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

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

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

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. ■

(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

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

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

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.

(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