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