Problem A. Count the Permutations

We will arrange the elements of the permutation in decreasing order. Consider the current number x. Notice that its occurrences in the array a cannot exceed three, as the number at position i can appear in the array a at positions i or $i-1$.

For convenience, consider the case $x = n$. If there are two positions $i < j$ such that $a_i = a_j = x$, then the condition $j = i + 1$ must hold; otherwise, the answer is zero. Thus, we know the position of the number x: it must be true that $p_{i+1} = x$.

Now notice that the array has split into two independent parts: the prefix $p[:i]$, corresponding to the prefix $a[:i-1]$, and the suffix $p[i+2:]$, corresponding to the suffix $a[i+2:]$, and the values in these two parts do not affect each other.

Now consider the case where there is exactly one position i such that $a_i = x$. Then one of two conditions must hold: either $i = 1$ or $i = n - 1$; otherwise, the element x would have appeared in the array a twice. Notice that in the case $i = 1$, we have $p_1 = x$, and in the case $i = n - 1$, we have $p_n = x$. In the case $n = 2$, we have two options for where x can be; otherwise, there is exactly one.

Now let's move on to the case for an arbitrary value of x. Notice that based on the analysis of the cases with two and one occurrence, the following holds: at any moment, we have subarrays of the array a that correspond to subarrays of the array p (formed by elements not exceeding x).

In the case of two occurrences, nothing changes; it is sufficient to check that these values are adjacent. In the case of one occurrence, we need to check that it is either the left boundary of the segment or the right boundary of the segment, or that both cases hold. To do this, we need to check that the element of the array a to the left or right is greater than x (this will mean that we have already considered it and it does not exist in the current segment).

Now let's turn our attention to the case where the value x does not appear in the array a. Notice that in this case, we must have a segment of length one in the array a, into which we have not yet placed any number; we can then choose any such segment and place x there. Thus, we need to maintain the count of free segments of length one, and we will need to multiply the answer by the current count and decrease it by one.

To maintain the count of segments of length one, we need to check in the case of two and one occurrence whether it is true that the part on the left or right has length one; then a new segment appears.

We obtain a solution in $\mathcal{O}(n)$.

Problem B. Forcefield

We will iterate through all pairs of circles and construct tangents. We will create an array of points that result from the intersections of the tangents and the circles. Next, we will construct the convex hull. After that, we need to account for the arcs — if two adjacent points in the hull belong to the same circle, we need to add the area under the arc to the area of the figure, and replace the length of the segment in the total length with the length of the arc.

The segments of the tangents can be obtained using the distance between the centers of the circles, their radii, and their difference, as well as through rotations and vector multiplications, since all angles and lengths can be easily determined.

Problem C. Partial Sum Operations

It is always possible to solve the problem using at most 3 operations. Let $s_0 = 0$, and define prefix sums $s_i = a_1 + a_2 + \cdots + a_i$.

After a second-type operation on $[l..r]$, the array becomes:

$$[a_1, \ldots, a_{l-1}, s_l - s_{l-1}, s_{l+1} - s_{l-1}, \ldots, s_r - s_{l-1}, a_{r+1}, \ldots, a_n]$$

After a third-type operation on $[l..r]$, the array becomes:

$$[a_1, \ldots, a_{l-1}, s_r - s_{l-1}, s_r - s_l, \ldots, s_r - s_{r-1}, a_{r+1}, \ldots, a_n]$$

Let s_x be the minimum and s_y the maximum value among the s_i. If $x > y$, apply the first-type operation (multiply all a_i by -1), which also swaps min and max in s_i.

Then apply:

1. A second-type operation on $[x+1..n]$
2. A third-type operation on $[1..y]$

After these, all array values become non-negative.

Now, let us solve the problem.

If all elements are 0 or 1, then the answer is 0.

If all elements are -1 or 0, the answer is 1 — we can apply a single operation of the first type.

Now we need to check whether we can solve with two operations. Suppose we want to apply a second-type operation. We show it's sufficient to apply it to the entire array. Let there be a subarray $[l..r]$ such that applying the second-type operation to it makes all elements non-negative. Then, all elements outside $[l..r]$ must already be non-negative. Thus, applying the second-type operation to the entire array results in at least as good an outcome. The same logic applies to the third-type operation.

Now we consider all combinations of two operations. If a first-type operation is used, then the second operation (of second or third type) must be applied to the entire array.

After considering rotations of array, we need to consider the case when we first apply suffix sum on some subsegment, and then apply prefix sum on whole array.

Use a divide-and-conquer strategy. Let $L \le l \le r \le R$, and define $M = \lfloor (L+R)/2 \rfloor$.

- If $r \le M$, recurse into $[L, M]$. Ensure all values from $M+1$ onward become non-negative.
- If $l > M$, recurse into $[M+1, R]$. Ensure values up to M become non-negative.
- If $l \le M < r$, proceed as follows.

Compute:

- y_1^r: value at $M+1$ after first operation
- y_2^r: minimum required at M after second operation (for values to the right to be non-negative)
- x_1^l: minimum required at $M+1$ after first operation (for values to the left to be non-negative)
- x_2^l, x_3^l: such that value at M after second operation equals $x_2^l \cdot y + x_3^l$, where y is value at $M+1$

Check if there exists a pair (l, r) such that:

$$y_1^r > x_1^l \quad \text{and} \quad x_2^l \cdot y_1^r + x_3^l > y_2^r$$

Sort all l, r by x_1^l and y_1^r descending. Use convex hull trick to maintain and query linear functions efficiently.

Time complexity: $\mathcal{O}(n \log^2 n)$

Problem D. Tricolored circle

If the initial or final string consists of identical characters, or if it has an even length and alternates between two characters, then it cannot be obtained from any other string, nor can any other string be

obtained from it. Therefore, in this case, the transformation is only possible if the initial and final strings are the same: then we can simply do nothing.

In all other cases, the transformation is always possible. Firstly, we will find in the first string a character s_i whose two neighbors are colored in different colors (if there is none, then this is the case from the previous paragraph). Starting from it and moving clockwise, we will replace each character s_j with any character that is different from s_{j-2} and s_{j+2} (indices are considered modulo n). At $j = i$, we know that the characters around s_j are different, and for the other j, by directly replacing the character s_{j-1} just before this, we ensured that it is different from s_{j+1}. After these n operations, we have achieved that each pair of characters standing one apart is different.

Now we will find in the second string a character t_k that has different neighbors (again, such a character exists). Starting from $j = k+1$, we will replace each character s_j with any character that is different from t_{j-2} and s_{j+2}. Again, we will understand that this can be done: during the first operation, all pairs of characters standing one apart are different, so any character can be changed, and during each subsequent operation, s_j can be changed since just before that s_{j-1} was replaced with some character that is not equal to s_{j+1}. With this second pass, we have ensured that each character s_j is not equal to t_{j-2}. Moreover, the last character we changed, namely s_k, is not equal to s_{k+2}.

In the final, third pass, we will start from the character s_{k+1} and replace all characters s_j with t_j. The first such operation on the character s_{k+1} can be performed since s_k and s_{k+2} are different. Each subsequent operation on the next character s_j is possible because, by construction from the previous paragraph, the character s_{j+1} is different from $t_{j-1} = s_{j-1}$ (since during the third pass we have not yet changed the character s_{j+1}). The only exception is the character s_k, because by the time we reach it, we have already changed the character s_{k+1} to t_{k+1}. Nevertheless, we will be able to change s_k to t_k, since $s_{k-1} = t_{k-1} \neq t_{k+1} = s_{k+1}$.

Thus, in $3n$ operations, we can definitely transform one string into another.

Problem E. Crazy dance

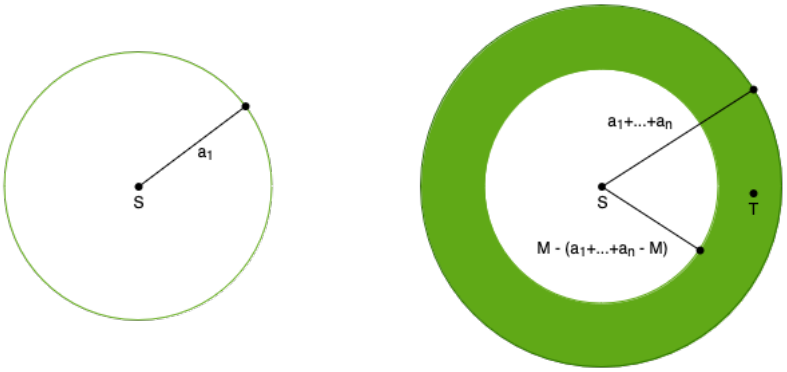
In the array c_i we will be counting, how many times Joker said digit i. Initially this array is filled with 0-s. Let's consider that each second every element of this array increases or stays the same and at lease one element increases. That's why the sum of all elements increases every second. That's why if Joker's dance is non-infinite than is can't last longer than the sum of all elements of the array b_i. Furthermore let's consider that because the sum of all elements of the array c_i increases we can binary-search the first moment when it becomes more or equal to the sum of all b_i elements. Than either at that moment the arrays are equal and then the answer is that moment or not and then Joker will dance forever.

Now we need to understand how to restore the array c_i by the time moment. Let's consider that the time moment is x. for numbers with the length less then the length of x we know that they all were said aloud. For numbers with the same length as the length of x, we need to go through the length of the prefix which coincide with the prefix of x, go through the digit which should be less than the corresponding digit in x, and than the number may end with any sequence of digits of the correct length.

Problem F. Decart the Grasshopper

Let us denote the starting point as point S and the ending point as T.

Consider the geometric locus of points (GLP) of the grasshopper's positions. After the first jump, the GLP will be a circle centered at the starting point with a radius equal to the first jump. After the second jump and onwards, the GLP becomes a ring, possibly with a zero inner radius.



Thus, the grasshopper will not be able to reach point T if it is either outside the outer radius of the ring formed after all jumps or inside the inner radius. Let M denote the maximum jump length from the set, and D the distance between S and T. Point T is reachable from S if and only if $M - (a_1 + a_2 + \cdots + a_n - M) \le D \le a_1 + a_2 + \cdots + a_n$.

Next, we need to determine the trajectory of the jumps. We will make several assumptions for the convenience of constructing the solution.

- Let the trajectory start and end at S, and we will add a jump of length D at the end of the jump set.
- $M' = \max(M, D)$
- We will remove the jump of length M' from the set and assume that it can be performed at any moment.

In this case, the trajectory of the jumps will be a triangle with sides $a_1 + a_2 + \cdots + a_k$, $a_{k+1} + \cdots + a_n$ and M'. Let k be the minimum k such that $a_1 + a_2 + \cdots + a_k + M' \ge a_{k+1} + \cdots + a_n$.

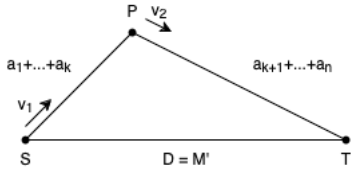
We will prove the existence of such a triangle (check the triangle inequality). Note that the previously verified condition guarantees that $M' \le a_1 + a_2 + \cdots + a_n$. We also specifically chose k so that $a_1 + a_2 + \cdots + a_k + M' \ge a_{k+1} + \cdots + a_n$, so it remains to prove that $a_1 + a_2 + \cdots + a_k \le a_{k+1} + \cdots + a_n + M'$.

Assume this is not the case, then it must be true that $a_1 + a_2 + \cdots + a_{k-1} + a_k > a_{k+1} + \cdots + a_n + M'$. We chose k such that it holds that $a_1 + a_2 + \cdots + a_{k-1} + M' < a_k + a_{k+1} + \cdots + a_n$. Subtracting the first inequality from the second, canceling common terms, we get $a_k - M' > M' - a_k$, which simplifies to $a_k > M'$, which is a contradiction. Thus, the existence of a triangle with sides $a_1 + a_2 + \cdots + a_k$, $a_{k+1} + \cdots + a_n$ and M' is proven.

Now, for the final construction of the trajectory, we will consider two cases: $M' = D$ and $M' \ne D$.

$M' = D$

We will choose the segment from S to T as the base of the triangle. Next, we will find the third point of the triangle with sides $a_1 + a_2 + \cdots + a_k$, $a_{k+1} + \cdots + a_n$ and $M' = D$. This can be done using the formula for the coordinates of the intersection point of two circles. From the two options for the points, we will choose any. Let this be point P.



Then we will compute the unit vectors $\vec{v_1} = \vec{SP}/|SP|$ and $\vec{v_2} = \vec{PT}/|PT|$. Thus, to jump from point S to point P, we will compute each subsequent point as $P_{next} = P_{prev} + a_i \cdot \vec{v_1}$, and then, to jump from point P to point T — using the formula $P_{next} = P_{prev} + a_i \cdot \vec{v_2}$.

$M' \neq D$

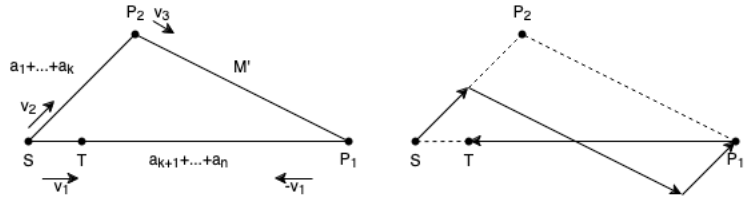
The second case is a bit more complicated. In this case, T is not a vertex of the triangle, but it must lie on one of its sides for everything to work out. Since we added the jump of length D at the end, point T must lie on the side of the triangle with length $a_{k+1} + \cdots + a_n$.

Also, recall the assumption that we removed the jump of length M' from the set and considered it could be performed at any moment. Since the order is strictly defined, when calculating $a_1 + a_2 + \cdots + a_k$, we will not consider M', if the first occurrence of the maximum is reached at index $i \leq k$, then we will not have confusion with indices in the further solution.

Let $\vec{v_1} = \vec{ST}/|ST|$. We will compute the vertex of the triangle P_1 as $P_1 = S + (a_{k+1} + \cdots + a_n) \cdot \vec{v_1}$. Then we will compute P_2, as the third point of the triangle, similarly to the previous case. Next, we need to compute two unit vectors $\vec{v_2}$ and $\vec{v_3}$, aligned with $\vec{SP_2}$ and $\vec{P_2P_1}$ respectively.

Then the solution looks similar to the previous case, we compute the next point from the previous one by adding the vector:

- $a_i \cdot \vec{v_3}$, if $a_i = M'$ and this is the first occurrence of the maximum.
- $a_i \cdot \vec{v_2}$, if $i \leq k$
- $a_i \cdot -\vec{v_1}$, if $i > k$



It is also necessary to carefully consider the case where $S = T$, in which case we can choose any unit vector as $\vec{v_1}$.

Problem G. Heat Exchangers

First, we will use a disjoint set system and merge the intersecting triangles, or rather, their entries. After that, it is necessary to apply numerical integration. We need to conduct a vertical line through a very small interval, observe where the triangles intersect, and then, through events and something like a one-dimensional sweeping line, find out how much heat is released in this small section. After that, we need to sum all the obtained values.

Problem H. Induced Subgraphs

Let us quickly quickly explain the problem statement: We are given a tree. Count the number of ways to assign labels $1, 2, \ldots, n$ to each node in the tree such that the sets of vertices $\{1, 2, \ldots, k\}, \{2, 3, \ldots, k+1\}, \ldots, \{n-k+1, \ldots, n\}$ are each connected components. This editorial will call these sets 'target' sets.

Small k versus large k

A good starting point is to notice that the problem can change dramatically depending on how large the value of k is. The main difference between small and large values of k is that when k is large enough, there will exist a group of labels that are common to every target set. For example, with $n = 10$ and $k = 7$:

```
1 2 3 4 5 6 7
2 3 4 5 6 7 8
3 4 5 6 7 8 9
4 5 6 7 8 9 10
```

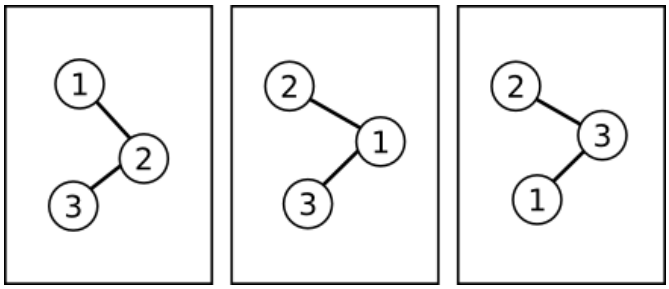
The labels $\{4, 5, 6, 7\}$ exist in each of the target sets. Which does not happen with small values of k, for example $n = 6$ and $k = 3$:

```
1 2 3
2 3 4
3 4 5
4 5 6
```

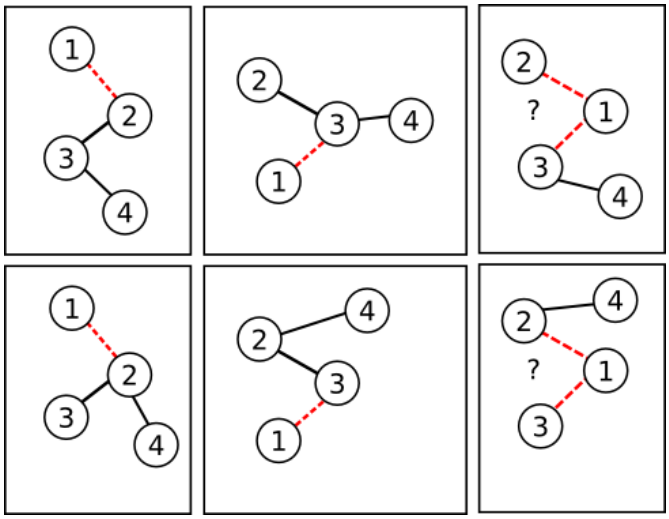
There is no vertex that is present in all of the sets. The requirement for us to consider a value of k large is that the set of the k smallest values intersects with the set of the k largest values. In other words: $2k > n$. This small difference makes a case much harder than the other. Let us begin with the small case, which is the simplest.

$2k \leq n$

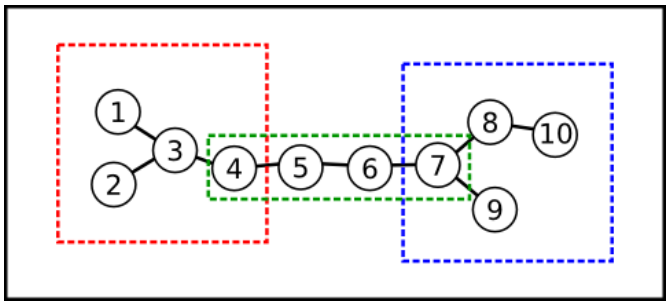
In order to understand this case, let us think about what is needed to transition from the first target set to the next one making sure they stay connected. Imagine that we already found out the connected component of the first target set, the nodes with the k smallest labels. Since we are dealing with a tree and its subtrees, these labels will form a tree. Like in the following two examples for $k = 3$:



The tree can so far have any shape, but with k connected elements. It is interesting what happens when we transition to the next target set. From $\{1, 2, \ldots, k\}$ to $\{2, \ldots, k + 1\}$. What it means is that the new tree will have $\{1\}$ removed and $\{k + 1\}$ added, *but it still must be connected.* This has a consequence, in two of the examples above, there are only two super trees of 4 elements that allows the condition. The third example is simply not correct, if we add a $\{4\}$ and remove $\{1\}$ the tree will not remain connected.



Try making more transitions. Also consider that at the opposite part of the tree, the labels with larger values, the structure works the same when moving on from the largest k labels and removing $\{n\}$ and adding $\{n-k\}\ldots$ At the end you will find that the tree must follow a specific structure. For example, when $k=4$ and $n=10$:



- In the red square, we have a formed by the k smallest labels. They must follow a proper hierarchy structure. In every branch, the smallest label must be a label and the parent of each label must always be greater, up until k (4 in this case) which must be the root of the subtree. The reason for the hierarchy is that when transitioning from each target set that contains two of the smallest labels, the smallest label has to be removed, thus this label cannot act as a bridge between any other two larger labels.
- Inside the green square, a chain of all the nodes from k to $n-k+1$. This must be a chain, because of how transitions work. When transitioning from $\{3,4,5,6\}$ to $\{4,5,6,7\}$, 3 must be connected to 4. When transitioning from $\{4,5,6,7\}$ to $\{5,6,7,8\}$ then 4 must be connected to 5 and so and so...
- Inside the blue square, also a subtree. This time it is the k largest labels. This subtree must also have a proper hierarchy, this time each parent must be smaller than each of its children.

The solution for this small case is already accessible. Given the tree, we can pick the starting and ending node of the chain. We can tell the chain must contain $n-2(k-1)$ elements. Thus the distance between the two picked points must be $n-2(k-1)-1$. These end points would get labels k and $n-k+1$, respectively. Then we also need to verify that the subtrees rooted at each of the end points contain exactly k elements. If all checks up, the result is equal to the number of ways to assign labels to the first sub-tree multiplied by the number of ways to assign labels to the second sub-tree. This number of ways can be calculated using a dynamic programming approach.

How to assign labels to a sub-tree following a hierarchy

Let us quickly sum up the dynamic programming approach. We are given a subtree rooted at a given vertex. We also need to know its parent so that we understand what direction not to go when calculating the children (For different end points, the parents of the subtree roots change).

The base case is when the subtree contains only the root. Then we can only assign one label to the root (Either the lowest or the highest, depending on which of the subtrees we want to count, but for the sub-problem it does not matter).

The remaining case is when the root has children. We can assume that we already know the ways to assign labels to each children following the hierarchy. For example, when there are two children subtrees one with n_1 nodes in total and the other with n_2 nodes in total, we can assume that currently the labels that were assigned to each subtree are in the form $\{1, 2, \ldots, n_1\}$ and $\{1, 2, \ldots, n_2\}$. We want to assign labels $\{1, 2, \ldots, n_1 + n_2\}$ to the two subtrees at the same time. This is equal to, out of a total $n_1 + n_2$ labels, picking n_1 labels to go to the first subtree. After picking which labels go to each subtree, we replace the original labels $\{1, \ldots, n_1\}$ and $\{1, \ldots, n_2\}$ using the same order to make sure the hierarchies are not lost, so there is only one possible order for the labels. Then we put a label to the root, it must be larger than all the other labels, so there is only one way to assign a label to the root. This means that in total there will be: That is $\binom{n_1+n_2}{n_1} \times \text{ways(first)} \times \text{ways(second)}$ ways to assign labels to the larger subtree still following the hierarchy. This approach can be adapted to when there are more than 2 children: First combine the first two subtrees into one hierarchy, then combine this new hierarchy with the third subtree and so and so.

$2k > n$

The harder case is harder because of the labels that are common to each of the target sets. Each of the target sets must be a connected component. It is possible to show that this common intersection must also be a connected component. This intersection will have the labels that are not the largest and not the smallest. We will call this group of labels 'middle'.

Once again we have to see what happens when we transition from a target subset to the next one. This time considering that the middle labels are always connected and remain so. Let us go back to the case with $n = 10$, $k = 7$.

- The first target set is $\{1, 2, 3, 4, 5, 6, 7\}$. $\{4, 5, 6, 7\}$ must form a connected component. Thus we only need to add $\{1\}, \{2\}, \{3\}$ to whatever structure the connected component has. We can add $\{1, 2, 3\}$ as a single subtree. Or in separate partitions. We can connect these subtrees to any of $\{4, 5, 6, 7\}$ and the structure $\{1, 2, 3, 4, 5, 6, 7\}$ will be connected.
- But then we transition from $\{1, 2, 3, 4, 5, 6, 7\}$ to $\{2, 3, 4, 5, 6, 7, 8\}$ note that we deleted the smallest label, and added a new one. We once again have a process in which the smallest label is removed when performing transitions. In short, and reusing the knowledge from the previous analysis, all the subtrees of $\{1, 2, 3\}$ that we added to $\{4, 5, 6, 7\}$ must follow a hierarchy.
- The same will happen for $\{8, 9, 10\}$, all of their subtrees must follow the hierarchy.
- $\{1, 2, 3, 4, 5, 6, 7\}$ must be connected without the help of any of the vertices in $\{8, 9, 10\}$, this means that the subtrees of $\{1, 2, 3\}$ will not have vertices from $\{8, 9, 10\}$. Similarly, the subtrees of $\{8, 9, 10\}$ cannot have vertices from $\{1, 2, 3\}$.
- Note that $\{4, 5, 6, 7\}$ can have any shape as long as it is a connected tree. It is not necessarily a chain. And the various subtrees do not necessarily have to connect to 4 or 7.

Problem I. Magic Spell

We will create a directed graph with $n+1$ vertices, numbered from 1 to $n+1$, where each edge will have a color — a number from 1 to k. Consider a subarray $[l_q : r_q]$ of the array a. Note that we are interested in several cases for using it in the answer:

1. p is a subarray of $a[l_q; r_q]$ — the answer is 1.
2. $a[l_q : r_q] = p[i : j]$ — edge $i \to j+1$ has color k.
3. $a[l_q : l_q + x] = p[n - x : n]$ — edge $n - x \to n + 1$ has color k.
4. $a[r_q - x : r_q] = p[1 : x + 1]$ — edge $1 \to x + 2$ has color k.

To determine which of the cases hold, we will precompute the arrays left$[i]$ and right$[i]$, which store the length of the maximum subarray from i to the left and right respectively, that is a subarray of p. To do this efficiently, we will precompute the inverse permutation for p, after which we will compute the arrays left and right in a linear pass. Using these arrays and the inverse permutation, we can determine cases 2-4.

To efficiently determine the first case, we will find all occurrences of p in the array a in advance; this can be done naively by looping through each occurrence of p_1 in the array a, which will work in linear time overall since p is a permutation. After that, for each index i, we will precompute the nearest occurrence to the right, and when considering the segment $[l_q; r_q]$, we will find the nearest occurrence to the right of l_q and check that it fits within the segment.

Now, if the answer is not equal to one, the problem can be reformulated as follows: we need to find the shortest path in terms of the number of edges from vertex 1 to vertex $n + 1$, where all edge colors are different. Note that all edges in our graph go from a smaller vertex to a larger one, so it is acyclic. We also note that the number of edges of each color is no more than two, and if there are two, one edge goes from vertex 1, and the other edge goes to vertex $n + 1$.

Without loss of generality regarding colors, the problem of finding the shortest path in an acyclic graph can be solved using dynamic programming (dp$[v]$ — the length of the shortest path from vertex v to vertex $n + 1$) in linear time. To account for the possibility of two edges of the same color, we will count two shortest paths from each vertex v, which have different colors for the last used edge (in other words, edges to vertex $n + 1$). To do this, we will maintain the arrays dp$[v]$ and dp$_2[v]$, as well as the array edgeInd$[v]$, which store the length of the minimum shortest path, the second shortest path differing by the index of the last edge, and the index of the last edge used in the shortest path, respectively. This can be computed in a linear pass from the end, similar to finding the second minimum in an array.

Now we will find the answer as follows: we will iterate over the first edge on the path, let this edge be $1 \to v$ with color c. If edgeInd$[v] \neq c$, we can update the answer via $1 + $ dp$[v]$, otherwise via $1 + $ dp$_2[v]$. We will find the minimum over all first edges, after which we will reconstruct the answer in the standard way, saving the ancestor arrays for dp$[v]$ and dp$_2[v]$. We have obtained a solution in linear time.

Problem J. Least Common Multiple

This problem can be solved by factorizing number n and then applying inclusion-exclusion technique. The simplest way of factorization in $\mathcal{O}(\sqrt{n})$ time is too slow for this problem. However, we will note that for the sake of finding the number of divisors, there is a simple $\mathcal{O}(\sqrt[3]{n})$ algorithm which does not require any complex techniques such as ρ-Pollard algorithm. Simply perform a trial division by all divisors up to $\sqrt[3]{n}$; after that, the remaining number r can only have prime divisors which are greater than $\sqrt[3]{n}$. In this case, since this number r is less or equal than n, this number can only be of these of these forms: 1, p, p^2 and pq. These cases are easily distinguishable: $r = 1$ is trivial, $r = p$ is checked with a Miller-Rabin check, $r = p^2$ can be done via checking that r is a perfect square (find $\lfloor\sqrt{r}\rfloor$ with Newton method or binary search, then compare r with $\lfloor\sqrt{r}\rfloor^2$), $r = pq$ is the remaining case.

Problem K. Tennis

We will solve the problem using dynamic programming. Specifically, we will look for the answer from the end. That is, we will compute the shortest path from (a, b) to $(0, 0)$, rather than the other way around. To recalculate, we can use the formula $dp_{a,b} = \min(dp_{a-1,b}, dp_{a,b-1}) + \gcd(a, b)$. However, this solution currently works in $O(a \cdot b)$. Let's try to speed it up. Without loss of generality, assume $a \leq b$. Let $a < b$ and b be a prime number. Then the weight of the shortest path from $(0, 0)$ to (a, b) is equal to $a + b$. It is clear that it cannot be less, because after each move the weight increases by at least 1. To achieve this, we can follow the next algorithm:

- $(0, 0)$
- $(1, 0)$
- $(1, x)$, where x increases from 1 to b. At this point, after each change, the GCD is equal to 1, since one of the numbers is equal to 1.
- (y, b), where y increases from 2 to a. Here, after each change, the GCD is equal to 1, since $y < b$ and b is prime.

Also, if $a = 0$, there is a unique path from $(0, 0)$ to $(0, b)$, and its weight is equal to $\sum_{x=1}^{b} x = \frac{(b+1) \cdot b}{2}$.

Instead of standard dynamic programming, we will implement a memoization approach, which is also called lazy dynamic programming. If we can prune the search for such a and b and immediately return the answer in $O(1)$, the solution will work in approximately $O(a \cdot \log(b))$. This follows from the fact that prime numbers occur on average once in the logarithm of the numbers. In practice, there is not a significant gap between consecutive prime numbers up to 10^9.

We still need to learn how to prune the search based on a. Analogously, we would like to do this when a becomes a prime number. However, there may be a situation where a is prime, $a < b$, but the weight of the shortest path from $(0, 0)$ to (a, b) is not equal to $a + b$. Therefore, we need to add an additional condition. Let p be the largest prime number such that $p \leq b$. Then, if additionally $a < p$ and there are no numbers divisible by a in the interval $[p, b]$, the weight of the shortest path from $(0, 0)$ to (a, b) is exactly equal to $a + b$. To achieve this, we can follow the next algorithm:

- $(0, 0)$
- $(1, 0)$
- $(1, x)$, where $x \in [1, p]$
- (y, p), where $y \in [2, a]$
- (a, z), where $z \in [p + 1, b]$. Here, after each change, the GCD is equal to 1, since a is prime, $a < z$, and z is not divisible by a.

Thus, we do not always prune when a first becomes prime, but we will definitely prune at one of the nearest primes. In practice, such a search with pruning will visit at most $\sim 10^5$ different states, with constraints up to 10^9.

Problem L. Minimum Prefix

Let's assume that K is fixed. We can ignore all strings shorter than k (they're unique, anyway). Let's imagine that we can construct a bipartite graph, where there's a vertex for each string in the left part and a vertex for each cyclic substring of length k in the right part. If we add edges from each string to its substrings in this graph, we just need to check if this graph has a matching that covers all vertices in the left part. There're at most N vertices in the left part and at most S edges in this graph (where S is the total length of all strings).

We can binary search over k and use a standard algorithm to find a matching. The time complexity of this solution is $O(N \cdot S \cdot \log S)$.

The only question is how to build such a graph efficiently. One can use hashes or some suffix data structure to do it.

Another idea is we can solve this problem bit faster than $O(N \cdot S \cdot \log S)$. If a string has at least N different substrings of length k, we can ignore the string when considering bipartite matching. This reduces the number of edges in the graph to $O(N^2)$. Thus the overall time complexity becomes $O((N^3 + S) \cdot \log S)$.

The correctness of this algorithm is confirmed by Hall's Theorem.

Problem M. Four Villages

Four vertices that are equidistant from each other are four vertices that are equidistant from some center.

We will use a divide-and-conquer approach with centroids. We will consider all quadruples such that the centroid lies on some path between these four vertices and count them. We will calculate $f[v][k]$ — the number of groups of k vertices that are at the same distance down from v. Next, we will iterate over the center; if it is not the centroid, then there are three vertices below the center and one vertex in another subtree of the centroid. We need to precompute how many vertices are in the other subtrees of the centroid at each depth and $f[v][3]$. For the center coinciding with the centroid, we simply take $f[v][4]$.

To calculate $f[v][k]$, for each vertex we compute $g[d][k]$ — the number of groups of k vertices at depth d from v. This can be recalculated for a vertex through its children by merging a smaller structure into a larger one, since d is the depth, g will be computed for all vertices in linear time.

In total, $O(n \log n)$