the greedy solution produced is optimal using induction on the number of jobs n. Here is how: For $n = 1$ there is nothing to prove. So let $n > 1$, and assume the algorithm find the optimal schedule for any set of at most $n-1$ jobs. Let job J_n be job scheduled as last. There are two cases to consider.

- Case 1: If job J_n finishes last on its PC among all jobs. Then this last finish time is $S + f_n$, where S is the total time needed on the supercomputer. Now whatever order we use we need S time on the supercomputer. There is always a last job, and that job needs to be finished on the PC. So the time is at least $S + f_j$ for some j (where j was the last job in an optimal schedule). Now f_n is minimal, so $S + f_n$ as finish time is clearly optimal.
- Case 2: If job n does not finish last. Then the finish time of the schedule generated by greedy is the finish time of the first $n-1$ jobs. By the induction hypothesis the first $n-1$ jobs finish at the minimal possible time. This finishes the proof.

An alternate direct way to phrase this argument is this:

Let job J_j be the job that finishes last on the PC in the algorithm's schedule, and let S_j be the time this job finishes on the supercomputer. So the overall finish time is $S_j + f_j$. In any other schedule, one of the first j jobs (first in the greedy order) must finish on the supercomputer at some time $T \geq S_j$ (as the first j jobs give exactly S_j work to the supercomputer). Let that be job J_i. Now job J_i needs PC time at least as much as job j (due to our ordering), and so it finishes at time $T + f_i \geq S_j + f_j$. So this other schedule is no better.

8. Suppose you have n video streams that need to be sent, one after another, over a communication link. Stream i consists of a total of b_i bits that need to be sent, at a constant rate, over a period of t_i seconds. You cannot send two streams at the same time, so you need to determine a *schedule* for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time 0 (and therefore ends at time $\sum_{i=1}^{n} t_i$, whichever order you choose). We assume that all the values b_i and t_i are positive integers.

   Now, because you're just one user, the link does not want you taking up too much bandwidth — so it imposes the following constraint, using a fixed parameter r:

   > $(*)$ For each natural number $t > 0$, the total number of bits you send over the time interval from 0 to t cannot exceed rt.

Note that this constraint is only imposed for time intervals that start at 0, *not* for time intervals that start at any other value.

We say that a schedule is *valid* if it satisfies the constraint $(*)$ imposed by the link.

The problem is: 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 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 $b_i < rt_i$.

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. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

9. **(a)** Suppose you're a consultant for a communications company in northern New Jersey, and they come to you with the following problem. Consider a fiber-optic cable that passes through a set of n *terminals*, $t_1, \ldots, t_n$, in sequence. (I.e. it begins at terminal t_1, then passes through terminals $t_2, t_3, \ldots, t_{n-1}$, and ends at t_n.) Certain pairs of terminals wish to establish a *connection* on which they can exchange data; for t_i to t_j to establish a connection, they need to reserve access to the portion of the cable that runs between t_i and t_j.

Now, the magic of fiber-optic technology is that you can accomodate all connections simultaneously as follows. You assign a *wavelength* to each connection in such a way that two connections requiring overlapping portions of the cable need to be assigned different wavelengths. (So you can assign the same wavelength more than once, provided it is to connections using non-overlapping portions of the cable.) Of course, you could safely assign a different wavelength to every single connection, but this would be wasteful: the goal is to use as few distinct wavelengths as possible.

Define the *load* of the set of connection to be the maximum number of connections that require access to any single point on the cable. The load gives a natural lower bound on the number of distinct wavelengths you need: if the load is L, then there is some point on the cable through which L connections will be sending data, and each of these needs a different wavelength.

For an arbitrary set of connections (each specified by a pair of terminals) having a load of L, is it always possible to accomodate all connections using only L wavelengths? If so, give an algorithm to assign each connection one of L possible wavelengths in a conflict-free fashion; if not, give an example of a set of connections requiring a number of wavelengths greater than its load.

(b) Instead of routing on a linear cable, let's look at the problem of routing on a *ring.* So we consider a circular fiber-optic cable, passing through terminals $t_1, \ldots, t_n$ in clockwise order. For t_i and t_j to establish a connection, they must reserve the portion of the cable extending clockwise from t_i to t_j.

The rest of the set-up is the same as in part (a), and we can ask the same question: For an arbitrary set of connections (each specified by a pair of terminals) having a load of L, is it always possible to accomodate all connections using only L wavelengths? If so, give an algorithm to assign each connection one of L possible wavelengths in a conflict-free fashion; if not, give an example of a set of connections requiring a number of wavelengths greater than its load.

Solution. **(a)** The algorithm is analogous to the one used for the interval scheduling problem in class. Assume we have k communication requests; the i^{th} request has terminals (s_i, f_i), with $s_i < f_i$, for $i = 1, \ldots, k$. We will sort the requests by their end points f_i and assign them in turn to any wavelength where it is compatible. We will prove that L wavelengths are enough. (We will make the natural assumption that if two connections only intersect at their endpoints, then they can use the same wavelength; however, the solution is essentially the same if we make the opposite assumption.)

> Sort all requests by their end points f_i Let W denote a set of L wavelengths. Consider requests in the sorted order. When considering request i If there is any wavelength ℓ such that request i is compatible with the requests assigned to the wavelength ℓ Select such a wavelength ℓ and assign request i to ℓ. Otherwise reject request i End Return the assignments of requests to wavelength.

The algorithm clearly returns feasible assignments of requests to the L wavelengths. The running time will also be easy to analyze. It is harder to show that no request will be rejected.

- *The algorithm can be implemented in $O(k \log k) + O(kL)$ time.*

Proof. The first step of the algorithm is sorting, which takes $O(k \log k)$ time. The rest can be implemented in $O(kL)$ time. We consider the requests in order of their finishing points, and so in order to decide if a new request is compatible with the requests already assigned to a wavelength ℓ, all we need to know is if the finishing point f_j of the last interval assigned ℓ satisfies $f_j \leq s_i$. We maintain the last finishing points for each wavelength in an array of length L. Using this array we can assign a request to a wavelength in $O(L)$ time by looking through this array. ■

- *No request is rejected.*

Proof. We prove the statement by contradiction. Let i be a rejected request. When considering request i, we could not assign it to any of the L wavelengths. We are considering the requests in order of their finish point f_j. If request i is not compatible with a previously assigned request j, then it must be the case that $s_i < f_j$. We assumed that request i could not be assigned to any of the L wavelengths. This implies that each of the L wavelengths has a request that contains the segment immediately to the right of s_i. Request i also contains this segment, and hence the load of the set of requests is at least $L + 1$ — a contradiction. ■

(b)

L wavelengths may not be enough when routing on a ring. Consider the case when we have three connection requests on a cable with six terminals, as follows. Request 1 needs the cable from t_1 to t_4 (clockwise), request 2 needs the cable from t_3 to t_6, and request 2 needs the cable from t_5 to t_2. The load is 2, but we need 3 wavelengths, as no two requests can be assigned to the same wavelength.

10. Timing circuits are a crucial component of VLSI chips; here's a simple model of such a timing circuit. Consider a complete binary tree with n leaves, where n is a power of two. Each edge e of the tree has an associated length ℓ_e, which is a positive number. The *distance* from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.

    The root generates a *clock signal* which is propagated along the edges to the leaves. We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

    Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem: we want the leaves to be completely synchronized, and all receive the signal at the same time. To make this happen, we will have to *increase* the lengths of certain of the edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have *zero skew*. Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible.

Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew, and the total edge length is as small as possible.

Example. Consider the tree in accompanying figure, with letters naming the nodes and numbers indicating the edge lengths.

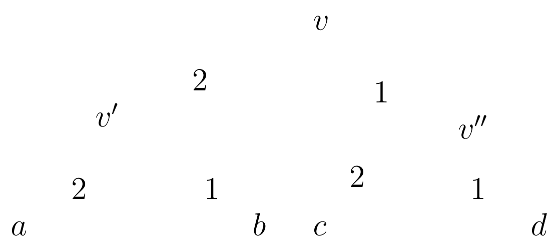


Figure 2: An instance of the zero-skew problem.

The unique optimal solution for this instance would be to take the three length-1 edges, and increase each of their lengths to 2. The resulting tree has zero skew, and the total edge length is 12, the smallest possible.

Solution. To design an optimal solution, we apply a general technique that is known as *Deferred Merge Embedding* (DME) by researchers in the VLSI community. It's a greedy algorithm that works as follows. Let v denote the root, with v' and v'' its two children. Let d' denote the maximum root-to-leaf distance over all leaves that are descendants of v', and let d'' denote the maximum root-to-leaf distance over all leaves that are descendants of v''. Now:

- If $d' > d''$, we add $d' - d''$ to the length of the v-to-v'' edge and add nothing to the length of the v-to-v' edge.
- If $d'' > d'$, we add $d'' - d'$ to the length of the v-to-v' edge and add nothing to the length of the v-to-v'' edge.
- $d' = d''$ we add nothing to the length of either edge below v.

We now apply this procedure recursively to the subtrees rooted at each of v' and v''.

Let T be the complete binary tree in the problem. We first develop two basic facts about the optimal solution, and then use these in an "exchange" argument to prove that the DME algorithm is optimal.

(i) Let w be an internal node in T, and let e', e'' be the two edges directly below w. If a solution adds non-zero length to both e' and e'', then it is not optimal.

Proof. Suppose that $\delta' > 0$ and $\delta'' > 0$ are added to $\ell_{e'}$ and $\ell_{e''}$ respectively. Let $\delta = \min(\delta', \delta'')$. Then the solution which adds $\delta' - \delta$ and $\delta'' - \delta$ to the lengths of these edges must also have zero skew, and uses less total length. ■

(ii) Let w be a node in T that is neither the root nor a leaf. If a solution increases the length of every path from w to a leaf below w, then the solution is not optimal.

Proof. Suppose that $x_1, \ldots, x_k$ are the leaves below w. Consider edges e in the subtree below w with the following property: the solution increases the length of e, and it does not increase the length of any edge on the path from w to e. Let F be the set of all such edges; we observe two facts about F. First, for each leaf x_i, the first edge on the w-x_i path whose length has been increased must belong to F, (and no other edge on this path can belong to F); thus there is exactly one edge from F on every w-x_i path. Second, $|F| \geq 2$, since a path in the left subtree below w shares no edges with a path in the right subtree below w, and yet each contains an edge of F.

Let e_w be the edge entering w from its parent (recall that w is not the root). Let δ be the minimum amount of length added to any of the edges in F. If we subtract δ from the length added to each edge in F, and add δ to the edge above w, the length of all root-to-leaf paths remains the same, and so the tree remains zero-skew. But we have subtracted $|F|\delta \geq 2\delta$ from the total length of the tree, and added only δ, so we get a zero skew tree with less total length. ■

We can now prove a somewhat stronger fact than what is asked for.

(iii) The DME algorithm produces the unique optimal solution.

Proof. Consider any other solution, and let v be any node of T at which the solution does not add lengths in the way that DME would. We use the notation from the problem, and assume without loss of generality that $d' \geq d''$. Suppose the solution adds δ' to the edge (v, v') and δ'' to the edge (v, v'').

If $\delta'' - \delta' = d' - d''$, then it must be that $\delta' > 0$ or else the solution would do the same thing as DME; in this case, by (i) it is not optimal. If $\delta'' - \delta' < d' - d''$, then the solution will still have to increase the length of the path from v'' to each of its leaves in order to make the tree zero-skew; so by (ii) it is not optimal. Similarly, if $\delta'' - \delta' > d' - d''$, then the solution will still have to increase the length of the path from v' to each of its leaves in order to make the tree zero-skew; so by (ii) it is not optimal. ■

11. Given a list of n natural numbers $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. (That is, if $V = \{v_1, v_2, \ldots, v_n\}$, then the degree

of v_i should be exactly d_i.) G should not contain multiple edges between the same pair of nodes, or "loop" edges with both endpoints equal to the same node.

Solution. Clearly if any of the putative degrees d_i is equal to 0, then this must be an isolated node in the graph; thus we can delete d_i from the list and continue by recursion on the smaller instance.

Otherwise, all d_i are positive. We sort the numbers, relabeling as necessary, so that $d_1 \geq d_2 \geq \cdots \geq d_n > 0$. We now look at the list of numbers

$$L = \{d_1 - 1, d_2 - 1, \ldots, d_{d_n} - 1, d_{d_n+1}, \ldots, d_{n-2}, d_{n-1}\}.$$

(In other words, we subtract 1 from the first d_n numbers, and drop the last number.) We claim that there exists a graph whose degrees are equal to the list $d_1, \ldots, d_n$ if and only if there exists a graph whose degrees form the list L. Assuming this claim, we can proceed recursively.

Why is the claim true? First, if there is a graph with degree sequence L, then we can add an n^{th} node with neighbors equal to nodes $v_1, v_2, \ldots, v_{d_n}$, thereby obtaining a graph with degree sequence $d_1, \ldots, d_n$. Conversely, suppose there is a graph with degree sequence $d_1, \ldots, d_n$, where again we have $d_1 \geq d_2 \geq \cdots \geq d_n$. We must show that in this case, there is in fact such a graph where node v_n is joined to precisely the nodes $v_1, v_2, \ldots, v_{d_n}$; this will allow us to delete node n and obtain the list L. So consider any graph G with degree sequence $d_1, \ldots, d_n$; we show how to transform G into a graph where v_n is joined to $v_1, v_2, \ldots, v_{d_n}$. If this property does not already hold, then there exist $i < j$ so that v_n is joined to v_j but not v_i. Since $d_i \geq d_j$, it follows that there must be some v_k not equal to any of v_i, v_j, v_n with the property that (v_i, v_k) is an edge but (v_j, v_k) is not. We now replace these two edges by (v_i, v_n) and (v_j, v_k). This keeps all degrees the same; and repeating this transformation will convert G into a graph with the desired property.

12. Your friends are planning an expedition to a small town deep in the Canadian north next winter break. They've researched all the travel options, and have drawn up a directed graph whose nodes represent intermediate destinations, and edges represent the roads between them.

    In the course of this, they've also learned that extreme weather causes roads in this part of the world to become quite slow in the winter, and may cause large travel delays. They've found an excellent travel Web site that can accurately predict how fast they'll be able to travel along the roads; however, the speed of travel depends on the time of year. More precisely, the Web site answers queries of the following form: given an 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 backwards 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.

Solution. It is clear that when the travel time does not vary, this problem is equivalent to the classical problem of finding the shortest path in the graph. We will extend Dijksra's algorithm to this problem.

To explain the idea we will use the analogy with the water pipes. Imagine the water spreads in the graph in all direction starting from the initial node at time 0. Suppose the water spreads with the same speed as we travel, i.e. if water reaches the node v at time t and there is an edge $e = (v, w)$ then the water reaches w at time $f_e(t)$. The question is at what time water reaches any given site.

The following modification of Dijksra's algorithm solves this problem. Also we need to find the fastest way to the destination (not only the traveling time). To do this we will store for each explored node u the last site before u on the fastest way to u. Then we can restore the fastest way going backward from the destination.

> Let S be the set of explored nodes. For each $u \in S$, we store a minimal time $d(u)$ when we can arrive at u and the last site before u on the fastest way to u
> Initially $S = \{s\}$ and $d(s) = 0$. While $S \neq V$ Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min\limits_{e=(u,v):u \in S} f_e(d(u))$ is as small as possible. Add v to S and define $d(v) = d'(v)$ and $r(v) = u$. EndWhile

To establish the correctness of our algorithm, we will prove that on the k-th step, S consists of the first k nodes that will be filled with water and $d(v)$ is the time when water reaches the node $v \in S$.

We prove it by induction on k. The case $k = 1$ is clear, because we know that water start spreads from the node s at time 0. Suppose this claim holds for $k \geq 1$. It is clear that water reaches the $k + 1$st node from some node in the set S (because we cannot travel backward in time). Water reaches a node $v \notin S$ from $u \in S$ in time $f_{(u,v)}(d(u))$ (because by induction hypothesis $d(u)$ is the time when water reaches u and we cannot arrive earlier by starting later). Therefore the $k + 1$st node v that can be reached by water has minimal $f_{(u,v)}(d(u))$ for all $u \in S$. This node is added to S on the $k + 1$st step. Thus we prove the induction step.

It is also clear that we can restore the reverse fastest path starting from the destination, and going form a site v to the site $r(v)$.

Grader's comments: Almost everyone saw that this was quite similar to Dijkstra's algorithm for shortest paths. However, the given problem is definitely more general than the shortest path problem that Dijkstra's algorithm solves, since the given problem does not actually have a static notion of "edge length" — the length depends on the time at which an edge is considered.

So it was definitely not enough, as a proof of correctness, to write something to the effect that this is "essentially Dijkstra's algorithm." A detailed proof is really needed. It can be a fairly close to the proof of (3.13), but it must explicitly refer to the fact that we now have edges whose "lengths" are time-changing. Moreover, the proof must somewhere make use of the fact that $f_e(t) \geq t$ and that $f_e(t)$ is monotone increasing, since the adapted version of Dijkstra's algorithm would not actually be correct without these assumptions. (To see why, consider the version of shortest paths with negative edge lengths, which we considered as part of the chapter on dynamic programming.)

13. Suppose you are given an undirected graph G, with edge weights that you may assume are all distinct. G has n vertices and m edges. A particular edge e of G is specified. Give a algorithm with running time $O(m + n)$ to decide whether e is contained in a minimum-weight spanning tree of G.

    Solution. Suppose that e has endpoints v and w. The crucial fact is the following:

    Fact 3.1 *e does not belong to a minimum spanning tree of G if and only if v and w can be joined by a path consisting entirely of edges of weight less than w_e.*

    Proof. To prove the "if" direction, let P be a v-w path consisting entirely of edges of weight less than w_e. If we add e to P, we get a cycle on which e is the heaviest edge. Thus, by the cycle property, e does not belong to a minimum spanning tree of G.

    To prove the "only if" direction, suppose the v and w cannot be joined by a path consisting entirely of edges of weight less than w_e, and suppose by way of contradiction that T is a minimum spanning tree of G that does not contain e. On the path P in T from v to w, there must be an edge f of weight $w_f > w_e$. Choose the heaviest such edge f. If we add e to this path P, then we get a cycle on which f is the heaviest edge. This contradicts the cycle property, which says that f could not belong to a minimum spanning tree. ■

    Given this fact, our algorithm is now the following. We form a graph G' by deleting from G all edges of weight greater than or equal to w_e. In $O(m)$ time, we perform a depth-first search on G' to determine whether v and w belong to the same connected component. If they do, then e does not belong to a minimum spanning tree of G; if they don't, then e does belong to a minimum spanning tree of G.

14. Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum cost spanning tree T in G. Now assume that a new edge is added, connecting two nodes $v, w \in V$ with cost c.

    a Give an efficient algorithm to test if T remains the minimum cost spanning tree with the new edge added. Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time? Please note any assumption you make about what data structure is used to represent the tree T and the graph G.

    b Suppose T is no longer the minimum cost spanning tree. Give a linear time algorithm to update the tree T to the new minimum cost spanning tree.

15. One of the basic motivations behind the minimum spanning tree problem is the goal of designing a spanning network for a set of nodes with minimum *total* cost. Here, we explore another type of objective: designing a spanning network for which the *most expensive* edge is as cheap as possible.

    Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge weights that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with the greatest weight.

    A spanning tree T of G is a *minimum bottleneck spanning tree* if there is no spanning tree T' of G with a lighter bottleneck edge.

    (a) Is every minimum bottleneck tree of G a minimum spanning tree of G? Prove or give a counter-example.

    (b) Is every minimum spanning tree of G a minimum bottleneck tree of G? Prove or give a counter-example.

    (c)$(*)$ Give an algorithm with running time $O(m + n)$ that, on input G, computes a minimum bottleneck spanning tree of G. (*Hint: You may use the fact that the median of a set of k numbers can be computed in time $O(k)$.*)

    Solution. **(a)** This is false. Let G have vertices $\{v_1, v_2, v_3, v_4\}$, with edges between each pair of vertices, and with the weight on the edge from v_i to v_j equal to $i + j$. Then every tree has a bottleneck edge of weight at least 5, so the tree consisting of a path through vertices v_3, v_2, v_1, v_4 is a minimum bottleneck tree. It is a not a minimum spanning tree, however, since its total weight is greater than that of the tree with edges from v_1 to each other vertex.

    (b) This is true; the easiest way to prove it may be to use the matroid exchange property of forests in a graph. Suppose that T is a minimum spanning tree of G, and T' is a spanning tree with a lighter bottleneck edge. Thus, T contains an edge e that is heavier than every edge in T'. Now, $T - \{e\}$ has fewer edges than T', so by the matroid exchange property of forests, we know that there is some edge $e' \in T'$ so that $(T - \{e\}) \cup \{e'\}$ is a spanning tree of G. But this tree has total weight less than that of T, contradicting our assumption that T is a minimum spanning tree.

(c) We show how, in $O(m)$ time, to reduce the problem to one with half as many edges. Thus, if $T(m)$ denotes the running time of our algorithm, it will satisfy the recurrence $T(m) \leq T(m/2) + O(m)$, which implies $T(m) = O(m)$.

Let E^- denote the set of all edges of weight at most the median edge value, and E^+ denote the set of all edges of weight greater than the median edge value. We first test, by depth-first search, whether the graph (V, E^-) is connected. If it is, then the minimum bottleneck tree is a subgraph of (V, E^-), so we can delete E^+ from G and continue by recursion. If (V, E^-) is not connected, then we run apply a Boruvka step to the graph, contracting its connected components in $O(m)$ time. Recursively, we find a minimum bottleneck tree in this contracted graph, which has half as many edges. We then extend this to a minimum bottleneck tree of G by choosing an arbitrary spanning tree inside each contracted component; since each of the contracted edges has weight less than that of the bottleneck edge, our choice of tree in these components does not matter.

16. In trying to understand the combinatorial structure of spanning trees, we can consider the space of *all* possible spanning trees of a given graph, and study the properties of this space. This is a strategy that has been applied to many similar problems as well.

Here is one way to do this. Let G be a connected graph, and T and T' two different spanning trees of G. We say that T and T' are *neighbors* if T contains exactly one edge that is not in T', and T' contains exactly one edge that is not in T.

Now, from any graph G, we can build a (large) graph $\mathcal{H}$ as follows. The nodes of $\mathcal{H}$ are the spanning trees of G, and there is an edge between two nodes of $\mathcal{H}$ if the corresponding spanning trees are neighbors.

Is it true that for any connected graph G, the resulting graph $\mathcal{H}$ is connected? Give a proof that $\mathcal{H}$ is always connected, or provide an example (with explanation) of a connected graph G for which $\mathcal{H}$ is not connected.

Solution. Yes, $\mathcal{H}$ will always be connected. To show this, we prove the following fact.

Fact 3.2 *Let $T = (V, F)$ and $T' = (V, F')$ be two spanning trees of G so that $|F - F'| = |F' - F| = k$. Then there is a path in $\mathcal{H}$ from T to T' of length k.*

Proof. We prove this by induction on k, the case $k = 1$ constituting the definition of edges in $\mathcal{H}$. Now, if $|F - F'| = k > 1$, we choose an edge $f' \in F' - F$. The tree $T \cup \{f'\}$ contains a cycle C, and this cycle must contain an edge $f \notin F'$. The tree $T \cup \{f'\} - \{f\} = T'' = (V, F'')$ has the property that $|F'' - F'| = |F' - F''| = k - 1$. Thus, by induction, there is a path of length $k - 1$ from T'' to T'; since T and T'' are neighbors, it follows that there is a path of length k from T to T'. ■

17. 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.

Their 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 agreement with companies X and Y stipulating a number k so that in the tree T that is chosen, k of the edges will be owned by X and $n - k - 1$ 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, and how to find one if it exists. 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 (i) returns a spanning tree with exactly k edges labeled X, or (ii) reports correctly that no such tree exists.

Solution. We begin by noticing two facts related to the graph $\mathcal{H}$ defined in the previous problem. First, if T and T' are neighbors in $\mathcal{H}$, then the number of X-edges in T can differ from the number of X-edges in T' by at most one. Second, the solution given above in fact provides a polynomial-time algorithm to construct a T-T' path in H.

We call a tree *feasible* if it has exactly k X-edges. Our algorithm to search for a feasible tree is as follows. Using a minimum-spanning tree algorithm, we compute a spanning tree T with the minimum possible number a of X-edges. We then compute a spanning tree T with the maximum possible number b of X-edges. If $k < a$ or $k > b$, then there clearly there is no feasible tree. If $k = a$ or $k = b$, then one of T or T' is a feasible tree. Now, if $a < k < b$, we construct a sequence of trees corresponding to a T-T' path in $\mathcal{H}$. Since the number of X-edges changes by at most one on each step of this path, and overall it increases from a to b, one of the trees on this path is a feasible tree, and we return it as our solution.

18. Suppose you are given a connected graph $G = (V, E)$, with a weight w_e on each edge e. On the first problem set, we saw that when all edge weights are distinct, G has a unique minimum-weight spanning tree. However, G may have many minimum-weight spanning trees when the edge weights are not all distinct. Here, we formulate the question: can Kruskal's algorithm be made to find all the minimum-weight spanning trees of G?

    Recall that Kruskal's algorithm sorted the edges in order of increasing weight, then greedily processed edges one-by-one, adding an edge e as long as it did not form a cycle. When some edges have the same weight, the phrase "in order of increasing weight" has to be specified a little more carefully: we'll say that an ordering of the edges is *valid* if the corresponding sequence of edge weights is non-decreasing. We'll say that a *valid execution* of Kruskal's algorithm is one that begins with a valid ordering of the edges of G.

    For any graph G, and any minimum spanning tree T of G, is there a valid execution of Kruskal's algorithm on G that produces T as output? Give a proof or a counter-example.

Solution. Label the edges arbitrarily as $e_1, \ldots, e_m$ with the property that $e_{m-n+1}, \ldots, e_m$ belong to T. Let δ be the minimum difference between any two non-equal edge weights; subtract $\delta i/n^3$ from the weight of edge i. Note that all edge weights are now distinct, and the sorted order of the new weights is the same as some valid ordering of the original weights. Over all spanning trees of G, T is the one whose total weight has been reduced by the most; thus, it is now the unique minimum spanning tree of G and will be returned by Kruskal's algorithm on this valid ordering.

19. 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 clearly 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 a somewhat counter-intuitive claim, 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:

> *A subgraph (V, E') is a minimum-altitude connected subgraph if and only if it contains the edges of the minimum spanning tree.*

Note that this second conjecture would immediately imply the first one, since the minimum spanning tree contains its own edges.

So here's the question:

(a) Is the first conjecture true, for all choices of G and altitudes a_e? Give a proof or a counter-example with explanation.

(b) Is the second conjecture true, for all choices of G and altitudes a_e? Give a proof or a counter-example with explanation.

Solution. Consider the minimum spanning tree T of G under the edge weights $\{a_e\}$, and suppose T were not a minimum-altitude connected subgraph. Then there would be some pair of nodes u and v, and two u-v paths $P \neq P^*$ (represented as sets of edges), so that P is the u-v path in T but P^* has smaller height. In other words, there is an edge $e' = (u', v')$ on P that has the maximum altitude over all edges in $P \cup P^*$. Now, if we consider the edges in $(P \cup P^*) - \{e'\}$, they contain a (possibly self-intersecting) u'-v' path; we can construct such a path by walking along P from u' to u, then along P^* from u to v, and then along P from v to v'. Thus $(P \cup P^*) - \{e'\}$ contains a simple path Q. But then $Q \cup \{e\}$ is a cycle on which e' is the heaviest edge, contradicting the Cycle Property. Thus, T must be a minimum-altitude connected subgraph.

Now consider a connected subgraph $H = (V, E')$ that does not contain all the edges of T; let $e = (u, v)$ be an edge of T that is not part of E'. Deleting e from T partitions T into two connected components; and these two components represent a partition of V into sets A and B. The edge e is the minimum-altitude edge with one end in A and the other in B. Since any path in H from u to v must cross at some point from A to B, and it cannot use e, it must have height greater than a_e. It follows that H cannot be a minimum-altitude connected subgraph.

20. One of the first things you learn in calculus is how to minimize a differentiable function like $y = ax^2 + bx + c$, 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 :

$\mathbf{R} \to \mathbf{R}$. 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 function f_e 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 $\{(a_e, b_e, c_e) : e \in E\}$, 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 numbers $\{(a_e, b_e, c_e)\}$ can be done in constant time per operation.

Solution. First, we observe the following fact. If we consider two different sets of costs $\{c_e\}$ and $\{c'_e\}$ on the edge set of E, with the property that the sorted order of $\{c_e\}$ and $\{c'_e\}$ is the same, then Kruskal's algorithm will output the same minimum spanning tree with respect to these two sets of costs.

It follows that for our time-changing edge costs, the set of edges in the minimum spanning tree only changes when two edge costs change places in the overall sorted order. This only happens when two of the parabolas defining the edge costs intersect. Since two parabolas can cross at most twice, the structure of the minimum spanning tree can change at most $2\binom{m}{2} \leq m^2$ times, where m is the number of edges. Moreover, we can enumerate all these crossing points in polynomial time.

So our algorithm is as follows. We determine all crossing points of the parabolas, and divide the "time axis" $\mathbf{R}$ into $\leq m^2$ intervals over which the sorted order of the costs remains fixed. Now, over each interval I, we run Kruskal's algorithm to determine the minimum spanning tree T_I. The cost of T_I over the interval I is a sum of $n-1$ quadratic functions, and hence is itself a quadratic function; thus, having determined the sum of these $n-1$ quadratic functions, we can determine its minimum over I in constant time.

Finally, minimizing over the best solution found in each interval gives us the desired tree and value of t.

21. Suppose we are given a set of points $P = \{p_1, p_2, \ldots, p_n\}$, together with a distance function d on the set P; as usual, d is simply a function on pairs of points in P with the properties that $d(p_i, p_j) = d(p_j, p_i) > 0$ if $i \neq j$, and that $d(p_i, p_i) = 0$ for each i.

We define a *stratified metric* on P to be any distance function τ that can be constructed as follows. We build a rooted tree T with n leaves, and we associate with each node v of T (both leaves and internal nodes) a *height* h_v. These heights must satisfy the properties that $h(v) = 0$ for each leaf v, and if u is the parent of v in T, then $h(u) \geq h(v)$. We place each point in P at a distinct leaf in T. Now, for any pair of points p_i and p_j, their distance $\tau(p_i, p_j)$ is defined as follows. We determine the least common ancestor v in T of the leaves containing p_i and p_j, and define $\tau(p_i, p_j) = h_v$.

We say that a stratified metric τ is *consistent* with our distance function d if for all pairs i, j, we have $\tau(p_i, p_j) \leq d(p_i, p_j)$.

Give a polynomial-time algorithm that takes the distance function d and produces a stratified metric τ with the following properties:

(i) τ is consistent with d, and

(ii) if τ' is any other stratified metric consistent with d, then $\tau'(p_i, p_j) \leq \tau(p_i, p_j)$ for each pair of points p_i and p_j.

Solution. We run Kruskal's MST algorithm, and build τ inductively as we go. Each time we merge components C_i and C_j by an edge of length ℓ, we create a new node v to be the parent of the subtrees that (by induction) consist of C_i and C_j, and give v a height of ℓ.

Note that for any p_i and p_j, the quantity $\tau(p_i, p_j)$ is equal to the edge length considered when the components containing p_i and p_j were first joined. We must have $\tau(p_i, p_j) \leq d(p_i, p_j)$, since p_i and p_j will belong to the same component by the time the direct edge (p_i, p_j) is considered.

Now, suppose there were a hierarchical metric τ' such that $\tau'(p_i, p_j) > \tau(p_i, p_j)$. Let T' be the tree associated with τ', v' the least common ancestor of p_i and p_j in T', and T'_i and T'_j the subtrees below v' containing p_i and p_j. If h'_v is the height of v' in T', then $\tau'(p_i, p_j) = h'_v > \tau(p_i, p_j)$.

Consider the p_i-p_j path P in the minimum spanning tree. Since $p_j \notin T'_i$, there is a first node $p' \in P$ that does not belong to T'_i. Let p be the node immediately preceding p on P. Then $d(p, p') \geq h'_v > \tau(p_i, p_j)$, since the least common ancestor of p and p' in T' must lie above the root of T'_i. But by the time Kruskal's algorithm merged the components containing p_i and p_j, all edges of P were present, and hence each has length at most $\tau(p_i, p_j)$, a contradiction.

22. Let's go back to the original motivation for the minimum spanning tree problem: we are given a connected, undirected graph $G = (V, E)$ with positive edge lengths $\{\ell_e\}$, and we want to find a spanning subgraph of it. Now, suppose we are willing to settle for a subgraph $H = (V, F)$ that is "denser" than a tree, and we are interested in guaranteeing that for each pair of vertices $u, v \in V$, the length of the shortest u-v path in H is not much longer than the length of the shortest u-v path in G. By the *length* of a path P here, we mean the sum of ℓ_e over all edges e in P.

Here's a variant of Kruskal's algorithm designed to produce such a subgraph.

- First, we sort all the edges in order of increasing length. (You may assume all edge lengths are distinct.)
- We then construct a subgraph $H = (V, F)$ by considering each edge in order.

- When we come to edge $e = (u, v)$, we add e to the subgraph H if there is currently no u-v path in H. (This is what Kruskal's algorithm would do as well.) On the other hand, if there is a u-v path in H, we let d_{uv} denote the total length of the shortest such path; again, length is with respect to the values $\{\ell_e\}$. We add e to H if $3\ell_e < d_{uv}$.

In other words, we add an edge even when u and v are already in the same connected component, provided that the addition of the edge reduces their shortest-path distance by a sufficient amount.

Let $H = (V, F)$ be the subgraph of G returned by the algorithm.

(a) Prove that for every pair of nodes $u, v \in V$, the length of the shortest u-v path in H is at most 3 times the length of the shortest u-v path in G.

(b)(*) Despite its ability to approximately preserve shortest-path distances, the subgraph H produced by the algorithm cannot be too dense. Let $f(n)$ denote the maximum number of edges that can possibly be produced as the output of this algorithm, over all n-node input graphs with edge lengths. Prove that

$$\lim_{n\to\infty} \frac{f(n)}{n^2} = 0.$$

Solution. **A useful fact.** The solution to (b) involves a sum of terms (the sum of node degrees) that we want to show is asymptotically sub-quadratic. Here's a fact that's useful in this type of situation.

> *Lemma: Let $a_1, a_2, \ldots, a_n$ be integers, each between 0 and n, such that $\sum_i a_i \geq \varepsilon n^2$. Then at least $\frac{1}{2}\varepsilon n$ of the a_i have value at least $\frac{1}{2}\varepsilon n$.*

To prove this lemma, let k denote the number of a_i whose value is at least $\frac{1}{2}\varepsilon n$. Then we have $\varepsilon n^2 \leq \sum_i a_i \leq kn + \frac{1}{2}(n-k)\varepsilon n \leq kn + \frac{1}{2}\varepsilon n^2$, from which we get $k \geq \frac{1}{2}\varepsilon n$.

(a) For each edge $e = (u, v)$, there is a path P_{uv} in H of length at most $3\ell_e$ — indeed, either $e \in F$, or there was such a path at the moment e was rejected. Now, given an pair of nodes $s, t \in V$, let Q denote the shortest s-t path in G. For each edge (u, v) on Q, we replace it with the path P_{uv}, and then short-cut any loops that arise. Summing the length edge-by-edge, the resulting path has length at most 3 times that of Q.

(b) We first observe that H can have no cycle of length ≤ 4. For suppose there were such a cycle C, and let $e = (u, v)$ be the last edge added to it. Then at the moment e was considered, there was a u-v path Q_{uv} in H of at most three edges, on which each edge had length at most ℓ_e. Thus ℓ_e is not less than a third the length of Q_{uv}, and so it should not have been added.

This constraint implies that H cannot have $\Omega(n^2)$ edges, and there are several different ways to prove this. One proof goes as follows. If H has at least εn^2 edges, then the sum of all degrees is $2\varepsilon n^2$, and so by our lemma above, there is a set S of at least εn nodes each of whose degrees is at least εn. Now, consider the set Q of all pairs of edges (e, e') such e and e' each have an end equal to the same node in S. We have $|Q| \geq \varepsilon n\binom{\varepsilon n}{2}$, since there are at least εn nodes in S, and each contributes at least $\binom{\varepsilon n}{2}$ such pairs. For each edge pair $(e, e') \in Q$, they have one end in common; we *label* (e, e') with the pair of nodes at their other ends. Since $|Q| > \binom{n}{2}$ for sufficiently large n, the pigeonhole principle implies that some two pairs of edges $(e, e'), (f, f') \in Q$ receive the same label. But then $\{e, e', f, f'\}$ constitutes a four-node cycle.

For a second proof, we observe that an n-node graph H with no cycle of length ≤ 4 must contain a node of degree at most $\sqrt{n}$. For suppose not, and consider any node v of H. Let S denote the set of neighbors of v. Notice that there is no edge joining two nodes of S, or we would have a cycle of length 3. Now let $N(S)$ denote the set of all nodes with a neighbor in S. Since H has no cycle of length 4, each node in $N(S)$ has exactly one neighbor in S. But $|S| > \sqrt{n}$, and each node in S has $\geq \sqrt{n}$ neighbors other than v, so we would have $|N(S)| > n$, a contradiction. Now, if we let $g(n)$ denote the maximum number of edges in an n-node graph with no cycle of length 4, then $g(n)$ satisfies the recurrence $g(n) \leq g(n-1) + \sqrt{n}$ (by deleting the lowest-degree node), and so we have $g(n) \leq n^{3/2} = o(n^2)$.

23. Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j); and we will assume these weights satisfy the *triangle inequality* $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V'.

    We are given a set $X \subseteq V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subseteq Z \subseteq V$, together with a sub-tree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T.

    Show that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(n^{O(k)})$.

    Solution. In a Steiner tree T on $X \cup Z \subseteq V$, $|X| = k$, we will refer to X as the *terminals* and Z as the *extra nodes*. We first claim that each extra node has degree at least 3 in T; for if not, then the triangle inequality implies we can replace its two incident edges by an edge joining its two neighbors. Since the sum of the degrees in a t-node tree is $2t-2$, every tree has at least as many leaves as it has nodes of degree greater than 2. Hence $|Z| \leq k$. It follows that if we compute the minimum spanning tree on all sets of the form $X \cup Z$ with $|Z| \leq k$, the cheapest among these will be the minimum Steiner tree. There are at most $\binom{n}{2k} = n^{O(k)}$ such sets to try, so the overall running time will be $n^{O(k)}$.

24. Recall the problem of computing a minimum-cost arborescence in a directed graph $G = (V, E)$, with a cost $c_e \geq 0$ on each edge. Here we will consider the case in which G is a directed acyclic graph; that is, it contains no directed cycles.

As in general directed graphs, there can in general be many distinct minimum-cost solutions. Suppose we are given a directed acyclic graph $G = (V, E)$, and an arborescence $A \subseteq E$ with the guarantee that for every $e \in A$, e belongs to *some* minimum-cost arborescence in G. Can we conclude that A itself must be a minimum-cost arborescence in G? Give a proof, or a counter-example with explanation.

Solution. We can conclude that A must be a minimum-cost arborescence. Let r be the designated root vertex in $G = (V, E)$; recall that a set of edges $A \subseteq E$ forms an arborescence if and only if (i) each node other than r has in-degree 1 in (V, A), and (ii) r has a path to every other node in (V, A).

We claim that in a directed acyclic graph, any set of edges satisfying (i) must also satisfy (ii). (Note that this is certainly not true in an arbitrary directed graph.) For surpose that A satisfies (i) but not (ii), and let v be a node not reachable from r. Then if we repeatedly follow edges backwards starting at v, we must re-visit a node eventually, and this would be a cycle in G.

Thus, every way of choosing a single incoming edge for each $v \neq r$ yields an arborescence. It follows that an arborescence A has minimum cost if and only, for each $v \neq r$, the edge in A entering v has minimum cost over all edges entering v; and similarly, an edge (u, v) belongs to a minimum-cost arborescence if and only if it has minimum cost over all edges entering v.

Hence, if we are given an arborescence $A \subseteq E$ with the guarantee that for every $e \in A$, e belongs to *some* minimum-cost arborescence in G, then for each $e = (u, v)$, e has minimum cost over all edges entering v, and hence A is a minimum-cost arborescence.

25. Consider a directed graph $G = (V, E)$ with a root $r \in V$ and nonnegative costs on the edges. In this problem we consider variants of the min-cost arborescence algorithm.

(a) The algorithm discussed in Section **??** works as follows: we modify the costs, consider the subgraph of zero-cost edges, look for a directed cycle in this subgraph, and contract it (if one exists). Argue briefly that instead of looking for cycles, we can instead identify and contract strongly connected components of this subgraph.

(b) In the course of the algorithm, we defined y_v to be the min cost of an edge entering v, and we modified the costs of all edges e entering node v to be $c'_e = c_e - y_v$. Suppose we instead use the following modified cost: $c''_e = \max(0, c_e - 2y_v)$. This new change is likely to turn more edges 0 cost. Suppose, now we find an arborescence T of 0 cost. Prove that this T has cost at most twice the cost of the minimum cost arborescence in the original graph.

(c)$(*)$ Assume you do not find an arborescence of 0 cost. Contract all 0-cost strongly connected components, and recursively apply the same procedure on the resulting graph till an arborescence is found. Prove that this T has cost at most twice the cost of the minimum cost arborescence in the original graph.

26. $(*)$ Suppose you are given a directed graph $G = (V, E)$ in which each edge has a cost of either 0 or 1. Also, suppose that G has a node r such that there is a path from r to each other node in G. You are also given an integer k. Give a polynomial-time algorithm that either constructs an arborescence rooted at r of cost *exactly* k, or reports (correctly) that no such arborescence exists.

    Solution. Let (V, F) and (V, F') be distinct arborescences rooted at r. Consider the set of edges that are in one of F or F' but not the other; and over all such edges, let e be one whose distance to r in its arborescence is minimum. Suppose $e = (u, v) \in F'$. In (V, F), there is some other edge (w, v) entering v.

    Now define $F'' = F - (w, v) + e$. We claim that (V, F'') is also an arborescence rooted at r. Clearly F'' has exactly one edge entering each node, so we just need to verify that there is an r-x path for every node x. For those x such that the r-x path in (V, F) does not use v, the same r-x path exists in F''. Now consider an x whose r-x path in (V, F) does use v. Let Q denote the r-u path in (V, F'), and let P denote the v-x path in (V, F). Note that all the edges of P belong to F'', since they all belong to F and (w, v) is not among them. But we also have $Q \subseteq F \cap F'$, since e was the closest edge to r that belonged to one of F or F' but not the other. Thus in particular, $(w, v) \notin Q$, and hence $Q \subseteq F''$. Hence the concatenated path $Q \cdot e \cdot P \subseteq F''$, and so there is an r-x path in (V, F'').

    The arborescence (V, F'') has one more edge in common with (V, F') than (V, F) does. Performing a sequence of these operations, we can thereby transform (V, F) into (V, F') one edge at a time. But each of these operations changes the cost of the arborescence by at most 1 (since all edges have cost 0 or 1). So if we let (V, F) be a minimum-cost arborescence (of cost a) and we let (V, F') be a maximum-cost arborescence (of cost b), then if $a \le k \le b$, there must be an arborescence of cost exactly k.

    Note: The proof above follows the strategy of "swapping" from the min-cost arborescence to the max-cost arborescence, changing the cost by at most one every time. The swapping strategy is a little complicated — choosing the highest edge that is not in both arborescences — but some complication of this type seems necessary. To see this, consider what goes wrong with the following, simpler, swapping rule: find any edge $e' = (u, v)$ that is in F' but not in F; find the edge $e = (w, v)$ that enters v in F; and update F to be $F - e + e'$ The problem is that the resulting structure may not be an arborescence. For example, suppose V consists of the four nodes $\{0, 1, 2, 3\}$ with the root at 0, $F = \{(0, 1), (1, 2), (2, 3)\}$, and $F' = \{(0, 3), (3, 1), (1, 2)\}$. Then if we find $(3, 1)$ in F' and update F to be $F - (0, 1) + (3, 1)$, we end up with $\{(1, 2), (2, 3), (3, 1)\}$, which is not an arborescence.

4 Divide and Conquer

1. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values — so there are $2n$ values total — and you

may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the n^{th} smallest value.

However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\log n)$ queries.

Solution. Say A and B are the two databases and $A(i)$, $B(i)$ are i^{th} smallest elements of A, B.

First, let us compare the medians of the two databases. Let k be $\lceil \frac{1}{2}n \rceil$, then $A(k)$ and $B(k)$ are the medians of the two databases. Suppose $A(k) < B(k)$ (the case when $A(k) > B(k)$ would be the same with interchange of the role of A and B). Then one can see that $B(k)$ is greater than the first k elements of A. Also $B(k)$ is always greater than the first $k-1$ elements of B. Therefore $B(k)$ is at least $2k^{th}$ element in the combine databases. Since $2k \geq n$, all elements that are greater than $B(k)$ are greater than the median and we can eliminate the second part of the B database. Let B' be the half of B (i.e., the first k elements of B).

Similarly, the first $\lfloor \frac{1}{2}n \rfloor$ elements of A are less than $B(k)$, and thus, are less than the last $n-k+1$ elements of B. Also they are less than the last $\lceil \frac{1}{2}n \rceil$ elements of A. So, they are less than at least $n-k+1+\lceil \frac{1}{2}n \rceil = n+1$ elements of the combine database. It means that they are less than the median and we can eliminate them as well. Let A' be the remaining parts of A (i.e., the $\left[\lfloor \frac{1}{2}n \rfloor + 1; n\right]$ segment of A).

Now we eliminate $\lfloor \frac{1}{2}n \rfloor$ elements that are less than the median, and the same number of elements that are greater than median. It is clear that the median of the remaining elements is the same as the median of the original set of elements. We can find a median in the remaining set using recursion for A' and B'. Note that we can't delete elements from the databases. However, we can access i^{th} smallest elements of A' and B': the i^{th} smallest elements of A' is $i + \lfloor \frac{1}{2}n \rfloor^{th}$ smallest elements of A, and the i^{th} smallest elements of B' is i^{th} smallest elements of B.

Formally, the algorithm is the following. We write recursive function `median(`n`,`a`,`b`)` that takes integers n, a and b and find the median of the union of the two segments $A[a+1; a+n]$ and $B[b+1; b+n]$.

```
median($n$, $a$, $b$) if $n$=1 then return $\min(A(a+k), B(b+k))$ // base case
$k$=$\lceil \frac{1}{2}n \rceil$ if $A(a+k)$¡$B(b+k)$ then return median ($k$, $a + \lfloor \frac{1}{2}n \rfloor$ ,b) else return
median ($k$, $a$, $b + \lfloor \frac{1}{2}n \rfloor$)
```

To find median in the whole set of elements we evaluate `median(`n`,0,0)`.

Let $Q(n)$ be the number of queries asked by our algorithm to evaluate `median(`n`,`a`,`b`)`. Then it is clear that $Q(n) = Q(\lceil \frac{1}{2}n \rceil) + 2$. Therefore $Q(n) = 2\lceil \log n \rceil$.

Grader's comments. Most people had the overall structure of the solution right, but a number of solutions were missing some of the central ideas in the proof. Specifically, the proof needed to make two key points: first, in the recursive call, the median remains in the set of numbers considered; and second, the median value in the recursive call will in fact be the same as the median value in the original call. (Note that the second, stronger point is enough; but a number of people made the only the first poin and not the second.)

The algorithm cannot invoke the recursive call by simply saying, "Delete half of each database." The only way in which the algorithm can interact with the database is to pass queries to it; and so a conceptual "deletion" must in fact be implemented by keeping track of a particular interval under consideration in each database.

Also, the algorithm needs to keep track of the fact that the databases cannot always be divided evenly in half for the recursive call.

Finally, there was a category of solution which worked roughly as follows. A pointer m is kept into database D_1. If the m^{th} largest value in D_1 is between values $n-m$ and $n-m+1$ in D_2, then it is the median; but if it is smaller than both, the pointer m is replaced by $m+\lceil m/2 \rceil$. (There is a symmetric step to test if the $(n-m+1)^{\text{st}}$ largest value in D_2 is between values $m-1$ and m in D_1.) The problem with most of these solutions is that one could apply the pointer increment from m to $m+\lceil m/2 \rceil$. when m is a number as large as e.g. $(3/4)n$, which would cause a query to the database that is not between 1 and n. This type of solution is close to something that works correctly; however, to prevent this kind of difficult, it is necessary to keep track of a full interval in each database, rather than just a single pointer.

2. Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, \ldots, a_n$, which we assume be all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

   We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

3. $(*)$ *Hidden surface removal* is a problem in computer graphics that scarcely needs an introduction — when Woody is standing in front of Buzz you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, ... well, you get the idea.

   The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n non-vertical lines in the plane, labeled $L_1, \ldots, L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is *uppermost* at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0: $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x-coordinate at which it is uppermost — intuitively, some portion of it can be seen if you look down from "$y = \infty$."

Give an algorithm that takes n lines as input, and in $O(n \log n)$ time returns all of the ones that are visible. The accompanying figure gives an example.

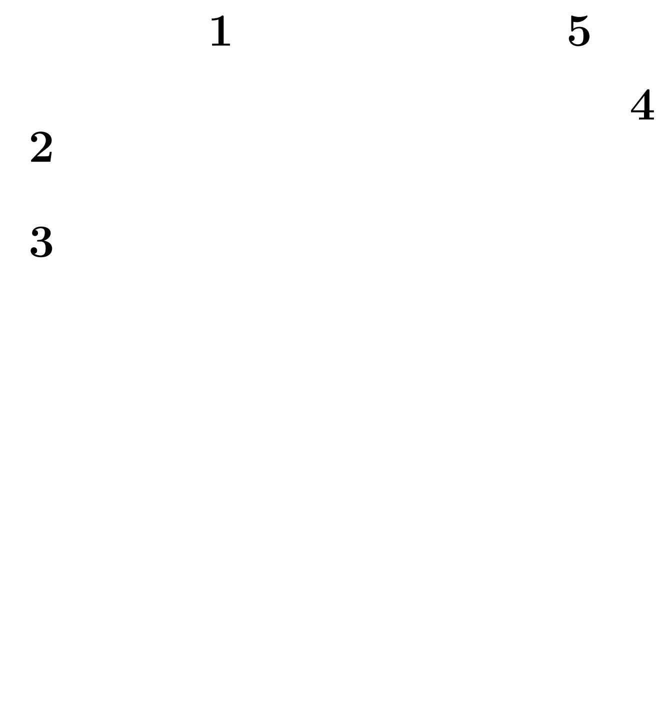


Figure 3: An instance with five lines (labeled "1"–"5" in the figure). All the lines except for "2" are visible.

Solution. We first label the lines in order of increasing slope, and then use a divide-and-conquer approach. If $n \leq 3$ — the base case of the divide-and-conquer approach — we can easily find the visible lines in constant time. (The first and third lines will always be visible; the second will be visible if and only if it meets the first line to the left of where the third line meets the first line.)

Let $m = \lceil n/2 \rceil$. We first recursively compute the sequence of visible lines among $L_1, \ldots, L_m$ — say they are $\mathcal{L} = \{L_{i_1}, \ldots, L_{i_p}\}$ in order of increasing slope. We also compute, in this recursive call, the sequence of points $a_1, \ldots, a_{p-1}$ where a_k is the intersection of line L_{i_k} with line $L_{i_{k+1}}$. Notice that $a_1, \ldots, a_{p-1}$ will have increasing x-coordinates; for if two lines are both visible, the region in which the line of smaller slope is uppermost lies to the left of the region in which the line of larger slope is uppermost. Similarly, we recursively compute the sequence $\mathcal{L}' = \{L_{j_1}, \ldots, L_{j_q}\}$ of visible lines among $L_{m+1}, \ldots, L_n$, together with the sequence of intersection points $b_k = L_{j_k} \cap L_{j_{k+1}}$ for $k = 1, \ldots, q-1$.

To complete the algorithm, we must show how to determine the visible lines in $\mathcal{L} \cup \mathcal{L}'$, together with the corresponding intersection pionts, in $O(n)$ time. (Note that $p+q \le n$, so it is enough to run in time $O(p+q)$.) We know that L_{i_1} will be visible, because it has the minimum slope among all the lines in this list; similarly, we know that L_{j_q} will be visible, because it has the maximum slope.

We merge the sorted lists $a_1, \ldots, a_{p-1}$ and $b_1, \ldots, b_{q-1}$ into a single list of points $c_1, c_2, \ldots, c_{p+q-2}$ ordered by increasing x-coordinate. This takes $O(n)$ time. Now, for each k, we consider the line that is uppermost in $\mathcal{L}$ at x-coordinate c_k, and the line that is uppermost in $\mathcal{L}'$ at x-coordinate c_k. Let ℓ be the smallest index for which the uppermost line in $\mathcal{L}'$ lies above the uppermost line in $\mathcal{L}$ at x-coordinate c_ℓ. Let the two lines at this point be $L_{i_s} \in \mathcal{L}$ and $L_{j_t} \in \mathcal{L}'$. Let (x^*, y^*) denote the point in the plane at which L_{i_s} and L_{j_t} intersect. We have thus established that x^* lies between the x-coordinates of $c_{\ell-1}$ and c_ℓ. This means that L_{i_s} is uppermost in $\mathcal{L} \cup \mathcal{L}'$ immediately to the left of x^*, and L_{j_t} is uppermost in $\mathcal{L} \cup \mathcal{L}'$ immediately to the right of x^*. Consequently, the sequence of visible lines among $\mathcal{L} \cup \mathcal{L}'$ is $L_{i_1}, \ldots, L_{i_s}, L_{j_t}, \ldots, L_{j_q}$; and the sequence of intersection points is $a_{i_1}, \ldots, a_{i_{s-1}}, (x^*, y^*), b_{j_t}, \ldots, b_{j_{q-1}}$. Since this is what we need to return to the next level of the recursion, the algorithm is complete.

4. Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n "smart-cards" that they've confiscated, suspecting them of being used in fraud. Each smart-card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many smart-cards corresponding to it, and we'll say that two smart-cards are *equivalent* if they correspond to the same account.

   It's very difficult to read the account number off a smart-card directly, but the bank has a high-tech "equivalence tester" that takes two smart-cards and, after performing some computations, determines whether they are equivalent.

   Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

   Solution. We give two solutions for this problem. The first solution is a divide and conquer algorithm, which is easier to think of. The second solution is a clever linear time algorithm.

   Via divide and conquer: Let $e_1, \ldots, e_n$ denote the equivalence classes of the cards: cards i and j are equivalent if $e_i = e_j$. What we are looking for is a value x so that more than $n/2$ of the indices have $e_i = x$.

   Divide the set of cards into two roughly equal piles: a set of $\lfloor n/2 \rfloor$ cards and a second set for the remaining $\lceil n/2 \rceil$ cards. We will recursively run the algorithm on the two sides, and will assume that if the algorithm finds an equivalence class containing more than half of the cards, then it returns a sample card in the equivalence class.

Note that if there are more than $n/2$ cards that are equivalent in the whole set, say have equivalence class x, than at least one of the two sides will have more than half the cards also equivalent to x. So at least one of the two recursive calls will return a card that has equivalence class x.

The reverse of this statement is not true: there can be a majority of equivalent cards in one side, without that equivalence class having more than $n/2$ cards overall (as it was only a majority on one side). So if a majority card is returned on either side we must test this card against all other cards.

```
If $|S| = 1$ return the one card
If $|S| = 2$ test if the two cards are equivalent
return either card if they are equivalent
Let $S_1$ be the set of the first $\lfloor n/2 \rfloor$ cards
Let $S_2$ be the set of the remaining cards
Call the algorithm recursively for $S_1$.
If a card is returned then test this against all other cards
If no card with majority equivalence has yet been found then call the algorithm recursively for $S_2$.
If a card is returned then test this against all other cards
Return a card from the majority equivalence class if one is found
```

The correctness of the algorithm follows from the observation above: that if there is a majority equivalence class, than this must be a majority equivalence class for at least one of the two sides.

To analyze the running time, let $T(n)$ denote the maximum number of tests the algorithm does for any set of n cards. The algorithm has two recursive calls, and does at most $2n$ tests outside of the recursive calls. So we get roughly the following recurrence (ignoring lower and upper integer parts for the moment)

$$T(n) \leq 2T(n/2) + 2n.$$

As we have seen in class, this recurrence implies that $T(n) = O(n \log n)$.

In linear time: Pair up all cards, and test all pairs for equivalence. If n was odd, one card is unmatched. For each pair that is not equivalent, discard both cards. For pairs that are equivalent, keep one of the two. Keep also the unmatched card, if n is odd. We can call this subroutine ELIMINATE.

The observation that leads to the linear time algorithm is as follows. If there is an equivalence class with more then $n/2$ cards, than the same equivalence class must also have more than half of the cards after calling ELIMINATE. This is true, as when we discard both cards in a pair, then at most one of them can be from the majority equivalence class. One call to ELIMINATE on a set of n cards takes $n/2$ tests, and as a result, we have only $\leq \lceil n/2 \rceil$ cards left. When we are down to a single card, then its equivalence is the only candidate for having a majority. We test this card against all others to check if its equivalence class has more than $n/2$ elements.

This method takes $n/2 + n/4 + \ldots$ tests for all the eliminates, plus $n-1$ tests for the final counting, for a total of less than $2n$ tests.

5 Dynamic Programming

✓1. Suppose you're running a lightweight consulting business — just you, two associates, and some rented equipment. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

Each month, you can either run your business from an office in New York (NY), or from an office in San Francisco (SF). In month i, you'll incur an *operating cost* of N_i if you run the business out of NY; you'll incur an operating cost of S_i if you run the business out of SF. (It depends on the distribution of client demands for that month.)

However, if you run the business out of one city in month i, and then out of the other city in month $i+1$, then you incur a fixed *moving cost* of M to switch base offices.

Given a sequence of n months, a *plan* is a sequence of n locations — each one equal to either NY or SF — such that the i^{th} location indicates the city in which you will be based in the i^{th} month. The *cost* of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city.

The problem is: Given a value for the moving cost M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, $M = 10$, and the operating costs are given by the following table.

	Month 1	Month 2	Month 3	Month 4
NY	1	3	20	30
SF	50	20	2	4

Then the plan of minimum cost would be the sequence of locations

$$[NY, NY, SF, SF],$$

with a total cost of $1 + 3 + 2 + 4 + 10 = 20$, where the final term of 10 arises because you change locations once.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For $i = 1$ to $n$
    If $N_i < S_i$ then
        Output "NY in Month $i$"
    Else
        Output "SF in Month $i$"
End
```

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an example of an instance in which every optimal plan must move (i.e. change locations) at least three times.

Provide an explanation, of at most three sentences, saying why your example has this property.

(c) Give an algorithm that takes values for n, M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, and returns the *cost* of an optimal plan.

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

Solution. **(a)** Suppose that $M = 10$, $\{N_1, N_2, N_3\} = \{1, 4, 1\}$, and $\{S_1, S_2, S_3\} = \{20, 1, 20\}$. Then the optimal plan would be $[NY, NY, NY]$, while this greedy algorithm would return $[NY, SF, NY]$.

(b) Suppose that $M = 10$, $\{N_1, N_2, N_3, N_4\} = \{1, 100, 1, 100\}$, and $\{S_1, S_2, S_3, S_4\} = \{100, 1, 100, 1\}$.

Explanation: The plan $[NY, SF, NY, SF]$ has cost 34, and it moves three times. Any other plan pays at least 100, and so is not optimal.

(c) The basic observation is: The optimal plan either ends in NY, or in SF. If it ends in NY, it will pay N_n plus one of the following two quantities:

- The cost of the optimal plan on $n - 1$ months, ending in NY, or
- The cost of the optimal plan on $n - 1$ months, ending in SF, plus a moving cost of M.

An analogous observation holds if the optimal plan ends in SF. Thus, if $OPT_N(j)$ denotes the minimum cost of a plan on months $1, \ldots, j$ ending in NY, and $OPT_S(j)$ denotes the minimum cost of a plan on months $1, \ldots, j$ ending in SF, then

$$OPT_N(n) = N_n + \min(OPT_N(n-1), M + OPT_S(n-1))$$

$$OPT_S(n) = S_n + \min(OPT_S(n-1), M + OPT_N(n-1))$$

This can be translated directly into an algorithm:

```
$OPT_N(0) = OPT_S(0) = 0$
For $i = 1, \ldots, n$
    $OPT_N(i) = N_i + \min(OPT_N(i-1), M + OPT_S(i-1))$
    $OPT_S(i) = S_i + \min(OPT_S(i-1), M + OPT_N(i-1))$
End
Return the smaller of $OPT_N(n)$ and $OPT_S(n)$
```

The algorithm has n iterations, and each takes constant time. Thus the running time is $O(n)$.

✓2. Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer *weight* w_i.

Consider, for example, the 5-node path drawn in the figure below. The *weights* are the numbers drawn next to the nodes.

The goal in this question is to solve the following algorithmic problem:

> (*) *Find an independent set in a path G whose total weight is as large as possible.*

(a) Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

> The "heaviest-first" greedy algorithm: Start with S equal to the empty set. While some node remains in G Pick a node v_i of maximum weight. Add v_i to S. Delete v_i and its neighbors from G. end while Return S

(b) Give an example to show that the following algorithm also *does not* always find an independent set of maximum total weight.

> Let S_1 be the set of all v_i where i is an odd number. Let S_2 be the set of all v_i where i is an even number. /* Note that S_1 and S_2 are both independent sets. */ Determine which of S_1 or S_2 has greater total weight, and return this one.

(c) Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

Solution. **(a)** Consider the sequence of weights 2, 3, 2. The greedy algorithm will pick the middle node, while the maximum weight independent set consists of the first and third.

(b) Consider the sequence of weights 3, 1, 2, 3. The given algorithm will pick the first and third nodes, while the maximum weight independent set consists of the first and fourth.

(c) Let S_i denote an independent set on $\{v_1, \ldots, v_i\}$, and let X_i denote its weight. Define $X_0 = 0$ and note that $X_1 = w_1$. Now, for $i > 1$, either v_i belongs to S_i or it doesn't. In the first case, we know that v_{i-1} cannot belong to S_i, and so $X_i = w_i + X_{i-2}$. In the second case, $X_i = X_{i-1}$. Thus we have the recurrence

$$X_i = \max(X_{i-1}, w_i + X_{i-2}).$$

We thus can compute the values of X_i, in increasing order from $i = 1$ to n. X_n is the value we want, and we can compute S_n by tracing back through the computations of the *max* operator. Since we spend constant time per iteration, over n iterations, the total running time is $O(n)$.

✓ 3. Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is a *line-graph* if it has the following properties:

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i, $i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

The length of a path is the number of edges in it. The goal in this question is to solve the following algorithmic problem:

Given a line-graph G, find the length of the longest path that begins at v_1 and ends at v_n.

Thus, a correct answer for the line-graph in the figure would be 3: the longest path from v_1 to v_n uses the three edges (v_1, v_2),(v_2, v_4), and (v_4, v_5).

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of a line-graph on which it does not return the correct answer.

Set $w = v_1$. Set $L = 0$ While there is an edge out of the node w Choose the edge (w, v_j) for which j is as small as possible. Set $w = v_j$ Increase L by 1. end while Return L as the length of the longest path.

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an algorithm that takes a line graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n. (Again, the *length* of a path is the number of edges in the path.)

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

Solution. **(a)** The graph on nodes $v_1, \ldots, v_5$ with edges (v_1, v_2), (v_1, v_3), (v_2, v_5), (v_3, v_4) and (v_4, v_5) is such an example. The algorithm will return 2 corresponding to the path of edges (v_1, v_2) and (v_2, v_5), while the optimum is 3 using the path (v_1, v_3), (v_3, v_4) and (v_4, v_5).

(b) The idea is to use dynamic programming. The simplest version to think of uses the subproblems $OPT[i]$ for the length of the longest path from v_1 to v_i. One point to be careful of is that not all nodes v_i necessarily have a path from v_1 to v_i. We will use the value "$-\infty$" for the $OPT[i]$ value in this case. We use $OPT(1) = 0$ as the longest path from v_1 to v_1 has 0 edges.

Long-path(n) Array $M[1 \dots n]$ $M[1] = 0$ For $i = 2, \dots, n$ $M = -\infty$ For all edges (j, i) then if $M[j] \neq -\infty$ if $M < M[j] + 1$ then $M = M[j] + 1$ endif endif endfor $M[i] = M$ endfor Return $M[n]$ as the length of the longest path.

The running time is $O(n^2)$ if you assume that all edges entering a node i can be listed in $O(n)$ time.

✓ 4. Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are *low-stress* (e.g. setting up a Web site for a class of fifth-graders at the local elementary school) and those that are *high-stress* (e.g. protecting America's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers.) The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i, then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i - 1$.

So given a sequence of n weeks, a *plan* is specified by a choice of "low-stress", "high-stress", or "none" for each of the n weeks — with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i - 1$. (It's okay to choose a high-stress job in week 1.) The *value* of the plan is determined in the natural way: for each i, you add ℓ_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

The problem is: Given sets of values $\ell_1, \ell_2, \dots, \ell_n$ and $h_1, h_2, \dots, h_n$, find a plan of maximum value. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, and the values of ℓ_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0 + 50 + 10 + 10 = 70$.

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

For iterations $i = 1$ to n If $h_{i+1} > \ell_i + \ell_{i+1}$ then Output "Choose no job in week i" Output "Choose a high-stress job in week $i+1$" Continue with iteration $i+2$ Else Output "Choose a low-stress job in week i" Continue with iteration $i+1$ Endif End

To avoid problems with overflowing array bounds, we define $h_i = \ell_i = 0$ when $i > n$.

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an algorithm that takes values for $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$, and returns the *value* of an optimal plan.

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

5. Suppose you are managing the construction of billboards on the Stephen Daedalus Memorial Highway, a heavily-traveled stretch of road that runs west-east for M miles. The possible sites for billboards are given by numbers $x_1, x_2, \ldots, x_n$, each in the interval $[0, M]$ (specifying their position along the highway, measured in miles from its western end). If you place a billboard at location x_i, you receive a revenue of $r_i > 0$.

You want to place billboards at a subset of the sites in $\{x_1, \ldots, x_n\}$ so as to maximize your total revenue, subject to the following restrictions.

(i) (*The environmental constraint.*) You cannot build two billboards within less than 5 miles of one another on the highway.

(ii) (*The boundary constraint.*) You cannot build a billboard within less than 5 miles of the western or eastern ends of the highway.

A subset of sites satisfying these two restrictions will be called *valid.*

Example. Suppose $M = 20$, $n = 4$,

$$\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\},$$

and

$$\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}.$$

Then the optimal solution would be to place billboards at x_1 and x_3, for a total revenue of 10.

Give an algorithm that takes an instance of this problem as input, and returns the maximum total revenue that can be obtained from any valid subset of sites.

The running time of the algorithm should be polynomial in n. Include a brief analysis of the running time of your algorithm, and a proof that it is correct.

✓ 6. You're trying to run a large computing job, in which you need to simulate a physical system for as many discrete *steps* as you can. The lab you're working in has two large supercomputers (which we'll call A and B) which are capable of processing this job. However, you're not one of the high-priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available.

Here's the problem you're faced with. Your job can only run on one of the machines in any given minute. Over each of the next n minutes you have a "profile" of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps of the simulation if your job is on machine A, and $b_i > 0$ steps of the simulation if your job is on machine B. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job.

So given a sequence of n minutes, a *plan* is specified by a choice of A, B, or "*move*" for each minute — with the property that choices A and B cannot appear in consecutive minutes. E.g. if your job is on machine A in minute i, and you want to switch to machine B, then your choice for minute $i+1$ must be *move*, and then your choice for minute $i+2$ can be B. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so it's the sum of a_i over all minutes in which the job is on A, plus the sum of b_i over all minutes in which the job is on B.

The problem is: Given values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, find a plan of maximum value. (Such a strategy will be called *optimal*.) Note that your plan can start with either of the machines A or B in minute 1.

Example. Suppose $n = 4$, and the values of a_i and b_i are given by the following table.

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Then the plan of maximum value would be to choose A for minute 1, then *move* for minute 2, and then B for minutes 3 and 4. The value of this plan would be $10 + 0 + 20 + 20 = 50$.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

In minute 1, choose machine achieving the larger of a_1, b_1. Set $i = 2$ While $i \leq n$ What was the choice in minute $i - 1$? If A: If $b_{i+1} > a_i + a_{i+1}$ then Choose *move* in minute i and B in minute $i + 1$ Proceed to iteration $i + 2$ Else Choose A in minute i Proceed to iteration $i + 1$ Endif If B: behave as above with roles of A and B reversed. EndWhile

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an algorithm that takes values for $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ and returns the *value* of an optimal plan.

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

Solution. Here are two examples:

	Minute 1	Minute 2
A	2	10
B	1	20

The greedy algorithm would choose A for both steps, while the optimal solution would be to choose B for both steps.

	Minute 1	Minute 2	Minute 3	Minute 4
A	2	1	1	200
B	1	1	20	100

The greedy algorithm would choose A, then move, then choose B for the final two steps. The optimal solution would be to choose A for all four steps.

(1b) Let $Opt(i, A)$ denote the maximum value of a plan in minutes 1 through i that ends on machine A, and define $Opt(i, B)$ analogously for B.

Now, if you're on machine A in minute i, where were you in minute $i - 1$? Either on machine A, or in the process of moving from machine B. In the first case, we have $Opt(i, A) = a_i + Opt(i - 1, A)$. In the second case, since you were last at B in minute $i - 2$, we have $Opt(i, A) = a_i + Opt(i - 2, B)$. Thus, overall, we have

$$Opt(i, A) = a_i + \max(Opt(i - 1, A), Opt(i - 2, B)).$$

A symmetric formula holds for $Opt(i, B)$.

The full algorithm initializes $Opt(1, A) = a_1$ and $Opt(1, B) = b_1$. Then, for $i = 2, 3, \ldots, n$, it computes $Opt(i, A)$ and $Opt(i, B)$ using the recurrence. This takes constant time for each of $n - 1$ iterations, and so the total time is $O(n)$.

Here is an alternate solution. Let $Opt(i)$ be the maximum value of a plan in minutes 1 through i. Also, initialize $Opt(-1) = Opt(0) = 0$. Now, in minute i, we ask: when was the most recent minute in which we moved? If this was minute $k-1$ (where perhaps $k-1=0$), then $Opt(i)$ would be equal to the best we could do up through minute $k-2$, followed by a move in minute $k-1$, followed by the best we could do on a single machine from minutes k through i. Thus, we have

$$Opt(i) = \max_{1 \le k \le i} Opt(k-2) + \max\left[\sum_{\ell=k}^{i} a_\ell, \sum_{\ell=k}^{i} b_\ell\right].$$

The full algorithm then builds up these values for $i = 2, 3, \ldots, n$. Each iteration takes $O(n)$ time to compute the maximum, so the total running time is $O(n^2)$.

The most common error was to use a single-variable set of sub-problems as in the second correct solution (using $Opt(i)$ to denote the maximum value of a plan in minutes 1 through i), but with a recurrence that computed $Opt(i)$ by looking only at $Opt(i-1)$ and $Opt(i-2)$. For example, a common recurrence was to let m_j denote the machine on which the optimal plan for minutes 1 through j ended, let $c(m_j)$ denote the number of steps available on machine m_j in minute j, and then write $Opt(i) = \max(Opt(i-1) + c(m_{i-1}), Opt(i-2) + c(m_{i-2}))$. But if we consider an example like

	Minute 1	Minute 2	Minute 3
A	2	10	200
B	1	20	100

then $Opt(1) = 2$, $Opt(2) = 21$, $m_1 = A$, and $m_2 = B$. But $Opt(3) = 212$, which does not follow from the recurrence above. There are a number of variations on the above recurrence, but they all break on this example.

7. Suppose you're consulting for a small computation-intensive investment company, and they have the following type of problem that they want to solve over and over. A typical instance of the problem is: they're doing a simulation in which they look at n consecutive days of a given stock, at some point in the past. Let's number the days $i = 1, 2, \ldots, n$; for each day i, they have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) Suppose during this time period, they wanted to buy 1000 shares on some day, and sell all these shares on some (later) day. They want to know: when should they have bought and when should they have sold in order to have made as much money as possible? (If there was no way to make money during the n days, you should report this instead.)

   Example: Suppose $n = 3$, $p(1) = 9$, $p(2) = 1$, $p(3) = 5$. Then you should return *"buy on* 2*, sell on* 3*"*; i.e. buying on day 2 and selling on day 3 means they would have made \$4 per share, the maximum possible for that period.

Clearly, there's a simple algorithm that takes time $O(n^2)$: try all possible pairs of buy/sell days and see which makes them the most money. Your investment friends were hoping for something a little better.

Show how to find the correct numbers i and j in time $O(n)$.

Solution. Let X_j (for $j = 1, \ldots, n$) denote the maximum possible return the investors can make if they sell the stock on day j. Note that $X_1 = 0$. Now, in the optimal way of selling the stock on day j, the investors were either holding it on day $j-1$ or there weren't. If they weren't, then $X_j = 0$. If they were, then $X_j = X_{j-1} + (p(j) - p(j-1))$. Thus, we have

$$X_j = \max(0, X_{j-1} + (p(j) - p(j-1))).$$

Finally, the answer is the maximum, over $j = 1, \ldots, n$, of X_j.

8. Eventually your friends from the previous problem move up to more elaborate simulations, and they're hoping you can still help them out. As before, they're looking at n consecutive days of a given stock, at some point in the past. The days are numbered $i = 1, 2, \ldots, n$; for each day i, they have a price $p(i)$ per share for the stock on that day.

   For certain (possibly large) values of k, they want to study what they call *k-shot strategies.* A k-shot strategy is a collection of m pairs of days $(b_1, s_1), \ldots, (b_m, s_m)$, where $0 \leq m \leq k$ and

$$1 \leq b_1 < s_1 < b_2 < s_2 \cdots < b_m < s_m \leq n.$$

   We view these as a set of up to k non-overlapping intervals, during each of which the investors buy 1000 shares of the stock (on day b_i) and then sell it (on day s_i). The *return* of a given k-shot strategy is simply the profit obtained from the m buy-sell transactions, namely

$$1000 \sum_{i=1}^{m} p(s_i) - p(b_i).$$

   The investors want to assess the value of k-shot strategies by running simulations on their n-day trace of the stock price. Your goal is to design an efficient algorithm that determines, given the sequence of prices, the k-shot strategy with the maximum possible return. Since k may be relatively large in these simulations, your running time should be polynomial in both n and k; it should not contain k in the exponent.

   Solution. By *transaction* (i, j), we mean the single transaction that consists of buying on day i and selling on day j. Let $P[i, j]$ denote the monetary return from transaction (i, j). Let $Q[i, j]$ denote the maximum profit obtainable by executing a single transaction somewhere in the interval of days between i and j. Note that the transaction achieving the maximum in $Q[i, j]$ is either the transaction (i, j), or else it fits into one of the intervals $[i, j-1]$ or $[i+1, j]$. Thus we have

$$Q[i, j] = \max(P[i, j], Q[i, j-1], Q[i+1, j]).$$

Using this formula, we can build up all values of $Q[i, j]$ in time $O(n^2)$. (By going in order of increasing $i + j$, spending constant time per entry.)

Now, let us say that an *m-exact strategy* is one with *exactly* m non-overlapping buy-sell transactions. Let $M[m, d]$ denote the maximum profit obtainable by an m-exact strategy on days $1, \ldots, d$, for $0 \leq m \leq k$ and $0 \leq d \leq n$. We will use $-\infty$ to denote the profit obtainable if there isn't room in days $1, \ldots, d$ to execute m transactions. (E.g. if $d < 2m$.) We can initialize $M[m, 0] = -\infty$ and $M[0, d] = -\infty$ for each m and each d.

In the optimal m-exact strategy on days $1, \ldots, d$, the final transaction occupies an interval that begins at i and ends at j, for some $1 \leq i < j \leq d$; and up to day $i - 1$ we then have an $(m - 1)$-exact strategy. Thus we have

$$M[m, d] = \max_{1 \leq i < j \leq d} Q[i, j] + M[m - 1, i - 1].$$

We can fill in these entries in order of increasing $m + d$. The time spent per entry is $O(n)$, since we've already computed all $Q[i, j]$. Since there are $O(kn)$ entries, the total time is therefore $O(kn^2)$. We can determine the strategy associated with each entry by maintaining a pointer to the entry that produced the maximum, and tracing back through the dynamic programming table using these pointers.

Finally, the optimal k-shot strategy is, by definition, an m-exact strategy for some $m \leq k$; thus, the optimal profit from a k-shot strategy is

$$\max_{0 \leq m \leq k} M[m, n].$$

✓ 9. Suppose you're consulting for a company that manufactures PC equipment, and ships it to distributors all over the country. For each of the next n weeks they have a projected *supply* s_i of equipment (measured in pounds), which has to be shipped by an air freight carrier.

Each week's supply can be carried by one of two air freight companies, A or B.

- Company A charges a fixed rate r per pound (so it costs $r \cdot s_i$ to ship a week's supply s_i).
- Company B makes contracts for a fixed amount c per week, independent of the weight. However, contracts with company B must be made in blocks of 4 consecutive weeks at a time.

A *schedule*, for the PC company, is a choice of air freight company (A or B) for each of the n weeks, with the restriction that company B, whenever it is chosen, must be chosen for blocks of 4 contiguous weeks at a time. The *cost* of the schedule is the total amount paid to A and B, according to the description above.

Give a polynomial-time algorithm that takes a sequence of supply values $s_1, s_2, \ldots, s_n$, and returns a *schedule* of minimum cost.

Example: Suppose $r = 1$, $c = 10$, and the sequence of values is

$$11, 9, 9, 12, 12, 12, 12, 9, 9, 11.$$

Then the optimal schedule would be to choose company A for the first three weeks, then company B for a blocks of 4 consecutive weeks, and then company A for the final three weeks.

10. Suppose it's nearing the end of the semester and you're taking n courses, each with a final project that still has to be done. Each project will be graded on the following scale: it will be assigned an integer number on a scale of 1 to $g > 1$, higher numbers being better grades. Your goal, of course, is to maximize your average grade on the n projects.

Now, you have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume H is a positive integer, and you'll spend an integer number of hours on each project. So as to figure out how best to divide up your time, you've come up with a set of functions $\{f_i : i = 1, 2, \ldots, n\}$ (rough estimates, of course) for each of your n courses; if you spend $h \leq H$ hours on the project for course i, you'll get a grade of $f_i(h)$. (You may assume that the functions f_i are *non-decreasing*: if $h < h'$ then $f_i(h) \leq f_i(h')$.)

So the problem is: given these functions $\{f_i\}$, decide how many hours to spend on each project (in integer values only) so that your average grade, as computed according to the f_i, is as large as possible. In order to be efficient, the running time of your algorithm should be polynomial in n, g, and H; none of these quantities should appear as an exponent in your running time.

Solution. First note that it is enough to maximize one's *total* grade over the n courses, since this differs from the average grade by the fixed factor of n. Let the (i, h)*-subproblem* be the problem in which one wants to maximize one's grade on the first i courses, using at most h hours.

Let $A[i, h]$ be the maximum total grade that can be achieved for this subproblem. Then $A[0, h] = 0$ for all h, and $A[i, 0] = \sum_{j=1}^{i} f_j(0)$. Now, in the optimal solution to the (i, h)-subproblem, one spends k hours on course i for some value of $k \in [0, h]$; thus

$$A[i, h] = \max_{0 \leq k \leq h} f_i(k) + A[i - 1, h - k].$$

We also record the value of k that produces this maximum. Finally, we output $A[n, H]$, and can trace-back through the entries using the recorded values to produce the optimal distribution of time. The total time to fill in each entry $A[i, h]$ is $O(H)$, and there are nH entries, for a total time of $O(nH^2)$.

11. A large collection of mobile wireless devices can naturally form a network in which the devices are the nodes, and two devices x and y are connected by an edge if they are able to directly communicate with one another (e.g. by a short-range radio link). Such a network of wireless devices is a highly dynamic object, in which edges can appear and disappear over time as the devices move around. For instance, an edge (x, y) might disappear as x and y move far apart from one another and lose the ability to communicate directly.

In a network that changes over time, it is natural to look for efficient ways of *maintaining* a path between certain designated nodes. There are two opposing concerns in maintaining such a path: we want paths that are short, but we also do not want to have to change the path frequently as the network structure changes. (I.e. we'd like a single path to continue working, if possible, even as the network gains and loses edges.) Here is a way we might model this problem.

Suppose we have a set of mobile nodes V, and at a particular point in time there is a set E_0 of edges among these nodes. As the nodes move, the set of edges changes from E_0 to E_1, then to E_2, then to E_3, and so on to an edge set E_b. For $i = 0, 1, 2, \ldots, b$, let G_i denote the graph (V, E_i). So if we were to watch the structure of the network on the nodes V as a "time lapse", it would look precisely like the sequence of graphs $G_0, G_1, G_2, \ldots, G_{b-1}, G_b$. We will assume that each of these graphs G_i is connected.

Now, consider two particular nodes $s, t \in V$. For an s-t path P in one of the graphs G_i, we define the *length* of P to be simply the number of edges in P, and denote this $\ell(P)$. Our goal is to produce a sequence of paths $P_0, P_1, \ldots, P_b$ so that for each i, P_i is an s-t path in G_i. We want the paths to be relatively short. We also do not want there to be too many *changes* — points at which the identity of the path switches. Formally, we define *changes*$(P_0, P_1, \ldots, P_b)$ to be the number of indices i $(0 \leq i \leq b-1)$ for which $P_i \neq P_{i+1}$

Fix a constant $K > 0$. We define the *cost* of the sequence of paths $P_0, P_1, \ldots, P_b$ to be

$$cost(P_0, P_1, \ldots, P_b) = \sum_{i=0}^{b} \ell(P_i) + K \cdot changes(P_0, P_1, \ldots, P_b).$$

(a) Suppose it is possible to choose a single path P that is an s-t path in each of the graphs $G_0, G_1, \ldots, G_b$. Give a polynomial-time algorithm to find the shortest such path.

(b) Give a polynomial-time algorithm to find a sequence of paths $P_0, P_1, \ldots, P_b$ of minimum cost, where P_i is an s-t path in G_i for $i = 0, 1, \ldots, b$.

Solution. **(b)** We are given graphs $G_0, \ldots, G_b$. While trying to find the last path P_b, we have several choices. If G_b contains P_{b-1}, then we may use P_{b-1}, adding $l(P_{b-1})$ to the cost function (but not adding the cost of change K.) Another option is to use the shortest s-t path, call it S_b, in G_b. This adds $l(S_b)$ and the cost of change K to the cost function. However, we may want to make sure that in G_{b-1} we use a path that is also available in G_b so we can avoid the change penalty K. This effect of G_b on the earlier part of the solution is hard to anticipate in a greedy-type algorithm, so we'll use dynamic programming.

We will use subproblems $Opt(i)$ to denote minimum cost of the solution for graphs $G_0, \ldots, G_i$.

To compute $Opt(n)$ it seems most useful to think about where the last changeover occurs. Say the last changeover is between graphs G_i and G_{i+1}. This means that we use the path P in graphs $G_{i+1}, \ldots, G_b$, hence the edges of P must be in every one of these graphs.

Let $G(i, j)$ for any $0 \leq i \leq j \leq b$ denote the graph consisting of the edges that are common in $G_i, \ldots, G_j$; and let $\ell(i, j)$ be the length of the shortest path from s to t in this graph (where $\ell(i, j) = \infty$ if no such path exists).

If the last change occurs between graphs G_i and G_{i+1} then then we get that $Opt(b) = Opt(i) + (b - i)\ell(i + 1, b) + K$. We have to deal separately with the special case when there are no changes at all. In that case $Opt(b) = (b + 1))\ell(0, b)$.

So we get argued that $Opt(b)$ can be expressed via the following recurrence:

$$Opt(b) = \min[(b + 1))\ell(0, b), \min_{1 \leq i < b} Opt(i) + (b - i)\ell(i + 1, b) + K].$$

Our algorithm will first compute all $G(i, j)$ graphs and $\ell(i, j)$ values for all $1 \leq i \leq j \leq b$. There are $O(b^2)$ such pairs and to compute one such subgraph can take $O(n^2 b)$ time, as there are up to $O(n^2)$ edges to consider in each of at most b graphs. We can compute the shortest path in each graph in linear time via BFS. This is a total of $O(n^2 b^3)$ time, polynomial but really slow. We can speed things up a bit to $O(b^2 n^2)$ by computing the graphs $G(i, j)$ and $\ell(i, j)$ for a fixed value of i in order of $j = i \ldots b$.

Once we have precomputed these values the algorithm to compute the optimal values is simple and takes only $O(b^2)$ time. We will use $M[0 \ldots b]$ to store the optimal values.

```
For i=0,...,b  $M[i] = \min((i + 1)\ell(0, i); \min_{1 \leq j < i} M[j] + (i - j)\ell(j + 1, i))$
EndFor
```

12. Recall the scheduling problem from the text in which we sought to minimize the maximum *lateness*. There are n jobs, each with a deadline d_i and a required processing time t_i, and all jobs are available to be scheduled starting at time s. For a job i to be done it needs to be assigned a period from $s_i \geq s$ to $f_i = s_i + t_i$, and different jobs should be assigned non-overlapping intervals. As usual, an assignment of times in this way will be called a *schedule*.

    In this problem, we consider the same set-up, but want to optimize a different objective. In particular, we consider the case in which each job must either be done by its deadline or not at all. We'll say that a subset J of the jobs is *schedulable* if there is a schedule for the jobs in J so that each of them finishes by its deadline. Your problem is to select a schedulable subset of maximum possible size, and give a schedule for this subset that allows each job to finish by its deadline.

(a) Prove that there is an optimal solution J (i.e. a schedulable set of maximum size) in which the jobs in J are scheduled in increasing order of their deadlines.

(b) Assume that all deadlines d_i and required times t_i are integers. Give an algorithm to find an optimal solution. Your algorithm should run in time polynomial in the number of jobs n, and the maximum deadline $D = \max_i d_i$.

Solution. **(a)** Let J be the optimal subset. By definition all jobs in J can be scheduled to meet their deadline. Now consider the problem of scheduling to minimize the maximum lateness from class, but consider the jobs in J only. We know by the definition of J that the minimum lateness is 0 (i.e., all jobs can be scheduled in time), and in class we showed that the greedy algorithm of scheduling jobs in the order of their deadline, is optimal for minimizing maximum lateness. Hence ordering the jobs in J by the deadline generates a feasible schedule for this set of jobs.

(b) The problem is analogous to the subset selection problem we considered in class. We will have subproblems analogous to the subproblems for that problem. The first idea is to consider subproblems using a subset of jobs $\{1, \ldots, m\}$. As always we will order the jobs by increasing deadline, and we will assume that they are numbered this way, i.e., we have that $d_1 \leq \ldots \leq d_n = D$. To solve the original problem we consider two cases: either the last job n is accepted or it is rejected. If job n is rejected, then the problem reduces to the subproblem using only the first $n-1$ items. Now consider the case when job n is accepted. By part (a) we know that we may assume that job n will be scheduled last. In order to make sure that the machine can finish job n by its deadline D, all other jobs accepted by the schedule should be done by time $D - t_n$. We will define subproblems so that this problem is one of our subproblems.

For a time $0 \leq d \leq D$ and $m = 0, \ldots, n$ let $OPT(d, m)$ denote the the maximum subset of requests in the set $\{1, \ldots, m\}$ that can be satisfied by the deadline d. What we mean is that in this subproblem the machine is no longer available after time d, so all requests either have to be scheduled to be done by deadline d, or should be rejected (even if the deadline d_i of the job is $d_i > d$). Now we have the following statement.

(5.1)

- *If job m* **is not** *in the optimal solution $OPT(d, m)$ then $OPT(m, d) = OPT(m-1, d)$.*

- *If job m* **is** *in the optimal solution $OPT(m, d)$ then $OPT(m, d) = OPT(m-1, d-t_m) + 1$.*

This suggests the following code.

```
Select-Jobs(n,D)
  Array $M[0\dots n, 0\dots D]$
  Array $S[0\dots n, 0\dots D]$
  For $d = 0,\dots,D$
    $M[0,d] = 0$
    $S[0,d] = \phi$
  Endfor
  For $m = 1,\dots,n$
    For $d = 0,\dots,D$
      If $M[m-1,d] > M[m-1,d-t_m]+1$ then
        $M[m,d] = M[m-1,d]$
        $S[m,d] = S[m-1,d]$
      Else
        $M[m,d] = M[m-1,d-t_m]+1$
        $S[m,d] = S[m-1,d-t_m]\cup\{m\}$
      Endif
    Endfor
  Endfor
  Return $M[n,D]$ and $S[n,D]$
```

The correctness follows immediately from the statement (5.1) . The running time of $O(n^2D)$ is also immediate from the for loops in the problem, there are two nested `for` loops for for m and one for d. This means that the internal part of the loop gets invoked $O(nD)$ time. The internal part of this `for` loop takes $O(n)$ time, as we explicitly maintain the optimal solutions. The running time can be improved to $O(nD)$ by not maintaining the S array, and only recovering the solution later, once the values are known.

13. Consider the sequence alignment problem over a four-letter alphabet $\{z_1, z_2, z_3, z_4\}$, with a cost δ for each insertion or deletion, and a cost α_{ij} for a substitution of z_i by z_j (for each pair $i \neq j$). Assume that δ and each α_{ij} is a positive integer.

Suppose you are given two strings $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$, and a proposed alignment between them. Give an $O(mn)$ algorithm to decide whether this alignment is the *unique* minimum-cost alignment between A and B.

Solution. Consider the directed acyclic graph $G = (V, E)$ constructed in class, with vertices s in the upper left corner and t in the lower right corner, whose s-t paths correspond to global alignments between A and B. For a set of edges $F \subset E$, let $c(F)$ denote the total cost of the edges in F. If P is a path in G, let $\Delta(P)$ denote the set of diagonal edges in P (i.e. the *matches* in the alignment).

Let Q denote the s-t path corresponding to the given alignment. Let E_1 denote the horizontal or vertical edges in G (corresponding to indels), E_2 denote the diagonal edges in G that do not belong to $\Delta(Q)$, and $E_3 = \Delta(Q)$. Note that $E = E_1 \cup E_2 \cup E_3$.

Let $\varepsilon = 1/2n$ and $\varepsilon' = 1/4n^2$. We form a graph G' by subtracting ε from the cost of every edge in E_2 and adding ε' to the cost of every edge in E_3. Thus, G' has the same structure as G, but a new cost function c'.

Now we claim that path Q is a minimum-cost s-t path in G' if and only if it is the unique minimum-cost s-t path in G. To prove this, we first observe that

$$c'(Q) = c(Q) + \varepsilon'|\Delta(Q)| \leq c(Q) + \frac{1}{4},$$

and if $P \neq Q$, then

$$c'(P) = c(P) + \varepsilon'|\Delta(P \cap Q)| - \varepsilon|\Delta(P - Q)| \geq c(P) - \frac{1}{2}.$$

Now, if Q was the unique minimum-cost path in G, then $c(Q) \leq c(P) + 1$ for every other path P, so $c'(Q) < c'(P)$ by the above inequalities, and hence Q is a minimum-cost s-t path in G'. To prove the converse, we observe from the above inequalities that $c'(Q) - c(Q) > c'(P) - c(P)$ for every other path P; thus, if Q is a minimum-cost path in G', it is the unique minimum-cost path in G.

Thus, the algorithm is to find the minimum cost of an s-t path in G', in $O(mn)$ time and $O(m + n)$ space by the algorithm from class. Q is the unique minimum-cost A-B alignment if and only if this cost matches $c'(Q)$.

14. Consider the following inventory problem. You are running a store that sells some large product (let's assume you sell trucks), and predictions tell you the quantity of sales to expect over the next n months. Let d_i denote the number of sales you expect in month i. We'll assume that all sales happen at the beginning of the month, and trucks that are not sold are *stored* until the beginning of the next month. You can store at most S trucks, and it costs C to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee of K each time you place an order (regardless of the number of trucks you order). You start out with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands $\{d_i\}$, and minimize the costs. In summary:

    - There are two parts to the cost. First, storage: it costs C for every truck on hand that is not needed that month. Scecond, ordering fees: it costs K for every order placed.
    - In each month you need enough trucks to satisfy the demand d_i, but the amount left over after satisfying the demand for the month should not exceed the inventory limit S.

    Give an algorithm that solves this problem in time that is polynomial in n and S.

    Solution. The subproblems will be as follows: The optimum way to satisfy orders $1, \ldots, i$ with having an inventory of s truck left over after the month i. Let $OPT(i, s)$ denote the value of the optimal solution for this subproblem.

    - The problem we want to solve is $OPT(n, 0)$ as we do not need any leftover inventory at the end.
    - The number of subproblems is $n(S + 1)$ as there could be $0, 1, \ldots, S$ trucks left over after a period.
    - To get the solution for a subproblem $OPT(i, s)$ given the values of the previous subproblems, we have to try every possible number of trucks that could have been left over after the previous period. If the previous period had z trucks left over, then so far we paid $OPT(i - 1, z)$ and now we have to pay zC for storage. In order to satisfy the demand of d_i and have s trucks left over, we need $s + d_i$ trucks. If $z < s + d_i$ we have to order more, and pay the ordering fee of K.

In summary the cost $OPT(i,s)$ is obtained by taking the smaller of $OPT(i-1, s+d_i) + C(s+d_i)$ (if $s + d_i \leq S$), and the minimum over smaller values of z, $\min_{z<\min(s+d_i,S)}(OPT(i-1,z) + zC + K)$.

We can also observe that the minimum in this second term is obtained when $z = 0$ (if we have to reorder anyhow, why pay storage for any extra trucks). With this extra observation we get that

- if $s + d_i > S$ then $OPT(i,s) = OPT(i-1,0) + K$,
- else $OPT(i,s) = \min(OPT(i-1, s+d_i) + C(s+d_i), OPT(i-1,0) + K)$.

15. You are consulting for an independently operated gas-station, and it faced with the following situation. They have a large underground tank in which they store gas; the tank can hold up to L gallons at one time. Ordering gas is quite expensive, so they want to order relatively rarely. For each order they need to pay a fix price P for delivery in addition to the cost of the gas ordered. However, it cost c to store a gallon of gas for an extra day, so ordering too much ahead increases the storage cost. They are planning to close for winder break, and want their tank to be empty, as they are afraid that any gas left in the tank would freeze over during break. Luckily, years of experience gives them accurate projections for how much gas they will need each day until winter break. Assume that there are n days left till break, and they need g_i gallons of gas for each of day $i = 1, ..., n$. Assume that the tank is empty at the end of day 0. Give an algorithm to decide which days they should place orders, and how much to order to minimize their total cost.

    The following two observations might help.

    - If $g_1 > 0$ then the first order has to arrive the morning of day 1.
    - If the next order is due to arrive on day i, then the amount ordered should be $\sum_{j=1}^{i-1} g_j$.

16. Through some friends of friends, you end up on a consulting visit to the cutting-edge biotech firm Clones 'R' Us. At first you're not sure how your algorithmic background will be of any help to them, but you soon find yourself called upon to help two identical-looking software engineers tackle a perplexing problem.

    The problem they are currently working on is based on the *concatenation* of sequences of genetic material. If X and Y are each strings over a fixed alphabet Σ, then XY denotes the string obtained by *concatenating* them — writing X followed by Y. CRU has identified a "target sequence" A of genetic material, consisting of m symbols, and they want to produce a sequence that is as similar to A as possible. For this purpose, they have a set of (shorter) sequences $B_1, B_2, \ldots, B_k$, consisting of $n_1, n_2, \ldots, n_k$ symbols respectively. They can cheaply produce any sequence consisting of copies of the strings in $\{B_i\}$ concatenated together (with repetitions allowed).

    Thus, we say that a *concatenation over* $\{B_i\}$ is any sequence of the form $B_{i_1}B_{i_2}\cdots B_{i_\ell}$, where each $i_j \in \{1, 2, \ldots, k\}$. So B_1, $B_1B_1B_1$, and $B_3B_2B_1$ are all concatenations over $\{B_i\}$. The problem is to find a concatenation over $\{B_i\}$ for which the sequence alignment cost is as small as possible. (For the purpose of computing the sequence alignment cost, you may assume that you are given a cost δ for each insertion or deletion, and a substitution cost α_{ij} for each pair $i, j \in \Sigma$.)

Give a polynomial-time algorithm for this problem.

✓17. Suppose we want to replicate a file over a collection of n servers, labeled $S_1, S_2, \ldots, S_n$. To place a copy of the file at server S_i results in a *placement cost* of c_i, for an integer $c_i > 0$.

Now, if a user requests the file from server S_i, and no copy of the file is present at S_i, then the servers $S_{i+1}, S_{i+2}, S_{i+3} \ldots$ are searched in order until a copy of the file is finally found, say at server S_j, where $j > i$. This results in an *access cost* of $j - i$. (Note that the lower-indexed servers $S_{i-1}, S_{i-2}, \ldots$ are not consulted in this search.) The access cost is 0 if S_i holds a copy of the file. We will require that a copy of the file be placed at server S_n, so that all such searches will terminate, at the latest, at S_n.

We'd like to place copies of the files at the servers so as to minimize the sum of placement and access costs. Formally, we say that a *configuration* is a choice, for each server S_i with $i = 1, 2, \ldots, n-1$, of whether to place a copy of the file at S_i or not. (Recall that a copy is always placed at S_n.) The *total cost* of a configuration is the sum of all placement costs for servers with a copy of the file, plus the sum of all access costs associated with all n servers.

Give a polynomial-time algorithm to find a configuration of minimum total cost.

18. $(*)$ Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j); and we will assume these weights satisfy the *triangle inequality* $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V'.

We are given a set $X \subseteq V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subseteq Z \subseteq V$, together with a sub-tree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T.

Show that there is function $f(\cdot)$ and a *polynomial function* $p(\cdot)$ so that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(f(k) \cdot p(n))$.

Solution. We will say that an *enriched* subset of V is one that contains at most one node not in X. There are $O(2^k n)$ enriched subsets. The overall approach will be based on *dynamic programming*: For each enriched subset Y, we will compute the following information, building it up in order of increasing $|Y|$.

- The cost $f(Y)$ of the minimum spanning tree on Y.
- The cost $g(Y)$ of the minimum Steiner tree on Y.

Consider a given Y, and suppose it has the form $X' \cup \{i\}$ where $X' \subseteq X$ and $i \notin X$. (The case in which $Y \subseteq X$ is easier.) For such a Y, one can compute $f(Y)$ in time $O(n^2)$.

Now, the minimum Steiner tree T on Y either has no extra nodes, in which case $g(Y) = f(Y)$, or else it has an extra node j of degree at least 3. Let $T_1, \ldots, T_r$ be the subtrees obtained by deleting j, with $i \in T_1$. Let p be the node in T_1 with an edge to j, let $T' = T_2 \cup \{j\}$, and let $T'' = T_3 \cdots T_r \cup \{j\}$. Let Y_1 be the nodes of Y in T_1, Y' those in T', and Y'' those in T''. Each of these is an enriched set of size less than $|Y|$, and T_1, T', and T'' are the minimum Steiner trees on these sets. Moreover, the cost of T is simply

$$g(Y_1) + g(Y') + g(Y'') + w_{jp}.$$

Thus we can compute $g(Y)$ as follows, using the values of $g(\cdot)$ already computed for smaller enriched sets. We enumerate all partitions of Y into Y_1, Y_2, Y_3 (with $i \in Y_1$), all $p \in Y_1$, and all $j \in V$, and we determine the value of

$$g(Y_1) + g(Y_2 \cup \{j\}) + g(Y_3 \cup \{j\}) + w_{jp}.$$

This can be done by looking up values we have already computed, since each of Y_1, Y', Y'' is a smaller enriched set. If any of these sums is less than $f(Y)$, we return the corresponding tree as the minimum Steiner tree; otherwise we return the minimum spanning tree on Y. This process takes time $O(3^k \cdot kn)$ for each enriched set Y.

✓ 19. Your friends have been studying the closing prices of tech stocks, looking for interesting patterns. They've defined something called a *rising trend* as follows.

They have the closing price for a given stock recorded for n days in succession; let these prices be denoted $P[1], P[2], \ldots, P[n]$. A *rising trend* in these prices is a subsequence of the prices $P[i_1], P[i_2], \ldots, P[i_k]$, for days $i_1 < i_2 < \ldots < i_k$, so that

- $i_1 = 1$, and
- $P[i_j] < P[i_{j+1}]$ for each $j = 1, 2, \ldots, k-1$.

Thus a rising trend is a subsequence of the days — beginning on the first day and not necessarily contiguous — so that the price strictly increases over the days in this subsequence.

They are interested in finding the longest rising trend in a given sequence of prices.

Example. Suppose $n = 7$, and the sequence of prices is

$$10, 1, 2, 11, 3, 4, 12.$$

Then the longest rising trend is given by the prices on days 1, 4, and 7. Note that days 2, 3, 5, and 6 consist of increasing prices; but because this subsequence does not begin on day 1, it does not fit the definition of a rising trend.

(a) Show that the following algorithm does not correctly return the *length* of the longest rising trend, by giving an instance on which it fails to return the correct answer.

```
Define $i = 1$. $L = 1$. For $j = 2$ to $n$ If $P[j] > P[i]$ then Set $i = j$. Add 1 to
$L$. Endif Endfor
```

In your example, give the actual length of the longest rising trend, and say what the above algorithm returns.

(b) Give an algorithm that takes a sequence of prices $P[1], P[2], \ldots, P[n]$ and returns the *length* of the longest rising trend.

The running time of your algorithm should be polynomial in the length of the input. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

20. Consider the Bellman-Ford minimum-cost path algorithm from the text, assuming that the graph has no negative cost cycles. This algorithm is both fairly slow and also memory-intensive. In many applications of dynamic programming, the large memory requirements can become a bigger problem than the running time. The goal of this problem is to decrease the memory requirement. The pseudo-code SHORTEST-PATH(G, s, t) in the text maintains an array $M[0...n-1; V]$ of size n^2, where $n = |V|$ is the number of nodes on the graph.

    Notice that the values of $M[i, v]$ are computed only using $M[i-1, w]$ for some nodes $w \in V$. This suggests he following idea: can we decrease the memory needs of the algorithm to $O(n)$ by maintaining only two columns of the M matrix at any time? Thus we will "collapse" the array M to an $2 \times n$ array B: as the algorithm iterates through values of i, $B[0, v]$ will hold the "previous" column's value $M[i-1, v]$, and $B[1, v]$ will hold the "current" column's value $M[i, v]$.

```
Space-Efficient-Shortest-Path$(G, s, t)$ $n =$ number of nodes in $G$ Array $B[0 \ldots 1, V]$
For $v \in V$ in any order $B[0, v] = \infty$ Endfor $B[0, s] = 0$ For $i = 1, \ldots, n-1$ For
$v \in V$ in any order $M = B[0, v]$ $M' = \min_{w \in V:(w,v) \in E} (B[0, w] + c_{wv})$ $B[1, v] =$
$\min(M, M')$ Endfor For $v \in V$ in any order $B[0, v] = B[1, v]$ Endfor Endfor
Return $B[1, t]$
```

    It is easy to verify that when this algorithm completes, the array entry $B[1, v]$ holds the value of $OPT(n-1, v)$, the minimum-cost of a path from s to v using at most $n-1$ edges, for all $v \in V$. Moreover, it uses $O(n^3)$ time and only $O(n)$ space. You do not need to prove these facts.

    The problem is: where is the shortest path? The usual way to find the path involves tracing back through the $M[i, v]$ values, using the whole matrix M, and we no longer have that. The goal of this problem is to show that if the graph has no negative cycles, then there is enough information saved in the last column of the matrix M, to recover the shortest path in $O(n^2)$ time.

Assume G has no negative or even zero length cycles. Give an algorithm FIND-PATH(t, G, B) that uses only the array B (and the graph G) to find the the minimum-cost path from s to t in $O(n^2)$ time.

21. The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm trades shares in n different companies. For each pair $i \neq j$ they maintain a trade ratio r_{ij} meaning that one share of i trades for r_{ij} shares of j. Here we allow the rate r to be fractional, i.e., $r_{ij} = \frac{2}{3}$ means that you can trade 3 shares of i to get a 2 shares of j.

    A *trading cycle* for a sequence of shares $i_1, i_2, \ldots, i_k$ consists of successively trading shares in company i_1 for shares in company i_2, then shares in company i_2 for shares i_3, and so on, finally trading shares in i_k back to shares in company i_1. After such a sequence of trades, one ends up with shares in the same company i_1 that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than what you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an *opportunity cycle*, if trading along the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles.

    Give a polynomial time algorithm that finds such an opportunity cycle, if one exists. Hint: a useful construction not covered in lecture is the augmented graph used in the statement (4.4.7).

22. ✓ As we all know, there are many sunny days in Ithaca, NY; but this year, as it happens, the spring ROTC picnic at Cornell has fallen on rainy day. The ranking officer decides to postpone the picnic, and must notify everyone by phone. Here is the mechanism she uses to do this.

    Each ROTC person on campus except the ranking officer reports to a unique *superior officer*. Thus, the reporting hierarchy can be described by a tree T, rooted at the ranking officer, in which each other node v has as a parent node u equal to his or her superior officer. Conversely, we will call v a *direct subordinate* of u. See Figure 1, in which A is the ranking officer, B and D are the direct subordinates of A, and C is the direct subordinate of B.

    To notify everyone of the postponement, the ranking officer first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates one at a time. The process continues this way, until everyone has been notified. Note that each person in this process can only call direct subordinates on the phone; for example, in Figure 1, A would not be allowed to call C.

    Now, we can picture this process as being divided into *rounds*: In one *round*, each person who has already learned of the postponement can call one of his or her direct subordinates on the phone. The number of rounds it takes for everyone to be notified depends on the sequence in which each person calls their direct subordinates. For example, in Figure 1, it will take only two rounds if A starts by calling B, but it will take three rounds if A starts by calling D.

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified, and outputs a sequence of phone calls that achieves this minimum number of rounds.

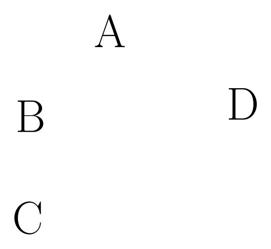


Figure 4: A hierarchy with four people. The fastest broadcast scheme is for A to call B in the first round. In the second round, A calls D and B calls C. If A were to call D first, then C could not learn the news until the third round.

Solution. The ranking officer must notify her subordinates in some sequence, after which they will recursively broadcast the message as quickly as possible to their subtrees. This is just like the homework problem on triathalon scheduling from the chapter on greedy algorithms: the subtrees must be "started" one at a time, after which they complete recursively in parallel. Using the solution to that problem, she should talk to the subordinates in decreasing order of the time it takes for their subtrees (recursively) to be notified.

Hence, we have the following set of sub-problems: for each subtree T' of T, we define $x(T')$ to be the number of rounds it takes for everyone in T' to be notified, once the root has the message. Suppose now that T' has child subtrees $T_1, \ldots, T_k$, and we label them so that $x(T_1) \geq x(T_2) \geq \cdots \geq x(T_k)$. Then by the argument in the above paragraph, we have the recurrence

$$x(T') = \min_j \left[j + x(T_j)\right].$$

If T' is simply a leaf node, then we have $x(T') = 0$.

The full algorithm builds up the values $x(T')$ using the recurrence, beginning at the leaves and moving up to the root. If subtree T' has d' edges down from its root (i.e. d' child subtrees), then the time taken to compute $x(T')$ from the solutions to smaller sub-problems is $O(d' \log d')$ — it is dominated by the sorting of the subtree values. Since a tree with n nodes has $n-1$ edges, the total time taken is $O(n \log n)$.

By tracing back through the sorted orders at every subtree, we can also reconstruct the sequence of phone calls that should be made.

23. In a word processor, the goal of "pretty-printing" is to take text with a ragged right margin — like this:

```
Call me Ishmael.
Some years ago,
never mind how long precisely,
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.
```

— and turn it into text whose right margin is as "even" as possible — like this:

```
Call me Ishmael.  Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.
```

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be "even." So suppose our text consists of a sequence of *words*, $W = \{w_1, w_2, \ldots, w_n\}$, where w_i consists of c_i characters. We have a maximum line length of L. We will assume we have a fixed-width font, and ignore issues of punctuation or hyphenation.

A *formatting* of W consists of a partition of the words in W into *lines*. In the words assigned to a single line, there should be a space after each word but the last; and so if $w_j, w_{j+1}, \ldots, w_k$ are assigned to one line, then we should have

$$\left[\sum_{i=j}^{k-1}(c_i + 1)\right] + c_k \le L.$$

We will call an assignment of words to a line *valid* if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the *slack* of the line — it's the number of spaces left at the right margin.

Give an efficient to find a partition of a set of words W into valid lines, so that the sum of the *squares* of the slacks of all lines (including the last line) is minimized.

Solution. This problem is very similar in flavor to the segmented least squares problem. We observe that the last line ends with word w_n and has to start with some word w_j; breaking off words $w_j, \ldots, w_n$ we are left with a recursive sub-problem on $w_1, \ldots, w_{j-1}$.

Thus, we define $OPT[i]$ to be the value of the optimal solution on the set of words $W_i = \{w_1, \ldots, w_i\}$. For any $i \le j$, let $S_{i,j}$ denote the slack of a line containing the words $w_i, \ldots, w_j$; as a notational device, we define $S_{i,j} = \infty$ if these words exceed total length L. For each fixed i, we can compute all $S_{i,j}$ in $O(n)$ time by considering values of j in increasing order; thus, we can compute all $S_{i,j}$ in $O(n^2)$ time.

As noted above, the optimal solution must begin the last line somewhere (at word w_j), and solve the sub-problem on the earlier lines optimally. We thus have the recurrence

$$OPT[n] = \min_{1 \leq j \leq n} S_{i,n}^2 + OPT[j-1],$$

and the line of words $w_j, \ldots, w_n$ is used in an optimum solution if and only if the minimum is obtained using index j.

Finally, we just need a loop to build up all these values:

```
Compute all values $S_{i,j}$ as described above. Set $OPT[0] = 0$ For $k = 1, \ldots, n$
$OPT[k] = \min_{1 \leq j \leq k} \left(S_{j,k}^2 + OPT[j-1]\right)$ Endfor Return $OPT[n]$.
```

As noted above, it takes $O(n^2)$ time to compute all values $S_{i,j}$. Each iteration of the loop takes time $O(n)$, and there are $O(n)$ iterations. Thus the total running time is $O(n^2)$.

By tracing back through the array OPT, we can recover the optimal sequence of line breaks that achieve the value $OPT[n]$ in $O(n)$ additional time.

24. You're consulting for a group of people, who would prefer not be mentioned here by name, whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: "untangling" a superposition of two known signals. Specifically, they're picturing a situation in which each of two ships is emitting a short sequence of 0's and 1's over and over, and they want to make sure that the signal they're hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

    This describes the whole problem; we can make it a little more explicit as follows. Given a string x consisting of 0's and 1's, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k. So $x' = 10110110110$ is a prefix of $x = 101$.

    We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of x and y, since characters 1,2,5,7,8,9 form 101101 — a repetition of x — and the remaining characters 3,4,6 form 000 — a repetition of y.

    In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we're listening to: we want to make sure it "unravels" into simple repetitions of x and y. Give an efficient algorithm that takes strings s, x, and y, and decides if s is an interleaving of x and y.

Solution. Let's suppose that s has n characters total. To make things easier to think about, let's consider the repetition x' of x consisting of exactly n characters, and the repetition y' of y consisting of exactly n characters. Our problem can be phrased as: is s an interleaving of x' and y'? The advantage of working with these elongated strings is that we don't need to "wrap around" and consider multiple periods of x' and y' — each is already as long as s.

Let $s[j]$ denote the j^{th} character of s, and let $s[1:j]$ denote the first j characters of s. We define the analogous notation for x' and y'. We know that if s is an interleaving of x' and y', then its last character comes from either x' or y'. Removing this character (wherever it is), we get a smaller recursive problem on $s[1:n-1]$ and prefixes of x' and y'.

Thus, we consider sub-problems defined by prefixes of x' and y'. Let $M[i,j] = \mathit{yes}$ if $s[1:i+j]$ is an interleaving of $x'[1:i]$ and $y'[1:j]$. If there is such an interleaving, then the final character is either $x'[i]$ or $y'[j]$, and so we have the following basic recurrence:

> $M[i,j] = \mathit{yes}$ if and only if $M[i-1,j] = \mathit{yes}$ and $s[i+j] = x'[i]$, or $M[i,j-1] = \mathit{yes}$ and $s[i+j] = y'[j]$.

We can build these up via the following loop.

> M[0,0] = yes For $k = 1, 2, \ldots, n$ For all pairs (i,j) so that $i+j=k$ If $M[i-1,j] = \mathit{yes}$ and $s[i+j] = x'[i]$ then $M[i,j] = \mathit{yes}$ Else if $M[i,j-1] = \mathit{yes}$ and $s[i+j] = y'[j]$ then $M[i,j] = \mathit{yes}$ Else $M[i,j] = \mathit{no}$ Endfor Endfor Return "yes" if and only there is some pair (i,j) with $i+j=n$ so that $M[i,j] = \mathit{yes}$.

There are $O(n^2)$ values $M[i,j]$ to build up, and each takes constant time to fill in from the results on previous sub-problems; thus the total running time is $O(n^2)$.

6 Network Flow

1. Suppose you are given a directed graph $G = (V,E)$, with a positive integer capacity c_e on each edge e, a designated source $s \in V$, and a designated sink $t \in V$. You are also given a maximum s-t flow in G, defined by a flow value f_e on each edge e. The flow $\{f_e\}$ is *acyclic*: there is no cycle in G on which all edges carry positive flow.

   Now, suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m)$, where m is the number of edges in G.

2. Consider the following problem. You are given a flow network with unit-capacity edges: it consists of a directed graph $G = (V,E)$, a source $s \in V$, and a sink $t \in V$; and $c_e = 1$ for every $e \in E$. You are also given a parameter k.

The goal is delete k edges so as to reduce the maxmimum s-t flow in G by as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum s-t flow in $G' = (V, E - F)$ is as small as possible subject to this.

Give a polynomial-time algorithm to solve this problem.

Solution. If the minimum s-t cut has size $\leq k$, then we can reduce the flow to 0. Otherwise, let $f > k$ be the value of the maximum s-t flow. We identify a minimum s-t cut (A, B), and delete k of the edges out of A. The resulting subgraph has a maximum flow value of at most $f - k$.

But we claim that for any set of edges F of size k, the subgraph $G' = (V, E - F)$ has an s-t flow of value at least $f - k$. Indeed, consider any cut (A, B) of G'. There are at least f edges out of A in G, and at most k have been deleted, so there are at least $f - k$ edges out of A in G'. Thus, the minimum cut in G' has value at least $f - k$, and so there is a flow of at least this value.

3. Suppose you're looking at a flow network G with source s and sink t, and you want to be able to express something like the following intuitive notion: some nodes are clearly on the "source side" of the main bottlenecks; some nodes are clearly on the "sink side" of the main bottlenecks; and some nodes are in the middle. However, G can have many minimum cuts, so we have to be careful in how we try making this idea precise.

   Here's one way to divide the nodes of G into three categories of this sort.

   - We say a node v is *upstream* if for all minimum s-t cuts (A, B), we have $v \in A$ — that is, v lies on the source side of every minimum cut.
   - We say a node v is *downstream* if for all minimum s-t cuts (A, B), we have $v \in B$ — that is, v lies on the sink side of every minimum cut.
   - We say a node v is *central* if it is neither upstream nor downstream; there is at least one minimum s-t cut (A, B) for which $v \in A$, and at least one minimum s-t cut (A', B') for which $v \in B'$.

   Give an algorithm that takes a flow network G, and classifies each of its nodes as being upstream, downstream, or central. The running time of your algorithm should be within in a constant factor of the time required to compute a *single* maximum flow.

   Solution. Consider the cut (A^*, B^*) found by performing breadth-first search on the residual graph G_f at the end of any maximum flow algorithm. We claim that a node v is upstream if and only if $v \in A^*$. Clearly, if v is upstream, then it must belong to A^*; since otherwise, it lies on the sink-side of the minimum cut (A^*, B^*). Conversely, suppose $v \in A^*$ were not upstream. Then there would be a minimum cut (A', B') with $v \in B'$. Now, since $v \in A^*$, there is a path P in G_f from s to v. Since $v \in B'$, this path must have an edge (u, w) with $u \in A'$ and $w' \in B'$. But this is a contradiction, since no edge in the residual graph can go from the source side to the sink side of any minimum cut.

A completely symmetric argument shows the following. Let B_* denote the nodes that can reach t in G_f, and let $A_* = V - B_*$. Then (A_*, B_*) is a minimum cut, and a node w is downstream if and only if $w \in B_*$.

Thus, our algorithm is to compute a maximum flow f, build G_f, and use breadth-first search to find the sets A^* and B_*. These are the upstream and downstream nodes respectively; the remaining nodes are central.

4. Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and non-negative edge capacities $\{c_e\}$. Give a polynomial time algorithm to decide whether G has a *unique* minimum s-t cut. (I.e. an s-t of capacity strictly less than that of all other s-t cuts.)

5. In a standard minimum s-t cut problem, we assume that all capacities are non-negative; allowing an arbitrary set of positive and negative capacities results in an NP-complete problem. (You don't have to prove this.) However, as we'll see here, it is possible to relax the non-negativity requirement a little, and still have a problem that can be solved in polynomial time.

   Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and edge capacities $\{c_e\}$. Suppose that for every edge e that has neither s nor t as an endpoint, we have $c_e \geq 0$. Thus, c_e can be negative for edges e that have at least one end equal to either s or t. Give a polynomial-time algorithm to find an s-t cut of minimum value in such a graph. (Despite the new non-negativity requirements, we still define the value of an s-t cut (A, B) to be the sum of the capacities of all edges e for which the tail of e is in A and the head of e is in B.)

6. Let M be an $n \times n$ matrix with each entry equal to either 0 or 1. Let m_{ij} denote the entry in row i and column j. A *diagonal entry* is one of the form m_{ii} for some i.

   Swapping rows i and j of the matrix M denotes the following action: we swap the values m_{ik} and m_{jk} for $k = 1, 2, \ldots, n$. Swapping two columns is defined analogously.

   We say that M is *re-arrangeable* if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that after all the swapping, all the diagonal entries of M are equal to 1.

   (a) Give an example of a matrix M which is not re-arrangeable, but for which at least one entry in each row and each column is equal to 1.

   (b) Give a polynomial-time algorithm that determines whether a matrix M with 0-1 entries, is re-arrangeable.

7. You're helping to organize a class on campus that has decided to give all its students wireless laptops for the semester. Thus, there is a collection of n wireless laptops; there is also have a collection of n wireless *access points*, to which a laptop can connect when it is in range.

The laptops are currently scattered across campus; laptop ℓ is within range of a *set* S_ℓ of access points. We will assume that each laptop is within range of at least one access point (so the sets S_ℓ are non-empty); we will also assume that every access point p has at least one laptop within range of it.

To make sure that all the wireless connectivity software is working correctly, you need to try having laptops make contact with access points, in such a way that each laptop and each access point is involved in at least one connection. Thus, we will say that a *test set* T is a collection of ordered pairs of the form (ℓ, p), for a laptop ℓ and access point p, with the properties that

(i) If $(\ell, p) \in T$, then ℓ is within range of p. (I.e. $p \in S_\ell$).

(ii) Each laptop appears in at least one ordered pair in T.

(iii) Each access point appears in at least one ordered pair in T.

This way, by trying out all the connections specified by the pairs in T, we can be sure that each laptop and each access point have correctly functioning software.

The problem is: Given the sets S_ℓ for each laptop (i.e. which laptops are within range of which access points), and a number k, decide whether there is a test set of size at most k.

Example: Suppose that $n = 3$; laptop 1 is within range of access points 1 and 2; laptop 2 is within range of access point 2; and laptop 3 is within range of access points 2 and 3. Then the set of pairs

(laptop 1, access point 1), (laptop 2, access point 2),
(laptop 3, access point 3)

would form a test set of size three.

(a) Give an example of an instance of this problem for which there is no test set of size n. (Recall that we assume each laptop is within range of at least one access point, and each access point p has at least one laptop within range of it.)

(b) Give a polynomial-time algorithm that takes the input to an instance of this problem (including the parameter k), and decides whether there is a test set of size at most k.

8. Back in the euphoric early days of the Web, people liked to claim that much of the enormous potential in a company like *Yahoo!* was in the "eyeballs" — the simple fact that it gets millions of people looking at its pages every day. And further, by convincing people to register personal data with the site, it can show each user an extremely targeted advertisement whenever he or she visits the site, in a way that

TV networks or magazines couldn't hope to match. So if the user has told *Yahoo!* that they're a 20-year old computer science major from Cornell University, the site can throw up a banner ad for apartments in Ithaca, NY; on the other hand, if they're a 50-year-old investment banker from Greenwich, Connecticut, the site can display a banner ad pitching Lincoln Town Cars instead.

But deciding on which ads to show to which people involves some serious computation behind the scenes. Suppose that the managers of a popular Web site have identified k distinct *demographic groups* $G_1, G_2, \dots, G_k$. (These groups can overlap; for example G_1 can be equal to all residents of New York State, and G_2 can be equal to all people with a degree in computer science.) The site has contracts with m different *advertisers*, to show a certain number of copies of their ads to users of the site. Here's what the contract with the i^{th} advertiser looks like:

- For a subset $X_i \subseteq \{G_1, \dots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i.
- For a number r_i, advertiser i wants its ads shown to at least r_i users each minute.

Now, consider the problem of designing a good *advertising policy* — a way to show a single ad to each user of the site. Suppose at a given minute, there are n users visiting the site. Because we have registration information on each of these users, we know that user j (for $j = 1, 2, \dots, n$) belongs to a subset $U_j \subseteq \{G_1, \dots, G_k\}$ of the demographic groups. The problem is: is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute? (That is, for each $i = 1, 2, \dots, m$, at least r_i of the n users, each belonging to at least one demographic group in X_i, are shown an ad provided by advertiser i.)

Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

Solution. We define a flow network $G = (V, E)$ as follows.

- There is a source s, vertices $v_1, \dots, v_n$ for each person vertices $w_1, \dots, w_m$ for each advertiser, and sink t.
- There is an edge of capacity 1 from v_i to each w_j for which person i belongs to a demographic group that advertiser j wants to target.
- There is an edge with a capacity of 1 from s to each v_i; and for each j, there is an edge with lower bound r_j from w_j to t.
- Finally, the source has a demand of $-\sum_j r_j$, and the sink has a demand of $\sum_j r_j$. All other nodes have demand 0.

Now, if there is a valid circulation in this graph, then there is an integer circulation. In such a circulation, one unit of flow on the edge (v_i, w_j) means that we show an ad from advertiser j to person i. With this meaning, each advertiser shows their required number of ads to the appropriate people.

Conversely, if there is a way to satisfy all the advertising contracts, then we can construct a valid circulation as follows. We place a unit of flow on each edge (v_i, w_j) for which i is shown an ad from advertiser j; we put a flow on the edge (w_j, t) equal to the number of ads shown from advertiser j; and we put a unit of flow on each edge (s, v_i) for which person i sees an ad.

Thus, there is a valid circulation in this graph if and only if there is a way to satisfy all the advertising contracts; and the flow values in an integer-valued circulation can be used, as above, to decide which ads to show to which people.

9. Some of your friends have recently graduated and started a small company called WebExodus, which they are currently running out of their parents' garages in Santa Clara. They're in the process of porting all their software from an old system to a new, revved-up system; and they're facing the following problem.

   They have a collection of n software applications, $\{1, 2, \ldots, n\}$, running on their old system; and they'd like to port some of these to the new system. If they move application i to the new system, they expect a net (monetary) benefit of $b_i \geq 0$. The different software applications interact with one another; if applications i and j have extensive interaction, then the company will incur an expense if they move one of i or j to the new system but not both — let's denote this expense by $x_{ij} \geq 0$.

   So if the situation were really this simple, your friends would just port all n applications, achieving a total benefit of $\sum_i b_i$. Unfortunately, there's a problem ...

   Due to small but fundamental incompatibilities between the two systems, there's no way to port application 1 to the new system; it will have to remain on the old system. Nevertheless, it might still pay off to port some of the other applications, accruing the associated benefit and incurring the expense of the interaction between applications on different systems.

   So this is the question they pose to you: which of the remaining applications, if any, should be moved? Give a polynomial-time algorithm to find a set $S \subseteq \{2, 3, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

   Solution. We define a directed graph $G = (V, E)$ with nodes $s, v_1, v_2, \ldots, v_n$, where our sink t will correspond to v_1. Define an edge (s, v_i) of capacity b_i, and edges (v_i, v_j) and (v_j, v_i) of capacity p_{ij} for each $p_{ij} > 0$. Define $B = \sum_i b_i$. Now, let (X, Y) be the minimum s-v_1 cut in G. The capacity of (X, Y) is

$$\begin{aligned} c(X,Y) &= \sum_{i \in Y} b_i + \sum_{i \in X, j \in Y} p_{ij} \\ &= B - \sum_{i \in X} b_i + \sum_{i \in X, j \in Y} p_{ij} \end{aligned}$$

Thus, finding a cut of minimum capacity is the same as finding a set X maximizing $\sum_{i \in X} b_i - \sum_{i \in X, j \notin X} p_{ij}$.

10. Consider a variation on the previous problem. In the new scenario, any application can potentially be moved, but now some of the benefits b_i for moving to the new system are in fact *negative*: if $b_i < 0$, then it is preferable (by an amount quantifed in b_i) to keep i on the old system. Again, give a polynomial-time algorithm to find a set $S \subseteq \{1, 2, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

    Solution. Choose a number r larger than any $|b_i|$, and say that the *scaled benefit* b_{i0} of keeping application i on the old system is $b_{i0} = r$, while the scaled benefit of moving it is $b_{i1} = r + b_i$. So our problem is equivalent to that of finding a partition of the applications that maximizes the sum of the scaled benefits minus the sum of p_{ij} for all pairs i and j that are split.

    Define a directed graph $G = (V, E)$ with nodes $s, t, v_1, v_2, \ldots, v_n$, edges $(s, v_i), (v_i, s)$ of capacity b_{i0}, $(t, v_i), (v_i, t)$ of capacity b_{i1}, and $(v_i, v_j), (v_j, v_i)$ of capacity p_{ij}. Let $B = \sum_i (b_{i0} + b_{i1})$. Now, the capacity of an s-t cut (X, Y) can be written as

$$\begin{aligned} c(X,Y) &= \sum_{i \notin X} b_{i0} + \sum_{i \notin Y} b_{i1} + \sum_{i \in X, j \in Y} p_{ij} \\ &= B - \sum_{i \in X} b_{i0} - \sum_{i \in Y} b_{i1} + \sum_{i \in X, j \in Y} p_{ij} \end{aligned}$$

    Thus, finding a cut of minimum capacity is the same as maximizing the objective function in the previous paragraph.

11. Consider the following definition. We are given a set of n countries that are engaged in trade with one another. For each country i, we have the value s_i of its budget surplus; this number may be positive or negative, with a negative number indicating a deficit. For each pair of countries i, j, we have the total value e_{ij} of all exports from i to j; this number is always non-negative. We say that a subset S of the countries is *free-standing* if the sum of the budget surpluses of the countries in S, minus the total value of all exports from countries in S to countries not in S, is non-negative.

    Give a polynomial-time algorithm that takes this data for a set of n countries, and decides whether it contains a non-empty free-standing subset that is not equal to the full set.

    Solution. We define the following flow network. There will be a node v_i for each country i, plus a source s and sink t. For each pair of nodes (v_i, v_j) with $e_{ij} > 0$, we include an edge of capacity e_{ij}. For each node v_i with $s_i > 0$, we include an edge (s, v_i) of capacity s_i and for each node v_i with $s_i < 0$, we include an edge (v_i, t) of capacity $-s_i$. Let

$$N = \sum_{i : s_i > 0} s_i.$$

Now, consider an s-t cut (A, B), and write $S = A - \{s\}$, $T = B - \{t\}$. This cut's capacity is

$$\begin{aligned} c(A,B) &= \sum_{i \in T, s_i > 0} s_i + \sum_{i \in S, s_i < 0} -s_i + \sum_{i \in S, j \in T} e_{ij} \\ &= N - \sum_{i \in S, s_i > 0} s_i - \sum_{i \in S, s_i < 0} s_i + \sum_{i \in S, j \in T} e_{ij} \\ &= N - \left(\sum_{i \in S} s_i - \sum_{i \in S, j \in T} e_{ij} \right). \end{aligned}$$

The last expression inside parentheses is precisely what is specified in the definition of *free-standing.*

Thus, there is a non-empty free-standing set if and only if there is a minimum s-t cut (A, B) in which the capacity is at most N and in which A contains at least one node other than s. (This latter part is crucial since otherwise we end up with an empty free-standing set S.) Hence, we need to be able to check whether $(\{s\}, V - \{s\})$ is the unique minimum cut. One can verify that this holds if and only if all nodes other than s have a path to t in the residual graph at the end of the Ford-Fulkerson algorithm.

12. $(*)$ In sociology, one often studies a graph G in which nodes represent people, and edges represent those who are friends with each other. Let's assume for purposes of this question that friendship is symmetric, so we can consider an undirected graph.

Now, suppose we want to study this graph G, looking for a "close-knit" group of people. One way to formalize this notion would be as follows. For a subset S of nodes let $e(S)$ denote the number of edges in S, i.e., the number of edges that have both ends in S. We define the *cohesiveness* of S as $e(S)/|S|$. A natural thing to search for would be the set S of people achieving the maximum cohesiveness.

(a.) Give a polynomial time algorithm that takes a rational number α and determines whether there exists a set S with cohesiveness at least α.

(b.) Give a polynomial time algorithm to find a set S of nodes with maximum cohesiveness

Solution. First, the maximum edge density of a subset has a numerator bounded by $\binom{n}{2}$ and a denominator bounded by n; thus, it can assume one of $O(n^3)$ possible values. Thus, if we can decide for a given rational Δ whether there is a subset of edge density at least Δ, we can find the maximum density using binary search over these possible values, incurring a multiplicative increase of $O(\log n)$ in the running time.

Now, for a node i in G, let $d(i)$ denote its degree. Also, for a set $R \subseteq V$, let $e(R)$ denote the number of edges induced among nodes of R, and let $d(R)$ denote the number of edges with exactly one end in R.

Consider the definition of a free-standing set from a previous question. We define an instance of the free-standing set problem in which $s_i = d(i) - 2\Delta$ and $e_{ij} = e_{ji} = 1$ for $(i, j) \in E$, $e_{ij} = e_{ji} = 0$ otherwise. Write

$$f(R) = \sum_{i \in R} s_i - \sum_{i \in R, j \notin R} e_{ij}.$$

We claim that R has edge density at least Δ if and only if it is a free-standing subset with respect to $\{s_i\}, \{e_{ij}\}$; that is, if and only if $f(R) \geq 0$. Indeed, observe that for any $R \subset V$, the quantity $\sum_{i \in R} d(i)$ is equal to $2e(R) + d(R)$. Thus,

$$f(R) = 2e(R) + d(R) - 2|R|\Delta - d(R) = 2e(R) - 2|R|\Delta,$$

and hence

$$\frac{e(R)}{|R|} - \Delta = \frac{f(R)}{2|R|}.$$

13. Suppose we are given a directed network $G = (V, E)$ with a root node r, and a set of *terminals* $T \subseteq V$. We'd like to disconnect many terminals from r, while cutting relatively few edges.

We make this trade-off precise as follows. For a set of edges $F \subseteq E$, let $q(F)$ denote the number of nodes $v \in T$ such that there is no r-v path in the subgraph $(V, E - F)$. Give a polynomial-time algorithm to find a set F of edges that maximizes the quantity $q(F) - |F|$. (Note that setting F equal to the empty set is an option.)

Solution. We first observe the following fact about flow networks: there is a minimum s-t cut (A, B) such that there is a path from s to each node in A, passing only through nodes in A. (We'll call such a cut a *compact cut.*) Indeed, consider any s-t cut (A, B), and let A' be the set of nodes $v \in A$ such that there is an s-v path using only nodes in A. We claim that the cut $(A', B \cup (A - A')$ has no greater capacity, since the only edges out of A' go to nodes of B; none go to $(A - A')$.

Now we consider the problem at hand. We can assume that in G, there is a path from r to each node in V; otherwise, we can initially re-define V to simply be the subset of nodes that are reachable from r. We can also assume $r \notin T$.

We now construct a flow network G' by adding to G a sink node t, and a node (v, t) for each $v \in T$. All edges are given capacity 1. Let c^* be the value of the minimum cut in G. We know from the argument above that there is in fact a compact cut (A, B) of capacity c^*; let F^* be the set of edges out of A that do not go directly to t. Then every terminal in A can still be reached from r after the deletion of F^*, since (A, B) is a compact cut, so $q(F^*) = |B \cap T|$. The capacity $c(A, B)$ is equal to $c^* = |F^*| + |A \cap T| = |T| - (q(F^*) - |F^*|)$. So, in particular, there is a set of edges F^* for which $q(F^*) - |F^*| = |T| - c^*$.

Now, consider a set of edges F' that achieves the optimal value of $q(F') - |F'|$. Let A denote the set of nodes that can be reached from r after the deletion of F', and let

$B = (V - A) \cup \{t\}$. We know that each edge $e \in F'$ must have its tail in A and its head in $V - A$, for otherwise we could delete e from F' and obtain a better set of edges. Thus we have

$$c^* \leq c(A, B) = |A \cap T| + |F'| = |T| - (q(F') - |F'|),$$

and so $q(F') - |F'| \leq |T| - c^*$. It follows that the set of edges F^* defined in the previous paragraph is optimal.

14. Some of your friends with jobs out West decide they really need some extra time each day to sit in front of their laptops, and the morning commute from Woodside to Palo Alto seems like the only option. So they decide to carpool to work.

 Unfortunately, they all hate to drive, so they want to make sure that any carpool arrangement they agree upon is fair, and doesn't overload any individual with too much driving. Some sort of simple round-robin scheme is out, because none of them goes to work every day, and so the subset of them in the car varies from day to day.

 Here's one way to define *fairness*. Let the people be labeled $S = \{p_1, \ldots, p_k\}$. We say that the *total driving obligation* of p_j over a set of days is the expected number of times that p_j would have driven, had a driver been chosen uniformly at random from among the people going to work each day. More concretely, suppose the carpool plan lasts for d days, and on the i^{th} day a subset $S_i \subseteq S$ of the people go to work. Then the above definition of the total driving obligation Δ_j for p_j can be written as $\Delta_j = \sum_{i: p_j \in S_i} \frac{1}{|S_i|}$. Ideally, we'd like to require that p_j drives at most Δ_j times; unfortunately, Δ_j may not be an integer.

 So let's say that a *driving schedule* is a choice of a driver for each day — i.e. a sequence $p_{i_1}, p_{i_2}, \ldots, p_{i_d}$ with $p_{i_t} \in S_t$ — and that a *fair driving schedule* is one in which each p_j is chosen as the driver on at most $\lceil \Delta_j \rceil$ days.

 (a) Prove that for any sequence of sets $S_1, \ldots, S_d$, there exists a fair driving schedule.

 (b) Give an algorithm to compute a fair driving schedule with running time polynomial in k and d.

 (c)$(*)$ One could expect k to be a much smaller parameter than d (e.g. perhaps $k = 5$ and $d = 365$). So it could be worth reducing the dependence of the running time on d even at the expense of a much worse dependence on k. Give an algorithm to compute a fair driving schedule whose running time has the form $O(f(k) \cdot d)$, where $f(\cdot)$ can be an arbitrary function.

 Solution. (a, b) We define a graph $G = (V, E)$ with source s, vertices $v_1, \ldots, v_d$ for each day, vertices $w_1, \ldots, w_k$ for each person, and sink t. There is an edge of capacity 1 from s to each v_i, an edge of capacity 1 from v_i to each w_j with $p_j \in S_i$, and an edge of capacity $\lceil \Delta_j \rceil$ from w_j to t. We know there is a feasible fractional flow in this graph of value d, obtained by assigning a flow of value of $\frac{1}{|S_i|}$ to each edge (v_i, w_j), and flow values to all other edges as implied by the conservation condition. Thus there is a feasible integer flow, and the flow values on the edges of the form (v_i, w_j) define a fair driving schedule in the following way: p_j drives on day i if and only if $f(v_i, w_j) = 1$. This also gives a polynomial-time algorithm to compute the schedule.

(c) In place of $\{v_1, \ldots, v_d\}$, we could make a node $u_{S'}$ for every subset $S' \subseteq S$. There is an edge $(s, u_{S'})$ of capacity equal to the number of days $d_{S'}$ that S' was the set of people going to work. There is an edge of capacity $d_{S'}$ from to w_j for each $p_j \in u_{S'}$, and an edge of capacity $\lceil \Delta_j \rceil$ from w_j to t. This graph G has a size that is dependent only on k (the linear dependence on d comes simply from constructing it); thus we can compute a maximum flow in G in time depending only on k. Again, there is a flow of value d in G — assign flow value $\frac{d_{S'}}{|S'|}$ to each edge $(u_{S'}, w_j)$, and other flow values to ensure conservation — so there is an integer flow of value d. We can use the integer flow values on edges of the form $(d_{S'}, w_j)$ to define a fair driving schedule: of the days on which the set S' goes to work, p_j should be the driver on $f(d_{S'}, w_j)$ of them.

15. Suppose you live in a big apartment with a bunch of friends. Over the course of a year, there are a lot of occasions when one of you pays for an expense shared by some subset of the apartment, with the expectation that everything will get balanced out fairly at the end of the year. For example, one of you may pay the whole phone bill in a given month, another will occasionally make communal grocery runs to the nearby organic food emporium; and a third might sometimes use a credit card to cover the whole bill at the local Italian-Indian restaurant, *Little Idli.*

    In any case, it's now the end of the year, and time to settle up. There are n people in the apartment; and for each ordered pair (i, j) there's an amount $a_{ij} \geq 0$ that i owes j, accumulated over the course of the year. We will require that for any two people i and j, at least one of the quantities a_{ij} or a_{ji} is equal to 0. This can be easily made to happen as follows: if it turns out that i owes j a positive amount x, and j owes i a positive amount $y < x$, then we will subtract off y from both sides and declare $a_{ij} = x - y$ while $a_{ji} = 0$. In terms of all these quantities, we now define the *imbalance* of a person i to be the sum of the amounts that i is owed by everyone else, minus the sum of the amounts that i owes everyone else. (Note that an imbalance can be positive, negative, or zero.)

    In order to restore all imbalances to 0, so that everyone departs on good terms, certain people will write checks to others; in other words, for certain ordered pairs (i, j), i will write a check to j for an amount $b_{ij} > 0$. We will say that a set of checks constitutes a *reconciliation* if for each person i, the total value of the checks received by i, minus the total value of the checks written by i, is equal to the imbalance of i. Finally, you and your friends feel it is bad form for i to write j a check if i did not actually owe j money, so we say that a reconciliation is *consistent* if, whenever i writes a check to j, it is the case that $a_{ij} > 0$.

    Show that for any set of amounts a_{ij} there is always a consistent reconciliation in which at most $n - 1$ checks get written, by giving a polynomial-time algorithm to compute such a reconciliation.

Solution. Construct a graph $G = (V, E)$ where the nodes represent people that are either owed or owe money. Let there be an edge (u, v) if person u owes person v money and let the cost of this edge be the amount of money owed. Note that if (u, v) exists, then (v, u) does not exist since we may just adjust for the difference.

Let $\overline{G}$ be the undirected graph obtained from G by ignoring the directions of the edges. We repeatedly run BFS on $\overline{G}$ to find undirected cycles and eliminate them as specified below. We do this as follows:

> Search for an undirected CYCLE While CYCLE != NULL (a cycle exists) Find the edge in CYCLE with minimum cost. Let this minimum cost be stored in MINCOST and the correspoding edge in MINEDGE. For each edge EDGE in CYCLE If EDGE has the same direction as MINEDGE reset EDGE's cost to its current cost less MINCOST Else reset EDGE's cost to its current cost plus MINCOST Endif Endfor Remove all edges whose cost has been reduced to 0 (including MINEDGE) Search for a new undirected CYCLE Endwhile

Note that each time through the while loop, we get rid of a cycle. Since there are m edges, we go through the outer loop at most m times. Also finding a cycle via BFS takes $O(m + n)$ time. Thus the overall running time is $O(m(m + n))$.

Note that in each iteration, we preserve all imbalances; so at the end we will have a reconciliation. Further, we preserve the direction of the edges since we modify according to the direction and only by the minimum cost edge in the cycle; thus, at the end, we will have a consistent reconciliation. Since $\overline{G}$ has no cycles at termination, it must be a tree or a forest, and therefore has at most $n - 1$ edges. Thus we have produced consistent reconciliation in which at most $n - 1$ checks get written.

Grader's Comments: It was necessary to search for undirected cycles in G, rather than directed cycles, since eliminating only the directed cycles in G will not necessarily reduce the number of edges to $n - 1$.

There were three key points in the proof. First, imbalances are preserved as each cycle is processed. Second, directions of edges are never reversed, so the resulting reconciliation is consistent. Third, since all undirected cycles are eliminated, there are at most n-1 edges and at most $n - 1$ checks written.

Some solutions simply ran a flow algorithm with a super-source attached to the people with positive imbalances, super-sink attached to the people with negative imbalances, and edges of infinite capacity joining pairs who owed each other initially; after this, they claimed that the resulting flow would be positive on at most $n - 1$ edges. It seems hard to find a rule for choosing augmenting paths that will guarantee this, and also hard to prove that this property holds. (Of course, if one explicitly cancels cycles after finding the flow, then this would be correct by analogy with the above solution.)

16. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible *base stations*. We'll suppose there are n clients, with the position of each client specified its by (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well.

    For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a *range parameter* r — a client can only be connected to a base station that is within distance r. There is also a *load parameter* L — no more than L clients can be connected to any single base station.

    Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

17. You can tell that cellular phones are at work in rural communities, from the giant microwave towers you sometimes see sprouting out of corn fields and cow pastures. Let's consider a very simplified model of a cellular phone network in a sparsely populated area.

    We are given the locations of n *base stations*, specified as points $b_1, \ldots, b_n$ in the plane. We are also given the locations of n cellular phones, specified as points $p_1, \ldots, p_n$ in the plane. Finally, we are given a *range parameter* $\Delta > 0$. We call the set of cell phones *fully connected* if it is possible to assign each phone to a base station in such a way that

    - Each phone is assigned to a different base station, and
    - If a phone at p_i is assigned to a base station at b_j, then the straight-line distance between the points p_i and b_j is at most Δ.

    Suppose that the owner of the cell phone at point p_1 decides to go for a drive, traveling continuously for a total of z units of distance due east. As this cell phone moves, we may have to update the assignment of phones to base stations (possibly several times) in order to keep the set of phones *fully connected*.

    Give a polynomial-time algorithm to decide whether it is possible to keep the set of phones fully connected at all times during the travel of this one cell phone. (You should assume that all other phones remain stationary during this travel.) If it is possible, you should report a sequence of assignments of phones to base stations that will be sufficient in order to maintain full connectivity; if it is not possible, you should report a point on the traveling phone's path at which full connectivity cannot be maintained.

    You should try to make your algorithm run in $O(n^3)$ time if possible.

    Example: Suppose we have phones at $p_1 = (0, 0)$ and $p_2 = (2, 1)$; we have base stations at $b_1 = (1, 1)$ and $b_2 = (3, 1)$; and $\Delta = 2$. Now consider the case in which the phone at p_1 moves due east a distance of 4 units, ending at $(4, 0)$. Then it is possible to keep the phones fully connected during this motion: We begin by assigning p_1 to b_1 and p_2 to b_2, and we re-assign p_1 to b_2 and p_2 to b_1 during the motion. (For example, when p_1 passes the point $(2, 0)$.)

Solution. We first decide whether the phones can be fully connected in the starting position of p_1. To do this, we construct a bipartite graph $G = (X \cup Y, E)$, where X is the set of phones, Y is the set of base stations, and there is an edge (p_i, b_j) if and only if the distance from p_i to b_j is at most Δ. By the definition of G, a perfect matching corresponds to a legal assignment of phones to base stations; thus, the phones can be fully connected if and only if G has a perfect matching, and this can be checked in $O(n^3)$ time.

Define an *event* to be a position t on the path of phone p_1 at which it first comes into range of a base station, or first goes out of range of a base station. The path of p_1 is a line, and the set of points in range of a base station b_j is a circle of radius Δ; since the line can intersect the circle at most twice, there is at most one event in which p_1 comes into range of b_j and at most one event in which it goes out of range of b_j. Thus, there are at most $2n$ events total; we can sort them by x-coordinate in the order $t_1, t_2, \ldots, t_k$, with $k \leq 2n$.

In the interval between consecutive events, the bipartite graph G does not change. At an event t_i, the bipartite graph changes because some edge incident to p_1 is either added or deleted. Let $G[t_i]$ denote the bipartite graph just after event t_i. Since these are the only points at which the graph changes, the phones can be fully connected if and only if each of $G, G[t_1], G[t_2], \ldots, G[t_k]$ has a perfect matching.

Thus, an $O(n^4)$ algorithm would be to test each of these graphs for a perfect matching. To bring the running time down to $O(n^3)$, we make the following observation. A perfect matching M in $G[t_i]$ becomes a matching of size either n or $n-1$ in $G[t_{i+1}]$, depending on whether one of the edges in M is the edge that is deleted during the event t_i. In the first case, M is already a perfect matching in $G[t_{i+1}]$, so we don't have to do any work at all for this event. In the second case, we simply to need to search for *a single augmenting path* in order to decide whether the existing matching of size $n - 1$ can be increased to a perfect matching. This takes time $O(|E|) = O(n^2)$. Moreover, the sequence of assignments can be recorded by keeping track of the sequence of matching constructed over all the events.

Thus, the total running time of the improved algorithm is $O(n^3)$ (for the initial perfect matching in G) plus $O(kn^2)$ (for the (at most) one augmenting path at each event). Since $k \leq 2n$, the total running time is $O(n^3 + kn^2) = O(n^3)$.

18. Suppose you're managing a collection of processors and must schedule a sequence of jobs over time.

The jobs have the following characteristics. Each job j has an arrival time a_j when it is first available for processing, a length ℓ_j which indicates how much processing time it needs, and a deadline d_j by which it must be finished. (We'll assume $0 < \ell_j \leq d_j - a_j$.) Each job can be run on any of the processors, but only on one at a time; it can also be pre-empted and resumed from where it left off (possibly after a delay) on another processor.

Moreover, the collection of processors is not entirely static either: you have an overall pool of k possible processors; but for each processor i, there is an interval of time $[t_i, t'_i]$ during which it is available; it is unavailable at all other times.

Given all this data about job requirements and processor availability, you'd like to decide whether the jobs can all be completed or not. Give a polynomial-time algorithm that either produces a schedule completing all jobs by their deeadlines, or reports (correctly) that no such schedule exists. You may assume that all the parameters associated with the problem are integers.

Example. Suppose we have two jobs J_1 and J_2. J_1 arrives at time 0, is due at time 4, and has length 3. J_2 arrives at time 1, is due at time 3, and has length 2. We also have two processors P_1 and P_2. P_1 is available between times 0 and 4; P_2 is available between times 2 and 3. In this case, there is a schedule that gets both jobs done:

- At time 0, we start job J_1 on processor P_1.
- At time 1, we pre-empt J_1 to start J_2 on P_1.
- At time 2, we resume J_1 on P_2. (J_2 continues processing on P_1.)
- At time 3, J_2 completes by its deadline. P_2 ceases to be available, so we move J_1 back to P_1 to finish its remaining one unit of processing there.
- At time 4, J_1 completes its processing on P_1.

Notice that there is no solution that does not involve pre-emption and moving of jobs.

Solution. Let a^* be the earliest arrival time of any job, and d^* the latest deadline of any job. We break up the interval $I = [a^*, d^*]$ at each value of any a_j, d_j, t_i, or t'_i. Let the resulting sub-intervals of I be denoted $I_1, I_2, \ldots, I_r$, with $I_i = [s_i, s'_i]$. Note that $s'_i = s_{i+1}$ in our notation; we let $q_i = s'_i - s_i$ denote the length of interval I_i in time steps. Observe that the set of processors available is constant throughout each interval I_i; let n_i denote the size of this set. Also, the set of jobs that have been released but are not yet due is constant throughout each interval I_i.

We now construct a flow network G that tells us, for each job j and each interval I_i, how much time should be spent processing job j during interval I_i. ¿From this, we can construct the schedule. We define a node u_j for each job j, and a node v_i for each interval I_i. If $I_i \subseteq [a_j, d_j]$, then we add an edge (u_j, v_i) of capacity q_i. We define an edge from the source s to each u_j with capacity ℓ_j, and define an edge from each v_i to the sink t with capacity $n_i q_i$.

Now, suppose there is a schedule that allows each job to complete by its deadline, and suppose it processes job j for z_{ji} units of time in the interval I_i. Then we define a flow with value ℓ_j on the edge (s, u_j), value z_{ji} on the edge (u_j, v_i), and sufficient flow on the edge (v_i, t) to satisfy conservation. Note that the capacity of (v_i, t) cannot be exceeded, since we have a valid schedule.

Conversely, given an integer flow of value $\sum_j \ell_j$ in G, we run job j for $z_{ji} = f(u_j, v_i)$ units of time during interval I_i. Since the flow has value $\sum_j \ell_j$, it clearly saturates each edge (s, u_j), and so u_j will be processed for ℓ_j units of time, as required, if we can guarantee that all jobs j can really be scheduled for z_{ji} units of time during interval I_i. The issue here is the following: we are told that job j must receive z_{ji} units of processing time during I_i, and it can move from one processor to another during the interval, but we need to avoid having two processors working on the same job at the same point in time. Here is a way to assign jobs to processors that avoids this. Let $P_1, P_2, \ldots, P_{n_i}$ denote the processors, and let $y_j = \sum_{r<j} z_{ri}$. For each $k = y_j + 1, y_j + 2, \ldots, y_{j+1}$, we have processor $\lceil k/q_i \rceil$ spend the $(k - q_i \lfloor k/q_i \rfloor)^{\text{th}}$ step of interval I_i working on job j. Since $\sum_j z_{ji} \leq n_i q_i$, each job gets a sufficient number of steps allocated to it; and since $z_{ji} \leq q_i$ for each j, this allocation scheme does not involve two processors working on the same job at the same point in time.

19. In a lot of numerical computations, we can ask about the "stability" or "robustness" of the answer. This kind of question can be asked for combinatorial problems as well; here's one way of phrasing the question for the minimum spanning tree problem.

Suppose you are given a graph $G = (V, E)$, with a cost c_e on each edge e. We view the costs as quantities that have been measured experimentally, subject to possible errors in measurement. Thus, the minimum spanning tree one computes for G may not in fact be the "real" minimum spanning tree.

Given error parameters $\varepsilon > 0$ and $k > 0$, and a specific edge $e' = (u, v)$, you would like to be able to make a claim of the following form:

> $(*)$ Even if the cost of *each* edge were to be changed by at most ε (either increased or decreased), and the costs of k of the edges *other than* e' were further changed to arbitrarily different values, the edge e' would still not belong to any minimum spanning tree of G.

Such a property provides a type of guarantee that e' is not likely to be belong to the minimum spanning tree, even assuming significant measurement error.

Give a polynomial-time algorithm that takes G, e', ε, and k, and decides whether or not property $(*)$ holds for e'.

Solution. We test whether $e' = (u, v)$ could enter the MST as follows. We first lower the cost of e' by ε and raise the cost of all other edges by ε. We then form a graph G' by deleting all edges whose cost is greater than or equal to that of e'. Finally, we determine whether the minimum u-v cut in G has at most k edges.

If the answer is "yes," then by raising the cost of these k edges to ∞, we see that e' is one of the cheapest edges crossing a u-v cut, and so it belongs to an MST. Conversely, suppose e' can be made to enter the MST. Then this remains true if we continue to lower the cost of e' by as much as possible, and if we continue to raise the costs of all other edges by as much as possible. Now, at this point, consider the set E' of k edges whose costs we need to alter arbitrarily. If we delete E' from the graph, then there cannot be a u-v path consisting entirely of edges lighter than e'; thus, there is no u-v path in the graph G' defined above, and so E' defines a u-v cut of value at most k.

20. Let $G = (V, E)$ be a directed graph, and suppose that for each node v, the number of edges into v is equal to the number of edges out of v. That is, for all v,

$$|\{(u,v) : (u,v) \in E)\}| = |\{(v,w) : (v,w) \in E)\}|.$$

Let x, y be two nodes of G, and suppose that there exist k mutually edge-disjoint paths from x to y. Under these conditions, does it follow that there exist k mutually edge-disjoint paths from y to x? Give a proof, or a counter-example with explanation.

Solution. If we put a capacity of 1 on each edge, then by the integrality theorem for maximum flows, there exist k edge-disjoint x-y paths if and only if there exists a flow of value k. By the max-flow min-cut theorem, this latter condition holds if and only if there is no x-y cut (A, B) of capacity less than k.

Now suppose there were a y-x cut (B', A') of capacity strictly less than k, and consider the x-y cut (A', B'). We claim that the capacity of (A', B') is equal to the capacity of (B', A'). For if we let $\delta^-(v)$ and $\delta^+(v)$ denote the number of edges into and out of a node v respectively, then we have

$$\begin{aligned} c(A', B') - c(B', A') &= |\{(u,v) : u \in A', v \in B'\}| - |\{(u,v) : u \in B', v \in A'\}| \\ &= |\{(u,v) : u \in A', v \in B'\}| + |\{(u,v) : u \in A', v \in A'\}| - \\ &\quad\; |\{(u,v) : u \in A', v \in A'\}| - |\{(u,v) : u \in B', v \in A'\}| \\ &= \sum_{v \in A} \delta^+(v) - \sum_{v \in A} \delta^-(v) \\ &= 0. \end{aligned}$$

It follows that $c(A', B') < k$ as well, contradicting our observation in the first paragraph. Thus, every y-x cut has capacity at least k, and so there exist k edge-disjoint y-x paths.

21. Given a graph $G = (V, E)$, and a natural number k, we can define a relation $\xrightarrow{G,k}$ on pairs of vertices of G as follows. If $x, y \in V$, we say that $x \xrightarrow{G,k} y$ if there exist k mutually edge-disjoint paths from x to y in G.

Is it true that for every G and every $k \geq 0$, the relation $\xrightarrow{G,k}$ is transitive? That is, is it always the case that if $x \xrightarrow{G,k} y$ and $y \xrightarrow{G,k} z$, then we have $x \xrightarrow{G,k} z$? Give a proof or a counter-example.

22. Give a polynomial time algorithm for the following minimization analogue of the max-flow problem. You are given a directed graph $G = (V, E)$, with a source $s \in V$ and sink $t \in V$, and numbers (capacities) $\ell(v, w)$ for each edge $(v, w) \in E$. We define a flow f, and the value of a flow, as usual, requiring that all nodes except s and t satisfy flow conservation. However, the given numbers are lower bounds on edge flow, i.e., they require that $f(v, w) \geq \ell(v, w)$ for every edge $(v, w) \in E$, and there is no upper bound on flow values on edges.

    (a) Give a polynomial time algorithm that finds a feasible flow of minimum possible value.

    (b) Prove an analog of the max-flow min-cut theorem for this problem (i.e., does min-flow = max-cut?)

23. We define the *escape problem* as follows. We are given a directed graph $G = (V, E)$ (picture a network of roads); a certain collection of nodes $X \subset V$ are designated as *populated nodes*, and a certain other collection $S \subset V$ are designated as *safe nodes.* (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that (i) each node in X is the tail of one path, (ii) the last node on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.

    (a) Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.

    (b) Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus, we change (iii) to say "the paths do not share any *nodes.*"

    With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists.

    Also, provide an example with a given G, X, and S, in which the answer is "yes" to the question in (a) but "no" to the question in (b).

24. You are helping the medical consulting firm *Doctors Without Weekends* set up a system for arranging the work schedules of doctors in a large hospital. For each of the next n days, the hospital has determined the number of doctors they want on hand; thus, on day i, they have a requirement that *exactly* p_i doctors be present.

    There are k doctors, and each is asked to provide a list of days on which he or she is willing to work. Thus, doctor j provides a set L_j of days on which he or she is willing to work.

    The system produced by the consulting firm should take these lists, and try to return to each doctor j a list L'_j with the following properties.

(A) L'_j is a subset L_j, so that doctor j only works on days he or she finds acceptable.

(B) If we consider the whole set of lists $L'_1, \ldots, L'_k$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

(a) Describe a polynomial-time algorithm that implements this system. Specifically, give a polynomial-time algorithm that takes the numbers $p_1, p_2, \ldots, p_n$, and the lists $L_1, \ldots, L_k$, and does one of the following two things.

- Return lists $L'_1, L'_2, \ldots, L'_k$ satisfying properties (A) and (B); or
- Report (correctly) that there is no set of lists $L'_1, L'_2, \ldots, L'_k$ that satisfies both properties (A) and (B).

You should prove your algorithm is correct, and briefly analyze the running time.

(b) The hospital finds that the doctors tend to submit lists that are much too restrictive, and so it often happens that the system reports (correctly, but unfortunately) that no acceptable set of lists $L'_1, L'_2, \ldots, L'_k$ exists.

Thus, the hospital relaxes the requirements as follows. They add a new parameter $c > 0$, and the system now should try to return to each doctor j a list L'_j with the following properties.

(A*) L'_j contains at most c days that do not appear on the list L_j.

(B) *(Same as before.)* If we consider the whole set of lists $L'_1, \ldots, L'_k$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

Describe a polynomial-time algorithm that implements this revised system. It should take the numbers $p_1, p_2, \ldots, p_n$, the lists $L_1, \ldots, L_k$, and the parameter $c > 0$, and do one of the following two things.

- Return lists $L'_1, L'_2, \ldots, L'_k$ satisfying properties (A*) and (B); or
- Report (correctly) that there is no set of lists $L'_1, L'_2, \ldots, L'_k$ that satisfies both properties (A*) and (B).

In this question, you need only describe the algorithm; you do not need to explicitly write a proof of correctness or running time analysis. (However, your algorithm must be correct, and run in polynomial time.)

Solution. We construct a flow network as follows. There is a node u_j for each doctor j and a node v_i for each day i. There is an edge (u_j, v_i) of capacity 1 if doctor j can work on day i, and no edge otherwise. There is a source s, and an edge (s, u_j) of capacity $|L_j|$ for each j. There is a sink t, and an edge (v_i, t) of capacity p_i for each i.

Now we ask: is there an s-t flow of value $\sum_{i=1}^{n} p_i$ in this flow network? If there is, then there is an integer-valued flow, and we can produce a set of lists $\{L'_i\}$ from this as follows: assign doctor j to day i if there is a unit of flow on the edge (u_j, v_i). In this way, each doctor only works on days he or she finds acceptable, and each day i has p_i working on it. Conversely, if there is a valid set of lists for the doctors, then there will be a flow of value $\sum_{i=1}^{n} p_i$ in the network: we simply raise one unit of flow on each path s-u_j-v_i-t, where doctor j works on day i.

The total running time for this algorithm is dominated by the time for a flow computation on a graph with $O(kn)$ edges, where the total capacity out of the sink is $O(kn)$; thus, the total running time is $O(k^2n^2)$.

(b) We take the previous flow network and add some nodes to it, modeling the requirements. For each doctor j, we add a "spill-over" node u'_j. There is an edge (u'_j, v_i) of capacity 1 for each day i such that doctor j *doesn't* want to work on i. There is an edge (s, u'_j) of capacity c for each j.

Again we ask: is there an s-t flow of capacity $\sum_{i=1}^{n} p_i$ in this flow network? If there is, then there is an integer-valued flow, and we can produce a set of lists $\{L'_i\}$ from this as follows: assign doctor j to day i if there is a unit of flow on the edge (u_j, v_i) *or* if there is a unit of flow on the edge (u'_j, v_i).

25. You are consulting for an environmental statistics firm. They collect statistics, and publish the collected data in a book. The statistics is about populations of different regions in the world, and is in the millions. Examples of such statistics would look like the top table.

Country	A	B	C	Total
grownup men	11.998	9.083	2.919	24.000
grownup women	12.983	10,872	3.145	27.000
children	1.019	2.045	0.936	4.000
Total	26.000	22.000	7.000	55.000

We will assume here for simplicity that our data is such that all row and column sums are integers. The census rounding problem is to round all data to integers without changing any row or column sum. Each fractional number can be rounded either up or down without. For example a good rounding for the above data would be as follows.

Country	A	B	C	Total
grownup men	11.000	10.000	3.000	24.000
grownup women	13.000	10.000	4.000	27.000
children	2.000	2.000	0.000	4.000
Total	26.000	22.000	7.000	55.000

(a) Consider first the special case when all data is between 0 and 1. So you have a matrix of fractional numbers between 0 and 1, and your problem is to round each fraction that is between 0 and 1 to either 0 or 1 without changing the row or column sums. Use a flow computation to check if the desired rounding is possible.

(b) Consider the census data rounding problem as defined above, where row and column sums are integers, and you want round each fractional number α to either $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. Use a flow computation to check if the desired rounding is possible.

(c) Prove that the rounding we are looking for in (a) and (b) always exists.

26. $(*)$ Some friends of yours have grown tired of the game "Six degrees of Kevin Bacon" (after all, they ask, isn't it just breadth-first search?) and decide to invent a game with a little more punch, algorithmically speaking. Here's how it works.

You start with a set X of n actresses and a set Y of n actors, and two players P_0 and P_1. P_0 names an actress $x_1 \in X$, P_1 names an actor y_1 who has appeared in a movie with x_1, P_0 names an actress x_2 who has appeared in a movie with y_1, and so on. Thus, P_0 and P_1 collectively generate a sequence $x_1, y_1, x_2, y_2, \ldots$ such that each actor/actress in the sequence has co-starred with the actress/actor immediately preceding. A player P_i $(i = 0, 1)$ loses when it is P_i's turn to move, and he/she cannot name a member of his/her set who hasn't been named before.

Suppose you are given a specific pair of such sets X and Y, with complete information on who has appeared in a movie with whom. A *strategy* for P_i, in our setting, is an algorithm that takes a current sequence $x_1, y_1, x_2, y_2, \ldots$ and generates a legal next move for P_i (assuming it's P_i's turn to move). Give a polynomial-time algorithm that decides which of the two players can force a win, in a particular instance of this game.

Solution. Build a flow network G with vertices s, v_i for each $x_i \in X$, w_j for each $y_j \in Y$, and t. There are edges (s, v_i) for all i, (w_j, t) for all j, and (v_i, w_j) iff x_i and y_j have appeared together in a movie. All edges are given capacity 1. Consider a maximum s-t flow f in G; by the integrality theorem, it consists of a set $\{R_1, \ldots, R_k\}$ of edge-disjoint s-t paths, where k is the value of f. (Note that this is simply the construction we used to reduce bipartite matching to maximum flow.)

Suppose the value of f is n. Then each time player 1 names an actress x_i, player 2 can name the actor y_j so that (v_i, w_j) appear on a flow path together. In this way, player 1 must eventually run out of actresses and lose.

On the other hand, suppose the value of f is less than n. Player 1 can thus start with an actress x_i so that v_i lies on no flow path. Now, at every point of the game, we claim the vertex for the actor y_j named by player 2 lies on some flow path R_t. For if not, consider the first time when w_j does not lie on a flow path; if we take the sequence of edges in G traversed by the two players thus far, add s to the beginning and t to the end, we obtain an augmenting s-t path, which contradicts the maximality of f. Hence, each time player 2 names an actor y_j, player 1 can name an actress x_ℓ so that (x_ℓ, y_j) appear on a flow path together. In this way, player 2 must eventually out of actors and lose.

27. Statistically, the arrival of spring typically results in increased accidents, and increased need for emergency medical treatment, which often requires blood transfusions. Consider the problem faced by a hospital that is trying to evaluate whether their blood supply is sufficient.

The basic rule for blood donation is the following. A person's own blood supply has certain *antigens* present (we can think of antigens as a kind of molecular signature); and a person cannot receive blood with a particular antigen if their own blood does not have this antigen present. Concretely, this principle underpins the division of blood into four *types*: A, B, AB, and O. Blood of type A has the A antigen, blood of type B has the B antigen, blood of type AB has both, and blood of type O has neither. Thus, patients with type A can receive only blood types A or O in a transfusion, patients with type B can receive only B or O, patients with type O can receive only O, and patients with type AB can receive any of the four types.[1]

(a) Let s_O, s_A, s_B and s_{AB} denote the supply in whole units of the different blood types on hand. Assume that the hospital knows the projected demand for each blood type d_O, d_A, d_B and d_{AB} for the coming week. Give a polynomial time algorithm to evaluate if the blood on hand would suffice for the projected need.

(b) Consider the following example. Over the next week, they expect to need at most 100 units of blood. The typical distribution of blood types in US patients is 45% type O, 42% type A, 10% type B, and 3% type AB. The hospital wants to know if the blood supply they have on hand would be enough if 100 patients arrive with the expected type distribution. There is a total of 105 units of blood on hand. The table below gives these demands, and the supply on hand.

blood type:	O	A	B	AB
supply:	50	36	11	8
demand:	45	42	10	3

Is the 105 units of blood on hand enough to satisfy the 100 units of demand? Find an allocation that satisfies the maximum possible number of patients. Use an argument based on a minimum capacity cut to show why not all patients can receive blood. Also, provide an explanation for this fact that would be understandable to the clinic administrators, who have not taken a course on algorithms. (So, for example, this explanation should not involve the words "flow," "cut," or "graph" in the sense we use them in CS 482.)

Solution. To solve this problem, construct a graph $G = (V, E)$ as follows: Let the set of vertices consist of a super source node, four supply nodes (one for each blood type) adjacent to the source, four demand nodes and a super sink node that is adjacent to the demand nodes. For each supply node u and demand node v, construct an edge

[1]The Austrian scientist Karl Landsteiner received the Nobel Prize in 1930 for his discovery of the blood types A, B, O, and AB.

(u, v) if type v can receive blood from type u and set the capactity to ∞ or the demand for type v. Construct an edge (s, u) between the source s and each supply node u with the capacity set to the available supply of type u. Similarly, for each each demand node v and the sink t, construct an edge (v, t) with the capacity set to the demand for type v.

Now run the F-F algorithm on this graph to find the max-flow. If the edges from the demand nodes to the sink are all saturated in the resulting max-flow, then there is sufficient supply for the projected need. Using the running time in Section 6.5, we get a time bound of $O(\log C)$, where C is the total supply. (Note that the graph itself has constant size.)

To see why this must be the case, consider a max-flow where at least one of the demand to sink edges is not saturated. Denote this edge by (v, t). Let S be the set of of blood types that can donate to a person with blood type v. Note that all edges of the form (s, u) with $u \in S$ must then be saturated. If this were not the case we could push more flow along (u, v) for some $u \in S$. Now also note that since the capacities on the demand-sink edges is the expected demand, the max-flow at best satisfies this demand. If the demand-sink edges are saturated, then clearly the demand can be satisfied.

(b) Consider a cut containing the source, and the supply and demand nodes for B and A. The capacity of this cut is $50 + 36 + 10 + 3 = 99$, and hence all 100 units of demand cannot be satisfied.

An explanation for the clinic administrators: There are 87 people with demand for blood types O and A; these can only be satisfied by donors with blood types O and A; and there are only 86 such donors.

Grader's Comments: Since part (b) asked for an argument based on a minimum cut, it was necessary to actually specify a cut.

Part (a) can also be solved by a greedy algorithm; basically, it works as follows: The O group can only receive blood from O donors; so if the O group is not satisfied, there is no solution. Otherwise, satisfy the A and B groups using the leftovers from the O group; if this is not possible, there is no solution. Finally, satisfy the AB group using any remaining leftovers. A short explanation of correctness (basically following the above reasoning) is necessary for this algorithm, as it was with the flow algorithm.

28. Suppose you and your friend Alanis live, together with $n-2$ other people, at a popular off-campus co-operative apartment, The Upson Collective. Over the next n nights, each of you is supposed to cook dinner for the co-op exactly once, so that someone cooks on each of the nights.

Of course, everyone has scheduling conflicts with some of the nights (e.g. prelims, concerts, etc.) — so deciding who should cook on which night becomes a tricky task. For concreteness, let's label the people

$$\{p_1, \ldots, p_n\},$$

the nights

$$\{d_1, \ldots, d_n\};$$

and for person p_i, there's a set of nights $S_i \subset \{d_1, \ldots, d_n\}$ when they are *not* able to cook.

A *feasible dinner schedule* is an assignment of each person in the co-op to a different night, so that each person cooks on exactly one night, there is someone cooking on each night, and if p_i cooks on night d_j, then $d_j \notin S_i$.

(a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible dinner schedule for the co-op.

(b) Anyway, your friend Alanis takes on the task of trying to construct a feasible dinner schedule. After great effort, she constructs what she claims is a feasible schedule, and heads off to class for the day.

Unfortunately, when you look at the schedule she created, you notice a big problem. $n-2$ of the people at the co-op are assigned to different nights on which they are available: no problem there. But for the other two people — p_i and p_j — and the other two days — d_k and d_ℓ — you discover that she has accidentally assigned both p_i and p_j to cook on night d_k, and assigned no one to cook on night d_ℓ.

You want to fix Alanis's mistake, but without having to re-compute everything from scratch. Show that it's possible, using her "almost correct" schedule, to decide in only $O(n^2)$ time whether there exists a feasible dinner schedule for the co-op. (If one exists, you should also output it.)

Solution. **(a)** Let $G = (V, E)$ be a bipartite graph with a node $p_i \in V$ representing each person and a node $d_j \in V$ representing each night. The edges consist of all pairs (p_i, d_j) for which $d_j \notin S_i$.

Now, a perfect matching in G is a pairing of people and nights so that each person is paired with exactly one night, no two people are paired with the same night, and each person is available on the night they are paired with. Thus, if G has a perfect matching, then it has a feasible dinner schedule. Conversely, if G has a feasible dinner schedule, consisting of a set of pairs $S = \{(p_i, d_j)\}$, then each $(p_i, d_j) \in S$ is an edge of G, no two of these pairs share an endpoint in G, and hence these edges define a perfect matching in G.

(b) An algorithm is as follows. First, construct the bipartite graph G from part (a): it takes $O(1)$ time to decide for each pair (p_i, d_j) whether it should be an edge in G, so the total time to construct G is $O(n^2)$. Now, let Q denote the set of edges constructed by Alanis. Delete the edge (p_j, d_k) from Q, obtaining a set of edges Q'. Q' has size $n-1$, and since no person or night appears more than once in Q', it is a matching.

We now try to find an augmenting path in G with respect to Q', in time $O(|E|) = O(n^2)$. If we find such an augmenting path P, then increasing Q' using P gives us a matching of size n, which is a perfect matching and hence by (a) corresponds to a feasible dinner schedule. If G has no augmenting path with respect to Q', then by a theorem from class, Q' must be a maximum matching in G. In particular, this means that G has no perfect matching, and hence by (a) there is no feasible dinner schedule.

29. We consider the *bipartite matching* problem on a bipartite graph $G = (V, E)$. As usual, we say that V is partitioned into sets X and Y, and each edge has one end in X and the other in Y.

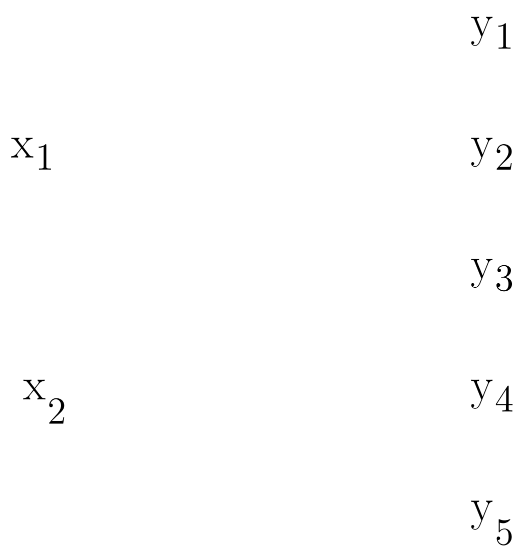


Figure 5: An instance of COVERAGE EXPANSION.

If M is a matching in G, we say that a node $y \in Y$ is *covered* by M if y is an end of one of the edges in M.

(a) Consider the following problem. We are given G and a matching M in G. For a given number k, we want to decide if there is a matching M' in G so that

(i) M' has k more edges than M does, *and*

(ii) every node $y \in Y$ that is covered by M is also covered by M'.

We call this the COVERAGE EXPANSION problem, with input G, M, and k. and we will say that M' is a *solution* to the instance.

Give a polynomial-time algorithm that takes an instance of COVERAGE EXPANSION and either returns a solution M' or reports (correctly) that there is no solution. (You should include an analysis of the running time, and a brief proof of why it is correct.)

Note: You may wish to also look at part (b) to help in thinking about this.

Example: Consider the accompanying figure, and suppose M is the matching consisting of the one edge (x_1, y_2). Suppose we are asked the above question with $k = 1$.

Then the answer to this instance of Coverage Expansion is "yes." We can let M' be the matching consisting (for example) of the two edges (x_1, y_2) and (x_2, y_4); M' it has 1 more edge than M, and y_2 is still covered by M'.

(b) Give an example of an instance of Coverage Expansion — specified by G, M, and k — so that the following situation happens.

> The instance has a solution; but in any solution M', the edges of M do not form a subset of the edges of M'.

(c) Let G be a bipartite graph, and let M be any matching in G. Consider the following two quantities.

- K_1 is the size of the largest matching M' so that every node y that is covered by M is also covered by M'.
- K_2 is the size of the largest matching M'' in G.

Clearly $K_1 \leq K_2$, since K_2 is obtained by considering *all possible* matchings in G.

Prove that in fact $K_1 = K_2$; that is, we can obtain a maximum matching even if we're constrained to cover all the nodes covered by our initial matching M.

Solution. **(a)** We set up a flow problem. We define a new graph G' for our flow problem by taking G and directing all edges from X to Y, and adding a super source s to the graph, and add edges (s, x) for all nodes $x \in X$ with capacity 1, and have each edge of G have capacity 1 also. We add a super sink t and add edges (y, s) for all nodes y that are **not covered** by the matching M. Now we define a flow problem where node s has a supply of $|M| + k$, node t has a demand of k, all nodes $y \in Y$ that are covered in M have a demand of 1. We claim that a flow satisfying these demands exists if an only if the matching M' exists.

If there is a flow in G' satisfying the demands, then by the integrality theorem there is an integer flow f'. The edges e in G that have $f'(e) = 1$ form the desired matching M'.

If there is a matching M', then we can define a flow f' in G' as follows. For edges $e = (x, y)$ we have $f'(e) = 1$ if e is in M' and 0 otherwise, we set $f'(e) = 1$ for edges $e = (s, x)$ if node x is covered by M', and finally, we set $f'(e) = 1$ for edges $e = (y, t)$ if node y is covered by M', but not covered by M. This satisfies all the demands.

It takes $O(m)$ time to build the flow network, and $O(mC) = O(mn)$ time to determine whether the desired flow exists.

(b) For example, take a 4 node graph with two nodes on each side, and edges (x_1, y_1), (x_1, y_2) and (x_2, y_2). Let M have the single edge (x_1, y_2) and set $k = 1$. Then the solution exists, but only by deleting the edge (x_1, y_2), and reassigning y_2 to x_2.

(c) Consider the graph G' analogous to the one we used in part (a), but connecting **all** nodes $y \in Y$ to the new sink t via an edge of capacity 1. Instead of using demands, we view this as a standard maximum flow problem — this is just the construction from the notes. We will show that there is a maximum matching that covers all nodes in Y that were covered by M. This implies that $K_1 \geq K_2$, and hence they are equal.

The matching M gives rise to a flow of value $|M|$ in G. Let f denote this flow. We use the Ford-Fulkerson algorithm, but instead of starting from the all-zero flow, we start from the integer flow f. This results in an integer maximum flow f'. The value of f' is K_2. We claim that the corresponding matching M' covers all nodes in Y that are covered by matching M. A matching corresponding to an integer flow f in G' covers exactly those nodes in Y for which $f(e) = 1$ for $s = (y, t)$. The the statement above follows from the observation that for any node $y \in Y$ and edge $e = (y, t)$ if $f(e) = 1$ and we obtain f' by the Ford-Fulkerson algorithm starting from the flow f, then $f'(e) = 1$ also. For a flow augmentation to decrease the value of the flow on an edge e, we would have to use the corresponding backwards edge in the augmenting path, but this backwards leaves t, and hence is not part of any simple s-t paths.

30. Suppose you're a consultant for the Ergonomic Architecture Commission, and they come to you with the following problem.

They're really concerned about designing houses that are "user-friendly," and they've been having a lot of trouble with the set-up of light fixtures and switches in newly designed houses. Consider, for example, a one-floor house with n light fixtures and n locations for light switches mounted in the wall. You'd like to be able to wire up one switch to control each light fixture, in such a way that a person at the switch can *see* the light fixture being controlled.

Sometimes this is possible, and sometimes it isn't. Consider the two simple floor plans for houses in the accompanying figure. There are three light fixtures (labeled a, b, c) and three switches (labeled 1, 2, 3). It is possible to wire switches to fixtures in the example on the left so that every switch has line-of-sight to the fixture, but this is not possible in the example on the right.

Let's call a floor plan — together with n light fixture locations and n switch locations — "ergonomic" if it's possible to wire one switch to each fixture so that every fixture is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments in the plane (the walls), where the i^{th} wall has endpoints $(x_i, y_i), (x'_i, y'_i)$. Each of the n switches and each of the n fixtures is given by its coordinates in the plane. A fixture is *visible* from a switch if the line segment joining them does not cross any of the walls.

Give an algorithm to decide if a given floor plan, in the above representation, is ergonomic. The running time should be polynomial in m and n. You may assume that you have a subroutine with $O(1)$ running time that takes two line segments as input and decides whether or not they cross in the plane.

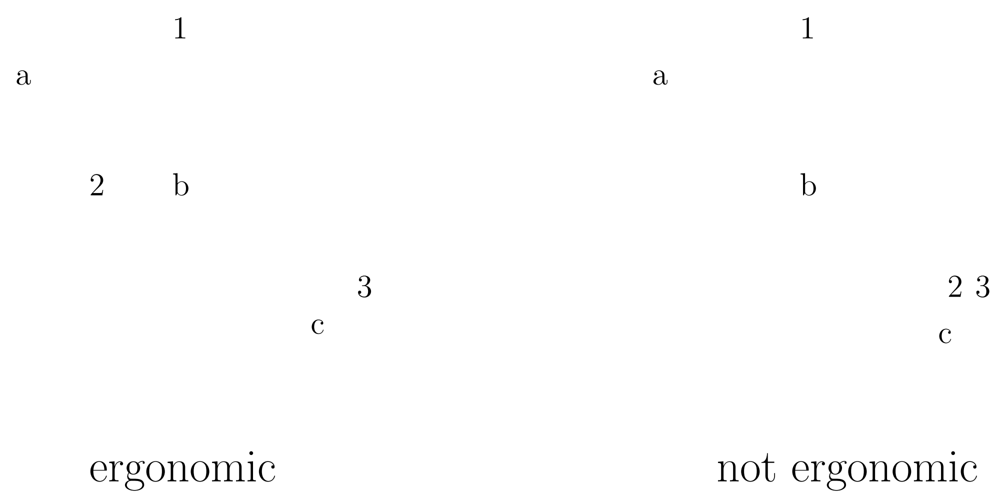


Figure 6: Two floor plans with lights and switches.

Solution. We build the following bipartite graph $G = (V, E)$. V is partitioned into sets X and Y, with a node $x_i \in X$ representing switch i, and a node $y_j \in Y$ representing fixture j. (x_i, y_j) is an edge in E if and only if the line segment from x_i to y_j does not intersect any of the m walls in the floor plan. Thus, whether $(x_i, y_j) \in E$ can be determined initially by m segment-intersection tests; so G can be built in time $O(n^2 m)$.

Now, we test in $O(n^3)$ time whether G has a perfect matching, and declare the floor plan to be "ergonomic" if and only if G does have a perfect matching. Our answer is always correct, since a perfect matching in G is a pairing of the n switches and the n fixtures in such a way that each switch can see the fixture it is paired with, by the definition of the edge set E; conversely, such a pairing of switches and fixtures defines a perfect matching in G.

31. Some of your friends are interning at the small high-tech company WebExodus. A running joke among the employees there is that the back room has less space devoted to high-end servers than it does to empty boxes of computer equipment, piled up in case something needs to be shipped back to the supplier for maintainence.

A few days ago, a large shipment of computer monitors arrived, each in its own large box; and since there are many different kinds of monitors in the shipment, the boxes do not all have the same dimensions. A bunch of people spent some time in the morning trying to figure out how to store all these things, realizing of course that less space would be taken up if some of the boxes could be *nested* inside others.

Suppose each box i is a rectangular parallelepiped with side lengths equal to (i_1, i_2, i_3); and suppose each side length is strictly between half a meter and one meter. Geometrically, you know what it means for one box to nest inside another — it's possible if you can rotate the smaller so that it fits inside the larger in each dimension. Formally, we can say that box i with dimensions (i_1, i_2, i_3) *nests* inside box j with dimensions (j_1, j_2, j_3) if there is a permutation a, b, c of the dimensions $\{1, 2, 3\}$ so that $i_a < j_1$, and $i_b < j_2$, and $i_c < j_3$. Of course, nesting is recursive — if i nests in j, and j nests in k, then by putting i inside j inside k, only box k is visible. We say that a *nesting arrangement* for a set of n boxes is a sequence of operations in which a box i is put inside another box j in which it nests; and if there were already boxes nested inside i, then these end up inside j as well. (Also notice the following: since the side lengths of i are more than half a meter each, and since the side lengths of j are less than a meter each, box i will take up more than half of each dimension of j, and so after i is put inside j, nothing else can be put inside j.) We say that a box k is *visible* in a nesting arrangement if the sequence of operations does not result in its ever being put inside another box.

So this is the problem faced by the people at WebExodus: Since only the visible boxes are taking up any space, how should a nesting arrangement be chosen so as to minimize the *number* of visible boxes?

Give a polynomial-time algorithm to solve this problem.

Example. Suppose there are three boxes with dimensions $(.6, .6, .6)$, $(.75, .75, .75)$, and $(.9, .7, .7)$. Then the first box can be put into either of the second or third boxes; but in any nesting arrangement, both the second and third boxes will be visible. So the minimum possible number of visible boxes is two, and one solution that achieves this is to nest the first box inside the second.

Solution. This is a complete parallel to the airline crew scheduling problem in the book. The boxes are like the flights, and where we previously encoded the idea that one crew can perform flight i followed by flight j, we analogously encode the idea that box i can be nested inside box j.

More precisely we reduce the given problem to a max flow problem where units of flow correspond to sets of boxes nested inside one visible box. We construct the following graph G:

- For each box i, G has two nodes u_i and v_i and an edge between them that corresponds to this box. This edge (u_i, v_i) has a lower bound of 1 and a capacity of 1. *(Each box is exactly in one set of boxes nested one in another.)*
- For each i and j so that box j nests inside box i, there is an edge (v_i, u_j). *(One can store box j inside i.)*
- G also has a source node s (corresponding to the back room where boxes are stored) and a sink node t (corresponding to nothing inside empty boxes).
- For each i, G has an edge (s, u_i) with a lower bound of 0 and a capacity of 1. *(Any box can be visible.)*

- For each j, G has an edge (v_j, t) with a lower bound of 0 and a capacity of 1. *(Any box can be empty.)*

We claim the following:

Fact 1 *There is a nesting arrangement with k visible boxes if and only of there is a feasible circulation in G with demand $-k$ in the source node s and demand k in the sink t.*

Proof. First, suppose there is a nesting arrangement with k visible boxes. Each sequence of nested boxes inside one visible box i_1, i_2, $\ldots$, i_n defines a path from s to t:

$$(s, u_{i_1}, v_{i_1}, u_{i_2}, v_{i_2}, \ldots, u_{i_n}, v_{i_n}, t)$$

Therefore we have k paths from s to t. The circulation corresponding to all these paths satisfy all demands, capacity and lower bound.

Conversely, consider a feasible circulation in our network. Without lost of generality, assume that this circulation has integer flow values.

There are exactly k edges going to t that carries one unit of flow. Consider one of such edges (v_i, t). We know that (u_i, v_i) has one unit of flow. Therefore, there is a unique edge into u_i that carries one unit of flow. If this edge is of the kind (v_j, u_i) then put box i inside j and continue with box j. If this edge of the kind (s, u_i), then put the box i in the back room. This box became visible. Continuing in this way we pack all boxes into k visible ones.

So we can answer the question whether there is a nesting arrangement with exactly k visible boxes. Now to find the minimum possible number of visible boxes we answer this question for $k = 1, 2, 3$, and so on, until we find a positive answer. The maximum number of this iteration is n, therefore the algorithm is polynomial since we can find a feasible circulation in polynomial time.

32. **(a)** Suppose you are given a flow network with integer capacities, together with a maximum flow f that has been computed in the network. Now, the capacity of one of the edges e out of the source is raised by one unit. Show how to compute a maximum flow in the resulting network in time $O(m + n)$, where m is the number of edges and n is the number of nodes.

    (b) You're designing an interactive image segmentation tool that works as follows. You start with the image segmentation set-up described in Section 6.8, with n pixels, a set of neighboring pairs, and parameters $\{a_i\}$, $\{b_i\}$, and $\{p_{ij}\}$. We will make two assumptions about this instance. First, we will suppose that each of the parameters $\{a_i\}$, $\{b_i\}$, and $\{p_{ij}\}$ is a non-negative integer between 0 and d, for some number d. Second, we will suppose that the neighbor relation among the pixels has the property that each pixel is a neighbor of at most four other pixels (so in the resulting graph, there are at most four edges out of each node).

You first perform an *initial segmentation* (A_0, B_0) so as to maximize the quantity $q(A_0, B_0)$. Now, this might result in certain pixels being assigned to the background, when the user knows that they ought to be in the foreground. So when presented with the segmentation, the user has the option of mouse-clicking on a particular pixel v_1, thereby bringing it to the foreground. But the tool should not simply bring this pixel into the foreground; rather, it should compute a segmentation (A_1, B_1) that maximizes the quantity $q(A_1, B_1)$ *subject to the condition that* v_1 *is in the foreground.* (In practice, this is useful for the following kind of operation: in segmenting a photo of a group of people, perhaps someone is holding a bag that has been accidentally labeled as part of the background. By clicking on a single pixel belonging to the bag, and re-computing an optimal segmentation subject to the new condition, the whole bag will often become part of the foreground.)

In fact, the system should allow the user to perform a sequence of such mouse-clicks $v_1, v_2, \ldots, v_t$; and after mouse-click v_i, the system should produce a segmentation (A_i, B_i) that maximizes the quantity $q(A_i, B_i)$ subject to the condition that all of $v_1, v_2, \ldots, v_i$ are in the foreground.

Give an algorithm that performs these operations so that the initial segmentation is computed within a constant factor of the time for a single maximum flow, and then the interaction with the user is handled in $O(dn)$ time per mouse-click.

(Note: Part (a) is a useful primitive for doing this. Also, the symmetric operation of forcing a pixel to belong to the background can be handled by analogous means, but you do not have to work this out here.)

Solution. This part is easy. We just check whether it's possible to augment once in the residual graph.

Let G be an original network and G' be a network where the capacity of edge e is raised by 1. Let f be a maximum flow of G.

Fact 2a If there is a augment path in the residual graph G'_f then the maximum flow of G' is f augmented by this path. Otherwise, the maximum flow of G' still f.

Proof. It is clear that in a new network the value of any cut may be increased at most by 1, because we increase the capacity of only one edge by 1. Therefore any integer flow in G' with the value grater than the value of f is a maximum flow.

If there is a path in the residual graph G'_f, then there is an augment of f. The resulting flow has a greater value than f has, therefore it is a maximum flow.

On the other hand, if there is no path in the residual graph G'_f, then we know that f is maximum flow.

Using BFS or DFS we can check whether there is a $s - t$ path in G'_f in time $O(m+n)$.

(b) To make sure that a selected pixel v will be in the foreground we consider the same network as in textbook, but increase the capacity on the edge (s, v) to $5d+1$. Note that this is greater than the total capacity of all edges outgoing from v (at most four edges to neighbor pixels, plus the edge to t). So the minimum cut will now definitely has v on the source side (otherwise we could move v to source side and decrease the capacity of the cut).

Therefore the following algorithm works. Compute a maximum flow f and find a minimum cut, as in the book. Then, for each pixel v selected by the user, we want to raise the capacity on the edge (s, v) to $5d+1$. To do so we increase the capacity of this edge $5d + 1 - a$ times by 1 each time (where a is the old capacity). Each time we will use algorithm of part (a) to compute a new maximum flow in $O(m + n)$ time. Then we compute a minimum cut in $O(m)$ time.

The overall time per mouse-click is $(5d + 1)O(m + n) + O(m) = O(d(m + n))$. Since each non-source node in our graph has at most 5 outgoing edges and s has n outgoing edges, the total number of edges in the graph is $m \leq 5n + n = O(n)$. Therefore $O(d(m + m)) = O(dn)$.

33. We now consider a different variation on the image segmentation problem from Section 6.8. We will develop a solution to an *image labeling* problem, where the goal is to label each pixel with a rough estimate of its distance from the camera (rather than the simple *foreground/background* labeling used in the text). The possible labels for each pixel will be $0, 1, 2, \ldots, M$ for some integer M.

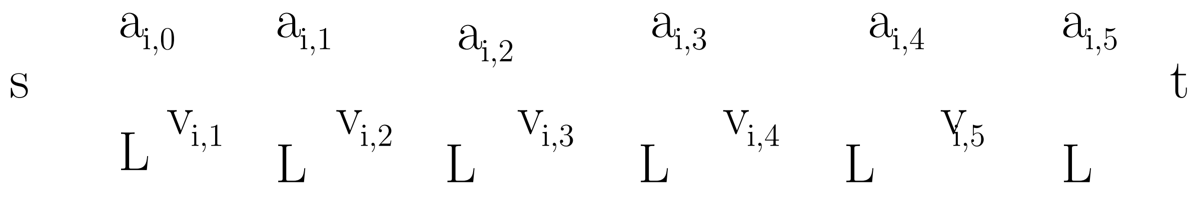
    Let $G = (V, E)$ denote the graph whose nodes are pixels, and edges indicate neighboring pairs of pixels. A *labeling* of the pixels is a partition of V into sets $A_0, A_1, \ldots, A_M$, where A_k is the set of pixels that is labeled with distance k for $k = 0, \ldots, M$. We will seek a labeling of minimum *cost*; the cost will come from two types of terms. By analogy with the foreground/background segmentation problem, we will have an *assignment cost*: for each pixel i and label k, the cost $a_{i,k}$ is the cost of assigning label k to pixel i. Next, if two neighboring pixels $(i, j) \in E$ are assigned different labels, there will be a *separation* cost. In the book, we use a separation penalty p_{ij}. In our current problem, the separation cost will also depend on how far the two pixels are separated; specifically, it will be proportional to the difference in value between their two labels.

    Thus, the overall cost q' of a labeling is defined as follows:

$$q'(A_0, \ldots, A_M) = \sum_{k=0}^{M} \sum_{i \in A_i} a_{i,k} + \sum_{k<\ell} \sum_{\substack{(i,j)\in E \\ i\in A_k, j\in A_\ell}} (\ell - k)p_{ij}.$$

The goal of this problem is to develop a polynomial-time algorithm that finds the optimal labeling given the graph G and the penalty parameters $a_{i,k}$ and p_{ij}. The algorithm will be based on constructing a flow network, and we will start you off on designing the algorithm by providing a portion of the construction.

The flow network will have a source s and a sink t. In addition, for each pixel $i \in V$ we will have nodes $v_{i,k}$ in the flow network for $k = 1, \ldots, M$ as shown in the accompanying figure. ($M = 5$ in the example in the figure.)



For notational convenience, the nodes $v_{i,0}$ and $v_{i,M+1}$ will refer to s and t respectively, for any choice of $i \in V$.

We now add edges $(v_{i,k}, v_{i,k+1})$ with capacity $a_{i,k}$ for $k = 0, \ldots, M$; and edges $(v_{i,k+1}, v_{i,k})$ in the opposite direction with very large capacity L. We will refer to this collection of nodes and edges as the *chain* associated with pixel i.

Notice that if we make this very large capacity L large enough, then there will be no minimum cut (A, B) so that an edge of capacity L leaves the set A. (How large do we have to make it for this to happen?). Hence, for any minimum cut (A, B), and each pixel i, there will be exactly one low-capacity edge in the chain associated with i that leaves the set A. (You should check that if there were two such edges, then a large-capacity edge would also have to leave the set A.)

Finally, here's the question: Use the nodes and edges defined so far to complete the construction of a flow network with the property that a minimum-cost labeling can be efficiently computed from a minimum (s, t)-cut. You should prove that your construction has the desired property, and show to recover the minimum-cost labeling from the cut.

Solution. First, consider a cut (A_0, B_0), where A_0 is just $\{s\}$. Note, that there no edges with capacity L going out s, therefore the capacity of the cut (A_0, B_0) does not depend on L. Let L be greater than this capacity. Then any cut cutting an edge of capacity L has capacity at least L. So such cut can not be a minimum cut, because it has larger capacity than (A_0, B_0) cut.

Now consider a minimum cut (A, B). Let i be an arbitrary pixel. If $v_{i,k} \in A$ for some k then $v_{i,k-1}$ is also in A, since there is an edge $(v_{i,k}, v_{i,k-1})$ with capacity L and no edge with capacity L is out of A. Let $f(i)$ is the maximum number such that $v_{i,f(i)} \in A$ (such number exists because $v_{i,0} = s \in A$). Then A contains all nodes $v_{i,k}$ where $k \leq f(i)$, and A contains all nodes $v_{i,k}$ where $k > f(i)$.

This number $f(i)$ will represent the label of the pixel i (i.e. we put i in A_k when $f(i) = k$). So any labeling of the pixels corresponds to a cut (A, B) where $A = \{(v_{i,k} \mid k \leq \text{label of } i\}$. We have just proved that any *minimum* cut corresponds to some labeling. Up to now the capacity of such cut is

$$\sum_{k=0}^{M} a_{i,f(i)} = \sum_{k=0}^{M} \sum_{i:f(i)=k} a_{i,k}$$

which is equal to the first term of our cost function.

So the main issue remaining is to encode the separation costs. If pixels i and j are neighbors, we join $v_{i,k}$ and $v_{j,k}$, for each k, by two edges in both directions of capacity $p_{i,j}$. Consequently, (A, B) cuts the edges of the kind $(v_{i,k}, v_{j,k})$ when $v_{i,k} \in A$ and $v_{j,k} \in B$, i.e., when $k \leq f(i)$ and $k > f(j)$. For any pair of neighbors i and j if $f(j) \geq f(i)$ then there are exactly $f(j) - f(i)$ such k that $f(i) \leq k < f(j)$. Therefore there are $f(j) - f(i)$ edges out of A. The overall capacity of these edges is $(f(j) - f(i))p_{i,j}$ which is exactly the desired separation cost.

So we have proved that the cost of any labeling is equal to capacity of the corresponding cut. Hence, the minimum cost labeling corresponds to the minimum cut.

34. The goal of this problem is to suggest variants of the preflow-push algorithm that speed up the practical running time without ruining its worst case complexity. Recall that the algorithm maintains the invariant that $h(v) \leq h(w) + 1$ for all edges (v, w) in the residual graph of the current preflow. We proved that if f is a flow (not just a preflow) with this invariant, then it is a maximum flow. Heights were monotone increasing and the whole running time analysis depended on bounding the number of times nodes can increase their heights. Practical experience shows that the algorithm is almost always much faster than suggested by the worst case, and that the practical bottleneck of the algorithm is relabeling nodes (and not the unsaturating pushes that lead to the worst case in the theoretical analysis). The goal of the problems below is the decrease the number of relabelings by increasing heights faster than one-by-one. Assume you have a graph G with n nodes, m edges, capacities c, source s and sink t.

(a) The preflow-push algorithm, as described in the text, starts by setting the flow equal to the capacity c_e on all edges e leaving the source, setting the flow to 0 on all other edges, setting $h(s) = n$, and setting $h(v) = 0$ for all other nodes $v \in V$. Give an $O(m)$ procedure that for initializing node heights that is better than what we had in class. Your method should set the height of each node v be as high as possible given the initial flow.

(b) In this part we will add a new step, called *gap relabeling* to preflow-push, that will increase the labels of lots of nodes by more than one at a time. Consider a preflow f and heights h satisfying the invariant. A *gap* in the heights is an integer $0 < h < n$ so that no node has height exactly h. Assume h is a gap value, and let A be the set of nodes v with heights $n > h(v) > h$. *Gap relabeling* is to change the height of all nodes in A to n. Prove that the preflow/push algorithm with Gap relabeling is a valid max-flow algorithm. Note that the only new thing that you need to prove is that gap relabeling preserves the invariant above.

(c) In Section **??** we proved that $h(v) \leq 2n - 1$ throughout the algorithm. Here we will have a variant that has $h(v) \leq n$ throughout. The idea is that we "freeze" all nodes when they get to height n, i.e., nodes at hight n are no longer considered active, and hence are not used for push and relabel. This way at the end of the algorithm we have a preflow and height function that satisfies the invariant above, and so that all excess is at height n. Let B be the set of nodes v so that there is a path from v to t in the residual graph of the current preflow. Let $A = V - B$. Prove that at the end of the algorithm (A, B) is a minimum capacity $s - t$ cut.

(d) The algorithm in part (c) computes a minimum $s - t$ cut, but fails to find a maximum flow (as it ends with a preflow that has excesses). Give an algorithm that takes the preflow f at the end of the algorithm of part (c) and converts it into a max flow in at most $O(mn)$ time. Hint: consider nodes with excess and try to send the excess back to s using only edges that the flow came on.

7 NP and Computational Intractability

1. You want to get a break from all the homework and projects that you're doing, so you're planning a large party over the weekend. Many of your friends are involved in group projects, and you are worried that if you invite a whole group to your party, the group might start discussing their project instead of enjoying your party.

   Before you realize it, you've formulated the PARTY SELECTION PROBLEM. You have a set of F friends whom you're considering to invite, and you're aware of a set of k project groups, $S_1, \ldots, S_k$, among these friends. The problem is to decide if there is a set of n of your friends whom you could invite so that not all members of any one group are invited.

   Prove that the PARTY SELECTION PROBLEM is NP-complete.

2. Given an undirected graph $G = (V, E)$, a *feedback set* is a set $X \subseteq V$ with the property that $G - X$ has no cycles. The UNDIRECTED FEEDBACK SET problem asks: given G and k, does G contain a feedback set of size at most k? Prove that UNDIRECTED FEEDBACK SET is NP-complete.

   Solution. Given a set X of vertices, we can use depth-first search to determine if $G - X$ has no cycles. Thus UNDIRECTED FEEDBACK SET is in NP.

   We now show that VERTEX COVER can be reduced to UNDIRECTED FEEDBACK SET. Given a graph $G = (V, E)$ and integer k, construct a graph $G' = (V', E')$ in which each edge $(u, v) \in E$ is replaced by the four edges (u, x^1_{uv}), (u, x^2_{uv}), (v, x^1_{uv}), and (v, x^2_{uv})

for new vertices x_{uv}^i that appear only incident to these edges. Now, suppose that X is a vertex cover of G. Then viewing X as a subset of V', it is easy to verify that $G' - X$ has no cycles. Conversely, suppose that Y is a feedback set of G' of minimum cardinality. We may choose Y so that it contains no vertex of the form x_{uv}^i — for it does, then $Y \cup \{u\} - \{x_{uv}^i\}$ is a feedback set of no greater cardinality. Thus, we may view Y as a subset of V. For every edge $(u, v) \in E$, Y must intersect the four-node cycle formed by u, v, x_{uv}^1, and x_{uv}^2; since we have chosen Y so that it contains no node of the form x_{uv}^i, it follows that Y contains one of u or v. Thus, Y is a vertex cover of G.

3. Consider a set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A. (That is, $B_i \subseteq A$ for each i.)

   We say that a set $H \subseteq A$ is a *hitting set* for the collection $B_1, B_2, \ldots, B_m$ if H contains at least one element from each B_i — that is, if $H \cap B_i$ is not empty for each i. (So H "hits" all the sets B_i.)

   We now define the HITTING SET problem as follows. We are given a set $A = \{a_1, \ldots, a_n\}$, a collection $B_1, B_2, \ldots, B_m$ of subsets of A, and a number k. We are asked: is there a hitting set $H \subseteq A$ for $B_1, B_2, \ldots, B_m$ so that the size of H is at most k?

   Prove that HITTING SET is NP-complete.

   Solution. *Hitting Set* is in NP: Given an instance of the problem, and a proposed set H, we can check in polynomial time whether H has size at most k, and whether some member of each set S_i belongs to H.

   Hitting Set looks like a covering problem, since we are trying to choose at most k objects subject to some constraints. We show that *Vertex Cover* $\leq_P$ *Hitting Set*. Thus, we begin with an instance of *Vertex Cover*, specified by a graph $G = (V, E)$ and a number k. We must construct an equivalent instance of *Hitting Set*. In *Vertex Cover*, we are trying to choose at most k nodes to form a vertex cover. In *Hitting Set*, we are trying to choose at most k elements to form a hitting set. This suggests that we define the set A in the *Hitting Set* instance to be the V of nodes in the *Vertex Cover* instance. For each edge $e_i = (u_i, v_i) \in E$, we define a set $S_i = \{u_i, v_i\}$ in the *Hitting Set* instance.

   Now we claim that there is a hitting set of size at most k for this instance, if and only if the original graph had a vertex cover of size at most k. For if we consider a hitting set H of size at most k as a subset of the nodes of G, we see that every set is "hit," and hence every edge has at least one end in H: H is a vertex cover of G. Conversely, if we consider a vertex cover C of G, and consider C as a subset of A, we see that each of the sets S_i is "hit" by C.

4. Suppose you're helping to organize a summer sports camp, and the following problem comes up. The camp is supposed to have at least one counselor who's skilled at each of the n sports covered by the camp (baseball, volleyball, and so on). They have received

job applications from m potential counselors. For each of the n sports, there is some subset of the m applicants that is qualified in that sport. The question is: for a given number $k < m$, is it possible to hire at most k of the counselors and have at least one counselor qualified in each of the n sports? We'll call this the EFFICIENT RECRUITING PROBLEM.

Show that EFFICIENT RECRUITING PROBLEM is NP-complete.

Solution. The problem is in NP since, given a set of k counselors, we can check that they cover all the sports.

Suppose we had such an algorithm $\mathcal{A}$; here is how we would solve an instance of VERTEX COVER. Given a graph $G = (V, E)$ and an integer k, we would define a sport S_e for each edge e, and a counselor C_v for each vertex v. C_v is qualified in sport S_e if and only if e has an endpoint equal to v.

Now, if there are k counselors that, together, are qualified in all sports, the corresponding vertices in G have the property that each edge has an end in at least one of them; so they define a vertex cover of size k. Conversely, if there is a vertex cover of size k, then this set of counselors has the property that each sport is contained in the list of qualifications of at least one of them.

Thus, G has a vertex cover of size at most k if and only if the instance of EFFICIENT RECRUITING that we create can be solved with at most k counselors. Moreover, the instance of EFFICIENT RECRUITING has size polynomial in the size of G. Thus, if we could determine the answer to the EFFICIENT RECRUITING instance in polynomial time, we could also solve the instance of VERTEX COVER in polynomial time.

5. We've seen the Interval Scheduling problem in class; here we consider a computationally much harder version of it that we'll call MULTIPLE INTERVAL SCHEDULING. As before, you have a processor that is available to run jobs over some period of time. (E.g. 9 AM to 5 PM.)

   People submit jobs to run on the processor; the processor can only work on one job at any single point in time. Jobs in this model, however, are more complicated than we've seen in the past: each job requires a *set* of intervals of time during which it needs to use the processor. Thus, for example, a single job could require the processor from 10 AM to 11 AM, and again from 2 PM to 3 PM. If you accept this job, it ties up your processor during those two hours, but you could still accept jobs that need any other time periods (including the hours from 11 to 2).

   Now, you're given a set of n jobs, each specified by a set of time intervals, and you want to answer the following question: For a given number k, is it possible to accept at least k of the jobs so that no two of the accepted jobs have any overlap in time?

   Show that MULTIPLE INTERVAL SCHEDULING is NP-complete.

   Solution. The problem is in NP since, given a set of k intervals, we can check that none overlap.

Suppose we had such an algorithm $\mathcal{A}$; here is how we would solve an instance of 3-DIMENSIONAL MATCHING.

We are given a collection C of ordered triples (x_i, y_j, z_k), drawn from sets X, Y, and Z of size n each. We create an instance of MULTIPLE INTERVAL SCHEDULING in which we conceptually divide time into $3n$ disjoint *slices*, labeled $s_1, s_2, \ldots, s_{3n}$. For each triple (x_i, y_j, z_k), we define a job that requires the slices s_i, s_{n+j}, and s_{2n+k}.

Now, if there is a perfect tripartite matching, then this corresponds to a set of n jobs whose slices are completely disjoint; thus, this is a set of n jobs that can be all be accepted, since they have no overlap in time. Conversely, if there is a set of jobs that can all be accepted, then because they must not overlap in time, the corresponding set of n triples consists of completely disjoint elements; this is a perfect tripartite matching.

Hence, there is a perfect tripartite matching among the triples in C if and only if our algorithm $\mathcal{A}$ reports that the constructed instance of MULTIPLE INTERVAL SCHEDULING contains a set of n jobs that can be accepted to run on the processor. The size of the MULTIPLE INTERVAL SCHEDULING instance that we construct is polynomial in n (the parameter of the underlying 3-DIMENSIONAL MATCHING instance).

Another way to solve this problem is via INDEPENDENT SET: here is how we could use an algorithm $\mathcal{A}$ for MULTIPLE INTERVAL SCHEDULING to decide whether a graph $G = (V, E)$ has an independent set of size at least k. Let m denote the number of edges in G, and label them $e_1, \ldots, e_m$. As before, we divide time into m disjoint *slices*, $s_1, \ldots, s_m$, with slice s_i intuitively corresponding to edge e_i. For each vertex v, we define a job that requires precisely the slices s_i for which the edge e_i has an endpoint equal to v. Now, two jobs can both be accepted if and only if they have no time slices in common; that is, if and only if they aren't the two endpoints of some edge. Thus one can check that a set of jobs that can be simultaneously accepted corresponds precisely to an independent set in the graph G.

6. Since the 3-DIMENSIONAL MATCHING problem is NP-complete, it is natural to expect that the corresponding 4-DIMENSIONAL MATCHING problem is at least as hard. Let us define 4-DIMENSIONAL MATCHING as follows. Given sets W, X, Y, and Z, each of size n, and a collection C of ordered 4-tuples of the form (w_i, x_j, y_k, z_ℓ), do there exist n 4-tuples from C so that no two have an element in common?

   Prove that 4-DIMENSIONAL MATCHING is NP-complete.

   Solution. 4-DIMENSIONAL MATCHING is in NP, since we can check in $O(n)$ time, using an $n \times 4$ array initialized to all 0, that a given set of n 4-tuples is disjoint.

   We now show that 3-DIMENSIONAL MATCHING $\leq_P$ 4-DIMENSIONAL MATCHING. So consider an instance of 3-DIMENSIONAL MATCHING, with sets X, Y, and Z of size n each, and a collection C of ordered triples. We define an instance of 4-DIMENSIONAL MATCHING as follows. We have sets W, X, Y, and Z, each of size n, and a collection C' of 4-tuples defined so that for every $(x_j, y_k, z_\ell) \in C$, and every i between 1 and n, there is a 4-tuple (w_i, x_j, y_k, z_ℓ). This instance has a size that is polynomial in the size of the initial 3-DIMENSIONAL MATCHING instance.

If $A = (x_j, y_k, z_\ell)$ is a triple in C, define $f(A)$ to be the 4-tuple (w_j, x_j, y_k, z_ℓ); note that $f(A) \in C'$. If $B = (w_i, x_j, y_k, z_\ell)$ is a 4-tuple in C', define $f'(B)$ to be the triple (x_j, y_k, z_ℓ); note that $f'(B) \in C$. Given a set of n disjoint triples $\{A_i\}$ in C, it is easy to show that $\{f(A_i)\}$ is a set of n disjoint 4-tuples in C'. Conversely, given a set of n disjoint 4-tuples $\{B_i\}$ in C', it is easy to show that $\{f'(B_i)\}$ is a set of n disjoint triples in C. Thus, by determining whether there is a perfect 4-Dimensional matching in the instance we have constructed, we can solve the initial instance of 3-DIMENSIONAL MATCHING.

7. The following is a version of the INDEPENDENT SET problem. You are given a graph $G = (V, E)$ and an integer k. For this problem, we will call a set $I \subset V$ *strongly independent* if for any two nodes $v, u \in I$, the edge (v, u) does not belong to E, and there is also no path of 2 edges from u to v, i.e., there is no node w such that both $(u, w) \in E$ and $(w, v) \in E$. The STRONGLY INDEPENDENT SET problem is to decide whether G has a strongly independent set of size k.

   Prove that the STRONGLY INDEPENDENT SET problem is NP-complete.

8. Consider the problem of reasoning about the identity of a set from the size of its intersections with other sets. You are given a finite set U of size n, and a collection $A_1, \ldots, A_m$ of subsets of U. You are also given numbers $c_1, \ldots, c_m$. The question is: does there exist a set $X \subset U$ so that for each $i = 1, 2, \ldots, m$, the cardinality of $X \cap A_i$ is equal to c_i? We will call this an instance of the *Intersection Inference* problem, with input U, $\{A_i\}$, and $\{c_i\}$.

   Prove that *Intersection Inference* is NP-complete.

9. You're consulting for a small high-tech company that maintains a high-security computer system for some sensitive work that it's doing. To make sure this system is not being used for any illicit purposes, they've set up some logging software that records the IP addresses that all their users are accessing over time. We'll assume that each user accesses at most one IP address in any given minute; the software writes a log file that records, for each user u and each minute m, a value $I(u, m)$ that is equal to the IP address (if any) accessed by user u during minute m. It sets $I(u, m)$ to the null symbol $\perp$ if u did not access any IP address during minute m.

   The company management just learned that yesterday, the system was used to launch a complex attack on some remote sites. The attack was carried out by accessing t distinct IP addresses over t consecutive minutes: in minute 1, the attack accessed address i_1; in minute 2, it accessed address i_2; and so on, up to address i_t in minute t.

   Who could have been responsible for carrying out this attack? The company checks the logs, and finds to its surprise that there's no single user u who accessed each of the IP addresses involved at the appropriate time; in other words, there's no u so that $I(u, m) = i_m$ for each minute m from 1 to t.

So the question becomes: what if there were a small *coalition* of k users that collectively might have carried out the attack? We will say a subset S of users is a *suspicious coalition* if for each minute m from 1 to t, there is at least one user $u \in S$ for which $I(u, m) = i_m$. (In other words, each IP address was accessed at the appropriate time by at least one user in the coalition.)

The *Suspicious Coalition* problem asks: given the collection of all values $I(u, m)$, and a number k, is there a suspicious coalition of size at most k?

10. As some people remember, and many have been told, the idea of hypertext predates the World Wide Web by decades. Even hypertext fiction is a relatively old idea — rather than being constrained by the linearity of the printed page, you can plot a story that consists of a collection of interlocked virtual "places" joined by virtual "passages."[2] So a piece of hypertext fiction is really riding on an underlying directed graph; to be concrete (though narrowing the full range of what the domain can do) we'll model this as follows.

    Let's view the structure of a piece of hypertext fiction as a directed graph $G = (V, E)$. Each node $u \in V$ contains some text; when the reader is currently at u, they can choose to follow any edge out of u; and if they choose $e = (u, v)$, they arrive next at the node v. There is a start node $s \in V$ where the reader begins, and an end node $t \in V$; when the reader first reaches t, the story ends. Thus, any path from s to t is a valid *plot* of the story. Note that, unlike a Web browser, there is not necessarily a way to go back; once you've gone from u to v, you might not be able to ever return to u.

    In this way, the hypertext structure defines a huge number of different plots on the same underlying content; and the relationships among all these possibilities can grow very intricate. Here's a type of problem one encounters when reasoning about a structure like this. Consider a piece of hypertext fiction built on a graph $G = (V, E)$ in which there are certain crucial *thematic elements* — love; death; war; an intense desire to major in computer science; and so forth. Each thematic element i is represented by a set $T_i \subseteq V$ consisting of the nodes in G at which this theme appears. Now, given a particular set of thematic elements, we may ask: is there a valid plot of the story in which each of these elements is encountered? More concretely, given a directed graph G, with start node s and end node t, and thematic elements represented by sets $T_1, T_2, \ldots, T_k$, the *Plot Fulfillment* problem asks: is there a path from s to t that contains at least one node from each of the sets T_i?

    Prove that *Plot Fulfillment* is NP-complete.

    Solution. *Plot Fulfillment* is in NP: Given an instance of the problem, and a proposed s-t path P, we can check that P is a valid path in the graph, and that it meets each set T_i.

[2]See e.g. http://www.eastgate.com

Plot Fulfillment also looks like a covering problem; in fact, it looks a lot like the *Hitting Set* problem from the previous question: we need to "hit" each set T_i. However, we have the extra feature that the set with which we "hit" things is a path in a graph; and at the same time, there is no explicit constraint on its size. So we use the path structure to impose such a constraint.

Thus, we will show that *Hitting Set* $\leq_P$ *Plot Fulfillment.* Specifically, let us consider an instance of *Hitting Set*, with a set $A = \{a_1, \ldots, a_n\}$, subsets $S_1, \ldots, S_m$, and a bound k. We construct the following instance of *Plot Fulfillment.* The graph G will have nodes s, t, and

$$\{v_{ij} : 1 \leq i \leq k, \ 1 \leq j \leq n\}.$$

There is an edge from s to each v_{1j} ($1 \leq j \leq n$), from each v_{kj} to t ($1 \leq j \leq n$), and from v_{ij} to $v_{i+1,\ell}$ for each $1 \leq i \leq k-1$ and $1 \leq j, \ell \leq n$. In other words, we have a *layered graph*, where all nodes v_{ij} ($1 \leq j \leq n$) belong to "layer i", and edges go between consecutive layers. Intuitively the nodes v_{ij}, for fixed j and $1 \leq i \leq k$ all represent the element $a_j \in A$.

We now need to define the sets T_ℓ in the *Plot Fulfillment* instance. Guided by the intuition that v_{ij} corresponds to a_j, we define

$$T_\ell = \{v_{ij} : a_j \in S_\ell, \ 1 \leq i \leq k\}.$$

Now, we claim that there is a valid solution to this instance of *Plot Fulfillment* if and only if our original instance of *Hitting Set* had a solution. First, suppose there is a valid solution to the *Plot Fulfillment* instance, given by a path P, and let

$$H = \{a_j : v_{ij} \in P \text{ for some } i\}.$$

Notice that H has at most k elements. Also for each ℓ, there is some $v_{ij} \in P$ that belongs to T_ℓ, and the corresponding a_j belongs to S_ℓ; thus, H is a hitting set.

Conversely, suppose there is a hitting set $H = \{a_{j_1}, a_{j_2}, \ldots, a_{j_k}\}$. Define the path $P = \{s, v_{1,j_1}, v_{2,j_2}, \ldots, v_{k,j_k}, t\}$. Then for each ℓ, some a_{j_q} lies in S_ℓ, and the correponding node v_{q,j_q} meets the set T_ℓ. Thus P is a valid solution to the *Plot Fulfillment* instance.

11. *(We thank Maverick Woo for the idea of the Star Wars theme.)* There are those who insist that the initial working title for Episode XXVII of the Star Wars series was "P = NP" — but this is surely apocryphal. In any case, if you're so inclined, it's easy to find NP-complete problems lurking just below the surface of the original Star Wars movies.

    Consider the problem faced by Luke, Leia, and friends as they tried to make their way from the Death Star back to the hidden Rebel base. We can view the galaxy as an undirected graph $G = (V, E)$, where each node is a star system and an edge (u, v) indicates that one can travel directly from u to v. The Death Star is represented by a node s, the hidden Rebel base by a node t. Certain edges in this graph represent longer distances than others; thus, each edge e has an integer *length* $\ell_e \geq 0$. Also, certain edges represent routes that are more heavily patrolled by evil Imperial spacecraft; so each edge e also has an integer *risk* $r_e \geq 0$, indicating the expected amount of damage incurred from special-effects-intensive space battles if one traverses this edge.

It would be safest to travel through the outer rim of the galaxy, from one quiet upstate star system to another; but then one's ship would run out of fuel long before getting to its destination. Alternately, it would be quickest to plunge through the cosmopolitan core of the galaxy; but then there would be far too many Imperial spacecraft to deal with. In general, for any path P from s to t, we can define its *total length* to be the sum of the lengths of all its edges; and we can define its *total risk* to be the sum of the risks of all its edges.

So Luke, Leia, and company are looking at a complex type of shortest-path problem in this graph: they need to get from s to t along a path whose total length and total risk are *both* reasonably small. In concrete terms, we can phrase the *Galactic Shortest Path* problem as follows: given a set-up as above, and integer bounds L and R, is there a path from s to t whose total length is at most L, *and* whose total risk is at most R?

Prove that *Galactic Shortest Path* is NP-complete.

Solution. *Galactic Shortest Path* is in NP: given a path P in a graph, we can add up the lengths and risks of its edges, and compare them to the given bounds L and R.

Galactic Shortest Path involves adding numbers, so we naturally consider reducing from the *Subset Sum* problem. Specifically, we'll prove that *Subset Sum* $\leq_P$ *Galactic Shortest Path.*

Thus, consider an instance of *Subset Sum*, specified by numbers $w_1, \ldots, w_n$ and a bound W; we want to know if there is a subset S of these numbers that add up to exactly W. *Galactic Shortest Path* looks somewhat different on the surface, since we have *two kinds* of numbers (lengths and risks), and we are only given *upper bounds* on their sums. However, we can use the fact that we also have an underlying graph structure. In particular, by defining a simple type of graph, we can encode the idea of choosing a subset of numbers.

We define the following instance of *Galactic Shortest Path.* The graph G has a nodes $v_0, v_1, \ldots, v_n$. There are two edges from v_{i-1} to v_i, for each $1 \leq i \leq n$; we'll name them e_i and e'_i. (If one wants to work with a graph containing no parallel edges, we can add extra nodes that subdivide these edges into two; but the construction turns out the same in any case.)

Now, any path from v_0 to v_n in this graph G goes through edge one from each pair $\{e_i, e'_i\}$. This is very useful, since it corresponds to making n independent binary choices — much like the binary choices one has in *Subset Sum.* In particular, choosing e_i will represent putting w_i into our set S, and e'_i will represent leaving it out.

Here's a final observation. Let $W_0 = \sum_{i=1}^{n} w_i$ — the sum of all the numbers. Then a subset S adds up to W if and only if its completement adds up to $W_0 - W$.

We give e_i a length of w_i and a risk of 0; we give e'_i a length of 0 and a risk of w_i. We set the bound $L = W$, and $R = W_0 - W$. We now claim: there is a solution to the *Subset Sum* instance if and only if there is a valid path in G. For if there is a set S adding up to W, then in G we use the edges e_i for $i \in S$, and e'_j for $j \notin S$. This path has length W and risk $W_0 - W$, so it meets the given bounds. Conversely, if there is a path P meeting the given bounds, then consider the set $S = \{w_i : e_i \in P\}$. S adds up to at most W and its complement adds up to at most $W_0 - W$. But since the two sets together add up to exactly W_0, it must be that S adds up to exactly W and its complement to exactly $W_0 - W$. Thus, S is valid solution to the *Subset Sum* instance.

12. The mapping of genomes involves a large array of difficult computational problems. At the most basic level, each of an organism's chromosomes can be viewed as an extremely long string (generally containing millions of symbols) over the four-letter alphabet $\{a, c, g, t\}$. One family of approaches to genome mapping is to generate a large number of short, overlapping snippets from a chromosome, and then to infer the full long string representing the chromosome from this set of overlapping substrings.

    While we won't be able to go into these string assembly problems in full detail, here's a simplified problem that suggests some of the computational difficulty one encounters in this area. Suppose we have a set $S = \{s_1, s_2, \ldots, s_n\}$ of short DNA strings over a q-letter alphabet; and each string s_i has length 2ℓ, for some number $\ell \geq 1$. We also have a library of additional strings $T = \{t_1, t_2, \ldots, t_m\}$ over the same alphabet; each of these also has length 2ℓ. In trying to assess whether the string s_b might come directly after the string s_a in the chromosome, we will look to see whether the library T contains a string t_k so that the first ℓ symbols in t_k are equal to the last ℓ symbols in s_a, and the last ℓ symbols in t_k are equal to the first ℓ symbols in s_b. If this is possible, we will say that t_k *corroborates* the pair (s_a, s_b). (In other words, t_k could be a snippet of DNA that straddled the region in which s_b directly followed s_a.)

    Now, we'd like to concatenate all the strings in S in some order, one after the other with no overlaps, so that each consecutive pair is corroborated by some string in the library T. That is, we'd like to order the strings in S as $s_{i_1}, s_{i_2}, \ldots, s_{i_n}$, where $i_1, i_2, \ldots, i_n$ is a permutation of $\{1, 2, \ldots, n\}$, so that for each $j = 1, 2, \ldots, n-1$, there is a string t_k that corroborates the pair $(s_{i_j}, s_{i_{j+1}})$. (The same string t_k can be used for more than one consecutive pair in the concatenation.) If this is possible, we will say that the set S has a *perfect assembly.*

    Given sets S and T, the *Perfect Assembly* problem asks: does S have an assembly with respect to T? Prove that *Perfect Assembly* is NP-complete.

    Example. Suppose the alphabet is $\{a, c, g, t\}$, the set $S = \{ag, tc, ta\}$, and the set $T = \{ac, ca, gc, gt\}$. (So each string has length $2\ell = 2$.) Then the answer to this instance of *Perfect Assembly* is "yes" — we can concatanate the three strings in S in the order *tcagta*. (So $s_{i_1} = s_2$, $s_{i_2} = s_1$, and $s_{i_3} = s_3$.) In this order, the pair (s_{i_1}, s_{i_2}) is corroborated by the string *ca* in the library T, and the pair (s_{i_2}, s_{i_3}) is corroborated by the string *gt* in the library T.

13. Suppose you're consulting for a company that's setting up a Web site. They want to make the site publically accessible, but only in a limited way; people should be allowed to navigate through the site provided they don't become too "intrusive."

We'll model the site as a directed graph $G = (V, E)$, in which the nodes represent Web pages and the edges represent directed hyperlinks. There is a distinguished *entry node* $s \in V$. The company plans to regulate the flow of traffic as follows. They'll define a collection of *restricted zones* $Z_1, \ldots, Z_k$, each of which is a subset of V. These zones Z_i need not be disjoint from one another. The company has a mechanism to track the path followed by any user through the site; if the user visits a single zone more than once, an *alarm* is set off and the user's session is terminated.

The company wants to be able to answer a number of questions about its monitoring system; among them is the following EVASIVE PATH problem: Given G, $Z_1, \ldots, Z_k$, $s \in V$, and a *destination node* $t \in V$, is there an s-t path in G that does not set off an alarm? (I.e. it passes through each zone at most once.) Prove that EVASIVE PATH is NP-complete.

Solution. EVASIVE PATH is in NP since we may check, for an s-t path P, whether $|P \cap Z_i| \leq 1$ for each i.

Now let us suppose that we have an instance of 3-SAT with n variables $x_1, \ldots, x_n$ and t clauses. We create the following directed graph $G = (V, E)$. The vertices will be partitioned into *layers*, with each node in one layer having edges to each node in the next. Layer 0 will consist only of the node s. Layer i will have two nodes for $i = 1, \ldots, n$, three nodes for $i = n+1, \ldots, n+t$, and only the node t in layer $n+t+1$. Each node in layers $1, 2, \ldots, n+t$ will also be assigned a *label*, equal to one of the $2n$ possible literals. In layer i, for $1 \leq i \leq n$, we label one node with the literal x_i and one with the literal $\overline{x_i}$. In layer $n+i$, for $1 \leq i \leq t$, we label the three nodes with $\ell_{i_1}, \ell_{i_2}, \ell_{i_3}$, where $\{\ell_{i_j}\}$ are the three literals appearing in clause i. Finally, for every pair of nodes whose labels correspond to a variable and its negation, we define a distinct zone Z.

Now, if there is a satisfying assignment for the 3-SAT instance, we can define an s-t path P that passes only through nodes with the labels of literals set to TRUE; P is thus an evasive path. Conversely, consider any evasive path P; we define variable x_i to be TRUE if P passes through the vertex in layer i with label x_i, and FALSE if P passes through the vertex in layer i with label $\overline{x_i}$. Since P does not visit any zone a second time, it must therefore visit only nodes whose labels correspond to literals set to TRUE, and hence the 3-SAT instance is satisfiable.

14. Consider an instance of the *Satisfiability* problem, specified by clauses $C_1, \ldots, C_k$ over a set of Boolean variables $x_1, \ldots, x_n$. We say that the instance is *monotone* if each term in each clause consists of a non-negated variable; that is, each term is equal to x_i, for some i, rather than $\overline{x_i}$. Monotone instances of *Satisfiability* are very easy to solve: they are always satisfiable, by setting each variable equal to 1.

For example, suppose we have the three clauses

$$(x_1 \vee x_2), (x_1 \vee x_3), (x_2 \vee x_3).$$

This is monotone, and indeed the assignment that sets all three variables to 1 satisfies all the clauses. But we can observe that this is not the only satisfying assignment; we could also have set x_1 and x_2 to 1, and x_3 to 0. Indeed, for any monotone instance, it is natural to ask how few variables we need to set to 1 in order to satisfy it.

Given a monotone instance of *Satisfiability*, together with a number k, the problem of *Monotone Satisfiability with Few True Variables* asks: is there a satisfying assignment for the instance in which at most k variables are set to 1? Prove this problem is NP-complete.

Solution. On the surface, *Monotone Satisfiability with Few True Variables* (which we'll abbreviate *MSwFTV*) is written in the language of the Satisfiability problem. But at a technical level, it's not so closely connected to *SAT*; after all no variables appear negated, and what makes it hard is the constraint that only a few variables can be set to true.

Really, what's going on is that one has to choose a small number of variables, in such a way that each clause contains one of the chosen variables. Phrased this way, it resembles a type of covering problem.

We choose *Vertex Cover* as the problem X, and show *Vertex Cover* $\leq_P$ *MSwFTV*. Suppose we are given a graph $G = (V, E)$ and a number k; we want to decide whether there is a vertex cover in G of size at most k. We create an equivalent instance of *MSwFTV* as follows. We have a variable x_i for each vertex v_i. For each edge $e_j = (v_a, v_b)$, we create the clause $C_j = (x_a \vee x_b)$. This is the full instance: we have clauses $C_1, C_2, \ldots, C_m$, one for each edge of G, and we want to know if they can all be satisfied by setting at most k variables to 1.

We claim that the answer to the *Vertex Cover* instance is "yes" if and only if the answer to the *MSwFTV* instance is "yes." For suppose there is a vertex cover S in G of size at most k, and consider the effect of setting the corresponding variables to 1 (and all other variables to 0). Since each edge is covered by a member of S, each clause contains at least one variable set to 1, and so all clauses are satisfied. Conversely, suppose there is a way to satisfy all clauses by setting a subset X of at most k variables to 1. Then if we consider the corresponding vertices in G, each edge must have at least one end equal to one of these vertices — since the clause corresponding to this edge contains a variable in X. Thus the nodes corresponding to the variables in X form a vertex cover of size at most k.

15. Suppose you're consulting for a group that manages a high-performance real-time system in which asynchronous process make use of shared resources. Thus, the system has a set of n *processes* and a set of m *resources*. At any given point in time, each process

specifies a set of resources that it requests to use. Each resource might be requested by many processes at once; but it can only be used by a single process at a time.

Your job is to allocate resources to processes that request them. if a process is allocated all the resources it requests, then it is *active*; otherwise it is *blocked*. You want to perform the allocation so that as many processes as possible are active. Thus, we phrase the RESOURCE RESERVATION problem as follows: given a set of process and resources, the set of requested resources for each process, and a number k, is it possible to allocate resources to processes so that at least k processes will be active?

Show that RESOURCE RESERVATION is NP-complete.

Solution. The RESOURCE RESERVATION problem can be restated as follows. We have a set of m resources, and n processes, each of which requests a subset of the resources. The problem is to decide if there is a set of k processes whose requested resource sets are disjoint.

We first show the problem is in NP. To see this, notice that if we are given a set of k processes, we can check in polynomial time that no resource is requested by more than one of them.

To prove that the RESOURCE RESERVATION problem in NP-complete we use the independent set problem, which is known to be NP-complete. We show that

INDEPENDENT SET $\leq_P$ RESOURCE RESERVATION

Given an instance of the independent set problem — specified by a graph G and a number k — we create an equivalent resource reservation problem. The resources are the edges, the processes correspond to the nodes of the graph, and the process corresponding to node v requests the resources corresponding to the edges incident on v. Note that this reduction takes polynomial time to compute. We need to show two things to see that the resource reservation problem we created is indeed equivalent.

First, if there are k processes whose requested resources are disjoint, then the k nodes corresponding to these processes form an independent set. This is true as any edge between these nodes would be a resource that they both request.

If there is an independent set of size k, then the k processes corresponding to these nodes form a set of k processes that request disjoint sets of resources.

16. You're configuring a large network of workstations, which we'll model as an undirected graph G — the nodes of G represent individual workstations and the edges represent direct communication links. The workstations all need access to a common *core database*, which contains data necessary for basic operating system functions.

You could replicate this database on each workstation — then lookups would be very fast from any workstation, but you'd have to manage a huge number of copies. Alternately, you could keep a single copy of the database on one workstation and have the remaining workstations issue requests for data over the network G; but this could result in large delays for a workstation that's many hops away from the site of the database.

So you decide to look for the following compromise: you want to maintain a small number of copies, but place them so that any workstation either has a copy of the database, or is connected by a direct link to a workstation that has a copy of the database.

Thus, we phrase the SERVER PLACEMENT problem as follows: Given the network G, and a number k, is there a way to place k copies of the database at k different workstations so that every workstation either has a copy of the database, or is connected by a direct link to a workstation that has a copy of the database?

Solution. The SERVER PLACEMENT problem is in NP: Given a set of k proposed servers, we can verify in polynomial time that all non-server workstations have a direct link to a workstation that is a server.

To prove that the problem in NP-complete we use the VERTEX COVER problem that is known to be NP-complete. We show that

VERTEX COVER $\leq_P$ SERVER PLACEMENT

It is worth noting that SERVER PLACEMENT is not the *same* problem as VERTEX COVER: for example, on a graph consisting of three mutually adjacent nodes, we need to choose only one node to have a copy of the database *adjacent* to all other nodes; but we need to choose two nodes to obtain a *vertex cover*.

Consider an instance of VERTEX COVER with a graph G and a number k. Let r denote the number of nodes in G that are not adjacent to any edge; these nodes will be called *isolated*. To create the equivalent SERVER PLACEMENT problem we create a new graph G' by adding a parallel copy to every edge, and adding a new node in the middle of each of the new edges. More precisely, if G has an edge (i, j) we add a new node v_{ij}, and add new edges (i, v_{ij}) and (v_{ij}, j). Now we claim that G contains a set S of vertices of size at most k so that every edge of G has at least one end in S if and only if G' has a solution to the SERVER PLACEMENT problem with $k+r$ servers. Note again that this reduction takes polynomial time to compute.

Let I denote the set of isolated nodes in G. If G has a vertex cover S of size k, then placing servers on the set $S \cup I$ we get a solution to the server placement problem with at most $k + r$ servers. We need to show that this set is a solution to the server placement; that is, we must show that for each vertex that is not in S, there is an adjacent vertex is in S. First consider a new vertex v_{ij}. The set S is a vertex cover, so either i or j must be in the set S, and so S contains a node directly connected to v_{ij} in G'. Now consider a node i of the original graph G, and let (i, j) be any edge adjacent to i. Either i or j is in S, so again there is an node directly connected to i that has a server.

Now consider the other direction of the equivalence: Assume that there is a solution to the Server Placement problem with at most $k + r$ servers. First notice that all isolated nodes must be servers. Next notice that we can modify the solution so that we only use the original nodes of the graph as servers. If a node v_{ij} is used as a server than we can replace this node by either i and j and get an alternate solution with the same number of servers: the node v_{ij} is can serve requests from the nodes v_{ij}, i, j, and either i or j can do the same.

Next we claim that if there is a solution to the SERVER PLACEMENT problem where a set S of nodes of the original graph are used as servers, then S forms a vertex cover of size k. This is true, as for each edge (i, j) the new node v_{ij} must have a directly connected node with a server, and hence we must have that either i or j is in S.

17. Three of your friends work for a large computer-game company, and they've been working hard for several months now to get their proposal for a new game, *Droid Trader!*, approved by higher management. In the process, they've had to endure all sorts of discouraging comments, ranging from, "You're really going to have to work with Marketing on the name," to, "Why don't you emphasize the parts where people get to kick each other in the head?"

    At this point, though, it's all but certain that the game is really heading into production, and your friends come to you with one final issue that's been worrying them: What if the overall premise of the game is too simple, so that players get really good at it and become bored too quickly?

    It takes you a while, listening to their detailed description of the game, to figure out what's going on; but once you strip away the space battles, kick-boxing interludes, and Stars-Wars-inspired-pseudo-mysticism, the basic idea is as follows. A player in the game controls a spaceship and is trying to make money buying and selling droids on different planets. There are n different types of droids, and k different planets. Each planet p has the following properties: there are $s(j, p) \geq 0$ droids of type j available for sale, at a fixed price of $x(j, p) \geq 0$ each, for $j = 1, 2, \ldots, n$; and there is a demand for $d(j, p) \geq 0$ droids of type j, at a fixed price of $y(j, p) \geq 0$ each. (We will assume that a planet does not simultaneously have both a positive supply and a positive demand for a single type of droid; so for each j, at least one of $s(j, p)$ or $d(j, p)$ is equal to 0.)

    The player begins on planet s with z units of money, and must end at planet t; there is a directed acyclic graph G on the set of planets, such that s-t paths in G correspond to valid routes by the player. (G is chosen to be acyclic to prevent arbitrarily long games.) For a given s-t path P in G, the player can engage in transactions as follows: whenever the player arrives at a planet p on the path P, she can buy up to $s(j,p)$ droids of type j for $x(j,p)$ units of money each (provided she has sufficient money on hand) and/or sell up to $d(j,p)$ droids of type j for $y(j,p)$ units of money each (for $j = 1, 2, \dots, n$). The player's *final score* is the total amount of money she has on hand when she arrives at planet t. (There are also bonus points based on space battles and kick-boxing, which we'll ignore for the purposes of formulating this question ...)

So basically, the underlying problem is to achieve a high score. In other words, given an instance of this game, with a directed acyclic graph G on a set of planets, all the other parameters described above, and also a target bound B, is there an path P in G and a sequence of transactions on P so that the player ends with a final score that is at least B? We'll call this an instance of the *High-Score-on-Droid-Trader!* problem, or *HSoDT!* for short.

Prove that *HSoDT!* is NP-complete, thereby guaranteeing (assuming $P \neq NP$) that there isn't a simple strategy for racking up high scores on your friends' game.

Solution. We start off by showing that *HSoDT!* is in *NP*. Let the certificate t consist of a path P and a sequence of transactions to be performed along P. Then the certifier B should check if performing the given transactions along the given path P achieves the target bound.

We shall now show *HSoDT!* is NP-complete by showing *3-SAT* $\leq_P$ *HSoDT!*. Consider a *3-SAT* instance with n variables and k clauses. Construct a layered graph $G = (V, E)$ with $n + k$ layers. The first n layers correspond to the n variables and their negations and the last k layers correspond to the clauses. More specifically, layer i of the first n layers consists of two nodes (not adjacent), one that sells droid types corresponding to variable x_i and the other sells droid types corresponding to variable $\overline{x}_i$. The supply of x_i and $\overline{x}_i$ is the total number of times each of them occurs in the k clauses. Also, let their prices be zero. For layer i of the last k layers, construct three nodes (not adjacent) corresponding to the variables or their negations in clause i. If x is a variable or its negation in clause i, then the corresponding node in layer i of the last k layers has a demand for one unit of droid type x with unit cost. Now for each of the first $n+k-1$ layers, construct directed edges from each of the nodes in layer i to each of the nodes in layer $i + 1$. Construct a starting node s with edges from s to each node in layer 1 and an ending node t with edges from each node in layer $n + k$ to t. Note that there are $2n$ droid types, $2 + 2n + 3k$ nodes including s and t. Now let the target bound be k. We claim that this bound can be reached on this instance of *HSoDT!* if and only if the given *3-SAT* instance has a solution.

Assume we have an *HSoDT!* solution. Note that for each of the layers, we have to pass through exactly one of the nodes. Layer i of the first n layers has two nodes, x_i and $\overline{x}_i$. If the solution passes through node x_i, then let variable x_i have a true assignment else let it have a false assignment. Since the target bound of k is reached, then one droid is sold at each of the last k layers which implies that each clause evaluates to true. Thus we have a *3-SAT* solution.

Now assume we have a *3-SAT* solution. Then we must have each clause evaluate to true, i.e. for each clause C_i, there must be some x_j or $\overline{x}_j$ in C_i such that the one in C_i evaluates to true. Now construct the path P such that for each of the first n layers we pass through node x_i if variable x_i has a true assignment else we pass through node $\overline{x}_i$. When passing through each node in the first n layers, take the available supply of droids. When passing through layer i of the last k layers, visit a node that causes clause i to evaluate to true and sell a unit of the corresponding droid. Since we sell a droid at each of the k layers, the target bound of k is achieved.

18. $(*)$ Suppose you're consulting for one of the many companies in New Jersey that designs communication networks, and they come to you with the following problem. They're studying a specific n-node communication network, modeled as a directed graph $G = (V, E)$. For reasons of fault-tolerance they want to divide up G into as many virtual "domains" as possible: a *domain* in G is a set X of nodes, of size at least 2, so that for each pair of nodes $u, v \in X$ there are directed paths from u to v and v to u that are contained entirely in X.

    Show that the following DOMAIN DECOMPOSITION problem is NP-complete. Given a directed graph $G = (V, E)$ and a number k, can V be *partitioned* into at least k sets, each of which is a domain?

    Solution. Given a proposed solution to domain decomposition, we can test each domain in turn to see whether every node has a path to every other node. (This can be done very efficiently by testing for strong connectivity.) Thus DOMAIN DECOMPOSITION is in NP.

    We now show that TRIPARTITE MATCHING $\leq_P$ DOMAIN DECOMPOSITION. To do this, we start with an instance of TRIPARTITE MATCHING, with sets X, Y, and Z of size n each, and a collection C of t ordered triples.

    We construct the following instance of DOMAIN DECOMPOSITION. We construct a graph $G = (V, E)$, where V consists of a node x'_i for each $x_i \in X$, y'_j for each $y_j \in Y$, and z'_k for each $z_k \in Z$. For each triple A_m in C, we will also define three nodes v^x_m, v^y_m, and v^z_m. Let U denote all nodes of the form x'_i, y'_j, or z'_k. We now define the following edges in G. For each triple of nodes v^x_m, v^y_m, and v^z_m, we construct a directed triangle via edges $(v^x_m, v^y_m), (v^y_m, v^z_m), (v^z_m, v^x_m)$. For each node x'_i, and each node v^x_m for which x_i appears in the triple A_m, we define edges (x'_i, v^x_m) and (v^x_m, x'_i). We do the analogous thing for each node y'_j and z'_k.

    So the idea is to create a directed triangle for each triple, and a pair of bi-directional edges between each element and each triple that it belongs to. We want to encode the existence of a perfect tripartite matching as follows. For each triple $A_m = (x_i, y_j, z_k)$ in the matching, we will construct three 2-element domains consisting of the nodes x'_i, y'_j, z'_k together with the nodes v^x_m, v^y_m, and v^z_m respectively. For each triple A_m that is *not* in the matching, we will simply construct the 3-element domain on v_m^x, v_m^y, and v_m^z.

Thus, we claim that G has a decomposition into at least $3n + t - n = 2n + t$ domains if and only there is a perfect tripartite matching in C. If there is a perfect tripartite matching, then the construction of the previous paragraph produces a partition of V into $2n + t$ domains. So let us prove the other direction; suppose there is a partition of V into $2n + t$ domains. Let p denote the number of domains containing elements from U. Note that $p \leq 3n$, and $p = 3n$ if and only if each element of U appears in a 2-element domain. Let q denote the number of domains not containing elements from U. Each such domain must consist of a single triangle; since at least n triangles are involved in domains with elements of U, we have $q \leq t - n$, and $q = t - n$ if and only if the domains involving U intersect only n triangles. Now, the total number of domains is $p + q$, and so this number is $2n + t$ if and only the domains consist of $t - n$ triangles, together with $3n$ two-element domains involving elements of U. In this case, the triangles that are *not* used in the domain decomposition correspond to triples in the Tripartite Matching instance that are all disjoint.

Thus, by deciding whether G has a decomposition into at least $2n + t$ domains, we can decide whether our original instance of Tripartite Matching has a solution.

19. You and a friend have been trekking through various far-off parts of the world, and have accumulated a big pile of souvenirs. At the time you weren't really thinking about which of these you were planning to keep, and which your friend was going to keep, but now the time has come to divide everything up.

    Here's a way you could go about doing this. Suppose there are n objects, labeled $1, 2, \ldots, n$, and object i has an agreed-upon *value* x_i. (We could think of this, for example, as a monetary re-sale value; the case in which you and your friend don't agree on the value is something we won't pursue here.) One reasonable way to divide things would be to look for a *partition* of the objects into two sets, so that the total value of the objects in each set is the same.

    This suggests solving the following *Number Partitioning* problem. You are given positive integers $x_1, \ldots, x_n$; you want to decide whether the numbers can be partitioned into two sets S_1 and S_2 with the same sum:

$$\sum_{x_i \in S_1} x_i = \sum_{x_j \in S_2} x_j.$$

    Show that *Number Partitioning* is NP-complete.

20. Consider the following problem. You are given positive integers $x_1, \ldots, x_n$, and numbers k and B. You want to know whether it is possible to *partition* the numbers $\{x_i\}$ into k sets $S_1, \ldots, S_k$ so that the squared sums of the sets add up to at most B:

$$\sum_{i=1}^{k} \left(\sum_{x_j \in S_i} x_j \right)^2 \leq B.$$

Show that this problem is NP-complete.

21. You are given a graph $G = (V, E)$ with weights w_e on its edges $e \in E$. The weights can be negative or positive. The ZERO-WEIGHT-CYCLE problem is to decide if there is a simple cycle in G so that the sum of the edge weights on this cycle is exactly 0. Prove that this problem is NP-complete.

22. Consider the following version of the Steiner tree problem, which we'll refer to as GRAPHICAL STEINER TREE. You are given an undirected graph $G = (V, E)$, a set $X \subseteq V$ of vertices, and a number k. You want to decide whether there is a set $F \subseteq E$ of at most k edges so that in the graph (V, F), X belongs to a single connected component.

    Show that GRAPHICAL STEINER TREE is NP-complete.

23. The *Directed Disjoint Paths* problem is defined as follows. We are given a directed graph G and k pairs of nodes $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$. The problem is to decide whether there exist node-disjoint paths $P_1, P_2, \ldots, P_k$ so that P_i goes from s_i to t_i.

    Show that *Directed Disjoint Paths* is NP-complete.

24. Given a directed graph G a *cycle cover* is a set of node-disjoint cycles so that each node of G belongs to a cycle. The *Cycle Cover* problem asks whether a given directed graph has a cycle cover.

    (a) Show that the cycle cover problem can be solved in polynomial time. (Hint: use bipartite matching.)

    (b)$(*)$ Suppose we require each cycle to have at most 3 edges. Show that determining whether a graph G has such a cycle cover is NP-complete.

25. Given two undirected graphs G and H, we say that H is a *subgraph* of G if we can obtain a copy of the graph H by starting from G and deleting some of the vertices and edges.

    The SUBGRAPH CONTAINMENT problem is defined as follows: given G and H, is H a subgraph of G? Show that SUBGRAPH CONTAINMENT is NP-complete.

26. Let $G = (V, E)$ be a graph with n nodes and m edges. If $E' \subset E$, we say that the *coherence* of E' is equal to n minus the number of connected components of the graph (V, E'). (So a graph with no edges has coherence 0, while a connected graph has coherence $n - 1$.)

    We are given subsets $E_1, E_2, \ldots, E_m$ of E. Our goal is to choose k of these sets — $E_{i_1}, \ldots, E_{i_k}$ — so that the *coherence* of the union $E_{i_1} \cup \cdots \cup E_{i_k}$ is as large as possible.

    Given this input, and a bound C, prove that it is NP-complete to decide whether it is possible to achieve a coherence of at least C.

27. Let $G = (V, E)$ be a bipartite graph; suppose its nodes are partitioned into sets X and Y so that each edge has one end in X and the other in Y. We define an *(a, b)-skeleton* of G to be a set of edges $E' \subseteq E$ so that *at most a* nodes in X are incident to an edge in E', and *at least b* nodes in Y are incident to an edge in E'.

    Show that, given a bipartite graph G and numbers a and b, it is NP-complete to decide whether G has an (a, b)-skeleton.

28. After a few too many days immersed in the popular entrepreneurial self-help book *Mine Your Own Business*, you've come to the realization that you need to upgrade your office computing system. This, however, leads to some tricky problems . . .

    In configuring your new system, there are k *components* that must be selected: the operating system, the text editing software, the e-mail program, and so forth; each is a separate component. For the j^{th} component of the system, you have a set A_j of options; and a *configuration* of the system consists of a selection of one element from each of the sets of options $A_1, A_2, \ldots, A_k$.

    Now, the trouble arises because certain pairs of options from different sets may not be compatible: we say that option $x_i \in A_i$ and option $x_j \in A_j$ form an *incompatible pair* if a single system cannot contain them both. (For example, Linux (as an option for the operating system) and Word (as an option for the text-editing software) form an incompatible pair.) We say that a configuration of the system is *fully compatible* if it consists of elements $x_1 \in A_1, x_2 \in A_2, \ldots x_k \in A_k$ such that none of the pairs (x_i, x_j) is an incompatible pair.

    We can now define the *Fully Compatible Configuration* (FCC) problem. An instance of FCC consists of the sets of options $A_1, A_2, \ldots, A_k$, and a set P of *incompatible pairs* (x, y), where x and y are elements of different sets of options. The problem is to decide whether there exists a fully compatible configuration: a selection of an element from each option set so that no pair of selected elements belongs to the set P.

    Example. Suppose $k = 3$, and the sets A_1, A_2, A_3 denote options for the operating system, the text editing software, and the e-mail program respectively. We have

    $$A_1 = \{\texttt{Linux}, \texttt{Windows NT}\},$$

    $$A_2 = \{\texttt{emacs}, \texttt{Word}\},$$

    $$A_3 = \{\texttt{Outlook}, \texttt{Eudora}, \texttt{rmail}\},$$

    with the set of incompatible pairs equal to

    $$P = \{(\texttt{Linux}, \texttt{Word}), (\texttt{Linux}, \texttt{Outlook}), (\texttt{Word}, \texttt{rmail})\}.$$

    Then the answer to the decision problem in this instance of FCC is "yes" — for example, the choices $\texttt{Linux} \in A_1, \texttt{emacs} \in A_2, \texttt{rmail} \in A_3$ is a fully compatible configuration according to the above definitions.

    Prove that *Fully Compatible Configuration* is NP-complete.

29. For functions $g_1, \ldots, g_\ell$, we define the function $\max(g_1, \ldots, g_\ell)$ via

$$[\max(g_1, \ldots, g_\ell)](x) = \max(g_1(x), \ldots, g_\ell(x)).$$

Consider the following problem. You are given n piecewise linear, continuous functions $f_1, \ldots, f_n$ defined over the interval $[0, t]$ for some integer t. You are also given an integer B. You want to decide: do there exist k of the functions $f_{i_1}, \ldots, f_{i_k}$ so that

$$\int_0^t [\max(f_{i_1}, \ldots, f_{i_k})](x)\; dx \geq B?$$

Prove that this problem is NP-complete.

Solution. First, we claim the problem is in NP. For consider any set of k of the functions $f_{i_1}, \ldots, f_{i_k}$. If q is the maximum number of "break-points" in the piecewise linear representation of any one of them, then $F = \max(f_{i_1}, \ldots, f_{i_k})$ has at most k^2q^2 break-points. Between each pair of break-points, we can compute the area under F by computing the area of a single trapezoid; thus we can compute the integral of F in polynomial time to verify a purported solution.

We now show how the Vertex Cover problem could be solved using an algorithm for this problem. Given an instance of Vertex Cover with graph $G = (V, E)$ and bound k, we write $V = \{1, 2, \ldots, n\}$ and $E = \{e_0, \ldots, e_{m-1}\}$. We construct a function f_i for each vertex i as follows. First, let $t = 2m - 1$, so each f_i will be defined over $[0, 2m-1]$. If e_j is incident on i, we define $f_i(x) = 1$ for $x \in [2j, 2j + 1]$; if e_j is not incident on i, we define $f_i(x) = 0$ for $x \in [2j, 2j + 1]$. We also define $f_i(x) = \frac{1}{2}$ for each x of the form $2j + \frac{3}{2}$. Finally, to define $f_i(x)$ for $x \in [2j+1, 2j+2]$ for an integer $j \in \{0, \ldots, m-2\}$, we simply connect $f_i(2j + 1)$ to $f_i(2j + \frac{3}{2})$ to $f_i(2j + 2)$ by straight lines.

Now, if there is a vertex cover of size k, then the pointwise maximum of these k functions has covers an area of 1 on each interval of the form $[2j, 2j + 1]$ and an area of $\frac{3}{4}$ on each interval of the form $[2j + 1, 2j + 2]$, for a total area of $B = m + \frac{3}{4}(m - 1)$. Conversely, any k functions that cover this much area must cover an area of 1 on each interval of the form $[2j, 2j + 1]$, and so the corresponding nodes constitute a vertex cover of size k.

30. Consider the following *Broadcast Time* problem. We are given a directed graph $G = (V, E)$, with a designated node $r \in V$ and a designated set of "target nodes" $T \subseteq V - \{r\}$. Each node v has a *switching time* s_v, which is a positive integer.

At time 0, the node r generates a message that it would like every node in T to receive. To accomplish this, we want to find a scheme whereby r tells some of its neighbors (in sequence), who in turn tell some of their neighbors, and so on, until every node in T has received the message. More formally, a *broadcast scheme* is defined as follows. Node r may send a copy of the message to one of its neighbors at time 0; this neighbor will receive the message at time 1. In general, at time $t \geq 0$, any node v that has already received the message may send a copy of the message to one of its neighbors, provided it has not sent a copy of the message in any of the time steps $t-s_v+1, t-s_v+2, \dots, t-1$. (This reflects the role of the *switching time*; v needs a pause of $s_v - 1$ steps between successive sendings of the message. Note that if $s_v = 1$, then no restriction is imposed by this.)

The *completion time* of the broadcast scheme is the minimum time t by which all nodes in T have received the message. The *Broadcast Time* problem is the following: given the input described above, and a bound b, is there a broadcast scheme with completion time at most b?

Prove that *Broadcast Time* is NP-complete.

Example. Suppose we have a directed graph $G = (V, E)$, with $V = \{r, a, b, c\}$; edges (r, a), (a, b), (r, c); the set $T = \{b, c\}$; and switching time $s_v = 2$ for each $v \in V$. Then a broadcast scheme with minimum completion time would be as follows: r sends the message to a at time 0; a sends the message to b at time 1; r sends the message to c at time 2; and the scheme completes at time 3 when c receives the message. (Note that a can send the message as soon as it receives it at time 1, since this is its first sending of the message; but r cannot send the message at time 1 since $s_r = 2$ and it sent the message at time 0.)

Solution. The problem is in NP since, given a schedule for message transmissions, one can verify that it obeys the switching time constraints, and that all nodes in the target set receive the message by the desired time.

We now show how to reduce 3-*Dimensional Matching* to *Broadcast Time*. Consider an instance of 3-*Dimensional Matching*, with triples from $X \times Y \times Z$, where $n = |X| = |Y| = |Z|$. We construct a graph G with a root r, a node v_t for each triple t, and a node w_a for each element $a \in X \cup Y \cup Z$. There are directed edges (r, v_t) for each t, and (v_t, w_a) when a belongs to triple t. The target set T is $\{w_a : a \in X \cup Y \cup Z\}$. The root r has switching time 4, and all other nodes have switching time 1.

We claim that there exists a set of n triples covering $X \cup Y \cup Z$ if and only if there is a broadcast scheme with completion time at most $4n - 1$. Indeed, if there are n such triples, then r sends messages to each corresponding node v_t at times $0, 4, 8, \dots, 4n-4$; and if node v_t receives a message from r at time $4j$, it sends messages to its three elements at times $4j+1, 4j+2, 4j+3$. Conversely, if there is a broadcast scheme with completion time at most $4n - 1$, then r must have sent at most n messages, since its switching time is 4. Now, if R is the set of nodes that received messages from r, then $M = \{t : v_t \in R\}$ must cover $X \cup Y \cup Z$, since every node w_a has received the message. Thus M forms a solution to the instance of 3-*Dimensional Matching*.

31. Suppose that someone gives you a black-box algorithm $\mathcal{A}$ that takes an undirected graph $G = (V, E)$, and a number k, and behaves as follows.

    - If G is not connected, it simply returns "G is not connected."

- If G is connected and has an independent set of size at least k, it returns "yes."
- If G is connected and does not have an independent set of size at least k, it returns "no."

Suppose that the algorithm $\mathcal{A}$ runs in time polynomial in the size of G and k.

Show how, using calls to $\mathcal{A}$, you could then solve the INDEPEDENT SET problem in polynomial time: Given an arbitrary undirected graph G, and a number k, does G contain an independent set of size at least k?

Solution. There are two basic ways to do this. Let $G = (V, E)$; we can give the answer without using $\mathcal{A}$ if $V = \phi$ or $k = 1$, and so we will suppose $V \neq \phi$ and $k > 1$.

The first approach is to add an extra node v^* to G, and join it to each node in V; let the resulting graph be G^*. We ask $\mathcal{A}$ whether G^* has an independent set of size at least k, and return this answer. (Note that the answer will be yes or no, since G^* is connected.) Clearly if G has an independent set of size at least k, so does G^*. But if G^* has an independent set of size at least k, then since $k > 1$, v^* will not be in this set, and so it is also an independent set in G. Thus (since we're in the case $k > 1$), G has an independent set of size at least k if and only if G^* does, and so our answer is correct. Moreover, it takes polynomial time to build G^* and ask call $\mathcal{A}$ once.

Another approach is to identify the connected components of G, using breadth-first search in time $O(|E|)$. In each connected component C, we call $\mathcal{A}$ with values $k = 1, \ldots, n$; we let k_C denote the largest value on which $\mathcal{A}$ says "yes." Thus k_C is the size of the maximum independent set in the component C. Doing this takes polynomial time per component, and hence polynomial time overall. Since nodes in different components have no edges between them, we now know that the largest independent set in G has size $\sum_C k_C$; thus, we simply compare this quantity to k.

32. Given a set of finite binary strings $S = \{s_1, \ldots, s_k\}$, we say that a string u is a *concatenation* over S if it is equal to $s_{i_1} s_{i_2} \cdots s_{i_t}$ for some indices $i_1, \ldots, i_t \in \{1, \ldots, k\}$.

    A friend of yours is considering the following problem: Given two sets of finite binary strings, $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$, does there exist any string u so that u is both a concatenation over A and a concatenation over B?

    You friend announces, "At least the problem is in NP, since I would just have to exhibit such a string u in order to prove the answer is 'yes.' " You point out — politely, of course — that this is a completely inadequate explanation; how do we know that the shortest such string u doesn't have length exponential in the size of the input, in which case it would not be a polynomial-size certificate?

    However, it turns out that this claim can be turned into a proof of membership in NP. Specifically, prove the following statement:

    > If there is a string u that is a concatenation over both A and B, then there is such a string whose length is bounded by a polynomial in the sum of the lengths of the strings in $A \cup B$.

Solution. Suppose $m \le n$, and let L denote the maximum length of any string in $A \cup B$. Suppose there is a string that is a concatenation over both A and B, and let u be one of minimum length. We claim that the length of u is at most n^2L^2.

For suppose not. First, we say that position p in u is of *type* (a_i, k) if in the concatenation over A, it is represented by position k of string a_i. We define *type* (b_i, k) analogously. Now, if the length of u is greater than n^2L^2, then by the pigeonhole principle, there exist positions p and p' in u, $p < p'$, so that both are of type (a_i, k) and (b_j, k) for some indices i, j, k. But in this case, the string u' obtained by deleting positions $p, p+1, \ldots, p'-1$ would also be a concatenation over both A and B. As u' is shorter than u, this is a contradiction.

33. Your friends at WebExodus have recently been doing some consulting work for companies that maintain large, publicly accessible Web sites — contractual issues prevent them from saying which ones — and they've come across the following Strategic Advertising problem.

A company comes to them with the map of a Web site, which we'll model as a directed graph $G = (V, E)$. The company also provides a set of t *trails* typically followed by users of the site; we'll model these trails as directed paths $P_1, P_2, \ldots, P_t$ in the graph G. (I.e. each P_i is a path in G.)

The company wants WebExodus to answer the following question for them: Given G, the paths $\{P_i\}$, and a number k, is it possible to place advertisements on at most k of the nodes in G, so that each path P_i includes at least one node containing an advertisement? We'll call this the Strategic Advertising problem, with input G, $\{P_i : i = 1, \ldots, t\}$, and k.

Your friends figure that a good algorithm for this will make them all rich; unfortunately, things are never quite this simple …

(a) Prove that Strategic Advertising is NP-complete.

(b) Your friends at WebExodus forge ahead and write a pretty fast algorithm $\mathcal{S}$ that produces yes/no answers to arbitrary instances of the Strategic Advertising problem. You may assume that the algorithm $\mathcal{S}$ is always correct.

Using the algorithm $\mathcal{S}$ as a black box, design an algorithm that takes input G, $\{P_i\}$, and k as in part (a), and does one of the following two things:

- Outputs a set of at most k nodes in G so that each path P_i includes at least one of these nodes, *or*
- Outputs (correctly) that no such set of at most k nodes exists.

Your algorithm should use at most a polynomial number of steps, together with at most a polynomial number of calls to the algorithm $\mathcal{S}$.

Solution. **(a)** We'll say a set of advertisements is "valid" if it covers all paths in $\{P_i\}$. First, Strategic Advertising (SA) is in NP: Given a set of k nodes, we can check in $O(kn)$ time (or better) whether at least one of them lies on a path P_i, and so we can check whether it is a valid set of advertisements in time $O(knt)$.

We now show that Vertex Cover $\leq_P$ SA. Given an undirected graph $G = (V, E)$ and a number k, produce a directed graph $G' = (V, E')$ by arbitrarily directing each edge of G. Define a path P_i for each edge in E'. This construction involves one pass over the edges, and so takes polynomial time to compute. We now claim that G' has a valid set of at most k advertisements if and only if G has a vertex cover of size at most k. For suppose G' does have such a valid set U; since it meets at least one end of each edge, it is a vertex cover for G. Conversely, suppose G has a vertex cover T of size at most k; then, this set T meets each path in $\{P_i\}$ and so it is a valid set of advertisements.

(b) We construct the algorithm by induction on k. If $k = 1$, we simply check whether there is any node that lies on all paths. Otherwise, we ask the fast algorithm $\mathcal{S}$ whether there is a valid set of advertisements of size at most k. If it says "no," we simply report this. If it says "yes", we perform the following test for each node v: we delete v and all paths through it, and ask $\mathcal{S}$ whether, on this new input, there is a valid set of advertisements of size at most $k-1$. We claim that there is at least one node v where this test will succeed. For consider any valid set U of at most k advertisements (we know one exists since $\mathcal{S}$ said "yes"): The test will succeed on any $v \in U$, since $U - \{v\}$ is a valid set of at most $k-1$ advertisements on the new input.

Once we identify such a node, we add it to a set T that we maintain. We are now dealing with an input that has a valid set of at most $k-1$ advertisements, and so our algorithm will finish the construction of T correctly by induction. The running time of the algorithm involves $O(n+t)$ operations and calls to $\mathcal{S}$ for each fixed value of k, for a total of $O(n^2 + nt)$ operations.

34. Consider the following problem. You are managing a communication network, modeled by a directed graph $G = (V, E)$. There are c *users* who are interested in making use of this network. User i (for each $i = 1, 2, \ldots, c$) issues a *request* to reserve a specific path P_i in G on which to transmit data.

You are interested in accepting as many of these path requests as possible, subject to the following restriction: if you accept both P_i and P_j, then P_i and P_j cannot share any nodes.

Thus, the *Path Selection* problem asks: given a directed graph $G = (V, E)$, a set of requests $P_1, P_2, \ldots, P_c$ — each of which must be a path in G — and a number k, is it possible to select at least k of the paths so that no two of the selected paths share any nodes?

Prove that *Path Selection* is NP-complete.

Solution. *Path Selection* is in NP, since we can be shown a set of k paths from among $P_1, \ldots, P_c$ and check in polynomial time that no two of them share any nodes.

Now, we claim that *3-Dimensional Matching* $\leq_P$ *Path Selection.* For consider an instance of *3-Dimensional Matching* with sets X, Y, and Z, each of size n, and ordered triples $T_1, \ldots, T_m$ from $X \times Y \times Z$. We construct a directed graph $G = (V, E)$ on the node set $X \cup Y \cup Z$. For each triple $T_i = (x_i, y_j, z_k)$, we add edges (x_i, y_j) and (y_j, z_k) to G. Finally, for each $i = 1, 2, \ldots, m$, we define a path P_i that passes through the nodes $\{x_i, y_j, z_k\}$, where again $T_i = (x_i, y_j, z_k)$. Note that by our definition of the edges, each P_i is a valid path in G. Also, the reduction takes polynomial time.

Now we claim that there are n paths among $P_1, \ldots, P_m$ sharing no nodes if and only if there exist n disjoint triples among $T_1, \ldots, T_m$. For if there do exist n paths sharing no nodes, then the corresponding triples must each contain a different element from X, a different element from Y, and a different element from Z — they form a perfect three-dimensional matching. Conversely, if there exist n disjoint triples, then the corresponding paths will have no nodes in common.

Since *Path Selection* is in NP, and we can reduce an NP-complete problem to it, it must be NP-complete.

(Other direct reductions are from Set Packing and from Independent Set.)

35. A store trying to analyze the behavior of its customers will often maintain a two-dimensional array A, where the rows correspond to its customers and the columns correspond to the products it sells. The entry $A[i, j]$ specifies the quantity of product j that has been purchased by customer i.

Here's a tiny example of such an array A.

	liquid detergent	beer	diapers	cat litter
Raj	0	6	0	3
Alanis	2	3	0	0
Chelsea	0	0	0	7

One thing that a store might want to do with this data is the following. Let us say that a subset S of the customers is *diverse* if no two of the of the customers in S have ever bought the same product. (I.e. for each product, at most one of the customers in S has ever bought it.) A diverse set of customers can be useful, for example, as a target pool for market research.

We can now define the DIVERSE SUBSET problem as following: given an $m \times n$ array A as defined above, and a number $k \leq m$, is there a subset of at least k of customers that is *diverse*?

Show that DIVERSE SUBSET is NP-complete.

36. A *combinatorial auction* is a particular mechanism developed by economists for selling a collection of items to a collection of potential buyers. (The Federal Communications Commission has studied this type of auction for assigning stations on the radio spectrum to broadcasting companies.)

    Here's a simple type of combinatorial auction. The are n items for sale, labeled $I_1, \ldots, I_n$. Each item is indivisible, and can only be sold to one person. Now, m different people place *bids*: the i^{th} bid specifies a subset S_i of the items, and an *offering price* x_i that the offeror of this bid is willing to pay for the items in the set S_i, as a single unit. (We'll represent this bid as the pair (S_i, x_i).)

    An auctioneer now looks at the set of all m bids; she chooses to *accept* some of these bids, and *reject* the others. The offeror of an accepted bid gets to take all the items in S_i. Thus, the rule is that no two accepted bids can specify sets that contain a common item, since this would involve giving the same item to two different people.

    The auctioneer collects the sum of the offering prices of all accepted bids. (Note that this is a "one-shot" auction; there is no opportunity to place further bids.) The auctioneer's goal is to collect as much money as possible.

    Thus, the *Combinatorial Auction* problem asks: Given items $I_1, \ldots, I_n$, bids $(S_1, x_1), \ldots, (S_m, x_m)$, and a bound B, is there a collection of bids that the auctioneer can accept so as to collect an amount of money that is at least B?

    Example. Suppose an auctioneer decides to use this method to sell some excess computer equipment. There are four items labeled "`PC`," "`monitor`," "`printer`", and "`modem`"; and three people place bids. Define

    $$S_1 = \{\texttt{PC}, \texttt{monitor}\}, S_2 = \{\texttt{PC}, \texttt{printer}\}, S_3 = \{\texttt{monitor}, \texttt{printer}, \texttt{modem}\}$$

    and

    $$x_1 = x_2 = x_3 = 1.$$

    The bids are $(S_1, x_1), (S_2, x_2), (S_3, x_3)$ and the bound B is equal to 2.

    Then the answer to this instance is "no": The auctioneer can accept at most one of the bids (since any two bids have a desired item in common), and this results in a total monetary value of only 1.

    Prove that *Combinatorial Auction* is NP-complete.

8 PSPACE

1. *(Based on a problem proposed by Maverick Woo and Ryan Williams.)* Let's consider a special case of *Quantified 3-SAT* in which the underlying Boolean formula is *monotone*

(in the sense of Problem Set 5). Specifically, let $\Phi(x_1, \ldots, x_n)$ be a Boolean formula of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

where each C_i is a disjunction of three terms. We say Φ is *monotone* if each term in each clause consists of a non-negated variable — i.e. each term is equal to x_i, for some i, rather than $\overline{x_i}$.

We define *Monotone QSAT* to be the decision problem

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)?$$

where the formula Φ is monotone.

Do one of the following two things: (a) prove that *Monotone QSAT* is PSPACE-complete; or (b) give an algorithm to solve arbitrary instances of *Monotone QSAT* that runs in time polynomial in n. (Note that in (b), the goal is polynomial *time*, not just polynomial space.)

Solution. This problem can be decided in polynomial time. It helps to view the *QSAT* instance as a *CSAT* instance instead: Player 1 controls the set A of odd-indexed variables while Player 2 controls the set B of even-indexed variables. Our question then becomes: can Player 1 force a win?

We claim that Player 1 can force a win if and only if each clause C_i contains a variable from A. If this is the case, Player 1 can win by setting all variables in A to 1. If this is not the case, then some clause C_i has no variable from A. Player 2 can then win by setting all variables in B to 0: in particular, this will cause the clause C_i to evaluate to 0.

2. *Self-avoiding walks* are a basic object of study in the area of statistical physics; they can be defined as follows. Let $\mathcal{L}$ denote the set of all points in $\mathbf{R}^2$ with integer coordinates. A *self-avoiding walk* W of length n is a sequence of points $(p_1, p_2, \ldots, p_n) \in \mathcal{L}^n$ so that

   (i) $p_1 = (0, 0)$. *(The walk starts the origin.)*

   (ii) No two of the points are equal. *(The walk "avoids" itself.)*

   (iii) For each $i = 1, 2, \ldots, n-1$, the points p_i and p_{i+1} are at distance 1 from each other. *(The walk moves between neighboring points in $\mathcal{L}$.)*

   Self-avoiding walks (in both two and three dimensions) are used in physical chemistry as a simple geometric model for the possible conformations of long-chain polymer molecules. Such molecules can be viewed as a flexible chain of beads that flop around in solution, adopting different geometric layouts, and self-avoiding walks are a simple combinatorial abstraction for these layouts.

   A famous unsolved problem in this area is the following. For a natural number $n \geq 1$, let $A(n)$ denote the number of distinct self-avoiding walks of length n. Note that we view walks as *sequences* of points rather than sets; so two walks can be distinct even if they pass through the same set of points, provided that they do so in different orders. (Formally, the walks $(p_1, p_2, \ldots, p_n)$ and $(q_1, q_2, \ldots, q_n)$ are distinct if there is some i $(1 \leq i \leq n)$ for which $p_i \neq q_i$.) See the figure below for an example. In polymer models based on self-avoiding walks, $A(n)$ is directly related to the *entropy* of a chain molecule, and so it appears in theories concering the rates of certain metabolic and organic synthesis reactions.

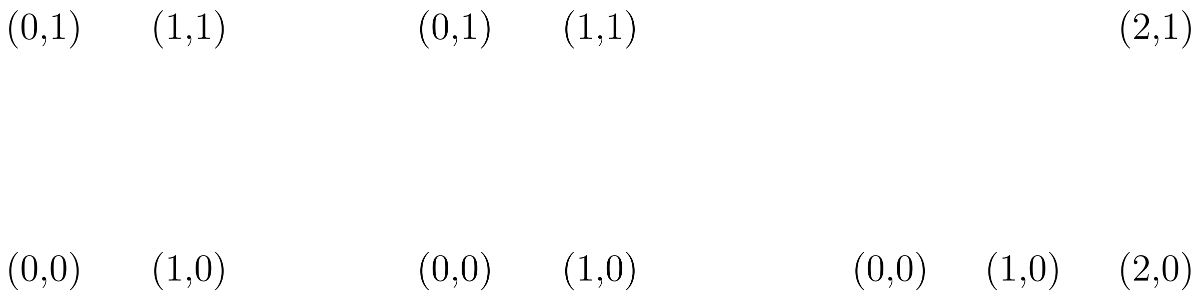


Figure 7: Three distinct self-avoiding walks of length 4. Note that although walks (a) and (b) involve the same set of points, they are considered different walks because they pass through them in a different order.

Despite its importance, no simple formula is known for the value $A(n)$. Indeed, no algorithm is known for computing $A(n)$ that runs in time polynomial in n.

(a) Show that $A(n) \geq 2^{n-1}$ for all natural numbers $n \geq 1$.

(b) Give an algorithm that takes a number n as input, and outputs $A(n)$ as a number in binary notation, using space (i.e. memory) that is polynomial in n.

(Thus the running time of your algorithm can be exponential, as long as its space usage is polynomial. Note also that "polynomial" here means "polynomial in n," not "polynomial in $\log n$". Indeed, by part (a), we see that it will take at least $n - 1$ bits to write the value of $A(n)$, so clearly $n - 1$ is a lower bound on the amount of space you need for producing a correct answer.)

Solution. Each step from p_i to p_{i+1} in a self-avoiding walk can be viewed as moving in one of four directions: north, south, east, or west. Thus, any self-avoiding walk can be mapped to a distinct string of length $n - 1$ over the alphabet $\{N, S, E, W\}$. (The three walks in the figure would be ENW, NES, and EEN.) However, not all such strings correspond to walks that are self-avoiding: for example, the walk NESW re-visits the point $(0, 0)$.

(a) Observe that all walks that can be encoded using only the letters $\{N, E\}$ are self-avoiding. As there are 2^{n-1} strings of length $n - 1$ over these two letters, the lower bound follows.

Note that our encoding technique also shows that $A(n) \leq 4^{n-1}$.

(b) Using a counter in base 4, we can enumerate all strings of length $n-1$ over the alphabet $\{N, S, E, W\}$. For each one, we construct the corresponding walk and test, in polynomial space, whether it is self-avoiding. Finally, we increment a second counter A (initialized to 0) if the current walk is self-avoiding.

At the end of this algorithm, A will hold the value of $A(n)$. Since A can be incremented at most 4^{n-1} times, we need at most $2n$ bits for A.

9 Extending the Limits of Tractability

1. Earlier, we considered an exercise in which we claimed that the *Hitting Set* problem was NP-complete. To recap the definitions, Consider a set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A. We say that a set $H \subseteq A$ is a *hitting set* for the collection $B_1, B_2, \ldots, B_m$ if H contains at least one element from each B_i — that is, if $H \cap B_i$ is not empty for each i. (So H "hits" all the sets B_i.)

   Now, suppose we are given an instance of this problem, and we'd like to determine whether there is a hitting set for the collection of size at most k. Furthermore, suppose that each set B_i has at most c elements, for a constant c. Give an algorithm that solves this problem with a running time of the form $O(f(c,k) \cdot p(n,m))$, where $p(\cdot)$ is a polynomial function, and $f(\cdot)$ is an arbitrary function that depends only on c and k, not on n or m.

   Solution. Our solution will be similar to the algorithm for VERTEX COVER presented in class. Consider the notation defined in the problem. For an element $a \in A$, we *reduce the instance by a* by deleting a from A, and deleting all sets B_i that contain a. Thus, reducing the instance by a producing a new, presumably smaller, instance of HITTING SET.

   We observe the following fact. Let $B_i = \{x_1, \ldots, x_c\} \subseteq A$ be any of the given sets in the HITTING SET instance. Then at least one of $x_1, \ldots, x_c$ must belong to any hitting set H. So by analogy with (2.3) from the notes, we have the following fact

   - Let $B_i = \{x_1, \ldots, x_c\}$ There is k-element hitting set for the original instance if and only if, for some $i = 1, \ldots, c$, the instance reduced by x_i has a $(k-1)$-element hitting set.

   The proof is completely analogous to that of (2.3). If H is a k-element hitting set, then some $x_i \in H$, and so $H - \{x_i\}$ is a $(k-1)$-element hitting set for the instance reduced by x_i. Conversely, if the instance reduced by x_i has a $(k-1)$-element hitting set H', then $H' \cup \{x_i\}$ is a k-element hitting set for the original instance.

   Thus, our algorithm is as follows. We pick any set $B_i = \{x_1, \ldots, x_c\}$. For each x_i, we recursively test if the instance reduced by x_i has a $(k-1)$-element hitting set. We return "yes" if and only if the answer to one of these recursive calls is "yes." Our running time satisfies $T(m,k) \leq cT(m,k-1) + O(cm)$, and so it satisfies $T(m,k) = O(c^k \cdot kcm)$. This gives the desired bound, with $f(c,k) = kc^{k+1}$ and $p(m) = m$.

2. Consider a network of workstations modeled as an undirected graph G, where each node is a workstation, and the edges represent direct communication links. We'd like to place copies of a database at nodes in G, so that each node is close to at least one copy.

   Specifically, assume that each node v in G has a cost c_v charged for placing a copy of the database at node v. The Min-Cost Server Placement problem is as follows. Given the network G, and costs $\{c_v\}$, find a set of nodes $S \subseteq V$ of minimum total cost $\sum_{v \in S} c_v$, so that if we place copies of a database at each node in S, then every workstation either has a copy of the database, or is connected by a direct link to a workstation that has a copy of the database.

   Give a polynomial time algorithm for the special case of the Min-Cost Server Placement where the graph G is a tree.

   Note the difference between Server Placement and Vertex Cover. If the graph G is a path of consisting of 6 nodes, then Vertex Cover needs to select at least 3 of the 6 nodes, while the second and the 5th node form a valid solution of the Min-Cost Server Placement problem, requiring only two nodes.

3. Suppose we are given a directed graph $G = (V, E)$, with $V = \{v_1, v_2, \ldots, v_n\}$, and we want to decide whether G has a Hamiltonian path from v_1 to v_n. (That is, is there a path in G that goes from v_1 to v_n, passing through every other vertex exactly once?)

   Since the Hamiltonian path problem is NP-complete, we do not expect that there is a polynomial-time solution for this problem. However, this does not mean that all non-polynomial-time algorithms are equally "bad." For example, here's the simplest brute-force approach: for each permutation of the vertices, see if it forms a Hamiltonian path from v_1 to v_n. This takes time roughly proportional to $n!$, which is about 3×10^{17} when $n = 20$.

   Show that the Hamiltonian path problem can in fact be solved in time $O(2^n \cdot p(n))$, where $p(n)$ is a polynomial function of n. This is a much better algorithm for moderate values of n; 2^n is only about a million when $n = 20$.

   Solution. Consider an ordered triple (S, i, j), $1 \leq i, j \leq n$ and S is a subset of the vertices that includes v_i and v_j. Let $B[S, i, j]$ denote the answer to the question, "Is there a Hamiltonian path on $G[S]$ that starts at v_i and ends at v_j?" Clearly, we are looking for the answer to $B[V, 1, n]$.

   We now show how to construct the answers to all $B[S, i, j]$, starting from the smallest sets and working up to larger ones, spending $O(n)$ time on each. Thus the total running time will be $O(2^n \cdot n^3)$.

   $B[S, i, j]$ is true if and only if there is some vertex $v_k \in S - \{v_i\}$ so that (v_i, v_k) is an edge, and there is a Hamiltonian path from v_k to v_j in $G[S - \{v_i\}]$. Thus, we set

$B[S, i, j]$ to be true if and only if there is some $v_k \in S - \{v_i\}$ for which $(v_i, v_k) \in E$ and $B[S - \{v_i\}, k, j]$ is true. This takes $O(n)$ time to determine.

4. $(*)$ Give a polynomial time algorithm for the following problem. We are given a binary tree $T = (V, E)$ with an even number of nodes, and a non-negative cost on each edge. Find a partition of the nodes V into two sets of *equal* size so that the weight of the cut between the two sets is as large as possible. (I.e. the total weight of edges with one end in each set is as large as possible.) Note that the restriction that the graph is a tree crucial here, but the assumption that the tree is binary is not. The problem is NP-hard in general graphs.

5. We say that a graph $G = (V, E)$ is a *triangulated cycle graph* if it consists of the vertices and edges of a triangulated convex n-gon in the plane — in other words, if it can be drawn in the plane as follows.

   The vertices are all placed on the boundary of a convex set in the plane (we may assume on the boundary of a circle), with each pair of consecutive vertices on the circle joined by an edge. The remaining edges are then drawn as straight line segments through the interior of the circle, with no pair of edges crossing in the interior. If we let S denote the set of all points in the plane that lie on vertices or edges of the drawing, then each bounded component of $\mathbf{R}^2 - S$ is bordered by exactly three edges. (This is the sense in which the graph is a "triangulation.")

   A triangulated cycle graph is pictured below.

   Prove that every triangulated cycle graph has a tree decomposition of width at most 2, and describe an efficient algorithm to construct such a decomposition.

   Solution. We claim that such a graph G has a tree decomposition $(T, \{V_t\})$ in which each piece V_t corresponds uniquely to an internal triangular face of G. We prove this by induction on the number of nodes in G.

   Choose any internal edge $e = (u, v)$ of G; deleting u and v produces two components A and B. Let G_1 be the subgraph induced on $A \cup \{u, v\}$ and G_2 the subgraph induced on $B \cup \{u, v\}$. By induction, there are tree decompositions $(T_1, \{X_t\})$ and $(T_2, \{Y_t\}$ of G_1 and G_2 respectively in which the pieces correspond uniquely to internal faces.

Thus there are nodes $t_1 \in T_1$ and $t_2 \in T_2$ that correspond to the faces containing the edge (u, v). If we let T denote the tree obtained by adding an edge (t_1, t_2) to $T_1 \cup T_2$, then $(T, \{X_t\} \cup \{Y_t\})$ is a tree decomposition having the desired properties.

6. The *Minimum-cost Dominating Set Problem* is specified by an undirected graph $G = (V, E)$ and costs $c(v)$ on the nodes $v \in V$. A subset $S \subset V$ is said to be a *dominating set* if all nodes $u \in V - S$ have an edge (u, v) to a node v in S. (Note the difference between dominating sets and vertex covers: in a dominating set, it is fine to have an edge (u, v) with neither u nor v in the set S as long as both u and v have neighbors in S.)

   (a.) Give a polynomial time algorithm for the Dominating Set problem for the special case in which G is a tree.

   (b.) Give a polynomial time algorithm for the Dominating Set problem for the special case in which G has tree-width 2, and we are also given a tree-decomposition of G with width 2.

7. The *Node-Disjoint Paths Problem* is given by an undirected graph G and k pairs of nodes (s_i, t_i) for $i = 1, \ldots, k$. The problem is to decide whether there are a node-disjoint paths P_i so that path P_i connects s_i to t_i. Give a polynomial time algorithm for the Node-Disjoint Paths Problem for the special case in which G has tree-width 2, and we are also given a tree-decomposition T of G with width 2.

8. A *k-coloring* of an undirected graph $G = (V, E)$ is an assignment of one of the numbers $\{1, 2, \ldots, k\}$ to each node, so that if two nodes are joined by an edge, then they are assigned different numbers. The *chromatic number* of G is the minimum k such that it has a k-coloring. For $k \geq 3$, it is NP-complete to decide whether a given input graph has chromatic number $\leq k$. (You don't have to prove this.)

   (a) Show that for every natural number $w \geq 1$, there is a number $k(w)$ so that the following holds. If G is a graph of tree-width at most w, then G has chromatic number at most $k(w)$. (The point is that $k(w)$ depends only on w, not on the number of nodes in G.)

   (b) Given an undirected n-node graph $G = (V, E)$ of tree-width at most w, show how to compute the chromatic number of G in time $O(f(w) \cdot p(n))$, where $p(\cdot)$ is a polynomial but $f(\cdot)$ can be an arbitrary function.

   Solution. Checking whether G is 1- or 2-colorable is easy. For $k = 3, 4, \ldots, w + 1$, we test whether G is k-colorable by dynamic programming. We use notation similar to what we used in the Maximum-Weight Independent Set problem for graphs of bounded tree-width. Let $(T, \{V_t : t \in T\})$ be a tree decomposition of G. For the subtree rooted at t, and every coloring χ of V_t using the color set $\{1, 2, \ldots, k\}$, we have a predicate $q_t(\chi)$ that says whether there is a k-coloring of G_t that is equal to χ when restricted to V_t. This requires us to maintain $k^{(w+1)} \leq (w+1)^{(w+1)}$ values for each piece of the tree decomposition.

We compute the values $q_t(\chi)$ when t is a leaf by simply trying all possible colorings of G_t. In general, suppose t has children $t_1, \ldots, t_d$, and we know the values of $q_{t_i}(\chi)$ for each choice of t_i and χ. Then there is a coloring of G_t consistent with χ on V_t if and only if there are colorings of the subgraphs $G_{t_1}, \ldots, G_{t_d}$ that are consistent with χ on the parts of V_{t_i} that intersect with V_t. Thus we set $q_t(\chi)$ equal to *true* if and only if there are colorings χ_i of V_{t_i} such that $q_{t_i}(\chi_i) = \mathit{true}$ and χ_i is the same as χ when restricted to $V_t \cap V_{t_i}$.

10 Approximation Algorithms

1. Consider the following *greedy algorithm* for finding a matching in a bipartite graph:

   As long as there is an edge whose endpoints are unmatched, add it to the current matching.

   (a) Give an example of a bipartite graph G for which this greedy algorithm does not return the maximum matching.

   (b) Let M and M' be matchings in a bipartite graph G. Suppose that $|M'| > 2|M|$. Show that there is an edge $e' \in M'$ such that $M \cup \{e'\}$ is a matching in G.

   (c) Use (b) to conclude that any matching constructed by the greedy algorithm in a bipartite graph G is at least *half* as large as the maximum matching in G.

Solution. **(a)** Consider a path with three edges, and an execution of the greedy algorithm in which the middle edge is added first.

(b) Consider the k connected components $C_1, \ldots, C_k$ of $M \cup M'$ — each is path or a cycle. Label a component C_i by an ordered pair $(|M \cap C_i|, |M' \cap C_i|)$. Now, if some C has a label of the form $(0, j)$, then it follows that $j = 1$, and this is an edge of M' that can be added to M. Otherwise, the labels are $\{x_i, y_i\}$, where $x_i \geq 1$ and $y_i \leq x_i + 1$ for each i. But then $|M'| - |M| = \sum_i (y_i - x_i) \leq k$ while $|M| = \sum_i x_i \geq k$, so we have $|M| \geq |M'| - |M|$. Rearranging this last inequality, we get $|M'| \leq 2|M|$.

Another way to prove this is the following. Since no edge of M' can be added to M, each $e \in M$ shares an endpoint with some $e' \in M'$. (It may share an endpoint with two edges in M'; then pick one arbitrarily.) Make the edge $e' \in M$ "pay for" the edge $e \in M$. Now, each edge $e \in M$ has been paid for by some edge $e' \in M'$, but each $e' \in M'$ has only two endpoints and hence pays for at most two edges in M. It follows that M' contains at most twice as many edges as M.

(c) Let M' be a matching of maximum size, and let M be the matching obtained by the greedy algorithm when it finally terminates. Then since there is no edge from M' that can be added to M, it follows from (a) that $|M| \geq \frac{1}{2}|M'|$.

2. Recall the *Shortest-First* greedy algorithm for the Interval Scheduling problem: Given a set of intervals, we repeatedly pick the shortest interval I, delete all the other intervals I' that intersect I, and iterate.

   In class, we saw that this algorithm does *not* always produce a maximum-size set of non-overlapping intervals. However, it turns out to have the following interesting approximation guarantee. If s^* is the maximum size of a set of non-overlapping intervals, and s is the size of the set produced by the *Shortest-First* algorithm, then $s \geq \frac{1}{2}s^*$. (That is, *Shortest-First* is a 2-approximation.)

   Prove this fact.

   Solution. The algorithm clearly accepts a feasible set of requests. To prove that it is a 2-approximation consider a set of optimal requests O. For each request $i \in O$ consider what requests in A it conflicts with. We claim that each request in O can conflict with at most 2 request in O. To see this assume that the there is an interval in $j \in A$ that conflicts with at least three requests in $i_1, i_2, i_3 \in O$. These three requests do not conflict with each other, as they are part of the optimum solution O, so they are ordered in time. Say they are ordered as i_1 first, then i_2 followed by i_3. Since requests j conflicts with both i_1 and i_3 it must contain the requests i_2 as a subset. This is not possible as the algorithm always selects minimal requests.

   The algorithm terminates only when R is empty, so each request in O must either be included in A or must conflict with a request in A. Each request in A conflicts with at most two requests in O, so we must have $|O| \leq 2|A|$ as claimed.

3. Suppose you are given a set of positive integers $A = \{a_1, a_2, \ldots, a_n\}$ and a positive integer B. A subset $S \subseteq A$ is called *feasible* if the sum of the numbers in S does not exceed B:

$$\sum_{a_i \in S} a_i \leq B.$$

   The sum of the numbers in S will be called the *total sum* of S.

   You would like to select a feasible subset S of A whose total sum is as large as possible.

   Example. If $A = \{8, 2, 4\}$ and $B = 11$, then the optimal solution is the subset $S = \{8, 2\}$.

   (a) Here is an algorithm for this problem.

   Initially $S = \phi$ Define $T = 0$ For $i = 1, 2, \ldots, n$ If $T + a_i \leq B$ then $S \leftarrow S \cup \{a_i\}$ $T \leftarrow T + a_i$ Endif Endfor

   Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A.

   (b) Give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$.

You should give a proof that your algorithm has this property, and give a brief analysis of its running time.

(c) In this part, we want you to give an algorithm with a *stronger* guarantee than what you produced in part (b). Specifically, give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least (2/3) times as large as the maximum total sum of any feasible set $S' \subseteq A$.

You should give a proof that your algorithm has this property, and give a brief analysis of its running time.

4. In the *bin packing problem*, we are given a collection of n items with *weights* $w_1, w_2, \ldots, w_n$. We are also given a collection of *bins*, each of which can hold a total of W units of weight. (We will assume that W is at least as large as each individual w_i.)

   You want to pack each item in a bin; a bin can hold multiple items, as long as the total of weight of these items does not exceed W. The goal is to pack all the items using as few bins as possible.

   Doing this optimally turns out to be NP-complete, though you don't have to prove this.

   Here's a *merging heuristic* for solving this problem: We start with each item in a separate bin and then repeatedly "merge" bins if we can do this without exceeding the weight limit. Specifically:

   > Merging Heuristic: Start with each item in a different bin While there exist two bins so that the union of their contents has total weight $\leq W$ Empty the contents of both bins Place all these items in a single bin. Endwhile Return the current packing of items in bins.

   Notice that the merging heuristic sometimes has the freedom to choice several possible pairs of bins to merge. Thus, on a given instance, there are multiple possible executions of the heuristic.

   Example. Suppose we have four items with weights $1, 2, 3, 4$, and $W = 7$. Then in one possible execution of the merging heuristic, we start with the items in four different bins; then we merge the bins containing the first two items; then we merge the bins containing the latter two items. At this point we have a packing using two bins, which cannot be merged. (Since the total weight after merging would be 10, which exceeds $W = 7$.)

   (a) Let's declare the size of the input to this problem to be proportional to

$$n + \log W + \sum_{i=1}^{n} \log w_i.$$

(In other words, the number of items plus the number of bits in all the weights.)

Prove that the merging heuristic always terminates in time polynomial in the size of the input. (In this question, as in NP-complete number problems from class, you should account for the time required to perform any arithmetic operations.)

(b) Give an example of an instance of the problem, and an execution of the merging heuristic on this instance, where the packing returned by the heuristic does not use the minimum possible number of bins.

(c) Prove that in any execution of the merging heuristic, on any instance, the number of bins used in the packing returned by the heuristic is at most twice the minimum possible number of bins.

5. Here's a way in which a different heuristic for bin packing can arise. Suppose you're acting as a consultant for the Port Authority of a small Pacific Rim nation. They're currently doing a multi-billion dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port.

   Here's a basic sort of problem they face. A ship arrives, with n containers of weight $w_1, w_2, \ldots, w_n$. Standing on the dock is a set of trucks, each of which can hold K units of weight. (You can assume that K and each w_i is an integer.) You can stack multiple containers in each truck, subject to the weight restriction of K; the goal is to minimize the number of trucks that are needed in order to carry all the containers. This problem is NP-complete (you don't have to prove this).

   A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until you get to a container that would overflow the weight limit. Now declare this truck "loaded" and send it off; then continue the process with a fresh truck.

   (a) Give an example of a set of weights, and a value of K, where this algorithm does not use the minimum possible number of trucks.

   (b) Show that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

   Solution. **(a)** Let $\{w_1, w_2, w_3\} = \{1, 2, 1\}$, and $K = 2$. Then the greedy algorithm here will use three trucks, whereas there is a way to use just two.

   (b) Let $W = \sum_i w_i$. Note that in *any* solution, each truck holds at most K units of weight, so W/K is a lower bound on the number of trucks needed.

   Suppose the number of trucks used by our greedy algorithm is an odd number $m = 2q + 1$. (The case when m is even is essentially the same, but a little easier.) Divide the trucks used into consecutive groups of two, for a total of $q + 1$ groups. In each group but the last, the total weight of containers must be *strictly* greater than K (else, the second truck in the group would not have been started then) — thus, $W > qK$, and so $W/K > q$. It follows by our argument above that the optimum solution uses at least $q + 1$ trucks, which is within a factor of 2 of $m = 2q + 1$.

6. You are asked to consult for a business where clients bring in jobs each day for processing. Each job has a processing time t_i that is known when the job arrives. The company has a set of 10 machines, and each job can be processed on any of these 10 machines.

   At the moment the business is running the simple *Greedy-Balance* algorithm we discussed in class. They have been told that this may not be the best approximation algorithm possible, and they are wondering if they should be afraid of bad performance. However, they are reluctant to change the scheduling as they really like the simplicity of the current algorithm: jobs can be assigned to machines as soon as they arrive, without having to defer the decision until later jobs arrive.

   In particular, they have heard that this algorithm can produce solutions with makespan as much as twice the minimum possible; but their experience with the algorithm has been quite good: they have been running it each day for the last month, and they have not observed it to produce a makespan more than 20% above the average load, $\frac{1}{10}\sum_i t_i$.

   To try understanding why they don't seem to be encountering this factor-of-two behavior, you ask a bit about the kind of jobs and loads they see. You find out that the sizes of jobs range between 1 and 50, i.e., $1 \le t_i \le 50$ for all jobs i; and the total load $\sum_i t_i$ is quite high each day: it is always at least 3000.

   Prove that on the type of inputs the company sees, the *Greedy-Balance* algorithm will always find a solution whose makespan is at most 20% above the average load.

   Solution. In the textbook we prove that

$$T - t_j \le T^*.$$

   where T is our makespan, t_j is a size of a job and T^* is the optimal mak espan. We also proved that the optimal makespan is at least the average load, which is at least 300 in our case:

$$T^* \ge \frac{1}{m}\sum_j t_j \ge \frac{1}{10}3000 = 300.$$

   We also know that $t_j \le 50$. Therefore the ratio of difference between our makespan and the optimal makespan to the optimal makespan is at most

$$\frac{T - T^*}{T^*} \le \frac{t_j}{T^*} \le \frac{50}{300} = \frac{1}{6} \le 20\%$$

7. Consider an optimization version of the *Hitting Set* problem defined as follows. We are given a set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A. Also, each element $a_i \in A$ has a *weight* $w_i \ge 0$. The problem is to find a hitting set $H \subseteq A$ such that the total weight of the elements in H, $\sum_{a_i \in H} w_i$, is as small as possible.

(Recall from Problem Set 7 that H is a hitting set if $H \cap B_i$ is not empty for each i). Let $b = \max_i |B_i|$ denote the maximum size of any of the sets $B_1, B_2, \ldots, B_m$. Give a polynomial time approximation algorithm for this problem that finds a hitting set whose total weight is at most b times the minimum possible.

Solution. Note that in case when all sets B_i have exactly 2 elements (i.e. $b = 2$), the Hitting Set problem is equivalent to the Vertex Cover problem (two-element sets B_i correspond to edges). In section 10.4 we learn how to find a 2-approximation for Vertex Cover problem. Now we generalize this technique for arbitrary b.

Consider the following problem for Linear Programming:

$$\begin{array}{lll} \text{Min} & \sum\limits_{i=1}^{n} w_i x_i & \\ \text{s.t.} & 0 \le x_i \le 1 & \text{for all } i = 1, \ldots, n \\ & \sum\limits_{i: a_i \in B_j} x_i \ge 1 & \text{for all } j = 1, \ldots, m \text{ (all sets are hit)} \end{array}$$

Let x be the solution of this problem, and w_{LP} is a value of this solution (i.e. $w_{LP} = \sum_{i=1}^{n} w_i x_i$).

Now define the set S to be all those elements where $x_i \ge 1/b$:

$$S = \{a_i \mid x_i \ge 1/b\}$$

Claim 10.1 *S is a hitting set.*

Proof. W ■

e want to prove that any set B_j intersects with S. We know that the sum of all x_i where $a_i \in B_j$ is at least 1. The set B_j contains at most b elements. Therefore some $x_i \ge 1/b$, for some $a_i \in B_j$. By definition of S, this element $a_i \in S$. So, B_j intersects with S by a_i.

Claim 10.2 *The total weight of all elements in S is at most $b \cdot w_{LP}$.*

Proof. F ■

or each $a_i \in S$ we know that $x_i \ge 1/b$, i.e., $1 \le bx_i$. Therefore

$$w(S) = \sum_{a_i \in S} w_i \le \sum_{a_i \in S} w_i \cdot bx_i \le b \sum_{i=1}^{n} w_i x_i = bw_{LP}$$

Claim 10.3 *Let S^* be the optimal hitting set. Then $w_{LP} \le w(S^*)$.*

Proof. S ■

et $x_i = 1$ if a_i is in S^*, and $x_i = 0$ otherwise. Then the vector x satisfy constrains of our problem for Linear Programming:

$$\begin{array}{ll} 0 \le x_i \le 1 & \text{for all } i = 1, \dots, n \\ \sum\limits_{i:a_i \in B_j} x_i \ge 1 & \text{for all } j = 1, \dots, m \text{ (because all sets are hit)} \end{array}$$

Therefore the optimal solution is not worse that this particular one. That is,

$$w_{LP} \le \sum_{i=1}^{n} w_i x_i = \sum_{a_i \in S} w_i = w(S^*)$$

Therefore we have a hitting set S, such that $w(S) \le b \cdot w(S^*)$.

8. Consider the following maximization version of the *three-dimensional matching* problem. Given disjoint sets X, Y, and Z, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, a subset $M \subset T$ is a *three-dimensional matching* if each element of $X \cup Y \cup Z$ is contained in at most one of these triples. The *maximum three-dimensional matching* problem is to find a three-dimensional matching M of maximum size. (The size of the matching, as usual, is the number of triples it contains. You may assume $|X| = |Y| = |Z|$ if you want.)

   Give a polynomial time algorithm that finds a three-dimensional matching of size at least $\frac{1}{3}$ times the maximum possible size.

   Solution. We will use the following simple algorithm. Consider triples of T in any order, and add them if they do not conflict with previously added triples. Let M denote the set returning by this algorithm and M^* be the optimal three-dimensional matching.

   Claim 10.4 *The size of M is at least $1/3$ of the size of M^*.*

   Proof. E ■

   ach triple (a, b, c) in M^* must intersect at least one triple in our matching M (or else we could extend M greedily with (a, b, c)). One triple in M can only be in conflict with at most 3 triples in M^* as edges in M^* are disjoint. So M^* can have at most 3 times as many edges as M has.

9. At a lecture in a computational biology conference one of us attended about a year ago, a well-known protein chemist talked about the idea of building a "representative set" for a large collection of protein molecules whose properties we don't understand. The idea would be to intensively study the proteins in the representative set, and thereby learn (by inference) about all the proteins in the full collection.

   To be useful, the representative set must have two properties.

- It should be relatively small, so that it will not be too expensive to study it.
- Every protein in the full collection should be "similar" to some protein in the representative set. (In this way, it truly provides some information about all the proteins.)

More concretely, there is a large set P of proteins. We define similarity on proteins by a *distance function* d — given two proteins p and q, it returns a number $d(p,q) \geq 0$. In fact, the function $d(\cdot,\cdot)$ most typically used is the *edit distance*, or *sequence alignment* measure, which we looked at when we studied dynamic programming. We'll assume this is the distance being used here. There is a pre-defined distance cut-off Δ that's specified as part of the input to the problem; two proteins p and q are deemed to be "similar" to one another if and only if $d(p,q) \leq \Delta$.

We say that a subset of P is a *representative set* if for every protein p, there is a protein q in the subset that is similar to it — i.e. for which $d(p,q) \leq \Delta$. Our goal is to find a representative set that is as small as possible.

(a) Give a polynomial-time algorithm that approximates the minimum representative set to within a factor of $O(\log n)$. Specifically, your algorithm should have the following property: if the minimum possible size of a representative set is s^*, your algorithm should return a representative set of size at most $O(s^* \log n)$.

(b) Note the close similarity between this problem and the Center Selection problem — a problem for which we considered approximation algorithms in the text. Why doesn't the algorithm described there solve the current problem?

10. Suppose you are given an $n \times n$ *grid graph* G, as in the figure below.

Figure 8: A grid graph.

Associated with each node v is a *weight* $w(v)$, which is a non-negative integer. You may assume that the weights of all nodes are distinct. Your goal is to choose an independent set S of nodes of the grid, so that the sum of the weights of the nodes in S is as large as possible. (The sum of the weights of the nodes in S will be called its *total weight*.)

Consider the following greedy algorithm for this problem:

> The ”heaviest-first” greedy algorithm: Start with S equal to the empty set. While some node remains in G Pick a node v_i of maximum weight. Add v_i to S. Delete v_i and its neighbors from G. end while Return S

(a) Let S be the indepedent set returned by the “heaviest-first” greedy algorithm, and let T be any other independent set in G. Show that for each node $v \in T$, either $v \in S$, or there is a node $v' \in S$ so that $w(v) \leq w(v')$ and (v, v') is an edge of G.

(b) Show that the “heaviest-first” greedy algorithm returns an independent set of total weight at least 1/4 times the maximum total weight of any independent set in the grid graph G.

11 Local Search

1. Consider the load balancing problem from the chapter on approximation algorithms. Some friends of yours are running a collection of Web servers, and they've designed a local search heuristic for this problem, different from the algorithms described in that chapter.

   Recall that we have m machines $M_1, \ldots, M_m$, and we must assign each job to a machine. The load of the i^{th} job is denoted t_i. The *makespan* of an assignment is the *maximum load* on any machine:
$$\max_{\text{machines } M_i} \sum_{\text{jobs } j \text{ assigned to } M_i} t_j.$$

   Your friends' local search heuristic works as follows. They start with an arbitrary assignment of jobs to machines, and they then repeatedly try to apply the following type of “swap move”:

   > Let $A(i)$ and $A(j)$ be the jobs assigned to machines M_i and M_j respectively. To perform a swap move on M_i and M_j, choose subsets of jobs $B(i) \subseteq A(j)$ and $B(j) \subseteq A(j)$, and “swap” these jobs between the two machines. That is, update $A(i)$ to be $A(i) \cup B(j) - B(i)$ and update $A(j)$ to be $A(j) \cup B(i) - B(j)$. (One is allowed to have $B(i) = A(i)$, or to have $B(i)$ be the empty set; and analogously for $B(j)$.)

   Consider a swap move applied to machines to machines M_i and M_j. Suppose the loads on M_i and M_j before the swap are T_i and T_j respectively, and the loads after the swap are T_i' and T_j'. We say that the swap move is *improving* if $\max(T_i', T_j') < \max(T_i, T_j)$; in other words, the larger of the two loads involved has strictly decreased. We say that an assignment of jobs to machines is *stable* if there does not exist an improving swap move, beginning with the current assignment.

Thus, the local search heuristic simply keeps executing improving swap moves until a stable assignment is reached; at this point, the resulting stable assignment is returned as the solution.

Example: Suppose there are two machines: in the current assignment, the machine M_1 has jobs of sizes $1, 3, 5, 8$, and machine M_2 has jobs of sizes $2, 4$. Then one possible improving swap move would be to define $B(1)$ to consist of the job of size 8, and define $B(2)$ to consist of the job of size 2. After these two sets are "swapped," the resulting assignment has jobs of size $1, 2, 3, 5$ on M_1, and jobs of size $4, 8$ on M_2. This assignment is stable. (It also has an optimal makespan of 12.)

(a) As specified, there is no explicit guarantee that this local search heuristic will always terminate — i.e., what if it keeps cycling forever through assignments that are not stable?

Prove that in fact the local search heuristic terminates in a finite number of steps, with a stable assignment, on any instance.

(b) Show that any stable assignment has a makespan that is within a factor of 2 of the minimum possible makespan.

2. Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

   You are given such a complete binary tree T, but the labeling is only specified in the following *implicit* way: for each node v, you can determine the value x_v by *probing* the node v. Show how to find a local minimum of T using only $O(\log n)$ *probes* to the nodes of T.

   Solution. For simplicity, we will say u *is smaller than* v, or $u \prec v$, if $x_u < x_v$. We will extend this to sets: if S is a set of nodes, we say $u \prec S$ if u has a smaller value than any node in S.

   The algorithm is the following. We begin at the root r of the tree, and see if r is smaller than its two children. If so, the root is a local minimum. Otherwise, we move to any smaller child and iterate.

   The algorithm terminates when either (1) we reach a node v that is smaller than both its children, or (2) we reach a leaf w. In the former case, we return v; in the latter case, we return w.

   The algorithm performs $O(d) = O(\log n)$ probes of the tree; we must now argue that the returned value is a local minimum. If the root r is returned, then it is a local minimum as explained above. If we terminate in case (1), v is a local minimum because v is smaller than its parent (since it was chosen in the previous iteration) and its two children (since we terminated). If we terminate in case (2), w is a local minimum because w is smaller than its parent (again since it was chosen in the previous iteration).

3. $(*)$ Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, ℓ) are joined by an edge if and only if $|i-k| + |j-\ell| = 1$.)

   We use some of the terminology of the previous question. Again, each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

   Solution. Let B denote the set of nodes on the *border* of the grid G — i.e. the outermost rows and columns. Say that G has *Property (∗)* if it contains a node $v \notin B$ that is adjacent to a node in B and satisfies $v \prec B$. Note that in a grid G with Property $(*)$, the *global minimum* does not occur on the border B (since the global minimum is no larger than v, which is smaller than B) — hence G has at least one local minimum that does not occur on the border. We call such a local minimum an *internal local minimum*

   We now describe a recursive algorithm that takes a grid satisfying Property $(*)$ and returns an internal local minimum, using $O(n)$ probes. At the end, we will describe how this can be easily converted into a solution for the overall problem.

   Thus, let G satisfy Property $(*)$, and let $v \notin B$ be adjacent to a node in B and smaller than all nodes in B. Let C denote the union of the nodes in the middle row and middle column of G, not counting the nodes on the border. Let $S = B \cup C$; deleting S from G divides up G into four sub-grids. Finally, let T be all nodes adjacent to S.

   Using $O(n)$ probes, we find the node $u \in S \cup T$ of minimum value. We know that $u \notin B$, since $v \in S \cup T$ and $v \prec B$. Thus, we have two cases. If $u \in C$, then u is an internal local minimum, since all of the neighbors of u are in $S \cup T$, and u is smaller than all of them. Otherwise, $u \in T$. Let G' be the sub-grid containing u, together with the portions of S that border it. Now, G' satisfies Property $(*)$, since u is adjacent to the border of G' and is smaller than all nodes on the border of G'. Thus, G' has an internal local minimum, which is also an internal local minimum of G. We call our algorithm recursively on G' to find such an internal local minimum.

   If $T(n)$ denotes the number of probes needed by the algorithm to find an internal local minimum in an $n \times n$ grid, we have the recurrence $T(n) = O(n) + T(n/2)$, which solves to $T(n) = O(n)$.

   Finally, we convert this into an algorithm to find a local minimum (not necessarily internal) of a grid G. Using $O(n)$ probes, we find the node v on the border B of minimum value. If v is a corner node, it is a local minimum and we're done. Otherwise, v has a unique neighbor u not on B. If $v \prec u$, then v is a local minimum and again we're done. Otherwise, G satisfies Property $(*)$ (since u is smaller than every node on B), and we call the above algorithm.

12 Randomized Algorithms

1. In the first lecture on randomization, we saw a simple distributed protocol to solve a particular contention-resolution problem. Here is another setting in which randomization can help with contention-resolution, through the distributed construction of an independent set.

   Suppose we have a system with n processes. Certain pairs of processes are in *conflict*, meaning that they both require access to a shared resource. In a given time interval, the goal is to schedule a large subset S of the processes to run — the rest will remain idle — so that no two conflicting processes are both in the scheduled set S. We'll call such a set S *conflict-free*.

   One can picture this process in terms of a graph $G = (V, E)$ with a node representing each process and an edge joining pairs of processes that are in conflict. It is easy to check that a set of processes S is conflict-free if and only if it forms an independent set in G. This suggests that finding a maximum-size conflict-free set S, for an arbitrary conflict G will be difficult (since the general independent set problem is reducible to this problem). Nevertheless, we can still look for heuristics that find a reasonably large conflict-free set. Moreover, we'd like a simple method for achieving this without centralized control: each process should communicate with only a small number of other processes, and then decide whether or not it should belong to the set S.

   We will suppose for purposes of this question that each node has exactly d neighbors in the graph G. (That is, each process is in conflict with exactly d other processes.)

   (a) Consider the following simple protocol.

   > Each process P_i independently picks a random value x_i; it sets x_i to 1 with probability $\frac{1}{2}$ and set x_i to 0 with probability $\frac{1}{2}$. It then decides to enter the set S if and only if it chooses the value 1, and each of the processes with which it is in conflict chooses the value 0.

   Prove that the set S resulting from the execution of this protocol is conflict-free. Also, give a formula for the expected size of S in terms of n (the number of processes) and d (the number of conflicts per process).

   (b) The choice of the probability $\frac{1}{2}$ in the protocol above was fairly arbitrary, and it's not clear that it should give the best system performance. A more general specification of the protocol would replace the probability $\frac{1}{2}$ by a parameter p between 0 and 1, as follows:

Each process P_i independently picks a random value x_i; it sets x_i to 1 with probability p and set x_i to 0 with probability $1 - p$. It then decides to enter the set S if and only if it chooses the value 1, and each of the processes with which it is in conflict chooses the value 0.

In terms of the parameters of the graph G, give a value of p so that the expected size of the resulting set S is as large as possible. Give a formula for the expected size of S when p is set to this optimal value.

Solution. **(a)** Assume that using the described protocol, we get a set S that is not conflict free. Then there must be 2 processes P_i and P_j in the set S that both picked the value 1 and are going to want to share the same resource. But this contradicts the way our protocol was implemented, since we selected processes that picked the value 1 and whose set of conflicting processes all picked the value 0. Thus if P_i and P_j both picked the value 1, neither of them would be selected and so the resulting set S is conflict free. For each process P_i, the probability that it is selected depends on the fact that P_i picks the value 1 and all its d conflicting processes pick the value 0. Thus $P[P_i\text{selected}] = \frac{1}{2} * (\frac{1}{2})^d$. And since there are n processes that pick values independently, the expected size of the set S is $n * (\frac{1}{2})^{d+1}$

(b) Now a process P_i picks the value 1 with probability p and 0 with probability $1 - p$. So the probability that P_i is selected (i.e. P_i picks the value 1 and its d conflicting processes pick the value 0) is $p * (1 - p)^d$. Now we want to maximize the probability that a process is selected. Using calculus, we take the derivative of $p(1 - p)^d$ and set it equal to 0 to solve for the value of p that gives the objective it's maximum value. The derivative of $p(1 - p)^d$ is $(1 - p)^d - dp(1 - p)^{d-1}$. Solving for p, we get $p = \frac{1}{d+1}$. Thus the probability that a process is selected is $\frac{d^d}{(d+1)^{d+1}}$ and the expected size of the set S is $n * \frac{d^d}{(d+1)^{d+1}}$. Note that this is $\frac{n}{d}$ times $(1 - \frac{1}{d+1})^{d+1}$ and this later term is $\frac{1}{e}$ in the limit and so by changing the probability, we got a fraction of $\frac{n}{d}$ nodes. Note that with $p = 0.5$, we got an exponentially small subset in terms of d.

2. In class, we designed an approximation algorithm to within a factor of 7/8 for the *MAX 3-SAT* problem, where we assumed that each clause has terms associated with 3 different variables. In this problem we will consider the analogous *MAX SAT* problem: given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_n\}$, find a truth assignment satisfying as many of the clauses as possible. Each clause has at least one term in it, but otherwise we do not make any assumptions on the length of the clauses: there may be clauses that have a lot of variables, and others may have just a single variable.

   (a) First consider the randomized approximation algorithm we used for *MAX 3-SAT*, setting each variable independently to *true* or *false* with probability 1/2 each. Show that the expected number of clauses satisfied by this random assignment is at least $k/2$, i.e., half of the clauses is satisfied in expectation. Give an example to show that there are *MAX SAT* instances such that no assignment satisfies more than half of the clauses.

(b) If we have a clause that consists just of a single term (e.g. a clause consisting just of x_1, or just of $\overline{x_2}$), then there is only a single way to satisfy it: we need to set the corresponding variable in the appropriate way. If we have two clauses such that one consists of just the term x_i, and the other consists of just the negated term $\overline{x_i}$, then this is a pretty direct contradiction.

Assume that our instance has no such pair of "conflicting clauses"; that is, for no variable x_i do we have both a clause $C = \{x_i\}$ and a clause $C' = \{\overline{x_i}\}$. Modify the above randomized procedure to improve the approximation factor from $1/2$ to at least a .6 approximation, that is, change the algorithm so that the expected number of clauses satisfied by the process is at least $.6k$.

(c) Give a randomized polynomial time algorithm for the general *MAX SAT* problem, so that the expected number of clauses satisfied by the algorithm is at least a .6 fraction of the maximum possible.

(Note that by the example in part (a), there are instances where one cannot satisfy more than $k/2$ clauses; the point here is that we'd still like an efficient algorithm that, in expectation, can satisfy a .6 fraction *of the maximum that can be satisfied by an optimal assignment.*)

Solution.

(a) Consider a clause C_i with n variables. The probability that the clause is not satisfied is $\frac{1}{2^m}$ and so the probability that it is satisfied is 1 less this quantity. The worst case is when C_i has just one variable, i.e. $n = 1$, in which case the probability of the clause being satisfied is $\frac{1}{2}$. Since there are k clauses, the expected number of clauses being satisfied is atleast $\frac{k}{2}$. Consider the two clauses x_1 and $\overline{x_1}$. Clearly only one of these can be satisfied.

(b) For variables that occur in single variable clauses, let the probability of setting the variable so as to satisfy the clause be $p \geq \frac{1}{2}$. For all other variables, let the probabilities be $\frac{1}{2}$ as before. Now for a clause C_i with n variables, $n \geq 2$, the probability of satisfying it is at worst $(1 - \frac{1}{2^n}) \geq (1 - p^2)$ since $p \geq \frac{1}{2}$. Now to solve for p, we want to satisfy all clauses, so solve $p = 1 - p^2$ to get $p \approx 0.62$. And hence the expected number of satisfied clauses is $0.62n$.

(c) Let the total number of clauses be k. For each pair of single variable conflicting clauses, i.e. x_i and $\overline{x_i}$, remove one of them from the set of clauses. Assume we have removed m clauses. Then the maximum number of clauses we could satisfy is $k - m$. Now apply the algorithm described in the previous part of the problem to the $k - 2m$ clauses that had no conflict to begin with. The expected number of clauses we satisfy this way is $0.62 * (k - 2m)$. In addition to this we can also satisfy m of the $2m$ conflicting clauses and so we satisfy $0.62 * (k - 2m) + m \geq 0.62 * (k - m)$ clauses which is our desired target. Note that this algorithm is polynomial in the number of variables and clauses since we look at each clause once.

3. Let $G = (V, E)$ be an undirected graph. We say that a set of vertices $X \subseteq V$ is a *dominating set* if every vertex of G is a member of X or is joined by an edge to a member of X.

   Give a polynomial-time algorithm that takes an arbitrary d-regular graph and finds a dominating set of size $O\left(\frac{n \log n}{d}\right)$. (A graph is *d-regular* if each vertex is incident to exactly d edges.) Your algorithm can be either deterministic or randomized; if it is randomized, it must always return the correct answer and have an expected running time that is polynomial in n.

4. Consider a very simple on-line auction system that works as follows. There are n *bidding agents*; agent i has a bid b_i, which is a positive natural number. We will assume that all bids b_i are distinct from one another. The bidding agents appear in an order chosen uniformly at random, each proposes its bid b_i in turn, and at all times the system maintains a variable b^* equal to the highest bid seen so far. (Initially b^* is set to 0.)

   What is the expected number of times that b^* is updated when this process is executed, as a function of the parameters in the problem?

   Example: Suppose $b_1 = 20$, $b_2 = 25$, and $b_3 = 10$, and the bidders arrive in the order $1, 3, 2$. Then b^* is updated for 1 and 2, but not for 3.

   Solution. Let X be a random variable equal to the number of times that b^* is updated. We write $X = X_1 + X_2 + \cdots + X_n$, where $X_i = 1$ if the i^{th} bid in order causes b^* to be updated, and $X_i = 0$ otherwise.

   So $X_i = 1$ if and only if, focusing just on the sequence of the first i bids, the largest one comes at the end. But the largest value among the first i bids is equally likely to be anywhere, and hence $EX_i = 1/i$.

   Alternately, the number of permutations in which the number at position i is larger than any of the numbers before it can be computed as follows. We can choose the first i numbers in $\binom{n}{i}$ ways, put the largest in position i, order the remainder in $(i-1)!$ ways, and order the subsequent $(n-i)$ numbers in $(n-i)!$ ways. Multiplying this together, we have $\binom{n}{i}(i-1)!(n-i)! = n!/i$. Dividing by $n!$, we get $EX_i = 1/i$.

   Now, by linearity of expectation, we have $EX = \sum_{i=1}^{n} EX_i = \sum_{i=1}^{n} 1/i = H_n = \Theta(\log n)$.

5. One of the (many) hard problems that arises in genome mapping can be formulated in the following abstract way. We are given a set of n *markers* $\{\mu_1, \ldots, \mu_n\}$ — these are positions on a chromosome that we are trying to map — and our goal is to output a linear ordering of these markers. The output should be consistent with a set of k

constraints, each specified by a triple (μ_i, μ_j, μ_k), requiring that μ_j lie *between* μ_i and μ_k in the total ordering that we produce.

Now, it is not always possible to satisfy all constraints simultaneously, so we wish to produce an ordering that satisfies as many as possible. Unfortunately, deciding whether there is an ordering that satisfies at least k' of the k constraints is an NP-complete problem (you don't have to prove this.)

Give a constant $\alpha > 0$ (independent of n) and an algorithm with the following property. If it is possible to satisfy k^* of the constraints, then the algorithm produces an ordering of markers satisfying at least αk^* of the constraints. You can provide either be a deterministic algorithm running in polynomial time, or a randomized algorithm which has expected polynomial running time and always produces an approximation within a factor of α.

Solution.

We interpret the constraint (μ_i, μ_j, μ_k) to mean that we require one of the subsequences $\ldots, \mu_i, \ldots, \mu_j, \ldots, \mu_k, \ldots$ or $\ldots, \mu_k, \ldots, \mu_j, \ldots, \mu_i, \ldots$ to occur in the ordering of the markers. (One could also interpret it to mean that just the first of these subsequences occurs; this will affect the analysis below by a factor of 2.)

Suppose that we choose an order for the n markers uniformly at random. Let X_t denote the random variable whose value is 1 if the t^{th} constraint (μ_i, μ_j, μ_k) is satisfied, and 0 otherwise. The six possible subsequences of $\{\mu_i, \mu_j, \mu_k\}$ occur with equal probability, and two of them satisfy the constraint; thus $EX_t = \frac{1}{3}$. Hence if $X = \sum_t X_t$ gives the total number of constraints satisfied, we have $EX = \frac{1}{3}k$.

So if our random ordering satisfies a number of constraints that is at least the expectation, we have satisfied at least $\frac{1}{3}$ of all constraints, and hence at least $\frac{1}{3}$ of the maximum number of constraints that can be simultaneously satisfied.

To construct an algorithm that *only* produces solutions within a factor of $\frac{1}{3}$ of optimal, we simply repeatedly generate random orderings until $\frac{1}{3}k$ of the constraints are satisfied. To bound the expected running time of this algorithm, we must give a lower bound on the probability p^+ that a single random ordering will satisfy at least the expected number of constraints; the expected running time will then be at most $1/p^+$ times the cost of a single iteration.

First note that k is at most n^3, and define $k' = \frac{1}{3}k$. Let k'' denote the greatest integer strictly less than k'. Let p_j denote the probability that we satisfy j of the constraints. Thus $p^+ = \sum_{j \geq k'} p_j$; we define $p^- = \sum_{j<k'} p_j = 1 - p^+$. Then we have

$$\begin{aligned}
k' &= \sum_j jp_j \\
&= \sum_{j<k'} jp_j + \sum_{j \geq k'} jp_j \\
&\leq \sum_{j<k'} k''p_j + \sum_{j \geq k'} n^3 p_j
\end{aligned}$$

$$= k''(1-p^+) + n^3 p^+$$

from which it follows that

$$(k'' + n^3)p^+ \geq k' - k'' \geq \frac{1}{3}.$$

Since $k'' \leq n^3$, we have $p^+ \geq \frac{1}{6n^3}$, and so we are done.

6. Suppose that a tree network is grown according to the following randomized process. We begin in step 1, with a single isolated node v_1. In any step $k \geq 2$, we introduce a new node v_k and draw an edge from v_k to a node chosen uniformly at random from $v_1, v_2, \ldots, v_{k-1}$. We stop after step n, with a tree on n nodes.

   What is the expected number of leaves in the resulting tree?

   Solution. We let X denote the random variable equal to the number of leaves; we define $X_i = 1$ if node v_i is a leaf, and $X_i = 0$ otherwise. Thus $X = X_1 + \cdots + X_n$.

   Now, in order for v_i to be a leaf, it must be that none of $v_{i+1}, v_{i+2}, \ldots, v_n$ attach to it. This means that $EX_1 = 0$, $EX_n = 1$, and for $1 < i < n$, we have

   $$EX_i = \frac{i-1}{i} \cdot \frac{i}{i+1} \cdots \frac{n-2}{n-1} = \frac{i-1}{n-1}.$$

   Hence

   $$EX = \sum_{i=1}^{n} \frac{i-1}{n-1} = \frac{n(n-1)}{2(n-1)} = \frac{n}{2}.$$

7. Let $G = (V, E)$ be an undirected graph with n nodes and m edges. For a subset $X \subseteq V$, we use $G[X]$ to denote the subgraph *induced* on X — that is, the graph $(X, \{(u,v) \in E : u, v \in X\})$.

   We are given a natural number $k \leq n$, and are interested in finding a set of k nodes that induces a "dense" subgraph of G; we'll phrase this concretely as follows. Give a polynomial-time algorithm that produces, for a given natural number $k \leq n$, a set $X \subseteq V$ of k nodes with the property that the induced subgraph $G[X]$ has at least $\frac{mk(k-1)}{n(n-1)}$ edges.

   You may give either (a) a deterministic algorithm, or (b) a randomized algorithm that has an expected running time achieving the given bound, and which only outputs correct answers.

8. We are given a set of variables $x_1, x_2, \ldots, x_n$, each of which can take one of the three values $\{0, 1, 2\}$. We are also given a set of k *constraints* $C_1, C_2, \ldots, C_k$, each of which has the form $x_i \neq x_j$ for some choice of i and j. An assignment of values to the variables *satisfies* a constraint $x_i \neq x_j$ if the value assigned to x_i is different from the value assigned to x_j.

Consider the problem of finding an assignment of values to variables that maximizes the number of satisfied constraints. This problem is NP-hard, though you don't have to prove this.

Let c^* denote the maximum possible number of constraints that can be satisfied by an assignment of values to variables. Give a polynomial-time algorithm that produces an assignment satisfying at least $\frac{2}{3}c^*$ constraints. If you want, your algorithm can be randomized; in this case, the *expected* number of constraints it satisfies should be at least $\frac{2}{3}c^*$. In either case, you should prove that your algorithm has the desired performance guarantee.

9. Suppose you're a consultant for Price Gouging Unlimited, and they come to you with the following problem. They've contracted out their algorithmic skills to a large long-distance telephone carrier, which is trying to redesign its area-code system to improve revenue. Their area-code system works as follows: calls between two numbers within one area code are *local* and cost c_1 cents per minute; calls between two numbers in different area codes are *long distance* and cost c_2 cents per minute, where $c_2 > c_1$.

   The particular problem being tossed around in the halls of PGU at the moment is as follows. The phone company has selected a specific one of its area codes — call it A — and it plans to split it so as to maximize its revenue on the resulting set of long-distance calls. More concretely, they have a graph $G = (V, E)$ whose vertex set is the set of all phone number within area code A. Between each pair $u, v \in V$ there is an edge (u, v) whose *weight* w_{uv} is equal to the total number of minutes per month that phone numbers u and v are connected to each other. So the current revenue generated by calls within area code A is $\sum_{u,v \in V} c_1 w_{uv}$. We'll assume (unrealistically) that these weights w_{uv} remain the same from one month to the next. The phone company plans to

   - create two new area codes B and C,
   - partition the phone numbers in the set V into sets V_1 and V_2,
   - assign V_1 to area code B, and
   - assign V_2 to area code C.

   A phone call from B to C (or from C to B) now becomes a long-distance call and costs c_2 cents per minute — thus, the phone company stands to make more money per month on phone calls among numbers in the set V after it splits the area code.

   Your goal is: find the partition (V_1, V_2) of V that maximizes the phone company's monthly revenue, subject to the weights $\{w_{uv}\}$.

   Here's an extremely simple randomized algorithm for this problem:

   > For each phone number, assign it independently at random to one of the two area codes, with equal probability.

Show that the expected value of the revenue from the partition generated by this algorithm is at least 50% as large as the revenue generated by the optimal partition.

Solution. Let X_{uv} be a random variable that is equal to 1 if u and v are placed in different area codes, and 0 if they are placed in the same area code. Let $Y_{uv} = w_{uv}(c_1 + (c_2 - c_1)X_{uv})$ and $X = \sum_{u,v} Y_{uv}$; so X is the revenue of the randomized algorithm.

Now,

$$EX_{uv} = \Pr[(u \in B \wedge v \in C) \vee (u \in C \wedge v \in B)] = \frac{1}{2},$$

so $EY_{uv} = \frac{1}{2}(c_1 + c_2)w_{uv}$, and by linearity of expectation we have $EX = \sum_{u,v} EY_{uv} = \frac{1}{2}(c_1 + c_2)W$, where $W = \sum_{u,v} w_{uv}$. Since the optimal revenue is at most c_2W, the expected revenue of the randomized algorithm is within a 50% factor.

10. Assume you have n balls and n bins, and each ball is placed in a bin selected independently at random (with each bin equally likely). Throughout this problem use the approximation $(1 - 1/n)^n \approx 1/e$ whenever it is useful.

    (a.) Prove that the expected number of empty bins is approaches n/e for large n. Hint: remember that expectation is linear.

    (b.) Assume that you have n jobs and n machines, and each job selects a machine independently at random (with each machine equally likely). Assume that if a machine is selected by more than one job, it will do the first job, and reject the rest. What is the expected number of rejected jobs?

    (c.) Now assume in the above job-machine example each machine will do the first two jobs, and reject the rest if more than two jobs are assigned to it. What is the expected number of rejected jobs now?

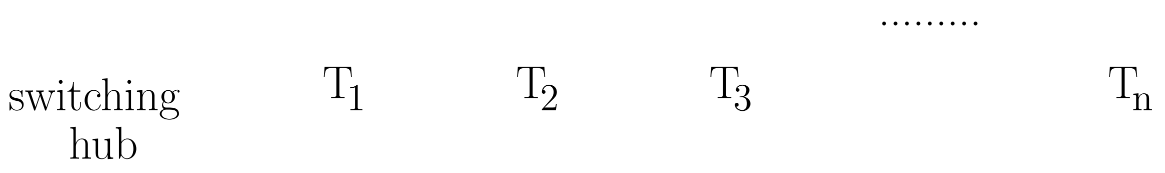
11. Consider the following analog of Karger's algorithm for finding minimum $s-t$-cuts. We will contract edges analogous to Karger's algorithm. Let s and t denote the possibly contracted node that contains the original nodes s and t respectively. To make sure that s and t do not get contracted, at each iteration we delete the edges connecting s and t, and select a random edge to contract among the remaining edges. Give an example to show that the probability that this method finds a minimum $s - t$ cut can be exponentially small. Hint: How many minimum capacity s-t cuts can there be?

12. Consider a county in which $100,000$ people vote in an election. There are only two candidates on the ballot: a Democratic candidate (denoted D) and a Republican candidate (denoted R). As it happens, this county is heavily Democratic, so $80,000$ people go to the polls with the intention of voting for D and $20,000$ go to the polls with the intention of voting for R.

    However, the layout of the ballot is a little confusing, so each voter, independently and with probability $\frac{1}{100}$, votes for the wrong candidate; i.e. the one that he or she *didn't* intend to vote for. (Remember that in this election, there are only two candidates on the ballot.)

Let X denote the random variable equal to the number of votes received by the Democratic candidate D, when the voting is conducted with this process of error. Determine the expected value of X, and give an explanation of your derivation of this value.

13. Out in a rural part of the state somewhere, n small towns have decided to get connected to a large Internet switching hub via a high-volume fiber-optic cable.



The towns are labeled $T_1, T_2, \ldots, T_n$, and they are all arranged on a single long highway, so that town T_i is i miles from the switching hub.

Now, this cable is quite expensive; it costs k dollars per mile, resulting in an overall cost of kn dollars for the whole cable. The towns get together and discuss how to divide up the cost of the cable.

First, one of the towns way out at the far end of the highway makes the following proposal.

> **Proposal A.** Divide the cost evenly among all towns, so each pays k dollars.

There's some sense in which Proposal A is fair, since it's like each town is paying for the mile of cable directly leading up to it.

But one of the towns very close to the switching hub objects, pointing out that the far-away towns are actually benefitting from a large section of the cable, whereas the close-in towns only benefit from a short section of it. So they make the following counter-proposal.

> **Proposal B.** Divide the cost so that the contribution of town T_i is proportional to i, its distance from the switching hub.

One of the other towns very close to the switching hub points out that there's another way to do a non-proportional division which is also natural. This is based on conceptually dividing the cable into n equal-length "edges" $e_1, \ldots, e_n$, where the first edge e_1 runs from the switching hub to T_1, and the i^{th} edge e_i $(i > 1)$ runs from T_{i-1} to T_i. Now we observe that while all the towns benefit from e_1, only the last town benefits from e_n. So they suggest

Proposal C. Divide the cost separately for each edge e_i. The cost of e_i should be shared equally by the towns $T_i, T_{i+1}, \dots, T_n$, since these are the towns "downstream" of e_i.

0.2inSo now the towns have many different options; which is the fairest? To resolve this they turn to the work of Lloyd Shapley, one of the most famous mathematical economists of the 20th century; he proposed what is now called the *Shapley value* as a general mechanism for sharing costs or benefits among several parties. It can be viewed as determining the "marginal contribution" of each party, *assuming the parties arrive in a random order.*

Here's how it would work concretely in our setting. Consider an ordering $\mathcal{O}$ of the towns, and suppose that the towns "arrive" in this order. The *marginal cost of town T_i in order $\mathcal{O}$* is determined as follows. If T_i is first in the order $\mathcal{O}$, then T_i pays ki, the cost of running the cable all the way from the switching hub to T_i. Otherwise, look at the set of towns that come before T_i in the order $\mathcal{O}$, and let T_j be the farthest among these towns from the switching hub. When T_i arrives, we assume the cable already reaches out to T_j but no farther. So if $j > i$ (T_j is farther out than T_i), then the marginal cost of T_i is 0, since the cable already runs past T_i on its way out to T_j. On the other hand, if $j < i$, then the marginal cost of T_i is $k(i - j)$: the cost of extending the cable from T_j out to T_i.

(For example, suppose $n = 3$ and the towns arrive in the order T_1, T_3, T_2. First T_1 pays k when it arrives. Then, when T_3 arrives, it only has to pay $2k$ to extend the cable from T_1. Finally, when T_2 arrives, it doesn't have to pay anything since the cable already runs past it out to T_3.)

0.2inNow, let X_i be the random variable equal to the marginal cost of town T_i when the order $\mathcal{O}$ is selected uniformly at random from all permutations of the towns. Under the rules of the Shapley value, the amount that T_i should contribute to the overall cost of the cable is the expected value of X_i.

The question is: Which of the above three proposals, if any, gives the same division of costs as the Shapley value cost-sharing mechanism? Give a proof for your answer.

Solution.

The Shapley value mechanism is equivalent to Proposal C. Under Proposal C, town T_i pays

$$\sum_{j=1}^{i} \frac{k}{n-j+1},$$

since there are $n - j + 1$ towns downstream of edge e_j, and so the contribution of T_i to edge e_j (for $1 \leq j \leq i$) is $k/(n - j + 1)$.

Now, let X_i be the random variable in the question. We decompose X_i into a random variable corresponding to each of the edges: let Y_j be the random variable equal to the amount town T_i pays for edge e_j ($1 \leq j \leq i$). Note that $X_i = \sum_{j=1}^{i} Y_j$, and so by linearity of expectation, $EX_i = \sum_{j=1}^{i} EY_j$.

What is EY_j? With probability $1/(n-j+1)$, town T_i is the first among the towns downstream of e_j to arrive, and it incurs a cost of k. Otherwise, when town T_i arrives, some other town downstream of e_j has already arrived, and so it incurs a cost of 0. Thus,

$$EY_j = 0\left(1 - \frac{1}{n-j+1}\right) + k\left(\frac{1}{n-j+1}\right) = \frac{k}{n-j+1}.$$

Hence

$$EX_i = \sum_{j=1}^{i} \frac{k}{n-j+1},$$

which is the same as the contribution of T_i under Proposal C.

14. Suppose you're designing strategies for selling items on a popular auction Web site. Unlike other auction sites, this one uses a *one-pass auction*, in which each bid must be immediately (and irrevocably) accepted or refused. Specifically,

    - First, a seller puts up an item for sale.
    - Then buyers appear in sequence.
    - When buyer i appears, he or she makes a bid $b_i > 0$.
    - The seller must decide immediately whether to accept the bid or not. If the seller accepts the bid, the item is sold and all future buyers are turned away. If the seller rejects the bid, buyer i departs and the bid is withdrawn; and only then does the seller see any future buyers.

    Suppose an item is offered for sale, and there are n buyers, each with a distinct bid. Suppose further that the buyers appear in a random order, and that the seller knows the number n of buyers. We'd like to design a *strategy* whereby the seller has a reasonable chance of accepting the highest of the n bids. By a "strategy," we mean a rule by which the seller decides whether to accept each presented bid, based only on the value of n and the sequence of bids seen so far.

    For example, the seller could always accept the first bid presented. This results in the seller accepting the highest of the n bids with probability only $1/n$, since it requires the highest bid to be the first one presented.

    Give a strategy under which the seller accepts the highest of the n bids with probability at least $1/4$, regardless of the value of n. (For simplicity, you are allowed to assume that n is an even number.) Prove that your strategy achieves this probabilistic guarantee.

    Solution.

    The strategy is as follows. The seller watches the first $n/2$ bids without accepting any of them. Let b^* be the highest bid among these. Then, in the final $n/2$ bids, the seller accepts any bid that is larger than b^*. (If there is no such bid, the seller simply accepts the final bid.)

Let b_i denote the highest bid, and b_j denote the second highest bid. Let S denote the underlying sample space, consisting of all permutations of the bids (since they can arrive in any order.) So $|S| = n!$. Let E denote the event that b_j occurs among the first $n/2$ bids, and b_i occurs among the final $n/2$ bids.

What is $|E|$? We can place b_j anywhere among the first $n/2$ bids ($n/2$ choices); then we can place b_i anywhere among the final $n/2$ bids ($n/2$ choices); and then we can order the remaining bids arbitrarily ($(n-2)!$ choices). Thus $|E| = \frac{1}{4}n^2(n-2)!$, and so

$$P[E] = \frac{n^2(n-2)!}{4n!} = \frac{n}{4(n-1)} \geq \frac{1}{4}.$$

Finally, if event E happens, then the strategy will accept the highest bid; so the highest bid is accepted with probability at least 1/4.

15. Suppose you are presented with a very large set S of real numbers, and you'd like to approximate the median of these numbers by sampling. You may assume all the numbers in S are distinct. Let $n = |S|$; we will say that a number x is an *ε-approximate median* of S if at least $(\frac{1}{2} - \varepsilon)n$ numbers in S are less than x, and at least $(\frac{1}{2} - \varepsilon)n$ numbers in S are greater than x.

    Consider an algorithm that works as follows. You select a subset $S' \subseteq S$ uniformly at random, compute the median of S', and return this as an approximate median of S. Show that there is an absolute constant c, independent of n, so that if you apply this algorithm with a sample S' of size c, then with probability at least .99, the number returned will be a (.05)-approximate median of S. (You may consider either the version of the algorithm that constructs S' by sampling with replacement, so that an element of S can be selected multiple times, or without replacement.)

    Solution. We imagine dividing the set S into 20 *quantiles* $Q_1, \ldots, Q_{20}$, where Q_i consists of all elements that have at least $.05(i-1)n$ elements less than them, and at least $.05(20-i)n$ elements greater than them. Choosing the sample S' is like throwing a set of numbers at random into bins labeled with $Q_1, \ldots, Q_{20}$.

    Suppose we choose $|S'| = 40,000$ and sample with replacement. Consider the event $\mathcal{E}$ that $|S' \cap Q_i|$ is between 1800 and 2200 for each i. If $\mathcal{E}$ occurs, then the first nine quantiles contain at most 19,800 elements of S', and the last nine quantiles do as well. Hence the median of S' will belong to $Q_{10} \cup Q_{11}$, and thus will be a (.05)-approximate median of S.

    The probability that a given Q_i contains more than 2200 elements can be computed using the Chernoff bound (4.1), with $\mu = 2000$ and $\delta = .1$; it is less than

$$\left[\frac{e^{.05}}{(1.05)^{(1.05)}}\right]^{10000} < .0001.$$

The probability that a given Q_i contains fewer than 1800 elements can be computed using the Chernoff bound (4.2), with $\mu = 2000$ and $\delta = .1$; it is less than

$$e^{-(.5)(.1)(.1)2000} < .0001.$$

Applying the Union Bound over the 20 choices of i, the probability that $\mathcal{E}$ does not occur is at most $(40)(.0001) = .004 < .01$.

16. Consider the following simple model of gambling in the presence of bad odds. At the beginning, your net profit is 0. You play for a sequence of n rounds; and in each round, your net profit increases by 1 with probability $1/3$, and decreases by 1 with probability $2/3$.

    Show that the expected number of steps in which your net profit is positive can be upper-bounded by an absolute constant, independent of the value of n.

    Solution. Let Y denote the number of steps in which your net profit is positive. Then $Y = Y_1 + Y_2 + \cdots + Y_n$, where $Y_k = 1$ if your net profit is positive at step k, and 0 otherwise.

    Now, consider a particular step k. $Y_k = 1$ if and only if you have had more than $k/2$ steps in which your profit increased. Since the expected number of steps in which your profit increased is $k/3$, we can apply the Chernoff bound (4.1) with $\mu = k/3$ and $1 + \delta = 3/2$ to conclude that EY_k is bounded by

    $$\left[\frac{e^{1/2}}{(3/2)^{(3/2)}}\right]^{(k/3)} < (.97)^k.$$

    Thus,

    $$EY = \sum_{k=1}^{n} EY_k < \sum_{k=1}^{n} (.97)^k < \frac{1}{1-(.97)} < 34,$$

    which is a constant independent of n.

17. Consider a balls and bins experiment with $2n$ balls but only 2 bins. As usual, each ball independently selects one of the two bins, both bins equally likely. The expected number of balls in each bin is n. In this problem we explore the question of how big their difference is likely to be. Let X_1 and X_2 denote the number of balls in the two bins respectively. (X_1 and X_2 are random variables.) Prove that for any $\varepsilon > 0$ there is a constant $c > 0$ such that the probability $\Pr[X_1 - X_2 > c\sqrt{n}] \le \varepsilon$.

18. Consider the following (partially specified) method for transmitting a message securely between a sender and a receiver. The message will be represented as a string of bits. Let $\Sigma = \{0, 1\}$, and let Σ^* denote the set of all strings of 0 or more bits. (E.g. $0, 00, 1110001 \in \Sigma^*$. The "empty string", with no bits, will be denoted $\lambda \in \Sigma^*$.)

    The sender and receiver share a secret function $f : \Sigma^* \times \Sigma \to \Sigma$. That is, f takes a word and a bit, and returns a bit. When the receiver gets a sequence of bits $\alpha \in \Sigma^*$, he/she runs the following method to decipher it:

```
Let $\alpha = \alpha_1\alpha_2\cdots\alpha_n$, where $n$ is the number of bits in $\alpha$. The goal is to produce an $n$-bit deciphered message, denoted $\beta = \beta_1\beta_2\cdots\beta_n$.
Set $\beta_1 := f(\lambda, \alpha_1)$.
For $i = 2, 3, 4, \ldots, n$
    Set $\beta_i := f(\beta_1\beta_2\cdots\beta_{i-1}, \alpha_i)$.
End for
Output $\beta$.
```

One could view this is as type of "stream cipher with feedback." One problem with this approach is that if any bit α_i gets corrupted in transmission, it will corrupt the computed value of β_j for all $j \geq i$.

We consider the following problem. A sender S wants to transmit the same (plain-text) message β to each of k receivers $R_1, \ldots, R_k$. With each one, he shares a different secret function $f^{\langle i\rangle}$. Thus, he sends a different encrypted message $\alpha^{\langle i\rangle}$ to each receiver, so that $\alpha^{\langle i\rangle}$ decrypts to β when the above algorithm is run with the function $f^{\langle i\rangle}$.

Unfortunately, the communication channels are very noisy, so each of the n bits in each of the k transmissions is *independently* corrupted (i.e. flipped to its complement) with probability 1/4. Thus, no single receiver on his/her own is likely to be able to decrypt the message correctly. Show, however, that if k is large enough as a function of n, then the k receivers can jointly reconstruct the plain-text message in the following way. They get together, and without revealing any of the $\alpha^{\langle i\rangle}$ or the $f^{\langle i\rangle}$, they interactively run an algorithm that will produce the correct β with probability at least 9/10. (How large do you need k to be in your algorithm?)

Solution. One algorithm is the following.

```
For $i = 1, 2, \ldots, n$
    Receiver $j$ computes $\beta_{ij} = f(\beta^*_1\cdots\beta^*_{i-1}, \alpha^{\langle j\rangle}_i)$.
    $\beta^*_i$ is set to the majority value of $\beta_{ij}$, for $j = 1, \ldots, k$.
End for
Output $\beta^*$
```

We'll make sure to choose an odd value of k to prevent ties.

Let $X_{ij} = 1$ if $\alpha^{\langle j\rangle}_i$ was corrupted, and 0 otherwise. If a majority of the bits in $\{\alpha^{\langle j\rangle}_i : j = 1, 2, \ldots, k\}$ are corrupted, then $X_i = \sum_j X_{ij} > k/2$. Now, since each bit is corrupted with probability $\frac{1}{4}$, $\mu = \sum_j EX_{ij} = k/4$. Thus, by the Chernoff bound, we have

$$\begin{aligned}\Pr[X_i > k/2] &= \Pr[X_i > 2\mu] \\ &< \left(\frac{e}{4}\right)^{k/4} \\ &\leq (.91)^k.\end{aligned}$$

Now, if

$$k \geq 11\ln n > \frac{\ln n - \ln .1}{\ln(1/.91)},$$

then

$$\Pr[X_i > k/2] < .1/n.$$

(So it is enough to choose k to be the smallest odd integer greater than $11 \ln n$.) Thus, by the union bound, the probability that *any* of the sets $\{\alpha_i^{\langle j \rangle} : j = 1, 2, \ldots, k\}$ have a majority of corruptions is at most .1.

Assuming that a majority of the bits in each of these sets are not corrupted, which happens with probability at least .9, one can prove by induction on i that all the bits in the reconstructed message β^* will be correct.

19. Some people designing parallel physical simulations come to you with the following problem. They have a set P of k *basic processes*, and want to assign each process to run on one of two machines, M_1 and M_2. They are then going to run a sequence of n *jobs*, $J_1, \ldots, J_n$. Each job J_i is represented by a set $P_i \subseteq P$ of exactly $2n$ basic processes which must be running (each on its assigned machine) while the job is processed. An assignment of basic processes to machines will be called *perfectly balanced* if, for each job J_i, exactly n of the basic processes associated with J_i have been assigned to each of the two machines. An assignment of basic processes to machines will be called *nearly balanced* if, for each job J_i, no more than $\frac{4}{3}n$ of the basic processes associated with J_i have been assigned to the same machine.

    (a) Show that for arbitrarily large values of n, there exist sequences of jobs $J_1, \ldots, J_n$ for which no perfectly balanced assignment exists.

    (b) Suppose that $n \geq 200$. Give an algorithm that takes an arbitrary sequence of jobs $J_1, \ldots, J_n$ and produces a nearly balanced assignment of basic processes to machines. Your algorithm may be randomized, in which case its expected running time should be polynomial, and it should always produce the correct answer.

    Solution. **(a)** Let n be odd, $k = n^2$, and reprsent the set of basic processes as the disjoint union of n sets $X_1, \ldots, X_n$ of cardinality n each. The set of processes P_i associated with job J_i will be equal to $X_i \cup X_{i+1}$, addition taken modulo n.

    We claim there is no perfectly balanced assignment of processes to machines. For suppose there were, and let Δ_i denote the number of processes in X_i assigned to machine M_1 minus the number of processes in X_i assigned to machine M_2. By the perfect balance property, we have $\Delta_{i+1} = -\Delta_i$ for each i; applying these equalities transitively, we obtain $\Delta_i = -\Delta_i$, and hence $\Delta_i = 0$, for each i. But this is not possible since n is odd.

    (b) Consider independently assigning each process i a *label* L_i equal to either 0 or 1, chosen uniformly at random. Thus we may view the label L_i as a 0-1 random variable. Now for any job J_i, we assign each process in P_i to machine M_1 if its label is 0, and machine M_2 if its label is 1.

    Consider the event E_i, that more than $\frac{4}{3}n$ of the processes associated with J_i end up on the same machine. The assignment will be nearly balanced if none of the E_i happen. E_i is precisely the event that $\sum_{t \in J_i} L_t$ either exceeds $\frac{4}{3}$ times its mean (equal to n), or that it falls below $\frac{2}{3}$ times its mean. Thus, we may upper-bound the probability of E_i

as follows.

$$\begin{aligned}
\Pr[E_i] &\le \Pr[\sum_{t \in J_i} L_t < \frac{2}{3}n] + \Pr[\sum_{t \in J_i} L_t > \frac{4}{3}n] \\
&\le \left(e^{-\frac{1}{2}(\frac{1}{3})^2}\right)^n + \left(\frac{e^{\frac{1}{3}}}{\left(\frac{4}{3}\right)^{\frac{4}{3}}}\right)^n \\
&\le 2 \cdot .96^n.
\end{aligned}$$

Thus, by the union bound, the probability that any of the events E_i happens is at most $2n \cdot .96^n$, which is at most .06 for $n \ge 200$.

Thus, our randomized algorithm is as follows. We perform a random allocation of each process to a machine as above, check if the resulting assignment is perfectly balanced, and repeat this process if it isn't. Each iteration takes polynomial time, and the expected number of iterations is simply the expected waiting time for an event of probability $1 - .06 = .94$, which is $1/.94 < 2$. Thus the expected running time is polynomial.

This analysis also proves the *existence* of a nearly balanced allocation for any set of jobs.

(Note that the algorithm can run forever, with probability 0. This doesn't cause a problem for the expectation, but we can deterministically guarantee termination without hurting the running time very much as follows. We first run k iterations of the randomized algorithm; if it still hasn't halted, we now find the nearly balanced assignment that is guaranteed to exist by trying all 2^k possible allocations of processes to machines, in time $O(n^2 \cdot 2^k)$. Since this brute-force step occurs with probability at most $.06^k$, it adds at most $O(n^2 \cdot .12^k) = O(n^2 \cdot .12^n) = o(1)$ to the expected running time.)