Problem A. Colony of Bacteria

Author of the problem: Egor Yulin, developer: Nikolay Vedernikov

This problem involves deriving a formula. There are many patterns that need to be noticed to derive the final formula. Let's consider one of them.

Consider the horizontal and vertical lines where the bacteria are initially placed. Every second, it expands upwards and downwards, so in one second, the colony will fit into a square of size $(2k-1) \times (2k-1)$. However, every odd second, we have an unfilled `corner` of the square growing. It can be observed that its length will be equal to $\lfloor (k-1)/2 \rfloor$. Therefore, the area will be equal to $1 + 2 + .. + \lfloor (k-1)/2 \rfloor$. Using the formula for the arithmetic progression, we find that the area of the corner is $\lfloor (k-1)/2 \rfloor \cdot \lfloor (k+1)/2 \rfloor / 2$. Since there are four such corners, the total area cut off from the square will be $4 \cdot \lfloor (k-1)/2 \rfloor \cdot \lfloor (k+1)/2 \rfloor / 2 = 2 \cdot \lfloor (k-1)/2 \rfloor \cdot \lfloor (k+1)/2 \rfloor$. Thus, the area of the bacterial colony will be equal to $(2k-1) \cdot (2k-1) - 2 \cdot \lfloor (k-1)/2 \rfloor \cdot \lfloor (k+1)/2 \rfloor$.

Problem B. Two-Story Advent Calendar

Author and developer of the problem: Rita Sablin

Let's consider one possible solution to this problem.

First, we will represent each box as a segment on a line. Each box i is characterized by its left and right boundaries l_i and r_i, as well as the number x_i written on it. The first boxes on both levels have a left boundary at point 0.

We will divide all boxes into two types: good and problematic. For a good box, if it is on the upper level, then all boxes it lies on have a greater number, and vice versa for a good box on the lower level. Problematic boxes are those that are under boxes with a greater number or lie on boxes with a smaller number. In this problem, boxes that only intersect at their ends do not interfere with each other. We will not consider boxes that only intersect at their ends as lying on top of each other.

Formally, box i with boundaries l_i and r_i and number x_i is considered good if for any box j on the other level with boundaries and number l_j, r_j, and x_j, if $l_j \le l_i$ and $r_j > l_i$, or $l_j < r_i$ and $r_j \ge r_i$, or $l_i \le l_j < r_j \le r_i$, then $x_i < x_j$ if i is on the upper level, and $x_i > x_j$ in the opposite case. Problematic boxes are all the others.

Note that a good box can always be left in place. We will determine for each box, using two pointers, whether it is good or problematic. Also, for each problematic box i on the upper level, we will find the right boundary of the rightmost problematic box j on the lower level that it lies on, and for each problematic box j on the lower level, we will find the right boundary of the rightmost box i on the upper level that it lies under. Additionally, for each problematic box i on the upper level, we will need to keep the right end of box j on the lower level such that $l_j < l_i < r_j$, if such a box exists. For each problematic box j on the lower level, we will keep the right end of box i on the upper level such that $l_i < l_j < r_i$, if such a box exists. We will call this boundary r. If such a box cannot be found, we will keep the left end of the current box in r.

Next, we will solve this problem using dynamic programming. The parameter for dynamic programming will be the coordinates of the ends of the boxes. We will represent the coordinates as events and sort them in non-decreasing order. We denote $dp[i]$ as the minimum number of boxes that need to be removed to make the calendar from coordinate 0 to i inclusive convenient. Note that there may be pairs of identical coordinates in the event array if boxes end at the same point. We set $dp[0] = 0$.

If the box is good, then $dp[i] = dp[i-1]$.

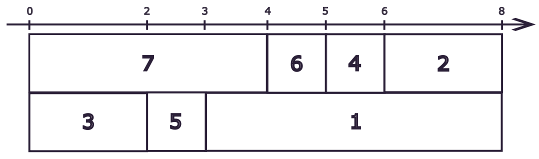
If the box ending at i is problematic and it lies on or under a problematic box that ends to its right, then box i cannot be left if we want to make the calendar from 0 to i convenient. In this case, $dp[i] = \min(dp[l]+1, dp[r]+1)$, where $dp[l]$ is the minimum number of boxes that needed to be removed before the left boundary of the current box. r may coincide with l, but if not, then $dp[r]$ may be less than

$dp[l]$ — if the result at the end of the box it lies on is better, but it was calculated later since the current box starts before the one it lies on or under ends.

If the box ending at i is problematic and it lies on or under a problematic box that ends to its left or at the same point, then we need to choose the best result between removing the current box $dp[i] = \min(dp[l] + 1, dp[r] + 1)$ and $dp[i - 1]$ — the previous coordinate will correspond to the situation where we leave the current one but remove the problematic one on or under which the current one lies.

The answer will be in $dp[a_1 + a_2 + \ldots + a_n]$.

Example in the figure:



$dp[0] = 0$. $dp[2] = 1$, because box number 3 lies on box number 7, which ends to the right, so box 3 cannot be removed for the calendar to be convenient at coordinate 2. $dp[3] = 1$, because box number 5 lies on box number 7, which ends to the right, so box 5 cannot be removed for the calendar to be convenient at coordinate 3. Then $dp[3] = dp[2] + 1$, where 2 is the left boundary of box 5.

$dp[4] = dp[0] + 1 = 1$, because box 7 cannot be removed for the calendar to be convenient at coordinate 4. $dp[5] = dp[4] + 1 = 2$, $dp[6] = dp[5] + 1 = 3$ for the same reason. $dp[8]$ will be considered twice. If we consider 8 as the right boundary of box 2, then it cannot be removed, and $dp[8] = dp[6] + 1 = 4$. When we consider 8 as the right boundary of box 1, it is beneficial to remove box 1 rather than 2, 4, 6, under which it lies, but the leftmost problematic box that lies on 1 — box 7 — ends to the right of where 1 starts, and the dynamic value for the right end of 7 is better. $dp[8] = \min(dp[3], dp[4]) + 1 = 2$. In the end, the answer for this example will be 2 — we need to remove boxes 1 and 7.

Problem C. Intermediate Verticality

Author and problem developer: Mikhail Ivanov

It turns out that in any connected graph, it is possible to construct a tree of any intermediate verticality in linear time. We start with a tree with the maximum possible verticality — a depth-first search tree, which has a verticality of $m - n + 1$. If we perform $(m - n + 1) - h$ times a transform that increases the number of horizontal edges by one, then we will obtain the required tree. Firstly, we need to learn how to perform this transform in linear time. We will traverse the graph along the spanning tree and find a vertical edge uv with the largest depth (distance to the root along the tree edges) of vertex u (assuming that the depth of v is greater than that of u). Since v has a positive depth, this vertex has a parent w; thus, there is a tree edge wv in the graph. Hence, let us remove the edge wv from the spanning tree and add uv instead. Then, the status of no edge, except for uv and wv, will change — this can be easily established by case analysis, using the fact that u is the deepest possible; wv will become horizontal from being a tree edge, and uv will become a tree edge from being vertical. Thus, the required transform can be implemented in linear time, resulting in a quadratic algorithm with a time complexity of $\mathcal{O}(m(m + 1 - n - h))$.

To perform all of the above in total linear time, we only need to learn how to execute the transform described above in $\mathcal{O}(1)$ time (possibly with linear preprocessing). Note that the required re-hanging not only does not add new vertical edges (and always removes only one old edge, while leaving the others unchanged), but also cannot change the depth of the top vertex u of any vertical edge uv. Therefore, the following algorithm will work: we will run a depth-first search once at the very beginning and obtain a normal spanning tree. During this process, we will store the parent of each vertex in an array p, and also sort all vertical edges by the depth of their top vertex using counting sort. We will iterate through the last $(m - n + 1) - h$ edges from this list (i.e., with the deepest u) and assign $p[v] := u$ to them. The edges connecting v and $p[v]$ will become horizontal. This algorithm has a time complexity of $\mathcal{O}(m)$.

Problem D. Two Arrays

Author and problem developer: Pavel Skobelin

Let's rephrase the problem. Let's consider a pair of numbers $(a_i,\, b_i)$ as a point on a plane. Then, using the given action, we can move this point diagonally by one cell $(+1,\, +1)$. The goal is to place all points within a rectangle of size $x \times y$ using this action.

It is clear that a specific point will always remain on the same diagonal. Let's number the diagonals: we say that the point $(x,\, y)$ belongs to the diagonal numbered $y - x$ (the diagonal number can be negative).

Thus, the criterion for the impossibility of a solution is clear: we find the number of the maximum diagonal — D_{max}, and the minimum diagonal — D_{min}, on which there is at least one point. It is not difficult to show that a solution always exists if $D_{max} - D_{min} \le x + y$ (since in a rectangle of size $x \times y$, the maximum difference in diagonal numbers is $x + y$ — that is, from the upper left to the lower right point). Otherwise, a solution does not exist.

Now let's understand how to find the number of necessary actions. Let us define:

$$X = \max_{i \in [1 \ldots n]} (x_i), Y = \max_{i \in [1 \ldots n]} (y_i)$$

Our goal is to fit all points into the rectangle $x \times y$. Let's determine where its upper right corner should be located. Obviously, its x-coordinate must be at least X (otherwise, it would be impossible to place a point with the first coordinate X in the rectangle, since performing an action on it would only increase its first coordinate). Similarly, its y-coordinate must be at least Y. So initially, we set the coordinates of the upper right corner to $(X,\, Y)$ — that is, the minimally possible values.

How can we minimize the number of actions? In fact, we need to choose the minimally possible coordinates x and y for the upper right corner (that is, so that they cannot be decreased).

If all points are already within the rectangle or can be moved there, then this is the optimal arrangement. If not, which points could be problematic (that is, which cannot fit into the current rectangle)? If such points exist, they are the points with the minimum or maximum diagonal. How do we determine that a point with the maximum diagonal does not fit into the rectangle? We need to check that its diagonal number is greater than the maximum diagonal number of the considered rectangle: $D_{max} > (Y - X) + x$. If this is the case, then let's move the rectangle up by $\Delta Y = D_{max} - (Y - X) - x$ $(Y := Y + \Delta Y)$. It is clear that this is a necessary condition for the point with the maximum diagonal to fit into the rectangle.

A similar reasoning applies to the point with the minimum diagonal: if $D_{min} < (Y - X) - y$, then we move the rectangle to the right by $\Delta X = D_{min} - (Y - X) + y$ $(X := X + \Delta X)$. It is also clear that this is a necessary condition, meaning that without it, we would not be able to obtain any solution.

Note that from the two actions above, we will perform at most one, meaning that both conditions cannot be satisfied simultaneously:

$$D_{max} > (Y - X) + x \tag{1}$$
$$D_{min} < (Y - X) - y \tag{2}$$

Because in such a case, by multiplying (2) by -1 and adding the two inequalities, we get

$$D_{max} - D_{min} > (Y - X) + x - (Y - X) + y = x + y$$

$$D_{max} - D_{min} > x + y$$

And we have ruled out such a case from the very beginning.

Thus, initially, both conditions could not be satisfied. It is also not difficult to show that after increasing X, the first condition could not start being satisfied $(D_{max} > (Y - X) + x \ge (Y - (X + \Delta X)) + x)$.

Similarly, the first condition could not be satisfied after increasing Y. Therefore, we made the necessary adjustment to the coordinates. It is evident that this adjustment is sufficient: it is clear that if the diagonal number of the vertex fits into the required rectangle, then the vertex itself can fit into it; otherwise, it is located above/right of it — which is impossible, since we initially chose X and Y as the maximum coordinates.

Thus, we have found the coordinates of the upper right corner of the rectangle. Now we just need to calculate the number of actions: it is clear that for a specific point $(x_i,\ y_i)$, the minimum number of actions to fit into the rectangle is $\max(0, X - x[i] - x, Y - Y[i] - y)$. We just need to sum this value over all points.

We obtain a solution to the problem in $\mathcal{O}(n)$ for each test case.

Problem E. Classics

Problem author: Fedor Ushakov, developer: Mikhail Perveev

First, let's construct an array $q_1, q_2, \ldots, q_n$, where q_i is the position of element i in the final array. There are many ways to construct this array.

For example, we can use a segment tree. We will maintain a segment tree over an array t of length n, where the i-th element will store 0 if the position i in the final array is not yet occupied, and 1 otherwise. We will then process addition queries from the end. Initially, we will assign 0 to all elements of the array t.

Suppose we want to process the addition of the number i. To determine its position in the final array, we need to find the p_i-th zero in the array t. The position of this zero will be the position of the number i in the final array. After that, we need to replace 0 with 1, and also assign the position of the found zero to q_i. Finding the k-th zero and updating the array element can be done using the segment tree in $\mathcal{O}(\log n)$ time.

If using C++, one can utilize the data structure `__gnu_pbds::tree` (this data structure is also known as `ordered_set`) instead of a segment tree.

Now that we have constructed the array q, we can proceed to solve the problem. Consider a moment in time when the numbers $1, 2, \ldots, i$ have been added to the array. Consider a certain sequence of numbers $x_1, x_2, \ldots, x_k$, such that $1 \le x_1 < x_2 < \ldots < x_k \le i$. This sequence will be an increasing subsequence of the array if and only if its elements are present in the array in that exact order. In other words, the inequality $q_{x_1} < q_{x_2} < \ldots < q_{x_k}$ must hold. Thus, the length of the LIS (Longest Increasing Subsequence) of the array after adding the first i numbers is equal to the length of the LIS of the array $q_1, q_2, \ldots, q_i$. Now the problem has reduced to finding the LIS for each prefix of the constructed array q.

To do this, we will use the classical solution for finding the LIS in $\mathcal{O}(n \log n)$ time. We will add elements one by one from left to right, maintaining an array $dp[i]$, where $dp[i]$ is the minimum value of the element that can end an increasing subsequence of length i. We will also maintain the current answer—the maximum length of the increasing subsequence that we can obtain.

Suppose we have considered the elements $q_1, q_2, \ldots, q_{i-1}$ while maintaining the array dp, and the current length of the LIS is k. Now we will consider the element q_i. To do this, we will use binary search to find the minimum number j such that $dp[j] > q_i$. After that, if $dp[j-1] < q_i$, we will assign $dp[j]$ the value of q_i. Finally, if the value of $dp[j]$ was updated, we will assign the variable k the value $\max(k, j)$.

Thus, both parts of the solution work in $\mathcal{O}(n \log n)$ time.

Problem F. Exchange and Deletion

Author and problem developer: Valery Rodionov

At each step, the last element jumps to the left and overwrites one of the previous elements. Note that the elements $1, 2, \ldots, n-k$ never jump, so their relative order is preserved. Let's see which elements will be removed after k steps. These will be the elements $n-k, n-k-1, \ldots, n-k-l+1$ (that is, some suffix), because otherwise the sequence would not be increasing. These elements will be overwritten by some l elements from $n-k+1$ to n. We will fix a set of l elements from $n-k+1$ to n that will remain after k steps. We will call them good, while the remaining $k-l$ will be called bad. We will fix the final position of each good element (that is, which of the elements $n-k, n-k-1, \ldots, n-k-l+1$ it will overwrite). The bad elements are indistinguishable, so we can consider that they do not jump but simply choose one of the bad elements to the left or the final position of some good element to jump to, while remaining in place.

There are a total of $k-l$ bad elements. Let the i-th ($1 \le i \le k-l$) bad element be b_i. We will choose a value p_i for each bad element, which is either a position from $n-k+1$ to b_i, or some good element greater than b_i. We will construct a bijection between the sequences $p_1, p_2, \ldots, p_{k-l}$ and the sequences of deletions that place the good elements in their established final positions, as follows:

Let there be a sequence $p_1, p_2, \ldots, p_{k-l}$. We will make k steps. At each step, we will do the following. Let the last element be good. Then, if it is chosen as p_i for some bad element, it jumps to the position of the rightmost of such bad elements. Otherwise, it jumps to its final position. Let the last element be bad. Then it has chosen some position to the left of itself as p_i (since otherwise it would have already been overwritten by some good element). If there is a good element at position p_i, we will jump to the final position of this good element; otherwise, we will simply jump to position p_i.

It is easy to see that in this way we can obtain any sequence of deletions that places all good elements in their final positions.

The number of ways to choose p_i is equal to $(k-l+i)$ (equivalent to choosing any good element or a bad element to the left), so the total number of ways is $k \cdot (k-1) \cdot \ldots \cdot (k-l+1) = k!/(k-l)!$.

We also need to multiply by $l!$ (the choice of final positions for good elements) and $\binom{k}{l}$ (the choice of good elements). Thus, the answer is the sum over l from 0 to $min(k, n-k)$ of the values $k!/(k-l)! \cdot l! \cdot \binom{k}{l}$.

Problem G. M-11 Highway

Problem authors: Alexandra Olemskaya, Alexander Ponkratov, developer: Alexander Ponkratov

Consider all points of type 1 and remember how many points of type 0 are to the left of it, let's denote this count as cnt_i. Next, for each point of type 1, find the farthest point from the current one such that the distance between them does not exceed k, let its index be j. Then all points of type 1 from j to $i-1$ can potentially form a convenient triplet with point i, specifically $cnt_i - cnt_p$ ($j \le p < i$) triplets. Thus, we need to add to the answer $\sum\limits_{p=j}^{i-1} cnt_i - cnt_p = (i-j) * cnt_i - \sum\limits_{p=j}^{i-1} cnt_p$. The farthest point can be found using binary search or two pointers, and the contribution to the answer for the current point can be calculated using prefix sums. Depending on the implementation, we can achieve a solution in $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$.

Problem H. Exploration Robots

Author and problem developer: Aleksandr Babin

Since the robots know their positions relative to each other, they can visit all the fields that are located between them. After they do this, some segment $[\min\{x,y\}, \max\{x,y\}]$ of fields between them will be visited. Furthermore, this segment can be expanded by one field on the left and right boundaries.

We will call a string u a *border* of the string s if u is simultaneously a suffix and a prefix of the string s. It is claimed that if some experiment ends with the robots visiting the segment of fields $[l, r]$, then the substring $s[l \ldots r]$ is a border of the string s.

Moreover, if at some point the robots visited a segment of fields $[l, r]$ such that $s[l \ldots r]$ is not a border, then $s[l \dots r]$ is neither a suffix nor a prefix of the string; thus, the robots can guarantee the expansion of this segment either to the left or to the right.

Therefore, we can reformulate the problem as follows: for a query (x, y), it is required to find a segment of minimal length $[l, r]$ such that $x, y \in [l, r]$, and also $s[l \dots r]$ is a border of the string s.

To do this efficiently, we will separately find the left and right boundaries. We will describe how to find the right boundary (the left one is found in the same way, but the string s needs to be reversed first).

We will compute the array $\pi_1, \dots, \pi_n$—the *prefix function* of the string s, which is an array such that $\pi_1 = 0$ and π_k equals the length of the longest non-trivial border of the string $s[1 \dots k]$. It is clear that $n, \pi[n], \pi[\pi[n]], \dots$ are the lengths of all borders of the string s. Then, in linear time, we can easily compute the array $q_1, \dots, q_n$ using dynamic programming, where q_k is the smallest value such that $s[q_k \dots k]$ is a border of the string s.

Now consider the query (x, y); we need to find the smallest $k \ge \max\{x, y\}$ such that $x, y \in [q_k, k]$. This will be the border with the smallest right boundary that contains the pair of positions x, y. We can find such a k for each query, for example, in an offline manner by sorting all queries in increasing order of $\min\{x, y\}$. After that, the set S, consisting of numbers k such that $q_k \le \min\{x, y\}$, will only expand, which means it can be maintained in `std::set` with a total time of $O(n \log n)$. Then, to answer the query, it is sufficient to use the `.lower_bound` method. As a result, we obtain an algorithm that runs in $O(n \log n)$ time.

Remark. It may seem incorrect that we are separately searching for the left and right boundaries in each query. A small mathematical reasoning below should convince the reader otherwise.

Lemma. Let $[l, r]$ and $[a, b]$ be segments such that $s[l \dots r]$ and $s[a \dots b]$ are borders of the string s; then $J = [l, r] \cap [a, b]$ is also a border of the string s. This is true because $s[J]$ is a suffix of either the string $s[l \dots r]$ or $s[a \dots b]$, and is also a prefix of one of these two strings. Thus, $s[J]$ is a suffix of some border of the string s, which means it is a suffix of the string s, and similarly, $s[J]$ is a prefix of the string s. Therefore, $s[J]$ is both a prefix and a suffix of s, which means $s[J]$ is a border of the string s.

Let $s[l \dots r]$ be the minimal border of the string s that contains the pair of positions (x, y), and let $s[p \dots r']$ and $s[l' \dots q]$ be borders of s, respectively, with the minimal right boundary and the maximal left boundary that contains the pair of positions (x, y). Then $l \le l' \le r' \le r$ by the definition of $s[p \dots r']$ and $[l' \dots q]$. But also, $s[l' \dots r']$ is a border of s by the lemma. However, due to the minimality of the length $[l, r]$ and the nesting $[l', r'] \subseteq [l, r]$, we have $[l', r'] = [l, r]$, which is what we needed to prove.

Problem I. Prank

Author and problem developer: Nikolay Vedernikov

We will create a stack and two pointers, one for each string. If the characters at the pointers in the strings match and there are no letters on the stack, we consider that this letter has been preserved from the initial string (and we move both pointers forward). Otherwise, there are two possible situations:

- The top of the stack has exactly the same letter. This means we can consider that these two identical letters were placed together, as we have already removed the other paired letters, if there were any. We will remove the letter from the stack and move the pointer in the string s_2.
- The stack is empty or the top has a different letter. In this case, we will place this letter on the stack and move the pointer for s_2.

If we have examined all the characters of the string s_1 (we can check that the pointer has reached the end of the string) and the stack is ultimately empty, we consider that it is possible to obtain the string s_2 from the string s_1 by applying `pranks`; otherwise, it is not possible.

Problem J. Nightmare Sum

Author and problem developer: Evgenii Pakhomov

Let $C = 300\,000$. Since all elements of the array are distinct, the total number of pairs $(\min, \texttt{quotient})$ is $O(C \log C)$ (the sum of the harmonic series).

Fix i — the position of the minimum. Notice that the quotient $\left\lfloor \frac{\max}{\min} \right\rfloor$ can only take values $d = 1, 2, \ldots, \left\lfloor \frac{C}{a_i} \right\rfloor$. As already noted, all these values can be honestly enumerated. Let us assume we know i and d. What segments will suit us?

The minimum on the segment must be a_i, so $l \le i \le r$ and there should be no numbers less than a_i on the segment. To achieve this, we can traverse the array from left to right and from right to left while maintaining a stack of minimums. Then for all i, we can count the nearest number in the array to the right/left of i that is less than a_i.

It remains to understand what the maximum on the segment should be. Since we have fixed d, the maximum must lie within the range $[a_i \cdot d; a_i \cdot d + a_i - 1]$. What restrictions does this condition impose on l and r? The segment must not contain numbers greater than or equal to $a_i \cdot (d+1)$, so l must be strictly greater than the index of the nearest number to the left of i that is greater than or equal to $a_i \cdot (d+1)$. Similarly, r must be strictly less than the index of the nearest number to the right of i that is greater than or equal to $a_i \cdot (d+1)$. Additionally, there are restrictions on l and r from the other side: the maximum must be at least $a_i \cdot d$, meaning l must not exceed the index of the nearest number to the left of i that is greater than or equal to $a_i \cdot d$. Similarly, r must not be less than the index of the nearest number to the right of i that is greater than or equal to $a_i \cdot d$.

Finding the nearest number to the left/right that is greater than or equal to x can be done using a segment tree or binary search on the maximums stack. Thus, counting the specified restrictions on l and r will take $O(n \log n \log C)$ time.

Therefore, for a specific pair (i, d), we have restrictions of the form: $x < l \le q$ and $p \le r < y$. How do we recalculate the answer?

Let $dl = q - x$, $dr = y - p$. We want to add to the answer $d \cdot \texttt{number of good segments}$. This value can be easily computed using the inclusion-exclusion formula:

$$\texttt{number of good segments} = dl \cdot (y - i) + dr \cdot (i - x) - dl \cdot dr$$

Thus, the asymptotic complexity of the final solution will be $O(n \log n \log C)$.

Problem K. Petya's Cryptography

Problem author: Demid Kucherenko, developer: Vladimir Riabchun

Any tree with $n \ge 2$ vertices can be constructed with the following algorithm: we start with an edge, on every step we chose a leaf v and a number $i \ge 1$, after it we link i new nodes with the leaf v. After it the number of paths of length 2 is increased by $\frac{i(i+1)}{2}$.

To solve this problem let's compute $dp[n][s]$, which is equal to true if we can construct a tree with n vertices with s paths of length 2.

$$dp[n][s] = \bigvee_{1 \le i \le n} dp[n-i][s - \frac{i(i+1)}{2}]$$

This value should be evaluated lazily, the number of calculated elements does not exceed ≈ 2000.

Problem L. Two Scooters

Problem author: Nikolay Vedernikov, developer: Ekaterina Vedernikova

To solve the problem, you need to calculate two values and compare them. For company `W`, you need to divide t by 60 to get the integer part, then multiply the resulting integer by 60. The resulting number of seconds should be multiplied by the price c_1 to obtain the cost of the trip on the scooter of company `W` in kopecks.

For company `Y`, you need to multiply t by the price c_2. After that, the resulting number should be divided by 100; if there is a remainder from the division, you should add 1 to the quotient. The resulting integer number of rubles should be multiplied by 100 to also obtain the cost in kopecks.