Problem A. Fence Painting

Problem author and developer: Margarita Sablina

Subtask 1

To solve the first subtask, it was sufficient to iterate over the meeting place, and then by passing through all the poles from 1 to $a + b$, find out how much money Tom and Huck will ultimately receive. For each pole i, one could quickly understand how many coins are given for painting it: x, if $i \le a$, and y otherwise. From all possible meeting places, it remained to choose one where the absolute difference between the amounts paid out is minimal.

Subtasks 2 and 3

In the second and third subtasks, partial solutions were considered. In the case of $x = y$, all poles are worth the same. In this case, it was required for Tom and Huck to paint approximately the same number of poles (either the same or differing by 1). As an answer, it was enough to output $\left\lfloor \frac{a+b}{2} \right\rfloor$.

In the case of $a = b$, it was possible to definitively determine who among the boys would have to paint more poles: for example, if $x < y$, then Tom would have to paint all the wooden poles, and possibly some metal ones as well. In this case, the number of metal poles that he would have to paint could be calculated based on how much ax is less than by: for each metal pole given to Tom instead of Huck, the former receives y more money, and the latter receives y less. Therefore, the number of metal poles that should be given to Tom should be either $\left\lfloor \frac{by-ax}{2y} \right\rfloor$ or $\left\lceil \frac{by-ax}{2y} \right\rceil$.

Subtask 4

As in the first subtask, it was enough to iterate over the meeting place, but here the constraints no longer allowed calculating the number of coins each boy received in linear time. Instead, one could use the formula: if $i \le a$, then Tom would have received for the first i poles $g_t = i \cdot x$, otherwise—$a \cdot x + (i - a) \cdot y$. The number of coins received by Huck can be calculated as $g_h = ax + by - g_t$.

Subtask 5

For a complete solution, it remained to notice that the function $g_t(i) - g_h(i)$ is monotonically increasing, or that $|g_t(i) - g_h(i)|$ first decreases monotonically, then increases. Thus, one could use binary search to find the values of $g_t(i) - g_h(i)$ closest to zero, or ternary search to find the absolute minimum of $|g_t(i) - g_h(i)|$.

Alternatively, one could use the formula. We want each of the boys to receive a value as close as possible to half, that is, $\frac{ax+by}{2}$. If $ax \ge by$, then Tom should have painted no more than a poles, namely—either $\left\lfloor \frac{ax+by}{2x} \right\rfloor$ or $\left\lceil \frac{ax+by}{2x} \right\rceil$. The symmetric case is similar, but in it, it is more convenient to count the number of poles painted by Huck. In any case, from the two possible options, one must explicitly choose the most optimal one (or use `round` instead of rounding to a specific side).

Problem B. Work, Sleep, Repeat

Problem author: Maria Zhogova, developer: Daniil Oreshnikov

An important fact for solving the problem is that due to the periodicity of the cycle with a period of $x + y$ days, each d_i can be considered as $d_i \bmod (x + y)$. It remains only to check the existence of such a day D, on which all d_i fall within the first x days of the cycle after it, that is, from D to $D + x - 1$.

Subtask 1

For $x = 1$, as one can notice, using the observation described above, it is necessary that all days d_i fall on the same remainder modulo $x + y$, that is, $y + 1$. Thus, it is sufficient to calculate $d_i \bmod (y + 1)$ for each d_i and check that all the resulting numbers are the same. If so, then this resulting number will be the sought answer; otherwise, there is no answer.

Subtask 2

In this subtask, the constraints are small enough to iterate over all possible D from 1 to $x + y$, and for each of them, check whether this value is suitable. Using the key observation, it is necessary that for each d_i, some number of whole cycles (possibly 0, or in the new condition, -1) of $x + y$ has passed since day D, and no more than x days after. That is, for each d_i it must hold that $d_i = D + k(x + y) + w$, where $w < x$.

To do this, it is enough to check that for all d_i it is true that $(d_i - D) \bmod (x + y) < x$. Thus, in the second subtask, by iterating over all possible D and checking each to see if it fits, the solution works in $\mathcal{O}(n(x + y))$, which was enough to pass all the tests.

Subtask 3

With the constraint $y = 1$, we encounter a situation completely opposite to the first subtask — in the first subtask, each d_i imposed a restriction for D to have the same remainder modulo $x + y$, but here, on the contrary, each d_i "prohibits" exactly one possible remainder. For example, with $x = 5$, the value $d_i = 12$ prohibits $D = 1$. Otherwise, the workdays would be 1–5 and 7–11, and day 12 would fall on a vacation.

Thus, it is enough to create a set of all "prohibited" values of D, and for each d_i add to this set $(d_i - x) \bmod (x + y)$. If the set of all prohibited values is smaller than $x + y$, then there is guaranteed to be a number D that fits all the conditions. It can be found in $\mathcal{O}(n)$, as there will be no more than n prohibitions, meaning some number from 1 to $n + 1$ will be missing in the set of prohibited numbers.

Subtasks 4 and 5

For a complete solution, it was enough to generalize the previous ideas. Namely, that each d_i imposes a restriction of the form "D must be within a certain segment of remainders modulo $x + y$". Specifically, the corresponding cycle cannot start earlier than day $d_i - x + 1$ and cannot start later than d_i, that is, $D \in [(d_i - x + 1) \bmod (x + y), d_i \bmod (x + y)]$.

Therefore, the complete solution is to list all such restrictions and find their intersection. The only problem is that the segment of remainders may have the form $[7, 3]$, that is, "either the remainder is at least 7, or the remainder is no less than 3". In subtask 4, it is guaranteed that $x < y$, and then it can be proven that the intersection of any two segments of length x will also yield one segment of length no more than the original ones, and the intersection of segments can be performed in $\mathcal{O}(n)$.

For subtask 5, one could use any convenient method of intersecting cyclic segments — for example, sort all $d_i \bmod (x + y)$ and with two pointers check that there exists such a D that all d_i lie from it to $D + x - 1$, or represent each restriction segment with two events of the type "restriction started" and "restriction ended", sort the events, and find the remainder at which the counter of active restrictions equals n. Thus, the running time of the complete solution is $\mathcal{O}(n \log n)$.

Problem C. Palindromization

Problem author and developer: Vladislav Vlasov

Let's start with some transformations. They are not necessary for solving all subtasks, but after applying

them, it will be easier to think about what is required in the problem and why, for example, the running time of a brute-force approach will be sufficient to fit within the constraints.

So, let's consider our array a.

1. Notice that there is no point in performing additions on segments that cross the middle of the array. Indeed, if m is the middle of the array, then adding x to the segment $[m-p, m+q]$, where $p < q$, is equivalent from the task's goal perspective to adding x to the segment $[m+p+1, m+q]$. Indeed, the difference between these two actions is only in adding x to $[m-p, m+p]$, and this action does not change the property of a being a palindrome.

2. Now let's reduce the problem to another one: consider the array b of the form

$$b = [a_1 - a_n, a_2 - a_{n-1}, \ldots, a_{\lfloor \frac{n}{2} \rfloor} - a_{\lceil \frac{n+1}{2} \rceil + 1}],$$

   that is, the array of differences of opposite elements of the array a. If a is a palindrome, then all elements of b are zeros. And adding x to a segment of the array a is equivalent to adding or subtracting x to a segment of the array b.

3. Now let's calculate c—the difference array of the array b, to which we have added 0 at the beginning and the end, i.e.,

$$c = [b_1, b_2 - b_1, b_3 - b_2, \ldots, b_{\lfloor \frac{n}{2} \rfloor} - b_{\lfloor \frac{n}{2} \rfloor - 1}, 0 - b_{\lfloor \frac{n}{2} \rfloor}].$$

   In such an array, almost nothing changes in terms of our goal—when all elements of b are zero, all elements of c are zero, and vice versa. However, now we have reduced adding x to a segment of b to adding x to one element of c and subtracting x from another.

Subtask 1

In the first subtask, it was possible to write a neat brute-force search of possible actions. We will analyze the array b instead of the array a. In it, numbers range from -9 to 9, and for each action, it is enough to choose two numbers of the same sign and add or subtract a certain value on the segment between them.

Notice that the order of actions does not matter, which means you can go through the array b from the beginning to the end and for each next b_i choose the end of the segment for it, the element on which is of the same sign, and make an addition to the segment. The array b has a size of no more than 5, so there are no more than $\frac{5 \cdot 4}{2} = 10$ different segments for addition (actually even less if the ends of the segments are chosen to be of the same sign).

Moreover, it is evident that heuristically it is only beneficial to make additions of smaller x when the original b has many numbers $< k$, and in such a case the answer will be smaller. Thus, we have less than 10 possible segments for addition, on each there are several ways to choose the added value, and in the end, a brute-force search of actions on the tests of the first group will fit within the constraints. The only necessary observation for this is that there is no need to consider in a segments that cross the middle of the array.

Subtask 2

In this subtask, you need to add no more than 1 to each element. Again, we will consider the array b, and note that exactly b_i operations need to be applied to the i-th element. It can be proven that in this case, the answer will be the number of continuous segments of ones and minus ones in the array b. Indeed, it is not possible to reduce the number of such segments by at least 2 for each action, so the answer cannot be less.

Subtask 3

Notice the following fact: if $a_i \le a_{i+1}$ for all i, then all numbers in the array b will be negative, and moreover, non-decreasing. In this case, the number of operations applied to b_i will be no less than the number of operations applied to b_{i+1}. Accordingly, apply the following algorithm:

1. We will add k to all numbers of the array b until the next addition makes its last element positive. Then instead, we will add minus the last element of b to all its elements.
2. We have achieved that $b_{\text{last}} = 0$ in the minimum number of actions. Now we will repeat the same actions, but already for the segment from the first element of b to the penultimate.
3. When the last two elements of b are zero, we will continue to narrow the prefix on which we perform additions, gradually coming to the fact that the entire array b consists of zeros.

The fact that we used the minimum number of actions can be proven constructively. For this, it is enough to show that any method that includes adding to two non-overlapping segments can be transformed into a no worse method in which these segments are not present. This is solved by a small case analysis, which we will not present here because it is not a key idea necessary for the solution.

Subtask 4

Starting with this subtask, we will use the array c for the solution. Once we have calculated the array c, our task becomes to achieve a completely zero array in the minimum number of actions of the type "add x to one element and subtract from another". In the subtask with $k = 1$, this is achieved quite straightforwardly: for each operation, you need to add 1 to a negative element and subtract 1 from a positive one.

Since the sum of the elements of the array c is zero, the necessary number of actions can simply be calculated as the sum of all its positive elements. The running time of the solution is $\mathcal{O}(n)$.

Subtask 5

Similarly to the previous subtask, we will construct the array c, but process it a little differently. Now we can add and subtract 1 or 2. The number c_i requires $\left\lceil \frac{|c_i|}{2} \right\rceil$ operations, it is impossible to do less. Let's calculate two sums of such quantities: for positive values and for negative ones. The answer will be the larger of them.

For this, first note that there is no point in subtracting from negative numbers and adding to positive ones—then it is always possible to either reduce the answer or get rid of such an operation without increasing it. And then note that it is not possible to achieve a smaller number of operations, and it is possible to get all zeros for such a number of operations. In the part where the maximum sum of $\left\lceil \frac{|c_i|}{2} \right\rceil$ was obtained, each c_i will be processed as $\left\lfloor \frac{|c_i|}{2} \right\rfloor$ operations of adding or subtracting two, and possibly one more operation of adding or subtracting one. In the other part, the sum turned out to be less, which means there were fewer odd numbers in it. Then each odd number will still have enough for one, and the "extra" ones will be grouped in pairs to add to those even numbers that lacked twos.

Subtask 6

For the last subtask, it was necessary to make a few more constructive observations. First, let's reformulate the problem as "given a set of positive c_i and a set of negative c_i, it is required to find such a set of addends from 1 to 3 that they can be used to decompose all numbers of both the first and second sets". Indeed, each operation is an addition to a negative and a subtraction from a positive. In essence, we are decomposing the sets of positive and negative numbers among c_i into addends.

Now note that:

- Any decomposition into addends from 1 to 3 of a number $c_i \geq 8$ always contains several addends that give a sum of exactly 6 (proven by a careful case analysis). From this, it follows that instead of decomposing $c_i \geq 8$, you can independently decompose $c_i - 6$ and 6.
- In the decomposition of numbers $c_i = 5$ or $c_i = 7$, there will always be several addends with a sum of 3. This means you can decompose independently $c_i - 3$ and 3.

Then we will divide c into a group of negative and a group of positive numbers and transform them as described. In each group, there will only be numbers from the set $\{1, 2, 3, 4, 6\}$. Now we will present a solution that allows you to find the answer in time $\mathcal{O}(n^2)$. To do this, note that the number of numbers in each such set, not equal to 6, is linear from n (from each original c_i no more than two numbers from 1 to 4 were obtained). Moreover, it cannot be that in both sets there is a decomposition $6 = 2 + 2 + 2$, because then it can be replaced with $6 = 3 + 3$ with a smaller number of addends. The same is true for ones.

Then at least in one set, the number of ones does not exceed $\texttt{cnt}[1] + 2\texttt{cnt}[2] + 3\texttt{cnt}[3] + 4\texttt{cnt}[4] + 6$, and the number of twos—$\texttt{cnt}[2] + \texttt{cnt}[3] + 2\texttt{cnt}[4] + 3$. In fact, it will be seen that the necessary number of ones and twos is even less, but for now, we are satisfied with the fact that they are linear from n.

Then let's iterate through the number of used ones (m_1) and twos (m_2), find the number of threes as $\frac{\texttt{total} - m_1 - 2m_2}{3}$, and check whether both sets can be broken down into such addends. To check whether a set can be broken down into such addends, it is enough to check the following conditions:

1. If $m_1 < \texttt{cnt}[1]$, then it is not possible, because ones cannot be obtained in any other way. Otherwise, we will reduce m_1 by $\texttt{cnt}[1]$ and forget about them for now.
2. If $m_2 \leq \texttt{cnt}[2]$, then the remaining twos and all fours should have enough ones, everything else will decompose into addends equal to 3.
3. If $m_2 \leq \texttt{cnt}[2] + 2\texttt{cnt}[4]$, then the remaining fours should have enough ones. Moreover, if $m_2 - \texttt{cnt}[2]$ is odd, then the remaining two will clearly require an extra one. Indeed, either in the decomposition of some number, it will be necessary to put $2 + 1$, or from the current distribution of twos, one decomposition will have to be removed, and in it, use ones instead of twos. A small case analysis of how the addends 2 can be distributed among the "targets" shows that one extra one is enough for the remaining numbers to fall apart into addends of 3, and without it, it will not be possible to decompose.
4. If $m_2 \leq \texttt{cnt}[2] + 2\texttt{cnt}[4] + 3\texttt{cnt}[6]$, then we will break down everything as $2 = 2$, $4 = 2 + 2$ and $6 = 2 + 2 + 2$, and there will be either one or two extra twos left. Another small case analysis shows that for each of the remaining extra twos, an extra one will be needed regardless of whether to distribute the twos in the shown way or not.
5. If $m_2 > \texttt{cnt}[2] + 2\texttt{cnt}[4] + 3\texttt{cnt}[6]$, then similarly, for each extra two, one will be needed to combine them into $3 = 1 + 2$.
6. If $m_2 > \texttt{cnt}[2] + 2\texttt{cnt}[4] + 3\texttt{cnt}[6] + \texttt{cnt}[3]$, then there will not be enough "places" where twos can be distributed.

Thus, for each pair (m_1, m_2), it is possible to determine whether they give a correct decomposition, and choose the one that minimizes the total number of addends. There is also an alternative solution that reduces this problem to a knapsack problem, but we will not present it here.

Problem D. Hackathon

Problem author: Egor Yulin, solution author and developer: Daniil Oreshnikov

Subtask 1

The first subtask was designed for a careful implementation of the process described in the statement. It was enough to move the participants on the field second by second, add new ones at the moment of their arrival, and recalculate their goals. To do this, we will maintain the current position of each participant and his current "goal" — the position of the minimum on his prefix in the array a.

After each "tick" of time (once a second), it is enough to recalculate the positions of the participants and group them by positions into sets. It can be done more efficiently by noticing that two participants, once on the same position, will continue to move together until one of them takes the desired piece of pizza, and maintain these very sets, rather than recalculating each time.

For each set of participants who have reached their goal, it is enough to choose the participant with the minimum number and give him the corresponding piece, and then recalculate the goals. The maximum time in this group will not exceed $t_i + v \cdot s_i \leq 20\,000$, and at each moment it is necessary to update the positions of all participants ($\mathcal{O}(m)$) plus recalculate the goals (with a linear pass through the array for $\mathcal{O}(n+m)$ or even for $\mathcal{O}(nm)$). It is worth noting that the goals are recalculated only when the array a changes, which will happen no more than m times. Thus, even the most inefficient implementation of this solution works for $\mathcal{O}(T \cdot m + m^2 n)$, which is more than enough for the constraints of this subtask.

Subtask 2

When all participants start at the same time from the same position, they should be perceived as a single whole — they will arrive at each goal together. And each time, the participant with the minimum number will get a piece of pizza and leave for the place.

We will maintain a heap or search tree of pairs $(a_i, -i)$ to find the minimum a_i, and in case of equal values — the rightmost of them. In the heap, we will store all elements at positions up to and including the starting position of all participants, and before moving, check whether the current position is the position with the thickest piece of pizza. When issuing some a_i to the next participant, it is enough to put back into the heap the element $(a_i + 1, -i)$.

There will be no more than $n + m$ operations with the heap (each element plus the number of times we gave a piece of pizza to a participant), so the running time of such a solution is $\mathcal{O}((n+m)\log n)$.

Subtask 3

This is another subtask that allows you to score points by writing a partial solution, not optimized to the full. The condition $|a_i - a_j| \geq n$ means that none of the participants will ever change their goal. Indeed, changing the goal means that the participant was going to a certain value $a_j = x$, it was taken, and now his new minimum is at least $a_i = x + 1$. But if some a_i increased from one of the initial values to another a_j, then it increased by 1 exactly $a_j - a_i$ times, which in this subtask is not less than m. Since increases occur only after the next participant takes a piece of pizza, it cannot be that some of the participants have not yet received their piece, and the value has already increased by m.

In this case, it is enough for each participant at the moment of his appearance near the row to find the minimum a_i on the prefix — this position will be the answer for him. This can be done in several ways, in particular — with the help of a segment tree with the operation of changing an element and finding the minimum on the prefix. In this case, participants should be initially sorted by the time they approach the row, determine the time for each when he will approach the final piece of pizza, and then process the changes of a_i in the order of increasing this time to determine the final answer for each participant. The running time of the solution is $\mathcal{O}((n+m)\log n)$.

Subtasks 4 and 5

Another possible approach to solving the problem was sorting events. Note that naive simulation is not efficient for several reasons:

1. we spend $\mathcal{O}(m)$ time processing each second, even those in which nothing happens;
2. we take a long time to find new goals for participants, even without separating those for whom they need to be recalculated and those for whom they do not.

These actions could have been implemented more efficiently. Let's create a heap that stores events like "participant came to his (possibly outdated) goal" and "participant approached the row of pizzas". We will order events in the heap by time, and if the time mark is the same, we will first process those who approached, and then those who reached the goal.

At the moment of approaching the row, we calculate the current goal of the participant — it will be the minimum on the prefix in the current state of the array a. We calculate the moment when he will approach this goal, and add the corresponding event to the heap. When processing the event "participant approached his goal", we recalculate events for all those who went to the same position. This can be done either immediately or deferred — recalculate the goal when the participant approaches his old goal and sees that the value known to him before is not relevant. These two approaches give the same result.

With an efficient implementation of finding the minimum on the prefix (for example, through a segment tree), such a solution passes both groups. With a less efficient one, but faster than $\mathcal{O}(nm^2)$ in total — passes the fourth. Alternatively, the fourth subgroup could have been solved by a variation of the full solution with worse asymptotics.

Subtask 6

The idea of processing "goals", that is, pieces of pizza, using a heap in ascending order, which appears in the subtask, is the key idea for the full solution. Indeed, it will be difficult to find the final point for the participant by position and start time of movement. It is easier for each position to find the participant who will stop there.

We will process pairs $(a_i, -i)$ in ascending order. For the next such pair, we will find the participant who will occupy it. For this participant j, three conditions must be met:

1. $s_j \geq i$, that is, the participant approached the row not to the left of this position;
2. by the time the participant approaches position i, that is, $t_j + v \cdot (s_j - i)$, a_i is the minimum on the corresponding prefix;
3. of all such participants, the one who approached this position earlier than everyone else is chosen, or, if there are several such, the one with the minimum number among them.

Let's transform the condition on time: the fact that at the time of approaching i it is the minimum on the prefix is equivalent to the fact that the time of approach is not less than any time of taking a piece of pizza located to the left (since we process positions in ascending order of a_i values). Let's call the moment when a_i becomes the minimum on the prefix $\texttt{stable}(a_i)$. Then we want $t_j + v \cdot (s_j - i) \geq \texttt{stable}(a_i)$ or $t_j + v \cdot s_j \geq \texttt{stable}(a_i) + v \cdot i$. On the left is a function of j, on the right — of i and the current state of a_i.

Now let's reformulate the task as "find a participant with $s_j \geq i$ and the minimum $t_j + v \cdot s_j$, not less than $\texttt{stable}(a_i) + v \cdot i$". This can be done in several ways, one of which is to sort the participants by s_j, calculate the desired value $\texttt{target}_j = t_j + v \cdot s_j$ for each, and divide into blocks of size $\sqrt{m \log m}$, after which sort the participants in each block by `target`. Then, to find a participant whose `target` is minimal among all not less than $\texttt{stable}_i + v \cdot i$, it is enough to do a binary search in several blocks, and part

of the block can possibly be processed separately, looking at each participant. Such a request works for $\mathcal{O}(\sqrt{m \log m})$.

It remains only to learn how to mark a participant who has received a piece of pizza as having left the row. To do this, you can either rebuild his block from scratch, or use the path compression technique — remember for each person the next in magnitude `target` in his block, which has not yet been deleted, and when finding a participant, go through these links, compressing them into shorter paths in parallel.

Separately, it should be noted that if there are no participants taking a piece for themselves at position i, then this position will never become a goal for anyone, so it does not need to be added back to the heap. The full solution, thus, works for $\mathcal{O}((n+m)\sqrt{m \log m})$. There is also a more efficient asymptotic solution with a segment tree on sets to search for a participant with the same criteria, but it has a larger constant, and the physical running time is longer.

Problem E. Tree Trisection

Problem author and developer: Daniil Oreshnikov

Let's introduce some notations first. Denote the minimal connected set containing all leaves from l to r with the root at v as $\texttt{cover}(v, l, r)$. Also, denote the minimal leaf in the subtree of vertex v as $\text{left}(v)$, and the maximal one as $\text{right}(v)$.

Now let's reformulate the problem statement a bit simpler and note several important facts. A complete binary tree is given, and there are queries specifying a segment of its leaves $[l, r]$. Initially, a set $V = \texttt{cover}(\text{lca}(l, r), l, r)$ is selected. Then it is required to choose two vertices x and y in this set so that if you "cut off" the set V itself and the subtrees of x and y, the maximum of the weights of the three resulting subtrees will be minimal.

Accordingly, two useful facts:

1. The vertex $\texttt{root}(\texttt{V})$ can be found as $\text{lca}(l, r)$, where in the numbering from 1 the left child of vertex i has number $2i$, and the right one has $2i + 1$, so $\text{lca}(l, r)$ has a number whose binary representation is the longest common prefix of the binary representations of l and r.
2. To minimize the maximum of the weights of the three trees, it is enough to bring their weights as close as possible to one third of the weight of V. Indeed, the sum of their weights is fixed and equal to the weight of V, and therefore the maximum of them will not be less than $\frac{\texttt{weight}(V)}{3}$.

Subtask 1

As usual, the first subtask is designed for a fairly straightforward implementation of what is asked in the problem statement, and we don't even need the facts considered above. Finding $\texttt{root}(V)$ can be done in time $\mathcal{O}(\log n)$ by simply considering all ancestors of l and r. And then it is enough to explicitly list all the vertices that are included in V: these are the vertices whose segment of leaves intersects with $[l, r]$, and iterate over all the options for choosing x and y.

Even if for each query we iterate over all possible pairs of x and y, and for both vertices calculate the weight of their subtree between the leaves l and r in time $\mathcal{O}(n)$, we get a solution in time $\mathcal{O}(mn^3)$, which is acceptable for the constraints of this subtask. It was also required to carefully check for each pair (x, y) being iterated that they are not an ancestor and a descendant — this can be done in $\mathcal{O}(\log n)$ by iterating over the ancestors of each of these two vertices.

Subtask 2

A slightly more careful calculation of the weights of the subtrees x and y in time $\mathcal{O}(\log n)$, for example, with precomputation of the weight of each complete subtree and descending with the addition of the necessary weights from x and y downwards, allowed to solve subtask 2, in which the time complexity

of $\mathcal{O}(mn^2 \log n)$ was sufficient. More specifically — to calculate the weight of some subtree in V, it was enough to descend from the root vertex of this subtree downwards, and for each vertex

- if the segment of its leaves does not intersect with $[l, r]$, return zero;
- if the segment of its leaves is contained within $[l, r]$, return the entire weight of its subtree;
- otherwise — descend to its children and return the sums of weights obtained from them.

Then, on each layer of the tree, no more than four vertices will be visited, and a descent to the next layer below will be made from no more than two of them, so the running time of such a weight calculation is proportional to the height of the tree.

It was also possible for each leaf a and its ancestor p to calculate the total weight of $\texttt{cover}(p, a, \text{right}(2p))$ or $\texttt{cover}(p, \text{left}(2p+1), a)$, i.e., the subtree containing either all the leaves between a and the "middle" of the subtree p. Then for $v = \text{lca}(l, r)$, the weight of $\texttt{cover}(v, l, r)$ decomposes into the weight of $\texttt{cover}(v, l, \text{right}(2v))$ and the weight of $\texttt{cover}(v, \text{left}(2v+1), r)$, which can be calculated in $\mathcal{O}(1)$ using precomputed values.

Subtask 3

In the case where two adjacent leaves are considered, V will always be split into two vertical paths from $\texttt{lca}(l, r)$ to l and to r. This means that the total size of the set V does not exceed $2 \log_2 n$, and in it, you can iterate over x and y in time $\mathcal{O}(\log^2 n)$, or use the two-pointer method to minimize the maximum of the weights of the three subtrees in time $\mathcal{O}(\log n)$.

The sizes of the subtrees x and y can be calculated on the fly during the iteration, or you can use the precomputation described above.

Subtask 4

In the subtask where the weights of all vertices are equal to 1, there is no need to think about complex calculations of the weights of the subtrees, but it is simply possible to notice that any subtree in V is either complete and has a weight of $2^k - 1$ for some integer k, or has its root on the path either from V to l or from V to r. Therefore, there are no more than $3 \log_2 n$ different weights of subtrees in V.

Then it is enough to carefully determine for which k in V there is a complete subtree of weight $2^k - 1$ (for this, it is necessary to quickly find the highest vertex with a complete subtree in V by a simple descent from $\text{lca}(l, r)$ to the left and to the right), and for each query look at $\mathcal{O}(\log n)$ possible sizes of subtrees that can be cut from V, which will give a running time of the solution $\mathcal{O}(m \log^2 n)$ or $\mathcal{O}(m \log n)$ if you also use two pointers.

It is only necessary to carefully process the fact that it is forbidden to choose x and y that are an ancestor and a descendant, so it is worth separately remembering all possible k from the left and right subtrees of v, and iterate over the options for x from the left subtree, and for y — from the right. Separately, it is necessary to take into account the options in which x and y are located on the same side — either by carefully considering the options for the structure of the trees, or by using the trick described at the end of the solution of the following subtasks.

Subtasks 5 and 6

For a complete solution or close to it, it was required to calculate various auxiliary information or the answers themselves faster. Since it is forbidden to choose x and y that are an ancestor and a descendant, we want to independently cut out two subtrees from V so that their weights and the weight of what remains are as close to each other as possible. As we have already noted, for this it is enough to choose

the subtrees x and y with weights closest to one third of the weight of V, although this statement requires clarification (see below).

But for now, let's consider two cases — either x and y lie on the same side of $v = \texttt{root}(V)$, or on different sides. If they are on different sides, then without loss of generality, x is in the left subtree, and y is in the right. We will then immediately respond to all queries for which $a = \texttt{lca}(l, r)$ is equal to the current vertex. To do this, we independently find x closest to one third of the weight of V in the left subtree of v, and y in the right.

For this, we remember for each query its l, and we will move along the leaves from the "central left" ($c_1 = \text{right}(2v)$) to the left, and for each leaf i we will recalculate $\texttt{cover}(v, i, c_1)$. Among all the vertices of this set, we want to be able to select a tree with a weight closest to a certain value at any moment. Note that the entire set $\texttt{cover}(v, i, c_1)$ consists of "complete" vertices, for which each vertex of the subtree also lies in this set, and "incomplete" ones, for which some of the vertices of the subtree have not yet entered this set. There are no more than $\log_2 n$ "incomplete" vertices, and the rest can be stored together with the weights of their subtrees in some kind of search tree that allows finding the nearest value element in logarithmic time (for example, in a Cartesian tree).

Then for each leaf i: we go through its ancestors, update the weights of their subtrees, those that have become "complete" are thrown into the search tree. After that, if for some query $l = i$, we look for a subtree of the required weight in the search tree, and check $\mathcal{O}(\log n)$ "incomplete" subtrees manually. A similar process is performed for the right borders, only starting from the leaf $c_2 = \text{left}(2v + 1)$ to the right up to $\text{right}(v)$.

Clarification about weights: in fact, the words about "weight closest to one third" are not entirely correct. It can be proven that at least one of x and y must have a subtree weight that is one of the two closest to $\frac{\texttt{weight}(V)}{3}$. But if one of the vertices has already been chosen, for example, x, then y must be chosen with a subtree weight closest to $\frac{\texttt{weight}(V) - \texttt{subtree}(x)}{2}$. Therefore:

1. we will make a pass through the leaves in the left subtree of v, find candidates for x with a weight closest to one third of the weight of V (the nearest not smaller and the nearest not larger);
2. we will make a pass through the leaves in the right subtree of v and for each corresponding candidate x find the most suitable y; at the same time, we will find candidates for y with a weight closest to one third of the weight of V;
3. we will make another pass through the left subtree, and find for each candidate y the most suitable x;
4. from all the found pairs of candidates (x, y) we will choose the optimal pair.

The running time of such a solution is $\mathcal{O}(n \log^2 n + m \log n)$, the first term comes from the fact that we will put each vertex once in the search tree during the processing of each of its ancestors, and the second from the response to queries.

It remains only to find answers to queries for which x and y lie on the same side of $v = \text{lca}(l, r)$. To do this, note that this can be beneficial only if the weight of some part of V on one side of v, including the weight of v, is less than the current answer found. This can only happen for one of the two sides (left or right), so we will "postpone" this query to the corresponding child of v.

For this, we will represent our queries not as (l, r), but as (l, r, Δ), where for queries from the input data $\Delta = 0$. This will mean that we are not minimizing $\max(w_v, w_x, w_y)$, but $\max(w_v + \Delta, w_x, w_y)$. "Postponing" the query down the tree, we will increase its Δ by the weight of the part that we left "above" — later this part will be attached to the upper of the three parts obtained from the postponed query. Such queries can be answered in the same way described above, but now we need to look for weights closest to $\frac{\texttt{weight}(V) + \Delta}{3}$. The running time of the responses to queries then becomes $\mathcal{O}(m \log^2 n)$, because each query generates no more than $\log_2 n$ new queries, postponed down the tree.