SOLUTIONS

0.1 Area

The area of a rectangle with sides A_k and B_k is $A_k \times B_k$, so we simply need to add these products up for each of the rectangles.

As noted in the problem statement, this problem was intended purely to get contestants to try out the 64-bit integer types in their chosen languages. The only thing one must keep in mind is that multiplying two 32-bit values will yield a 32-bit result, even if assigned to a 64-bit variable. The safest thing to do is to use 64-bit variables throughout.

0.2 Hill

First we make an observation: starting at any point in the grid, we can "walk uphill" until reaching a cell with no adjacent higher cells, which is a hill. Furthermore, if we start from a cell with height h inside a region bounded entirely by cells with height less than h (or by the boundaries of the entire grid), then this walk cannot leave this region, and hence we know that a hill must exist inside this region.

We start the algorithm knowing that somewhere in the entire grid we have a hill. The algorithm then repeatedly finds smaller and smaller rectangles known to contain hills. At each step of the algorithm, we will have a rectangle with the following properties:

1. The highest cell seen so far falls inside this rectangle[1].
2. The rectangle is bounded by cells strictly lower than this highest cell, or by the boundaries of the original grid.

Although we do not actually perform an uphill walk, the property noted above guarantees that this rectangle will contain a hill. If the rectangle is 1×1, then it

[1] At the start of the algorithm, we have not seen any cells, but this turns out not to be very important.

consists of just a cell which is a hill, and the problem is finished. Otherwise, we can produce a smaller rectangle as follows.

First, use the laser scanner on every cell in a line that cuts the rectangle in half (either horizontally or vertically, whichever will use the fewest scans). Let H be the highest cell that has been seen so far (including the cells that have just been scanned). Now if H does not lie on the cut, then it falls into one half of the rectangle. This half then satisfies the properties above, and we have successfully reduced the size of the rectangle. If H lies on the cut, then some additional work is required. Scan the cells immediately adjacent to H that have not yet been scanned, and let H' be the new highest cell seen. If $H' = H$ then H is a hill (since we have scanned all its neighbours), and we can immediately terminate. Otherwise, H' does not lie on the cut, and we can proceed to select one half of the rectangle as before.

By repeatedly finding smaller rectangles known to contain a hill, we must eventually find a 1×1 rectangle and the problem is solved. An upper bound on the number of scans required is

$$1002 + 502 + 502 + 252 + 252 + \cdots + 5 + 5 + 3 = 3023$$

Slightly tighter or looser bounds can be obtained depending on exact details of the implementation, but this is not important as full points are awarded as long as the number of scans is at most 3050.

0.3 Museum

To solve this problem, we start by observing that if we have three vases with heights A, B and C, such that either A is odd and C even, or A even and C odd, then no matter what B is these three vases will not violate the condition of the exhibition organisers. This is because $A + C$ must therefore be odd, and so is not divisible by two, meaning that it is impossible for B to be the average of A and C.

We therefore start by arranging the vases such that we place all the even vases first, and all the odd vases second. This gives an arrangement that looks like this:

$$E_1 \quad E_2 \quad \ldots \quad E_p \quad O_1 \quad O_2 \quad \ldots \quad O_q$$

where $E_1, E_2, \ldots, E_p$ are the even heights in some order, and $O_1, O_2, \ldots, O_q$ are the odd heights in some order.

Now consider any heights A, B, C which violate the organisers' requirements. By the observations above, either A and C are both even (in which case B is even,

since it appears between A and C and all the even values are grouped together), or A and C are both odd (in which case B is also odd). In other words, we can consider the problems of ordering the even numbers and the odd numbers separately.

Now suppose that $a_1, a_2, \ldots, a_p$ is a permutation of the heights $1, 2, \ldots, p$ which satisfies the organisers' requirements (this is a smaller instance of the problem, so it can be solved by divide-and-conquer). Then simply assigning $E_i = 2a_i$ will satisfy the requirements on the even values. Similarly, given a permutation $b_1, b_2, \ldots, b_q$ of the heights $1, 2, \ldots, q$ which satisfies the requirements, we can assign $O_i = 2b_i - 1$.

Examining the properties of the resulting sequence gives another approach to generating the same solution. We can write all the heights in binary form, and then sort them according to the *reverse* of their binary form. This sorts first on the least significant bit (i.e., on whether they are odd or even), then the next least significant bit and so on. To prove that this solution is valid, note that if B is the average of A and C, then at the least significant bit that A and B first differ, A and C must have the same value for that bit, placing A and C in a separate group from B when sorting on that bit.

1.1 Archery

The proofs below are very verbose and long-winded, but the ideas behind the algorithms are not all that complicated. The steps can be summarized as:

1. A trivial solution (trying all possibilities and simulating the tournament for each one) gives an $O(N^2R)$ algorithm.

2. One can observe that after no more than $2N$ rounds, the tournament becomes cyclical with all positions repeating every N rounds. This allows the trivial algorithm to be sped up to $O(N^3)$.

3. One can optimize the simulation of the tournament (once we have chosen an initial position) from $O(N^2)$ to $O(N)$. This is the most complicated part of the solution. The key here is that we are only interested in our own position at the end, not in everyone else's.

4. Once you have a subroutine that tells you where you will end up given where you start, you can use it with a modified binary search, improving the $O(N^2)$ algorithm to $O(N \log N)$; or alternatively, improving the $O(N^3)$ algorithm to $O(N^2 \log N)$.

5. The last two algorithms above also have slower versions ($O(N^2 \log N)$ and $O(N \log N \log N)$) if you try to solve the problem by also keeping track of other archers' positions, not just your own.

Optimising to $O(N^3)$

A trivial solution is to try all possible starting positions and simulate the tournament for each round, giving complexity of $O(N^2R)$. We now show how this can be reduced to $O(N^3)$.

Consider the archers numbered from $N+2$ to $2N$. Let's call them the weak archers.

Theorem 1. *After enough rounds (no more than $2N$) the weak archers will occupy the targets numbered 2 to N, one such archer on each target, and will stay there until the end of the tournament.*

Proof. After $N-1$ rounds, archer number 1 will be on target 1 and will stay there until the end. From this point on, if we consider the set of N archers consisting of archer number 1 plus the $N-1$ weak archers (let us call this the *weak+1* set), and if we imagine the targets arranged in a circle (1 to N and then again 1), we have the following scenario:

- When an archer from the weak+1 set competes with an archer outside of weak+1, then the weak+1 archer will stay on the target and the other archer will move.
- When two archers belonging to the weak+1 set compete against each other, one of them will stay and the other will move to the target on his left.

Lemma 1. *Within N rounds after archer number 1 has arrived on target 1, every target will have at least one weak+1 member on it.*

Proof. Suppose the above is not true. We know that once a target is occupied by a weak+1 member, then it will always be occupied by at least one (because weak+1 members never move out of their target unless there is another weak+1 archer to replace them there). Thus if Lemma 1 is false, there must exist a target that is never occupied by a weak+1 member (within the N rounds). Let's call this target A. If A is never occupied by a weak+1 archer, then the target to the left of A (let us call it B) would have at most one such archer within one round and would remain this way. Then within two rounds the target to the left of B would have

at most one weak+1 archer, and within three rounds the next target to the left would have at most one such archer. Continuing around the circle, within $N-1$ rounds the target to the right of A would have at most one weak+1 archer. Thus within $N-1$ rounds all targets except A would have at most one weak+1 archer. But since there are N such archers and N targets, this means that A must have at least one weak+1 archer. Since this contradicts our supposition that A remains free of weak+1 archer for N rounds, this proves Lemma 1. □

Now that we know every target has at least one weak+1 archer within $2N$ rounds from the start of the competition, and since we know that once a target has such an archer it never ceases to have at least one, we have proved Theorem 1. □

Now consider all archers that don't belong to weak+1. If we have one weak+1 archer on every target, this also means we also have one non-weak+1 archer on every target. Since under this scenario the weak+1 archers always stay where they are, this means the archers numbered 2 to N+1 will cyclically rotate around the N targets, periodically repeating their positions after every N rounds.

This means that if we replace R by $R' = 2N + (R \mod N)$ we would get an identical answer, since the outcome of the tournament after R rounds would be identical to the outcome after R' rounds (remember that $R \geq 2N$).

The above means that we can easily improve our $O(N^2R)$ algorithm to $O(N^3)$.

Optimising to $O(N^2)$

Currently, when we choose a starting position and we simulate what happens after R' rounds, we do $O(N^2)$ calculations per starting position. We can reduce the complexity of this part to $O(N)$ in the following way.

Observe that there are three types of archers: ones that are better than us, which we'll call the *black archers*; ones that are worse than us, which we'll call the *white archers*; and ourself (a single archer) which we'll denote as the *gray archer*. In order to solve our problem, we need not make any distinctions between archers of the same colour, as it is irrelevant to the final outcome. If two archers of the same colour compete against each other, it does not matter to us which one prevails (i.e., it is not going to impact the gray archer's final position). And we know that whenever two archers of different colours compete against each other, the archer of the darker colour wins.

Now there are three different cases which we'll handle separately.

Case 1 There are no black archers. This means we are the best archer and in this case it is trivial to show that the optimal target to start on is target N.

Case 2 There is at least one black archer, but no more than N. This means that our rank is between 2 and $N+1$, which means we are part of the group of archers that eventually ends up circling the targets periodically. In this case, it is notable that we do not need to simulate the full tournament, but only what happens on target 1. If we know who competes on target 1 every round, then just observing when between rounds $2N$ and $3N$ the gray archer gets to compete against archer number 1 will tell us where the gray archer will finish the tournament (which is all that we are interested in). We will track what happens on target number 1 using the following algorithm:

We assign each archer i a number P_i, which informally speaking indicates the earliest possible round where i might potentially compete on target 1. Initially each archer's P number equals his original target number. Then we simulate each round of the tournament with the following procedure:

1. We determine who is competing on target 1. The first archer there is clearly the winner on target 1 from the previous round (or initially the leftmost archer). We determine his opponent in the following way. We take all archers with P number less than or equal to the number of the current round. The best archer among them will now be competing on target 1 (the proof of why this is correct is further below).

2. We compare these two archers and assign the loser a P value equal to the number of the current (just passed) round plus N. This is the earliest round when we might potentially see this archer back on target 1.

Now let us prove that the above is correct. We will denote with A_j the archer who is competing on target 1 on round j, but who was competing on target 2 on round $j-1$. Every archer i has a value P_i that if he were to win every single contest since getting that P_i value, he would end up being A_{P_i}. Now let's look at the archer selected by our algorithm to be A_j (for some round j). We will denote him by W. Let $S = j - P_W$. If S is zero, this means that W didn't have the opportunity to become A_{j-1} even if he were to win all his contests. Hence, in this cycle W never met A_{j-1} (or any of the earlier A's). Since W never met these archers and since he is better than everybody else who is in the running for A_j, this means that he never lost in this cycle (until he got to target 1 at least), which means he truly is A_j (i.e., our algorithm is correct in this case).

If S is greater than zero, this means that W had the opportunity to become A_{j-1}, but lost it. This means he competed directly with A_{j-1} because the latter was the only candidate for A_{j-1} that was better than W[2]. Now if W competed

[2]This is true because by our algorithm every candidate for A_{j-1} automatically becomes a

with A_{j-1} and if he is better than every other A_j candidate, this means that after their meeting W was always "on the heels" of A_{j-1}: either on the same target, or on the one immediately to the right. This means that when A_{j-1} reached target 1 (which is in round $j-1$), W was on target 2. Since by definition he was better than the other archer on target 2, this means he was indeed the one to reach target 1 on round j.

Now that our algorithm for keeping track of target 1 is proved correct, we can analyze its time complexity. Since we make no distinction between same-coloured archers, we can represent any set of archers by just three numbers: the number of white, gray and black archers in that set. This allows us to execute every step of the algorithm (i.e., every round of simulation) in constant time, because all we have to do is determine the colour of the best archer in a set of candidates and then add to that set only one or two new archers (those whose P value equals the number of the next round). Since we only need to simulate up to round $3N$, and we are not interested in P values above $3N$, we can implement our simulation algorithm in $O(N)$ time and space.

Case 3 There are more than N black archers. This means our number is more than $N+1$, which means that we are one of the archers that eventually ends up standing on the same target indefinitely. We only need to determine which target that is.

We already showed that once archer 1 arrives on target 1, all that the weak+1 archers do is push each other around the targets until they settle on a different target each. Since our number is greater than $N+1$, this means that all white and gray archers belong to the weak set. Thus all we need to do is simulate how the white and gray archers push each other around. We start at target 1 where we know no white/gray archer would be allowed to stay. Then we attempt to count how many white/gray archers end up getting "pushed" around the circle after every target. Initially the white/gray archers pushed from 1 to N would be those that were initially at target 1 (note that our count is still a lower bound; later on we may find out there were even more white/gray archers pushed from target 1). Then we move to target N. We add any white/gray archers that start there to the ones we transferred from target 1 and we leave one of the combined set there (we always leave a white one, if we have one; if not, we leave the gray; if the set is empty, then we obviously do not leave anyone and let the black archers have this

candidate for A_j, except for the actual A_{j-1} — so if W was an A_{j-1} candidate, but did not succeed and is now the best among the A_j candidates, he must have been second to A_{j-1} among the A_{j-1} candidates.

spot). We keep going around the circle from N to $N-1$, to $N-2$, etc. On every target we "pick up" any white/gray archers and we leave one of those we have picked up either earlier or now. Eventually we get to target 1 and if we happen to have any white/gray archers pushed to target 1, we just transfer them to target N and keep going with the same procedure. The second time we return to target 1 we certainly will not have any more white/gray archers to push around, because by Theorem 1 we know that in $2N$ rounds every white or gray archer would have settled on a target. This algorithm clearly runs in linear time and space for the same reasons as the algorithm in Case 2 above. It is also correct because we only move around white/gray archers when necessary (i.e., when they would end up on the same target with another white/gray archer or on target 1) and we make sure that in the end every white/gray archer would have settled somewhere where he can remain undisturbed until the end of the tournament.

The optimization of the tournament simulation from $O(N^2)$ to $O(N)$ described above improves our solution to the whole problem from $O(N^3)$ to $O(N^2)$.

Optimising to $O(N \log N)$

The last optimization that we use to bring the complexity of our algorithm down to $O(N \log N)$ is based on the well-known technique of binary search. The efficient tournament simulation algorithms described above can easily be modified to also tell us how many times the gray archer moves from target 1 to target N (denoted by T). Combining this information with the final position of the gray archer (denoted X) allows us to view the final position on a linear (as opposed to circular) scale. If we describe the outcome of a simulation as being the number $X - N \times T$ we can think of every transfer of the gray archer from one target to another as decrementing the outcome by one. Then if we look at the simulation algorithms described above, we can observe that if the starting position is higher, then the final outcome can never be lower. For example if you choose to start with a larger P value this can never get you further ahead (on the linear scale, not the circular) than if you had chosen a smaller initial P value.

Given this monotonic relationship between the starting position and the final outcome of a tournament, can find the optimal starting position as follows:

1. Measure the outcome of starting on target 1.
2. Measure the outcome of starting on target N.
3. For each multiple of N in this range, use standard binary search to find the smallest possible ending position strictly greater than this multiple (and hence the closest to target 1 for a particular number of wrap-arounds).

4. Of the starting positions found above, pick the best.

Since there are only $O(N)$ rounds being considered, the range to search is $O(N)$ and hence only $O(1)$ binary searches are required. Each such binary search requires $O(\log N)$ starting positions to be tested, giving a time complexity of $O(N \log N)$.

Additional notes

Finally, we should note that the efficient simulation algorithms described above (which ignore distinctions between same-coloured archers and work in $O(N)$ time per simulation) can be implemented in a way that does distinguish between the different black or white archers, using binary heaps or other similar data structures. This would give a final algorithm of complexity $O(N^2 \log N)$ or $O(N \log N \log N)$, depending on whether one also uses the binary search. One can also achieve a time complexity of $O(N^2 \log N)$ or $O(RN \log N)$ by applying the binary search technique without optimizing the simulation.

It is also possible to solve the problem in linear time, but this is very difficult and left as an exercise to the reader. An $O(N \log N)$ solution is sufficient to receive a full score.

1.2 Hiring

Each worker k is described by two numbers: his minimum salary S_k and his qualification Q_k.

Imagine that we already picked a set K of workers we want to hire. How do we compute the total amount of money we need to pay them?

According to the problem statement, the salaries must be proportional to the qualification levels. Hence, there must be some unit salary u such that each employee $k \in K$ will be paid $u \cdot Q_k$ dollars. However, each employee's salary must be at least as large as his minimum salary. Therefore, u must be large enough to guarantee that for each $k \in K$ we have $u \cdot Q_k \geq S_k$.

For more clarity, we can rewrite the last condition as follows: For each $k \in K$ we must have $u \geq S_k/Q_k$. Let us label S_k/Q_k as U_k — the minimum unit cost at which worker k can be employed. We also want to pay as little as possible, hence we want to pick the smallest u that satisfies all the conditions. Therefore we get:

$$u = \max_{k \in K} U_k.$$

Note that this means that the unit salary is determined by a single employee in K — the one with the largest value of U_k.

We just showed that for any set of workers K (therefore also for the optimal set) the unit salary u is equal to the value U_k of one of the workers in K. This means that there are only $O(N)$ possible values of u.

Now imagine that we start constructing the optimal set of workers K by picking the unit salary u. Once we pick u, we know that we may hire only those workers k for which $U_k \leq u$. But how do we determine which of them to hire?

This is easy: if we hire a worker with qualification Q_k, we will have to pay him $u \cdot Q_k$ dollars. In order to maximize the number of workers we can afford (and minimize the cost at which we do so), we clearly want to hire the least-qualified workers.

Hence, we can compute the best solution for a given unit cost u by finding all the workers that we may hire, ordering them by qualification, and then greedily picking them one by one (starting from the least qualified) while we can still afford to pay them.

This gives us an $O(N^2 \log N)$ solution. The solution can easily be improved to $O(N^2)$, as we can sort the workers according to their qualification once in the beginning, and then each possible unit cost u can be tried in $O(N)$.

Finally, we'll show how to improve the above algorithm to $O(N \log N)$. We'll start by ordering all workers according to the value U_k in ascending order, and we label the workers $k_1, k_2, \ldots, k_N$ in this order.

In order to find the optimum set of workers, we'll do exactly the same as in the previous algorithm, only in a more efficient way.

Let $Z(m)$ be the following question: "What is the optimal subset of $\{k_1, \ldots, k_m\}$, given that the unit salary is $U_{k_m} = S_{k_m}/Q_{k_m}$?"

From the discussion above it follows that the optimal solution has to be the answer to a question $Z(m)$ for some m. Hence all we need to do is to answer these N questions.

The inefficient part of the previous solution lies in the fact that for each m we were doing the computation all over again. We can now note that we do not have to do this — we may compute the answer to $Z(m+1)$ from the answer to $Z(m)$, for each m.

Assume that we already know the optimal answer to $Z(m)$ for some m. We will store the workers we picked in a priority queue Q ordered according to their qualification, with more qualified workers having higher priority.

Now we want to add the worker k_{m+1}. His qualification level may make him a better candidate than some of the workers we have already processed. We add him into the priority queue Q. Q now contains all workers we need to consider when

looking for the current optimal solution, because if a worker had a qualification too large to be in the optimal solution for m, we will never want to use him again. This holds because the unit cost never decreases and the pool of workers only grows, so the cost of employing a worker together with all available less-qualified workers will only go up.

However, Q may still differ from the optimal answer to $Z(k+1)$, because the cost of paying all the workers in Q might exceed the budget W. There are two reasons for this: first, when adding the worker k_{m+1} the current unit salary u may have increased. And second, even if it stayed the same, we added another worker, and this alone could make the total salary of the chosen workers greater than W.

Hence, we now may need to adjust the set of chosen workers by repeatedly throwing away the most qualified one, until we can afford to pay them all. And this is where the priority queue comes in handy.

To summarize, the 100-point solution we just derived looks as follows: first, order the workers according to the unit salary they enforce. Then, process the workers in the order computed in step 1. Keep the currently optimal set of workers in a priority queue Q, and keep an additional variable T equal to the sum of qualifications of all workers in Q. For each worker k, we first add him into Q (and update T accordingly), and then we throw away the largest elements of Q while we cannot afford to pay them all — that is, while $T \cdot S_{k_m}/Q_{k_m}$ exceeds the amount of money we have.

Once we are finished, we know the numeric parameters of the optimal solution – the optimal number of workers, the minimum cost to hire that many workers, and the number f of the worker for which we found it. To actually construct the solution, it is easiest to start the process once again from the beginning, and stop after processing f workers.

The first step (sorting) can be done in $O(N \log N)$.

In the second step (finding the optimal number of workers and the cost of hiring them), for each worker we insert his qualification into Q once, and we remove it from Q at most once. Hence there are at most $2N$ operations with the priority queue, and each of those can be done in $O(\log N)$ (e.g., if the priority queue is implemented as a binary heap).

The third step (constructing one optimal set of workers) takes at most as long as the second step.

Therefore the total time complexity of this solution is $O(N \log N)$.

Alternative solution Instead of iterating m upwards, it is also possible to iterate it downwards. Suppose that P is the optimal subset of $\{k_1, \ldots, k_m\}$ with

$u = U_{k_m}$, and we wish to modify P to find the optimal subset of $\{k_1, \ldots, k_{m-1}\}$ with $u = U_{k_{m-1}}$. Firstly, we must remove k_m from Q if it is currently present. By potentially having reduced u and/or removed a worker, we may have freed up more money to hire workers. But which workers should we hire?

Clearly we cannot hire any workers that we are already employing. Also, the only reason we ever remove a worker k from P is because u fell below U_k, and since u only decreases that worker can never be hired again. Hence, we can maintain a simple queue of workers, ordered by qualification, and just hire the next available worker until there is not enough money to do so. It is also necessary to remove workers from this queue when u decreases, but this can be achieved by flagging workers as unemployable and skipping over them.

Each worker can be added to the optimal set at most once, and removed from the optimal set at most once. Each of these steps requires only constant time, so the core of this algorithm requires $O(N)$ time. However, the initial sorting still requires $O(N \log N)$ time.

1.3 POI

This problem is intended as a straight-forward implementation exercise. After the data is loaded from the file, a first pass over it can be used to count the number of people not solving each task (and hence the number of points assigned to each task). A second pass then suffices to determine, for each contestant, the number of tasks solved and the score.

It is not necessary to completely determine the final ranking: Philip's rank is simply the number of contestants that appear before him in the ranking, plus one. This can be determined by comparing each contestant to Philip. A contestant C will appear ahead of Philip in the ranking if and only if

- C has a higher score than Philip; or
- C has the same score as Philip, but has solved more tasks; or
- C has the same score as Philip and has solved the same number of tasks, but has a lower ID.

1.4 Raisins

At any moment during the cutting, we have a set of independent sub-problems — blocks of chocolate. If we find the optimal solution for each of the blocks,

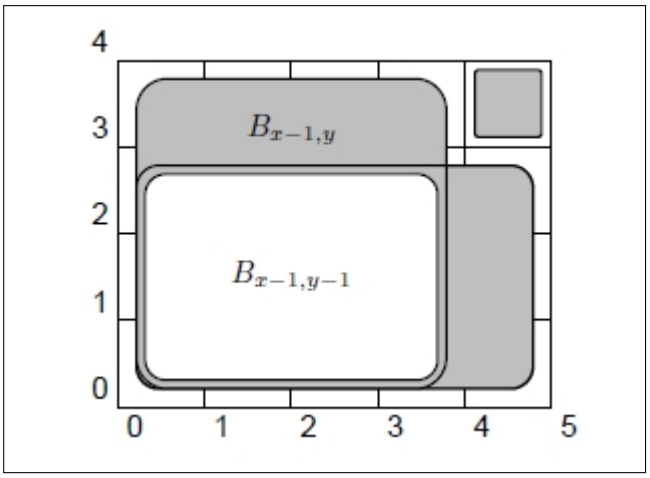
together we get the optimal solution for the whole chocolate. This clearly hints at a dynamic programming solution.

Each sub-problem we may encounter corresponds to a rectangular part of the chocolate, and it can be described by four coordinates: specifically, two x and two y coordinates — the coordinates of its upper left and lower right corner. Hence we have $O(N^4)$ sub-problems to solve.

Now to solve a given sub-problem, we have to try all possible cuts. There are $O(N)$ possible cuts to try — at most $N-1$ horizontal and $N-1$ vertical ones. Each possible cut gives us two new, smaller sub-problems we solve recursively. Obviously, the recursion stops as soon as we reach a 1×1 block.

Assume that someone has given us a function $S(x_1, y_1, x_2, y_2)$ that returns the number of raisins in the rectangle given by coordinates (x_1, y_1) and (x_2, y_2) in constant time.

Using this function we can solve the entire problem in $O(N^5)$. We will use recursion with memoization. Given any of the $O(N^4)$ sub-problems, first check the memoization table to see whether we have computed it already. If yes, simply return the previously computed value. Otherwise, proceed as follows: The cost of the first cut is $S(x_1, y_1, x_2, y_2)$, which we have supposed can be computed in $O(1)$ time. For each possible placement of the first cut, recursively determine the cost of the remaining cuts in each sub-problem, and pick the optimal choice, storing the answer in the memoization table.



We are only missing one piece of the puzzle: the function S. All possible values can easily be precomputed in $O(N^4)$ and stored in an array.

Alternatively, we can use two-dimensional prefix sums: let A be the input array, and let $B_{x,y} = \sum_{i<x} \sum_{j<y} A_{i,j}$. The values B are called two-dimensional prefix sums. They can be computed using the formula

$$\forall x, y > 0 : \; B_{x,y} = B_{x-1,y} + B_{x,y-1} - B_{x-1,y-1} + A_{x-1,y-1}.$$

Having the two-dimensional prefix sums, we can compute the sum in any rectangle, using a similar formula. The sum in the rectangle with corners (x_1, y_1) and (x_2, y_2) is

$$S(x_1, y_1, x_2, y_2) = B_{x_2,y_2} - B_{x_1,y_2} - B_{x_2,y_1} + B_{x_1,y_1}.$$

2.1 Garage

The problem is essentially a straight-forward simulation, but the data structures required are not completely trivial. A simple implementation that does not require any kind of data structure beyond a fixed-size array will keep track of:

- for each car, its state (absent, in the queue, or parked), its parking space (if parked), and its arrival time (if in the queue);
- for each parking space, whether there is a car parked there.

Now one can process the input events one at a time. When a car arrives, loop over all parking spaces to find the first empty one. If one is found, park the car there. Otherwise, the car will have to go into the queue — so record its arrival time.

When a car leaves the garage, it will be replaced by the car at the front of the queue (if any). Loop over all cars to find the car that arrived earliest and is still in the queue. If one is found, park it in the parking space that has just been freed up, and mark it as no longer in the queue.

This solution runs in $O(M^2 + MN)$ time. This can be improved: keeping the queue in a separate array reduces this to $O(MN)$, and also keeping the available parking spaces in a binary heap reduces it to $O(M \log N)$. However, these optimisations are not necessary to receive a full score.

2.2 Mecho

Solution 1

Firstly, working with fractional amounts of time is tricky, so we will measure time in units of $\frac{1}{S}$ seconds — let's call them *ticks*. Bees take S ticks to move from one cell to the next, while Mecho takes one tick.

Let us try to solve an easier problem first. Suppose we know when Mecho leaves the honey: can he get home safely? If we can solve this problem, then we can use it inside a binary search to find the last moment at which he can leave.

Mecho's moves will depend on the bees, but the bees' moves are fixed, so let us deal with the bees first. A standard breadth-first search will tell us when the bees reach each grassy cell (this just means simulating the spread of the bees over time).

Next, we can perform a similar breadth-first search for Mecho to answer the question "How soon (if at all) can Mecho reach each cell?" This can be implemented almost exactly as for the bees, except that one must exclude any steps that would have Mecho move to a cell where he would immediately be caught.

These breadth-first searches can each be implemented in $O(N^2)$ time. The problem statement guarantees that Mecho will eventually be caught if he stays with the honey, it takes $O(N^2)$ seconds for the bees to cover all the cells they can reach, and we are only interested in integer numbers of seconds in the binary search. Thus, the range of values explored by the binary search is $O(N^2)$ and hence the time complexity of this solution is $O(N^2 \log N)$.

Solution 2

Instead of using a binary search, we can use a more complicated method to directly determine the optimal time to leave any cell. The bees are processed as in the first solution. However, instead of working from the honey towards Mecho's home, we start from his home. Since he is safe in his home, there is no limit on how late he can arrive there.

Now suppose we know that for some cell Y, Mecho must leave no later than t ticks (from the time the alarm went off) and still make it home safely. If X is a neighbouring cell of Y, what is the latest time Mecho can leave cell X to safely make it home via Y? Clearly $t - 1$ is an upper bound, otherwise he will reach Y too late. However, he must also leave Y before the bees get there. The latest he can stay will be just the minimum of the two constraints.

One can now do a priority-first search: simulate backwards in time, keeping track of the latest time to leave each cell (keeping in mind that X has other neighbours, and it might be better to leave via those than via Y).

The time complexity of this solution depends on the priority queue used to order cells by leaving time. A binary heap gives an $O(N^2 \log N)$ implementation, and this is sufficient for a full score. However, it can be shown that the number of different priorities that are in the priority queue at any one time is $O(1)$, which makes an $O(N^2)$ solution possible.

2.3 Regions

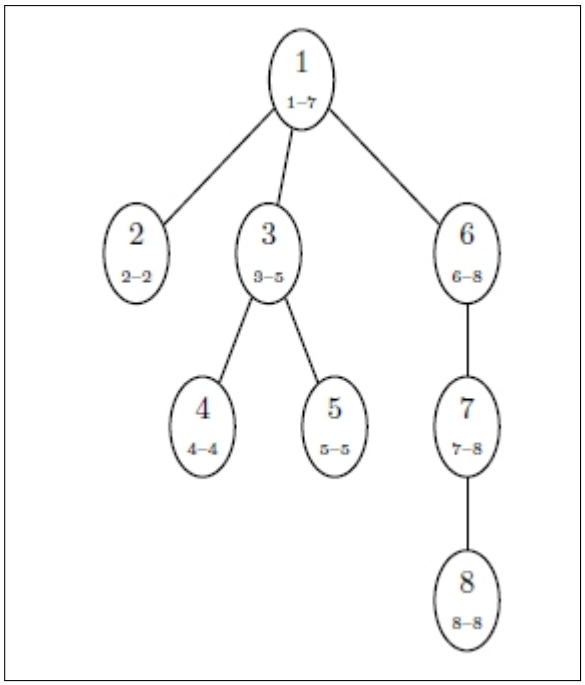
Although the employees are already assigned numbers in the input, the numbers can be reassigned in a way that makes them more useful. The supervisor relationships clearly organise the employees into a tree. Assign the new employee numbers in a pre-order walk of the tree[3]. Figure shows an example of such a numbering.

A useful property of this numbering is that all the employees in a sub-tree have sequential numbers. For a given employee e, let $[e]$ be the range of employee numbers managed by e. Notice that for a region, we can construct an ordered array of all the interval end-points for that region, and a list of all employees in that region. This can be done during the assignment of numbers in linear time.

Now let us consider how to answer queries (r_1, r_2). Let the sizes of the regions be S_1 and S_2 respectively. Given this data structure, a natural solution is to consider every pair of employees (e_1, e_2) from these regions and check whether e_2 lies in the interval $[e_1]$. However, this will take $O(S_1 S_2)$ time per query, which we can improve upon.

The interval end-points for region r_1 divide the integers into contiguous *blocks*. All employees in the same block have the same managers from r_1, and we can precompute the number of such managers for each such block. This gives us a faster way to answer queries. Rather than comparing every employee in r_2 with every block for r_1, we can observe that both are ordered by employee ID. Thus, one can maintain an index into each list, and in each step advance whichever index is lagging behind the other. Since each index traverses a list once, this takes $O(S_1 + S_2)$ time.

[3] A *pre-order walk* of a tree first processes the root of that tree, then recursively processes each sub-tree in turn.



An example of numbering employees by a pre-order walk. The bottom numbers indicate the range of employee numbers in each sub-tree.

Using just this query mechanism can still take $O(NQ)$ time, because all the queries might involve large regions. However, it is sufficient to earn the points for the tests where no region has more than 500 employees.

Precomputing queries

In the query algorithm above, it is also possible to replace the list of employees in r_2 with the entire list of employees, and thus compute the answer to all queries for a particular r_1. This still requires only a single pass over the blocks for r_1, so it takes $O(N)$ time to produce all the answers for a particular r_1. Similarly, one can iterate over all interval end-points while fixing r_2, giving all answers for a particular r_2.

This allows all possible queries to be pre-computed in $O(RN)$ time and $O(R^2)$ memory. This is sufficient to earn the points for the tests where $R \leq 500$.

This algorithm is too slow and uses too much memory to solve all the tests. However, it is not necessary to precompute all answers, just the most expensive ones. We will precompute the answers involving regions with size at least c. There are obviously at most N/c such regions, so this will take $O(N^2/c)$ time and $O(RN/c)$ memory. The remaining queries involve only small regions, so they can be answered in $O(Qc)$ time. Choosing $c = \sqrt{N}$ gives $O(N\sqrt{N} + Q\sqrt{N})$ time and $O(R\sqrt{N})$ memory, which is sufficient for a full score.

Caching queries

As an alternative to precomputation, one can cache the results of all queries, and take the answer from the cache if the same query is made again. Let Q' be the number of unique queries. The cost of maintaining the query cache depends on the data structure used; a balanced binary tree gives $O(Q \log N)$ overhead for this.

Combining the cache with the $O(S_1 + S_2)$ algorithm is sufficient to achieve the points for tests that have either no more than 500 employees per region (because this is the case even without the cache), as well as the cases with no more than 500 regions (since the total cost of all distinct queries together is $O(RN)$).

To achieve a full score with a cache rather than precomputation, one must use a better method for answering queries. Suppose we have a block in r_1, and wish to find all matching employees from r_2. While we have previously relied on a linear walk over the employees from r_2, we can instead use a binary search to find the start and end of the range in $O(\log S_2)$ time. This allows the entire query to be answered in $O(S_1 \log S_2)$ time. A similar transformation (binary searching the blocks for each employee in r_2) gives $O(S_2 \log S_1)$ time for each query.

Now when answering each query, choose the best out of the $O(S_1 \log S_2)$, $O(S_2 \log S_1)$ and $O(S_1 + S_2)$ query mechanisms. To establish an upper bound on run-time, we will make assumptions about which method is chosen to answer particular types of queries.

Again, divide the problem into large regions with at least c employees and the rest. For queries involving one of the large regions, use the $O(A \log B)$ algorithm (where A and B are respectively the smaller and larger of S_1 and S_2). The caching of queries ensures that this contributes no more than $O(N^2 \log N/c)$ time. For the remaining queries, use an $O(S_1 + S_2)$ algorithm. The smaller regions have at most c employees, so this contributes $O(Qc)$ time.

The optimal value of c occurs when the two parts account for equal time. Solving for this optimal c gives a bound of $O(N\sqrt{Q' \log N})$ for answering non-duplicate

queries; combined with the cost for the query cache, this gives an algorithm with time complexity $O(N\sqrt{Q' \log N} + Q \log N)$ and memory complexity $O(N + Q')$.

The time bound is marginally worse than for the first solution, but in practical terms this solution runs at about the same speed and uses significantly less memory.

2.4 Salesman

We'll start by considering only the case where no two fairs occur on the same day. Later we'll show how to modify our algorithm to incorporate fairs that occur on the same day.

The first polynomial solution

First we'll describe a fairly standard dynamic programming algorithm. We order the fairs according to the day when they take place. For each fair i we will compute the best profit P_i we can achieve immediately after visiting this fair.

To avoid special cases, we'll add dummy fairs 0 and $N + 1$ which both take place at the salesman's home, fair 0 being the first and fair $N + 1$ the last of all fairs. We can immediately tell that $P_0 = 0$ and that P_{N+1} is the answer we are supposed to compute.

The values P_1 to P_{N+1} can all be computed in order, using the same observation: we have to arrive from some fair, and we may pick which one it is.

Let $cost(x, y)$ be the cost of travelling from point x to point y on the river. If $x \leq y$, we have $cost(x, y) = (y - x)D$, otherwise we have $cost(x, y) = (x - y)U$.

We can then write:

$$\forall i \in \{1, \ldots, N + 1\} : \qquad P_x = \max_{0 \leq j < i} (P_j - cost(L_j, L_i)) + M_i$$

(Explanation: To compute P_i we pick the number j of the fair we visited immediately before fair i. Immediately after fair j the best profit we could have was P_j. We then have to travel to the location of the current fair, which costs us $cost(L_j, L_i)$, and finally we visit fair i for a profit M_i. To obtain the largest possible P_i we take the maximum over all possible choices of j.)

The time complexity of this algorithm is $O(N^2)$, which is sufficient to solve the cases where all the input values are at most 5,000.

An improved solution

We will now improve the previous algorithm. Note that the profit M_i from visiting fair i is the same for all choices of j. Thus, the optimal choice of j depends on the profits $P_0, \ldots, P_{i-1}$, the locations $L_0, \ldots, L_{i-1}$, and the location L_i of the current fair.

We can divide the fairs 0 to $i-1$ into two groups: those upstream of L_i, and those downstream. We can now divide our problem "find the optimal j" into two subparts: "find the optimal choice for the previous fair upstream" and "find the optimal choice for the previous fair downstream".

Consider locating the optimal previous fair upstream of L_i. If we were to change the value L_i (in such a way that it does not change which other fairs are upstream of fair i), can it influence our choice? No, it can not. If we, for example, increase L_i by Δ, this means that for *each* of the upstream fairs the cost of travelling to fair i increases by the same amount: $D\Delta$. Hence the optimal choice would remain the same.

We will now show a relatively simple data structure that will allow us to locate the optimal previous fair upstream of fair i in $O(\log N)$ time.

The data structure is commonly known as an interval tree. We can assign the fairs new labels according to their unique positions on the river. More precisely, let l_f be the number of fairs that are upstream of fair f (including those that occur after fair f).

Our interval tree is a complete binary tree with k levels, where k is the smallest integer such that $2^{k-1} \geq N + 2$. Note that $k = O(\log N)$.

Leaves in this binary tree correspond to the fairs, and the order in which fairs are assigned to leaves is given by the values l_i. That is, the leftmost leaf is the fair closest to the river source, the second leaf is the second-closest fair, and so on.

Now note that each node in our tree corresponds to an interval of fairs — hence the name "interval tree". In each node of the interval tree we will store the answer to the following question: "Let S be the set of fairs that correspond to leaves in this subtree and were already processed. Supposing that I'm downstream from each of them, which one is the optimal choice?"

Given this information, we can easily determine the optimal choice for the next fair i in $O(\log N)$. And it is also easy to update the information in the tree after fair i was processed; this too can be done in $O(\log N)$.

In our solution we will, of course, have two interval trees: one for the direction upstream and one for the direction downstream. For each fair i, we first make two queries to determine the best predecessor upstream and downstream, then we pick

the better of those two choices, compute P_i, and finally we update both interval trees.

Hence we process each fair in $O(\log N)$, leading to the total time complexity $O(N \log N)$.

Another equally good solution

In this section we will show another solution with the same complexity, which uses an "ordered set" data structure only, and can easily be implemented in C++ using the `set` class.

As before, we will process the fairs one by one, ordered by the day on which they occur. Imagine a situation after we have processed some fairs. Let a and b be two fairs that we have already processed. We say that a is *covered by* b if $P_a \leq P_b - cost(L_b, L_a)$.

In human words, a is covered by b if the strategy "visit fair b last and then move to the location of fair a" is at least as good as the strategy "visit fair a last".

Once a fair a is covered by some other fair b, this fair will never be an optimal predecessor for any later fair. Fair b will always (regardless of the location of the later fair) be at least as good a choice as a.

On the other hand, if a fair is currently not covered by any other fair, there are some locations on the river for which b would be the optimal predecessor — at least the location L_b and its immediate surroundings. We will call such fairs *active*.

In our solution we will maintain the set of currently active fairs, ordered by their position on the river. We will use an "ordered set" data structure, most commonly implemented as a balanced binary tree.

It can easily be shown that for each active fair f there is an interval of the river where f is the optimal choice. These intervals are obviously disjoint (except possibly for their endpoints), and together they cover the entire river. And as the interval for f contains f, the intervals are in the same order as their corresponding active fairs.

Hence whenever we are going to process a new fair i, we only have to locate the closest active fairs upstream and downstream of i — one of these two must be the optimal choice.

After we process the fair i and compute P_i, we have to update the set of active fairs. Clearly, i is now active, as we computed P_i by taking the best way of getting to L_i, and then added a positive profit M_i. We add it into the set of active fairs. But we are not done yet — i might now cover some of the previously active fairs.

But these are easy to find: if neither of the immediate neigbours of i (in the set of active fairs) is covered by i, we are obviously done. If some of them are covered by i, erase them from the set and repeat the check again.

In this solution, each fair is inserted into the set of active fairs once, and is erased from the set at most once. In addition, when processing each fair we make one query to find the closest two active fairs. Each of these operations takes $O(\log N)$, hence the total time complexity is $O(N \log N)$.

Multiple fairs on the same day

First of all, note that we cannot process fairs that are on the same day one by one — because we must allow the salesman to visit them in a different order than the one we picked.

There may be many ways in which to visit the fairs on a given day. However, we don't need to consider all of them, just some subset that surely contains the optimal solution.

Suppose that we already picked some order in which to visit the fairs on a given day. Let u and d be the fairs furthest upstream and downstream we visit. We can then, obviously, visit all fairs between u and v as well, as we'll surely be travelling through their locations. And clearly to visit all of these fairs, it's enough to travel first to u and then from u to v, or vice versa. We will only consider such paths.

We will process each day in two phases. In the first phase, we process each fair i separately, as if it were the only fair that day, and we determine a preliminary value P_i — the best profit we can have after coming to fair i from some fair on a previous day.

In the second phase we will take travelling upstream or downstream into account. We will consider each direction separately. When processing a direction, we'll process the fairs in order, and for each of them we'll determine whether it is more profitable to start at this fair (i.e., use the value computed in the previous step) or to start sooner (i.e., use the optimal value computed for the previous fair in this step, subtract the cost of travel from that fair to this one, and add the profit from this fair).

For each fair i, the actual value P_i is then equal to the larger of the two values we get for travelling upstream and downstream.

Finally, we need to update the set of active fairs. When using an interval tree data structure as in Section , this is accomplished simply by adding each fair to the upstream and downstream interval trees. When using an ordered set as in Section , one must take a little more care, as not all of the fairs that we have just processed will be active. This is easily catered for by modifying our update

process — before inserting a new active fair, we check that the fair is actually active by examining its potential neighbours in the data structure. If either of the neighbouring fairs covers the one being added, then it is not active, and so should not be added to the active fairs set. With this modification, we can update the entire data structure by sequentially attempting to add each fair (in any order).

Clearly, the additional time needed to process the second phase on any day is linear in the number of fairs that day, assuming we already have them sorted according to their location (which is easily accomplished by adding this as a tie-breaker to the comparison function used to sort all fairs in the beginning). Furthermore, the update steps for the interval tree and ordered set both take $O(\log N)$ time. Therefore this extra step does not change the total time complexity of our algorithm: it is still $O(N \log N)$.

Problem Proposers:

Day 0

0.2 Hill - Iskren Chernev
0.3 Museum - Boyko Bantchev

Day 1

1.1 Archery - Velin Tzanov
1.2 Hiring - Velin Tzanov
1.3 POI - Carl Hultquist
1.4 Raisins - Emil Kelevedjiev

Day 2

2.1 Garage - Carl Hultquist
2.2 Mecho - Carl Hultquist
2.3 Regions - Long Fan and Richard Peng
2.4 Salesman - Velin Tzanov

Reserve Problems

Bruce Merry, Mihai Patrascu, Kentaro Imajo