This is false. Consider, in fact, one of the examples from the text, with two men and two women.

m prefers w to w'.
m' prefers w' to w.
w prefers m' to m.
w' prefers m to m'.

Note that there are no pairs at all where each ranks the other first, so clearly no such pair can show up in a stable matching.

[1] ex359.539.196

This is true. Indeed, consider such a pair (m, w) and consider a perfectmatching containing pairs (m, w') and (m', w), and, hence, not (m, w). Then since m and w each rank the other first, they each prefer the other to their partners in this matching, and so this matching cannot be stable.

[1] ex742.158.311

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

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

[1] ex468.481.560

The algorithm is very similar to the basic Gale-Shapley algorithm from the text. At any point in time, a student is either "committed" to a hospital or "free." A hospital either has available positions, or it is "full." The algorithm is the following:

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

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

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

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

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

[1] ex304.339.892

(a) The answer is Yes. A simple way to think about it is to break the ties in some fashion and then run the stable matching algorithm on the resulting preference lists. We can for example break the ties lexicographically — that is if a man m is indifferent between two women w_i and w_j then w_i appears on m's preference list before w_j if $i < j$ and if $j < i$ w_j appears before w_i. Similarly if w is indifferent between two men m_i and m_j then m_i appears on w's preference list before m_j if $i < j$ and if $j < i$ m_j appears before m_i.

Now that we have concrete preference lists, we run the stable matching algorithm. We claim that the matching produced would have no strong instability. But this latter claim is true because any strong instability would be an instability for the match produced by the algorithm, yet we know that the algorithm produced a stable matching — a matching with no instabilities.

(b) The answer is No. The following is a simple counterexample. Let $n = 2$ and m_1, m_2 be the two men, and w_1, w_2 the two women. Let m_1 be indifferent between w_1 and w_2 and let both of the women prefer m_1 to m_2. The choices of m_2 are insignificant. There is no matching without weak stability in this example, since regardless of who was matched with m_1, the other woman together with m_1 would form a weak instability.

[1] ex734.923.393

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

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

(1) *A stable matching between ships and ports defines an acceptable assignment of stopping ports.*

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

[1] ex873.532.244

This is closely analogous to the previous problem, with input and output wires playing the roles of ships and ports.

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

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

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

(1) *A stable matching between input and output wires defines a valid switching.*

Proof. To prove this, suppose that this switching causes two data streams to cross at a junction box. Suppose that the junction box is at the meeting of Input i and Output j. Then one stream must be the one that originates on Input i; the other stream must have switched from a different input wire, say Input k, onto Output j. But in this case, Output j prefers Input i to Input k (since j meets i downstream from k); and Input i prefers Output j to the output wire, say Output ℓ, onto which it is actually switched — since it meets Output j upstream from Output ℓ. This contradicts the assumption that we chose a stable matching between input wires and output wires. ■

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

[1] ex852.589.348

Assume we have three men m_1 to m_3 and three women w_1 to w_3 with preferences as given in the table below. Column w_3 shows true preferences of woman w_3, while in column w_3' she pretends she prefers man m_3 to m_1.

m_1	m_2	m_3	w_1	w_2	w_3	(w_3')
w_3	w_1	w_3	m_1	m_1	m_2	m_2
w_1	w_3	w_1	m_2	m_2	m_1	m_3
w_2	w_2	w_2	m_3	m_3	m_3	m_1

First let us consider one possible execution of the G-S algorithm with the true preference list of w_3.

m_1	w_3			w_3
m_2		w_1		w_1
m_3			$[w_3][w_1]w_2$	w_2

First m_1 proposes to w_3, then m_2 proposes to w_1. Then m_3 proposes to w_2 and w_1 and gets rejected, finally proposes to w_2 and is accepted. This execution forms pairs (m_1, w_3), (m_2, w_1) and (m_3, w_2), thus pairing w_3 with m_1, who is her second choice.

Now consider execution of the G-S algorithm when w_3 pretends she prefers m_3 to m_1 (see column w_3'). Then the execution might look as follows:

m_1	w_3		$-$	w_1			w_1
m_2		w_1		$-$	w_3		w_3
m_3			w_3		$-$	$[w_1]w_2$	w_2

Man m_1 proposes to w_3, m_2 to w_1, then m_3 to w_3. She accepts the proposal, leaving m_1 alone. Then m_1 proposes to w_1 which causes w_1 to leave her current partner m_2, who consequently proposes to w_3 (and that is exactly what w_3 wants). Finally, the algorithm pairs up m_3 (recently left by w_3) and w_2. As we see, w_3 ends up with the man m_2, who is her true favorite. Thus we conclude that by falsely switching order of her preferences, a woman may be able to get a more desirable partner in the G-S algorithm.

[1] ex562.302.864

(a) When the input size is doubled, the algorithms get slower by

(i) a factor of 4.

(ii) a factor of 8.

(iii) a factor of 4.

(iv) a factor of 2, plus an additive $2n$.

(v) the square of the previous running time.

(b) When the input size is increased by an additive one, the algorithms get slower by

(i) an additive $2n + 1$.

(ii) an additive $3n^2 + 3n + 1$.

(iii) an additive $200n + 100$.

(iv) an additive $\log(n + 1) + n[\log(n + 1) - \log n]$.

(v) a factor of 2.

[1] ex561.359.766

(i) $6,000,000$.

(ii) 33015.

(iii) $600,000$.

(iv) About 9×10^{11}.

(v) 45.

(vi) 5.

[1] ex695.516.801

We know from the text that polynomials (i.e. a sum of terms where n is raised to fixed powers, even if they are not integers) grow slower than exponentials. Thus, we will consider f_1, f_2, f_3, f_6 as a group, and then put f_4 and f_5 after them.

For polynomials f_i and f_j, we know that f_i and f_j can be ordered by comparing the highest exponent on any term in f_i to the highest exponent on any term in f_j. Thus, we can put f_2 before f_3 before f_1. Now, where to insert f_6? It grows faster than n^2, and from the text we know that logarithms grow slower than polynomials, so f_6 grows slower than n^c for any $c > 2$. Thus we can insert f_6 in this order between f_3 and f_1.

Finally come f_4 and f_5. We know that exponentials can be ordered by their bases, so we put f_4 before f_5.

[1] ex831.202.488

We order the functions as follows.

- g_1 comes before g_5. This is like the solved exercise in which we saw $2^{\sqrt{\log n}}$. If we take logarithms, we are comparing $\sqrt{\log n}$ to $\log n + \log(\log n) \geq \log n$; changing variables via $z = \log n$, this is $\sqrt{z} = z^{1/2}$ versus $z + \log z \geq z$.

- g_5 comes before g_3, since $(\log n)^3$ grows faster than $\log n$. (They're both polynomials in $\log n$, but $(\log n)^3$ has the larger degree.)

- g_3 comes before g_4: Dividing both by n, we are comparing $(\log n)^3$ with $n^{1/3}$, or (taking cube roots), $\log n$ with $n^{1/9}$. Now we use the fact that logarithms grow slower than exponentials.

- g_4 comes before g_2, since polynomials grow slower than exponentials.

- g_2 comes before g_7: Taking logarithms, we are comparing n to n^2, and n^2 is the polynomial of larger degree.

- g_7 comes before g_6: Taking logarithms, we are comparing n^2 to 2^n, and polynomials grow slower than exponentials.

[1] ex413.866.86

(i) This is false in general, since it could be that $g(n) = 1$ for all n, $f(n) = 2$ for all n, and then $\log_2 g(n) = 0$, whence we cannot write $\log_2 f(n) \leq c \log_2 g(n)$.

On th other hand, if we simply require $g(n) \geq 2$ for all n beyond some n_1, then the statement holds. Since $f(n) \leq cg(n)$ for all $n \geq n_0$, we have $\log_2 f(n) \leq \log_2 g(n) + \log_2 c \leq (\log_2 c)(\log_2 g(n))$ once $n \geq \max(n_0, n_1)$.

(ii) This is false: take $f(n) = 2n$ and $g(n) = n$. Then $2^{f(n)} = 4^n$, while $2^{g(n)} = 2^n$.

(iii) This is true. Since $f(n) \leq cg(n)$ for all $n \geq n_0$, we have $(f(n))^2 \leq c^2(g(n))^2$ for all $n \geq n_0$.

[1] ex66.350.972

(a) We prove this for $f(n) = n^3$. The outer loop of the given algorithm runs for exactly n iterations, and the inner loop of the algorithm runs for at most n iterations every time it is executed. Therefore, the line of code that adds up array entries $A[i]$ through $A[j]$ (for various i's and j's) is executed at most n^2 times. Adding up array entries $A[i]$ through $A[j]$ takes $O(j - i + 1)$ operations, which is always at most $O(n)$. Storing the result in $B[i, j]$ requires only constant time. Therefore, the running time of the entire algorithm is at most $n^2 \cdot O(n)$, and so the algorithm runs in time $O(n^3)$.

(b) Consider the times during the execution of the algorithm when $i \leq n/4$ and $j \geq 3n/4$. In these cases, $j - i + 1 \geq 3n/4 - n/4 + 1 > n/2$. Therefore, adding up the array entries $A[i]$ through $A[j]$ would require at least $n/2$ operations, since there are more than $n/2$ terms to add up. How many times during the execution of the given algorithm do we encounter such cases? There are $(n/4)^2$ pairs (i, j) with $i \leq n/4$ and $j \geq 3n/4$. The given algorithm enumerates over all of them, and as shown above, it must perform at least $n/2$ operations for each such pair. Therefore, the algorithm must perform at least $n/2 \cdot (n/4)^2 = n^3/32$ operations. This is $\Omega(n^3)$, as desired.

(c) Consider the following algorithm.

```
For i = 1, 2, ... n
  Set B[i, i + 1] to A[i] + A[i + 1]
For k = 2, 3, ..., n − 1
  For i − 1, 2, ..., n − k
    Set j = i + k
    Set B[i, j] to be B[i, j − 1] + A[j]
```

This algorithm works since the values $B[i, j - 1]$ were already computed in the previous iteration of the outer for loop, when k was $j - 1 - i$, since $j - 1 - i < j - i$. It first computes $B[i, i + 1]$ for all i by summing $A[i]$ with $A[i + 1]$. This requires $O(n)$ operations. For each k, it then computes all $B[i, j]$ for $j - i = k$ by setting $B[i, j] = B[i, j - 1] + A[j]$. For each k, this algorithm performs $O(n)$ operations since there are at most n $B[i, j]$'s such that $j - i = k$. There are less than n values of k to iterate over, so this algorithm has running time $O(n^2)$.

[1] ex474.221.961

Suppose that to obtain n words, we need L lines (most of which will get repeated many times, as described above). We write the script as follows

```
line 1 = <text of line 1 here>
line 2 = <text of line 2 here>
...
line L = <text of line L here>
For i = 1, 2, ..., L
  For j = 1, 2, ..., i
    Sing lines j through 1
  Endfor
Endfor
```

Now, the nested `For` loops have length bounded by a constant c_1, so the real space in the script is consumed by the text of the lines. Each of these lines in the script has length at most c_2 (where c_2 is the maximum line length c plus the space to write the variable assignment). So in total, the space required by the script is $S = c_1 + c_2 L$.

Recall that n denotes the number of words this produces when sung. n is at least $1 + 2 + \cdots + L = \frac{1}{2}L(L-1)$; hence, $\frac{1}{2}(L-1)^2 \leq n$, and so $L \leq 1 + \sqrt{2n}$. Plugging this into our bound on the length of the script, we have $f(n) = S \leq c_1 + c_2\sqrt{2n} = O(\sqrt{n})$.

[1] ex434.486.949

(a) Suppose for simplicity that n is a perfect square. We drop the first jar from heights that are multiples of $\sqrt{n}$ (i.e. from $\sqrt{n}, 2\sqrt{n}, 3\sqrt{n}, \ldots$) until it breaks.

If we drop it from the top rung and it survives, then we're also done. Otherwise, suppose it breaks from height $j\sqrt{n}$. Then we know the highest safe rung is between $(j-1)\sqrt{n}$ and $j\sqrt{n}$, so we drop the second jar from rung $1 + (j-1)\sqrt{n}$ on upward, going up by one each time.

In this way, we drop each of the two jars at most $\sqrt{n}$ times, for a total of at most $2\sqrt{n}$. If n is not a perfect square, then we drop the first jar from heights that are multiples of $\lfloor\sqrt{n}\rfloor$, and then apply the above rule for the second jar. In this way, we drop the first jar at most $2\sqrt{n}$ times (quite an overestimate if n is reasonably large) and the second jar at most $\sqrt{n}$ times, still obtaining a bound of $O(\sqrt{n})$.

(b) We claim by induction that $f_k(n) \leq 2kn^{1/k}$. We begin by dropping the first jar from heights that are multiples of $\lfloor n^{(k-1)/k}\rfloor$. In this way, we drop the first jar at most $2n/n^{(k-1)/k} = 2n^{1/k}$ times, and thus narrow the set of possible rungs down to an interval of length at most $n^{(k-1)/k}$.

We then apply the strategy for $k-1$ jars recursively. By induction it uses at most $2(k-1)(n^{(k-1)/k})^{1/(k-1)} = 2(k-1)n^{1/k}$ drops. Adding in the $\leq 2n^{1/k}$ drops made using the first jar, we get a bound of $2kn^{1/k}$, completing the induction step.

[1] ex291.532.145

This is similar to the solved exercise, although the graph is a bit larger. Node a must come first, and node f must come last. Among the remaining four nodes, b must precede c, and d must precede e, but otherwise we can place them however we want.

To figure out how many orderings of these four middle nodes are possible, we note that either b or d must come first among them.

- If b comes first, then either c or d comes next.
  - If c comes next, the ordering must be b, c, d, e.

    If d comes next, then the ordering can be either b, d, c, e or b, d, e, c.
- If d comes first, we have three orderings obtained simply by reversing the roles of b, c and d, e in the above reasoning.

Thus the total number of orders for these four nodes is 6, and this is the total number of topological orderings.

[1] ex580.660.846

We assume the graph G is connected; otherwise we work with the connected components separately (after computing them in $O(m+n)$ time).

We run BFS starting from an arbitrary node s, obtaining a BFS tree T. Now, if every edge of G appears in the BFS tree, then $G = T$, so G is a tree and contains no cycles. Otherwise, there is some edge $e = (v, w)$ that belongs to G but not to T. Consider the least common ancestor u of v and w in T; we obtain a cycle from the edge e, together with the u-v and u-w paths in T.

[1] ex858.281.666

We run the same algorithm as in the text, with the following small modification. If in every iteration we find a node with no incoming edges, then we will succeed in producing a topological ordering. If in some iteration, it transpires that every node has at least one incoming edge, then we saw in the text a proof that G must contain a cycle. We can turn this proof into an algorithm with running time $O(n)$ to actually find a cycle: we repeatedly follow an edge into the node we're currently at (choosing the first one on the adjacency list of incoming edges, so that this can run in constant time per node). Since every node has an incoming edge, we can do this repeatedly until we re-visit a node v for the first time. The set of nodes encountered between these two successive visits is a cycle C (traversed in the reverse direction).

[1] ex947.5.72

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

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

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

[1] cx926.803.309

```
        If u and v have the same labels
           there is an inconsistency
        Endif
     Endif
  Endfor
```

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

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

We prove this by induction on the number of nodes in T. Let $n_0(T)$ denote the number of leaves of a binary tree T, and let $n_2(T)$ denote the number of ndoes with two children.

The basis of the induction is a tree with a single node. This node is the only leaf, and there are no nodes with two children.

Now, let T be an arbitrary binary tree on more than one node, and let v be a leaf. Since T has more than one node, v is not the root, so it has a parent u. Let T' be the tree obtained by deleting v.

If u had no other child in T, then it becomes a leaf in T', so we have $n_0(T') = n_0(T)$ and $n_2(T') = n_2(T')$. Applying the induction hypothesis to T' completes the induction step in this case. On the other hand, if u had another child in T, then it does not become a leaf after the deletion; but it used to have two children and now it doesn't. Thus we have $n_0(T') = n_0(T) - 1$ and $n_2(T') = n_2(T') - 1$. Again, applying the induction hypothesis to T' completes the induction step in this case.

[1] ex113.833.893

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

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

[1] ex374.652.223

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

[1] ex722.926.396

The claim is false; we show that for every natural number c, there exists a graph G so that $diam(G)/apd(G) > c$. First, we fix a number k (the relation to c will be determined later), and consider the following graph. We take a path on $k-1$ nodes $u_1, u_2, \ldots, u_{k-1}$ in this order. We then attach $n-k+1$ additional nodes $v_1, v_2, \ldots, v_{n-k+1}$, each by a single edge, to u_1; the number n will also be chosen below.

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

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

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

[1] ex85.422.171

We run BFS starting from node s. Let d be the layer in which node t is encountered; by assumption, we have $d > n/2$. We claim first that one of the layers $L_1, L_2, \ldots, L_{d-1}$ consists of a single node. Indeed, if each of these layers had size at least two, then this would account for at least $2(n/2) = n$ nodes; but G has only n nodes, and neither s nor t appears in these layers.

Thus, there is some layer L_i consisting of just the node v. We claim next that deleting v destroys all s-t paths. To see this, consider the set of nodes $X = \{s\} \cup L_1 \cup L_2 \cup \cdots \cup L_{i-1}$. Node s belongs to X but node t does not; and any edge out of X must lie in layer L_i, by the properties of BFS. Since any path from s to t must leave X at some point, it must contain a node in L_i; but v is the only node in L_i.

[1] ex758.356.752

We will solve the more general problem of computing the number of shortest paths from v to every other node.

We perform BFS from v, obtaining a set of layers $L_0, L_1, L_2, \ldots$, where $L_0 = \{v\}$. By the definition of BFS, a path from v to a node x is a shortest v-x path if and only if the layer numbers of the nodes on the path increase by exactly one in each step.

We use this observation to compute the number of shortest paths from v to each other node x. Let $S(x)$ denote this number for a node x. For each node x in L_1, we have $S(x) = 1$, since the only shortest-path consists of the single edge from v to x. Now consider a node y in layer L_j, for $j > 1$. The shortest v-y paths all have the following form; they are a shortest path to some node x in layer L_{j-1}, and then they take one more step to get to y. Thus, $S(y)$ is the sum of $S(x)$ over all nodes x in layer $L_{j\ 1}$ with an edge to y.

After performing BFS, we can thus compute all these values in order of the layers; the time spent to compute a given $S(y)$ is at most the degree of y (since at most this many terms figure in the sum from the previous paragraph). Since we have seen that the sum of the degrees in a graph is $O(m)$, this gives an overall running time of $O(m + n)$.

[1] ex427.691.966

The key here is that, while this kind of connectivity includes a notion of time, it can be converted into a graph connectivity problem of a more standard sort.

We construct the following directed graph G. We scan through the ordered triples in the trace data, maintaining an array pointing to linked lists associated with each computer C_a. (Each list is initalized to null.)

For each triple (C_i, C_j, t_k) we see in our scan, we create nodes (C_i, t_k) and (C_j, t_k), and we create directed edges joining these two nodes in both directions. Also, we append these nodes to the lists for C_i and C_j respectively. If this is not the first triple involving C_i, then we include a directed edge from (C_i, t) to (C_i, t_k), where t is the timestamp in the preceding element (the previously last one) in the list for C_i. We do the analogous thing for C_j. By explicitly maintaining these lists for each node, we are thus able to construct all these new nodes and edges in constant time per triple.

Now, given a collection of triples, we want to decide whether a virus introduced at computer C_a at time x could have infected computer C_b by time y. We walk through the list for C_a until we get to the last node (C_a, x') for which $x' \leq x$. We now run directed BFS from (C_a, x') to determine all nodes that are reachable from it. If a node of the form (C_b, y') with $y' \leq y$ is reachable, then we declare that C_b could have been infected by time y; otherwise we declare it could not have.

Let's argue first about the correctness of the algorithm, then its running time. First, we claim that if there is a path from (C_a, x') to (C_b, y') as in the previous paragraph, then C_b could have been infected by time y. To see this, we simply have the virus move between computers C_i and C_j at time t_k, whenever an edge from (C_i, t_k) to (C_j, t_k) is traversed by the BFS. This is a feasible sequence of virus transmissions that results in the virus first leaving C_a at time x or later (by the definition of x') and arriving at C_b by time y.

Conversely, suppose there were a sequence of virus transmissions that results in the virus first leaving C_a at time x or later and arriving at C_b by time y. Then we can build a path in our graph as follows. We start at node (C_a, x') and follow edges to (C_a, x''), for the x'' when the virus first leaves C_a. (Note that there are such edges since $x' \leq x \leq x''$; or else $x' = x''$.) In general, for each time that the virus moves from C_i to C_j at time t_k, we add the edge from (C_i, t_k) to (C_j, t_k) to the path; if it next moves out of C_j at time $t \geq t_k$, we add the the sequence of edges from (C_j, t_k) to (C_j, t). When the virus first arrives at node C_b, we will have just added a node (C_b, y'') to the path; since $y'' \leq y$ and y' is the largest y involving C_b in the trace data with this property, there is a sequence of edges from (C_b, y'') to (C_b, y'), completing the path.

Finally, we consider the running time. Each triple in the trace data causes us to add a constant number of nodes and edges to the graph, so the graph has $O(m)$ nodes and edges, and since we build it in constant time per node and edge, this takes time $O(m)$. Running BFS takes time linear in the size of the graph, so this too takes time $O(m)$.

[1] ex207.316.912

We construct a directed graph G; the test for consistency will turn out to be equivalent to testing whether G is acyclic.

For each person P_i, we define nodes b_i and d_i, representing their (unknown) birth and death dates respectively. Edges will correspond to one event preceding another. So to start, we include edges (b_i, d_i) for each i. When we are told that

- for some i and j, person P_i died before person P_j was born,

we include an edge (d_i, b_j). When we are told that

- for some i and j, the lifespans of P_i and P_j overlapped at least partially,

we include edges (b_i, d_j) and (b_j, d_i). This completes the construction of G.

Now suppose G has a cycle. Then each of the events corresponding to nodes in this cycle must precede the next; but this means that there is no event among these that can be put first in time, consistent with the given information. Hence the information is not internally consistent.

On the other hand, suppose G has no cycle. Then it has a topological ordering. If we use this as the order of the birth and death dates of all people, then we have an ordering that is consistent with all the given pieces of information.

[1] ex92.401.128

This is true. One argument is as follows: e^* is the first edge that would be considered by Kruskal's algorithm, and so it will be included in the minimum spanning tree.

[1] ex863.322.778

(a) True. If we feed the costs c_e^2 into Kruskal's algorithm, it will sort them in the same order, and hence put the same subset of edges in the MST.

Note: It is not enough just to say, "True, because the edge costs have the same order after they are sorted." The same sentence could be written about (b), which is false; it's crucial here to mention that there are minimum spanning tree algorithms that only care about the relative order of the costs, not their actual values.

(b) False. Let G have edges (s, v), (v, t), and (s, t), where the first two of these edges have cost 3 and the third has cost 5. Then the shortest path is the single edge (s, t), but after squaring the costs the shortest path would go through v.

[1] ex941.760.334

Say n boxes arrive in the order $b_1, \ldots, b_n$. Say each box b_i has a positive weight w_i, and the maximum weight each truck can carry is W. To pack the boxes into N trucks *preserving the order* is to assign each box to one of the trucks $1, \ldots, N$ so that:

- No truck is overloaded: the total weight of all boxes in each truck is less or equal to W.
- The order of arrivals is preserved: if the box b_i is sent before the box b_j (i.e. box b_i is assigned to truck x, box b_j is assigned to truck y, and $x < y$) then it must be the case that b_i has arrived to the company earlier than b_j (i.e. $i < j$).

We prove that the greedy algorithm uses the fewest possible trucks by showing that it "stays ahead" of any other solution. Specifically, we consider any other solution and show the following. If the greedy algorithm fits boxes $b_1, b_2, \ldots, b_j$ into the first k trucks, and the other solution fits $b_1, \ldots, b_i$ into the first k trucks, then $i \leq j$. Note that this implies the optimality of the greedy algorithm, by setting k to be the number of trucks used by the greedy algorithm.

We will prove this claim by induction on k. The case $k = 1$ is clear; the greedy algorithm fits as many boxes as possible into the first truck. Now, assuming it holds for $k - 1$: the greedy algorithm fits j' boxes into the first $k - 1$, and the other solution fits $i' \leq j'$. Now, for the k^{th} truck, the alternate solution packs in $b_{i'+1}, \ldots, b_i$. Thus, since $j' \geq i'$, the greedy algorithm is able at least to fit all the boxes $b_{j'+1}, \ldots, b_i$ into the k^{th} truck, and it can potentially fit more. This completes the induction step, the proof of the claim, and hence the proof of optimality of the greedy algorithm.

[1] ex73.193.591

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

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

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

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

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

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

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

[1] ex876.936.4

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

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

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

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

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

[1] ex198.453.676

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

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

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

[1] ex983.429.914

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

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

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

Now we show that G is actually the optimal schedule, using an exchange argument. We will show that for any given schedule $S \neq G$, we can repeatedly swap adjacent jobs so as to convert S into G without increasing the completion time.

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

If we define an inversion, as in the text, to be a pair of jobs whose order in the schedule does not agree with the order of their finishing times, then such a swap decreases the number of inversions in S while not increasing the completion time. Using a sequence of such swaps, we can therefore convert S to G without increasing the completion time. Therefore the completion time for G is not greater than the completion time for any arbitrary schedule S. Thus G is optimal.

Notes. As with all exchange arguments, there are some common kinds of mistakes to watch out for. We summarize some of these here; they illustrate principles that apply to others of the problems as well.

- The exchange argument should start with an arbitrary schedule S (which, in particular, could be an optimal one), and use exchanges to show that this schedule S can be turned into the schedule the algorithm produces without making the overall completion time worse. It does not work to start with the algorithm's schedule G and simply argue that G cannot be improved by swapping two jobs. This argument would show only that a schedule obtained from G be a *single swap* is not better; it would not rule out the possibility of other schedules, obtainable by multiple swaps, that are better.

[1] ex172.268.910

- To make the argument work smoothly, one should to swap neighboring jobs. If you swap two jobs J_l and J_k that are not neighboring, then all the jobs between the two also change their finishing times.

- In general, it does not work to phrase the above exchange argument as a proof by contradiction — that is, considering an optimal schedule $\mathcal{O}$, assuming it is not equal to G, and getting a contradiction. The problem is that there could many optimal schedules, so there is no contradiction in the existence of one that is not G. Note that when we swap adjacent, inverted jobs above, it does not necessarily make the schedule better; we only argue that such swaps do not make it worse.

Finally, it's worth noting the following alternate proof of the optimality of the schedule G, not directly using an exchange argument. Let job J_j be the job that finishes last on the PC in the greedy algorithm's schedule G, and let S_j be the time this job finishes on the supercomputer. So the overall finish time is $S_j + f_j$. In any other schedule, one of the first j jobs, in the other specified by G, must finish on the supercomputer at some time $T \geq S_j$ (as the first j jobs give exactly S_j work to the supercomputer). Let that be job J_i. Now job J_i needs PC time at least as much as job j (due to the ordering of G), and so it finishes at time $T + f_i \geq S_j + f_j$. So this other schedule is no better.

Suppose by way of contradiction that T and T' are two distinct minimum spanning trees of G. Since T and T' have the same number of edges, but are not equal, there is some edge e' in T' but not in T. If we add e' to T, we get a cycle C. Let e be the most expensive edge on this cycle. Then by the Cycle Property, e does not belong to any minimum spanning tree, contradicting the fact that it is in at least one of T or T'.

[1] ex150.396.359

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

(b) This is true. Suppose that T is a minimum spanning tree of G, and T' is a spanning tree with a lighter bottleneck edge. Thus, T contains an edge e that is heavier than every edge in T'. So if we add e to T', it forms a cycle C on which it is the heaviest edge (since all other edges in C belong to T'). By the Cut Property, then, e does not belong to any minimum spanning tree, contradicting the fact that it is in T and T is a minimum spanning tree.

[1] ex582.808.674

We first do this under the assumption that all edge costs are distinct. In this case, we can solve (a) as follows. Let $e = (v, w)$ be the new edge being added. We represent T using an adjacency list, and we find the v-w path P in T in time linear in the number of nodes and edges of T, which is $O(|V|)$. If every node on this path in T has cost less than c, then the Cycle Property implies that the new edge $e = (v, w)$ is not in the minimum spanning tree, since it is the most expensive edge on the cycle C formed from P and e, so the minimum spanning tree has not changed. On the other hand, if some edge on this path has cost greater than c, then the Cycle Property implies that the most expensive such edge f cannot be in the minimum spanning tree, and so T is no longer the minimum spanning tree.

For (b), we replace the heaviest edge on the v-w path P in T with the edge $e = (v, w)$, obtaining a new spanning tree T'. We claim that T' is a minimum spanning tree. To prove this, we consider any edge e' not in T', and show that we can apply the Cycle Property to conclude that e' is not in any minimum spanning tree. So let $e' = (v', w')$. Adding e' to T' gives us a cycle C' consisting of the v'-w' path P' in T', plus e'. If we can show e' is the most expensive edge on C', we are done.

To do this, we consider one further cycle: the cycle K formed by adding e' to T. By the Cycle Property, e' is the most expensive edge on K. So now there are three cycles to think about: C, C', and K. Edge f is the most expensive edge on C, and edge e' is the most expensive edge on K. Now, if the new edge e does not belong to C', then $C' = K$, and so e' is the most expensive edge on C'. Otherwise, the cycle K includes f (since C' needed to use e instead), and C' uses a portion of C (including e) and a portion of K. In this case, e' is more expensive than f (since f lies on K), and hence it is more expensive than everything on C (since f is the most expensive edge on C). It is also more expensive than everything else on K, and so it is the most expensive edge on C', as desired.

Now, if the edge costs are not all distinct, we apply the approach in the chapter: we first perturb all edge costs by extremely small amounts so they become distinct. Moreover, we do this so we add a very small quantity ϵ to the new edge e, and we perturb the costs of all other edges f by even much smaller, distinct, quantities δ_f. For a tree T, let $c(T)$ denote its real (original) cost, and let $c'(T)$ denote its perturbed cost.

Now we use the above solution with distinct edge costs. Our perturbation has the following two properties.

(i) First, for trees T_1 and T_2, if $c'(T_2) < c'(T_1)$, then $c(T_2) \leq c(T_1)$.

(ii) Second, if $c(T_1) = c(T_2)$, and T_2 contains e but T_1 doesn't, then $c(T_2) > c(T_1)$.

It follows from these two properties that our conclusion in (a) is correct: since $c'(T') < c'(T)$, and T' contains e but T doesn't, property (i) implies $c(T') \leq c(T)$, and then property (ii) implies $c(T') < c(T)$. Now, in (b), we compute a minimum spanning tree with respect to the perturbed costs which, by property (i), is also one of (possibly several) minimum spanning trees with respect to the real costs.

[1] ex833.93.54

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

[1] ex750.114.241

(a) The statement is false. The inequality $b_i > rt_i$ only means the stream cannot be first. For example, if $t_1 = t_2 = 1$ and $b_1 = r + 1$, and $b_2 = r - 1$, than stream 2 can go first, followed by stream 1.

(b) There are many correct ways to solve this. We will list here the main versions.

Version 1. Sort by "stream rate". Rate of stream i is $r_i = b_i/t_i$. Send the streams in increasing order of rates. Assume for simplicity of notation that the streams are given in this order.

Check if the total rate $\sum_{i=1}^{n} b_i \leq r \sum_{i=1}^{n} t_i$ holds. We claim the following

(1) *If this test fails than no ordering produces a feasible schedule. If the test succeeds, then we claim the above order gives a feasible schedule.*

Proof. If the test fails that no order can produce a feasible schedule, as in a total time of $\sum_{i=1}^{n} t_i$ we need to send $\sum_{i=1}^{n} b_i$ no matter what way we order it.

We claim that if the above sorted order sends too much for any initial time period $[0, t]$ than the above test will also fail. To see this consider a time t, and let i be the stream sent during the last time period. If the rate of stream i is at most r (i.e., $r_i \leq r$) then all streams sent so far have a rate at most r, so the total send is at most rt, contradicting the assumption that we sent too much in t time. So we must have $r_i > r$. However, streams are ordered in increasing rate, so in any time step after t we will also send at least r bits, and hence, the total rate at the end of all streams will also violate the "average rate at most r" rule. ■

The running time is $O(n)$. The problem only asked to decide if an ordering exists, and that is done by testing the if the one inequality $\sum_{i=1}^{n} b_i \leq r \sum_{i=1}^{n} t_i$ holds. If we also want to output the ordering we need to spend $O(n \log n)$ time sorting rates. It is also okay to test if the above ordering is feasible after each job (taking $O(n)$ time).

Version 2. Prove that sorting by rates is an optimal schedule by a "greedy stays ahead" argument. The key here is to state in what way the schedule is optimal, and in what way the greedy algorithm stays ahead.

(2) *In the above greedy schedule, after any time t the amount of data transmitted in the first t time steps is as low as possible.*

Proof. Each of the t_i seconds of the schedule transmitting the stream i will have transmission rate of r_i. For any t, the first t seconds are the t lowest transmission rates, so the first t seconds send the lowest total number of bits. If this schedule violates the bit-rate requirements after t seconds, then all other schedules will also violate the requirement as they send at least as many bits.

[1] ex232.234.783

An alternate way of phrasing this argument is to talk about the slack: allowed rate of rt minus the number of bits sent till time t. Using this notion, *the greedy schedule is ahead as for any t, the first t seconds have the highest total slack possible for any schedule.* This is true for the same reason as used above: the schedule sends the the t lowest transmission rates in the first t seconds. The schedule violates the bit-rate requirements after t seconds, if it has negative slack after t seconds, and then all other schedules will also violate the requirement as they have at most as much slack as this schedule. ■

Version 3. Prove that sorting by rates is an optimal schedule by an "exchange argument".

Assume there is a feasible schedule $\mathcal{O}$, and let the algorithm's schedule be $\mathcal{A}$. We say that two jobs are inverted if they occur in different order in $\mathcal{O}$ and in $\mathcal{A}$. As in earlier exchange arguments in the text, we know that if the two orders are different, then there are two adjacent jobs i and j in $\mathcal{O}$ (say i immediately follows j) that are inverted. We need to argue that if $\mathcal{O}$ is a feasible schedule, and i and j are inverted, than the schedule $\mathcal{O}'$ obtained by swapping i and j is also feasible. Let T be the time j starts in $\mathcal{O}$, and assume the schedule $\mathcal{O}$ sends B bits in the first T seconds. The only times when the total amount sent so far is affected by the swap are the times in the range $[T, T + t_j + t_i]$. Let $T + t$ be such a time. Assume that in the schedule $\mathcal{O}$ we send b_o bits during these t seconds. The schedule $\mathcal{O}$ is feasible, and so $B + b_o \leq r(T + t)$. By the ordering used of our algorithm (and the fact that the jobs i and j were inverted), the number of bits sent by the same t seconds in the swapped schedule $\mathcal{O}'$ is $b' \leq b_o$. Therefore, the new schedule satisfies $B + b' \leq B + b_o \leq r(T + t)$. Swapping a pair of adjacent inverted jobs decreases the number of inversions and keeps the schedule feasible. So we can repeatedly swap adjacent inverted jobs until the schedule $\mathcal{O}$ gets converted to the schedule of the greedy algorithm. This proves that the greedy algorithm produces a feasible schedule.

Version 4 In fact, we do not need to sort at all to produce a feasible schedule. For each stream i compute its slack $s_i = rt_i - b_i$. We claim that if a feasible schedule exists, then any order that starts will all steams that have positive slack is feasible. To see why, observe that an ordering is feasible if the sum of slacks is non-negative for any initial segment of the order. Starting with all streams of positive slack creates the highest possible total slack before we start adding the jobs with negative slacks. This observation allows us to create the feasible ordering in $O(n)$ time without sorting, by simply computing the sign of the slack for each stream.

Note that this argument also shows that sorting steams in decreasing in order of slacks works too, as this also orders streams with positive slack before those with negative slack.

Finally, note that one ordering that does not always work is to order steams in increasing order of bits b_i.

An optimal algorithm is to schedule the jobs in decreasing order of w_i/t_i. We prove the optimality of this algorithm by an exchange argument.

Thus, consider any other schedule. As is standard in exchange arguments, we observe that this schedule must contain an *inversion* — a pair of jobs i, j for which i comes before j in the alternate solution, and j comes before i in the greedy solution. Further, in fact, there must be an adjacent such pair i, j. Note that for this pair, we have $w_j/t_j \geq w_i/t_i$, by the definition of the greedy schedule. If we can show that swapping this pair i, j does not increase the weighted sum of completion times, then we can iteratively do this until there are no more inversions, arriving at the greedy schedule without having increased the function we're trying to minimize. It will then follow that the greedy algorithm is optimal.

So consider the effect of swapping i and j. The completion times of all other jobs remain the same. Suppose the completion time of the job before i and j is C. Then before the swap, the contribution of i and j to the total sum was $w_i(C + t_i) + w_j(C + t_i + t_j)$, while after the sum it is $w_j(C + t_j) + w_i(C + t_i + t_j)$. The difference between the value after the swap, compared to the value before the swap is (canceling terms in common between the two expressions) $w_i t_j - w_j t_i$. Since $w_j/t_j \geq w_i/t_i$, this difference is bounded above by 0, and so the total weighted sum of completion times does not increase due to the swap, as desired.

[1] ex948.540.252

(a) The algorithm goes like this:

- Organize all processes in a sequence S by non-decreasing order of their finish times
- While some process in S is still not covered: Insert a status_check right at the finish time of the first uncovered process in S

First, the algorithm won't stop until all the processes are covered; and it terminates since in every iteration at least one uncovered process gets covered. So it does compute a valid set of status_checks that covers all sensitive processes.

Second, we will show that the set of status_checks computed by the algorithm has the minimum possible size, via a "staying ahead" type of argument. Let us call the set of status_checks computed by the above algorithm as C. Take any other set C' that also covers all processes, and we will show by induction that in the interval up to the k^{th} status_check by C, there will also be at least k status_checks by C'.

- Base case: C' has to put a status_check no later than the first status_check in C, since a sensitive process ends at that time.
- Inductive step: Suppose there are at least k status_checks in C' up to the time of the k^{th} status_check in C. We already know that the process causing the insertion of the $(k+1)$-th status_check in C is not covered by all the k status_checks ahead, according to our algorithm, so C' has to put a status_check between the k^{th} and the $(k+1)^{\text{st}}$ status_checks in C to cover that process. Therefore C' has at least $k+1$ status_checks up to the time of the $(k+1)^{\text{st}}$ status_check in C.

So we know that the algorithm is correct.

The running time is $O(n \log n)$ to sort the start and finish times of all the processes, and then $O(n)$ to insert all the status_checks. To do this in linear time, we keep an array that records which processes have been covered so far, and a queue of all processes whose start times we've seen since the last status_check. When we first see a finish time of some uncovered process, we insert a status_check and mark all processes currently in the queue as covered. In this way, we do constant work per process over the course of this part of the algorithm.

(b) The claim is true.

Let us use C to denote the solution provided by our algorithm. The question can be rephrased as asking whether $|C| = k^*$. The question already argues that $k^* \leq |C|$, so what we need to do is to prove $|C| \leq k^*$. As k^* is the *largest* size of a set of sensitive processes with no two ever running at the same time, it is enough to find $|C|$ such "disjoint" processes to show $|C| \leq k^*$. Our algorithm from part (a) actually provides such a set of disjoint processes: the processes whose finish times cause the insertions of the status_checks are disjoint, since each is not covered by all the previous status_checks.

[1] ex559.176.225

At all times, some intervals will be marked (they're already intersected) and some won't. Iteratively, we look at the unmarked interval that ends earliest, and among the intervals that intersect it, we choose the interval I that ends the latest. We add I to our set and mark all intervals intersected by I.

Suppose we select $i_1, i_2, ..., i_k$, and an optimal solution selects $j_1, j_2, ..., j_m$. First note that no interval in either solution is "nested" inside another, so we can assume our two lists of indices are sorted both by start as well as finish time. Let x_t be the earliest-finishing unmarked interval in iteration t: this is the one that caused us to select i_t.

We claim that intervals $j_1, ..., j_{t-1}$ do not intersect x_t. It will then follow that we cannot have $m \leq k - 1$, for then x_k wouldn't be intersected by the optimal solution. The base case is trivial; in general, suppose we know the claim to be true up to x_t. Then the earliest optimal interval j_u that does intersect x_t has $u \geq t$. But i_t does not intersect x_{t+1}, and it is the latest-ending interval that intersects x_t; hence j_u does not intersect x_{t+1} either. So none of $j_1, ..., j_u$ intersect x_{t+1}, and $u \geq t$, so this completes the induction step.

[1] ex624.87.982

In this problem, you basically have a set of n points (the account events) and a set of intervals (the "error bars" around the suspicious transactions, i.e. $[t_i - e_i, t_i + e_i]$), and you want to know if there is a perfect matching between points and intervals so that each point lies in its corresponding interval). Without loss of generality, let us assume $x_1 \leq x_2 \leq \ldots \leq x_n$.

A greedy style algorithm goes like this:

```
for i = 1, 2, ..., n
  if there are unmatched intervals containing x_i
    Match x_i with the one that ends earliest
  else
    Declare that there is no perfect matching
```

It is obvious that if the algorithm succeeds, it really finds a perfect matching. We want to prove that if there is a perfect matching, the algorithm will find it. We prove this by an exchange argument, which we will express in the form of a proof by contradiction.

Suppose by way of contradiction that there is a perfect matching, but that the above greedy algorithm does not construct one. Choose a perfect matching M, in which the first i points $x_1, x_2, \ldots, x_i$ match to intervals in the same way described in the algorithm, and i is the largest number with this property. Now suppose x_{i+1} matches to an interval centered at t_l in M, but the algorithm matches x_{i+1} to another interval centered at t_j. According to the algorithm, we know that $t_j + e_j \leq t_l + e_l$. Suppose t_j is matched to x_k ($x_k \geq x_{i+1}$) in M. Then we have

$$t_l - e_l \leq x_{i+1} \leq x_k \leq t_j + e_j \leq t_l + e_l,$$

so in M we can instead match x_k to t_l and match x_{i+1} to t_j to have a new perfect matching M', which agrees with the algorithm. M' agrees with the output of the greedy algorithm on the first $i + 1$ points, contradicting our choice of i.

To bound the running time, note that if we simply enumerate all unmatched intervals in each iteration of the `for` loop, it will take $O(n)$ time to find the unmatched one that ends earliest. There are n iterations, so the algorithm takes $O(n^2)$ time.

[1] ex18.628.375

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

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

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

[1] ex434.357.684

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

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

The following modification of Dijkstra's algorithm solves this problem. Note that we need to find the fastest way to the destination, and not just the traveling time. To do this we will store for each explored node u the last site before u on the fastest way to u. Then we can restore the fastest path by traversing backward from the destination.

```
Let S be the set of explored nodes.
  For each u ∈ S, we store the earliest time d(u) when we can arrive
at u
  and the last site r(u) before u on the fastest path to u
Initially S = {s} and d(s) = 0.
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
       d'(v) = min_{e=(u,v):u∈S} f_e(d(u)) is as small as possible.
    Add v to S and define d(v) – d'(v) and r(v) – u.
EndWhile
```

To establish the correctness of the algorithm, we will prove by induction on the size of S that for all $v \in S$, the value $d(v)$ is the earliest time that water will reach v.

The case $|S| = 1$ is clear, because we know that water start spreads from the node s at time 0. Suppose this claim holds for $|S| = k \geq 1$; we now grow S to size $k + 1$ by adding the node v. For each $x \in S$, let P_x be a path taken by the water to reach x at time $d(x)$. Let (u, v) be the final edge on the path P_v from s to v.

Now, consider any other s-v path P; we wish to show that the time at which v is reached using P is at least $d(v)$. In order to reach v, this path P must leave the set S *somewhere*; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y.

Let P' be the sub-path of P up to node y. By the choice of node v in iteration $k + 1$, the travel time to y using P' is at least as large as the travel time to v using P_v. Since the time-varying edges do not allow traveling back in time, this means that the route to v through y is at least $d(v)$, as desired.

The algorithm can be implemented using a priority queue, as in the basic version of Dijkstra's algorithm: when node v is added, we look at all edges (v, w), and change their keys according to the travel time from v. This takes time $O(\log n)$ per edge, for a total of $O(m \log n)$.

[1] ex670.460.2

We claim that a minimum spanning tree, computed with each edge cost equal to the negative of its bandwidth, has this property; and so it is enough to compute a minimum spaning tree.

We first prove this with the assumption that all edge costs are distinct, and then show that it remains true even when this assumption is lifted. (Note that the algorithm remains the same either way; it is just the analysis that has to be extended.) So suppose the claim is not true. Then there is some pair of nodes u, v for which the path P in the minimum spanning tree does not have a bottleneck rate as high as some other u-v path P'. Let $e = (x, y)$ be an edge of minimum bandwidth on the path P; note that $e \notin P'$. Moreover, e has the smallest bandwidth of any edge in $P \cup P'$. Now, using the edges in $P \cup P'$ other than e, it is possible to travel from x to y (for example, by going from x back to u via P, then to v via P', then to y via P). Thus, there is a simple path from x to y using these edges, and so there is a cycle C on which e has the minimum bandwidth.

This means that in our minimum spanning tree instance, e has the maximum cost on the cycle C; but e belongs to the minimum spanning tree, contradicting the cycle property.

Now, if the edge costs are not all distinct, we apply the approach in the chapter: we perturb all edge bandwidths by extremely small amounts so they become distinct, and then find a minimum spanning tree. We therefore refer, for each edge e, to a *real bandwidth* b_e and a *perturbed bandwidth* b'_e. In particular, we choose perturbations small enough so that if $b_e > b_f$ for edges e and f, then also $b'_e > b'_f$. Our tree has the best bottleneck rate for all pairs, under the perturbed bandwidths. But suppose that the u-v path P in this tree did not have the best bottleneck rate if we consider the original, real bandwidths; say there is a better path P'. Then there is an edge e on P for which $b_e > b_f$ for all edges f on P'. But the perturbations were so small that they did not cause any edges with distinct bandwidths to change the relative order of their bandwidth values, so it would follow that $b'_e > b'_f$ for all edges f on P', contradicting our conclusion that P had the best bottleneck rate with respect to the perturbed bandwidths.

[1] ex152.208.224

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

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

[1] ex976.901.589

To do this, we apply the Cycle Property nine times. That is, we perform BFS until we find a cycle in the graph G, and then we delete the heaviest edge on this cycle. We have now reduced the number of edges in G by one, while keeping G connected, and (by the Cycle Property) not changing the identity of the minimum spanning tree. If we do this a total of nine times, we will have a connected graph H with $n-1$ edges and the same minimum spanning tree as G. But H is a tree, and so in fact it is the minimum spanning tree.

The running time of each iteration is $O(m+n)$ for the BFS and subsequent check of the cycle to find the heaviest edge; here $m \leq n+8$, so this is $O(n)$.

[1] ex258.711.547

Consider a graph on four nodes v_1, v_2, v_3, v_4 in which there are edges (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1), of cost 2 each, and an edge (v_1, v_3) of cost 1.

Then every edge belongs to some minimum spanning tree, but a spanning tree consisting of three of the edges of cost 2 would not be minmum.

[1] ex27.96.222

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

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

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

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

[1] ex910.984.431

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

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

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

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

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

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

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

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

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

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

[1] ex179.790.171

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

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

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

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

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

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

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

[1] ex947.542.655

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

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

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

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

[1] ex696.856.903

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

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

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

[1] ex135.857.224

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

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

[1] ex708.930.216

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

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

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

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

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

[1] ex168.851.857

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

[1] ex833.921.945

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

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

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

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

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

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

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

[1] ex616.972.80

a) We need to show that there exists a min-cost arborescence which enters every 0-cost strongly connected component(ZSCC) exactly once. The proof is very similar to the proof done in Section 4.9 for cycles. Let T be a min-cost arborescence and for any ZSCC, S, let $e = (u, v)$ be the edge closest to the root, r, entering S. Now we delete all other edges entering S and edges (v_1, v_2) where $v_1, v_2 \in S$, and add edges by doing a DFS on S starting from v. Clearly the resulting graph is an arborescence since we have exactly one edge entering every vertex and every vertex is reachable from the root(for $w \in S$, w is reachable from v; for $w \notin S$ if the path to vertex w went through S with l being the last vertex on the path in S, we now have the path $r - v, v - l, l - w$). Also the cost of the new arborescence is no greater than the cost of T since we only added 0-cost edges. Therefore while contracting we can contract ZSCCs and while opening out we do a DFS to add edges.

b) We have $c''_e = \max(0, c_e - 2y_v)$ where $e = (u, v) \Rightarrow c_e \leq c''_e + 2y_v$. Therefore $c''_e = 0 \Rightarrow c_e \leq 2y_v$. Also $\sum_{v \neq r} y_v$ is a lower bound on $c(T_{opt})$ where T_{opt} is the min-cost arborescence with costs c_e. Since T has 0 c''-cost, we have, $c(T) = \sum_{e \in T} c_e = \sum_{e=(u,v), v \neq r} c_e \leq 2 \sum_{v \neq r} y_v \leq 2c(T_{opt})$.

c) We will prove this by induction on the no. of recursive calls we make. Let G^i, c^i, T^i, T^i_{opt} denote respectively the graph, cost function, arborescence constructed by the algorithm, and the min-cost arborescence(wrt. costs c^i) at the i^{th} stage(recursive call) of the algorithm. For an edge $e = (u, v) \in E^i$, we have, $y_v \leq c^{i-1}_e - c^i_e \leq 2y_v$. Suppose the algorithm terminates after k recursive calls. We will show by induction(on $k - i$ to be precise) that $c^i(T^i) \leq 2c^i(T^i_{opt})$ $\forall i, 1 \leq i \leq k$. The base case is when $i = k$. So $c^k(T^k) - 0 \leq 2c^k(T^k_{opt})$. For the induction step assuming that $c^i(T^i) \leq 2c^i(T^i_{opt})$, we will show that $c^{i-1}(T^{i-1}) \leq 2c^{i-1}(T^{i-1}_{opt})$. Consider the arborescence T^{i-1}_{opt} with cost function c^i. We may modify T^{i-1}_{opt}(by deleting some edges and adding edges of 0 c^i-cost as in **a)**) so that it induces an arborescence, A of on greater c^i-cost on G^i. So we have, $c^i(T^{i-1}_{opt}) \geq c^i(A) \geq c^i(T^i_{opt})$ since T^i_{opt} is min-cost wrt. costs c^i. Now we have,

$$
\begin{aligned}
c^{i-1}(T^{i-1}) &\leq c^i(T^{i-1}) + 2\sum_{v \neq r} y_v && (c^{i-1}_c \leq c^i_c + 2y_v) \\
&= c^i(T^i) + 2\sum_{v \neq r} y_v && \text{(since the edges added to } T^i \text{ all have 0 } c^i- \text{ cost)} \\
&\leq 2(c^i(T^i_{opt}) + \sum_{v \neq r} y_v) && \text{(by the Induction Hypothesis)} \\
&\leq 2(c^i(T^{i-1}_{opt}) + \sum_{v \neq r} y_v) && \text{(using the above lower bound)} \\
&\leq 2c^{i-1}(T^{i-1}_{opt}) && (c^i_e + y_v \leq c^{i-1}_e)
\end{aligned}
$$

and hence by induction $c(T) = c^0(T^0) \leq 2c^0(T^0_{opt}) = 2c(T_{opt})$ where T is the arborescence returned by the algorithm and T_{opt} is the optimal arborescence.

[1] ex271.554.851

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

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

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

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

[1] ex632.624.238

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.

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

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.

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

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

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

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

(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

(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

(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

(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

The key observation to make in this problem is that if the segmentation $y_1 y_2 \ldots y_n$ is an optimal one for the string y, then the segmentation $y_1 y_2 \ldots y_{n-1}$ 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

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

would give us the correct optimal segmentation (where $Quality(\alpha \ldots \beta)$ 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

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

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

(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

(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.

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.

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

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

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

(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

(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

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

(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

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

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

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

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

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

(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.

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

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

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

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

(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.

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

(a) The flow has value 18. It is not a maximum flow.

(b) Define the set A to be s and the top node. The cut $(A, V - A)$ is a minimum cut and has capacity 21.

[1] ex700.306.334

(a) The value of this flow is 10. It is not a maximum flow.

(b) The minimum cut is $(\{s, a, b, c\}, \{d, t\})$. Its capacity is 11.

[1] ex84.475.557

This is is false. Consider a graph with nodes s, v, w, t, edges $(s, v), (v, w), (w, t)$, capacities of 2 on (s, v) and (w, t), and a capacity of 1 on (v, w). Then the maximum flow has value 1, and does not saturate the edge out of s.

[1] ex948.608.156

This is false. Consider a graph with nodes s, v_1, v_2, v_3, w, t, edges (s, v_i) and (v_i, w) for each i, and an edge (w, t). There is a capacity of 4 on edge (w, t), and a capacity of 1 on all other edges. Then setting $A = \{s\}$ and $B = V - A$ gives a minimum cut, with capacity 3. But if we add one to every edge then this cut has capacity 6, more than the capacity of 5 on the cut with $B = \{t\}$ and $A = V - B$.

[1] ex820.292.535

We build the following bipartite graph $G = (V, E)$. V is partitioned into sets X and Y, with a node $x_i \in X$ representing switch i, and a node $y_j \in Y$ representing fixture j. (x_i, y_j) is an edge in E if and only if the line segment from x_i to y_j does not intersect any of the m walls in the floor plan. Thus, whether $(x_i, y_j) \in E$ can be determined initially by m segment-intersection tests; so G can be built in time $O(n^2 m)$.

Now, we test in $O(n^3)$ time whether G has a perfect matching, and declare the floor plan to be "ergonomic" if and only if G does have a perfect matching. Our answer is always correct, since a perfect matching in G is a pairing of the n switches and the n fixtures in such a way that each switch can see the fixture it is paired with, by the definition of the edge set E; conversely, such a pairing of switches and fixtures defines a perfect matching in G.

[1] ex527.636.149

We build the following flow network. There is a node v_i for each client i, a node w_j for each base station j, and an edge (v_i, w_j) of capacity 1 if client i is within range of base station j. We then connect a super-source s to each of the client nodes by an edge of capacity 1, and we connect each of the base station nodes to a super-sink t by an edge of capacity L.

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

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

[1] ex751.45.676

To solve this problem, construct a graph $G = (V, E)$ as follows: Let the set of vertices consist of a super-source node, four supply nodes (one for each blood type) adjacent to the source, four demand nodes and a super sink node that is adjacent to the demand nodes. For each supply node u and demand node v, construct an edge (u, v) if type v can receive blood from type u and set the capacity to ∞ or the demand for type v. Construct an edge (s, u) between the source s and each supply node u with the capacity set to the available supply of type u. Similarly, for each demand node v and the sink t, construct an edge (v, t) with the capacity set to the demand for type v.

Now compute an (integer-valued) maximum flow on this graph. Since the graph has constant size, the scaling max-flow algorithm takes time $O(\log C)$, where C is the total supply, and the Preflow-Push algorithm takes constant time.

We claim that there is sufficient supply for the projected need. if and only if the edges from the demand nodes to the sink are all saturated in the resulting max-flow. Indeed, if there is sufficient supply, in which s_{ST} of type S are used for type T, then we can send a flow of s_{ST} from the supply node of type S to the demand node of type T, and respect all capacity conditions. Conversely, if there is a flow saturating all edges from demand nodes to the sink, then there is an integer flow with this property; if it sends f_{ST} units of flow from the supply node for type S to the demand node for type T, then we can use f_{ST} nodes of type S for patients of type T.

(b) Consider a cut containing the source, and the supply and demand nodes for B and A. The capacity of this cut is $50 + 36 + 10 + 3 - 99$, and hence all 100 units of demand cannot be satisfied.

An explanation for the clinic administrators: There are 87 people with demand for blood types O and A; these can only be satisfied by donors with blood types O and A; and there are only 86 such donors.

Note. We observe that part (a) can also be solved by a greedy algorithm; basically, it works as follows. The O group can only receive blood from O donors; so if the O group is not satisfied, there is no solution. Otherwise, satisfy the A and B groups using the leftovers from the O group; if this is not possible, there is no solution. Finally, satisfy the AB group using any remaining leftovers. A short explanation of correctness (basically following the above reasoning) is necessary for this algorithm, as it was with the flow algorithm.

[1] ex717.885.42

We build the following flow network. There is a node v_i for each patient i, a node w_j for each hospital j, and an edge (v_i, w_j) of capacity 1 if patient i is within a half hour drive of hospital j. We then connect a super-source s to each of the patient nodes by an edge of capacity 1, and we connect each of the hospital nodes to a super-sink t by an edge of capacity $\lceil n/k \rceil$.

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

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

[1] ex297.369.93

We will assume that the flow f is integer-valued. Let $e^* = (v, w)$. If the edge e^* is not saturated with flow, then reducing its capacity by one unit does not cause a problem. So assume it is saturated.

We first reduce the flow on e^* to satisfy the capacity conditions. We now have to restore the capacity conditions. We construct a path from w to t such that all edges carry flow, and we reduce the flow by one unit on each of these edges. We then do the same thing from v back to s. Note that because the flow f is acyclic, we do not encounter any edge twice in this process, so all edges we traverse have their flow reduced by exactly one unit, and the capacity condition is restored.

Let f' be the current flow. We have to decide whether f' is a maximum flow, or whether the flow value can be increased. Since f was a maximum flow, and the value of f' is only one unit less than f, we attempt to find a single augmenting path from s to t in the residual graph $G_{f'}$. If we fail to find one, then f' is maximum. Else, the flow is augmented to have a value at least that of f; since the current flow network cannot have a larger maximum flow value than the original one, this is a maximum flow.

[1] ex257.178.863

The statement is false and the following is a counterexample to it. Let us be given a number $b > 1$ (we will without loss of generality assume that it is an integer, otherwise we will round it up). We consider the following graph. It has $2(b+1)+2$ vertices: source s sink t, and vertices $u_1, u_2, \ldots, u_{b+1}$ that have an edge coming from the source and vertices $v_1, v_2, \ldots, v_{b+1}$ that have an edge going into the sink. There is also an edge from u_i to v_i and from v_i to u_{i+1}. All the edge capacities are 1.

Now assume that the first augmenting path was the path $s \to u_1 \to v_1 \to u_2 \to v_2 \to \ldots u_{b\,|\,1} \to v_{b\,|\,1} \to t$. Then since all the backward edges are deleted from the residual graph according to the super-fast algorithm, the residual graph would contain no path from s to t, and therefore our final flow would equal 1. But there is a flow of value $b+1$ by using the horizontal edges (that is $u_i \to v_i$). Therefore we failed to reach within b of the optimum.

Notice that for different b's we would be considering different graphs, but we are allowed to do this, since the problem asks whether there exists a *universal* b that is independent of the choice of the flow graph G.

[1] ex70.281.132

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

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

[1] ex225.750.725

(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

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

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

(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

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

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

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

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

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

(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

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

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

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

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

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

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

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

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

(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

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

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

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

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

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

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

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

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

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

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

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

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

To see that the *Perfect Assembly* problem is in NP we use the sequence forming the perfect assembly as the certificate. It is easy to check that a given sequence forms a perfect assembly.

To show that the problem is NP-complete, we show that *Hamiltonian Path* $\leq_P$ *Perfect Assembly*. Given an arbitrary instance of the *Hamiltonian Path* problem $G = (V, E)$, we use $\ell = 1$ and use 2 letters v_{in} and v_{out} for each node v in G. The set S of required sequences will be $S = \{v_{in}v_{out}$ for all $v \in V\}$, so a perfect assembly corresponds to ordering the nodes of the graph G. We set T of possible corroborating strings will correspond to edges of E. Let $T = \{v_{out}w_{in}$ for edges $(v, w) \in E\}$. We claim that this instance of *Perfect Assembly* is equivalent to the original *Hamiltonian Path* problem. To prove this assume first that $G = (V, E)$ has a Hamiltonian Path. Order the pairs in S in the order the path traverses the corresponding nodes. This forms a perfect assembly as the edges (v, w) of the path provide the necessary corroborating sequences in T. To see the other direction, assume there is a perfect assembly of S. This ordering of S corresponds to an ordering of the nodes on G, and the ordering is a Hamiltonian path in G, as whenever a node v is followed by a node w in the ordering, by the definition of perfect assembly the sequence $v_{in}v_{out}w_{in}w_{out}$ must be corroborated by a sequence $v_{out}w_{in}$ in T, and hence (v, w) must be an edge of G.

[1] ex47.725.856

Let us call this problem *Trade*. *Trade* is in NP, since a pair of subsets A_i and A_j can serve as a certificate. We can verify that

- $\sum_{a \in A_j} v_i(a) > \sum_{a \in A_i} v_i(a)$
- $\sum_{a \in A_i} v_j(a) > \sum_{a \in A_j} v_j(a)$

by summing up the valuation of each person, which is clearly polynomial time doable.

We will prove *Trade* is NP-complete by reduction from *Subset Sum* problem. Suppose we have an instance of *Subset Sum*, that is, n integers $w_1, w_2, \ldots, w_n$, and another integer W. The goal is to find a subset $S \subseteq w_1, w_2, \ldots, w_n$, s.t. $\sum_{w_k \in S} w_k = W$.

Construct an instance of *Trade* as follows: there are $n+1$ objects $a_1, a_2, \ldots, a_{n+1}$, p_i possesses objects $a_1, a_2, \ldots, a_n$, and p_j possesses a_{n+1}. The first n objects are corresponding to the n numbers in *Subset Sum* problem, and the valuations for them are $v_i(a_k) = v_j(a_k) = w_k \ \ (k = 1, 2, \ldots, n)$. The valuations of the last object are $v_i(a_{n+1}) = W + 1$, and $v_j(a_{n+1}) = W - 1$.

Since p_j only possesses a single object a_{n+1}, so if there exist A_i and A_j, then A_j must be $\{a_{n+1}\}$. A_i will be a subset of $\{a_1, a_2, \ldots, a_n\}$.

Now we will prove that if there is a subset of numbers that sums up to W, if and only if there is a subset $A_i \subseteq \{a_1, a_2, \ldots, a_n\}$, together with $A_j = \{a_{n+1}\}$, satisfying the two inequalities stated before.

If there is a subset S for *Subset Sum* problem, s.t. $\sum_{k \in S} w_k = W$, then we let

$$A_i = \{a_k : w_k \in S\}$$

According to our construction:

$$\sum_{a \in A_j} v_i(a) = v_i(a_{n+1}) = W + 1 > W = \sum_{a \in A_i} v_i(a)$$

and

$$\sum_{a \in A_i} v_j(a) = W > W - 1 = v_j(a_{n_1}) = \sum_{a \in A_j} v_j(a)$$

so A_i and A_j satisfy the requirement of $Trade$ problem.

If there is a subset $A_i \subseteq \{a_1, a_2, \ldots, a_n\}$, and $A_j = \{a_{n+1}\}$ satisfying the two inequalities, then we will let

$$S = \{w_k : a_k \in A_i\}$$

According to our construction, $\sum_{a \in A_i} v_j(a) > W - 1$, $\sum_{a \in A_i} v_i(a) < W + 1$, and $\sum_{w_k \in S} w_k = \sum_{a \in A_i} v_i(a) = \sum_{a \in A_i} v_j(a)$, so $\sum_{w_k \in S} w_k = W$.

[1] ex508.142.105

First we need to make sure that the problem is in NP. This is as in the previous cases easy to see. Given an initial set of size k, we can just run the influence algorithm mentioned in the assignment and obtain a final influence set, and then we can check whether the size of the influence set is at least b. The algorithm is polynomial, since at each iteration we are either terminating or at least adding another vertex to the already influenced list.

Now we need to prove that this problem is also NP-complete. We reduce from vertex cover. Let $G = (V, E)$ be a graph and k be a parameter for the instance of the vertex cover problem. We will create a node for each of the vertices v in G as well as for each of the edges e in G (we will call them vertex-nodes and edge-nodes). There is an edge (v, e) if v is one of the ends of the edge e. All the thresholds on nodes are $1/2$. We want to claim that a vertex cover of size k corresponds to a initial set of size k with the resulting influence set of size $(k + m)$ where m is the number of edges in G, and vice versa.

One direction is easy - if there is a vertex cover of size k then this vertex cover will infect all the edges, therefore we will have a influenced set of size $(k + m)$.

Now if there is an influenced set of size $(k + m)$ from an original set of k nodes, we claim that there is a vertex cover of size k. Notice that if there is an edge-node in the k size original set, then we can replace it by one of the vertex nodes adjacent to it, and the edge will get infected immediately. So therefore doing this operation repeatedly, we will get a set of size k consisting entirely of vertices that infects $(k + n)$ nodes. But this set would be a vertex cover, since each edge would have to have a neighbor among the original vertex nodes, otherwise it would not be infected.

[1] ex364.229.826

We start off by showing that *HSoDT!* is in *NP*. Let the certificate t consist of a path P and a sequence of transactions to be performed along P. Then the certifier B should check if performing the given transactions along the given path P achieves the target bound.

We shall now show *HSoDT!* is NP-complete by showing *3-SAT* $\leq_P$ *HSoDT!*. Consider a *3-SAT* instance with n variables and k clauses. Construct a layered graph $G = (V, E)$ with $n + k$ layers. The first n layers correspond to the n variables and their negations and the last k layers correspond to the clauses. More specifically, layer i of the first n layers consists of two nodes (not adjacent), one that sells droid types corresponding to variable x_i and the other sells droid types corresponding to variable $\overline{x}_i$. The supply of x_i and $\overline{x}_i$ is the total number of times each of them occurs in the k clauses. Also, let their prices be zero. For layer i of the last k layers, construct three nodes (not adjacent) corresponding to the variables or their negations in clause i. If x is a variable or its negation in clause i, then the corresponding node in layer i of the last k layers has a demand for one unit of droid type x with unit cost. Now for each of the first $n + k - 1$ layers, construct directed edges from each of the nodes in layer i to each of the nodes in layer $i + 1$. Construct a starting node s with edges from s to each node in layer 1 and an ending node t with edges from each node in layer $n + k$ to t. Note that there are $2n$ droid types, $2 + 2n + 3k$ nodes including s and t. Now let the target bound be k. We claim that this bound can be reached on this instance of *HSoDT!* if and only if the given *3-SAT* instance has a solution.

Assume we have an *HSoDT!* solution. Note that for each of the layers, we have to pass through exactly one of the nodes. Layer i of the first n layers has two nodes, x_i and $\overline{x}_i$. If the solution passes through node x_i, then let variable x_i have a true assignment else let it have a false assignment. Since the target bound of k is reached, then one droid is sold at each of the last k layers which implies that each clause evaluates to true. Thus we have a *3-SAT* solution.

Now assume we have a *3-SAT* solution. Then we must have each clause evaluate to true, i.e. for each clause C_i, there must be some x_j or $\overline{x}_j$ in C_i such that the one in C_i evaluates to true. Now construct the path P such that for each of the first n layers we pass through node x_i if variable x_i has a true assignment else we pass through node $\overline{x}_i$. When passing through each node in the first n layers, take the available supply of droids. When passing through layer i of the last k layers, visit a node that causes clause i to evaluate to true and sell a unit of the corresponding droid. Since we sell a droid at each of the k layers, the target bound of k is achieved.

[1] ex182.967.464

The problem is in $\mathcal{NP}$ since we can exhibit an order in which to make the specials, and it can be verified that the total amount of money spent on ingredients, using this order, is at most x.

We now show that *Hamiltonian Path* $\leq_P$ *Daily Special Scheduling*. We are given a directed graph $G = (V, E)$ with n nodes and m edges. We define a special at the restaurant corresponding to each node, and an ingredient corresponding to each edge. Each special requires one unit of each of the ingredients corresponding to edges incidents to it (both incoming and outgoing). All ingredients come in bundles of two units, at a cost of one, and they go bad after two days.

We claim there is a Hamiltonian path in G if and only if there is a way to schedule all specials at a cost of $2m - n + 1$. If there is a Hamiltonian path, then we use this order for the specials; rather than having to buy every ingredient twice (for the two recipes that need it), we can save money on the $n - 1$ ingredients that correspond to edges on the path. This is a total cost of $2m - n + 1$. Conversely, if there is a schedule of cost $2m - n + 1$, then since every ingredient is used twice, there must be $n - 1$ that were only bought once. This means that for every consecutive pair of specials, there is a directed edge joining the corresponding nodes in the right order, and so there is a Hamiltonian path in G.

[1] ex723.151.229

Galactic Shortest Path is in NP: given a path P in a graph, we can add up the lengths and risks of its edges, and compare them to the given bounds L and R.

Galactic Shortest Path involves adding numbers, so we naturally consider reducing from the *Subset Sum* problem. Specifically, we'll prove that *Subset Sum* $\leq_P$ *Galactic Shortest Path*.

Thus, consider an instance of *Subset Sum*, specified by numbers $w_1, \ldots, w_n$ and a bound W; we want to know if there is a subset S of these numbers that add up to exactly W. *Galactic Shortest Path* looks somewhat different on the surface, since we have *two kinds* of numbers (lengths and risks), and we are only given *upper bounds* on their sums. However, we can use the fact that we also have an underlying graph structure. In particular, by defining a simple type of graph, we can encode the idea of choosing a subset of numbers.

We define the following instance of *Galactic Shortest Path*. The graph G has a nodes $v_0, v_1, \ldots, v_n$. There are two edges from v_{i-1} to v_i, for each $1 \leq i \leq n$; we'll name them e_i and e'_i. (If one wants to work with a graph containing no parallel edges, we can add extra nodes that subdivide these edges into two; but the construction turns out the same in any case.)

Now, any path from v_0 to v_n in this graph G goes through edge one from each pair $\{e_i, e'_i\}$. This is very useful, since it corresponds to making n independent binary choices — much like the binary choices one has in *Subset Sum*. In particular, choosing e_i will represent putting w_i into our set S, and e'_i will represent leaving it out.

Here's a final observation. Let $W_0 = \sum_{i=1}^{n} w_i$ — the sum of all the numbers. Then a subset S adds up to W if and only if its complement adds up to $W_0 - W$.

We give e_i a length of w_i and a risk of 0; we give e'_i a length of 0 and a risk of w_i. We set the bound $L - W$, and $R - W_0 - W$. We now claim: there is a solution to the *Subset Sum* instance if and only if there is a valid path in G. For if there is a set S adding up to W, then in G we use the edges e_i for $i \in S$, and e'_j for $j \notin S$. This path has length W and risk $W_0 - W$, so it meets the given bounds. Conversely, if there is a path P meeting the given bounds, then consider the set $S = \{w_i : e_i \in P\}$. S adds up to at most W and its complement adds up to at most $W_0 - W$. But since the two sets together add up to exactly W_0, it must be that S adds up to exactly W and its complement to exactly $W_0 - W$. Thus, S is valid solution to the *Subset Sum* instance.

[1] ex129.970.939

The problem is in $\mathcal{NP}$, since we can exhibit a set of k edges, and it can be checked in polynomial time that all nodes in X can reach one another on paths using these edges.

Now we show *Vertex Cover* $\leq_P$ *Graphical Steiner Tree*. Given a graph G on n nodes and m edges, and a number k, we construct a new graph H as follows. We insert a new node w_e in the middle of each edge $e = (u, v)$ (connected only to u and v); let W denote this set of new nodes. We also include one additional node r connected to all the original nodes V of the graph G. We define $X = \{r\} \cup W$, and we ask whether X can be connected using $k' = k + m$ edges.

If G has a vertex cover S of size k, this can be done: we include edges from r to each node in S, and from each node in W to one of its neighbors in S.

Conversely, suppose there is a Steiner tree for X using a set F of at most k' edges. We may assume that F includes at most one edge incident to each node w_e: if it included both edges, we could modify it so that one of these edges was retained, and the other edge, say (u, w_e), was replaced by the edge (r, u). The resulting set would still form a Steiner tree.

Given this structure for F, we see that m edges are used to connect to nodes in W, and that leaves k edges from r to nodes of V. Let S be the ends in V of these edges from r. We claim that S is a vertex cover. Indeed, every node w_e has a path to r, and by the structure we have imposed on F, this path must consist of two steps: from w_e to one of its ends u, and then directly to r. Thus, every edge e is incident to at least one node in S, so S is a vertex cover.

[1] ex569.422.522

The problem is in $\mathcal{NP}$, since we can exhibit a set of disjoint paths P_i, and it can be checked in polynomial time that they are paths in G, connect the corresponding nodes, and are disjoint.

Now we show *3-SAT* $\leq_P$ *Directed Disjoint Paths.* Consider a *3-SAT* problem given by a set of clauses $C_1, \ldots, C_k$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$. To create the corresponding instance of the Directed Disjoint Paths problem, we will have $2n$ directed paths, each of length k, one paths P_i corresponding to variable x_i and one path P_i' corresponding to $\overline{x_i}$. We add n source-sink pairs corresponding to the n variables, and connect source s_i to the first node on paths P_i and P_i' and connect the last nodes in paths P_i and and P_i' to sink t_i. Note that there are two directed paths connecting s_i to t_i: the path s_i, P_i, t_i, and the path s_i, P_i', t_i. We will think of selecting the first of these paths as setting the variable x_i to *false* (as the variable through the copies of $\overline{x_i}$ are left unused), and selecting the second path will correspond to setting the variable x_i to *true*.

Now will will add k additional source sink pairs, one corresponding to each clause C_j. Let S_j and T_j the source sink pair corresponding to clause C_j. We will claim that there is a path from S_j to T_j disjoint from the path selected to connect the s_i-t_i source-sink pairs if and only of clause C_j is satisfied by the corresponding assignment. Assume clause C_j contains the literal t_{j1}, t_{j2} and t_{j3}. Now we have a path P_i or P_i' corresponding to each of these variables or negated variables. The paths have n nodes each, let v_{j1}, v_{j2} and v_{j3} denote the jth node on the 3 corresponding paths. We add the edges $(S_j, v_{j\ell})$ and $(v_{j\ell}, T_j)$ for each of $\ell = 1, 2, 3$.

Now we claim that the resulting directed graph has node disjoint paths connecting the source-sink pairs s_i-t_i and S_j-T_j for $i = 1, 2, \ldots, n$ and $j = 1, \ldots, k$ if and only if the *3-SAT* instance is satisfiable. One direction is easy to see: if the *3-SAT* instance is satisfiable, then select the paths connecting s_i to t_i corresponding to the satisfying assignment, as suggested above. Then the source-sink pair S_j and T_j can be connected through the path using the true variable in the clause.

Finally, we need to show that if the disjoint paths exists, than the *3-SAT* formula has a satisfying assignment. Note that the paths P_i and P_j are disjoint, and the graph has no edges connecting different paths. The only edges outside these paths in the graph are edges entering one of the sinks, or leaving a source. As a result the only paths in the graph connecting an s_i-t_i pair are the two paths s_i, P_i, t_i and s_i, P_i', t_i, and the only paths in G connecting S_j-T_j pairs are the three possible paths through each of the 3 variable nodes in C_j. Hence, sets of disjoint paths connecting the source-sink pairs, correspond to satisfying assignments.

[1] ex563.824.406

The problem is in NP since, given a schedule for message transmissions, one can verify that it obeys the switching time constraints, and that all nodes in the target set receive the message by the desired time.

We now show how to reduce 3-*Dimensional Matching* to *Broadcast Time*. Consider an instance of 3-*Dimensional Matching*, with triples from $X \times Y \times Z$, where $n = |X| = |Y| = |Z|$. We construct a graph G with a root r, a node v_t for each triple t, and a node w_a for each element $a \in X \cup Y \cup Z$. There are directed edges (r, v_t) for each t, and (v_t, w_a) when a belongs to triple t. The target set T is $\{w_a : a \in X \cup Y \cup Z\}$. The root r has switching time 4, and all other nodes have switching time 1.

We claim that there exists a set of n triples covering $X \cup Y \cup Z$ if and only if there is a broadcast scheme with completion time at most $4n - 1$. Indeed, if there are n such triples, then r sends messages to each corresponding node v_t at times $0, 4, 8, \ldots, 4n-4$; and if node v_t receives a message from r at time $4j$, it sends messages to its three elements at times $4j + 1, 4j + 2, 4j + 3$. Conversely, if there is a broadcast scheme with completion time at most $4n - 1$, then r must have sent at most n messages, since its switching time is 4. Now, if R is the set of nodes that received messages from r, then $M = \{t : v_t \in R\}$ must cover $X \cup Y \cup Z$, since every node w_a has received the message. Thus M forms a solution to the instance of 3-*Dimensional Matching*.

[1] ex597.881.577

(a) Assume graph $G = (V, E)$ has vertex set V and edge set E. Create a bipartite graph with the two sides both corresponding to V, that is, for each node $v \in V$ we add two nodes v_{in} and w_{out} to the bipartite graph. For each undirected edge $e = (v, w) \in G$ we add the edge (v_{out}, w_{in}) to the bipartite graph G'. We claim that cycle covers in G are in one-to-one correspondence with perfect matchings in G'. This is true as a set of edges forms a cycle cover if and only if it contains exactly one edge entering and exactly one edge leaving each vertex v. Hence we can find a cycle cover in $O(mn)$ time by finding a perfect matching in G'.

(b) Clearly the Cycle Cover Problem is in NP, as given the set of edges that form a cycle cover, it is easy to check if they form a cycle cover of G. We will prove that Cycle Cover with at most 3 edges in each cycle is NP-complete by a reduction from *3-Dimensional Matching*. Consider an instance of the *3-Dimensional Matching* Problem, given by disjoint sets X, Y, and Z, each of size n; and a set $T \subseteq X \times Y \times Z$ of ordered triples. We create an instance of the cycle cover problem as follows. Let $S = X \cup Y \cup Z$ denote the set of nodes. First add 3 nodes with a triangle connecting them for each triple $t \in T$. These will be the only triangles used in our construction. Let the 3 nodes in the triangle correspond to the 3 elements of the set t, but node that if a node $s \in S$ is covered by multiple triples, than we add separate nodes corresponding to v for each triple. For a node $s \in S$ let A_s be the set of these nodes corresponding to node v. We will also add a set of B_v additional nodes corresponding to v with $|B_v| = |A_v| - 1$, and add edges between all nodes in the sets A_v and B_v in both direction. This finishes the construction of the graph G.

Now we claim that there is a perfect matching in the *3-Dimensional Matching* Problem if and only if G has a Cycle Cover with at most 3 edges in each cycle. First assume we have perfect 3D matching. The matching corresponds to a cycle cover as follows. For each triple t in the 3D matching, select the corresponding triangle. This covers exactly one node in each set A_s for each $s \in S$. Now cover the remaining nodes by cycles of length 2 going between the sets A_s and B_s.

Finally, we need to show that if G has a Cycle Cover with at most 3 edges in each cycle, than there is a perfect 3D matching in the *3-Dimensional Matching* Problem. So see this note that the elements of the sets B_s can only be covered by 2 cycles, as no 3 cycle passes through these nodes. Any set of 2 cycles covering the node $\cup_s B_s$ leaves one node uncovered from each set A_s. The union of the set $\cup_s A_s$ consists of node-disjoint triangles corresponding to the triples in T, so these remaining nodes can only be covered if they correspond to triples in T.

[1] ex300.637.662

Given a proposed solution to domain decomposition, we can test each domain in turn to see whether every node has a path to every other node. (This can be done very efficiently by testing for strong connectivity.) Thus *Domain Decomposition* is in NP.

We now show that *Three-Dimensional Matching* $\leq_P$ *Domain Decomposition*. To do this, we start with an instance of *Three-Dimensional Matching*, with sets X, Y, and Z of size n each, and a collection C of t ordered triples.

We construct the following instance of *Domain Decomposition*. We construct a graph $G = (V, E)$, where V consists of a node x'_i for each $x_i \in X$, y'_j for each $y_j \in Y$, and z'_k for each $z_k \in Z$. For each triple A_m in C, we will also define three nodes v^x_m, v^y_m, and v^z_m. Let U denote all nodes of the form x'_i, y'_j, or z'_k. We now define the following edges in G. For each triple of nodes v^x_m, v^y_m, and v^z_m, we construct a directed triangle via edges $(v^x_m, v^y_m), (v^y_m, v^z_m), (v^z_m, v^x_m)$. For each node x'_i, and each node v^x_m for which x_i appears in the triple A_m, we define edges (x'_i, v^x_m) and (v^x_m, x'_i). We do the analogous thing for each node y'_j and z'_k.

So the idea is to create a directed triangle for each triple, and a pair of bi-directional edges between each element and each triple that it belongs to. We want to encode the existence of a perfect tripartite matching as follows. For each triple $A_m = (x_i, y_j, z_k)$ in the matching, we will construct three 2-element domains consisting of the nodes x'_i, y'_j, z'_k together with the nodes v^x_m, v^y_m, and v^z_m respectively. For each triple A_m that is *not* in the matching, we will simply construct the 3-element domain on v^x_m, v^y_m, and v^z_m.

Thus, we claim that G has a decomposition into at least $3n+t-n - 2n+t$ domains if and only there is a perfect tripartite matching in C. If there is a perfect tripartite matching, then the construction of the previous paragraph produces a partition of V into $2n + t$ domains. So let us prove the other direction; suppose there is a partition of V into $2n + t$ domains. Let p denote the number of domains containing elements from U. Note that $p \leq 3n$, and $p = 3n$ if and only if each element of U appears in a 2-element domain. Let q denote the number of domains not containing elements from U. Each such domain must consist of a single triangle; since at least n triangles are involved in domains with elements of U, we have $q \leq t - n$, and $q = t - n$ if and only if the domains involving U intersect only n triangles. Now, the total number of domains is $p + q$, and so this number is $2n + t$ if and only the domains consist of $t - n$ triangles, together with $3n$ two-element domains involving elements of U. In this case, the triangles that are *not* used in the domain decomposition correspond to triples in the *Three-Dimensional Matching* instance that are all disjoint.

Thus, by deciding whether G has a decomposition into at least $2n + t$ domains, we can decide whether our original instance of *Three-Dimensional Matching* has a solution.

[1] ex742.89.672

This problem can be decided in polynomial time. It helps to view the $QSAT$ instance as a $CSAT$ instance instead: Player 1 controls the set A of odd-indexed variables while Player 2 controls the set B of even-indexed variables. Our question then becomes: can Player 1 force a win?

We claim that Player 1 can force a win if and only if each clause C_i contains a variable from A. If this is the case, Player 1 can win by setting all variables in A to 1. If this is not the case, then some clause C_i has no variable from A. Player 2 can then win by setting all variables in B to 0: in particular, this will cause the clause C_i to evaluate to 0.

[1] ex63.946.695

We show that *Competitive 3-SAT* $\leq_P$ *Geography on a Graph* by encoding the Boolean formula Φ over which the players are competing as a directed graph G. The construction we use is depicted in the accompanying figure.

For each variable x_i, we create a "diamond" consisting of vertices labeled $a_i, x_i, \overline{x_i}, b_i$ with directed edges as in the figure. The starting node for the game is a_1, and there is an edge (a_{i+1}, b_i) for each $i < n$. Thus, in the initial moves of the game, Player 1 essentially chooses a side of the i^{th} diamond, for each odd value of i, while Player 2 choose a side of the i^{th} diamond for each even value of i. At the end of this process, the current node is b_n, and it is Player 2's turn to move.

The next two moves of the game will now determine which player wins. Below b_n, we create a node c_i, $i = 1, \ldots, k$, representing the k clauses of Φ. Out of each node c_i, we construct edges to nodes associated with terms occurring in clause i, as follows: if x_j occurs in clause i, then we add the edge (c_i, x_j); and if $\overline{x_j}$ occurs in clause i, then we add the edge $(c_i, \overline{x_j})$. These edges model the following construction: Player 2 chooses a clause of Φ, and Player 1 must then choose a term in this clause that has been set to T by the earlier play. If he can do this for any clause, then he must have had a winning strategy for the *Competitive 3-SAT* instance; otherwise, Player 2 must have had a winning strategy. We make this concrete in the following claim.

(1) *Player 1 has a forced win in the* Competitive 3-SAT *instance if and only if Player 1 has a forced win from* a_1 *in the* Geography on a Graph *instance on* G.

Proof. First suppose that Player 1 has a forced win in the *Competitive 3-SAT* instance. Then he applies this strategy in the *Geography on a Graph* instance as follows. Each time Player 2 moves to a node x_j (respectively $\overline{x_j}$), Player 1 interprets this as setting $x_j = F$ (respectively $x_j = T$), i.e., the true value is the one that is left blank for later use. When Player 1 has to decide whether to move to node x_i or $\overline{x_i}$, he consults his *Competitive 3-SAT* strategy. If it calls for him to set $x_i = T$, then he moves to $\overline{x_i}$; if it calls for him to set $x_i = F$, then he moves to x_i.

Now consider the final two moves of the game. Player 2 chooses the node c_i. We know, since Player 1's choices forced a win in the *Competitive 3-SAT* instance, that there is a term t in c_i that evaluates to T. This means that in the earlier moves, someone chose the node labeled with the negation of t, but no one chose the node labeled t. Consequently, Player 1 can take the edge (c_i, t), and then Player 2 will not be able to make a move.

To show the converse direction, suppose that Player 1 has a forced win in the *Geography on a Graph* instance. Then we can convert this to a *Competitive 3-SAT* strategy in a very similar way: moves to x_i (respectively $\overline{x_i}$) in *Geography on a Graph* correspond to setting $x_i = F$ (respectively $x_i = T$) in the *Competitive 3-SAT* instance. Now, because Player 1 can force a win in *Geography on a Graph*, he must have a legal move for every choice of a node c_i by Player 2. This means that in every clause of the *Competitive 3-SAT* instance, there is some term that evaluates to T — so Player 1 has won the *Competitive 3-SAT* instance. ■

[1] ex554.747.411

We label the vertices $v_1, v_2, \ldots, v_n$ according to a topological ordering. We now define $Win(j)$ to be equal to 1 if the player whose turn it is to move can force a win starting at node v_j, and define $Win(j)$ to be equal to 0 if the other (who isn't about to move) can force a win starting at node v_j.

We can initialize $Win(j) = 0$ for every node v_j with no out-going edges. In particular, this means that we will set $Win(n) = 0$. We now use dynamic programming to compute the values of $Win(j)$ in descending order of j. When we get to a particular value of j, we may assume that we have already computed $Win(k)$ for all $k > j$. Now, a player starting from v_j can force a win if and only if there is some node v_k for which (v_j, v_k) is an edge and a player starting from v_k has a forced loss. Thus, $Win(j) = 1$ if and only if $Win(k) = 0$ for some k with (v_j, v_k) an edge; and otherwise $Win(j) = 0$.

We thus compute all these values in $O(n)$ time per entry, for a total of $O(n^2)$. We then simply check the value of $Win(j)$ for the node v_j on which the game is designated to start.

[1] ex701.675.797

Our solution will be similar to the algorithm for VERTEX COVER from the first section of this chapter. Consider the notation defined in the problem. For an element $a \in A$, we *reduce the instance by* a by deleting a from A, and deleting all sets B_i that contain a. Thus, reducing the instance by a producing a new, presumably smaller, instance of HITTING SET.

We observe the following fact. Let $B_i = \{x_1, \ldots, x_c\} \subseteq A$ be any of the given sets in the HITTING SET instance. Then at least one of $x_1, \ldots, x_c$ must belong to any hitting set H. So by analogy with (2.3) from the notes, we have the following fact

- Let $B_i = \{x_1, \ldots, x_c\}$ There is k-element hitting set for the original instance if and only if, for some $i = 1, \ldots, c$, the instance reduced by x_i has a $(k \quad 1)$-element hitting set.

The proof is completely analogous to that of (2.3). If H is a k-element hitting set, then some $x_i \in H$, and so $H - \{x_i\}$ is a $(k-1)$-element hitting set for the instance reduced by x_i. Conversely, if the instance reduced by x_i has a $(k-1)$-element hitting set H', then $H' \cup \{x_i\}$ is a k-element hitting set for the original instance.

Thus, our algorithm is as follows. We pick any set $B_i = \{x_1, \ldots, x_c\}$. For each x_i, we recursively test if the instance reduced by x_i has a $(k-1)$-element hitting set. We return "yes" if and only if the answer to one of these recursive calls is "yes." Our running time satisfies $T(m,k) \leq cT(m,k-1) + O(cm)$, and so it satisfies $T(m,k) = O(c^k \cdot kcm)$. This gives the desired bound, with $f(c,k) = kc^{k+1}$ and $p(m) = m$.

[1] ex579.588.787

(a) We prove this by induction on d. If $d = 0$, then Φ is a satisfying assignment, and $Explore(\Phi, d)$ returns "yes."

Now consider $d > 0$. If $Explore(\Phi, d)$ returns "yes," it is because one of the recursive calls $Explore(\Phi_i, d-1)$ returns "yes"; by induction, this means that Φ_i has distance $d-1$ to a satisfying assignment, and so Φ has distance d to a satisfying assignment.

Conversely, suppose Φ has distance d to a satisfying assignment Φ'. Consider any clause unsatisfied by Φ; since Φ' satisfies it, it must disagree with Φ on the setting of at least one of the variables in this clause. Thus, one of the assignments Φ_i, which changes the assignment to this variable, is at distance $d-1$ to Φ'; by induction the recursive call $Explore(\Phi_i, d-1)$ will return "yes," and so the full call $Explore(\Phi, d)$ will also return "yes."

The running time for $Explore$ satisfies the recurrence $T(n, d) \leq 3T(n, d-1) + p(n)$, for a polynomial p. Unwinding this to get d down to 0, we have a running time of $O(3^d \cdot p(n))$.

(b) We let Φ_0 denote the assignment in which all variables are set to 0, and we let Φ_1 denote the assignment in which all variables are set to 1. If there is any satisfying assignment, it is within distance at most $n/2$ of one of these, so we can call both $Explore(\Phi_0, n/2)$ and $Explore(\Phi_1, n/2)$, and see if either of these returns "yes."

The running time of each of these calls is $O(p(n) \cdot 3^{n/2}) = O(p(n) \cdot (\sqrt{3})^n)$.

[1] ex695.88.327

Consider an ordered triple (S, i, j), $1 \leq i, j \leq n$ and S is a subset of the vertices that includes v_i and v_j. Let $B[S, i, j]$ denote the answer to the question, "Is there a Hamiltonian path on $G[S]$ that starts at v_i and ends at v_j?" Clearly, we are looking for the answer to $B[V, 1, n]$.

We now show how to construct the answers to all $B[S, i, j]$, starting from the smallest sets and working up to larger ones, spending $O(n)$ time on each. Thus the total running time will be $O(2^n \cdot n^3)$.

$B[S, i, j]$ is true if and only if there is some vertex $v_k \in S - \{v_i\}$ so that (v_i, v_k) is an edge, and there is a Hamiltonian path from v_k to v_j in $G[S - \{v_i\}]$. Thus, we set $B[S, i, j]$ to be true if and only if there is some $v_k \in S - \{v_i\}$ for which $(v_i, v_k) \in E$ and $B[S - \{v_i\}, k, j]$ is true. This takes $O(n)$ time to determine.

[1] ex386.623.944

We claim that such a graph G has a tree decomposition $(T, \{V_t\})$ in which each piece V_t corresponds uniquely to an internal triangular face of G. We prove this by induction on the number of nodes in G.

Choose any internal edge $e = (u, v)$ of G; deleting u and v produces two components A and B. Let G_1 be the subgraph induced on $A \cup \{u, v\}$ and G_2 the subgraph induced on $B \cup \{u, v\}$. By induction, there are tree decompositions $(T_1, \{X_t\})$ and $(T_2, \{Y_t\}$ of G_1 and G_2 respectively in which the pieces correspond uniquely to internal faces. Thus there are nodes $t_1 \in T_1$ and $t_2 \in T_2$ that correspond to the faces containing the edge (u, v). If we let T denote the tree obtained by adding an edge (t_1, t_2) to $T_1 \cup T_2$, then $(T, \{X_t\} \cup \{Y_t\})$ is a tree decomposition having the desired properties.

[1] ex203.262.545

(a) Let us root the tree at an arbitrary node r, and define subtrees T_u as we have done in Chapter 10.2 when solving the Weighted Independent Set Problem. There we defined two subproblems corresponding to each subtree depending on whether or not we include the root u in the set. We will use the same subproblems: $OPT_{in}(u)$ denotes the maximum weight of an independent set of T_u that includes u, and $OPT_{out}(u)$ denotes the maximum weight of an independent set of T_u that does not include u. Now the optimum we are looking for is $\min(OPT_{in}(r), OPT_{out}(r))$. It helps us to define a third subproblem for each subtree: $OPT_{un}(u)$ denotes the maximum weight of an independent set of T_u that does not have to dominate u. When u's parent is included in the dominating set, u is already dominated by its parent, hence the set selected in the subtree T_u does not have to dominate u.

Now that we have our sub-problems, it is not hard to see how to compute these values recursively. For a leaf $u \neq r$ we have that $OPT_{out}(u) = \infty$, $OPT_{in}(u) = c(u)$, and $OPT_{un}(u) = 0$. For all other nodes u we get a recurrence that defines $OPT_{out}(u)$, $OPT_{in}(u)$, and $OPT_{un}(v)$ using the values for u's children.

(1) *For a node u, the following recurrence defines the values of the sub-problems:*

- $OPT_{in}(u) = c(u) + \sum_{v \in children(u)} OPT_{un}(v)$
- $OPT_{un}(u) = \sum_{v \in children(u)} \min(OPT_{in}(v), OPT_{out}(v))$
- $OPT_{out}(u) = \min_{v \in children(u)} (OPT_{in}(v) + \sum_{w \in children(u), w \neq v} \min(OPT_{out}(v), OPT_{in}(v)))$.

Using this recurrence, we get a dynamic programming algorithm by building up the optimal solutions over larger and larger sub-trees. We define arrays $Mo[u]$, $Mi[u]$ and $Mu[u]$, which hold the values $OPT_{out}(u)$, $OPT_{in}(u)$ and $OPT_{un}(u)$ respectively. For building up solutions, we need to process all the children of a node before we process the node itself.

```
To find a minimum-weight dominating set of a tree T:
    Root the tree at a node r.
    For all nodes u of T in post-order
        If u is a leaf then set the values:
            Mo[u] = ∞
            Mi[u] = c(u)
            Mu[u] = 0
        Else set the values:
            Mi[u] = c(u) + Σ_{v∈children(u)} Mu[v].
            Mu[u] = Σ_{v∈children(u)} min(Mi[v], Mo[v]).
```

[1] ex573.411.14

$$Mo[u] = \min_{v \in children(u)} Mi[v] + \sum_{w \in children(u), w \neq v} \min(Mo[v], Mi[v])$$

```
    Endif
Endfor
Return max(Mo[r], Mi[r]).
```

The algorithms clearly runs in polynomial time, as there are $3n$ subproblems for an n node tree, and each value associated with a subproblem of u of degree n_u can be determined in $O(n_u)$ time. So the total time is $O(\sum_u n_u) = O(n)$.

(b) We extend the algorithm for the case of bounded tree-width by having subproblems associated with the nodes of the tree-decomposition. For each node t of the tree-decomposition, let V_t be the subset of nodes corresponding to tree-node t, T_t the subtree rooted at t, and G_t the corresponding subgraph. Now for each disjoint sets $U, W \subset V_t$ we will define a subproblem and have $OPT(t, U, W)$ the minimum weight of a set in G_t that contains exactly the nodes U in V_t and covers all nodes of G_t except possibly not dominating the a subset of W. Recall that in the case of a tree, the recurrence for $OPT_{out}(u)$ was a little awkward, as we needed to select a child of u that is covering u. Here we would need to do this for each node in $V_t - (U \cup W)$. To make this simpler to write, we will define more subproblems. Let t be a node of the tree decomposition, and let $t_1, \ldots, t_d$ be its children, then we define subproblems $OPT(t, i, U, W)$ for each $0 \leq i \leq d$ to be the minimum weight of a set in the graph corresponding to the union of subtrees $T_{t_1}, \ldots, T_{t_i}$ and the set V_t that contains exactly the nodes U in V_t and dominates all nodes of this subgraph except possibly not dominating the a subset of W.

So if d_r is the degree of the root, then the optimum value we are looking for is $\min_{U \subset V_r} OPT(r, d_r, U, \emptyset)$ For any node t we have $OPT(t, 0, U, W) = \sum_{u \in U} c(u)$ if U dominates the nodes $V_t - W$, and ∞ otherwise. This defines the values at the leafs.

Given the subproblems, we will get the value at a node t using the values for smaller i, and the values at the children of t as follows. For a set U we will use the notation $c(U) = \sum_{u \in U} c(u)$, and $\delta(U)$ is the set dominated by U. Let t be a node of the tree decomposition and $t_1, \ldots, t_d$ its children, let n_i be the degree of child i.

(2) *The value of $OPT(t, i, U, W)$ for $i \geq 1$ is given by the following recurrence:*

$$\begin{aligned} OPT(t, i, U, W) \;=\; & c(U) + \min_{U_i \subseteq V_{t_i} : U_i \cap V_t = U \cap V_{t_i}} (OPT(t, i-1, U, W \cup \delta(U_i)) \\ & + OPT(t_i, n_i, U_i, (\delta(U) \cup W) \cap V_{t_i}) - c(U \cap V_{t_i})) \end{aligned}$$

For a tree-decomposition of width k there are 3^{k+1} subproblems associated with a (t, i) pair, and there are n such pairs if the tree is of size n. Computing each value takes only $O(1)$ time, so the total time required is $O(3^k n)$.

Let $(T; \{V_t | t \in T\})$ be the given tree decomposition rooted at r. There are k $(s_i; t_i)$ terminal pairs. We focus on the tree-width 2 case. For convenience, we assume that there are no two pieces V_{t_1} and V_{t_2} where (t_1, t_2) is an edge and $V_{t_1} \subset V_{t_2}$. Consider the subgraph G_t. Note that there can be at most one i for which P_i both enters and leaves G_t, since any such path uses up at least 2 vertices of V_t. Note also that there can be at most 3 s_i-t_i terminal pairs that have one end in G_t and the other one outside (as the paths connecting such pairs must go through V_t.

- If there are 3 such pairs, that each node $v \in V_t$ must be connected via disjoint paths to one of them, and terminal pairs inside G_t must connect via paths inside G_t. There are $O(1)$ cases here to consider depending on which of the nodes in V_t is used to connect which of the 3 separated terminal pairs.

- If there are 2 such terminal pairs, than of the at most 3 nodes in V_t 2 must be connected via disjoint paths to one of them, the third is either not used in any of the paths or is used by path connecting two terminals s_i and t_i inside, or two terminals outside G_t. Now there are $O(k)$ cases to consider, depending on which terminal pair i is using the extra node.

- If there is only one such pair, than we can have one pair i with both terminals inside or outside of G_t, that uses 2 nodes in V_t, or the one path leaving G_t, can leave, come back and leave again, or one or two paths can use just one node in V_t while having both terminals inside or both outside of G_t. There are $O(k^2)$ cases to consider here.

- If there are no such pairs, than one path can use 2 or 3 nodes in V_t, or multiple paths can use one node each. Now there are $O(k^3)$ cases to consider.

We define multiple subproblems for each t according to the possibilities discussed above. For the at most 3 nodes in V_t there are at most $O(k^3)$ possible cases. This defines $Ok^3n)$ subproblems. The value of a subproblem is simply 0 or 1 (or true or false) depending whether or not there are disjoint paths in G_t that satisfy the state of the nodes in V_t corresponding to the subproblem, that is, connect each $v \in V_t$ to the terminal in question inside G_t (and possibly connect the two nodes in V_t to each other, if needed), via disjoint paths inside G_t. The desired disjoint paths exists if and only if the value of one of the subproblems that connects all terminal pairs within the subgraph G_r (which is the whole graph).

Given values for all the subproblems associated with the children $t_1, \ldots, t_d$ of a node t, we want to get the value of the given subproblem efficiently. To do this consider a node $v \in V_t$, the subproblem under question wants a particular paths P_i to go through this vertex in $O(d)$ time.

[1] ex209.650.476

Checking whether G is 1- or 2-colorable is easy. For $k = 3, 4, \ldots, w+1$, we test whether G is k-colorable by dynamic programming. We use notation similar to what we used in the Maximum-Weight Independent Set problem for graphs of bounded tree-width. Let $(T, \{V_t : t \in T\})$ be a tree decomposition of G. For the subtree rooted at t, and every coloring χ of V_t using the color set $\{1, 2, \ldots, k\}$, we have a predicate $q_t(\chi)$ that says whether there is a k-coloring of G_t that is equal to χ when restricted to V_t. This requires us to maintain $k^{(w+1)} \leq (w+1)^{(w+1)}$ values for each piece of the tree decomposition.

We compute the values $q_t(\chi)$ when t is a leaf by simply trying all possible colorings of G_t. In general, suppose t has children $t_1, \ldots, t_d$, and we know the values of $q_{t_i}(\chi)$ for each choice of t_i and χ. Then there is a coloring of G_t consistent with χ on V_t if and only if there are colorings of the subgraphs $G_{t_1}, \ldots, G_{t_d}$ that are consistent with χ on the parts of V_{t_i} that intersect with V_t. Thus we set $q_t(\chi)$ equal to *true* if and only if there are colorings χ_i of V_{t_i} such that $q_{t_i}(\chi_i) = \mathit{true}$ and χ_i is the same as χ when restricted to $V_t \cap V_{t_i}$.

[1] ex897.854.812

We build the following bipartite graph G. There is a node u_i for each variable x_i, a node v_j for each clause C_j, and an edge (u_i, v_j) whenever x_i appears in C_j.

We first note that since each variable appears three times, and each clause has length three, the number of nodes on the left side of G equals the number of nodes on the right side. More strongly, we claim that G in fact has a perfect matching.

This is a consequence of a more general claim: that if every node in a bipartite graph G has the same degree d, then G has a perfect matching. (Here $d = 3$.) Indeed, if G does not have a perfect matching, then by Hall's Theorem it has a set A on the left side for which $|A| < |\Gamma(A)|$, where $\Gamma(A)$ denotes the set of neighbors of any node in A. But A has $d|A|$ nodes coming out of it, and $\Gamma(A)$ can only absorb $d|\Gamma(A)|$ of them, so this is not possible.

Consider the perfect matching in the graph G we constructed from the *3-SAT* instance. For each variable, we set it in a way that satisfies the clause it is matched to. In this way, we satisfy the full collection of clauses.

[1] ex592.206.332

We root the tree at some node r and associate a subtree T_v with each node $v \in T$, and let n_v denote the number of nodes in the subtree T_v. A solutions for the graph partitioning problem may split the subtree T_v in any ratio, however, it is not hard to see that of (A, B) is the maximum weight cut splitting the tree T into two equal sides, and assume $v \in A$ and $|A \cap T_v| = k$, then the induced partition of T_v by having $A_v = A \cap T_v$, and $B_v = A \cap T_v$ is the maximum weight cut separating T_v into two sides, where the side containing v has size k. This is true as the only interaction of the partition of T_v with the remaining graph is through node v. More precisely, assume it is false, and consider the optimal such cut (A', B'). Replacing A_v by A' keeps all edges across the cut, expect replacing edges that cross the (A_v, B_v) cut with edges that cross the (A', B') cut, and hence it results in a cut of larger weight. This contradiction proves the claim.

Given the above observation, we define subproblems for each subtree T_v and each integer $k \leq n_v$, where $OPT(v, k)$ is the maximum cut of the subgraph T_v separating T_v into two sides where the side containing v has size k. The final answer is $OPT(r, n/2)$ if $n = n_r$ is the number of nodes in T.

We will use $c(u, v)$ as the cost or weight of the edge $e = (v, w)$ of the tree. For a leaf v we have $k = 1$ as the only possible value and $OPT(v, 1) = 0$. Now consider $OPT(v, k)$ for some non-leaf v. The side A containing v must contain $k - 1$ nodes in addition to node v. If v has only one child u then we need to consider two cases depending on whether or not the child u is on the same side of the cut as v, so we get that in this case

$$OPT(v, k) = \max(opt(u, k - 1), c(v, u) + OPT(u, n_u - k + 1)).$$

If v has two children u and w, then we will consider all possible ways of dividing these $k - 1 = \ell_1 + \ell_2$ nodes between the two subtrees rooted at the two children of v. For each such division, there are further case depending whether or not the root of these subtrees is on the same side of the cut as v. We get the following recurrence in this case

$$\begin{aligned} OPT(v, k) \quad = \quad & \max_{0 \leq \ell_1 \leq k-1, \ell_2 = k-1-\ell_1} (\max(OPT(u, \ell_1), c(v, u) + OPT(u, n_u - \ell_1)) \\ & + \max(OPT(w, \ell_2), c(v, w) + OPT(w, n_w - \ell_2)). \end{aligned}$$

There are $O(n^2)$ subproblems. We can compute the values of the subproblems starting at the leaves, and gradually considering bigger subtrees. This recurrence allows us to compute the value of a subproblem in $O(1)$ time given the values of the subproblems associated with smaller trees. So the total time required is $O(n^2)$.

[1] ex34.58.909

(a) Let $\{w_1, w_2, w_3\} = \{1, 2, 1\}$, and $K = 2$. Then the greedy algorithm here will use three trucks, whereas there is a way to use just two.

(b) Let $W = \sum_i w_i$. Note that in *any* solution, each truck holds at most K units of weight, so W/K is a lower bound on the number of trucks needed.

Suppose the number of trucks used by our greedy algorithm is an odd number $m = 2q+1$. (The case when m is even is essentially the same, but a little easier.) Divide the trucks used into consecutive groups of two, for a total of $q + 1$ groups. In each group but the last, the total weight of containers must be *strictly* greater than K (else, the second truck in the group would not have been started then) — thus, $W > qK$, and so $W/K > q$. It follows by our argument above that the optimum solution uses at least $q + 1$ trucks, which is within a factor of 2 of $m = 2q + 1$.

[1] ex667.592.236

(a) For each protein p, we define a set S_p consisting of all proteins similar to it; we do this by simply enumerating all proteins q for which $d(p, q) \leq \Delta$. With respect to these sets, a representative set $R \subseteq P$ is simply a set for which $\{S_p : p \in R\}$ if a set cover for P.

Thus, to approximate the size of the smallest representative set, we can use the approximation algorithm for Set Cover from this chapter, obtaining an approximation guarantee of $O(\log n)$.

(b) The problem with using the approximation algorithm for Center Selection is that we'd obtain a set R of proteins for which every protein is within distance 2Δ of some element of R. But this doesn't satisfy the requirements for a representative, which stipulated that every protein had to be within distance Δ of some element of R.

[1] ex815.903.104

(a) Let $a_1 = 1$ and $a_2 = 100$, and consider the bound $B = 100$. Only a_1 will be chosen, while an optimal solution would choose a_2.

(b) In fact, this can be done in $O(n)$ time. We first go through all the numbers a_i and delete any whose value exceeds B — such numbers cannot be used in any solution, including the optimal one, so we have not changed the value of the optimum by doing this.

We then go through the numbers $a_1, a_2, \ldots, a_n$ in order until the sum of numbers we've seen so far first exceeds B. Let a_j be the number on which this happens. Thus we have $\sum_{i-1}^{j} a_i \geq B$, but $\sum_{i-1}^{j-1} a_i \leq B$ and also $a_j \leq B$. Thus, one of the sets $\{a_1, a_2, \ldots, a_{j\ 1}\}$ or $\{a_j\}$ is at least $B/2$ and at most B; we select this set as our solution. Since the optimum has a sum of at most B, our solution is at least half the optimal value.

[1] ex650.691.264

Note that in case when all sets B_i have exactly 2 elements (i.e. $b = 2$), the Hitting Set problem is equivalent to the Vertex Cover problem (two-element sets B_i correspond to edges). In the chapter we saw two approximation algorithm for Vertex Cover; here we generalize the one based on linear programming to arbitrary b.

Consider the following problem for Linear Programming:

$$\begin{aligned} \text{Min} \quad & \sum_{i=1}^{n} w_i x_i \\ \text{s.t.} \quad & 0 \le x_i \le 1 \qquad \text{for all } i = 1, \ldots, n \\ & \sum_{i:a_i \in B_j} x_i \ge 1 \quad \text{for all } j = 1, \ldots, m \text{ (all sets are hit)} \end{aligned}$$

Let x be the solution of this problem, and w_{LP} is a value of this solution (i.e. $w_{LP} = \sum_{i=1}^{n} w_i x_i$).

Now define the set S to be all those elements where $x_i \ge 1/b$:

$$S = \{a_i \mid x_i \ge 1/b\}$$

(1) *S is a hitting set.*

Proof. We want to prove that any set B_j intersects with S. We know that the sum of all x_i where $a_i \in B_j$ is at least 1. The set B_j contains at most b elements. Therefore some $x_i \ge 1/b$, for some $a_i \in B_j$. By definition of S, this element $a_i \in S$. So, B_j intersects with S by a_i. ■

(2 *The total weight of all elements in S is at most $b \cdot w_{LP}$.*

Proof. For each $a_i \in S$ we know that $x_i > 1/b$, i.e., $1 < bx_i$. Therefore

$$w(S) = \sum_{a_i \in S} w_i \le \sum_{a_i \in S} w_i \cdot bx_i \le b \sum_{i=1}^{n} w_i x_i = b w_{LP}$$

■

(3 *Let S^* be the optimal hitting set. Then $w_{LP} \le w(S^*)$.*

Proof. Set $x_i = 1$ if a_i is in S^*, and $x_i = 0$ otherwise. Then the vector x satisfy constrains of our problem for Linear Programming:

$$\begin{aligned} & 0 \le x_i \le 1 \qquad \text{for all } i = 1, \ldots, n \\ & \sum_{i:a_i \in B_j} x_i \ge 1 \quad \text{for all } j = 1, \ldots, m \text{ (because all sets are hit)} \end{aligned}$$

Therefore the optimal solution is not worse that this particular one. That is,

$$w_{LP} \le \sum_{i=1}^{n} w_i x_i = \sum_{a_i \in S} w_i = w(S^*)$$

■

Therefore we have a hitting set S, such that $w(S) \le b \cdot w(S^*)$.

[1] ex53.496.888

In the textbook we prove that

$$T - t_j \leq T^*.$$

where T is our makespan, t_j is a size of a job and T^* is the optimal makespan. We also proved that the optimal makespan is at least the average load, which is at least 300 in our case:

$$T^* \geq \frac{1}{m} \sum_j t_j \geq \frac{1}{10} 3000 = 300.$$

We also know that $t_j \leq 50$. Therefore the ratio of difference between our makespan and the optimal makespan to the optimal makespan is at most

$$\frac{T - T^*}{T^*} \leq \frac{t_j}{T^*} \leq \frac{50}{300} = \frac{1}{6} \leq 20\%$$

[1] ex995.878.454

One way to do this works as follows: When each job arrives, we put it on the machine that currently ends the soonest. (Note that this determination involves taking into account the speeds of the machines.)

To give a bound on this algorithm, we first give some lower bounds on the optimum makespan T^*. The total time of all jobs is $\sum_j t_j$. Let

$$t = \frac{\sum_j t_j}{m + 2k}.$$

If jobs could be assigned to machines so that each slow machine had a set of jobs summing to less than t, and each fast machine had a set of jobs summing to less than $2t$, then we would have

$$\sum_j t_j < mt + 2kt = \sum_j t_j,$$

a contradiction. Thus, some machine runs for at least t time units, and hence

$$T^* \geq \frac{\sum_j t_j}{m + 2k}.$$

Also, we have

$$T^* \geq \frac{1}{2} t_j,$$

for every job j, since at best it runs on one of the fast machines.

Let $M(r)$ denote the set of jobs assigned to machine r. Consider a machine i that achieves the makespan, and let j be the last job to go on it. Let x denote the time it uses for all jobs before j. (This means that $\sum_{j \in M(i)} t_j$ is equal to x if it's a slow machine, and it is equal to $2x$ if it's a fast machine.) Then at the moment j is added, every slow machine s has $\sum_{j \in M(s)} t_j \geq x$, and and every fast machine f has $\sum_{j \in M(f)} t_j \geq 2x$. Thus we have $\sum_j t_j \geq mx + 2kx$, and hence $T^* \geq x$. Also, $2T^* \geq t_j$.

Since the makespan is achieved by i, it is at most $x + t_j \leq T^* + 2T^* = 3T^*$.

An alternate solution is to simply sort the jobs in decreasing order of size, and then run the Greedy-Balance algorithm as though all machines were slow. We know from the chapter that this would give a $\frac{3}{2}$-approximation if all machines really were slow. However, we are comparing to the optimum as though all its machines are slow; in reality, the optimum's makespan might be half as large as we think, since some of its machines are fast. Thus, this gives a 3-approximation.

[1] ex829.220.704

(a) We process the customers in an arbitrary order. At any given point in time, let V_j denote the total value of all customers who have been shown ad j. As we see each new customer, we show him or her the ad for which V_j is as small as possible.

Let s' denote the spread of this algorithm. We first claim that $s' \geq \overline{v}/2m$. To prove this, suppose that ad j is the one achieving the spread (i.e., $V_j = s'$), and let i be any other ad. Let c be the last customer to be shown ad i. Before c was shown ad i, the value of V_i was at most V_j (by the definition of our greedy algorithm), and so $V_i \leq V_j + v_c \leq V_j + (\overline{v}/2m)$ by our assumption about the maximum customer value. Thus, if $s' = V_j < \overline{v}/2m$, then

$$\overline{v} = \sum_j V_j < V_j + (m-1)(V_j + (\overline{v}/2m)) < mV_j + (\overline{v}/2m) < (\overline{v}/2m) + (\overline{v}/2m) = \overline{v},$$

a contradiction.

Next we claim that the optimum spread s satisfies $s \leq \overline{v}/m$. Indeed, the total customer value is $\overline{v}$, and there are m ads, so one must be allocated at most a customer value of $\overline{v}/m$.

Combining these two claims, we get $s \leq \overline{v}/m \leq 2s'$.

(b) Suppose the input begins with $N + m$ customers of value 1, for some very large N, and then $m/2$ customers of value 2. (Suppose m is even and N is divisible by m.) Then our greedy algorithm will produce a spread of $1 + N/m$, while the optimal spread is $2 + N/m$, obtained by grouping the final m customers of value 1 onto $m/2$ ads, and showing the remaining $m/2$ ads to the customers of value 2.

[1] ex43.640.595

The conjecture is true. Consider the assignment of jobs to machines in an arbitrary optimal solution, and order the jobs arbitrarily on each machine. We say that the *base height* of a job j is the total time requirements of all jobs that precede it on its assigned machine.

We order all jobs by their base heights (breaking ties arbitrarily), and we feed them to the Greedy-Balance algorithm in this order. (We will label the jobs $1, 2, \ldots, n$ according to this order.)

We claim the following by induction on r: after the first r jobs have been processed by Greedy-Balance, the set of machine loads is the same as the set of machine loads if we consider the assignment of these r jobs made by the optimal solution.

This is clearly true for $r = 1$, since one machine will have load t_1, and all others will have load 0. Now suppose it is true up to some r, with loads $T_1, \ldots, T_m$, and consider job $r + 1$. Because we have sorted jobs by base height, job $r + 1$ comes from the machine that, in the optimal solution, has load $\min_i T_i$. By the definition of Greedy-Balance, this is the machine on which job $r + 1$ will be placed, giving it a load of $t_{r+1} + \min_i T_i$. This completes the induction step.

[1] ex286.347.713

We will use the following simple algorithm. Consider triples of T in any order, and add them if they do not conflict with previously added triples. Let M denote the set returning by this algorithm and M^* be the optimal three-dimensional matching.

(1) *The size of M is at least $1/3$ of the size of M^*.*

Proof. Each triple (a, b, c) in M^* must intersect at least one triple in our matching M (or else we could extend M greedily with (a, b, c)). One triple in M can only be in conflict with at most 3 triples in M^* as edges in M^* are disjoint. So M^* can have at most 3 times as many edges as M has. ■

[1] ex271.721.76

(a) If $v \notin S$, it must have never been chosen by the greedy algorithm. This means that it was deleted in some iteration by the selection of a node v': by the definition of the selection rule, this node v' must both be a neighbor of v, and have at least as much weight as v.

(b) Consider any other independent set T. For each node $v \in T$, we *charge* it to a node in S as follows. If $v \in S$, then we charge v to itself. Otherwise, by (a), v is a neighbor of some node $v' \in S$ whose weight is at least as large. We charge v to v'.

Now, if v is charged to itself, then no other node is charged to v, since S and T are independent sets. Otherwise, at most four neighboring nodes of no greater weight are charged to v. Either way, the total weight of all nodes charged to v is at most $4w(v)$. Since these charges account for the total weight of T, it follows that the total weight of nodes in T is at most four times the total weight of nodes in S.

[1] ex727.874.96

This means that our knapsack has capacity $(1+2\epsilon)W$. We throw out all items of weight exceeding W, since these can't be used in the solution we're comparing against.

We now round all remaining weights down to the nearest multiple of $\epsilon W/n$, and then multiply them all by $n/(\epsilon W)$. This means that all weights are now integers between 0 and n/ϵ. (Note that the items of weight less than $\epsilon W/n$ do get rounded down to 0, and yes, this means we will probably take them all, but as we'll see this is not a problem.)

So in time polynomial in n and $1/\epsilon$, by the dynamic programming algorithm for the knapsack problem with small weights, we can find the subset of (rounded) weight at most W which achieves the greatest value. The solution of (true) weight W and value V that was promised to exist must be available as an option, since its weight only went down, and since we find the best subset, we find one of value at least V. When we put all these items in our knapsack, each has a weight that may be up to $\epsilon W/n$ more than we thought (due to rounding down), so we use a weight of at most $W + n(\epsilon W/n) = W(1+\epsilon)$.

[1] ex662.412.328

We'll select the sites and the users they cover using the idea of the Set-Cover greedy algorithm. If a side s is used to cover the a subset U_s of users, then the average user cost is $(f_s + \sum_{u \in U_s} d_{us})/|U_s|$. The idea behind the greedy algorithm is to select the site s with a subset U_s that minimizes this quantity. First we need to argue that this minimum can be found.

(1) *Given a set R of uncovered users, and a site possible s, one can find the subset $U_s \subset U$ that minimizes the average cost $(f_s + \sum_{u \in U_s} d_{us})/|U_s|$ in polynomial time.*

Proof. Sort the users by increasing distance d_{us} from site s. The set U_s will be an initial set of this sorted sequence: $U_s = \{u \in R : d_{su} \leq \alpha\}$ for some value α. ■

Now the algorithm will be analogous to the Set Cover greedy algorithm. We select sites s with subsets U_s by the above greedy rule: selecting the site and the set that minimizes the average cost of covering a new user. There is one more option to consider. Suppose T is the subset of sites already selected. For a site $s \in T$ we can add a new node $u \in R$ to U_s, covering the new user u at the cost of d_{us}. In the algorithm given below, we will also save the cost c_u at which user u got covered by the algorithm. These values will be used by the analysis.

```
Start with R = U and T = ∅.
While R is not empty
   Let c = min_{u∈R,s∈T} d_us
   Select s ∈ S−T, and set U_s ⊂ R that minimizes
          c' − (f_s + Σ_{u∈U_s} d_us)/|U_s|.
   If c' ≤ c then
      Select the site s and set U_s used to obtain c' above.
      Add s to T, and delete U_s from R.
      Set c_u = c' for all u ∈ U_s.
   Else
      Select s and u obtaining the first minimum.
      Add u to U_s,
      Set c_u = c.
Endwhile
```

First, note that if we select the set of sites T, and have each site $s \in T$ cover the users in U_s then we get a solution to the problem with total cost $\sum_{u \in U} c_u$. Also, the algorithm runs in polynomial time. It remains to show that this is an $H(n)$ approximation algorithm.

The proof of the approximation ratio follows very closely the proof for the set cover algorithm. Consider an optimum solution. Assume it contains a subset T^* of sites, and $s \in T^*$ is used to cover a set U_s^* of users. The cost of using s to cover U_s^* is $f_s + \sum_{u \in U_s^*} d_{us}$. We will want to compare the optimum's cost, and $\sum_{u \in U_s^*} c_u$, which is the cost our greedy algorithm paid for the users in U_s^*.

[1] ex37.588.671

(2) *Using the notation introduced above, and the costs defined by the algorithm, we have that* $\sum_{u \in U_s^*} c_u \leq H(d)(f_s + \sum_{u \in U_s^*} d_{us})$, *where* $d = |U_s^*|$.

Proof. For notational simplicity, let $C = f_s + \sum_{u \in U_s^*} d_{us}$. Consider the elements in U_s^* in the order the algorithm covered them. Assume they are $u_1, u_2, \ldots, u_d$. Consider the moment the algorithm covers the ith node u_i from U_s^*. There are two cases to consider.

Case 1 At this point of the algorithm $s \notin T$.

Case 2 At this point of the algorithm $s \in T$.

When the algorithm covered u_i it selected the smallest average cost. In Case 1 this implies that the cost c_{u_i} is at most the cost of selecting cite s with the set $U_s^* \cap R$, which is at most $c_{u_i} \leq C/(d - i + 1)$ (as $i - 1$ previously covered nodes are no longer in the set). In Case 2, this implies that $c_{u_i} \leq d_{us}$. Assume that Case 1 applies when the first k nodes are covered, and after that Case 2 applies (k may be equal to d). Now summing all costs in U_s^* we get that

$$\sum_{u \in U_s^*} c_u \leq C/d + C/(d-1) + \ldots C/(d-k+1) + \sum_{i>d} d_{u_i,s}.$$

Now if $d = k$ then the upper bound on the cost is $H(d)C$ as claimed. If $k < d$ then note that the costs $\sum_{i>d} d_{u_i,s}$ is bounded by C, and so we also can bound the total cost by $H(d)C$. ∎

Now we are ready to prove that the algorithm is an $H(n)$ approximation algorithm. Let T^* and U_s^* be the optimal solution. The total cost of the solution is $\sum_{s \in T^*}(f_s + \sum_{u \in U_s^*} d_{us})$. We use the above Lemma to bound each term of the cost, and upper bound $II(d)$ by $II(n)$ for each set U_s^* in the optimum, to get the following.

$$OPT - \sum_{s \in T^*}(f_s + \sum_{u \in U_s^*} d_{us}) \leq \sum_{s \in T^*} H(n) \sum_{u \in U_s^*} c_u - H(n) \sum_{u \in U} c_u,$$

where the last sum is the algorithm's cost as claimed by the first Lemma.

The state-flipping algorithm will not always find this configuration

For example, Let G be a graph consisting of a cycle of length four: there are nodes v_1, v_2, v_3, v_4 and edges (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1). Then if we start the state-flipping algorithm in a configuration where nodes v_1 and v_2 have state $+1$, and nodes v_3 and v_4 have state -1, then no improving move is possible.

[1] ex652.266.676

(a) Consider a path with three edges, and an execution of the greedy algorithm in which the middle edge is added first.

(b) Consider the k connected components $C_1, \ldots, C_k$ of $M \cup M'$ — each is path or a cycle. Label a component C_i by an ordered pair $(|M \cap C_i|, |M' \cap C_i|)$. Now, if some C has a label of the form $(0, j)$, then it follows that $j = 1$, and this is an edge of M' that can be added to M. Otherwise, the labels are $\{x_i, y_i\}$, where $x_i \geq 1$ and $y_i \leq x_i + 1$ for each i. But then $|M'| - |M| = \sum_i (y_i - x_i) \leq k$ while $|M| = \sum_i x_i \geq k$, so we have $|M| \geq |M'| - |M|$. Rearranging this last inequality, we get $|M'| \leq 2|M|$.

Another way to prove this is the following. Since no edge of M' can be added to M, each $e \in M$ shares an endpoint with some $e' \in M'$. (It may share an endpoint with two edges in M'; then pick one arbitrarily.) Make the edge $e' \in M$ "pay for" the edge $e \in M$. Now, each edge $e \in M$ has been paid for by some edge $e' \in M'$, but each $e' \in M'$ has only two endpoints and hence pays for at most two edges in M. It follows that M' contains at most twice as many edges as M.

(c) Let M' be a matching of maximum size, and let M be the matching obtained by the greedy algorithm when it finally terminates. Then since there is no edge from M' that can be added to M, it follows from (a) that $|M| \geq \frac{1}{2}|M'|$.

[1] ex406.701.840

(a) We can assume by symmetry that the first machine M_1 has the higher load. We need to prove that $T_1 \leq 2T_2$. Notice that our assumption that no single job takes more than half of the processing time implies the machine with higher load must have at least two jobs. Let job j be the smallest job on machine M_1. Clearly, $T_1 \geq 2t_j$. The local search algorithm terminated, so moving job j from machine M_1 to machine M_2 does not decrease the difference between the processing times. This implies that $t_j \geq T_1 - T_2$. We get that $T_1 \leq t_j + T_2 \leq \frac{1}{2}T_1 + T_2$, and multiplying by two we get that $T_1 \leq 2T_2$.

(b) We observed above that if a job j satisfies $t_j \geq |T_1 - T_2|$ it cannot no move. Further, the difference $|T_1 - T_2|$ decreases throughout the algorithm, so once this condition holds, job j will never move.

Now consider a job j, and we aim to prove that j will move at most once. Assume that jobs j starts on machine M_1. So the first time it moves it will move from machine M_1 to machine M_2. We always move the largest job, so at this point all remaining jobs on machine M_1 will have processing time at most t_j. Consider the sequence of consecutive moves all from machine M_1 to M_2, and let $t_{j'}$ be the last job that moves in this direction (possibly $j = j'$). At this time $T_2 \geq T_1$ and $T_2 - T_1 \leq t_{j'}$ as before moving job j' machine M_1 had more work. Now we have that $t_j \geq t_{j'} \geq |T_1 - T_2|$, so by the observation above job j will not move again.

(c) As an example of a bad local optimum let machine M_1 have two jobs with processing time 3 each. While machine M_2 has two jobs with processing time 2 each. Now the difference in loads is 2, no single job can move, but swapping a pair of jobs yields a solution with identical loads.

[1] ex511.777.306

(a) Consider a progress measure Φ defined as the sum of the squares of the loads on all machines. We claim that after an improving swap move, this quantity must strictly decrease. Indeed, suppose we execute an improving swap move on machines M_1 and M_2; suppose the loads on M_i and M_j before the swap are T_i and T_j respectively, and the loads after the swap are T_i' and T_j'. Then Φ decreases by $T_i^2 + T_j^2$ and increases by $(T_i')^2 + (T_j')^2$; since $T_i + T_j = T_i' + T_j'$, and $\max(T_i', T_j') < \max(T_i, T_j)$, it follows that the decrease is larger than the increase: in other words, Φ strictly decreases.

Since there are only a finite number of ways to partition jobs across machines, Φ can only decrease a finite number of times. This implies that the algorithm terminates.

(b) Consider the machine M_i with the maximum load at the end of the algorithm. If M_i contains a single job, then since this job has to go *somewhere* in any solution, the makespan of our solution is in fact at most the optimal makespan, so our solution is optimal.

Otherwise, M_i has at least two jobs on it. We now claim that the load on M_i is at most twice the load on any other machine M_j. Since in particular this will apply to the machine M_j of minimum load, it follows also that M_i is at most twice the average, $\frac{1}{m}\sum_r t_r$, from which it follows that we are within a factor of 2 of optimal.

So suppose $T_i > 2T_j$. Since M_i has at least two jobs, the lightest job r on M_i has size $t_r \leq \frac{1}{2}T_i$. So consider the swap move in which job r simply moves to M_i. If T_i' and T_j' are the loads after this move, we have $T_i' < T_i$ and $T_j' = T_j + t_r < \frac{1}{2}T_i + \frac{1}{2}T_i = T_i$. Thus $\max(T_i', T_j') < \max(T_i, T_j)$, so this is an improving swap move, contradicting the termination of the algorithm.

[1] ex798.837.852

As this is a maximization problem, we need an upper bound of c^*, and there is an easy one:

$$c^* \leq m$$

where $m = |E|$.

The algorithm is: coloring every node independently with one of the three colors, each with probability $\frac{1}{3}$.

Let random variable

$$X_e = \begin{cases} 1 & \text{edge } e \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases}$$

Then for any given edge e, there are 9 ways to color its two ends, each of which appears with the same probability, and 3 of them are not satisfying.

$$Exp[X_e] = Pr[e \text{ is satisfied}] = \frac{6}{9} = \frac{2}{3}$$

Let Y be the random variable denoting the number of satisfied edges, then by linearity of expectations,

$$Exp[Y] = Exp\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} Exp[X_e] = \frac{2}{3}m \geq \frac{2}{3}c^*$$

[1] ex568.721.313

Number the voters $1, 2, \ldots, 100,000$, where voters 1 through 20000 are the Republican voters. Let X_i be the random variable equal to 0 if i votes for R, and 1 if i votes for D. So $X = \sum_{i=1}^{100000} X_i$.

Now, for $i \leq 20000$, $EX_i = .99 \cdot 0 + .01 \cdot 1 = .01$. For $i > 20000$, $EX_i = .01 \cdot 0 + .99 \cdot 1 = .99$. By linearity of expectation,

$$EX - \sum_{i=1}^{100000} EX_i - 20000 \cdot .01 + 80000 \cdot .99 - 79400.$$

[1] ex734.264.279

(a) Assume that using the described protocol, we get a set S that is not conflict free. Then there must be 2 processes P_i and P_j in the set S that both picked the value 1 and are going to want to share the same resource. But this contradicts the way our protocol was implemented, since we selected processes that picked the value 1 and whose set of conflicting processes all picked the value 0. Thus if P_i and P_j both picked the value 1, neither of them would be selected and so the resulting set S is conflict free. For each process P_i, the probability that it is selected depends on the fact that P_i picks the value 1 and all its d conflicting processes pick the value 0. Thus $P[P_i\text{selected}] = \frac{1}{2} * (\frac{1}{2})^d$. And since there are n processes that pick values independently, the expected size of the set S is $n * (\frac{1}{2})^{d+1}$

(b) Now a process P_i picks the value 1 with probability p and 0 with probability $1 - p$. So the probability that P_i is selected (i.e. P_i picks the value 1 and its d conflicting processes pick the value 0) is $p * (1 - p)^d$. Now we want to maximize the probability that a process is selected. Using calculus, we take the derivative of $p(1 - p)^d$ and set it equal to 0 to solve for the value of p that gives the objective it's maximum value. The derivative of $p(1 - p)^d$ is $(1 - p)^d - dp(1 - p)^{d-1}$. Solving for p, we get $p - \frac{1}{d+1}$. Thus the probability that a process is selected is $\frac{d^d}{(d+1)^{d+1}}$ and the expected size of the set S is $n * \frac{d^d}{(d+1)^{d+1}}$. Note that this is $\frac{n}{d}$ times $(1 - \frac{1}{d+1})^{d\,|\,1}$ and this later term is $\frac{1}{e}$ in the limit and so by changing the probability, we got a fraction of $\frac{n}{d}$ nodes. Note that with $p = 0.5$, we got an exponentially small subset in terms of d.

[1] ex131.386.529

(a) For every node v_k that comes later than v_j, i.e. $k > j$, it has probability $\frac{1}{k-1}$ to link to v_j, since v_k chooses from the $k-1$ existing nodes with equal probabilities. For all the nodes coming before v_j, such probability is obviously zero.

So the expected number of incoming links to node v_j is

$$\begin{aligned}\sum_{k=j+1}^{n} \frac{1}{k-1} &= \sum_{k=1}^{n-1} \frac{1}{k} - \sum_{k=1}^{j-1} \frac{1}{k} \\ &= H(n-1) - H(k-1) \\ &= \Theta(\ln n) - \Theta(\ln k) \\ &= \Theta(\ln \frac{n}{k})\end{aligned}$$

(b) Consider a node v_j, every node v_k with $k > j$ has probability $1 - \frac{1}{k-1}$ not to link to v_j. So if we have random variable X_j s.t.

$$X_j = \begin{cases} 1 & \text{node } v_j \text{ has no in-coming links} \\ 0 & \text{otherwise} \end{cases}$$

then

$$\begin{aligned}Exp[X_j] &= Pr[\text{no nodes links to } v_j] \\ &= \prod_{k=j+1}^{n} \left(1 - \frac{1}{k-1}\right) \\ &= \frac{j-1}{j} \cdot \frac{j}{j+1} \cdot \frac{j+1}{j+2} \cdots \frac{n-2}{n-1} \\ &= \frac{j-1}{n-1}\end{aligned}$$

Therefore, by linearity of expectations, we get the expected number of nodes without in-coming links

$$\sum_{j=1}^{n} Exp[X_j] = \sum_{j=1}^{n} \frac{j-1}{n-1} = \frac{1}{n-1} \sum_{j=1}^{n} (j-1) = \frac{1}{n-1} \cdot \frac{n(n-1)}{2} = \frac{n}{2}$$

[1] ex976.627.720

(a) Consider a clause C_i with n variables. The probability that the clause is not satisfied is $\frac{1}{2^m}$ and so the probability that it is satisfied is 1 less this quantity. The worst case is when C_i has just one variable, i.e. $n = 1$, in which case the probability of the clause being satisfied is $\frac{1}{2}$. Since there are k clauses, the expected number of clauses being satisfied is at least $\frac{k}{2}$. Consider the two clauses x_1 and $\overline{x_1}$. Clearly only one of these can be satisfied.

(b) For variables that occur in single variable clauses, let the probability of setting the variable so as to satisfy the clause be $p \geq \frac{1}{2}$. For all other variables, let the probabilities be $\frac{1}{2}$ as before. Now for a clause C_i with n variables, $n \geq 2$, the probability of satisfying it is at worst $(1 - \frac{1}{2^n}) \geq (1 - p^2)$ since $p \geq \frac{1}{2}$. Now to solve for p, we want to satisfy all clauses, so solve $p = 1 - p^2$ to get $p \approx 0.62$. And hence the expected number of satisfied clauses is $0.62n$.

(c) Let the total number of clauses be k. For each pair of single variable conflicting clauses, i.e. x_i and $\overline{x_i}$, remove one of them from the set of clauses. Assume we have removed m clauses. Then the maximum number of clauses we could satisfy is $k - m$. Now apply the algorithm described in the previous part of the problem to the $k - 2m$ clauses that had no conflict to begin with. The expected number of clauses we satisfy this way is $0.62 * (k - 2m)$. In addition to this we can also satisfy m of the $2m$ conflicting clauses and so we satisfy $0.62 * (k - 2m) + m \geq 0.62 * (k - m)$ clauses which is our desired target. Note that this algorithm is polynomial in the number of variables and clauses since we look at each clause once.

[1] ex633.413.669

We interpret the constraint (μ_i, μ_j, μ_k) to mean that we require one of the subsequences $\ldots, \mu_i, \ldots, \mu_j, \ldots, \mu_k, \ldots$ or $\ldots, \mu_k, \ldots, \mu_j, \ldots, \mu_i, \ldots$ to occur in the ordering of the markers. (One could also interpret it to mean that just the first of these subsequences occurs; this will affect the analysis below by a factor of 2.)

Suppose that we choose an order for the n markers uniformly at random. Let X_t denote the random variable whose value is 1 if the t^{th} constraint (μ_i, μ_j, μ_k) is satisfied, and 0 otherwise. The six possible subsequences of $\{\mu_i, \mu_j, \mu_k\}$ occur with equal probability, and two of them satisfy the constraint; thus $EX_t = \frac{1}{3}$. Hence if $X = \sum_t X_t$ gives the total number of constraints satisfied, we have $EX = \frac{1}{3}k$.

So if our random ordering satisfies a number of constraints that is at least the expectation, we have satisfied at least $\frac{1}{3}$ of all constraints, and hence at least $\frac{1}{3}$ of the maximum number of constraints that can be simultaneously satisfied.

We can extend this to construct an algorithm that *only* produces solutions within a factor of $\frac{1}{3}$ of optimal: We simply repeatedly generate random orderings until $\frac{1}{3}k$ of the constraints are satisfied. To bound the expected running time of this algorithm, we must give a lower bound on the probability p^+ that a single random ordering will satisfy at least the expected number of constraints; the expected running time will then be at most $1/p^+$ times the cost of a single iteration.

First note that k is at most n^3, and define $k' = \frac{1}{3}k$. Let k'' denote the greatest integer strictly less than k'. Let p_j denote the probability that we satisfy j of the constraints. Thus $p^+ - \sum_{j \geq k'} p_j$; we define $p^- - \sum_{j < k'} p_j - 1 - p^+$. Then we have

$$\begin{aligned} k' &= \sum_j j p_j \\ &= \sum_{j<k'} j p_j + \sum_{j \geq k'} j p_j \\ &\leq \sum_{j<k'} k'' p_j + \sum_{j \geq k'} n^3 p_j \\ &= k''(1 - p^+) + n^3 p^+ \end{aligned}$$

from which it follows that

$$(k'' + n^3)p^+ \geq k' - k'' \geq \frac{1}{3}.$$

Since $k'' \leq n^3$, we have $p^+ \geq \frac{1}{6n^3}$, and so we are done.

[1] ex449.507.100

First we give an algorithm that produces a subgraph whose expected number of edges has the desired value. For this, we simply choose k nodes uniformly at random from G. Now, for $i < j$, let X_{ij} be a random variable equal to 1 if there is an edge between our i^{th} and j^{th} node choices, and equal to 0 otherwise.

Of the $n(n-1)$ choices for i and j, there are $2m$ that yield an edge (since an edge (u, v) can be chosen either by picking u in position i and v in position j, or by picking v in position i and u in position j). Thus $E[X_{ij}] = \frac{2m}{n(n-1)}$.

The expected number of edges we get in total is

$$\sum_{i<j} E[X_{ij}] = \binom{k}{2} \cdot \frac{2m}{n(n-1)} = \frac{mk(k-1)}{n(n-1)}.$$

We now want to turn this into an algorithm with expected polynomial running time, which always produces a subgraph with at least this many edges. The analogous issue came up with MAX 3-SAT, and we use the same idea here: For this we use the same idea as in the analogous MAX 3-SAT: we run the above randomized algorithm repeatedly until it produces a subgraph with at least the desired number of edges.

Let p^+ be the probability that one iteration of this succeeds; our overall running time will be the (polynomial) time for one iteration, times $1/p^+$. First note that the maximum number of edges we can find is $e = \frac{k(k-1)}{2}$, and we're seeking $e' = e \cdot \frac{2m}{n(n-1)}$. Let e'' denote the greatest integer strictly less than e'. Let p_j denote the probability that we find a subgraph with exactly j edges. Thus $p^+ = \sum_{j>e'} p_j$; we define $p^- = \sum_{j<e'} p_j = 1 - p^+$. Then we have

$$\begin{aligned} e' &= \sum_j j p_j \\ &= \sum_{j<e'} j p_j + \sum_{j \geq e'} j p_j \\ &\leq \sum_{j<e'} e'' p_j + \sum_{j \geq e'} e p_j \\ &= e''(1-p^+) + \binom{k}{2} p^+ \end{aligned}$$

from which it follows that

$$(e'' + \binom{k}{2}) p^+ \geq e' - e'' \geq \frac{1}{n(n-1)}.$$

Since $e'' \leq \binom{k}{2}$, we have $p^+ \geq \frac{1}{k(k-1)n(n-1)}$, and so we are done.

[1] ex553.136.7

The strategy is as follows. The seller watches the first $n/2$ bids without accepting any of them. Let b^* be the highest bid among these. Then, in the final $n/2$ bids, the seller accepts any bid that is larger than b^*. (If there is no such bid, the seller simply accepts the final bid.)

Let b_i denote the highest bid, and b_j denote the second highest bid. Let S denote the underlying sample space, consisting of all permutations of the bids (since they can arrive in any order.) So $|S| = n!$. Let E denote the event that b_j occurs among the first $n/2$ bids, and b_i occurs among the final $n/2$ bids.

What is $|E|$? We can place b_j anywhere among the first $n/2$ bids ($n/2$ choices); then we can place b_i anywhere among the final $n/2$ bids ($n/2$ choices); and then we can order the remaining bids arbitrarily ($(n-2)!$ choices). Thus $|E| = \frac{1}{4}n^2(n-2)!$, and so

$$P[E] = \frac{n^2(n-2)!}{4n!} = \frac{n}{4(n-1)} \geq \frac{1}{4}.$$

Finally, if event E happens, then the strategy will accept the highest bid; so the highest bid is accepted with probability at least 1/4.

[1] ex437.89.251

Let X be a random variable equal to the number of times that b^* is updated. We write $X = X_1 + X_2 + \cdots + X_n$, where $X_i = 1$ if the i^{th} bid in order causes b^* to be updated, and $X_i = 0$ otherwise.

So $X_i = 1$ if and only if, focusing just on the sequence of the first i bids, the largest one comes at the end. But the largest value among the first i bids is equally likely to be anywhere, and hence $EX_i = 1/i$.

Alternately, the number of permutations in which the number at position i is larger than any of the numbers before it can be computed as follows. We can choose the first i numbers in $\binom{n}{i}$ ways, put the largest in position i, order the remainder in $(i-1)!$ ways, and order the subsequent $(n-i)$ numbers in $(n-i)!$ ways. Multiplying this together, we have $\binom{n}{i}(i-1)!(n-i)! = n!/i$. Dividing by $n!$, we get $EX_i = 1/i$.

Now, by linearity of expectation, we have $EX = \sum_{i=1}^{n} EX_i = \sum_{i=1}^{n} 1/i = H_n = \Theta(\log n)$.

[1] ex547.67.324

(a) Let's look at a given machine p. In order for it to have no job, every job must be assigned to a different machine. As the jobs are assigned randomly and uniformly, the probability that a given job j is not assigned to p is $(1 - \frac{1}{k})$ and therefore the probability that p doesn't get any job is $(1 - \frac{1}{k})^k$. Therefore the expected number of machines with no jobs is $N(k) = k(1 - \frac{1}{k})^k$.

Finally $N(k)/k = (1 - \frac{1}{k})^k$, which goes to $1/e$ as k goes to infinity. Also notice that in the limit the number of machines with no jobs is k/e.

(b) There is a very simple solution to this problem. We notice that the number of rejected jobs (denote it by N_{rej}) is the number of total jobs k minus the number of accepted jobs N_{acc} ($N_{rej} = k - N_{acc}$). The number of jobs accepted is the k minus the number of machines with no jobs N_{nojob} (since the rest of the people do exactly 1 job). Therefore $N_{rej} = k - N_{acc} = k - (k - N_{nojob}) = N_{nojob}$. Therefore the answer to part **(b)** is the same as the answer to part **(a)**.

(c) This part will involve slight calculations. We know that the number of machines with no jobs is k/e (from the first part). We first calculate the number of machines with exactly one job. Again look at a machine p. The probability that only 1 job is assigned to that machine is $k\frac{1}{k}(1 - \frac{1}{k})^{k-1}$. (The chance of a given job j being assigned to p is $1/k$ and the probability that the remaining jobs will not be assigned to p is $(1 - \frac{1}{k})^{k-1}$. Finally there are k choices of the "given" job j which puts the coefficient k in the beginning). Notice that this also in the limit $1/e$ therefore the number of machines with exactly 1 jobs is also k/e.

Finally the remaining machines regardless of how many jobs they were assigned will perform exactly two jobs. There are $k - \frac{2k}{e}$ of these.

The final tally is k/e machines with one job and $k - \frac{2k}{e}$ people with two jobs. Subtracting this from k (the total number of jobs) we get that $\frac{k(3-e)}{e}$ jobs are rejected, which is approximately 11%.

[1] ex16.34.694

Consider a graph G with nodes s and t, and $n-2$ other nodes $v_1, \ldots, v_{n-2}$. There are two parallel edges from s to each v_i, and one edge from v_i to t. The minimum s-t cut is to separate t by itself.

If we run the version of the contraction algorithm described in the problem, it will independently contract each of the length-2 paths from s to t in some order. In order for it to find the minimum s-t cut, it must contract each v_i into s, not into t. There is a $2/3$ chance of this happening for each i, so the probability that the minimum s-t cut is found is $(2/3)^{n-2}$, an exponentially small quantity.

(Note that this example poses no problem for the global minimum cut, which consists of any of the nodes v_i on its own.)

[1] ex242.186.32

The mean for X_2 is n, so in order to have $X_1 - X_2 > c\sqrt{n}$, we need $X_2 < E[X_2] - \frac{c}{2}\sqrt{n} = (1-\delta)E[X_2]$ for $\delta = \frac{c}{2\sqrt{n}}$. Plugging this into the Chernoff lower bound, the probability this happens is

$$e_{-\frac{1}{2}\delta^2 E[X_2]} = e^{-c^2/4}.$$

This can be made smaller than a constant ε by choosing the undetermined constant c large enough.

[1] ex646.944.578

(a) Let n be odd, $k = n^2$, and represent the set of basic processes as the disjoint union of n sets $X_1, \ldots, X_n$ of cardinality n each. The set of processes P_i associated with job J_i will be equal to $X_i \cup X_{i+1}$, addition taken modulo n.

We claim there is no perfectly balanced assignment of processes to machines. For suppose there were, and let Δ_i denote the number of processes in X_i assigned to machine M_1 minus the number of processes in X_i assigned to machine M_2. By the perfect balance property, we have $\Delta_{i+1} = -\Delta_i$ for each i; applying these equalities transitively, we obtain $\Delta_i = -\Delta_i$, and hence $\Delta_i = 0$, for each i. But this is not possible since n is odd.

(b) Consider independently assigning each process i a *label* L_i equal to either 0 or 1, chosen uniformly at random. Thus we may view the label L_i as a 0-1 random variable. Now for any job J_i, we assign each process in P_i to machine M_1 if its label is 0, and machine M_2 if its label is 1.

Consider the event E_i, that more than $\frac{4}{3}n$ of the processes associated with J_i end up on the same machine. The assignment will be nearly balanced if none of the E_i happen. E_i is precisely the event that $\sum_{t \in J_i} L_t$ either exceeds $\frac{4}{3}$ times its mean (equal to n), or that it falls below $\frac{2}{3}$ times its mean. Thus, we may upper-bound the probability of E_i as follows.

$$\begin{aligned}
\Pr[E_i] &\leq \Pr[\sum_{t \in J_i} L_t < \frac{2}{3}n] + \Pr[\sum_{t \in J_i} L_t > \frac{4}{3}n] \\
&\leq \left(e^{-\frac{1}{2}(\frac{1}{3})^2}\right)^n + \left(\frac{e^{\frac{1}{3}}}{\left(\frac{4}{3}\right)^{\frac{4}{3}}}\right)^n \\
&\leq 2 \cdot .96^n.
\end{aligned}$$

Thus, by the union bound, the probability that any of the events E_i happens is at most $2n \cdot .96^n$, which is at most .06 for $n \geq 200$.

Thus, our randomized algorithm is as follows. We perform a random allocation of each process to a machine as above, check if the resulting assignment is perfectly balanced, and repeat this process if it isn't. Each iteration takes polynomial time, and the expected number of iterations is simply the expected waiting time for an event of probability $1 - .06 - .94$, which is $1/.94 < 2$. Thus the expected running time is polynomial.

This analysis also proves the *existence* of a nearly balanced allocation for any set of jobs.

(Note that the algorithm can run forever, with probability 0. This doesn't cause a problem for the expectation, but we can deterministically guarantee termination without hurting the running time very much as follows. We first run k iterations of the randomized algorithm; if it still hasn't halted, we now find the nearly balanced assignment that is guaranteed to exist by trying all 2^k possible allocations of processes to machines, in time $O(n^2 \cdot 2^k)$. Since this brute-force step occurs with probability at most $.06^k$, it adds at most $O(n^2 \cdot .12^k) = O(n^2 \cdot .12^n) = o(1)$ to the expected running time.)

[1] ex41.971.873

We imagine dividing the set S into 20 *quantiles* $Q_1, \ldots, Q_{20}$, where Q_i consists of all elements that have at least $.05(i-1)n$ elements less than them, and at least $.05(20-i)n$ elements greater than them. Choosing the sample S' is like throwing a set of numbers at random into bins labeled with $Q_1, \ldots, Q_{20}$.

Suppose we choose $|S'| = 40,000$ and sample with replacement. Consider the event $\mathcal{E}$ that $|S' \cap Q_i|$ is between 1800 and 2200 for each i. If $\mathcal{E}$ occurs, then the first nine quantiles contain at most $19,800$ elements of S', and the last nine quantiles do as well. Hence the median of S' will belong to $Q_{10} \cup Q_{11}$, and thus will be a (.05)-approximate median of S.

The probability that a given Q_i contains more than 2200 elements can be computed using the Chernoff bound (4.1), with $\mu = 2000$ and $\delta = .1$; it is less than

$$\left[\frac{e^{.05}}{(1.05)^{(1.05)}}\right]^{10000} < .0001.$$

The probability that a given Q_i contains fewer than 1800 elements can be computed using the Chernoff bound (4.2), with $\mu = 2000$ and $\delta = .1$; it is less than

$$e^{-(.5)(.1)(.1)2000} < .0001.$$

Applying the Union Bound over the 20 choices of i, the probability that $\mathcal{E}$ does not occur is at most $(40)(.0001) = .004 < .01$.

[1] ex835.763.619

One algorithm is the following.

```
For i = 1, 2, ..., n
  Receiver j computes β_ij = f(β*_1 ··· β*_{i-1}, α_i^⟨j⟩).
  β*_i is set to the majority value of β_ij, for j = 1, ..., k.
  End for
Output β*
```

We'll make sure to choose an odd value of k to prevent ties.

Let $X_{ij} = 1$ if $\alpha_i^{\langle j \rangle}$ was corrupted, and 0 otherwise. If a majority of the bits in $\{\alpha_i^{\langle j \rangle} : j = 1, 2, \ldots, k\}$ are corrupted, then $X_i = \sum_j X_{ij} > k/2$. Now, since each bit is corrupted with probability $\frac{1}{4}$, $\mu = \sum_j EX_{ij} = k/4$. Thus, by the Chernoff bound, we have

$$\begin{aligned} \Pr[X_i > k/2] &- \Pr[X_i > 2\mu] \\ &< \left(\frac{e}{4}\right)^{k/4} \\ &\leq (.91)^k. \end{aligned}$$

Now, if

$$k \geq 11 \ln n > \frac{\ln n - \ln .1}{\ln(1/.91)},$$

then

$$\Pr[X_i > k/2] < .1/n.$$

(So it is enough to choose k to be the smallest odd integer greater than $11 \ln n$.) Thus, by the union bound, the probability that *any* of the sets $\{\alpha_i^{\langle j \rangle} : j = 1, 2, \ldots, k\}$ have a majority of corruptions is at most .1.

Assuming that a majority of the bits in each of these sets are not corrupted, which happens with probability at least .9, one can prove by induction on i that all the bits in the reconstructed message β^* will be correct.

[1] ex482.918.336

Let Y denote the number of steps in which your net profit is positive. Then $Y = Y_1 + Y_2 + \cdots + Y_n$, where $Y_k = 1$ if your net profit is positive at step k, and 0 otherwise.

Now, consider a particular step k. $Y_k = 1$ if and only if you have had more than $k/2$ steps in which your profit increased. Since the expected number of steps in which your profit increased is $k/3$, we can apply the Chernoff bound (4.1) with $\mu = k/3$ and $1 + \delta = 3/2$ to conclude that EY_k is bounded by

$$\left[\frac{e^{1/2}}{(3/2)^{(3/2)}}\right]^{(k/3)} < (.97)^k.$$

Thus,

$$EY = \sum_{k=1}^{n} EY_k < \sum_{k=1}^{n} (.97)^k < \frac{1}{1-(.97)} < 34,$$

which is a constant independent of n.

[1] ex251.139.906

(a) False. A bad example can consist of a single edge $e = (u, v)$. Assume the cost of u is 1 while the cost of v is more than $2c$. The minimum cost of a vertex cover is 1, while the algorithm selects node v with probability $1/2$, and hence has expected cost more than c. Alternately we could have u at most 0 and v at most 1. Now the algorithm's expected cost is $1/2$, while the optimum is 0.

(b) This is true. Let p_e be the probability that edge e is selected by the algorithm. Note that the algorithm, as given by the problem set, does not specify the selection rule of edges. You may select uncovered edges at random, or by smallest index, etc. The probability p_e will of course depend on what selection rule was used. But any selection rule gives rise to such probabilities. Now we need to notice two facts. First that $\sum_{e \in E} p_e$ is exactly, the expected number of nodes selected by the algorithm. This is true, as every time we select an edge e we add one node to the vertex cover.

Next we consider the sum of the probabilities p_e for edges adjacent to a vertex v. Let $\delta(v)$ denote the set of edges adjacent to vertex v, and consider $\sum_{e \in \delta(v)} p_e$. Note that this is exactly the expected number of edges selected that are adjacent to node v. Let $S(v)$ be the random variable indicating the selected edges adjacent to v. We have that $Exp(|S(v)|) = \sum_{e \in \delta(v)} p_e$. We claim that this expectation is at most 2. This is true as each time an edge in $\delta(v)$ is selected, with $1/2$ probability, we use node v cover edge e, and then all edges in $\delta(v)$ are covered, and no more edges in this set will be selected selected. To make this argument precise, let E_i denote the event that at least i edges are selected adjacent to i. Now we have the following inequality for the expected number of edges selected.

$$Exp(|S(v)|) - \sum_i iProb(E_i - E_{i+1}) - \sum_i Prob(E_i) \leq 1 + \sum_{i>1} 2^{i-1} \leq 2,$$

where the inequality $Prob(E_i) \leq 2^{i-1}$ follows for $I > 1$ as after each edge selected adjacent to v we add v to the vertex cover with probability $1/2$.

Now we are ready to bound the expected size of the vertex cover compared to the optimum. Let S^* be an optimum vertex cover.

$$\sum_e p_e \leq \sum_{v \in S^*} \sum_{e \in \delta(v)} p_e \leq \sum_{v \in S^*} 2 = 2|S^*|,$$

where the first inequality follows as S^* is a vertex cover, and so the second sum must cover each edge e.

[1] ex593.991.129