Problem A. Alien Homophones

First, let's identify connected components by identical sounds, now every sound inside one component is the same.

Now, we need to calculate the max jump for each position in the text. Build a trie on reversed strings. We will calculate the suffix links on the trie as in Aho-Corasick, as well as terminal suffix links from each vertex to the nearest terminal (a terminal vertex of the trie is a vertex where some sound ends). Now we will go through the text from right to left and move through our trie. By following the link to closest terminal suffix link, we get a longest pattern occuring in this position. The precomputation of suffix links works in $O(26 \cdot S)$, and the pass through the text takes a total of $O(|t|)$.

We need to figure out how to process our queries fast. For each position in the text, we know the max jump, so let's build a sparse table on these jumps, which will store for each power of two 2^k the hash of the 2^k sounds we pronounce during these 2^k jumps and the position we will end up at after these jumps. We will spend $O(|t| \log |t|)$ on precomputation. Now, in a query, we can find the position after which the max jump skips the boundary for each substring in $O(\log |t|)$, as well as the polynomial hash of the max sounds we pronounced on the way to that position.

Now we need to solve the problem for the last part, which is shorter than the corresponding max jump. Notice that it means that our remaining substring is a prefix of some sound. Therefore, we need to be able to compute hash for every prefix of some sound.

To do so, we will calculate DP for every prefix of the sounds, in order of increasing length. Let us calculate max jumps for every position inside one sound, without considering taking the whole sound. Now, when considering some prefix, we first want to skip max jumps while possible as when processing queries. Then, we are left with some suffix, but it is strictly smaller and also is a prefix of some sound, so we use calculated answer in DP.

The total complexity is $O(26 \cdot S + |t| \cdot \log |t| + q \cdot \log |t|)$.

Problem B. Tickets for the Train

The formal statement: there is a train consisting of m cars, the ticket fare and the capacity of the i-th car equal c_i and p_i, and for each group of k (or less, if there is residue) students, an additional ticket for a teacher in the same car. There are n students; we need to supply them (and necessary teachers) in the least expensive way.

Let us proceed through several solutions in increasing order of complexity, resulting in a solution that passes all tests.

1. $\mathcal{O}(mn^2)$: just a regular DP for a knapsack problem with m types of items and n total weight of the required items. The only tricky part is to take into consideration extra tickets for teachers.

2. $\mathcal{O}(m^2k^2)$: firstly, sort the cars in increasing order of the cost of one ticket. Let us call a "full block" the set of k students and one teacher in the same car, and less students with a teacher will be called a "residual block"; then the solution for each car can be fully described with two numbers: the number of full blocks and the size of the residue. Note that the optimal answer fills the full blocks in the order from the cheaper ones to the more expensive. For residual blocks, the DP can be calculated in $\mathcal{O}(m^2k^2)$ time; then we iterate the position of the last full block and update the answer with the previously calculated DP in $\mathcal{O}(mk)$.

3. $\mathcal{O}(mk^3)$: there is an approach, the proof of which is left as an exercise for the reader, that the following approach produces a correct answer: let us call the *size* of a block the number of students in it (then the size is from 1 to k, and the total number of tickets for a block is always one more than the size); then let us divide all cars into full and residual blocks greedily. Then let us sort the blocks by their *specific cost*, a.k.a. cost per student, defined as $\frac{(s+1)\cdot \text{cost}}{s}$ where s is the size of the blocks. Let us take the blocks greedily until we get n students. Then we don't necessarily get an

optimal answer; however, now, if for each $s \in \{1, \ldots, k\}$ we throw away k last blocks of this size, then there is a way to redistribute these $\mathcal{O}(k^3)$ students to get the optimal answer.

The remaining $\mathcal{O}(k^3)$ need to be added with a DP on $\mathcal{O}(m)$ blocks. Obviously, if one of the blocks is only used partially, then we never need to use more expensive blocks (otherwise there is an improving relaxation of this answer). Therefore, we can sort the remaining cars by the cost of a ticket, and DP in $\mathcal{O}(mk^3)$ tries to take several first blocks completely and the last one partially.

4. $\mathcal{O}(m \log m + k^7)$: the beginning of the solution is the same as in the previous one, but, to solve the problem for the last $\mathcal{O}(k^3)$ students, we will iterate through the possible values of s (the size of the block); for each separate size, the optimal answer can be got via considering the tickets greedily in the order from cheaper to more expensive ones. This will allow us to update the answer in $\mathcal{O}(k^6)$ for each value of s.

Problem C. Spanning Trees

Let's understand how a spanning tree can be structured. Consider segments of vertices on a circle that are sequentially connected by edges. Each such segment must be connected to a central vertex by exactly one edge.

We will solve the problem using the]]Divide and Conquer" method. Consider a segment of vertices on the circle with indices $[l, r]$ $(1 \le l \le r \le n)$. We will examine the segments of vertices connected by edges within $[l, r]$. All segments, except for the leftmost and rightmost ones, are definitely connected to the central vertex by an edge. Note that in order to merge two segments in "Divide and Conquer" it is sufficient for a segment to know whether its leftmost segment is connected to the central vertex and whether its rightmost segment is connected to the central vertex.

We will store a vector for each of the 4 types of segments, where the i-th element represents the number of ways to choose edges in the segment to achieve that type, with exactly i white edges among the chosen edges. After that, to obtain similar vectors for merging two segments, we need to compute $O(1)$ convolutions. For this, we can use the Fast Fourier Transform algorithm. In the end, we also need to account for the edge connecting vertices n and 1. The final asymptotic complexity is $O(n \cdot \log^2(n))$. To find out which convolutions need to be computed for the recount, you can refer to the author's solution.

Problem D. Bounded Arithmetic Expression

The first step towards finding the solution is noticing that the given formula can be interpreted as a simple undirected graph on n vertices. We want to assign values to the vertices so that the sum of products of neighboring vertices is maximized.

In the analysis below, for a vertex v we denote by $\text{val}(v)$ the value assigned to v, and by $\text{sum}(v)$ the sum of values assigned to neighbors of v. We will also sometimes use v itself in place of $\text{val}(v)$ for brevity.

Assume we found some optimal solution, and let S be the set of vertices for which we did not assign an extreme value (that is, neither their lower nor upper bound). Suppose there exist two vertices $u, v \in S$ that are not connected. Assume that $\text{sum}(u) \ge \text{sum}(v)$. Let

$$x = \min\big(\text{upper}(u) - \text{val}(u),\ \text{val}(v) - \text{lower}(v)\big).$$

We can increase $\text{val}(u)$ by x and decrease $\text{val}(v)$ by x without violating any constraints, while increasing our result by $(\text{sum}(u) - \text{sum}(v)) \cdot x$.

Since the size of S is reduced in such a step, we can prove by induction that in some optimal solution, all pairs of vertices in S are neighbors. That is, S forms a **clique** in the formula graph.

Assume $S = \{x_1, x_2, \ldots, x_k\}$. Suppose we have already assigned values to all vertices outside S — this can be done in $2^{n-|S|}$ ways. Assume that the sum of vertices in S is equal to M.

Let a_i be the sum of values assigned to the neighbors of x_i, excluding those from S. Note that the sum of all neighbors of x_i will be equal to $a_i + M - x_i$. Our total result will therefore be equal to

$$x_1\left(a_1 + \frac{M}{2} - \frac{x_1}{2}\right) + x_2\left(a_2 + \frac{M}{2} - \frac{x_2}{2}\right) + \cdots + x_k\left(a_k + \frac{M}{2} - \frac{x_k}{2}\right).$$

We have to divide $(M - x_i)$ by 2, since the values of edges inside S are counted twice in this sum.

Note that the derivative of the expression $f(x) = x(a - x/2)$ is equal to $a - x$. Using a similar argument as before, we can prove that in an optimal assignment of x_i, the derivatives of all these expressions are equal. Therefore, there exists some constant c such that

$$x_i = a_i + \frac{M}{2} + c.$$

We can compute c as

$$c = \frac{M - \left(a_1 + a_2 + \cdots + a_k + \frac{kM}{2}\right)}{k}.$$

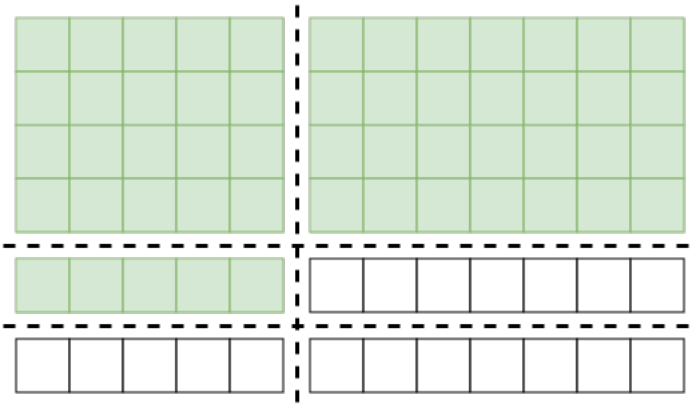
It remains to check whether such x_i fit the bounds from the input.

Problem E. Cactus

Formal problem statement: You need to color a cactus with three colors while minimizing the number of vertices of the third color, or report that it is impossible to color the given cactus with three colors. It is not difficult to notice that a cactus can always be colored with three colors. To find the minimum required number of vertices of the third color, we will use dynamic programming. We will construct a depth-first search (DFS) traversal tree of the cactus. Now, for each subtree, we will calculate the minimum number of vertices of the third color needed to color the subtree with three colors, ensuring that for all edges entirely within the subtree, the colors of the endpoints are different. It is important to note that since the graph is a cactus, at most two edges lead from each subtree upwards: one from the root of the subtree to its parent, and at most one edge not in the DFS tree. Therefore, we only need to add the colors of the two lower vertices of these edges to the state. As a result, we obtain a linear number of states, and the computation for a vertex works in the time proportional to the number of its children, which sums up to linear across all vertices. This leads to a linear solution with a sufficiently large constant multiplier.

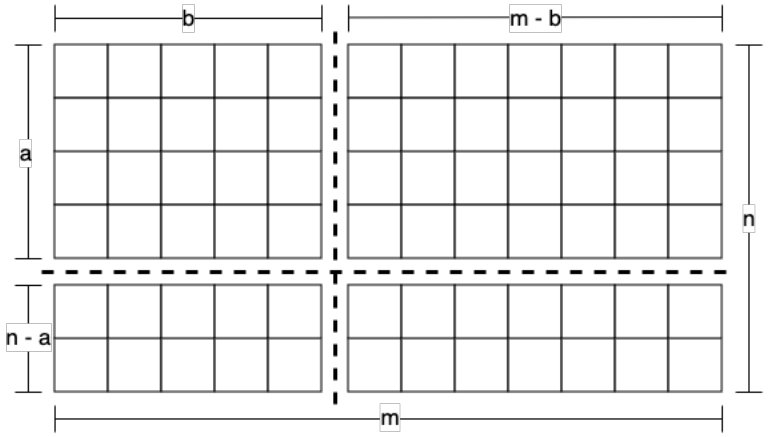
Problem F. Lazy Cuts

Note that to obtain an area s, no more than three cuts are needed. Using one cut, we can cut off the first k rows, and then, using two more cuts, obtain any number of cells from the next row.



Let's analyze the cases when fewer than three cuts are sufficient, namely zero, one, or two. Zero cuts can occur if and only if: $s = n \cdot m$. One cut is enough if s is a multiple of n or s is a multiple of m.

With two cuts, it makes sense to make one cut vertically and the second horizontally. Let these cuts divide the grid after the first a rows and the first b columns ($0 < a < n$ and $0 < b < m$), as shown in the figure below.



There can be three ways to obtain s cells:

1. Take the upper left piece if $s = a \cdot b$.
2. Take everything except the upper left piece if $s = n \cdot m - a \cdot b$.
3. Take the upper left and lower right pieces if $s = a \cdot b + (n - a) \cdot (m - b)$.

All methods of obtaining can be checked in $O(n)$ by iterating through the first cut and verifying in $O(1)$ the existence of a suitable second cut by solving an equation with one unknown.

Problem G. Checkered Pattern

The problem can be solved using a dynamic programming method based on a broken profile. The state is defined as the partitioning of black cells in the current profile into connected components. When transitioning, you need to choose the color of the current cell from two options: white and black. The cost of the transition is 0 if the cell already has that color, and 1 if the original color of the cell is the opposite. The state can be represented by an array of integers of length m, where the i-th number is -1 if the corresponding cell is painted white, and some non-negative number otherwise. Moreover, if two black cells are in the same connected component, they correspond to the same number; otherwise, they correspond to different numbers.

When transitioning where the current cell is painted white, it is necessary to check that each connected component still has a representative in the next broken profile. This can only be violated if the neighboring cell above is black and is the only representative of its connected component. When transitioning where the current cell is painted black, it is necessary to check that no cycle has formed among the black cells. This can only occur if the neighboring cells to the left and above are black and belong to the same connected component.

If the transition is valid, a new array must be constructed that corresponds to the current partitioning of cells in the broken profile into connected components. If, while moving from left to right across the profile, you always choose the smallest unused non-negative integer as the number for the next component, the maximum number of distinct reachable states for $m = 10$ turns out to be approximately $7,500$. The final time complexity is $O(n \cdot m \cdot S \cdot m)$, where $S \approx 7,500$ is the number of distinct states.

Problem H. Road Lighting

Consider the following greedy algorithm for finding the maximum matching in a tree:

- Suspend the tree from vertex 1 and start a `dfs` from it.
- In the `dfs` function, first recursively call `dfs` for each of its children u_i, which will find the maximum matching in the subtree of vertex u_i. Let the size of this matching be denoted as m_{u_i}.
- Let c_v be the number of children of vertex v that are not occupied by the matching.
- If $c_v = 0$, then we do not add new edges to the matching. Thus, the size of the matching in the subtree of vertex v is $m_v = \sum m_{u_i}$.
- If $c_v > 0$, then we can add one additional edge to the matching. We will add the edge between v and any of its children that is not occupied by the matching. In this case, $m_v = 1 + \sum m_{u_i}$.

The correctness of this algorithm can be proven by induction:

- We will prove that `dfs`(v) finds the maximum matching in the subtree of vertex v. Moreover, if there exists a maximum matching in which vertex v is not occupied, it will find exactly such a matching.
- Base case: if v is a leaf, then the correctness of the algorithm is obvious.
- Inductive step: if v is not a leaf, it is easy to see that $m_v \leq 1 + \sum m_{u_i}$. If all vertices u_i are saturated by the matching, then $m_v = \sum m_{u_i}$, since if we take any edge from v to a child, the size of the maximum matching in the subtree of that child will decrease by 1, otherwise the property that among maximum matchings the algorithm finds one in which the upper vertex is not occupied would be violated. In this case, the maximum matching has already been found, and vertex v is not saturated by it, so there is no need to do anything.

  Otherwise, we can add an edge to any unsaturated child and increase the maximum matching by 1. In this case, vertex v must be saturated, since without vertex v, the size of the maximum matching is $\sum m_{u_i}$. Thus, this algorithm is indeed correct.

The problem required to "include"and "exclude"edges in the tree and find the size of the maximum matching in the connected component of connectivity formed by the included edges. We will optimize the algorithm described above. First, we will run it explicitly and find m_v and c_v for all vertices v, as defined above. After the queries, the tree may break into independent connected components. In each such connected component, we will identify the highest (closest to the root) vertex. We will assume that we have run the greedy algorithm in each connected component from each identified vertex, which will calculate the values of m_v and c_v. We will then maintain these values efficiently.

Let's understand how m_v and c_v change with each query:

1. If the edge (v, u) is "excluded without loss of generality, let v be an ancestor of vertex u. Then nothing has changed in the subtree of vertex u. We subtract m_u from all m_p, where p is an ancestor of vertex v in the new connected component of vertex v. Now consider two cases:
   - If the greedy algorithm can construct a matching that does not include the edge (v, u), then in the new component of vertex v, no values have changed. To check whether such a matching can be constructed, it is sufficient to check that $c_u > 0$ or $c_v \neq 1$.
   - Otherwise, the matching in the component of vertex v has decreased by 1. That is, we need to subtract another 1 from all m_p. In this case, we also need to update the values of c_p. Since $c_v = 1 \implies u$ was the only unsaturated child before, we need to subtract 1 from c_v and add 1 to m_v. Now let's look at the ancestor x of vertex v (if it does not exist or is not in the

same component as vertex v, then there is nothing more to do). Previously, we counted in the ancestor that vertex v was saturated, but now it is not, so we need to increase c_x by 1. If after this $c_x = 1$, it means that previously vertex x was not connected to a child. In this case, we need to increase m_x by 1 and perform the same algorithm for the ancestor of vertex x, since it results in exactly the same case: the edge to its unsaturated child can no longer be used, so this is analogous to removing such an edge.

2. If the edge (v, u) is "included let again v be an ancestor of vertex u. Again, nothing has changed in the subtree of vertex u. Now we need to add m_u to all m_p. Now we note that the previously considered case for the ancestor x of vertex v is completely analogous to adding a new child to vertex x. Therefore, we need to apply the same algorithm here, but with the first step omitted.

3. To find the maximum matching in the connected component of vertex v, it is sufficient to ascend to the highest vertex r in this component and output m_r.

To implement this method in the general case, we will store the values of m_v and c_v in a Heavy-light decomposition. Then, let's consider what needs to be done when "excluding"the edge (v, u). First, we need to subtract m_u from all m_p on some vertical path, which is a trivial mass query. Now we need to find the longest sequence $1, 0, 1, 0, \ldots$ from vertex v upwards through the values of c_i. On this vertical path, we need to replace the values of c_i with $0, 1, 0, 1, \ldots$. We also need to subtract 1 from every second value of m_i on this path.

In the case of "including"an edge, we need to find the maximum sequence of $0, 1, 0, 1, \ldots$ through the values of c and perform similar transformations as in the previous case.

To search for the longest sequence of the form $1, 0, 1, 0, \ldots$ in HLD, we can, for example, store the minimum and maximum values for each parity. This can be implemented with a regular descent through HLD. To then replace these values with $0, 1, 0, 1, \ldots$, it is sufficient to add 1 to all even indices and subtract from all odd ones. This is also supported in a regular segment tree.

Thus, such a solution can be implemented in $\mathcal{O}(n \log^2 n)$, which is quite efficient.

In this problem, there is also a solution that runs in $\mathcal{O}(n \log n)$ using a link-cut tree. Moreover, it allows solving the problem in an even more general case, where the original tree is not given and edges can be removed and added arbitrarily so that no cycles are formed. This solution is left to the reader as an exercise.

Problem I. Golden Ring

Sort all points by angle from the river source. Build convex hull. Take all points that didn't end up in convex hull in the sorted order, and then take all points that did end up in the convex hull in the reversed order. If the cycle does intersect the river, then solution doesn't exist.

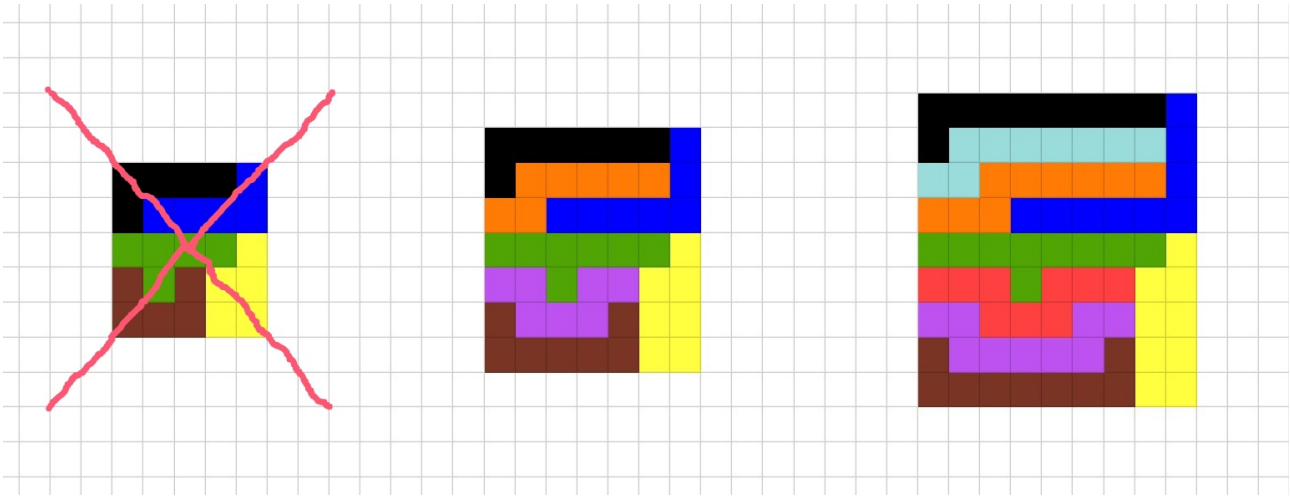
Problem J. Multimino

We want to find some pattern that, when repeated with different parameters, will yield many different n-minos that are structurally similar and fit well together. There are surely many implementations of this idea, and below is one of the quite natural ones.

Let's devise a construction for even sufficiently large n. We will build it sequentially from the example for $n = 6$ shown on the left, adding additional n-minos—"zigzags— on top of the cut square and n-minos—"trays— on the bottom each time we transition from n to $n + 2$:

We just need to slightly modify the construction so that it also works for odd n. For example, this can be done as shown on the right:

This results in a sequence of cuts that is almost identical to the sequence for even n, with the only important note being that they are valid only starting from $n = 7$, since for $n = 5$, the black and blue n-minos turn out to be equal:



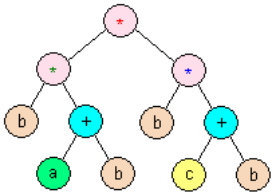
For small n, we will find the answer manually. Fortunately, the most complex cases $n = 4, 5$ have already been addressed in the problem statement; for $n = 2, 3$, it can be trivially proven that the required partition does not exist; for $n = 1$, one 1-mino will suffice, and for $n \geqslant 6$, our construction above is already fully valid, so we will apply it.

Problem K. Postfix Notation Blocks

This problem is quite involved. You are given a postfix expression and asked to optimize it according to its RLE-size. To do this optimization you may freely use associativity and commutativity. The solution to this problem can be done in phases:

1. Parse the expression into a more suitable form, namely a syntax tree.
2. Modify the syntax tree to take into account associativity.
3. Use commutativity to get the optimal size.

Step 1: Building the syntax tree



A syntax tree has a node for each character in the input. If the character is a variable, then it will be a leaf in the tree. If the character is an operator, then it will become an intermediate node in the tree, with its operands as its children.

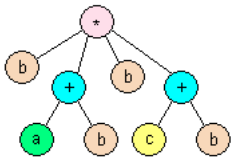
The problem statement hints at how to construct the parse tree: create a "stack calculator" program. Loop through each character of the string:

- If it is a letter, create a new leaf node and push it on the stack.
- If it is an operator, create a new intermediate node, pop two nodes from the stack, and add the popped nodes as its children.

For a well-formed postfix expression, you will never need to pop from an empty stack, and the stack will have exactly one node at the end.

Note the difference between this algorithm and evaluating the expression: here we are placing nodes of a tree onto the stack, not intermediate values.

Step 2: Modifying the syntax tree to account for associativity



What does associativity mean in a syntax tree? If you look at the tree, you can merge nodes representing the same operator if they are connected. For example, consider two $*$ nodes: if one is a child of the other, then due to associativity we can merge them into a single node with more than two children.

The algorithm is simple: traverse the tree, and whenever an operator node has a child which is the same operator, merge the two nodes. This process results in a tree where each operator node can have more than two children, representing multiple operands combined by the same operator.

Step 3: Accounting for commutativity and finding the optimal size

Associativity says the order of operations does not matter. Commutativity says the order of operands does not matter. Therefore, we can reorder the subtrees of an operator node to minimize the final RLE-size.

Suppose we have k subtrees for a given operator. The postfix form of such a node will be:

$$\langle \text{subtree}_1 \rangle \langle \text{subtree}_2 \rangle \dots \langle \text{subtree}_k \rangle \langle \text{operator} \rangle^{k-1}.$$

The resulting RLE-size is approximately the sum of the sizes of the subtrees plus $k-1$, except that we can reduce it by 1 whenever the last character of one subtree matches the first character of the next subtree.

Key Observations

- The start of every subtree must be a variable.
- If the root of a subtree is an operator, the last character in its postfix form is that operator.
- If the subtree is just a variable, it begins and ends with that letter.

Thus, we classify subtrees into two sets:

1. Variables (single-letter subtrees).
2. Operations (subtrees whose root is an operator).

Greedy Pairing. If you have two identical variables, always put them next to each other for a savings of 1. This does not prevent future optimizations because combined identical variables still start and end with the same letter.

There is no benefit to putting two different variables together, nor two operations together (since operations end with operators). The best savings occur when pairing variables with operations.

This leads to a **maximum bipartite matching** problem: we want to match as many variables as possible with operations such that the starting letter of the operation matches the variable.

Determining Starting Variables

Each subtree should keep track of which variables it can start with. For example:

- The subtree `ab+` can start with either `a` or `b`.

- The subtree `cb+` can start with either `c` or `b`.

After computing maximum bipartite matching, we determine which operations remain unmatched. For each unmatched operation, we add all its valid starting variables to the list of possible starting variables of the entire expression.

To find unmatched operations, run the bipartite matching routine normally, and then re-run it while excluding one operation at a time. If the size of the matching does not decrease, that operation can remain unmatched.

Following this procedure bottom-up through the syntax tree yields the minimal possible RLE-size.

Problem L. Transformations

Note that to solve the problem, it is sufficient to obtain any permutation from the identity permutation $\{1, 2, \ldots, n\}$. Obtaining permutation b from permutation a is equivalent to obtaining permutation $a^{-1}b$ from the identity permutation, where a^{-1} is the inverse permutation of a.

Suppose we want to obtain permutation p from the identity permutation. We will solve the problem recursively. Let's consider the last rearrangement after which we will obtain permutation p. We will find a solution in which $k = \lfloor \frac{n}{2} \rfloor$ people left the line during this rearrangement. Clearly, these are the people numbered $p_1, p_2, \ldots, p_k$, since they must be at the front of the line at the end. We will divide all the people into two groups: those with numbers $p_1, p_2, \ldots, p_{k-1}, p_k$ and those with numbers $p_{k+1}, p_{k+2}, \ldots, p_{n-1}, p_n$.

To solve the problem, it is necessary and sufficient that before the last rearrangement, the people in the first group were in the order $p_k, p_{k-1}, \ldots, p_2, p_1$ relative to each other (since their relative order will be reversed during the last rearrangement), and the people in the second group were in the order $p_{k+1}, p_{k+2}, \ldots, p_{n-1}, p_n$ relative to each other. The relative order of people from different groups is not important to us.

Note that now the problem has been split into two independent problems: for the people within each group, we need to obtain a different order from some initial order. These problems can be solved in parallel, meaning that with one rearrangement of the entire line, we can simultaneously perform an arbitrary rearrangement within the first group and an arbitrary rearrangement within the second group. Since the relative order of people from different groups is not important to us, this will not change the result.

Thus, the problem can be solved recursively. We will construct the necessary rearrangements for each group, and after that, we will combine the pairs of rearrangements within the groups into rearrangements of the entire line.

We have reduced the problem for a permutation of size n to two problems for permutations of size $\frac{n}{2}$ and one additional rearrangement. Therefore, the total number of rearrangements for a permutation of size n will not exceed $\lceil \log_2 n \rceil$. This is sufficient to stay within 15 rearrangements for permutations of size no more than $10,000$.

Problem M. Distinctive Features

We will learn to find the nearest smartphone to the left with the same feature for each smartphone and each of its features in total time $O(n + m + s)$ (or -1 if there is none).

To do this, we will create an array *prev* of length m, initially filled with -1. We will iterate from left to right over the smartphones, and for each feature of the current smartphone i, the answer will be stored in the *prev* array. We will save it and replace it with i, as this is now the last smartphone with that feature.

In a similar manner, for each feature of each smartphone, we will find the nearest smartphone to the right with the same feature (or n, if there is none) by iterating from right to left.

Thus, instead of features for smartphone i, we now have several pairs (l_i^j, r_i^j), and to answer the query (l, r, p), we need to count how many such j satisfy $l < l_p^j$ and $r_p^j < r$.

We will divide all queries by p. Now for each p, we have several segments from Reduction 1: (l_p^j, r_p^j), and several queries (lq_p^i, rq_p^i). For each query, we need to count the number of segments for which $l_p^j < lq_p^i$ and $rq_p^i < r_p^j$. If we consider l and r as coordinates x and y on a plane, we obtain the problem of counting points in a rectangle.

The standard problem of counting points in a rectangle can be solved in $O((m_p + q_p)^{(3/2)})$ or $O((m_p + q_p) \cdot \log^2(m_p + q_p))$ or $O((m_p + q_p) \cdot \log(m_p + q_p))$ for each smartphone p with m_p features and q_p queries with that p. For this, we can use a root, a two-dimensional Fenwick tree, or a scanline with a segment tree or a Fenwick tree, respectively, obtaining different scores depending on the quality of the implementation.

Thus, the best obtained asymptotic complexity is: $O((s + q) \cdot \log(s + q))$.