Problem A. Anxious About Airdrops

Let's make the following observation: if we have only two points, we can merge them into one by placing an attraction point roughly in the middle between them.

It turns out that if you take a point and its nearest neighbor, this same method will work successfully for any number of points.

Indeed, let A and B be a point and its nearest neighbor among the remaining points (excluding A itself), and let M be any "middle" point on the segment between them (in the sense that the distances between A and M and between B and M differ by no more than 1, and their sum is equal to $|AB|$). Then for any $C \neq A$, due to the triangle inequality, $|MC| \geq \min(|AM|, |BM|)$ holds (otherwise $|AB| = |AM| + |BM| > |MC| + |AM| \geq |AC|$);

This observation is enough to write a compete solution with time complexity of $O(n^2)$;

Problem B. Base By Base

There are several ways to solve this task.

One idea is to ask two bases of the form $p_i + 1$ (where p_i are prime numbers such that $p_1 \cdot p_2 \geq 10^{15}$). Using the sum of digits and the property that $S_{x,b} = x \bmod (b-1)$, we know two remainders $n \bmod p_1$ and $n \bmod p_2$, then we can apply the Chinese remainder theorem.

Another solution is based only on simple arithmetics. Lets ask 10^9 and 999 999 999. After the first query, we will have the equation $n = 10^9 a + b$, where $0 \leq a, b \leq 10^9$ (more, $a \leq 10^6$ due to the limitations on n). After the second query, we will have the notation $n = (10^9 - 1) \cdot a_1 + b_1 = 10^9 \cdot a_1 + (b_1 - a_1)$.

1. If $a_1 > b_1$, then $n = 10^9 \cdot (a_1 + 1) + (10^9 + b_1 - a_1)$, where both brackets are positive and less than 10^9, i.e $a_1 + 1 = a$ and $10^9 + b_1 - a_1 = b$, so $a + b = a_1 + 1 + 10^9 + b_1 - a_1 = 10^9 + 1 + b_1$ that is greater than $a_1 + b_1$.
2. If $a_1 \leq b_1$, then $n = 10^9 \cdot a_1 + (b_1 - a_1)$, where both brackers are positive and less than 10^9, i.e $a_1 = a$ and $b_1 - a_1 = b$, and $a + b = a_1 + b_1 - a_1 = b_1$ that is less or equal than $a_1 + b_1$.

After the first query we know $a + b$, after the second query we know $a_1 + b_1$. Comparing them, we can find the case and then derive the values of a, b, a_1, b_1 and then finally calculate n.

Problem C. Cheater Changes Costs

Consider two rounds with players i, j. If $a_i + a_j > s$, then players cannot agree in both rounds and will get 0. Otherwise in the round where i is proposer they will get $s - a_j$ coins, in the round where i is responder they will get a_i coins. So in two rounds with player j player i will get $s - a_j + a_i$ coins.

Let's define $f_j(x) = \begin{cases} s - a_j + x, \text{ if } x \leq s - a_j \\ 0, \text{otherwise} \end{cases}$.

If player i will change a_i to x they will get $\sum\limits_{\substack{j=1 \\ j \neq i}}^{n} f_j(x)$ coins.

Our task is to find the maximum value of the function $\sum\limits_{\substack{j=1 \\ j \neq i}}^{n} f_j(x)$ and the number of points where maximum holds for each i.

Let's define $sum(x) = \sum\limits_{j=1}^{n} f_j(x)$. Then we should find the maximum of the function $sum(x) - f_i(x)$ and the number of points where maximum holds for each i.

Let's note, that the function $sum(x) - f_i(x)$ is piecewise linear with vertices in $x = s - a_j$ for all $j \neq i$. If $a_j = s$ for all $j \neq i$, then $sum(x) - f_i(x) = 0$ for all x and maximum of the function is 0 and it holds for all $s + 1$ values of x. Otherwise the maximum can be only in vertices $s - a_j$.

Let's consider a sorted array of all possible vertices $s - a_j$ in sorted order. We can find the value of $sum(x)$ in these points with prefix sums.

When we subtract $f_i(x)$ from $sum(x)$ we subtract $s - a_i + x$ on prefix $[0, s - a_i]$ from the function. Let's calculate prefix maximums of $sum(x) - x$ and suffix maximums of $sum(x)$ together with the number of points where they hold. Having these values we can calculate the maximum of $sum(x) - f_i(x)$ and the number of points where it holds in $O(1)$.

So we get the total complexity $O(n \log n)$ for sorting coordinates and $O(n)$ for solution.

Problem D. Different Dominoes Divisions

If R is a subrectangle of the board of size $2^i \times 2^{i+1}$ for some i (possibly rotated), we will say that it *admits* a domino tiling τ of R if this tiling can be obtained using the splitting algorithm from the statement. We also define $f(R)$ as the number of tilings of this rectangle that contain all already placed dominoes intersecting this rectangle and that R admits. Also, let $\mathcal{P}(R)$ be the family of the sets of subrectangles we can divide R into in a single step of the algorithm (so, in particular, $|\mathcal{P}(R)| = 5$ unless R is a domino). The answer for the problem is f(the whole board), and if every possible combination of splitting choices on each step resulted in a unique tiling, we would have the following recursive formula for f:

$$f(R) = \sum_{S \in \mathcal{P}(R)} \prod_{s \in S} f(s),$$

and if R is a domino, then $f(R) = 1$ when R doesn't contain a half of an already placed domino and 0 otherwise.

However, since some tilings could be obtained in various ways, this formula may count some tilings more than once. To correct the formula, we will use the inclusion-exclusion principle:

$$f(R) = \sum_{\varnothing \neq \mathcal{S} \subseteq \mathcal{P}(R)} (-1)^{|\mathcal{S}|+1} \cdot (\text{the number of tilings possible for all choices from } \mathcal{S}).$$

Fix some family $\mathcal{S}$ and see what it means for a tiling τ to be admitted by all partitions $S \in \mathcal{S}$ on this step (a partition admits a tiling if every its rectangle admits its corresponding subtiling). First of all, if we consider R as a 2×4 rectangle consisting of big cells, then each partition $S \in \mathcal{S}$ draws some borders between some cells of R, thus dividing R into dominoes and squares. For a tiling τ of R to be achievable by all partitions from $\mathcal{S}$, we need all *dominoes* in the partition to admit τ. Let's find out the criteria that must hold for the *squares*.

Each square of the intersected partition is a part of (at least) two different dominoes corresponding to where this square belongs in different scenarios. We claim that it is necessary and sufficient that every such square can be divided into two equal halves (horizontally or vertically) in such a way that each of these halves is a domino that admits τ. Sufficiency is obvious: if every square can be splitted into two halves that admit τ, then every rectangle r of every partition $S \in \mathcal{S}$ is either a domino in our picture, and therefore admits τ by assumption, or two squares, each of which can be divided into two dominoes that admit τ, which provides a division from $\mathcal{P}(r)$. We are going to prove the necessity.

Assume the contrary and consider the smallest possible square that contradicts the assumption. More specifically, consider a rectangle R of the smallest size $2^i \times 2^{i+1}$, its square half C, and a tiling τ so that R admits τ, each domino of τ is either inside C or outside C, but C cannot be divided into two halves both admitting τ. Since R admits τ, there is a division $S \in \mathcal{P}(R)$ so that all rectangles of S admit τ. If two of these rectangles are halves of C, this contradicts the assumption. Otherwise, we know exactly what S is (it is that partition that looks like two horizontal dominoes in the middle and two vertical on the sides, like |=|). We know that S has one rectangle r completely in C and two halves of rectangles that represent $C \setminus r$. Due to the minimality of our example, these squares of size $2^{i-1} \times 2^{i-1}$ can be split into halves, each of which admits τ. Combining these divisions gives us a partition of $C \setminus r$. Contradiction, because we splitted C into two halves r and $C \setminus r$, both admitting τ.

If we extend the definition of admitting a tiling and the function f onto squares in a natural way, we obtain the following recursive formula for f: if R is a rectangle $2^i \times 2^{i+1}$, then

$$f(R) = \sum_{\varnothing \neq \mathcal{S} \subseteq \mathcal{P}(R)} (-1)^{|\mathcal{S}|+1} \prod \left\{ f(s) : \ s \text{ is a piece in the partition of } R \text{ by all } r \in \bigcup \mathcal{S} \right\},$$

and if C is a square of size $2^i \times 2^i$, then, in the same inclusion-exclusion manner, if it consists of squares c_{00}, c_{01}, c_{10}, c_{11} of size $2^{i-1} \times 2^{i-1}$ with indices denoting the locations, then

$$f(C) = f(c_{00} \sqcup c_{01}) f(c_{10} \sqcup c_{11}) + f(c_{00} \sqcup c_{10}) f(c_{01} \sqcup c_{11}) - f(c_{00}) f(c_{01}) f(c_{10}) f(c_{11}).$$

We can calculate this using dynamic programming. More specifically, for all i from 1 to k in increasing order, and for every rectangle R of sizes $2^i \times 2^i$, $2^i \times 2^{i+1}$, $2^{i+1} \times 2^i$, calculate $f(R)$ using the formulas above. To make it fast, we need the following things:

- If we unroll the final answer f(the whole board), we will find that we only require values of f for the rectangles whose corner coordinates are divisible by 2^i. This reduces the number of DP states from about $k \cdot 4^k$ to roughly $3(2^{2k+1} + 2^{2k-1} + \ldots + 2) = 2 \cdot (4^{k+1} - 1)$.
- In the formula for a rectangle, precalculate all coefficients in all partition intersections once instead of intersecting them from scratch every time.
- This is optional, but reusing the values of smaller rectangles in computing a single $f(R)$ can make running time 10x faster.

Problem E. Eirt Eht Esrever

Let's first solve a slightly different problem: suppose we insert a string into a trie only if it is reversed. Let's start by solving this problem in quadratic time, it's straightforward: we will insert all reversed prefixes into a trie, and in each of the terminal vertices, we will record the probability of the corresponding string appearing in the trie.

Now, for each vertex, the probability that it does not appear in the trie is the product of $1 - p_i$ for all terminal vertices in the subtree, which can be easily computed by a simple depth-first search. To solve it with better time complexity, we can use suffix structures.

Let's build a suffix tree (compressed suffix trie) of `reverse`(s). Each leaf corresponds to some reversed prefix of the string. For each prefix put the number p into the vertex corresponding to its reverse (the probability that this reversed string will not be inserted into the trie) and then run `dfs` to calculate t_v — the product of all values in the subtree for each vertex. To calculate the expected number it takes to sum $1 - t_v$ over all vertices of the tree.

Now let's return to the original problem and remember that we forgot to add the non-reversed prefixes. To do this, we'll traverse down the suffix tree, following the characters of the original string, until the string ends (or until we "fall out" of the suffix tree). Note that all the visited vertices will necessarily be in the final trie, as they are part of a certain reversed prefix as well as part of a (same length) non-reversed prefix;

We still need to deal with the suffix of the original string. It is easy to see that for each character in the suffix, we need to add $1 - \prod\limits_{j \geq i} p_j$ to the answer;

This solution words in either $O(n)$ or $O(n \log n)$, depending on the method of constructing the suffix tree.

Problem F. F For Function

Let's note, that $b + i \,|\, a + w \cdot i^2 \iff b + i \,|\, a + w \cdot i^2 + w(b+i)(b-i) = a + wb^2$.

So, $f_w(a, b)$ is the maximum k, such that $b+i \,|, \ a+wb^2$ for all $0 \leq i \leq k \iff \operatorname{lcm}(b, b+1, \ldots, b+k) \,|\, a+wb^2$.

Let's note, that $\sum\limits_{a=1}^{m_a}\sum\limits_{b=1}^{m_b} f_w(a,b) = \sum\limits_{k=1}^{\infty}\sum\limits_{a=1}^{m_a}\sum\limits_{b=1}^{m_b} \mathbb{I}(f_w(a,b) \ge k) = \sum\limits_{k=1}^{\infty}\sum\limits_{a=1}^{m_a}\sum\limits_{b=1}^{m_b} \mathbb{I}(\mathrm{lcm}(b,b{+}1,\ldots,b{+}k) \mid a{+}wb^2)$.

Let's fix b and k. What is the number of possible $1 \le a \le m_a$, such that $\mathrm{lcm}(b,b+1,\ldots,b+k) \mid a+wb^2$? It is equal to the number of integers in $[1+wb^2, m_a+wb^2]$, that are divisible by $\mathrm{lcm}(b,b+1,\ldots,b+k)$. This number is equal to

$$\left\lfloor \frac{m_a+wb^2}{\mathrm{lcm}(b,b+1,\ldots,b+k)} \right\rfloor - \left\lfloor \frac{wb^2}{\mathrm{lcm}(b,b+1,\ldots,b+k)} \right\rfloor$$

So the answer to our problem is:

$$\sum_{b=1}^{m_b}\sum_{k=1}^{\infty}\left(\left\lfloor \frac{m_a+wb^2}{\mathrm{lcm}(b,b+1,\ldots,b+k)} \right\rfloor - \left\lfloor \frac{wb^2}{\mathrm{lcm}(b,b+1,\ldots,b+k)} \right\rfloor \right)$$

Let us consider two separate cases:

Case 1: $k \ge 2$

In this case $\mathrm{lcm}(b,b+1,\ldots,b+k) \ge \frac{b(b+1)(b+2)}{2} \ge \frac{b^3}{2}$.

So, if $b > 2(w+\sqrt[3]{m_a})$ we will have $\frac{b^3}{2} > wb^2 + m_a$, so terms will be equal to zero.

So, in this case we need to consider only $b \le \min(m_b, 2(w+\sqrt[3]{m_a}))$. Let's iterate such b, also iterate k until $\mathrm{lcm}(b,b+1,\ldots,b+k) \le m_a + wb^2$ and make a summation.

So the total complexity in this case will be $O(w+\sqrt[3]{m_a})$ (it can be carefully proved, that there is no additional log multiplier to the complexity).

Case 2: $k = 1$

It is a special case. Here $\mathrm{lcm}(b,b+1) = b(b+1)$ and we need to find the sum:

$$\sum_{b=1}^{m_b}\left(\left\lfloor \frac{m_a+wb^2}{b(b+1)} \right\rfloor - \left\lfloor \frac{wb^2}{b(b+1)} \right\rfloor \right) = \sum_{b=1}^{m_b}\left(\left\lfloor \frac{m_a-wb}{b(b+1)} \right\rfloor - \left\lfloor \frac{-wb}{b(b+1)} \right\rfloor \right) = \sum_{b=1}^{m_b}\left(\left\lfloor \frac{m_a-wb}{b(b+1)} \right\rfloor - \left\lfloor \frac{-w}{b+1} \right\rfloor \right)$$

The sum $\sum\limits_{b=1}^{m_b}\left(\left\lfloor \frac{-w}{b+1} \right\rfloor\right)$ can be easily found by iterating all $b \le \min(m_b, w)$.

We need to find the sum $\sum\limits_{b=1}^{m_b}\left(\left\lfloor \frac{m_a-wb}{b(b+1)} \right\rfloor\right)$.

Let's bruteforce all $b \le w + \sqrt[3]{m_a}$ and find the sum of these terms. Now assume, that $b > w + \sqrt[3]{m_a}$.

If $b \ge \lceil \frac{m_a+1}{w} \rceil$ then $m_a - wb < 0$ and $0 < |m_a - wb| < b(b+1)$, so $\left\lfloor \frac{m_a-wb}{b(b+1)} \right\rfloor = -1$. These terms can be added to sum.

If $b \le \frac{m_a}{w}$ we have $m_a - wb \ge 0$ and we need to sum $\left\lfloor \frac{m_a-wb}{b(b+1)} \right\rfloor$. Note, that $\frac{m_a-wb}{b(b+1)}$ is decreasing function by b. Due to $b \ge \sqrt[3]{m_a}$ we have $\frac{m_a-wb}{b(b+1)} \le \sqrt[3]{m_a}$. Let's iterate value v and find the number of b, such that $\left\lfloor \frac{m_a-wb}{b(b+1)} \right\rfloor = v$. It can be done by solving the quadratic equation or by binary search. The first way is much harder and can have some numerical problems. The complexity of this step is $\sqrt[3]{m_a}\log m_b$ (with binary search).

The complexity in this case is $O(w+\sqrt[3]{m_a}\log m_b)$. Note, that we can easily make it $O(w+\sqrt[3]{m_a}\sqrt{\log m_b})$ if we will consider all $b \le w + \sqrt[3]{m_a}C$ for some constant C in the bruteforce iteration.

The total complexity of the solution is $O(w + \sqrt[3]{m_a} \log m_b)$, $O(w + \sqrt[3]{m_a}\sqrt{\log m_b})$, or $O(w + \sqrt[3]{m_a})$ depending on the implementation.

Problem G. Giant Gorilla's Gift

First, let's ignore the condition of discarding some numbers and solve the problem without it. For that let's construct a segment tree, but for queries of the second type, we will descend as long as there is at least one element in the subtree that will change as a result of the query.

Notice that this solution will work efficiently (the main thing is not to forget the case of multiplication by 1). Indeed, let's assign each vertex in the segment tree a potential equal to the binary logarithm of the minimum element in it. Then in each vertex from which we descend during a query, this potential increases by at least 1;

Now let's understand that an operation of the first type cannot decrease the total potential by more than $\log_2 n \log_2 10^6$, so the total running time will be $O((n + q) \log_2 n \log_2 10^6)$.

How do we solve the original problem? Note that any number in the range $[1, 10^6]$ has no more than 7 distinct prime factors, so let's maintain 8 segment trees that will compute the answer for $k = 0, \dots, 7$. We now just need to combine these two ideas to get a solution with a running time of $O(k(n+q) \log_2 n \log_2 10^6)$.

Problem H. Hidden Hierarchy Hints

Fix any vertex (for example, vertex 1) as the root and, using n queries, determine which of the edges are of the form (a, b), where a is further from vertex 1 than b (change the orientation of this edge only and observe how the answer changes);

Now we can assume that all edges are directed away from vertex 1. Let's take a string of zeros (i.e., so that all edges are directed away from vertex 1) and make queries with the root at each vertex. The answer to the i-th query will match the distance from vertex 1 to vertex i. Thus, we can find arrays $d_1, \dots, d_{n-1}$, representing the vertices at distance i from vertex 1.

Then create $\lceil \log_2 n \rceil$ series of n queries (within a series, the edge orientations will remain the same, while the root will cycle through all vertices) in such a way that for any two edges there is a series where they are directed in different directions relative to vertex 1 (for example, in the i-th step, consider only those edges where the i-th bit is equal to 1). Using the information retrieved we can determine all edges that lead to vertices at depth 1: for each series, the difference in answers between such a vertex and vertex 1 will be ± 2 (depending on the orientation of the edge between them), so it is enough to look at the series where the difference is 2 and identify which edge was directed away from the root in this set of series (and towards the root in the rest).

Moving on to the vertices at depth 2. Take a vertex v from d_2 and make $\lceil \log_2 |d_1| \rceil$ queries (with the root at v) that "distinguish" the edges leading from d_1 to 1. It turns out that from these answers, we can identify the ancestor of vertex v. Indeed, the answer to a query equals the number of edges directed towards the root (a number we know), and $+1$ if we did not reverse the edge from our ancestor, or -1 if we did. But once we have found all the ancestors of the vertices in d_2, we can determine the numbers of the corresponding edges in the same way we determined the numbers of the edges from d_1 to vertex 1;

By repeating this algorithm for all d_i, we will determine all the edges. The number of queries does not exceed $2n + 2n \log_2 n$, and the time complexity is $O(n^2 \log_2 n)$.

Problem I. Interpreting Indicated Integers

Notice that there are 10^n different states of the display. Thus, we need to count the number of states that meet the condition, and then output this number, inserting a decimal point in the correct place.

Which states of the display will remain valid when rotated 180°? Only those states that include only the digits 0, 2, 5, 6, 8, 9. There are 6^n such states;

Which of these states will map to themselves when rotated? Only those in which all digits (when read from the edges to the center) form pairs like $(2, 2)$, $(5, 5)$, $(6, 9)$, $(8, 8)$, $(0, 0)$. It's easy to see that there are $6^{\lfloor n/2 \rfloor} \cdot 4^{n \bmod 2}$ such states.

Therefore, the answer is $\frac{6^n - 6^{\lfloor n/2 \rfloor} \cdot 4^{n \bmod 2}}{10^n}$, calculated in $O(n)$ time.

Problem J. Jumping Jumbo Jets

Consider the point S in the aisle near the first seat. Obviously, each of the $(k+l) \cdot N$ passengers must pass through this point; since the aisle has a width of 1, this takes $(k+l) \cdot N$ seconds. Additionally, passengers seated in the first row will have to stand an extra second in the aisle $(k+l)$ times (as they turn towards their seats), meaning the total boarding time cannot be less than $(k+l)(N+1)$ seconds.

Let us show that if the optimum $(k+l)(N+1)$ is reached, the last passenger in the queue must be seated in the first row. If this is not the case, there will be an additional moment between the entry of the last passenger and the end of boarding when the boarding is not yet finished, but point S is empty (i.e., already at least $(k+l)(N+1)+1$ seconds).

Also, notice that if there are N consecutive passengers in the queue without a first-row passenger among them, then at the moment the first of them turns, the entire queue will be waiting. Suppose this is not the case, and the tail of the queue continues moving. Then there will be $N+1$ passengers in the cabin (none of whom have yet turned), which is impossible. Therefore, a first-row passenger cannot be the second-to-last. If the second-to-last passenger is not from the second row, then when the last passenger takes a seat, the second-to-last still hasn't sat down (they might be turning if they are in the third row or on the way if farther). Similarly, we analyze the next one, who must be from the third row, and so on (having more than one passenger per row above the second leads to an extra second delay, as the last one waits while both take their seats). So, the last N passengers in the queue must be from rows $1, 2, \ldots N$. We decrease the capacity of the plane by these seats and repeat the reasoning from the beginning.

Thus, the boarding method involves passengers from rows N to 1, starting from one end, reaching the back of the cabin, simultaneously turning, and moving towards their rows. Then, the next block (from one of the remaining end rows) follows, and so on.

Now, it remains to count the number of permutations. It is clear that it is sufficient to do this for each row separately and then multiply. In the first row, a person cannot climb over others (which would slow down the entire queue), so there are $\binom{r+l}{r}$ possibilities. In the remaining rows, any person except the last one can slow down, and the last person has 2 choices of where to sit. Specifically, the i-th person ($i > 2$) from the end must sit at the $(n+1)(i-1)$-th second, so the number of possible seats is $\min(r, (n+1)(i-1)) + \min(l, (n+1)(i-1)) - (i-1)$;

The time complexity of the solution is $\mathcal{O}(l + r + \log n)$ per testcase.

Problem K. Kalevich's Key Knowledge

Note that if $b \notin \{0, 1, 4, 5, 6, 9\}$, then the answer is obviously -1;

It turns out that in all other cases such numbers exist. All the answers: 100, 1, 144, 1225, 16, 169, 2500, 2401, 2304, 25, 256, 289, 3600, 361, 324, 3025, 36, 3249, 400, 441, 4, 4225, 4096, 49, 52900, 5041, 5184, 5625, 576, 529, 6400, 6241, 64, 625, 676, 6889, 72900, 7921, 784, 7225, 7056, 729, 8100, 81, 8464, 81225, 8836, 8649, 900, 961, 9604, 9025, 9216, 9.

They can easily be found by brute force in $O(\texttt{answer})$ time.

Problem L. Latin Language Lesson

We will construct a graph where the vertices are numbers, and two numbers are connected by a directed edge if one can transition from the second number, written in Roman numerals, to the first number by appending one Roman numeral.

On such a graph, we will run retro-analysis — we start `dfs` or `bfs` from all vertices from which no move can be made, marking them as "losing" (i.e., those with no incoming edges). When considering a vertex, we will do the following: if it is a "losing" position, mark all its neighbors as "winning" positions and start from them/put them in the queue. If the vertex is winning, for each of its neighbors, increment a special counter by $+1$, and then start from/put in the queue only those whose counter equals their incoming degree (marking them as "losing");

It is not difficult to prove that in this way we will mark all vertices. For each winning vertex, we also store the number of its "losing predecessor"; Now, for the game, it is sufficient to ensure that during our move, we are in a winning vertex and make a move according to the information stored in it.

Problem M. Minor Maze Modifications

Let's consider a corner case. If the maze is initially connected, then the answer is $b(b-1)/2$, where b is the number of blocked cells.

We start a `dfs` from the initial and final vertices, coloring their components in different colors. Then we iterate over each blocked cell, and if it neighbors both components, we add it to the array `both`. Note that removing any cell from `both` makes the maze connected. We add $|\texttt{both}| * (b - |\texttt{both}|) + |\texttt{both}|(|\texttt{both}| - 1)/2$ to the answer, after which we mark the vertices from `both` (we have already considered all options with them);

Now we are left only with the options where the maze becomes connected by removing two cells (meaning both cells significantly contribute to creating the path). Note that in this case, one of the removed cells adjoins the initial component, and the other adjoins the final component.

Next, there are two possible cases: either these two cells are adjacent (such pairs are $O(rc)$, and we can enumerate them explicitly), or there is a connected component that both cells adjoin;

To count the options in the latter case, it is sufficient to find the sets of its unmarked neighbors for each component, those neighboring the initial and final components, and multiply their sizes together, after which we carefully process the pairs considered in the previous point. The time complexity is $O(rc)$.