Problem A. Fugaku NK-kei

Approach

Each painting's replica is collected in whichever of its depicted regions is visited first. So we can view each painting as a cyclic interval of length at most K, assigned to the first visited region inside it. We call $(C_1, \ldots, C_N)$ the *count sequence*.

If $K = 1$, every painting depicts exactly one region. So for every visiting order, $(C_1, \ldots, C_N) = (1, \ldots, 1)$, and the answer is 1.

From now on, assume $K \geq 2$.

First, suppose the vertices $1, 2, \ldots, n$ lie on a line instead of a circle. Let a_n be the number of distinct count sequences we get by assigning all contiguous intervals of length at most K in the same way, where $a_0 = a_1 = 1$. Also, let $\mathrm{Cat}_n = \frac{1}{n+1}\binom{2n}{n}$ be the n-th Catalan number.

Lemma 1. For $n \geq 2$,

$$a_n = 2a_{n-1} + \sum_{j=2}^{\min(K-1,n-1)} \mathrm{Cat}_{j-1} a_{n-j}.$$

Moreover, if $n < K$, then $a_n = \mathrm{Cat}_n$.

Proof. To deal with intervals that cross a boundary, we also consider a variant in which h dummy vertices are placed to the left of the line, where $0 \leq h \leq K - 1$.

The dummy vertices stand for vertices on the left that have already been split off. Intervals containing dummy vertices are also taken into account. However, an interval that contains at least one actual vertex is never assigned to a dummy vertex. We record only the counts of the n actual vertices.

Let $L_{n,h}$ be the set of count sequences obtainable in this variant. The case $h = 0$ is the original problem on a line.

We prove the following two statements simultaneously by induction on n:

- $|L_{n,h}| = a_n$, regardless of h.
- If $n < K$, then $a_n = \mathrm{Cat}_n$.

Both statements are immediate for $n = 0, 1$.

Let j be the largest integer with $1 \leq j \leq \min(K, n)$ such that vertex 1 is visited first among vertices $1, 2, \ldots, j$. Then, if $j < \min(K, n)$, vertex $j + 1$ is visited before vertex 1.

The number of intervals assigned to vertex 1 is

$$\sum_{q=1}^{j} \min(h + 1, K - q + 1).$$

This is strictly increasing in j, so j is uniquely determined by the first entry of the count sequence.

First, consider $j = 1$. Vertex 2 is visited before vertex 1, so no interval that contains vertex 1 and reaches vertex 2 or beyond is assigned to vertex 1.

After removing vertex 1, we can treat the original h dummy vertices together with vertex 1 as $h + 1$ dummy vertices on the left. Dummy vertices too far away to share an interval of length at most K with an actual vertex have no effect, so it suffices to keep at most $K - 1$ of them.

Hence the remaining problem is equivalent to

$$L_{n-1,\min(K-1,h+1)},$$

which has a_{n-1} possibilities by the induction hypothesis.

Next, consider $j = \min(K, n)$. Every interval of length at most K containing vertex 1 is assigned to vertex 1. So after vertex 1 is removed, no interval coming from the left affects the remaining vertices. This leaves the ordinary problem on a line with $n-1$ vertices, which has a_{n-1} possibilities.

Finally, consider

$$2 \le j \le \min(K-1, n-1).$$

Here, write $u \to v$ to mean that vertex u is visited before vertex v. In this case,

$$j+1 \to 1 \to \{2, 3, \ldots, j\}.$$

The intervals assigned to vertices $2, \ldots, j$ are exactly those lying entirely within this range. This part has $j - 1 < K$ vertices, so by the induction hypothesis it has Cat_{j-1} possibilities.

Now consider vertices $j+1, \ldots, n$. Any interval containing both a vertex of this part and a vertex to its left must contain vertex $j+1$. Moreover, vertex $j+1$ is visited before every vertex among $1, \ldots, j$, so such an interval is never assigned to a vertex on the left.

Therefore all $h + j$ vertices to the left can be treated as dummy vertices, and the remaining problem is equivalent to

$$L_{n-j,\min(K-1,h+j)},$$

which has a_{n-j} possibilities.

The left and right parts can be chosen independently, and j is uniquely determined by the first entry. Hence,

$$a_n = 2a_{n-1} + \sum_{j=2}^{\min(K-1,n-1)} \mathrm{Cat}_{j-1} a_{n-j}.$$

The right-hand side does not depend on h, so this also proves $|L_{n,h}| = a_n$.

Furthermore, when $n < K$, substituting the induction hypothesis gives

$$a_n = 2\mathrm{Cat}_{n-1} + \sum_{j=2}^{n-1} \mathrm{Cat}_{j-1}\mathrm{Cat}_{n-j} = \mathrm{Cat}_n,$$

where the last equality is the standard Catalan recurrence.

This completes the proof. □

Let

$$H(x) = \sum_{n \ge 1} a_n x^{n-1}.$$

By Lemma 1,

$$H(x) = \frac{1}{F(x)}, \qquad F(x) = 1 - 2x - \sum_{j=2}^{K-1} \mathrm{Cat}_{j-1} x^j.$$

Applying the result to the circle

Region i is contained in exactly d intervals of length d, so

$$C_i \le 1 + 2 + \cdots + K = \frac{K(K+1)}{2}.$$

We call a region a *cut point* if equality holds.

Lemma 2. There is at least one cut point, and there are at least $K-1$ regions between any two consecutive cut points.

Proof. The region visited first overall is a cut point, since every interval containing it is assigned to it.

Also, if there were at most $K-2$ regions between two cut points, some interval of length at most K would contain both of them. That is impossible, since an interval cannot be assigned to two regions. □

Select some of the cut points, and suppose there are g regions between two consecutive selected cut points. These g regions form an instance of the problem on a line, which has a_g possibilities. By Lemma 2, $g \geq K-1$.

Call a selected cut point together with the g regions that follow it, up to the next selected cut point, a *block*, and define

$$D(x) = \sum_{g \geq K-1} a_g x^{g+1}.$$

Selecting r cut points splits the circle into r blocks. There are N choices for where the first block starts, and each selection is counted r times, once for each selected cut point taken as the start. So the number of such configurations is

$$\frac{N}{r}[x^N]D(x)^r.$$

A count sequence with exactly m cut points is counted $\binom{m}{r}$ times when r cut points are selected. Since

$$\sum_{r=1}^{m}(-1)^{r+1}\binom{m}{r} = 1,$$

inclusion–exclusion gives

$$\text{Ans} = \sum_{r \geq 1}(-1)^{r+1}\frac{N}{r}[x^N]D(x)^r = N[x^N]\log(1+D(x)).$$

Computing the answer

Let

$$A(x) = \sum_{n \geq 0} a_n x^n.$$

Then

$$A(x) = 1 + xH(x) = 1 + \frac{x}{F(x)}.$$

Define

$$E(x) = F(x) + x = 1 - \sum_{j=1}^{K-1} \text{Cat}_{j-1} x^j.$$

Then $A(x) = E(x)/F(x)$.

Also, by Lemma 1, $a_g = \text{Cat}_g$ for $g \leq K-2$, so

$$D(x) = x\left(A(x) - \sum_{g=0}^{K-2} \text{Cat}_g x^g\right).$$

By the definition of $E(x)$,

$$1 - E(x) = x\sum_{g=0}^{K-2} \text{Cat}_g x^g.$$

Substituting this and simplifying gives

$$1 + D(x) = \frac{E(x)^2}{F(x)}.$$

Therefore,

$$\mathrm{Ans} = N[x^N]\left(2\log E(x) - \log F(x)\right).$$

Since $(\log G(x))' = G'(x)/G(x)$, we obtain

$$\mathrm{Ans} = [x^{N-1}]\left(2\frac{E'(x)}{E(x)} - \frac{F'(x)}{F(x)}\right).$$

So the answer can be computed from the formal power series inverses $E(x)^{-1}$ and $F(x)^{-1}$.

Complexity

Computing the first N coefficients of $E(x)^{-1}$ and $F(x)^{-1}$ with NTT and Newton's method solves each test case in $O(N \log N)$ time and $O(N)$ space.

Alternatively, the Bostan–Mori algorithm gives a solution in $O(K \log K \log N)$ time and $O(K)$ space.

Problem B. PerMutation

Solution for a Fixed Permutation

First, set aside the queries that update P and consider the problem with P fixed.

The update operation on the sequence A is defined by

$$A_j \leftarrow \left\lfloor \frac{A_{j-1}A_{j+1}}{A_j} \right\rfloor + 1.$$

This can be rewritten as

$$A_j \leftarrow \left\lceil \frac{A_{j-1}A_{j+1}+1}{A_j} \right\rceil.$$

For now, consider the following operation, obtained by removing the ceiling:

$$A_j \leftarrow \frac{A_{j-1}A_{j+1}+1}{A_j}.$$

As we will show later, $A_{j-1}A_{j+1}+1$ is always divisible by A_j under this operation, so A_j always remains an integer. Assuming this fact, the new operation always produces the same result as the original operation. Thus, it suffices to solve the problem using the new operation.

Consider an infinite two-dimensional grid, and define cells $S_1, \ldots, S_N$ as follows. Here, $(1, 0)$ points to the right, and $(0, 1)$ points upward.

- Set $S_1 = (1, -1)$.
- For each $j = 2, \ldots, N$, if $j-1$ is updated before j, that is, if $P_{j-1}^{-1} < P_j^{-1}$, set $S_j = S_{j-1} + (1, 0)$, the cell immediately to the right of S_{j-1}.
- Otherwise, if j is updated before $j-1$, that is, if $P_{j-1}^{-1} > P_j^{-1}$, set $S_j = S_{j-1} + (0, -1)$, the cell immediately below S_{j-1}.

For convenience, also set $S_0 = (1, 0)$ and $S_{N+1} = S_N + (1, 0)$.

We will write numbers on the grid while moving these cells according to the updates to A. Initially, write 1 in each cell S_i. Whenever A_j is updated, perform the following operation:

- Replace S_j with $S'_j = S_j + (1, 1)$, the cell diagonally above and to the right, and write the updated value of A_j in the new cell S_j.

In addition, after every N updates to A, when one full pass through the indices is complete, replace S_0 and S_{N+1} with $S_0 + (1,1)$ and $S_{N+1} + (1,1)$, respectively, and write 1 in both new cells.

Repeat this process indefinitely. Let $B_{i,j}$ denote the value written in cell (i,j). If the initial coordinates of S_j are (a_j, b_j), then after NK updates to A, we have

$$f(P,K,x) = B_{a_x+K,b_x+K}.$$

Whenever A_j is updated, the cells S_j, S_{j+1}, S_{j-1}, and S'_j form a 2×2 square. Therefore, wherever the relevant values have been written, we have

$$B_{i,j}B_{i+1,j+1} = B_{i+1,j}B_{i,j+1} + 1.$$

We can also extend the region in which values are defined by setting

$$B_{x,x} = B_{x+N+3,x} = 0 \qquad (x \in \mathbb{Z}).$$

The same identity continues to hold wherever all its terms are defined.

Conversely, the recurrence uniquely determines the remaining values of B from the following conditions:

- $B_{a_j,b_j} = 1$ for $j = 0, \ldots, N$.
- $B_{x,x} = B_{x+N+3,x} = 0$ for all $x \in \mathbb{Z}$.

All values other than those on the boundary remain positive, so division by zero never occurs.

Next, consider a sequence $\{v_i\}_{i\in\mathbb{Z}}$ of two-dimensional column vectors satisfying

$$\det(v_i \ v_{i+1}) = 1 \qquad (i \in \mathbb{Z}).$$

Define

$$C_{i,j} = \det(v_j \ v_i).$$

Using properties of determinants, we obtain

$$C_{i,j}C_{i+1,j+1} = C_{i+1,j}C_{i,j+1} + 1.$$

Therefore, if such a vector sequence also satisfies the following conditions, the recurrence implies $B_{i,j} = C_{i,j}$ for every cell in which a value has been written:

- $C_{a_j,b_j} = 1$ for $j = 1, \ldots, N$.
- $C_{x,x} = C_{x+N+3,x} = 0$ for all $x \in \mathbb{Z}$.

We can construct the vectors v_i explicitly as follows.

- Set
$$v_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad v_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$
- For $j = 1, 2, \ldots, N+1$ in order, define the next vector as follows:
  - If $(a_j, b_j) = (a_{j-1} + 1, b_{j-1})$, set
  $$v_{a_j} = v_{a_{j-1}} - v_{b_{j-1}}.$$
  - If $(a_j, b_j) = (a_{j-1}, b_{j-1} - 1)$, set
  $$v_{b_j} = v_{b_{j-1}} - v_{a_{j-1}}.$$

- Since $a_{N+1} - b_{N+1} = N + 2$, we have now defined the vectors at $N + 3$ consecutive indices. Extend the sequence so that

$$v_{i+N+3} = -v_i.$$

We can verify that this construction satisfies all the required conditions:

- $\det(v_i\ v_{i+1}) = 1$ for all $i \in \mathbb{Z}$.
- $C_{a_j,b_j} = 1$ for $j = 1, \ldots, N$.
- $C_{x,x} = C_{x+N+3,x} = 0$ for all $x \in \mathbb{Z}$.

Every component of every v_i in this construction is an integer. Thus, $B_{i,j} = C_{i,j}$ is also always an integer, proving that A_j always remains an integer under the update operation.

After precomputing the vectors v_i, we can compute $f(P, K, x)$ in $O(1)$ time.

Handling Updates

Now consider the queries that update P.

The discussion above shows that it suffices to maintain the vectors v_i efficiently. Their values depend on the comparisons between P_{j-1}^{-1} and P_j^{-1}. Swapping P_i and P_j changes the inverse-permutation entries corresponding to the two swapped elements. Only comparisons involving these elements can change, so at most four comparisons are affected. It therefore suffices to update these positions locally.

Consider the construction of v_i. The matrix

$$(v_{a_j}\ v_{b_j})$$

is obtained by multiplying the preceding matrix

$$(v_{a_{j-1}}\ v_{b_{j-1}})$$

on the right by an appropriate 2×2 matrix.

Thus, $(v_{a_j}\ v_{b_j})$ can be expressed using a prefix product of these matrices. A segment tree allows us to update the matrices and retrieve the required products in $O(\log N)$ time.

To answer a query, we also need to find the step j corresponding to the desired vector index i, where

$$i \equiv a_j \pmod{N+3} \qquad \text{or} \qquad i \equiv b_j \pmod{N+3}.$$

This reduces to finding the k-th smallest element of the set of indices j satisfying $P_{j-1}^{-1} < P_j^{-1}$, or of the set satisfying $P_{j-1}^{-1} > P_j^{-1}$.

These queries can be handled efficiently using binary search on a Binary Indexed Tree, a balanced binary search tree, or a similar data structure.

With a suitable implementation, the total time complexity is $O(N + Q \log N)$.

Problem C. Moving Silver

Approach

For a board B, let $\mathrm{Max}(B)$ be the board whose value at each cell is the maximum value of B over the cells reachable from that cell in one move of the silver general. Define $\mathrm{Min}(B)$ analogously with the minimum.

Let

$$F(B) = \mathrm{Max}(\mathrm{Min}(B)),$$

which corresponds to two moves: one by Alice followed by one by Bob. The board of game values is $F^k(A)$ when $N = 2k$, and $F^k(\mathrm{Max}(A))$ when $N = 2k + 1$.

We will prove the following property:

Let $K = \lfloor H/2 \rfloor + 1$. For every board B,

$$F^{K+1}(B) = F^K(B).$$

Given this property, we only need to apply F at most K times. For an even number of moves, compute

$$A, F(A), F^2(A), \ldots$$

in order. For an odd number of moves, compute

$$\mathrm{Max}(A), F(\mathrm{Max}(A)), F^2(\mathrm{Max}(A)), \ldots$$

in order.

For each query, let

$$k = \min(\lfloor N/2 \rfloor, K).$$

Counting the initial board as the 0-th board, the answer is the value at (X, Y) in the k-th board of the first sequence if N is even, and of the second sequence if N is odd.

By grouping the queries in advance by the parity of N and the value of k, we can answer each group while updating the boards one by one.

The property also justifies a dynamic programming solution: for each number of remaining moves, maintain the boards of game values with Alice to move and with Bob to move, update them one move at a time, and stop as soon as the state coincides with the state two moves earlier.

Proof

Fix an arbitrary integer a, and consider only whether the value at each cell is at least a.

Let S_t be the set of cells whose values in $F^t(B)$ are at least a, and let $M(v)$ be the set of cells reachable from cell v in one move of the silver general.

Min gives a value of at least a at a cell if and only if every destination from that cell has a value of at least a, and the subsequent Max only needs one destination with this property. Thus,

$$v \in S_{t+1} \iff \text{there exists } u \in M(v) \text{ such that } M(u) \subseteq S_t.$$

For this proof only, number the rows $0, 1, \ldots, H-1$.

Cells entering S_t

Let $t \geq 1$. We show that if row r contains a cell in $S_{t+1} \setminus S_t$, then row $r-2$ contains a cell in $S_t \setminus S_{t-1}$.

Take $v = (r, c) \in S_{t+1} \setminus S_t$. Since $v \in S_{t+1}$, there exists $u \in M(v)$ such that $M(u) \subseteq S_t$.

If the move from v to u were diagonal, the reverse move would also be legal, which would imply $v \in M(u) \subseteq S_t$, a contradiction. Therefore, $u = (r-1, c)$.

Also, since $v \notin S_t$, we have $M(u) \not\subseteq S_{t-1}$. Hence, some $w \in M(u)$ belongs to $S_t \setminus S_{t-1}$.

The cells reachable from u are the cells in row $r-2$ and the cells immediately to the left and right of v in row r.

Suppose that $w = (r, c \pm 1)$. Since $w \in S_t \setminus S_{t-1}$, the same argument shows that the cell witnessing $w \in S_t$ must be $(r-1, c \pm 1)$. However, v can also reach this cell by a diagonal move upward, which would imply $v \in S_t$, a contradiction. Therefore, w lies in row $r-2$.

Repeating this argument, if row r contains a cell in $S_{t+1} \setminus S_t$, then row $r - 2t$ contains a cell in $S_1 \setminus S_0$.

Every legal move from row 0 is a diagonal move, which is reversible, so no cell in row 0 can ever enter the set. Thus, $r - 2t \geq 1$. In particular, if $S_{t+1} \setminus S_t$ is nonempty, then

$$t \leq \left\lfloor \frac{H-2}{2} \right\rfloor.$$

Cells leaving S_t

For each t, define

$$Z_t = \{u \mid M(u) \cap S_t^c \neq \emptyset\},$$

that is, the set of cells from which the piece can move outside S_t in one move.

Negating the update rule above gives

$$v \notin S_{t+1} \iff M(v) \subseteq Z_t.$$

Moreover, Z_t follows the same update rule as S_t. Indeed, $u \in Z_{t+1}$ holds if and only if some $v \in M(u)$ does not belong to S_{t+1}. Therefore,

$$u \in Z_{t+1} \iff \text{there exists } v \in M(u) \text{ such that } M(v) \subseteq Z_t,$$

which is exactly the update rule for S_t.

Now let $t \geq 1$, and take $v \in S_t \setminus S_{t+1}$. Since $v \notin S_{t+1}$, we have $M(v) \subseteq Z_t$. On the other hand, since $v \in S_t$, we have $M(v) \nsubseteq Z_{t-1}$. Hence, some $w \in M(v)$ belongs to $Z_t \setminus Z_{t-1}$.

That is, if some cell leaves the set between S_t and S_{t+1}, then some cell enters the set between Z_{t-1} and Z_t.

Applying the bound proved above to Z_t, we obtain

$$t - 1 \leq \left\lfloor \frac{H-2}{2} \right\rfloor,$$

or equivalently,

$$t \leq \lfloor H/2 \rfloor.$$

This bound also holds trivially for $t = 0$. Therefore, S_t and S_{t+1} can differ only when $t \leq \lfloor H/2 \rfloor$.

Consequently, for $K = \lfloor H/2 \rfloor + 1$, we have $S_{K+1} = S_K$. Since this holds for every integer a, the original integer-valued boards also satisfy

$$F^{K+1}(B) = F^K(B).$$

Complexity

Both Max and Min can be computed in $O(HW)$ time, since each cell has at most five destinations.

We apply F a total of $O(H)$ times and answer each query in $O(1)$ time. Therefore, the time complexity is $O(H^2W + Q)$, and the space complexity is $O(HW + Q)$.

Problem D. Go Robots

Simulating the movement of the robot K times will not run in time because K can be extremely large.

For each switch, its state toggles between `L` and `R` every time a robot passes through it.

In other words, if we can determine the total number of times robots pass through each vertex, we can find the answer by summing up the total number of passes across all vertices.

Since the given graph is a directed acyclic graph (DAG), a single robot will never visit the same vertex more than once.

Therefore, we can consider computing the values in topological order (which, in this problem, corresponds to the increasing order of vertex indices).

The number of robots passing through vertex 1 is K.

When the number of robots C_i passing through vertex i is known, we can also compute the number of robots moving along the edges $i \to L_i$ and $i \to R_i$.

Since the robots reaching vertex i are dispatched alternately as $\texttt{R} \to \texttt{L} \to \texttt{R} \to \texttt{L} \to \cdots$, we can see that:

- The number of robots moving to L_i is $\lfloor C_i/2 \rfloor$
- The number of robots moving to R_i is $\lceil C_i/2 \rceil$

Using this rule, we can compute C_i in ascending order of vertex indices, and the answer will be $\sum_{i=1}^{N-1} C_i$.

The total time complexity is $O(N)$.

Problem E. K-Independent Matrix

Approach

Output the matrix defined by $A_{i,j} = \gcd(i, j)$ for $1 \le i, j \le N$.

Correctness Proof

We show that this matrix satisfies all the required conditions.

First, $1 \le \gcd(i, j) \le N$ for all $1 \le i, j \le N$, so every entry is between 1 and N, inclusive.

Next, fix any K with $1 \le K \le N$ and consider any $K \times K$ subsquare, consisting of rows $r, r+1, \ldots, r+K-1$ and columns $c, c+1, \ldots, c+K-1$.

Among any K consecutive integers, exactly one is a multiple of K. Therefore:

- Exactly one of the row indices $r, r+1, \ldots, r+K-1$ is a multiple of K.
- Exactly one of the column indices $c, c+1, \ldots, c+K-1$ is a multiple of K.

Thus, the subsquare contains a unique cell (i, j) such that $K \mid i$ and $K \mid j$.

Since $A_{i,j} = \gcd(i, j)$, we have

$$K \mid A_{i,j} \iff K \mid \gcd(i, j) \iff (K \mid i \text{ and } K \mid j).$$

Hence, this $K \times K$ subsquare contains a unique entry that is a multiple of K.

Complexity

Computing $\gcd(i, j)$ for each entry in $O(\log N)$ time takes $O(N^2 \log N)$ time per test case, and $O\left(\sum N^2 \log N\right)$ time in total.

Problem F. Path XOR on MST

Define the color of a cycle in G as the bitwise XOR of the colors of all its edges.

First, suppose that the minimum spanning tree T of G is already given, and consider how to compute the answer. Let T' be the graph obtained from T by adding an edge of color 0 between A_i and B_i for each $i = 1, \ldots, K$.

For each added edge (A_i, B_i), combining it with the path between A_i and B_i in T gives a cycle in T'. These K cycles form a cycle basis of T', and their colors are exactly the values P_i.

Changing the cycle basis does not change the vector space over $\mathbb{F}_2$ that it spans. Thus, it suffices to maintain an XOR basis of the colors of the cycles in any cycle basis of T'. We can therefore compute the answer using the following algorithm.

1. Prepare an array for maintaining an XOR basis.
2. Prepare a disjoint set union structure with XOR potentials. For each $i = 1, \ldots, K$, merge A_i and B_i with XOR weight 0.

3. Process the edges (u, v) of T one by one, merging their endpoints with XOR weight equal to the edge color. If u and v already belong to the same connected component, take the XOR of the edge color and the existing XOR potential difference to obtain the color of the resulting cycle. Reduce this value against the XOR basis using Gaussian elimination, and insert it into the basis if the result is nonzero.

4. Compute the answer using the resulting XOR basis.

An advantage of this approach is that, by making the disjoint set union structure with XOR potentials support undo operations, we can add edges and undo the most recent update.

We now use this method to solve the original problem by divide and conquer on the queries.

Prepare a rollback disjoint set union structure D for maintaining the minimum spanning tree T, a rollback disjoint set union structure with XOR potentials D' for computing the answers, and an array for maintaining an XOR basis.

Suppose that we have a set S of candidate edges that may be added to T, and want to process the queries in the interval $[L, R)$ recursively. Although the weights of the edges added by these queries are not yet known, we must use edges of S to connect the parts that cannot be connected by the query edges alone. Also, whenever the edges of S contain a cycle, we may discard an edge of maximum weight on that cycle.

Thus, define solve(L, R, S), which processes the queries in $[L, R)$ with candidate edge set S, as follows.

1. Sort the edges of S by increasing weight, breaking ties by increasing index.

2. Add all edges corresponding to queries in $[L, R)$ to D.

3. Add the edges of S to D in order. Let T be the set of edges of S that join different connected components.

4. Roll back D to its state at the beginning of this call.

5. Add the edges of T to D. At the same time, add them to D' and update the XOR basis.

6. Add the edges of $S \setminus T$ to D in order. Let S' be the set of edges that join different connected components. Then roll back D to its state immediately after step 5.

7. Let $M = \lfloor (L + R)/2 \rfloor$, and call solve$(L, M, S')$.

8. Add the edges introduced by queries in $[L, M)$ to S', and call solve(M, R, S').

9. Roll back D, D', and the XOR basis to their states at the beginning of this call.

Initially, put the M edges of the initial graph into S and call solve$(0, Q + 1, S)$. When $R - L = 1$, process the corresponding query.

Time complexity

Each operation on a rollback disjoint set union structure takes $O(\log N)$ time. Inserting a value into the XOR basis or finding the maximum value takes $O(\log \max C)$ time. Thus, excluding recursive calls, solve(L, R, S) takes

$$O\big(|S|(\log |S| + \log N + \log \max C) + (R - L) \log N\big)$$

time.

In solve(L, R, S), after adding the edges of T, the number of connected components in D is at most $R - L + 1$. This follows because adding all edges corresponding to queries in $[L, R)$ at this point would make the graph connected. Therefore, S' contains at most $R - L$ edges.

Consequently, the sum of $|S|$ over all calls at any fixed recursion depth is $O(Q)$, except at the top level, where it is $O(M)$. The recursion depth is $O(\log Q)$, so the overall time complexity is

$$O\big(M(\log M + \log \max C) + K \log N + Q \log Q(\log \max C + \log N + \log Q)\big).$$

Problem G. Source to Sink

By considering each weakly connected component separately, we may assume that G is weakly connected.

If G is a DAG, it has at least one vertex with indegree 0. Choose such a vertex v and perform the operation on it. The resulting graph is also a DAG, so we can continue performing operations indefinitely.

Now suppose that G is not a DAG. Let A_v be the number of operations performed on vertex v up to some point.

If v belongs to a directed cycle in the initial graph G, then $A_v = 0$. Indeed, suppose that an operation is performed on a vertex belonging to a directed cycle, and consider the first such operation. Immediately before this operation, none of the edges of that cycle have been reversed. Thus, the chosen vertex cannot have indegree 0, a contradiction.

Also, for each edge $u \to v$ in the initial graph G, we must operate on u before we can operate on v. Each operation on either endpoint reverses this edge, so operations on these two vertices must alternate in the order $u, v, u, \ldots$. Therefore,

$$A_v \le A_u \le A_v + 1.$$

We have shown that the operation counts must satisfy the following conditions:

- $A_v \ge 0$ for every vertex v.
- $A_v = 0$ for every vertex v belonging to a directed cycle in G.
- $A_v \le A_u \le A_v + 1$ for every edge $u \to v$ in G.

In fact, whenever these conditions hold, there exists a valid sequence of operations that operates on each vertex v exactly A_v times.

Proof.

We use induction on the total number of operations $\sum_v A_v$.

The case $\sum_v A_v = 0$ is immediate.

Suppose that $\sum_v A_v > 0$, and let S be the set of vertices where A_v attains its maximum. No vertex in S belongs to a directed cycle in G, so the subgraph induced by S is a DAG. Hence, we can choose a vertex $v \in S$ with no incoming edge from another vertex in S.

For every vertex $u \notin S$, we have $A_u < A_v$, so the edge $u \to v$ cannot exist either. Thus, v has indegree 0 in G, and we can perform the operation on v.

Let G' be the resulting graph, and define

$$A'_v = A_v - 1, \qquad A'_u = A_u \quad (u \ne v).$$

It is straightforward to check that G' and A' satisfy the same conditions. By the induction hypothesis, we can perform the remaining $\sum_u A'_u$ operations with the prescribed counts. This completes the proof. □

Reduction to shortest paths.

Therefore, we need to maximize $\sum_v A_v$ subject to these conditions. This is a system of difference constraints, which can be reduced to the following shortest path problem:

- For each edge $u \to v$ in G, add an edge of length 0 from u to v and an edge of length 1 from v to u.

- Use all vertices belonging to directed cycles as sources, and let A_v be the shortest distance from these sources to vertex v.

The vertices belonging to directed cycles can be found in $O(N + M)$ time using strongly connected components. The shortest distances from these sources can then be computed in $O(N + M)$ time using 0-1 BFS.

Problem H. Please Insert Coin

Approach

A simple approach is to maintain an array dp, where dp[j] is the minimum cost of being in column j, and update it at each time step.

During the movement phase, propagate the DP values in both directions, taking movement costs into account. During the block placement phase, set the DP values to ∞ throughout the interval of columns that receive new blocks. This gives an $O(KW)$-time solution.

However, in a case such as $H = 1$ and $K = W$, this takes $O(W^2)$ time and exceeds the time limit.

Optimization

Consider the blocks placed at time k. Since the cells with value k are connected and blocks are stacked from bottom to top, the columns receiving new blocks always form a contiguous interval $[L, R]$.

As we will prove later, immediately before blocks are placed in $[L, R]$, it suffices to consider only the following three actions for *soryuusi*:

- Stay in the current column.
- Move to column $L - 1$, if $L \geq 2$.
- Move to column $R + 1$, if $R \leq W - 1$.

With this restriction, only DP values in and around $[L, R]$ need to be updated at each time step.

Specifically, maintain the DP values in a segment tree. For each interval of columns, store the following information:

- The minimum, over all columns x in the interval, of the DP value at x plus the cost of moving from x to the left endpoint of the interval.
- The minimum, over all columns x in the interval, of the DP value at x plus the cost of moving from x to the right endpoint of the interval.
- The cost of moving from the left endpoint of the interval to the right endpoint, or equivalently in the opposite direction.

When blocks are placed in an interval $[L, R]$, perform the following updates:

1. If $L \geq 2$, query the segment tree for the minimum cost of reaching column $L - 1$, and relax the DP value at column $L - 1$.
2. If $R \leq W - 1$, query the segment tree for the minimum cost of reaching column $R + 1$, and relax the DP value at column $R + 1$.
3. Set the DP values throughout $[L, R]$ to ∞.
4. Update the segment tree to reflect the changes in movement costs between adjacent columns caused by the new block heights.

The sum of the interval widths $R - L + 1$ over all time steps is at most the total number of cells, HW. Therefore, even if we update movement costs by performing a point update for each column in $[L, R]$, the total number of such updates is $O(HW)$.

Thus, the overall time complexity is $O(HW \log W)$, which is fast enough.

Correctness Proof

Suppose survival is possible. Fix an optimal strategy and call it strategy A. We construct a strategy B that uses only the restricted actions.

Both strategies start in column s. Whenever blocks are about to be placed in an interval $[L, R]$, let p be the position under strategy A immediately before block placement, and let q be the current position under strategy B. Strategy B acts as follows:

- If $p \leq L - 1$ and $q \geq L$, move to column $L - 1$.
- If $p \geq R + 1$ and $q \leq R$, move to column $R + 1$.
- Otherwise, stay in the current column.

Let C_A and C_B be the cumulative costs of the two strategies, and let $d(q, p)$ be the minimum movement cost from column q to column p under the current block heights. We show that the following invariant is maintained:

$$C_B + d(q, p) \leq C_A.$$

Initially, both positions are s and both cumulative costs are 0, so the invariant holds.

For the comparison, at each time step, first consider the moves of strategy A, then the moves of strategy B, and finally block placement.

1. Movement under strategy A.

   Consider each move to an adjacent column separately. If the move takes p farther from q, then $d(q, p)$ and C_A increase by the same amount, so the invariant is preserved. If the move takes p closer to q, then C_A increases while $d(q, p)$ decreases, so the invariant is again preserved.

2. Movement under strategy B.

   By construction, strategy B always moves toward p without passing it. The increase in C_B is exactly canceled by the decrease in $d(q, p)$, so the invariant is preserved.

3. Block placement.

   Since strategy A survives, p lies outside $[L, R]$. The movement rules ensure that strategy B also lies outside $[L, R]$, on the same side as p.

   Consequently, none of the column heights along the path between q and p change during block placement, so $d(q, p)$ remains unchanged. Neither cumulative cost changes during block placement, so the invariant is preserved.

After the final block placement, the invariant and $d(q, p) \geq 0$ give

$$C_B \leq C_A.$$

Thus, strategy B survives with a cost no greater than that of the optimal strategy A. Therefore, it suffices to consider only the following three actions:

- Stay in the current column.
- Move to column $L - 1$, if $L \geq 2$.

- Move to column $R+1$, if $R \leq W-1$.

Problem I. tatesoto decomposition

Let $N = |X|$. Since $X = T + A + T + E + S + O + T + O$, the strings T and O are a prefix and a suffix of X, respectively. Apply the Z-algorithm to X and its reversal to precompute the following arrays:

- $Z[i]$: the length of the longest common prefix of X and $X[i, N]$.
- $Z'[i]$: the length of the longest common suffix of X and $X[1, i]$.

Here, $X[i, j]$ denotes the substring from position i through position j, using 1-based indexing.

Let $a = |A|$, $e = |E|$, $o = |O|$, $s = |S|$, and $t = |T|$. Consider a fixed value of $k = t + o$. We must have

$$X[N-k+1, N-o] = T = X[1, t]$$

and

$$X[N-k-o+1, N-k] = O = X[N-o+1, N].$$

Thus, we need $t \leq Z[N-k+1]$ and $o \leq Z'[N-k]$. Since $o = k - t$, these conditions can be rewritten as

$$k - Z'[N-k] \leq t \leq Z[N-k+1].$$

Now let $l = t + a$. The total length is

$$N = t + a + t + e + s + o + t + o = l + e + s + 2k,$$

so $e + s = N - 2k - l$. Since every string must be nonempty, we have

$$t + 1 \leq l \leq N - 2k - 2.$$

We also need $X[l+1, l+t] = T = X[1, t]$, which is equivalent to $t \leq Z[l+1]$.

For a fixed l in this range, the conditions on t are therefore

$$\begin{gathered} 1 \leq t \leq k-1, \\ k - Z'[N-k] \leq t \leq Z[N-k+1], \\ t \leq Z[l+1], \\ t \leq l-1. \end{gathered}$$

Conversely, whenever these conditions hold, the positions of A, T, and O are uniquely determined. Define

$$M = \min(k-1, Z[N-k+1]), \qquad m = \max(k - Z'[N-k], 1), \qquad C_l = \min(Z[l+1], l-1).$$

Then the conditions become

$$m \leq t \leq \min(M, C_l).$$

The remaining substring can be split into nonempty strings E and S in any way satisfying $e+s = N-2k-l$. There are $N - 2k - l - 1$ such splits.

Hence, for a fixed k, the number of valid tuples (A, E, O, S, T) is

$$\sum_{l=2}^{N-2k-2} \max(\min(M, C_l) - m + 1, 0)\,(N - 2k - l - 1).$$

We now compute this contribution efficiently. If $m > M$, the contribution is 0, so assume $m \le M$. The summand can be classified according to C_l:

$$\begin{cases} 0, & 0 \le C_l < m, \\ (C_l - m + 1)(N - 2k - l - 1), & m \le C_l < M, \\ (M - m + 1)(N - 2k - l - 1), & C_l \ge M. \end{cases}$$

After expansion, each expression is a linear combination of 1, l, C_l, and lC_l. Thus, for the indices l satisfying $2 \le l \le N - 2k - 2$, it suffices to maintain the sums of these four quantities, indexed by C_l, using a Binary Indexed Tree (BIT) or a similar data structure.

Process k in decreasing order. The upper bound $N - 2k - 2$ then increases monotonically, so we can add each newly eligible index l to the BIT as it enters the range. The contribution for each k can then be computed in $O(\log N)$ time.

The overall time complexity is $O(N \log N)$.

Problem J. Hugest Number

Ordering the Integers

Let $N_d = 9 \cdot 10^{d-1}$ and $M_d = 10^d - 1$ be the number of positive d-digit integers and the largest such integer, respectively.

For an a-digit integer x and a b-digit integer y, placing x before y is at least as good as the reverse when

$$x10^b + y \ge y10^a + x.$$

This is equivalent to

$$\frac{x}{M_a} \ge \frac{y}{M_b}.$$

In the case of equality, both orders give the same concatenation, so we may choose whichever is convenient.

Write the d-digit integers in descending order as $M_d - i$ for $0 \le i < N_d$. Their comparison values are $1 - \frac{i}{M_d}$. Thus, with $a = D_1$ and $b = D_2$, we need to merge

$$\frac{i}{M_a} \quad (0 \le i < N_a) \qquad \text{and} \qquad \frac{j}{M_b} \quad (0 \le j < N_b)$$

in ascending order. Denote the corresponding elements by A_i and B_j, respectively. When only their types matter, we write `A` and `B`.

Computing the Merged Sequence Recursively

Write $b = qa + r$, where $0 \le r < a$. Then

$$M_b = KM_a + M_r, \qquad K = 10^r(1 + 10^a + \cdots + 10^{(q-1)a}).$$

First, break ties by placing A_i before B_j. Let C_i be the number of `B` elements before A_i. This is the number of nonnegative integers j satisfying

$$\frac{j}{M_b} < \frac{i}{M_a},$$

so

$$C_i = \left\lceil \frac{iM_b}{M_a} \right\rceil.$$

Therefore, the number of `B` elements between A_i and A_{i+1} is

$$C_{i+1} - C_i = K + \left(\left\lceil \frac{(i+1)M_r}{M_a} \right\rceil - \left\lceil \frac{iM_r}{M_a} \right\rceil \right).$$

Since $0 \le M_r < M_a$, the expression in parentheses is either 0 or 1. Thus, every `A` except the last is followed by either K or $K+1$ consecutive `B` elements.

Group A_i and the K elements of type `B` immediately following it into a block $Y_i = AB^K$. It remains to determine which blocks Y_i are followed by one additional `B`.

If $r > 0$, an additional `B` appears after Y_i if and only if there exists an integer k such that

$$\frac{i}{M_a} \le \frac{k}{M_r} < \frac{i+1}{M_a}.$$

Thus, it suffices to merge the blocks Y_i, associated with $\frac{i}{M_a}$, and the additional `B` elements, associated with $\frac{k}{M_r}$, in ascending order.

There are $N_a - 1$ blocks Y_i, corresponding to $i = 0, \ldots, N_a - 2$. The additional `B` elements correspond to $k = 0, \ldots, N_r - 1$, so there are N_r of them. Indeed,

$$\frac{(N_a-1)M_r}{M_a} = N_r - 1 + \frac{M_a - M_r}{10M_a},$$

so their total number is

$$\left\lceil \frac{(N_a-1)M_r}{M_a} \right\rceil = N_r.$$

Consequently, this sequence corresponds to the merged sequence for (r, a) with the last element from the a-digit side removed.

To handle this recursively, define $F(a, b; U, V)$ as follows: take the merged sequence for (a, b), remove its last element, and replace each element from the a-digit side with the block U and each element from the b-digit side with the block V. Since

$$\frac{N_d - 1}{M_d} = \frac{9}{10} - \frac{1}{10M_d}$$

is increasing in d, the last element is always a `B` when $a < b$.

Let $Y = UV^K$. If $r > 0$, the additional `B` elements correspond to V, and the blocks Y_i correspond to Y. Thus, the sequence described above can be written as $F(r, a; V, Y)$. The recursion may change the relative order of elements with equal comparison values, but this amounts to swapping equivalent elements in the original merged sequence and therefore does not affect the concatenation.

Next, append U, corresponding to the last `A`, followed by the remaining copies of V. If $r = 0$, there are no additional `B` elements, so we simply repeat Y. Hence,

$$F(a, b; U, V) = \begin{cases} Y^{N_a - 1} U V^{t-1} & (r = 0), \\ F(r, a; V, Y) U V^{t-1} & (r > 0). \end{cases}$$

Here, t is the number of `B` elements after the last `A`. By the definition of $C_{N_a - 1}$,

$$t = N_b - \left\lceil \frac{(N_a-1)M_b}{M_a} \right\rceil = \left\lfloor \frac{9}{10} + \frac{M_b}{10M_a} \right\rfloor = \left\lceil \frac{K}{10} \right\rceil.$$

Therefore, we can represent the merged sequence via the recursion

$$(a, b) \longrightarrow (b \bmod a, a),$$

just as in the Euclidean algorithm.

Example: $(a, b) = (3, 5)$

Consider $(a, b) = (3, 5)$. Since $99999 = 100 \cdot 999 + 99$, we have $K = 100$ and $r = 2$. Initially, ties are broken in favor of A.

The merged sequence begins with

$$999,\ 99999, \ldots, 99900,\ 99899,\ 998,\ 99898, \ldots, 99799,\ 997, \ldots$$

Group each A with the 100 elements of type B immediately following it into a block Y_i. For example,

- $Y_0 = (999, 99999, \ldots, 99900)$;
- $Y_1 = (998, 99898, \ldots, 99799)$;
- $Y_{10} = (989, 98998, \ldots, 98899)$.

Every Y_i has the same symbolic form AB^{100}. Keeping the additional B elements that do not belong to these blocks, the sequence becomes

$$Y_0, B, Y_1, \ldots, Y_{10}, B, Y_{11}, \ldots$$

The first additional B is 99899, and the second is 98898.

The block Y_i corresponds to $\frac{i}{999}$, and an additional B corresponds to $\frac{k}{99}$. For example,

$$\frac{10}{999} < \frac{1}{99} < \frac{11}{999},$$

so an additional B appears immediately after Y_{10}.

This has the same structure as the merging problem for $(2, 3)$: the shorter-digit side corresponds to the additional B elements, and the longer-digit side corresponds to $Y = AB^{100}$. Since ties are broken in favor of Y, this reduces to $F_1(2, 3; B, Y)$.

Next, consider $(a, b) = (2, 3)$. Since $999 = 10 \cdot 99 + 9$, we now have $K = 10$ and $r = 1$.

At this stage, group each additional B with the 10 blocks Y immediately following it into a new block $Z = BY^{10}$. For example,

- $Z_0 = BY_1Y_2 \cdots Y_{10}$;
- $Z_1 = BY_{11}Y_{12} \cdots Y_{20}$;
- $\cdots$;
- $Z_{10} = BY_{101}Y_{102} \cdots Y_{110}$.

The B at the beginning of each Z_i is an additional B from the previous stage.

Here,

$$\frac{111}{999} = \frac{11}{99},$$

and ties at this stage are broken in favor of Y. Therefore, Y_{111} appears before the corresponding B, and after grouping the sequence becomes

$$Y_0, Z_0, Z_1, \ldots, Z_{10}, Y_{111}, Z_{11}, \ldots$$

This is the merging problem for $(1, 2)$, with the shorter-digit side replaced by Y and the longer-digit side replaced by Z. Ties are broken in favor of the shorter-digit side, so this reduces to $F_0(1, 2; Y, Z)$.

Finally, for $(a, b) = (1, 2)$, we have $99 = 11 \cdot 9$. The remainder is 0, so the recursion terminates. Since $N_1 = 9$,

$$F_0(1, 2; Y, Z) = (YZ^{11})^8 YZ.$$

Working backwards, we obtain

$$F_1(2, 3; B, Y) = F_0(1, 2; Y, Z)B,$$
$$F_0(3, 5; A, B) = F_1(2, 3; B, Y)AB^9.$$

Restoring the final B that was removed, the entire merged sequence is

$$(YZ^{11})^8 YZBAB^{10},$$

where $Y = AB^{100}$ and $Z = BY^{10}$.

In this way, we reduce the merging problem as

$$(3, 5) \longrightarrow (2, 3) \longrightarrow (1, 2)$$

and combine the resulting blocks in reverse order, representing the original sequence without constructing it explicitly.

Evaluating the Enormous Sequence

The actual merged sequence is extremely long, so we need to evaluate the concatenation of blocks efficiently.

Let z be the current concatenation, x the next D_1-digit integer to be used, and y the next D_2-digit integer to be used. A single `A` or `B` represents the transformation

$$(z, x, y) \longmapsto (10^{D_1} z + x,\ x - 1,\ y)$$

or

$$(z, x, y) \longmapsto (10^{D_2} z + y,\ x,\ y - 1),$$

respectively.

Transformations of this form are closed under composition. Thus, any block S with total digit length ℓ_S can be represented as

$$(z, x, y) \longmapsto \left(10^{\ell_S} z + \alpha_S x + \beta_S y + \gamma_S,\ x - c_{A,S},\ y - c_{B,S}\right).$$

It therefore suffices to store the six values

$$(c_A, c_B, 10^{\ell}, \alpha, \beta, \gamma)$$

modulo 998244353 for each block. Two blocks can be concatenated in $O(1)$ time, and repetitions of a block can be evaluated by binary exponentiation.

Handling Enormous Repetition Counts

Let $P = 998244353$. The matrix representing the transformation for a block S has diagonal entries

$$(10^{\ell_S}, 1, 1, 1).$$

By Fermat's little theorem,

$$(10^{\ell_S})^{P-1} = 1.$$

Thus, the matrix representing S^{P-1} can be written as $I + N$, where N is a strictly upper triangular 4×4 matrix, so $N^4 = 0$. Modulo P,

$$(I + N)^P = I + N^P = I.$$

Therefore, for every block S, the transformation represented by $S^{P(P-1)}$ is the identity.

Consequently, setting $L = P(P-1)$, we only need the repetition counts, such as K, t, and $N_a - 1$, modulo L.

Let

$$R = 10^a, \qquad G_n = 1 + R + \cdots + R^{n-1}.$$

Then $K = 10^r G_q$. Using the composition rule

$$(R^{n+m}, G_{n+m}) = (R^n R^m,\ G_n + R^n G_m),$$

we can compute K mod L by binary exponentiation. Also,

$$t = \begin{cases} 10^{r-1} G_q & (r > 0), \\ 1 + 10^{a-1} G_{q-1} & (r = 0), \end{cases}$$

so t mod L can be computed in the same way.

For multiplication modulo L, use a type such as `__int128` to avoid overflow. Alternatively, compute modulo P and modulo $P-1$ separately and combine the results using the Chinese remainder theorem.

Finally, append one `B` to $F(D_1, D_2; A, B)$ and apply the resulting transformation to the initial state

$$z = 0, \qquad x = 10^{D_1} - 1, \qquad y = 10^{D_2} - 1$$

to obtain the answer.

Complexity

There are $O(\log D_2)$ recursive steps, each taking $O(\log D_2)$ time with binary exponentiation. Therefore, the time complexity per test case is $O(\log^2 D_2)$, and the space complexity is $O(\log D_2)$.

Problem K. Dyads of Hanoi

Properties of Shortest Paths in the Tower of Hanoi

We identify a Tower of Hanoi state with the string representing it. For a state X, let X_k denote the peg containing disk k. Also, let $X_{<k}$ denote the state of the Tower of Hanoi consisting only of disks $1, \ldots, k-1$.

Let $d(X, Y)$ be the minimum number of moves required to transform state X into state Y. We establish several properties of $d(X, Y)$.

If all disks in X are on the same peg x, we write $d(x, Y)$ for $d(X, Y)$. If all disks in Y are also on the same peg y, we write $d_N(x, y)$ for $d(X, Y)$.

Property 1. We have $d(x, Y) \le 2^N - 1$, and there is exactly one sequence of $d(x, Y)$ moves. Moreover, if $x \neq y$, then $d_N(x, y) = 2^N - 1$.

Proof. We prove both claims simultaneously by induction on N. Both are trivial for $N = 1$. Suppose that $N > 1$ and that both claims hold for $N - 1$ disks.

First, we show that $d(x, Y) \le 2^N - 1$.

If $Y_N = x$, we can leave disk N in place and move only disks $1, \ldots, N-1$. By the induction hypothesis, this requires at most $2^{N-1} - 1$ moves.

If $Y_N \neq x$, use the following sequence:

- Move disks $1, \ldots, N-1$ to peg z, where z is distinct from both x and Y_N.
- Move disk N to peg Y_N.
- Transform the state of disks $1, \ldots, N-1$ into $Y_{<N}$.

By the induction hypothesis, this requires at most

$$(2^{N-1} - 1) + 1 + (2^{N-1} - 1) = 2^N - 1$$

moves. Thus, $d(x, Y) \le 2^N - 1$ in both cases.

Next, we show that an optimal sequence moves disk N at most once. Suppose that an optimal sequence moves disk N at least twice.

If the first two moves of disk N return it to peg x, we can shorten the sequence by removing these two moves while keeping all intervening moves of smaller disks. This contradicts optimality.

It therefore suffices to consider the case where the first two moves of disk N do not return it to peg x. In this case, the following steps are necessary:

- Move disks $1, \ldots, N-1$ to some peg a with $a \neq x$.
- Move disk N to the peg distinct from both x and a.
- Move disks $1, \ldots, N-1$ to peg x.
- Move disk N to peg a.

By the induction hypothesis, the number of moves required up to this point is at least

$$\begin{aligned} d_{N-1}(x,a) + 1 + d_{N-1}(a,x) + 1 &= (2^{N-1}-1) + 1 + (2^{N-1}-1) + 1 \\ &= 2^N. \end{aligned}$$

This contradicts $d(x, Y) \leq 2^N - 1$. Hence, an optimal sequence moves disk N at most once.

If $Y_N = x$, disk N is never moved, so the problem reduces to disks $1, \ldots, N-1$. By the induction hypothesis, the optimal sequence is unique.

If $Y_N \neq x$, disk N is moved exactly once, so an optimal sequence must have the form described earlier. By the induction hypothesis, the optimal sequences for moving disks $1, \ldots, N-1$ before and after this move are both unique. Therefore, the entire optimal sequence is unique.

This proves the first claim.

Finally, suppose that all disks in Y are on peg y, where $y \neq x$. Using the peg z distinct from both x and y, the optimal sequence has the following form:

- Move disks $1, \ldots, N-1$ to peg z.
- Move disk N to peg y.
- Move disks $1, \ldots, N-1$ to peg y.

By the induction hypothesis,

$$\begin{aligned} d_N(x,y) &= d_{N-1}(x,z) + 1 + d_{N-1}(z,y) \\ &= 2(2^{N-1}-1) + 1 \\ &= 2^N - 1. \end{aligned}$$

This proves the second claim and completes the induction. □

Next, consider an arbitrary initial state X.

Property 2. If $X_N = Y_N$, disk N is never moved in a sequence attaining $d(X, Y)$.

Proof. Let $a = X_N$, and suppose that disk N is moved.

Moving disk N along $a \to b \to a$ is clearly unnecessary, so it suffices to consider the case where it moves along $a \to b \to c \to a$. Such a sequence must include at least the following steps:

- Move disks $1, \ldots, N-1$ to peg c.
- Move disk N from peg a to peg b.
- Move disks $1, \ldots, N-1$ to peg a.
- Move disk N from peg b to peg c.
- Move disks $1, \ldots, N-1$ to peg b.
- Move disk N from peg c to peg a.
- Transform the state of disks $1, \ldots, N-1$ into $Y_{<N}$.

This requires at least

$$1 + d_{N-1}(c, a) + 1 + d_{N-1}(a, b) + 1$$

moves. Since

$$d_{N-1}(c, a) = d_{N-1}(a, b) = 2^{N-1} - 1,$$

the number of moves is at least

$$2(2^{N-1} - 1) + 3 = 2^N + 1.$$

On the other hand, we can leave disk N in place and transform $X_{<N}$ into $Y_{<N}$ using only the smaller disks. First gather disks $1, \ldots, N-1$ on a single peg, then transform their state into $Y_{<N}$. By Property 1 and the reversibility of moves, this requires at most

$$2(2^{N-1} - 1) = 2^N - 2$$

moves. Therefore, a sequence that moves disk N cannot be optimal. □

By this property, if k is the largest index such that $X_k \neq Y_k$, we only need to consider disks $1, \ldots, k$.

Property 3. Suppose that $X_N \neq Y_N$, and let z be the peg distinct from both X_N and Y_N. Every sequence attaining $d(X, Y)$ follows one of the two patterns below. Either pattern, or both, may attain the minimum.

Pattern 1

- Move disks $1, \ldots, N-1$ to peg z.
- Move disk N to peg Y_N.
- Transform the state of disks $1, \ldots, N-1$ into $Y_{<N}$.

Pattern 2

- Move disks $1, \ldots, N-1$ to peg Y_N.
- Move disk N to peg z.
- Move disks $1, \ldots, N-1$ to peg X_N.
- Move disk N to peg Y_N.
- Transform the state of disks $1, \ldots, N-1$ into $Y_{<N}$.

Proof. Among sequences that move disk N exactly once or exactly twice, the optimal sequences clearly follow Pattern 1 or Pattern 2, respectively. It therefore suffices to show that an optimal sequence never moves disk N three or more times.

As in the proof of Property 2, it suffices to consider a sequence in which disk N moves along $a \to b \to c \to a \to \cdots$. Such a sequence requires at least $2^N + 1$ moves.

On the other hand, Pattern 1 requires

$$d(X_{<N}, z) + 1 + d(z, Y_{<N}) \leq 2^N - 1$$

moves. Hence, a sequence that moves disk N three or more times cannot be optimal. □

There are exactly two sequences attaining $d(X, Y)$ if and only if Patterns 1 and 2 require the same number of moves.

Let k be the largest index such that $X_k \neq Y_k$, and let z be the peg distinct from both X_k and Y_k. The condition is

$$\begin{aligned} &d(X_{<k}, z) + 1 + d(Y_{<k}, z) \\ &\quad = d(X_{<k}, Y_k) + 1 + d_{k-1}(X_k, Y_k) + 1 + d(Y_{<k}, X_k). \end{aligned}$$

Here, we have used the symmetry relations

$$d(z, Y_{<k}) = d(Y_{<k}, z), \qquad d(X_k, Y_{<k}) = d(Y_{<k}, X_k).$$

Since $d_{k-1}(X_k, Y_k) = 2^{k-1} - 1$, rearranging gives

$$\big(d(X_{<k}, Y_k) - d(X_{<k}, z)\big) + \big(d(Y_{<k}, X_k) - d(Y_{<k}, z)\big) + 2^{k-1} = 0.$$

Define

$$\Delta(X, p, q) = d(X, q) - d(X, p) \qquad (p \neq q).$$

The recursive structure of d gives

$$\Delta(X_{<m+1}, p, q) = C(X_m, p, q)2^{m-1} + \Delta(X_{<m}, p', q').$$

The terms in this recurrence are defined as follows:

- $C(x, p, q)$ is the difference in costs: it is 1 if $x = p$, -1 if $x = q$, and 0 if $x \neq p, q$.
- The new target pair (p', q') is uniquely determined by x. Specifically, if $x = p$, then $p' = p$; otherwise, p' is the peg distinct from both x and p. The rule for q' is analogous.

Applying this recurrence repeatedly expresses

$$\Delta(X_{<k}, z, Y_k) + \Delta(Y_{<k}, z, X_k) + 2^{k-1}$$

as a sum of powers of two:

$$\Delta(X_{<k}, z, Y_k) + \Delta(Y_{<k}, z, X_k) + 2^{k-1} = 2^{k-1} + \sum_{m=1}^{k-1} c_m 2^{m-1},$$

where

$$c_m = C(X_m, p_m, q_m) + C(Y_m, r_m, s_m) \in \{-2, -1, 0, 1, 2\}.$$

Counting with Digit DP

We count pairs of strings X, Y satisfying

$$2^{k-1} + \sum_{m=1}^{k-1} c_m 2^{m-1} = 0.$$

Process the disks in decreasing order of size, from $k-1$ down to 1, using digit DP to maintain

$$V = 2^{k-i} + \sum_{m=i}^{k-1} c_m 2^{m-i}.$$

Initially, $V = 1$. At each bit m, update it as follows:

$$V_{\text{next}} = 2V + c_m.$$

Our goal is to have $V = 0$ after processing $m = 1$.

Since $c_m \in \{-2, -1, 0, 1, 2\}$, once $V \notin \{-1, 0, 1\}$, no subsequent transitions can bring it back to 0. Thus, it suffices to keep only the three values $V \in \{-1, 0, 1\}$.

The DP state is

$$\mathrm{dp}[x][y][V],$$

where:

- x is the ordered pair of targets for X, with 6 possibilities.
- y is the ordered pair of targets for Y, with 6 possibilities.
- V is the current difference, with 3 possibilities: -1, 0, and 1.

We also maintain a count c of assignments for which no disk with $X_k \neq Y_k$ has been found yet. Scan the strings from position N down to position 1, applying the following updates:

- For choices of characters with $A_m = B_m$, update c.
- For choices of characters with $A_m \neq B_m$, the paths branch at this position. Add the c assignments to the corresponding initial DP state with $V = 1$.
- For DP states in which the largest index k with $X_k \neq Y_k$ has already been determined, consider each choice of characters for A_m and B_m. Compute the cost c_m and the new target pairs for X and Y, and update the state.

The time complexity is $O(N)$, with a constant factor of approximately $6 \times 6 \times 3 \times 9 = 972$, which is sufficiently fast.

Problem L. Consecutive Pairs

Approach

Let $F(l, r)$ denote the response to the query on the interval $[l, r]$.

Consider the graph G on the vertices $1, 2, \ldots, N$ (positions), in which vertices i and j are joined by an edge if $|P_i - P_j| = 1$. Writing $\mathrm{pos}(x)$ for the position of value x, we see that G is a single path:

$$\mathrm{pos}(1) - \mathrm{pos}(2) - \cdots - \mathrm{pos}(N).$$

Moreover, $F(l, r)$ equals the number of edges of G with both endpoints in $[l, r]$. Thus, our goal is to reconstruct all $N - 1$ edges of G.

Process the right endpoint r in the order $2, 3, \ldots, N$. Let $F(1, 1) = 0$, which requires no query, and define

$$d_r = F(1, r) - F(1, r - 1).$$

Then d_r is the number of edges whose larger endpoint is exactly r. Since G is a path, $d_r \in \{0, 1, 2\}$.

Next, we find the smaller endpoints of these edges. Define

$$H_r(l) = F(l, r) - F(l, r - 1).$$

Then $H_r(l)$ is the number of edges joining vertex r to a vertex in $[l, r - 1]$.

For example, suppose that there are two edges (a, r) and (b, r) with larger endpoint r, where $a < b$. Then $H_r(l)$ is 2 when $l \leq a$, 1 when $a < l \leq b$, and 0 when $b < l$.

Therefore, the largest l with $H_r(l) \geq 1$ is b, and the largest l with $H_r(l) \geq 2$ is a. Since $H_r(l)$ is non-increasing in l, we can find these positions by binary search. The case $d_r = 1$ is handled in the same way.

A direct implementation of this method requires two queries, $F(l, r)$ and $F(l, r - 1)$, for each step of the binary search. This gives an upper bound of

$$(N - 1) + 2(N - 1)\lceil \log_2 N \rceil$$

queries, which exceeds the limit.

Here, we exploit the fact that r is processed in increasing order. When we start processing r, all edges with both endpoints in $[1, r-1]$ have already been reconstructed.

Thus, $F(l, r-1)$, the number of edges with both endpoints in $[l, r-1]$, can be computed from the reconstructed edges without asking a query.

Consequently, computing $H_r(l) = F(l, r) - F(l, r-1)$ requires only one query, namely $F(l, r)$, so each step of the binary search uses just one query.

Once all edges have been reconstructed, the two endpoints of the path G are $\mathrm{pos}(1)$ and $\mathrm{pos}(N)$. By the guarantee $\mathrm{pos}(1) < \mathrm{pos}(N)$, the endpoint with the smaller index is $\mathrm{pos}(1)$.

Let $v_1, v_2, \ldots, v_N$ be the vertices in the order they are visited when traversing the path from this endpoint. Then the permutation is recovered by setting $P_{v_i} = i$.

Algorithm

1. Query $F(1, r)$ for $r = 2, 3, \ldots, N$.
2. For $r = 2, 3, \ldots, N$, do the following:
   - Compute $d_r = F(1, r) - F(1, r-1)$.
   - For each $t = 1, \ldots, d_r$, find the largest l with $H_r(l) \geq t$ by binary search. At each step, query $F(l, r)$ and compute $F(l, r-1)$ from the edges already reconstructed.
   - Add an edge between r and each position l found.
3. Traverse the reconstructed path starting from the endpoint with the smaller index, assigning the values $1, 2, \ldots, N$ in order.

Complexity

The initial queries for $F(1, r)$ use $N-1$ queries.

Each edge is counted in exactly one d_r, namely for r equal to its larger endpoint. Therefore,

$$\sum_{r=2}^{N} d_r = N - 1,$$

so we perform exactly $N-1$ binary searches in total.

Each binary search uses at most $\lceil \log_2 N \rceil$ queries. Hence, the total number of queries is at most

$$(N-1) + (N-1)\lceil \log_2 N \rceil.$$

Since $N \leq 2000$, we have $\lceil \log_2 N \rceil \leq 11$, so

$$(N-1) + (N-1)\lceil \log_2 N \rceil \leq 12(N-1) < 12N.$$

Thus, the query limit is satisfied.

As for the computation outside the queries, even a straightforward implementation that maintains the reconstructed edges runs in $O(N^2)$ time and $O(N)$ space, which is fast enough.

Problem M. I hate prime numbers

Reduction to Formal Power Series

Instead of imposing $\max(B) = m$, we count pairs satisfying $\max(B) \le m$ and take consecutive differences at the end.

Let $C_{n,m}$ be the number of good pairs of sequences of length n satisfying $\max(B) \le m$. For convenience, define $C_{0,m} = 1$ for every $m \ge 0$.

First, consider the number of choices of A for a fixed B. For each b, let n_b be the number of indices i such that $B_i = b$, and let S_b be the number of indices i such that $B_i \ge b$.

For an index i with $B_i = b$, we may choose $A_i = j$ exactly when $B_j \ge b$, so there are S_b possible values. Moreover, for each j, the number of indices i with $B_i = b$ and $A_i = j$ must not be prime.

Define the exponential generating function of the nonnegative integers that are not prime:

$$P(X) = \sum_{\substack{k \ge 0 \\ k \text{ is not prime}}} \frac{X^k}{k!}.$$

Then the number of ways to choose A_i for all indices i with $B_i = b$ is

$$\left[\frac{X^{n_b}}{n_b!}\right] P(X)^{S_b}.$$

For fixed $(n_1, \ldots, n_m)$, the number of possible sequences B is

$$\frac{n!}{\prod_{b=1}^{m} n_b!}.$$

Therefore,

$$\frac{C_{n,m}}{n!} = \sum_{n_1 + \cdots + n_m = n} \prod_{b=1}^{m} [X^{n_b}] P(X)^{S_b}.$$

The transition from $C_{s,m}/s!$ to $C_{t,m+1}/t!$ has coefficient

$$[X^{t-s}] P(X)^t.$$

Let $Q(X)$ be the compositional inverse of $X/P(X)$. By the Lagrange inversion formula,

$$[X^{t-s}] P(X)^t = [X^t] \frac{XQ'(X)}{Q(X)} Q(X)^s.$$

Thus, defining

$$F_m(X) = \sum_{n=0}^{\infty} \frac{C_{n,m}}{n!} X^n,$$

we obtain

$$F_{m+1}(X) = \frac{XQ'(X)}{Q(X)} F_m(Q(X)).$$

It follows by induction that

$$F_m(X) = \frac{X(Q^{\langle m \rangle})'(X)}{Q^{\langle m \rangle}(X)},$$

where $Q^{\langle m \rangle}(X)$ denotes the m-fold compositional iterate of $Q(X)$.

Indeed, the claim is immediate for $m = 0$. If it holds for m, then

$$\begin{aligned} F_{m+1}(X) &= \frac{XQ'(X)}{Q(X)} F_m(Q(X)) \\ &= \frac{XQ'(X)}{Q(X)} \frac{Q(X)(Q^{\langle m \rangle})'(Q(X))}{Q^{\langle m+1 \rangle}(X)} \\ &= \frac{X(Q^{\langle m+1 \rangle})'(X)}{Q^{\langle m+1 \rangle}(X)}, \end{aligned}$$

which proves the claim for $m + 1$.

We have therefore reduced the problem to computing

$$[X^N] \frac{X(Q^{\langle m \rangle})'(X)}{Q^{\langle m \rangle}(X)}$$

for $m = 1, \ldots, M$.

Computation Using Conjugacy

There exists a formal power series $G(X) = X + O(X^2)$ such that

$$Q = G \circ \frac{X}{1 - X} \circ G^{-1}.$$

We will prove its existence and explain how to compute it later. First, suppose that G is already known. The m-fold compositional iterate of $X/(1 - X)$ is $X/(1 - mX)$. Hence,

$$Q^{\langle m \rangle} = G \circ \frac{X}{1 - mX} \circ G^{-1},$$

and therefore

$$\frac{X(Q^{\langle m \rangle})'(X)}{Q^{\langle m \rangle}(X)} = \frac{X\left(G\left(\frac{G^{-1}(X)}{1-mG^{-1}(X)}\right)\right)'}{G\left(\frac{G^{-1}(X)}{1-mG^{-1}(X)}\right)}.$$

Applying Lagrange inversion gives

$$[X^N] \frac{X\left(G\left(\frac{G^{-1}(X)}{1-mG^{-1}(X)}\right)\right)'}{G\left(\frac{G^{-1}(X)}{1-mG^{-1}(X)}\right)} = [X^N] A(X)^N B\left(\frac{X}{1 - mX}\right) \frac{1}{1 - mX},$$

where

$$A(X) = \frac{X}{G(X)}, \qquad B(X) = \frac{XG'(X)}{G(X)}.$$

Write

$$A(X)^N = \sum_k a_k X^k, \qquad B(X) = \sum_k b_k X^k.$$

Since

$$\begin{aligned} [X^l] B\left(\frac{X}{1 - mX}\right) \frac{1}{1 - mX} &= \sum_k b_k [X^l] \frac{X^k}{(1 - mX)^{k+1}} \\ &= \sum_k b_k \binom{l}{k} m^{l-k}, \end{aligned}$$

we obtain

$$[X^N] A(X)^N B\left(\frac{X}{1 - mX}\right) \frac{1}{1 - mX} = \sum_l \sum_k a_{N-l} b_k \binom{l}{k} m^{l-k}.$$

Thus, the coefficient of m^k is

$$\sum_{i-j=k} \binom{i}{j} a_{N-i} b_j.$$

These values can be computed for all $k = 0, \ldots, N$ by convolving suitably chosen sequences. This gives the coefficients of

$$[X^N] \frac{X(Q^{\langle m \rangle})'(X)}{Q^{\langle m \rangle}(X)}$$

as a polynomial in m.

Finally, multipoint evaluation gives its values at $m = 1, \ldots, M$. If $N \leq M$, we can improve the complexity by first evaluating it at $m = 1, \ldots, N+1$ and then computing the remaining values using a suitable convolution.

Excluding the computation of G, this solves the original problem in

$$O(N \log^2 N + M \log M)$$

time.

Computing G

It remains to solve the following general problem:

Given a formal power series

$$F(X) = X + X^2 + X^3 + O(X^4),$$

find a formal power series G such that

$$G^{-1} \circ F \circ G(X) \equiv \frac{X}{1-X} \pmod{X^N}.$$

We describe an $O(N \log^2 N)$-time algorithm based on Newton's method.

Suppose that, for some $n \geq 4$, we have a series G_0 satisfying

$$G_0^{-1} \circ F \circ G_0(X) \equiv \frac{X}{1-X} \pmod{X^n}.$$

It suffices to find a series G satisfying

$$G^{-1} \circ F \circ G(X) \equiv \frac{X}{1-X} \pmod{X^{2n-3}}$$

in $O(n \log^2 n)$ time.

Replacing F by $G_0^{-1} \circ F \circ G_0$ and G by $G_0^{-1} \circ G$, we may assume that

$$F(X) \equiv \frac{X}{1-X} \pmod{X^n}.$$

These transformations can be computed in $O(n \log^2 n)$ time.

Write

$$G(X) = X + X^{n-1} g(X).$$

Then

$$G^{-1}(X) \equiv X - X^{n-1} g(X) \pmod{X^{2n-3}}.$$

Indeed,

$$\begin{aligned} G(X - X^{n-1}g(X)) &= X - X^{n-1}g(X) \\ &\quad + (X - X^{n-1}g(X))^{n-1}g(X - X^{n-1}g(X)) \\ &\equiv X - X^{n-1}g(X) + X^{n-1}g(X) \\ &\equiv X \pmod{X^{2n-3}}. \end{aligned}$$

Also write

$$F(X) = \frac{X}{1-X} + X^n H(X).$$

Noting that

$$(G^{-1})'(X) \equiv 1 \pmod{X^{n-2}},$$

we obtain

$$\begin{aligned} G^{-1} \circ F \circ G(X) &= G^{-1}\left(\frac{G(X)}{1-G(X)} + G(X)^n H(G(X))\right) \\ &\equiv G^{-1}\left(\frac{G(X)}{1-G(X)}\right) \\ &\quad + G(X)^n H(G(X))(G^{-1})'\left(\frac{G(X)}{1-G(X)}\right) \\ &\equiv \frac{G(X)}{1-G(X)} - \left(\frac{G(X)}{1-G(X)}\right)^{n-1} g\left(\frac{G(X)}{1-G(X)}\right) + X^n H(X) \\ &\equiv \frac{X}{1-X} + \frac{X^{n-1}g(X)}{(1-X)^2} - \left(\frac{X}{1-X}\right)^{n-1} g\left(\frac{X}{1-X}\right) + X^n H(X) \pmod{X^{2n-3}}. \end{aligned}$$

Consequently, the condition

$$G^{-1} \circ F \circ G(X) \equiv \frac{X}{1-X} \pmod{X^{2n-3}}$$

is equivalent to

$$\frac{X^{n-1}g(X)}{(1-X)^2} - \left(\frac{X}{1-X}\right)^{n-1} g\left(\frac{X}{1-X}\right) + X^n H(X) \equiv 0 \pmod{X^{2n-3}}.$$

Dividing by $(X/(1-X))^{n-1}$ gives

$$(1-X)^{n-3}g(X) - g\left(\frac{X}{1-X}\right) + X(1-X)^{n-1}H(X) \equiv 0 \pmod{X^{n-2}}.$$

Write

$$g(X) = \sum_{i \geq 0} c_i X^i.$$

Expanding each term on the left and comparing coefficients allows us to determine the c_i in increasing order of degree. Specifically, the contributions of the lower-degree terms of g to the coefficient of X^i on the left can be computed using suitable convolutions.

Thus, divide-and-conquer FFT, also known as online convolution, computes g in $O(n \log^2 n)$ time.

This allows us to compute the first N terms of $G(X)$ in $O(N \log^2 N)$ time.

Problem N. Delete Sad Faces (Hard)

Relaxing the operation

We first consider *tatesoto*'s problem for a single string.

Consider the following relaxed problem (P): perform the following operation zero or more times until the string contains no sad faces, and maximize the number of happy faces in the resulting string.

- Choose any emoticon in the current string and delete its two characters, joining the remaining characters without changing their order.

Unlike the original operation, this operation allows us to delete happy faces as well. We will prove that this relaxation does not change the answer.

First, a string can be completely deleted using the relaxed operation if and only if the number of colons equals the total number of opening and closing parentheses.

Necessity follows because each operation deletes one colon and one parenthesis. Conversely, any nonempty string containing equally many colons and parentheses contains an adjacent pair consisting of one of each, which is an emoticon. Repeatedly deleting such a pair deletes the entire string.

Let A be the answer to the original problem and B the answer to (P). Every operation allowed in the original problem is also allowed in (P), so $A \leq B$. We will prove $A \geq B$.

Suppose that (P) produces a string $T = t_1 t_2 \cdots t_k$ containing no sad faces and exactly B happy faces. Write the original string as

$$S = X_0 t_1 X_1 t_2 \cdots t_k X_k.$$

Each X_i is a substring that was completely deleted. Since the surviving characters separate these substrings, each X_i contains equally many colons and parentheses.

For each X_i, repeatedly delete only sad faces until none remain, and let R_i be the resulting string. Each R_i also contains equally many colons and parentheses and contains no sad faces.

For a string W, let $a(W)$ and $b(W)$ be its numbers of happy and sad faces, respectively, and define

$$F(W) = a(W) - b(W).$$

A case analysis on the characters immediately before and after a deleted sad face shows that deleting one sad face changes F by 0, 1, or 2. In particular, deleting a sad face never decreases F.

We use the following lemma.

Lemma. Let R be a string containing equally many colons and parentheses and containing no sad faces. Inserting R between two adjacent characters that do not form a sad face, or at either end of a string, does not decrease F.

Proof. The claim is immediate if R is empty, so assume that R is nonempty.

Classify characters into two types: colons and parentheses. The string R contains both types, and every adjacent pair where the type changes is a happy face. Therefore,

$$F(R) \geq \begin{cases} 2 & \text{if the first and last characters have the same type,} \\ 1 & \text{if the first and last characters have different types.} \end{cases} \tag{1}$$

Suppose that we insert R between adjacent characters x, y that do not form a sad face. Let a and b be the first and last characters of R, respectively. Define $w(u, v)$ to be the contribution of an adjacent pair uv to F: it is 1 for a happy face, -1 for a sad face, and 0 otherwise.

Let

$$L = w(x, y) - w(x, a) - w(b, y)$$

be the loss in F at the boundaries.

The pair xy must be one of `::`, `:)`, `(:`, `((`, `()`, `)(`, and `))`. For each possible pair, the maximum value of L under each condition on the types of a, b is as follows:

xy	`::`	`:)`	`(:`	`((`	`()`	`)(`	`))`
a, b have the same type	2	2	2	0	0	2	0
a, b have different types	1	1	1	1	-1	1	1

For example, when $xy =$ `:)` and a, b have the same type, choosing $a =$ `(` and $b =$ `)` gives

$$L = 1 - (-1) - 0 = 2,$$

which is the maximum.

Thus,

$$L \le \begin{cases} 2 & \text{if } a, b \text{ have the same type,} \\ 1 & \text{if } a, b \text{ have different types.} \end{cases} \tag{2}$$

By (1) and (2), $F(R) \ge L$, so

$$F(xRy) - F(xy) = F(R) - L \ge 0.$$

If R is inserted at either end of the string, the boundary loss is at most 1, while $F(R) \ge 1$. The same conclusion follows. □

Now let

$$U = R_0 t_1 R_1 t_2 \cdots t_k R_k.$$

We can obtain U from S using only operations allowed in the original problem.

We can also construct U by inserting the strings R_i into the corresponding gaps of T. Before each insertion into an internal gap, the two characters bordering that gap do not form a sad face, since T contains no sad faces. By the lemma,

$$F(U) \ge F(T) = B.$$

If T is empty, this inequality follows from $F(R_0) \ge 0$.

Finally, delete sad faces from U until none remain, obtaining a string V. These deletions never decrease F, and V contains no sad faces. Consequently,

$$A \ge F(V) \ge F(U) \ge B.$$

Hence $A = B$, proving that we may use the relaxed operation.

Prefix sums

We now solve the relaxed problem for a string T.

Assign value 1 to each `(` and `)`, and value -1 to each `:`. Define

$$A_0 = 0, \qquad A_i = \sum_{j=1}^{i} \begin{cases} 1 & (T_j = \texttt{(} \text{ or } \texttt{)}), \\ -1 & (T_j = \texttt{:}). \end{cases}$$

Each operation deletes a 1 and a -1. Thus, it preserves the total sum, as well as the prefix-sum value immediately after each surviving character.

Let $i_1 < i_2 < \cdots < i_K$ be the original indices of the parentheses that contribute to happy faces in the final string.

Since the final string contains no sad faces, these parentheses alternate between `(` and `)`. Between a selected `(` and the following selected `)`, the final string contains one or more colons and no parentheses. Between a selected `)` and the following selected `(`, it contains zero or more parentheses and no colons.

Considering the corresponding prefix sums gives the following conditions:

- If $T_{i_1} =$ `(`, then $A_{i_1} > A_0 = 0$.
- If $T_{i_1} =$ `)`, then $A_{i_1} \le A_0$.
- For $1 \le j < K$, if $T_{i_j} =$ `(`, then $T_{i_{j+1}} =$ `)` and $A_{i_j} \ge A_{i_{j+1}}$.
- For $1 \le j < K$, if $T_{i_j} =$ `)`, then $T_{i_{j+1}} =$ `(` and $A_{i_j} < A_{i_{j+1}}$.

- If $T_{i_K} =$ `(`, then $A_{i_K} > A_{|T|}$.
- If $T_{i_K} =$ `)`, then $A_{i_K} \leq A_{|T|}$.

Conversely, suppose that a sequence of indices satisfies these conditions. By deleting emoticons as long as possible within the gaps separated by the selected characters, we can make all K selected parentheses contribute to happy faces.

Therefore, the problem reduces to finding the maximum length of an index sequence satisfying these conditions.

A greedy algorithm

For each index i, let next_i be the smallest index $j > i$ satisfying either of the following conditions:

- $T_j =$ `)` and $A_j \leq A_i$.
- $T_j =$ `(` and $A_j > A_i$.

If no such index exists, let $\text{next}_i = \infty$. This definition also applies to $i = 0$.

We can construct a maximum-length index sequence I as follows:

1. Initialize $I = (\text{next}_0)$. If $\text{next}_0 = \infty$, return the empty sequence.
2. Repeat the following:
   - Let j be the last index in I, and let $k = \text{next}_j$.
   - If $k = \infty$, proceed to the final step.
   - If $T_j = T_k$, replace the last index of I with k.
   - Otherwise, append k to I.
3. Let j be the last index in I. If either $T_j =$ `(` and $A_j \leq A_{|T|}$, or $T_j =$ `)` and $A_j > A_{|T|}$, remove j from I.

The greedy choices are justified by the following preferences:

- Between sequences of different lengths, a longer sequence is preferable.
- Between sequences of equal length ending in `)`, a smaller prefix-sum value at the last index is preferable. If they end in `(`, a larger value is preferable.

Thus, we can follow the next pointers, initially count the first selected character, and increase the count by one whenever we move to a different character. After applying the final endpoint adjustment, this count is the answer to *tatesoto*'s problem.

Answering interval queries

The same argument applies to interval queries.

Compute the prefix sums A_i and pointers next_i for the entire string S. For a query $[L, R]$, start at $L - 1$ and follow the next pointers while the next index does not exceed R.

The first visited index initializes the selected sequence. Subsequent transitions between different characters increase its length by one, while transitions between equal characters replace its last index. Finally, apply the endpoint adjustment using A_R.

The initial prefix-sum value is now A_{L-1} instead of 0. Since all comparisons are unchanged by subtracting this value, the same pointers can be used for every query.

Complexity

All values next_i can be computed in $O(|S| \log |S|)$ time using a segment tree or a similar data structure.

Using binary lifting, we can preprocess the pointer jumps and the numbers of transitions between different characters in $O(|S| \log |S|)$ time. Each query then takes $O(\log |S|)$ time.

Therefore, the total time complexity is

$$O((|S| + Q) \log |S|).$$