5.1

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=⌈½n⌉
   if A(a + k)<B(b + k)
     then return median (k, a + ⌊½n⌋ ,b)
     else return median (k, a, b + ⌊½n⌋)
```

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 $Q(n) - 2\lceil \log n \rceil$.

A final note. In order to prove this algorithm correct, note that it is not enough to prove simply that, in the recursive call, the median remains in the set of numbers considered; one must prove the stronger statement that the median value in the recursive call will in fact be the same as the median value in the original call. Also, 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

[1] ex132.487.812

"deletion" must in fact be implemented by keeping track of a particular interval under consideration in each database.

5.2

We'll define a recursive divide-and-conquer algorithm ALG which takes a sequence of distinct numbers $a_1, \ldots, a_n$ and returns N and $a'_1, \ldots, a'_n$ where

- N is the number of significant inversions
- $a'_1, \ldots, a'_n$ is same sequence sorted in the increasing order

ALG is similar to the algorithm from the chapter that computes the number of inversions. The difference is that in the 'conquer' step we merge twice: first we merge $b_1, \ldots, b_k$ with $b_{k+1}, \ldots, b_n$ just for sorting, and then we merge $b_1, \ldots, b_k$ with $2b_{k+1}, \ldots, 2b_n$ for counting significant inversions.

Let's define ALG formally. For $n = 1$ ALG just returns $N = 0$ and $\{a_1\}$ for the sequence. For $n > 1$ ALG does the following:

- let $k = \lfloor n/2 \rfloor$.
- call $\mathsf{ALG}(a'_1, \ldots, a'_k)$. Say it returns N_1 and $b_1, \ldots, b_k$.
- call $\mathsf{ALG}(a'_{k+1}, \ldots, a'_n)$. Say it returns N_2 and $b_{k+1}, \ldots, b_n$.
- compute the number N_3 of significant inversions (a_i, a_j) where $i \le k < j$.
- return $N - N_1 + N_2 + N_3$ and $a'_1, \ldots, a'_n - \mathsf{MERGE}(b_1, \ldots, b_k;\ b_{k+1}, \ldots, b_n)$

MERGE can be implemented in $O(n)$ time. According to the discussion in the book, it remains to find a way to compute N_3 in $O(n)$ time. We implement a variant of merge-count of $b_1, \ldots, b_k$ and $2\,b_{k+1}, \ldots, 2\,b_n$ as follows.

- Initialize counters: $i \leftarrow k$, $j \leftarrow n$, $N_3 \leftarrow 0$.
- If $b_i \le 2b_j$ then
  - if $j > k + 1$ decrease j by 1.
  - if $j = k + 1$ return N_3.
- If $b_i > 2b_j$ then increase N_3 by $j - k$. Then
  - if $i > 1$ decrease i by 1.
  - if $i = 1$ return N_3.

Explanation For every i we count the number of significant inversions between b_i and all b_j's. If $b_i \le 2b_j$ then there are no significant inversions between b_i and any b_m s.t. $m \ge j$, so we decrease j. If $b_i > 2b_j$ then $b_i > 2b_m$ for all m s.t. $k < m \le j$. In other words, we have detected $j - k$ significant inversions involving b_i. So we increase N_3 by $j - k$. Finally, when we are down to $i = 1$ and have counted significant inversions involving b_1, there are no more significant inversions to be detected.

[1] ex499.218.598

5.3

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 ⌊n/2⌋ 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 the following recurrence (assuming n is divisible by 2):

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

[1] ex628.974.324

As we have seen in the chapter, 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.

5.4

This can be accomplished directly using a convolution. Define one vector to be $a = (q_1, q_2, \ldots, q_n)$. Define the other vector to be $b = (n^{-2}, (n-1)^{-2}, ..., 1/4, 1, 0, -1, -1/4, ... - n^{-2})$. Now, for each j, the convolution of a and b will contain an entry of the form

$$\sum_{i<j} \frac{q_i}{(j-i)^2} + \sum_{i>j} \frac{-q_i}{(j-i)^2}.$$

From this term, we simply multiply by Cq_j to get the desired net force F_j.

The convolution can be computed in $O(n \log n)$ time, and reconstructing the terms F_j takes an additional $O(n)$ time.

[1] ex726.26.783

5.5

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 points, in $O(n)$ time. (Note that $p + q \leq n$, 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.

[1] ex428.913.582

5.6

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).

[1] ex739.448.876

5.7

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.

[1] ex624.352.598

6.1

(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 $O(n)$.

[1] ex301.516.319

6.2

(a) This algorithm is too short-sighted; it might take a high-stress job too early and an even better one later.

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

The algorithm in (a) would take a high-stress job in week 2, when the unique optimal solution would take a low-stress job in week 1, nothing in week 2, and then a high-stress job in week 3.

(b) Let $OPT(i)$ denote the maximum value revenue achievable in the input instance restricted to weeks 1 through i. The optimal solution for the input instance restricted to weeks 1 through i will select *some* job in week i, since it's not worth skipping all jobs — there are no future high-stress jobs to prepare for. If it selects a low-stress job, it can behave optimally up to week $i-1$, followed by this job, while if it selects a high-stress job, it can behave optimally up to week $i-2$, followed by this job. Thus we have justified the following recurrence.

$$OPT(i) = \max(\ell_i + OPT(i-1), h_i + OPT(i-2)).$$

We can compute all OPT values by invoking this recurrence for $i = 1, 2, \ldots, n$, with the initialization $OPT(1) = \max(\ell_1, h_1)$. This takes constant time for each value of i, for a total time of $O(n)$. As usual, the actual sequence of jobs can be reconstructed by tracing back through the set of OPT values.

An alternate, but essentially equivalent, solution is as follows. We define the following sub-problems. Let $L(i)$ be the maximum revenue achievable in weeks 1 through i, given that you select a low-stress job in week i, and let $H(i)$ be the maximum revenue achievable in weeks 1 through i, given that you select a high-stress job in week i.

Again, the optimal solution for the input instance restricted to weeks 1 through i will select some job in week i. Now, if it selects a low-stress job in week i, it can select anything it wants in week $i-1$; and if it selects a high-stress job in week i, it has to sit out week $i-1$ but can select anything it wants in week $i-2$. Thus we have

$$L(i) = \ell_i + \max(L(i-1), H(i-1)),$$

$$H(i) - h_i + \max(L(i-2), H(i-2)).$$

The L and H values can be built up by invoking these recurrences for $i = 1, 2, \ldots, n$, with the initializations $L(1) = \ell_1$ and $H_1 = h_1$.

[1] ex695.414.330

6.3

(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...n]
  M[1] = 0
  For i = 2,...,n
     M = -∞
     For all edges (j,i) then
        if M[j] ≠ -∞
             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.

[1] ex961.606.761

6.4

(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 $O(n)$.

[1] ex786.93.190

6.5

The key observation to make in this problem is that if the segmentation $y_1 y_2 \dots y_n$ is an optimal one for the string y, then the segmentation $y_1 y_2 \dots y_{n-1}$ would be an optimal segmentation for the prefix of y that excludes y_n (because otherwise we could substitute the optimal solution for the prefix in the original problem and get a better solution).

Given this observation, we design the subproblems as follows. Let $Opt(i)$ be the score of the best segmentation of the prefix consisting of the first i characters of y. We claim that the recurrence

$$Opt(i) = min_{j \leq i}\{Opt(j-1) + Quality(j \dots n)\}$$

would give us the correct optimal segmentation (where $Quality(\alpha \dots \beta)$ means the quality of the word that is formed by the characters starting from position α and ending in position β). Notice that the desired solution is $Opt(n)$.

We prove the correctness of the above formula by induction on the index i. The base case is trivial, since there is only one word with one letter.

For the inductive step, assume that we know that the Opt function as written above finds the optimum solution for the indices less than i, and we want to show that the value $Opt(i)$ is the optimum cost of any segmentation for the prefix of y up to the i-th character. We consider the last word in the optimal segmentation of this prefix. Let's assume it starts at index $j \leq i$. Then according to our key observation above, the prefix containing only the first $j-1$ characters must also be optimal. But according to our induction hypothesis, $Opt(j)$ will yield us the value of the aforementioned optimal segmentation. Therefore the optimal cost $Opt(i)$ would be equal to $Opt(j)$ plus the cost of the last word.

But notice that our above recurrence exactly does this calculation for each possibility of the last word. Therefore our recurrence will correctly find the cost of the optimal segmentation.

As for the running time, a simple implementation (direct evaluation of the above formula starting at index 1 until n, where n is the number of characters in the input string) will yield a quadratic algorithm.

[1] ex931.924.160

6.6

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

[1] ex771.275.715

6.7

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.

[1] ex244.420.389

6.8

(a) Change x_4 to 2 in the given example. Then this algorithm would activate the EMP at times 2 and 4, for a total of 4 destroyed; but activating at times 3 and 4 as before still gets 5.

(b) Let $OPT(j)$ be the maximum number of robots that can be destroyed for the instance of the problem just on $x_1, \ldots, x_j$. Clearly if the input ends at x_j, there is no reason not to activate the EMP then (you're not saving it for anything), so the choice is just when to last activate it before step j. Thus $OPT(j)$ is the best of these choices over all i:

$$OPT(j) = \max_{0 \le i < j} [OPT(i) + \min(x_j, f(j-i))],$$

where $OPT(0) = 0$. The full algorithm is just

```
Set OPT(0) = 0
For i − 1, 2, ..., n
  Compute OPT(j) using the recurrence
Endfor
Return OPT(n).
```

The running time is $O(n)$ per iteration, for a total of $O(n^2)$.

An alternate solution would define $OPT'(j,k)$ to be the best solution for steps j through n, given that the EMP in step j has already been charging for k steps. The optimal way to solve this sub-problem would be to either activate the EMP in step j or not, and $OPT'(j,k)$ is just the better of these two choices:

$$OPT'(j,k) = \max(\min(x_j, f(k)) + OPT'(j+1, 1), OPT'(j+1, k+1)).$$

We initialize $OPT'(n,k) = \min(x_n, f(k))$ for all k, and the full algorithm is

```
Set OPT'(n, k) = min(x_n, f(k)) for all k.
For j = n − 1, n − 2, ..., 1
  For k = 1, 2, ..., j
    Compute OPT'(j, k) using the recurrence
  Endfor
Endfor
Return OPT'(1, 1).
```

The running time is $O(1)$ per entry of OPT', for a total of $O(n^2)$.

[1] ex249.233.474

6.9

(a) Supopose $s_1 = 10$ and $s_i = 1$ for all $i > 1$; and $x_i = 11$ for all i. Then the optimal solution should re-boot in every other day, thereby processing 10 terabytes every two days.

(b) This problem has quite a few correct dynamic programming solutions; we describe several of the more natural ones here.

1. Let $\mathsf{Opt}(i, j)$ denote the maximum amount of work that can be done starting from day i through day n, given the last reboot occurred j days prior, i.e., the system was rebooted on day $i - j$.

   On each day, there are two options:

   - *Reboot*: which means you don't process anything on day i and day $i + 1$ is the first day after the reboot. Hence, the optimal solution in this case is

     $$\mathsf{Opt}(i, j) - \mathsf{Opt}(i + 1, 1).$$

   - *Continue Processing*: which means that on day i you process the minimum of x_i and s_j. Hence, the optimal solution in this case is

     $$\mathsf{Opt}(i, j) = min\{x_i, s_j\} + \mathsf{Opt}(i + 1, j + 1).$$

   On the last day, there is no advantage gained in rebooting and hence

   $$\mathsf{Opt}(n, j) = min\{x_n, s_j\}$$

   The Algorithm:

   Set $\mathsf{Opt}(n, j) = min\{x_n, s_j\}$, for all j from 1 to n
   for $i = n - 1$ downto 1
   for $j = 1$ to i
   $\mathsf{Opt}(i, j) = max\{\ \mathsf{Opt}(i + 1, 1),\ min\{x_i, s_j\} + \mathsf{Opt}(i + 1, j + 1)\}$
   endfor$_j$
   endfor$_i$
   return Opt(1,1)

   Running Time: Note that the *max* is over only 2 values and hence is a constant time operation. Since there are only $O(n^2)$ values being calculated, and each one takes $O(1)$ time to calculate, the algorithm takes $O(n^2)$ time.

2. Let $\mathsf{Opt}(i, j)$ to be the maximum number of terabytes that can be processed from days 1 to i, given that the last reboot occurred j days prior to the current day.

   When $j > 0$ (i.e., the system is not rebooted on day i), $min\{x_i, s_j\}$ terabytes are processed and hence,

   $$\mathsf{Opt}(i, j) = \mathsf{Opt}(i - 1, j - 1) + min\{x_i, s_j\}$$

[1] ex736.816.103

When $j = 0$ (i.e., the system is rebooted on day i), no processing is done on day i. Also, the previous reboot could have happened on any of the days prior to day i. Hence,

$$\mathsf{Opt}(i, 0) = max_{k=1}^{i-1}\{\mathsf{Opt}(i-1, k)\}$$

Strictly speaking k should run from 0 to $i-1$, i.e., the last reboot could have happened either on day $i-1$ or on day $i-2$ and so on ... or on day 0 (which means no previous reboot). In our case, however, it is not advantageous to reboot on 2 successive days – you might as well do some computation on the first day and reboot on the second day. Since there is a reboot on day i, we can be sure that there is no reboot on day $i-1$, and hence k starts from 1.

The base case for the recursion is:

$$\mathsf{Opt}(0, j) = 0, \forall j = 0, 1, \ldots, n$$

A simple algorithm calculating the $\mathsf{Opt}(i, j)$ values can be designed as before taking care that i runs from 1 to n. The final value to be returned is $max_{j=1}^{n}\{\mathsf{Opt}(n, j)\}$.

Running Time: All $\mathsf{Opt}(i, j)$ values take $O(1)$ time when $j \neq 0$. $\mathsf{Opt}(i, 0)$ values take $O(n)$ time. Hence the algorithm runs in $n^2 \times O(1) + n \times O(n) = O(n^2)$ time.

3. Let $\mathsf{Opt}(i)$ denote the maximum number of bytes that can be processed starting from day 1 to day i. Suppose that the system was last rebooted on day $j < i$ ($j = 0$ means there was no reboot). Then since day $j+1$, the total number of bytes processed will be $b_{ji} = \Sigma_{k=1}^{i-j} min\{x_{j+k}, s_k\}$. (*Remember that there is no use rebooting on the last day.*) The total work processed till day i would then be $\mathsf{Opt}(j-1) + b_{ji}$. To get the maximum number of bytes processed, maximize over all values of $j < i$. Hence,

$$\mathsf{Opt}(i) = max_{j-0}^{i-1}\{\mathsf{Opt}(j)\} + b_{ji}$$

The base case:

$$\mathsf{Opt}(0) = 0$$

Compute $\mathsf{Opt}(i)$ values starting from $i = 1$ and return the value of $\mathsf{Opt}(n)$.

Running Time: Each b_{ji} value takes $O(n)$ time to calculate, and since there are $O(n^2)$ such values being calculated, the algorithm takes, $O(n^3)$ time.

6.10

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

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

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 $O(n^2)$.

[1] ex803.497.915

A common type of error is 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.

6.11

Let $OPT(i)$ denote the minimum cost of a solution for weeks 1 through i. In an optimal solution, we either use company A or company B for the i^{th} week. If we use company A, we pay rs_i and can behave optimally up through week $i-1$. If we use company B for week i, then we pay $4c$ for this contract, and so there's no reason not to get the full benefit of it by starting it at week $i-3$; thus we can behave optimally up through week $i-4$, and then invoke this contract.

Thus we have

$$OPT(i) = \min(rs_i + OPT(i-1), 4c + OPT(i-4)).$$

We can build up these OPT values in order of increasing i, spending constant time per iteration, with the initialization $OPT(i) = 0$ for $i \leq 0$.

The desired value is $OPT(n)$, and we can obtain the schedule by tracing back through the array of OPT values.

[1] ex382.12.857

6.12

Let $OPT(j)$ denote the minimum cost of a solution on servers 1 through j, *given* that we place a copy of the file at server j. We want to search over the possible places to put the highest copy of the file before j; say in the optimal solution this at position i. Then the cost for all servers up to i is $OPT(i)$ (since we behave optimally up to i), and the cost for servers $i+1, \ldots, j$ is the sum of the access costs for $i+1$ through j, which is $0 + 1 + \cdots + (j-i-1) = \binom{j-i}{2}$. We also pay c_j to place the server at j.

In the optimal solution, we should choose the best of these solutions over all i. Thus we have

$$OPT(j) = c_j + \min_{0 \le i < j} (OPT(i) + \binom{j-i}{2}),$$

with the initializations $OPT(0) = 0$ and $\binom{1}{2} = 0$. The values of OPT can be built up in order of increasing j, in time $O(j)$ for iteration j, leading to a total running time of $O(n^2)$. The value we want is $OPT(n)$, and the configuration can be found by tracing back through the array of OPT values.

[1] ex25.372.49

6.13

A useful way to analyze large products in cases like this is to take logarithms, which causes them to become sums. Thus, let us build a graph G with a node for each stock, and a directed edge (i, j) for each pair of stocks. We put a cost of $-\log r_{ij}$ on edge (i, j).

Now, a trading cycle C in G is an opportunity cycle if and only if

$$\prod_{(i,j)\in C} r_{ij} > 1,$$

in other words, taking logarithms of both sides, if and only if

$$\sum_{(i,j)\in C} \log r_{ij} > 0,$$

or

$$\sum_{(i,j)\in C} -\log r_{ij} < 0.$$

Thus, a trading cycle C in G is an opportunity cycle if and only if it is a negative cycle. Hence we can use our polynomial-time algorithm for negative-cycle detection to determine whether an opportunity cycle exists.

[1] ex181.273.949

6.14

(a) To do this, we build a graph H as follows. H has the same nodes as all the G_i, and it consists precisely of those edges that occur in every one of $G_0, \ldots, G_b$. In this graph H, we simply perform breadth-first search to find the shortest path from s to t (if any s-t path exists).

Note that this idea, generalized to any sequence of graphs $G_i, \ldots, G_j$, will be useful in part (b).

(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 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)ℓ(0,i); min_{1≤j<i} M[j]  (i    j)ℓ(j  1,i))
EndFor
```

[1] ex377.520.504

6.15

(a) Suppose that $n = 6$ and the coordinates are $3, 2, -3, -2, -1, 0$. Then the greedy algorithm would observe events $2, 5, 6$, while an optimal solution would observe events $3, 4, 5, 6$.

(b) Let $OPT(j)$ denote the maximum number of events that can be observed, subject to the constraint that event j is observed. Note that $OPT(n)$ is the value that we want.

To define a recurrence for $OPT(j)$, we consider the previous event before j that is observed in an optimal solution. If it is i, then we need to have $|d_j - d_i| \leq j - i$, and we behave optimally up through observing event i. Thus we have

$$OPT(j) = 1 + \min_{i:|d_j - d_i| \leq j-i} OPT(i).$$

The values of OPT can be built up in order of increasing j, in time $O(j)$ for iteration j, leading to a total running time of $O(n^2)$. The value we want is $OPT(n)$, and the configuration can be found by tracing back through the array of OPT values.

[1] ex259.807.630

6.16

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.

[1] ex449.390.867

6.17

(a) Consider the sequence $1, 4, 2, 3$. The greedy algorithm produces the rising trend $1, 4$, while the optimal solution is $1, 2, 3$.

(b) Let $OPT(j)$ be the length of the longest increasing subsequence on the set $P[j], P[j+1], ldots, P[n]$, including the element $P[j]$. Note that we can initialize $OPT(n) = 1$, and $OPT(1)$ is the length of the longest rising trend, as desired.

Now, consider a solution achieving $OPT(j)$. Its first element is $P[j]$, and its next element is $P[k]$ for some $k > j$ for which $P[k] > P[j]$. From k onward, it is simply the longest increasing subsequence that starts at $P[k]$; in other words, this part of the sequence has length $OPT(k)$, so including $P[j]$, the full sequence has length $1 + OPT(k)$. We have thus justified the following recurrence.

$$OPT(j) = 1 + \max_{k>j:P[k]>P[j]} OPT(k).$$

The values of OPT can be built up in order of decreasing j, in time $O(n-j)$ for iteration j, leading to a total running time of $O(n^2)$. The value we want is $OPT(1)$, and the subsequence itself can be found by tracing back through the array of OPT values.

[1] ex219.570.316

6.18

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 $c'(Q)$.

[1] ex485.507.165

6.19

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 $M[i,j] = yes$ 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:

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

We can build these up via the following loop.

```
M[0,0] = yes
For k = 1, 2, ..., n
  For all pairs (i, j) so that i + j = k
    If M[i − 1, j] = yes and s[i + j] = x'[i] then
      M[i, j] = yes
    Else if M[i, j − 1] − yes and s[i + j] − y'[j] then
      M[i, j] = yes
    Else
      M[i, j] − no
  Endfor
Endfor
Return ‘‘yes’’ if and only there is some pair (i, j) with i + j = n
  so that M[i, j] = 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 $O(n^2)$.

[1] ex357.417.692

6.20

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

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

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 $O(nH^2)$.

[1] ex680.762.178

6.21

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

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

[1] ex541.91.349

6.22

Let c_e denote the cost of the edge e and we will overload the notation and write c_{st} to denote the cost of the edge between the nodes s and t.

This problem is by its nature quite similar to the shortest path problem. Let us consider a two-parameter function $Opt(i, s)$ denoting the optimal cost of shortest path to s using *exactly* i edges, and let $N(i, s)$ denote the number of such paths.

We start by setting $Opt(i, v) = 0$ and $Opt(i, v') = \infty$ for all $v' \neq v$. Also set $N(i, v) = 1$ and $N(i, v') = 0$ for all $v' \neq v$. Intuitively this means that the source v is reachable with cost 0 and there is currently one path to achieve this.

Then we compute the following recurrence:

$$Opt(i, s) = min_{t,(t,s)\in E}\{Opt(i-1, t) + c_{ts}\}. \tag{1}$$

The above recurrence means that in order to travel to node s using exactly i edges, we must travel a predecessor node t using exactly $i-1$ edges and then take the edge connecting t to s. Once of course the optimal cost value has been computed, the number of paths that achieve this optimum would be computed by the following recurrence:

$$N(i, s) = \sum_{t,(t,s)\in E \text{ and } Opt(i,s)=Opt(i-1,t)+c_{ts}} N(i-1, t). \tag{2}$$

In other words, we look at all the predecessors from which the optimal cost path may be achieved and add all the counters.

The above recurrences can be calculated by a double loop, where the outside loops over i and the inside loops over all the possible nodes s. Once the recurrences have been solved, our target optimal path to w is obtained by taking the minimum of all the paths of different lengths to w - that is:

$$Opt(w) = min_i\{Opt(i, w)\}. \tag{3}$$

And the number of such paths can be computed by adding up the counters of all the paths which achieve the minimal cost.

$$N(w) = \sum_{i,Opt(i,w)=Opt(w)} N(i, w). \tag{4}$$

[1] ex720.859.203

6.23

(a) The following lgorithm checks the validity of the given $d(v)$s in $O(m)$ time: If for any edge $e = (v, w)$, we have that $d(v) > d(w) + c_e$, then we can immediately reject. This follows since if $d(w)$ is correct, then there is a path from v to t via w of cost $d(w) + c_e$. The minimum distance from v to t is at most this value, so $d(v)$ must be incorrect. Now consider the graph G' formed by taking G and removing all edges except those $e = (v, w)$ for which $d(v) = d(w) + c_e$. We will now check that every node has a path to t in this new graph (this can easily be checked in $O(m)$ time by reversing all edges and doing a DFS or BFS). If any node fails to have such a path, reject. Otherwise accept. Observe that if $d(v)$ were correct for all nodes v, then if we consider those edges on the shortest path from any node v to t, these edges will all be in G'. Therefore we can safely reject if any node can not reach t in G'.

So far we have argued that when our algorithm rejects a set of distances, those distances must have been incorrect. We now need to show that if our algorithm accepts, then the distances are correct. Let $d'(v)$ be the actual distance in G from v to t. We want to show that $d'(v) - d(v)$ for all v. Consider the path found by our algorithm in G' from v to t. The real cost of this path is exactly $d(v)$. There may be shorter paths, but we know that $d'(v) \leq d(v)$, and this holds for all v. Now suppose that there is some v for which $d'(v) < d(v)$. Consider the actual shortest path P from v to t. Let x be the last on P for which $d'(x) < d(x)$. Let y be the node after x on P. By our choice of x, $d'(y) = d(y)$. Since P is the real shortest path to t from v, it is also the real shortest path from all nodes on P. So $d'(x) - d'(y) + c_e$ where $e - (x, y)$. This implies that $d(x) > d'(x) - d(y) + c_e$. But then we have a contradiction, since our algorithm would have rejected upon inspection of e. Hence $d'(v) = d(v)$ for all v, so the claim is proved.

The same solution can also be phrased in terms of the modified cost function given in part (b).

There are two common mistakes. One is to simply use the algorithm that just checks for $d(v) > d(w) + c_e$. This can be broken if the graph has a cycle of cost 0, as the nodes on such a cycle may be self consistent, but may have a much larger distance to the root than reported. Consider two nodes x and y connected to each other with 0 cost edges, and connected to t by an edge of cost 5. If we report that both of these are at distance 3 from the root, the algorithm faulty will accept the distances. The other common mistake was to fail to prove that the algorithm is correct on both yes and no instances of the problem.

(b) Given distances to a sink t, we can efficiently calculate distance to another sink t' in $O(m \log n)$ time. To do so, note that if only edge costs were non-negative, then we could just run Dijkstra's algorithm. We will modify costs to ensure this, without changing the min cost paths. Consider modifying the cost of each edge $e = (v, w)$ as follows: $c'_e = c_e - d(v) + d(w)$. First observe that the modiffied costs are non-negative, since if $c'_e < 0$ then $d(v) < d(w) + c_e$ which is not possible if d reflect true distances to t. Now consider any path P from some node x to t'. The real cost of P is $c(P) = \sum_{e \in P} c_e$. The modiffied cost of P is $c'(P) = \sum_{e \in P} c'e = \sum_{e \in P} c_e - d(v) + d(w)$. Note that all but the first and last node in P occur once positively and once negatively in this sum, so we get that $c'(P) - c(P) - d(x) + d(t')$. Hence

[1] ex738.372.857

the modiffied cost of any path from x to t' differs from the real cost of that same path an additive $d(t') - d(x)$, a constant. So the set of minimum cost paths from x to t' under c' is the same as the set under c. Furthermore, given the min distance from x to t' under c', we can calculate the min distance under c by simply adding $d(x) - d(t')$. Our algorithm is exactly that: calculate the modiffied costs, run Dijkstra's algorithm from t' (edges need to be reversed for the standard implementation), and then adjust the distances as described. This takes $O(m + m \log n + n) = O(m \log n)$ time.

6.24

The basic idea is to ask: How should we gerrymander precincts 1 through j, for each j? To make this work, though, we have to keep track of a few extra things, by adding some variables. For brevity, we say that the A-votes in a precinct are the votes for party A, and B-votes are the votes for party B. We keep track of the following information about a partial solution.

- How many precincts have been assigned to district 1 so far?
- How many A-votes are in district 1 so far?
- how many A-votes are in district 2 so far?

So let $M[j, p, x, y] = true$ if it is possible to achieve at least x A-votes in district 1 and y A-votes in district 2, while allocating p of the first j precincts to district 1. ($M[j, p, x, y] - false$ otherwise.) Now suppose precinct $j + 1$ has z A-votes. To compute $M[j + 1, p, x, y]$, you either put precinct $j + 1$ in district 1 (in which case you check the results of sub-problem $M[j, p - 1, x - z, y]$) or in precinct 2 (in which case you check the results of sub-problem $M[j, p, x, y - z]$). Now to decide if there's a solution to the whole problem, you scan the entire table at the end, looking for a value of "true" in any entry of the form $M[n, n/2, x, y]$, where each of x and y is greater than $mn/4$. (Since each district gets $mn/2$ votes total.)

We can build this up in order of increasing j, and each sub-problem takes constant time to compute, using the values of smaller sub-problems. Since there are n^2m^2 sub-problems, the running time is $O(n^2m^2)$.

[1] ex706.269.18

6.25

We define subproblems involving some of the last i days. Of course, having $Opt(i)$ denote the optimal division of stocks for the days i through n is hard, since we don't know how many stocks are available on the i-th day. So the logical continuation is to think of a two-parameter function $Opt(i, k)$ indicating the optimal division profit of k stocks starting from day i.

Here are two key observations to notice. First, from the i-th day on, our selling strategy does not depend on how selling was conducted on previous days because each of the remaining k stocks will incur the same *constant* penalty in profit due to the accumulated value of the function f from previous sales. So our total profit will just decrease by k times that constant.

The next observation is that if we decide to sell y stocks on i-th day, then all the k stocks will incur a profit loss of $f(y)$ because of this particular decision regardless of when they're sold.

Taking these two observations into account we are now ready to give the recurrence for $Opt(i, k)$. It is

$$Opt(i, k) = min_{y<=k}\{p_i y + Opt(i + 1, k - y) - kf(y)\}.$$

The first term of the above recurrence is the direct profit from selling y stocks with price p_i, the middle term is the optimal profit for selling the remaining $k \quad y$ stocks starting from day $i + 1$ (according to the first observation the optimal profit itself might differ from $Opt(i + 1, k - y)$ because of prior sales, but it will be within a fixed constant and therefore the choice of y minimizing the above recurrence will be unchanged), and finally the last term is due to the second observation above - the total profit loss incurred by the decision to sell y stocks on the i-th day on the remaining stocks.

An implementation would create a two dimensional table M, whose rows would stand for the days and the columns for the stocks to sell. The matrix would be filled using the above recurrence from the bottom row to the top one. The result is a cubic algorithm. Our desired goal is $Opt(1, x)$. The matrices could also keep track of how many we picked on i-th day in the optimal solution, and that will be our output.

Note that the running time of this algorithm polynomially depends on x, the total number of stocks. Note only is this permitted by the statement of the problem, but it is also polynomial in the input size, the values of the input function f are given as a set of x values $f(1), \ldots, f(x)$.

[1] ex571.682.218

6.26

The subproblems will represent the optimum way to satisfy orders $1, \ldots, i$ with an inventory of s trucks 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 $OPT(i \quad 1, s \mid d_i) \mid C(s{+}d_i)$ (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 $OPT(i, s) = \min(OPT(i - 1, s + d_i) + C(s + d_i), OPT(i - 1, 0) + K)$.

[1] ex304.359.690

6.27

A solution is specified by the days on which orders of gas arrive, and the amounts of gas that arrives on those days. In computing the total cost, we will take into account delivery and storage costs, but we can ignore the cost for buying the actual gas, since this is the same in all solutions. (At least all those where all the gas is exactly used up.)

Consider an optimal solution. It must have an order arrive on day 1, and it the next order is due to arrive on day i, then the amount ordered should be $\sum_{j=1}^{i-1} g_j$. Moreover, the capacity requirements on the storage tank say that i must be chosen so that $\sum_{j=1}^{i-1} g_j \leq L$.

What is the cost of storing this first order of gas? We pay g_2 to store the g_2 gallons for one day until day 2, and $2g_3$ to store the g_3 gallons for two days until day 3, and so forth, for a total of $\sum_{j=1}^{i-1}(j-1)g_j$. More generally, the cost to store an order of gas that arrives on day a and lasts through day b is $\sum_{j=a}^{b}(j \quad a)g_j$. Let us denote this quantity by $S(a,b)$.

Let $OPT(a)$ denote the optimal solution for days a through n, assuming that the tank is empty at the start of day a, and an order is arriving. We choose the next day b on which an order arrives: we pay P for the delivery, $S(a,b-1)$ for the storage, and then we can behave optimally from day b onward. Thus we have the following recurrence.

$$OPT(a) = P + \min_{b>a:\sum_{j=a}^{b-1} g_j \leq L} S(a,b-1).$$

The values of OPT can be built up in order of decreasing a, in time $O(n-a)$ for iteration a, leading to a total running time of $O(n^2)$. The value we want is $OPT(1)$, and the best ordering strategy can be found by tracing back through the array of OPT values.

[1] ex191.358.563

6.28

(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 Sum Problem. 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 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.

(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 way to build up values for the subproblems.

```
Select-Jobs(n,D)
  Array M[0...n, 0...D]
  Array S[0...n, 0...D]
  For d = 0, ..., D
     M[0, d] = 0
     S[0, d] = φ
   Endfor
   For m = 1, ..., n
     For d = 0, ..., D
       If M[m-1, d] > M[m-1, d-t_m] + 1 then
```

[1] ex601.300.669

```
        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] ∪ {m}
      Endif
    Endfor
  Endfor
  Return M[n, D] and S[n, D]
```

The correctness follows immediately from the statement (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 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.

6.29

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.

[1] ex420.690.864

8.1

(1a) Yes. One solution would be: *Interval Scheduling* can be solved in polynomial time, and so it can also be solved in polynomial time with access to a black box for *Vertex Cover*. (It need never call the black box.) Another solution would be: *Interval Scheduling* is in NP, and anything in NP can be reduced to *Vertex Cover*. A third solution would be: we've seen in the book the reductions *Interval Scheduling* $\leq_P$ *Independent Set* and *Independent Set* $\leq_P$ *Vertex Cover*, so the result follows by transitivity.

(1b) This is equivalent to whether P = NP. If P = NP, then *Independent Set* can be solved in polynomial time, and so *Independent Set* $\leq_P$ *Interval Scheduling*. Conversely, if *Independent Set* $\leq_P$ *Interval Scheduling*, then since *Interval Scheduling* can be solved in polynomial time, so could *Independent Set*. But *Independent Set* is NP-complete, so solving it in polynomial time would imply P = NP.

[1] ex370.181.361

8.2

The problem is in $\mathcal{NP}$ because we can exhibit a set of k customers, and in polynomial time is can be checked that no two bought any product in common.

We now show that *Independent Set* $\leq_P$ *Diverse Subset*. Given a graph G and a number k, we construct a customer for each node of G, and a product for each edge of G. We then build an array that says customer v bought product e if edge e is incident to node v. Finally, we ask whether this array has a diverse subset of size k.

We claim that this holds if and only if G has an independent set of size k. If there is a diverse subset of size k, then the corresponding set of nodes has the property that no two are incident to the same edge — so it is an independent set of size k. Conversely, if there is an independent set of size k, then the corresponding set of customers has the property that no two bought the same product, so it is diverse.

[1] ex640.690.659

8.3

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.

[1] ex195.705.667

8.4

(a) The general *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.

(b) The case $k = 2$ can be solved by brute force: we just try all $O(n^2)$ pairs of processes, and we see whether any pair has disjoint resource requirements. This is a polynomial-time solution.

(c) This is just the Bipartite Matching Problem. We define a node for each person and each piece of equipment, and each process is an edge joining the person and the piece of equipment it needs. A set of processes with disjoint resource requirements is then a set of edges with disjoint ends — in other words, it is a matching in this bipartite graph. Hence we simply check whether there is a matching of size at least k.

(d) We observe that our reduction in (a) actually created an instance of *Resource Reservation* that had this special form. (Each edge/resource in the reduction was requested only by the two nodes/processes that formed its ends.) Thus, even this special case is NP-complete.

[1] ex588.290.312

8.5

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 II: II 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.

[1] ex635.897.959

8.6

On the surface, *Monotone Satisfiability with Few True Variables* (which we'll abbreviate *Monotone Satisfiability with Few True Variables*) 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* $\leq_P$ *Monotone Satisfiability with Few True Variables*. 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 *Monotone Satisfiability with Few True Variables* 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 *Monotone Satisfiability with Few True Variables* 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.

[1] ex799.396.989

8.7

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*.

[1] ex420.972.30

8.8

Magnets is in NP. A witness can be a multiple set of words. We can count the number of each kind of magnets used in these words, and verify whether that is equal to the given number of that kind of magnets.

Now we will show that *Magnets* is NPC by reducing from *3D Matching*. We have an instance of *3D Matching*, which consists of three sets X, Y, C, s.t. $|X| = |Y| = |Z| = n$, and a set of tuples M. We want to find n tuples from M, s.t. each element is covered by exactly one tuple. Create an instance of *Magnets* as follows: each element in X, Y, or Z becomes a magnet with a unique letter, (so our alphabet will have $3n$ letters), and every tuple (x_i, y_j, z_k) becomes a word that Madison knows. Solving this instance of *Magnets* will solve the instance of *3D Matching*.

We must now show that there is a perfect matching in *3D Matching* problem iff all the magnets can be used up in *Magnets* problem. If all the magnets are used up, we must have got exactly n words, since each word has 3 letters, and there are totally $3n$ letters, and therefore we have the desired n tuples. If there is a perfect matching in *3D Matching*, then those words that are corresponding to the tuples in the matching will exactly use up all the magnets without overlapping.

[1] ex536.995.953

8.9

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.)

[1] ex529.979.546

8.10

(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.

[1] ex685.1.698

8.11

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.

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

$$H = \{a_j : v_{ij} \subset P \text{ for some } i\}.$$

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 corresponding node v_{q,j_q} meets the set T_ℓ. Thus P is a valid solution to the *Plot Fulfillment* instance.

[1] ex425.710.356

8.12

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.

[1] ex636.680.130

8.13

The problem is in $\mathcal{NP}$ because we can exhibit a set of bidders, and in polynomial time is can be checked that no two bought bid on the same item, and that the total value of their bids is at least B.

We now show that *Independent Set* $\leq_P$ *Diverse Subset*. Given a graph G and a number k, we construct a bidder for each node of G, and an item for each edge of G. Each bidder v places a bid on each item e for which e is incident to v in G. We set the value of each bid to 1. Finally, we ask whether the auctioneer can accept a set of bids of total value $B = k$.

We claim that this holds if and only if G has an independent set of size k. If there is a set of acceptable bids of total value k, then the corresponding set of nodes has the property that no two are incident to the same edge — so it is an independent set of size k. Conversely, if there is an independent set of size k, then the corresponding set of bidders has the property that their bids are disjoint, and their total value is k.

[1] ex617.432.555

8.14

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.

[1] ex346.976.515

8.15

The problem is in $\mathcal{NP}$ since we can exhibit a set of k locations, and it can be checked in polynomial time, for each frequency, that the frequency is unblocked in at least one of the locations.

Now we show that *Vertex Cover* $\leq_P$ *Nearby Electromagnetic Observation*. Given a graph $G = (V, E)$ and a number k, we define a location ℓ_i corresponding to each node v_i, and a frequency f_s corresponding to each edge e_s.

Now, for each edge $e_s = (v_i, v_j)$, there is an interference source that blocks frequency f_s at all but locations ℓ_i and ℓ_j. Finally, we ask whether there is a sufficient set of size at most k.

If there is such a sufficient set, then the corresponding set of locations has the property that each frequency is unblocked in at least one of them, so the corresponding set S of nodes has the property that each edge is incident to at least one of them. Thus S is a vertex cover in G. Conversely, if there a vertex cover consisting of k nodes in G, then the corresponding set of locations has the property that for every frequency f_s, at least one of the locations has access to f_s. Thus it is a sufficient set.

[1] ex759.113.462

8.16

The problem is in $\mathcal{NP}$ since we can exhibit a set X and check the size of its intersection with every set A_i.

We now show that *3-Dimensional Matching*$\leq_P$ *Intersection Inference*. Suppose we are given an instance of *3-Dimensional Matching*, consisting of sets X, Y, and Z, each of size n, and a set T of m triples from $X \times Y \times Z$. We define the following instance of *Intersection Inference*. We define $U = T$. For each element $j \in X \cup Y \cup Z$, we create a set A_j of these triples that contain j. We then ask whether there is a set $M \subseteq U$ that has an intersection of size 1 with each set A_j.

Such sets are precisely those collections of triples for which each element of $X \cup Y \cup Z$ appears in exactly one: in other words, they are precisely the perfect three-dimensional matchings. Thus, our instance of *Intersection Inference* has a positive answer if and only if the original instance of *3-Dimensional Matching* does.

[1] ex803.795.220

8.17

Zero-weight-cycle is in $\mathcal{NP}$ because we can exhibit a cycle in G, and it can be checked that the sum of of the edge weights on this cycle are equal to 0.

We now show that *Subset Sum* $\leq_P$ *Zero-weight-cycle*. We are given numbers $w_1, \ldots, w_n$, and we want to know if there is a subset that adds up to exactly W. We construct an instance of *Zero-weight-cycle* in which the graph has nodes $0, 1, 2, \ldots, n$, and an edge (i, j) for for all pairs $i < j$. The weight of edge (i, j) is equal to w_j. Finally, there is an edge $(n, 0)$ of weight $-W$.

We claim that there is a subset that adds up to exactly W if and only if G has a zero-weight cycle. If there is such a subset S, then we define a cycle that starts at 0, goes through the nodes whose indices are in S, and then returns to 0 on the edge $(n, 0)$. The weight of $-W$ on the edge $(n, 0)$ precisely cancels the sum of the other edge weights. Conversely, all cycles in G must use the edge $(n, 0)$, and so if there is a zero-weight cycle, then the other edges must exactly cancel $-W$ — in other words, their indices must form a set that adds up to exactly W.

[1] ex642.498.819

8.18

The problem *Decisive Subset (DS)* is in NP because we can check in polynomial time that a given subset of committee members is of size at most k, and that its voting outcome is the same as that of the whole committee.

Now we show that *Vertex Cover* $\leq_P$ *DS*. Given a graph $G = (V, E)$ and bound k, we create an issue I_e for each edge e, and a committee member m_v for each node v. If $e = (u, v)$, then we have members u and v vote "yes" on issue I_e, and all other committee members abstain. Note that the voting outcome by the whole committee on all issues is "yes." We now ask whether there is a decisive subset of size at most k.

If there is a decisive subset S of size at most k, then it must lead to an affirmative decision for each issue. In particular, this means that for each edge e, S must include at least one of the members m_u or m_v, and so the corresponding set of nodes will constitute a vertex cover in G of size at most k.

If there is a vertex cover C of size at most k, then for each edge $e = (u, v)$, at least one of u or v will be in C. For the set S of members corresponding to C, the voting outcome will thus be "yes" on each issue I_e, so S is decisive.

[1] ex7.111.521

8.19

(a) This problem can be solved using network flow. We construct a graph with a node v_i for each cannister, a node w_j for each truck, and an edge (v_i, w_j) of capacity 1 whenever cannister i can go in truck j. We then connect a super-source s to each of the cannister nodes by an edge of capacity 1, and we connect each of the truck nodes to a super-sink t by an edge of capacity k.

We claim that there is a feasible way to place all cannisters in trucks if and only if there is an s-t flow of value n. If there is a feasible placement, then we send one unit of flow from s to t along each of the paths s, v_i, w_j, t, where cannister i is placed in truck j. This does not violate the capacity conditions, in particular on the edges (w_j, t), due to the capacity constraints. Conversely, if there is a flow of value n, then there is one with integer values. We place cannister i in truck j if the edge (v_i, w_j) carries one unit of flow, and we observe that the capacity condition ensures that no truck is overloaded.

The running is the time required to solve a max-flow problem on a graph with $O(m+n)$ nodes and $O(mn)$ edges.

(b) When there are conflicts between pairs of cannisters, rather than between cannisters and trucks, the problem becomes NP-complete.

We show how to reduce *3-Coloring* to this problem. Given a graph G on n nodes, we define a cannister i for each node v_i. We have three trucks, each of capacity $k = n$, and we say that two cannisters cannot go in the same truck whenever there is an edge between the corresponding nodes in G.

Now, if there is a 3-coloring of G, then we can place all the cannisters corresponding to nodes assigned the same coloring in a single truck. Conversely, if there is a way to place all the cannisters in the three trucks, then we can use the truck assignments as colors; this gives a 3-coloring of the graph.

[1] ex460.602.46

8.20

W will prove LDC is NPC by reduction from k *Coloring*, that is, given a graph $G = (V, E)$ and an integer k, we want to know whether we can color V with k colors, s.t. no two adjacent nodes share the same color.

Construct an instance of LDC as follows: for each node $v_i \in V$, we have an object p_i, let

$$d(p_i, p_j) = \begin{cases} 0 & i = j \\ 1 & i \neq j \wedge (v_i, v_j) \notin E \\ 2 & i \neq j \wedge (v_i, v_j) \in E \end{cases}$$

and let $B = 1$. The goal is to partition $\{p_1, p_2, \ldots, p_n\}$ into k subsets.

Now we are going to prove that k *Coloring* is achievable if and only we can find a valid partition in LDC. If we have a valid coloring scheme, then we can partition those objects into k subsets, each of which is corresponding to a subset of nodes which have the same color. From the specification of k *Coloring* problem, we know that there is no edge connecting two nodes with the same color, and therefore their corresponding objects have distance no greater than 1, and hence we have a valid partition in LDC. If we have a valid partition in LDC, then each subset is corresponding to a different color, and we can color those nodes that have their counterparts in the same subset with the same color. By our construction of LDC instance, we know that any two objects in the same subset can't have distance of 2, which means that their corresponding nodes in k *Coloring* problem are not connected. So the coloring will be legal.

[1] ex463.411.47

8.21

The problem is in $\mathcal{NP}$ because we can exhibit a choice of one element from each option set, and it can be checked there are no incompatibilities between these.

We now show that *3-SAT*$\leq_P$ *Fully Compatible Configuration*. The reduction will be very similar to the reduction that showed *3-SAT*$\leq_P$ *Independent Set*. Given an instance of *3-SAT*, we create an option set for each clause, and each of these sets has three options, corresponding to the terms in the clause. We now declare an option in A_i to be incompatible with an option in A_j if the terms corresponding to these two options correspond to a variable and its negation.

If there is a satisfying assignment for the *3-SAT* instance, then we can choose a term from each clause in such a way that we never choose both a variable and its negation. Thus, the corresponding selection of options will have no incompatibilities. Conversely, if there is a way to select options without incompatibilities, then there is a corresponding selection of terms from clauses so that we never choose a variable and its negation. Thus, we can set all variables as indicated by the selected terms (setting variables arbitrary when they appear in no selected term) so as to satisfy the *3-SAT* instance.

[1] ex755.846.885

8.22

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.

[1] ex30.643.488

8.23

Suppose $m \leq n$, 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.

[1] ex690.144.299

8.24

The problem is in $\mathcal{NP}$ since we can exhibit a subset E' of the edges, and it can be checked in polynomial time 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'.

We now show that *Set Cover* is reducible to this problem. Given a collection of sets $S_1, \ldots, S_m$ over a ground set U of size n, we define a bipartite graph in which the nodes in X correspond to the sets S_i, and the nodes in Y correspond to the elements in U. We join each set node to the nodes corresponding to elements that it contains. We also set $a = k$ and $b = n$. (In particular, this means that our (a, b)-skeleton must touch every node in Y.)

Now, if G has an (a, b)-skeleton E', then the k nodes in X incident to edges in E' correspond to k sets that collectively contain all elements, so they form a set cover. Conversely, if there is a set cover of size k, then taking E' to be the set of all edges incident to the corresponding set nodes yields an (a, b)-skeleton.

[1] ex748.182.100

8.25

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.

[1] ex561.283.906

8.26

To show that *Number Partitioning* is in NP we use the subset $S \subset \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i = \frac{1}{2} \sum_{i-1}^{n} w_i$ as the certificate. Given a certificate we can add the corresponding numbers in polynomial time and test if the claimed equation holds.

To prove that *Number Partitioning* is NP-complete, we show that *Subset-Sum* $\leq_P$ *Number Partitioning*. Consider an arbitrary instance of *Subset Sum* with numbers $w_1, \ldots, w_n$ and target sum W. We will construct an equivalent instance of *Number Partitioning*. Let $T = \sum_{i=1}^{n} w_i$ be the total sum of all numbers. Add two numbers $w_{n+1} = W + 1$ and $w_{n+2} = T + 1 - W$. Note that the sum of all $n + 2$ numbers is $2T + 2$. We claim that the partition problem with these $n + 2$ numbers is equivalent to the original *Subset Sum* instance. To prove this, assume first that the answer in the *Subset Sum* problem is "yes", there is a subset S such that $\sum_{i \in S} w_i = W$. Now we can create a partition solution by adding w_{n+2} to the subset S, and using all other numbers as the other part. Now assume conversely that the answer in the *Number Partitioning* problem is "yes", there is a partition where the two parts have equal sums, that is, they both sum to $T + 1$. Note that w_{n+1} and w_{n+2} cannot be in the same part as $w_{n+1} + w_{n+2} > T + 1$. Consider the part that contains w_{n+2}. The sum of all numbers in this part is $T + 1$, so the numbers other than w_{n+2} must sum to W.

Note that it was important to add the $+1$ in both w_{n+1} and w_{n+2}. A natural first idea would have been to use $w_{n+1} = W$ and $w_{n+2} = T - W$. However, this instance of *Number Partitioning* is always "yes", independent of the answer in the original *Subset Sum* problem, as now the total sum is $2T$ and $w_{n+1} + w_{n+2} = T$.

[1] ex123.267.365

8.27

The problem is in $\mathcal{NP}$ because we can exhibit a partition of the numbers into sets, and then sum the squares of the totals in each set.

We now show that *Number Partitioning*, which we proved NP-complete in the previous problem, is reducible to this problem. Thus, given a collection of $x_1, \ldots, x_n$, where we want to know if they can be divided into two sets with the same sum, we construct an instance of this sum-of-squares problem in which $k = 2$ and $B = \frac{1}{2}S^2$, where $S = \sum_{i=1}^{n} x_i$.

If there is a partition of the numbers into two sets with the same sum, then the squared sum of each set is $(\frac{S}{2})^2 = \frac{1}{4}S^2$, and adding this together for the two sets gives $\frac{1}{2}S^2 = B$. Conversely, suppose we have two sets whose total sums are S_1 and S_2 respectively. Then we have $S_1 + S_2 = S$, and $S_1^2 + S_2^2 = \frac{1}{2}S^2$. The only solution to this is $S_1 = S_2 = \frac{1}{2}S$, so these two sets form a solution to the instance of *Number Partitioning*.

[1] ex981.457.448

8.28

The problem is in $\mathcal{NP}$ since we can exhibit a set of k nodes and check that the distance between all pairs is at least 3.

We now show *Independent Set* $\leq_P$ *Strongly Independent Set*. Given a graph G and a number k, we construct a new graph G' in which we replace each edge $e = (u, v)$ by a path of length two: we add a new node w_e, and we add edges $(u, w_e), (w_e, v)$. We also include edges between every pair of new nodes.

Now suppose that G has an independent set of size k. Then in this new graph G', all these k nodes are distance at least three from each other, so this is a strongly independent set of size k. Conversely, suppose G' has a strongly independent set of size k. Now, this set can't contain any of the new nodes, since all such nodes are within distance two of every node in the graph. Thus, it consists of nodes present in G. Moreover, no two of these nodes can be neighbors in G, since then they'd be at distance two in G'. Thus this set of nodes forms an independent set of size k in G.

[1] ex900.39.43

8.29

The *Dominating Set* 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 $\leq_P$ *Dominating Set*

It is worth noting that *Dominating Set* 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 *Dominating Set* 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 *Dominating Set* 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 Dominating Set 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 *Dominating Set* 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.

[1] ex771.562.634

8.30

We choose the problem Y to be *3-SAT*. Take an instance of *3-SAT* and define a real variable y_i in the polynomial for each Boolean variable x_i in the *3-SAT* instance. Now, each clause becomes a product three terms in the variables $\{y_i\}$, where we represent x_i by y_i and $\overline{x_i}$ by $(1 - y_i)$. So for example, the clause $x_i \vee \overline{x_j} \vee x_k$ becomes the corresponding product $(1 - y_i)y_j(1 - y_k)$. Now, by the distributive law, each of these products can be written as a sum of at most eight monomials.

We define our polynomial to be the sum of all these monomials, and our bound B to be 0. For thinking about the reduction, it is useful to think about the polynomial in its form before we applied the distributive law, when it had one product for each clause. In order for its value to be ≤ 0, each of these products must be 0; and the only way for this to happen is for one of the terms in the corresponding clause to be set so that it satisfies the clause. The polynomial can achieve a value ≤ 0 if and only if the original *3-SAT* instance was satisfiable.

[1] ex705.869.283

8.31

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^i_{uv} 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^i_{uv} — for it does, then $Y \cup \{u\} - \{x^i_{uv}\}$ 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^1_{uv}, and x^2_{uv}; since we have chosen Y so that it contains no node of the form x^i_{uv}, it follows that Y contains one of u or v. Thus, Y is a vertex cover of G.

[1] ex867.590.603