(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