Problem A. Astana Hard Metro

Root the tree at vertex 1. Let the number of children of vertex v be c_v. Its transfer complexity is

$$P(v) = \max(1, c_v).$$

First, consider a fixed final tree. Let

$$X = \prod_v P(v),$$

and suppose the tree has z leaves.

For a leaf l, let R_l be Pupa's result if she starts from l. After dividing R_l by X, all vertices outside the path disappear, so

$$\frac{R_l}{X} = \frac{1}{\prod P(u)},$$

where the product is taken over all proper ancestors of l.

Consider a random walk starting at the root. At every non-leaf vertex, choose one of its children uniformly. The probability that the walk ends at l is exactly the expression above. Since the probabilities of reaching all leaves sum to 1,

$$\sum_l R_l = X.$$

Pupa chooses a leaf uniformly, therefore the expected result for this fixed tree is

$$\frac{X}{z}.$$

Thus, we need to find the expected value of the product of all transfer complexities divided by the number of leaves.

To solve this problem, we will use the Product Trick.

Represent every vertex by a box containing one cookie for each of its children. When a new vertex chooses its parent, one cookie is added to the parent's box, and a new empty box is created.

Empty boxes correspond to leaves, while

$$X = \prod_{\text{nonempty boxes}} (\text{number of cookies}).$$

Use the product trick: in every nonempty box, choose exactly one cookie and color it red; all other cookies are white. A box with k cookies has k possible choices for its red cookie, so the total number of valid colorings is exactly X.

During the process, a nonempty box may still contain no red cookie. Call such a box waiting.

Let

$$dp_j[w][z]$$

be the number of ways to perform the first j additions and color all cookies such that there are exactly w waiting boxes and z empty boxes. Every other nonempty box must contain exactly one red cookie.

Let p_i be the initial number of children of vertex i, and let z_0 be the number of initial leaves.

To initialize the DP, process all initial nonempty boxes. For a box containing p_i cookies, either choose one of them as red in p_i ways, or leave all of them white and make the box waiting.

Using an auxiliary DP f, initially

$$f[0] = 1,$$

and for every $p_i > 0$,

$$f_{\text{new}}[w] = p_i f[w] + f[w-1].$$

After processing all initial vertices,

$$dp_0[w][z_0] = f[w].$$

Now process the new vertices. For $j = 1, \ldots, m$:

$$\begin{aligned} dp_j[w][z] = {} & z \cdot dp_{j-1}[w-1][z] \\ & + (n+j-z) \cdot dp_{j-1}[w][z-1] \\ & + z \cdot dp_{j-1}[w][z] \\ & + (w+1) \cdot dp_{j-1}[w+1][z-1]. \end{aligned}$$

The four terms correspond to:

- adding a white cookie to an empty box;
- adding a white cookie to a nonempty box;
- adding a red cookie to an empty box;
- adding a red cookie to a waiting box.

In the second transition, the previous state contains $n+j-1$ boxes and $z-1$ empty boxes, so the number of nonempty boxes is

$$(n+j-1) - (z-1) = n+j-z.$$

After all additions, only states with $w = 0$ are valid. For a fixed sequence of parent choices that produces a tree with z leaves, the number of valid colorings is exactly X. Therefore,

$$dp_m[0][z]$$

is the sum of X over all such sequences.

Since the expected result for this tree is X/z, we need

$$\sum_{z \geq 1} \frac{dp_m[0][z]}{z}.$$

At the j-th addition, the parent is chosen uniformly among $n+j-1$ existing vertices. Hence every sequence of parent choices has probability

$$\frac{1}{n(n+1)\cdots(n+m-1)}.$$

The final answer is

$$\left(\sum_{z\geq 1} dp_m[0][z]\cdot z^{-1}\right)\cdot\left(\prod_{x=n}^{n+m-1} x\right)^{-1} \pmod{998244353}.$$

All divisions are performed using modular inverses.

The time complexity is

$$O\left(m(n+m)^2\right),$$

and the memory complexity is

$$O\left((n+m)^2\right).$$

Problem B. Beggar Robin

Let us first determine the value of f for one fixed array.

For an array of length 1, clearly $f = 0$.

For an array of length 2, Robin earns a coin immediately if the two numbers have different parity, so $f = 1$. Otherwise, after averaging them, the two numbers become equal, and Robin can never earn a coin. Thus, $f = 0$.

Now consider an array of length at least 3. If all its elements are equal, then every operation leaves the array unchanged, so $f = 0$.

Otherwise, we claim that f is the minimum positive integer k such that the elements are not all equal modulo 2^k.

First, suppose that all elements are equal modulo 2^k. We prove that Robin cannot earn a coin during the first k operations.

The base case $k = 1$ is immediate: all numbers have the same parity, so the first average is an integer.

For the induction step, write the two chosen numbers as

$$x\cdot 2^k + r \quad \text{and} \quad y\cdot 2^k + r.$$

Their average is

$$(x+y)\cdot 2^{k-1} + r.$$

Therefore, after one operation, all elements are still equal modulo 2^{k-1}. By induction, Robin cannot earn a coin during the next $k-1$ operations.

It remains to prove that k operations are always sufficient. Since k is minimal, all elements are equal modulo 2^{k-1}, but not modulo 2^k. Hence there are two elements of the form

$$x\cdot 2^k + r \quad \text{and} \quad y\cdot 2^k + 2^{k-1} + r.$$

For $k = 1$, these elements have different parity, so one operation is sufficient.

For $k > 1$, average these two elements. The result is

$$(x+y)\cdot 2^{k-1} + r + 2^{k-2}.$$

Both chosen positions receive this value. Since the array has at least 3 elements, at least one position remains unchanged and has residue r modulo 2^{k-1}. Thus, the new array is equal modulo 2^{k-2}, but not modulo 2^{k-1}. By induction, another $k-1$ operations are sufficient.

Now we need to sum f^2 over all subarrays.

For every k, construct the array

$$b_i^{(k)} = a_i \bmod 2^k.$$

A subarray of length at least 3 satisfies $f > k$ exactly when all its values in $b^{(k)}$ are equal.

Let S_k be the number of constant subarrays of length at least 3 in $b^{(k)}$. To calculate S_k, split $b^{(k)}$ into maximal segments of equal values. A segment of length c contains

$$\frac{(c-1)(c-2)}{2}$$

subarrays of length at least 3.

Consider a nonconstant subarray with $f = s$. It is constant modulo 2^k for exactly

$$k = 0, 1, \ldots, s-1.$$

Therefore, using the coefficients

$$(k+1)^2 - k^2,$$

its total contribution is

$$\sum_{k=0}^{s-1} \left((k+1)^2 - k^2\right) = s^2.$$

Since $a_i < 2^{30}$, it is enough to consider k from 0 to 30.

A subarray whose elements are all equal is counted for every k, although its value of f is 0. Its total unwanted contribution is

$$31^2.$$

Therefore, after processing all k, we subtract 31^2 for every constant subarray of the original array.

Subarrays of length 2 are not counted by S_k. We process them separately: an adjacent pair contributes 1 exactly when its elements have different parity.

Thus, the answer is

$$\sum_{i=1}^{n-1} [a_i + a_{i+1} \text{ is odd}] + \sum_{k=0}^{30} \left((k+1)^2 - k^2\right) S_k - 31^2 S_{\text{equal}},$$

where S_{equal} is the number of constant subarrays of length at least 3 in the original array.

Each value of S_k is calculated in linear time. Therefore, the total time complexity is

$$O(n \log A),$$

and the memory complexity is

$$O(n).$$

Problem C. Cost of Conflicts

Let $w(u, v)$ be the final weight of the edge between vertices u and v. The graph is complete, so considering all $O(n^2)$ edges explicitly is not possible.

The solution uses Borůvka's algorithm. During one phase of the algorithm, we find the minimum-weight edge leaving every current connected component. All these edges are then added to the spanning forest.

The main part of the solution is finding all minimum outgoing edges in $O((n + q) \log n)$ time for one Borůvka phase.

Using logarithms

The edge weights may be extremely large because one edge can be multiplied by many values c_i. Therefore, we work with logarithms.

Define

$$p_v = \ln a_v$$

and

$$b_i = \ln c_i.$$

For $u < v$, the logarithm of the initial edge weight is

$$\ln \frac{a_v}{a_u} = p_v - p_u.$$

Every conflict adds b_i to the logarithm of each affected edge. Thus, if $D(u, v)$ is the total contribution of all conflicts to the pair (u, v), then

$$\ln w(u, v) = p_v - p_u + D(u, v)$$

for $u < v$.

Since the logarithm is strictly increasing, comparing edge weights is equivalent to comparing their logarithms.

Representing one conflict by rectangles

Consider one conflict and denote its two intervals by

$$A = [L^1, R^1], \qquad B = [L^2, R^2].$$

For ordered pairs of vertices, the conflict affects the union

$$(A \times B) \cup (B \times A).$$

If the intervals intersect, the pairs inside

$$(A \cap B) \times (A \cap B)$$

belong to both rectangles. However, one conflict must multiply the weight of an edge only once.

Therefore, one conflict with value $b = \ln c$ can be represented by the following three rectangle additions:

$$A \times B \mathrel{+}= b,$$

$$B \times A \mathrel{+}= b,$$

$$(A \cap B) \times (A \cap B) \mathrel{-}= b.$$

Indeed, a pair affected by exactly one orientation receives b once, while a pair belonging to both orientations receives

$$b + b - b = b.$$

Thus, all conflicts produce at most $3q$ rectangle additions in the matrix D.

Borůvka phase

At the beginning of one phase, every vertex belongs to some current connected component. Let

$$\mathrm{comp}(v)$$

denote the component containing vertex v.

For every component, we need to find the minimum-weight edge with exactly one endpoint in that component.

We process every vertex x as one endpoint of an edge. Its possible neighbours are divided into two groups:

$$y < x$$

and

$$y > x.$$

These two groups are handled by two sweep-line passes.

Sweep-line processing of rectangles

Consider a rectangle addition

$$X \times Y \mathrel{+}= d,$$

where

$$X = [l_x, r_x], \qquad Y = [l_y, r_y].$$

We sweep the first coordinate x from 1 to n. While $x \in X$, this rectangle adds d to all positions

$$y \in Y.$$

Therefore, the rectangle can be represented by two sweep events:

$$\text{at } x = l_x : \quad [l_y, r_y] \mathrel{+}= d,$$

$$\text{at } x = r_x + 1 : \quad [l_y, r_y] \mathrel{-}= d.$$

During the sweep, a range-add data structure maintains values for all possible second endpoints y.

All rectangle events depend only on the input and can be reused in every Borůvka phase.

Finding edges to the left

Suppose $y < x$. Then

$$\ln w(y, x) = p_x - p_y + D(x, y).$$

During the first sweep, the value stored for vertex y is

$$D(x, y) - p_y.$$

A query over the interval

$$[1, x - 1]$$

finds the minimum possible value of

$$D(x, y) - p_y.$$

After adding p_x, we obtain the logarithm of the minimum-weight edge from x to a vertex on its left.

However, vertices from the same component as x are not allowed. The data structure must therefore be able to find the minimum value whose vertex belongs to a different component.

Finding edges to the right

Suppose $x < y$. Then

$$\ln w(x, y) = p_y - p_x + D(x, y).$$

During the second sweep, the value stored for vertex y is

$$D(x, y) + p_y.$$

A query over the interval

$$[x + 1, n]$$

finds the minimum possible value of

$$D(x, y) + p_y.$$

After subtracting p_x, we obtain the logarithm of the minimum-weight edge from x to a vertex on its right.

Ignoring vertices from the same component

A usual range minimum is not sufficient because its vertex may belong to the same component as x.

For every segment of the range-add data structure, we keep the two smallest values belonging to two different components.

When combining two segments, there are at most four stored candidates. For every component, only its smallest candidate matters, and among all candidates we keep the two smallest ones with different component identifiers.

A range addition changes all values in a covered segment by the same amount, so it does not change their order or their component identifiers.

Suppose a query returns two candidates with different components.

If the first candidate does not belong to $\text{comp}(x)$, it is the answer.

Otherwise, the first candidate belongs to $\text{comp}(x)$, and the second candidate is the smallest candidate belonging to any other component.

Thus, storing two minima with different component identifiers is sufficient.

Obtaining the minimum outgoing edge

For every vertex x, the first sweep finds its best valid edge to the left, and the second sweep finds its best valid edge to the right.

The smaller of these two edges is the minimum-weight edge from x to a different component.

For every component, we take the minimum among the candidates found for all of its vertices. This is exactly the minimum-weight edge leaving that component.

After all such edges have been found, they are added to the current spanning forest. Edges whose endpoints have already become connected are ignored.

Computing the final answer

All edge comparisons are performed using logarithms.

When an edge with logarithmic weight z is added to the spanning tree, its real weight is

$$e^z.$$

It is enough to compute this value only for the edges that are actually added. If

$$z \geq \ln 10^9,$$

then the total answer is already at least 10^9.

Otherwise, the value e^z is added to the current sum. As soon as the sum reaches 10^9, the required output is 10^9.

Logarithms, exponentials, and the accumulated answer should be evaluated with sufficient floating-point precision.

Complexity analysis

Each conflict creates at most three rectangles and at most six sweep events.

In one Borůvka phase, both sweeps together perform:

- $O(q)$ range additions;
- $O(n)$ range queries.

Each operation takes $O(\log n)$ time. Therefore, one phase takes

$$O((n+q)\log n)$$

time.

There are $O(\log n)$ phases, so the total time complexity is

$$O((n+q)\log^2 n).$$

The memory complexity is

$$O(n+q).$$

Problem D. Digesting

Let M be the maximum element of the current array. If $M > 0$, then after one digesting operation every new element is strictly smaller than the element used as its modulus, and that element is at most M. Therefore, the maximum strictly decreases after every operation.

Initially, $M \le n$. Since we perform exactly $n-1$ operations, the only possible positive final value is 1. On the other hand, obtaining 0 is clearly always possible.

Thus, the answer is always either 0 or 1, and we only need to determine when obtaining 1 is possible.

If the final value is 1, the maximum must decrease by exactly one after every operation. In particular, the successive maxima must be

$$n, n-1, \ldots, 1.$$

Now consider some value

$$c \ge \left\lceil \frac{n}{2} \right\rceil.$$

We claim that such a value cannot appear during the process unless it was already present initially.

Indeed, suppose an operation produces

$$x \bmod y = c.$$

If $x < y$, then the result equals x, so c was already present as the numerator. Otherwise,

$$x \ge y + c.$$

Since a remainder is smaller than the modulus, $y > c$, and therefore

$$x > 2c \ge n,$$

which is impossible because all values are at most n.

Hence no new value from

$$\left[\left\lceil \frac{n}{2} \right\rceil, n\right]$$

can ever be created. Since every value in this interval must appear as one of the successive maxima, all of them must be present in the initial array.

It remains to prove that this condition is sufficient.

Assume that an array of length n contains every value from

$$h = \left\lceil \frac{n}{2} \right\rceil$$

to n. Choose one occurrence of every such value and make their positions form one permutation cycle. Arrange the cycle so that every value $x < n$ is taken modulo $x+1$, while n is taken modulo h. Make every other position a fixed point.

For every $h \le x < n$,

$$x \bmod (x+1) = x.$$

If $n = 2q$, then $h = q$ and

$$n \bmod h = 0.$$

Delete this zero. The remaining array contains every value from q to $2q-1$, which is exactly the required interval for an array of length $n-1$.

If $n = 2q+1$, then $h = q+1$ and

$$n \bmod h = q.$$

The selected cycle produces every value from q to $2q$. Delete any unselected position, which becomes zero because it is a fixed point. The remaining array again contains the required interval for length $n-1$.

Therefore, the same condition can be preserved after every operation. By induction, we can reduce the array to the single value 1.

So the answer is 1 if and only if every integer in

$$\left[\left\lceil \frac{n}{2} \right\rceil, n\right]$$

appears in the initial array. Otherwise, the answer is 0.

We only need to mark which values occur and check this interval.

The time complexity is

$$O(n),$$

and the memory complexity is

$$O(n).$$

Problem E. Easter Eggs

Every minute adds exactly one egg, so the number of minutes is equal to the total number of eggs in the final array.

For any strictly increasing array of non-negative integers,

$$a_i \geq i - 1.$$

Therefore, at least

$$S = 0 + 1 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

minutes are required.

However, no buckets can be locked before the first egg is dropped. If the first egg is placed into the first bucket, then its value will remain at least 1. Consequently, the final array must satisfy

$$a_i \geq i,$$

so in this case at least

$$1 + 2 + \ldots + n = S + n$$

minutes are required.

The first bucket is chosen with probability $\frac{1}{n}$. Thus, every strategy has expected duration at least

$$\left(1-\frac{1}{n}\right)S+\frac{1}{n}(S+n)=S+1.$$

Now we show a strategy that achieves this bound.

After the first egg is dropped, choose the target array as follows:

— if the first bucket was chosen, use $(1,2,\ldots,n)$; — otherwise, use $(0,1,\ldots,n-1)$.

In both cases, the current number of eggs in every bucket does not exceed its target value.

Lock every bucket as soon as it reaches its target. Therefore, an unlocked bucket is always strictly below its target, so no bucket can ever receive too many eggs. Eventually, all buckets reach their target values, and the resulting array is strictly increasing.

If the first bucket was chosen, the process takes exactly $S+n$ minutes. Otherwise, it takes exactly S minutes. Hence the expected number of minutes is

$$S+1=\frac{n(n-1)}{2}+1.$$

The solution works in $O(1)$ time and uses $O(1)$ memory.

Problem F. Fresh Cartridges

Observation

For one query $[l,r]$, let A be the given vectors used in the first matrix and B those used in the second one. The matrices can be completed by bought columns if and only if A and B are disjoint and both linearly independent. Indeed, every independent set in $\mathbb{F}_P^n$ can be extended to a basis.

Let $U(l,r)$ be the maximum of $|A|+|B|$ over such sets inside $[l,r]$. The number of bought columns is therefore

$$2n-U(l,r).$$

We only have to find $U(l,r)$.

One rank instead of two independent sets

For every v_i, choose independent random nonzero $a_i,b_i\in\mathbb{F}_P$ and construct a vector of dimension $d=2n$:

$$w_i=(a_iv_i,b_iv_i).$$

With high probability,

$$U(l,r)=\operatorname{rank}(w_l,\ldots,w_r).$$

First, consider a_i,b_i as formal variables. Fix a set of indices T. We show that the corresponding columns w_i are independent exactly when the original vectors with indices in T can be split into two disjoint independent sets.

Assume first that such a split exists; call the sets A and B. Since A is independent, there are $|A|$ coordinates in which its vectors form a matrix with a nonzero determinant. Similarly, choose $|B|$ coordinates for B. Now take these $|A|$ rows from the upper half of the w_i and these $|B|$ rows from the lower half. This gives a square minor of the columns indexed by T.

When this minor is expanded, consider the summand which uses the columns of A through the upper half and the columns of B through the lower half. Its coefficient is, up to sign, the product of the two nonzero determinants chosen above. Its monomial is

$$\prod_{i\in A} a_i \cdot \prod_{i\in B} b_i.$$

No other summand has this monomial: for every i, its factor tells us whether column i was taken from the upper or the lower half. Thus this minor is a nonzero polynomial, so the columns w_i with $i \in T$ are independent.

For the converse, suppose that these columns are independent. Then some square minor is a nonzero polynomial. Its rows are divided between the upper and lower halves. Expand this determinant by choosing which columns are used by the upper rows; all remaining columns are used by the lower rows. Since the polynomial is nonzero, one of its monomials has a nonzero coefficient. Put in A exactly the indices whose factor in this monomial is a_i, and put all other indices in B.

The coefficient of this monomial is, up to sign, the product of a determinant of the original columns in A and a determinant of the original columns in B, using the selected upper and lower rows respectively. Since the product is nonzero, both determinants are nonzero. Hence A and B are independent, which gives the required split of T.

Therefore the formal rank of any segment is exactly $U(l, r)$.

After substituting random field values, a formally nonzero minor can become zero, but a new nonzero minor cannot appear. Thus the computed rank can only be too small. If the true rank is K, one nonzero minor has degree K, so by Schwartz–Zippel the failure probability of one trial is at most

$$\frac{K}{P-1} \le \frac{2n}{P-1}.$$

Run two or three independent trials and keep the maximum rank for every query.

Ranks of all segments

It remains to answer segment-rank queries for vectors $w_i \in \mathbb{F}_P^d$. Process the right endpoint from left to right. After processing positions $1..r$, maintain a Gaussian basis of their vectors which prefers vectors from the largest possible original positions.

For each leading coordinate c, store a basis vector `basis[c]` and its original position `last[c]`. To insert a vector x from position p, scan coordinates from left to right.

- If $x[c] = 0$, continue.
- If `basis[c]` is empty, put x there and set `last[c]` to p.
- Otherwise, if `last[c]` $< p$, swap x with `basis[c]` and also swap p with `last[c]`. The later vector stays in the basis.
- Eliminate coordinate c from the current x. Without row normalization, if $\alpha = x[c]$ and $\beta =$ `basis[c]`$[c]$, use

$$x \leftarrow \beta x - \alpha\,\texttt{basis[c]}.$$

All operations are modulo P. Coordinates before c are already zero, so an elimination takes $O(d)$ time. One insertion takes $O(d^2)$ time.

Why this basis answers queries

Consider the greedy algorithm which scans a prefix in the order $r, r-1, \ldots, 1$ and takes a vector if it is independent of the vectors already taken. It produces a basis that prefers rightmost positions.

The insertion procedure maintains exactly this basis. A new vector has the largest position seen so far; if it conflicts with an old pivot vector, the swap keeps the new vector and continues inserting the old one. This is the standard Gaussian-elimination implementation of the basis exchange performed by the greedy algorithm. By induction over r, `last[c]` are precisely the positions of this right-greedy basis.

For a query $[l, r]$, run the same greedy algorithm only on $r, r-1, \ldots, l$. It selects $\operatorname{rank}(w_l, \ldots, w_r)$ vectors. The choices made before reaching l are identical to the choices in the full prefix; vectors at positions below l are considered later and cannot replace them. Hence, after processing r,

$$\operatorname{rank}(w_l, \ldots, w_r) = \left|\{c : \texttt{basis[c]} \text{ is nonempty and } \texttt{last[c]} \geq l\}\right|.$$

Group queries by their right endpoint and answer them when that endpoint is processed.

To count the displayed quantity quickly, keep a Fenwick tree over positions. Whenever `last[c]` changes from *old* to *new*, add -1 at *old* and $+1$ at *new*. Then the rank for left endpoint l is the suffix sum on $[l, m]$.

Complexity

For one trial, all insertions take $O(md^2)$ time and all Fenwick queries take $O(q \log m)$ time. With $d = 2n$ and a constant number of trials, the total complexity is

$$O\big(\texttt{TRIALS} \cdot (mn^2 + q \log m)\big),$$

and the memory consumption is $O(mn + q + n^2 + m)$.

Problem G. Game Show: XOR Edition

Let the current XOR of the numbers in the bag be x, and let the offered number be y.

If the contestant takes it, the new XOR becomes

$$x \text{ xor } y.$$

Since y is uniformly distributed over $[0, 2^k - 1]$, this new XOR is also uniformly distributed over the same range, regardless of the current value x.

Therefore, every round can be viewed as follows: the contestant is offered a uniformly random value and may either keep the current XOR or replace it with the offered value. Future offers have the same distribution regardless of this choice, so it is always optimal to keep the larger value.

Thus, the contestant takes y exactly when

$$x \text{ xor } y \geq x.$$

For every nonzero number, consider the position of its highest set bit. Split all numbers into groups according to this position and output the groups in increasing order.

Suppose we are currently processing the group with highest bit i. All previously processed numbers have smaller highest bits, so bit i of the current XOR is zero.

The first number from this group changes bit i from zero to one. Therefore, the XOR increases and the contestant takes this number.

Afterwards, bit i is one. Every other number from the same group would change it back to zero, so the XOR would decrease and the contestant skips it.

Consequently, the contestant takes exactly the first number from every nonempty group. The final prize is the XOR of these selected representatives.

Zeroes can be placed at the end. They are taken because of the tie-breaking rule, but they do not change the XOR.

If the group with highest bit $k-1$ is nonempty, its first number is necessarily taken. Numbers from lower groups cannot change this bit, so the final prize is at least

$$2^{k-1}.$$

The probability that this group is empty is only 2^{-n}. Therefore, it remains to select one representative from every nonempty group so that their XOR is exactly

$$2^{k-1}.$$

We find these representatives using a DP over the groups.

Initially, only XOR zero is reachable. When processing a nonempty group, try every candidate from this group together with every previously reachable XOR. For every new reachable value, remember the selected candidate, so that the representatives can later be reconstructed.

Using all n numbers in this DP would be too slow. Instead, take all nonzero numbers among the first $10k$ input positions. If some highest-bit group is absent among them, additionally take one arbitrary number from that group.

Thus, at most $11k$ numbers participate in the DP. All remaining numbers can later be placed after the selected representative of their group, so the contestant skips them.

After reconstructing the representatives, output the groups in increasing order of their highest bit. In every group, output the selected representative first and all other numbers afterwards. Finally, output all zeroes.

The existence of the required XOR is not guaranteed for an adversarial array. This solution relies on the statement's guarantee that the input numbers were generated independently and uniformly.

As an experimental check, we generated 10^4 independent tests for all values of k, using the minimum allowed length $n = 10k$.

For

$$k = 4, \qquad n = 40,$$

the target value

$$2^{k-1} = 8$$

was reachable in all 10000 tests.

For

$$k = 18, \qquad n = 180,$$

the target value

$$2^{k-1} = 2^{17}$$

was also reachable in all 10000 tests.

At most $O(k)$ numbers participate in the DP, and there are at most 2^k reachable XOR values. Each insertion into a set takes $O(k)$ time, so the total time complexity is

$$O\left(n + k^2 2^k\right).$$

The memory complexity is

$$O\left(n + 2^k\right).$$

Problem H. Hockey Hog

From geometry to array

All indices are cyclic. For every net, its visibility region is the convex cone at the corresponding vertex of A, which contains A. Fix any point O inside A. If a valid point P sees some nets, segment OP is in all their cones, so its intersection with the boundary of B sees the same nets. Thus, it is enough to consider points on the boundary of B.

For net i, let L_i be where ray A_iA_{i+1} meets the boundary of B, and let R_i be where ray A_iA_{i-1} meets it. The net is visible exactly on the counterclockwise boundary arc from L_i to R_i, inclusive. Thus each net gives one circular interval.

Sort all points L_i, R_i along the boundary of B counterclockwise and merge equal ones. Let the resulting circular array be $g_0, \ldots, g_{q-1}$, where $q \le 2n$. It is enough to check these positions: visibility changes only at an endpoint, and the intervals are closed. Every net becomes a circular interval $[l_i, r_i]$ after enumerating endpoints. The problem now does not need geometry anymore.

Process the permutation backwards. Maintain the number of active nets visible from every g_j. Adding net p_k is a range addition of 1 on cyclic segment $[l_{p_k}, r_{p_k}]$; the global maximum is the answer for k. Use a segment tree with range addition and maximum.

Computing the endpoints

Find the L_i and R_i endpoints in two separate, identical two-pointer scans. The only difference is the ray: use $A_i \to A_{i+1}$ for L_i and $A_i \to A_{i-1}$ for R_i. In either scan, the endpoints move counterclockwise along B as i increases. Viewed from their common vertex, successive L rays follow consecutive sides of A, while successive R rays follow those sides in reverse. Both direction sequences turn counterclockwise, as do their intersections with the boundary of B. Thus, the pointer over the edges of B only moves forward.

Fix one of these scans. Represent an endpoint on edge B_jB_{j+1} as pair (j, t), where $B_j + te$ is the point, $e = B_{j+1} - B_j$, and $t \in [0, 1)$. This assigns every vertex to the following edge and makes the representation unique. For the pointer test, write the intersection of the two supporting lines as

$$S + \lambda d = B_j + te.$$

Taking the cross product with d gives $0 = T - tD$, where

$$D = \mathrm{cross}(d, e), \qquad T = \mathrm{cross}(B_j - S, d).$$

Hence, $t = T/D$. Advance the pointer until

$$D > 0, \qquad 0 \le T < D.$$

The first condition means that d points out of the CCW edge. Since S is inside B, an intersection on this edge is then in front of S, so no separate check for λ is needed. If the ray hits B_{j+1}, then $T = D$: skip this edge and accept the next one, where the same point has $T = 0$. Store $t = T/D$ as an exact fraction and compare fractions by cross multiplication in 128-bit integers, since floating point arithmetic will most likely fail due to precision issues.

Complexity

The two-pointer search takes $O(n + m)$ time. Sorting endpoints and the segment tree take $O(n \log n)$ time, so one test case is solved in $O(n + m + n \log n)$ time and $O(n + m)$ memory.

Problem I. Incident in the Bakery

Let

$$T(P)_i = \frac{P_i + P_{i+1}}{2}, \qquad \varepsilon_i = (-1)^{i-1}, \qquad S(P) = \sum_{i=1}^{n} \varepsilon_i P_i.$$

All indices are cyclic. We denote the cross product by $[u, v] = u_x v_y - u_y v_x$.

Inverting one operation

This section only describes the algebraic inversion of T for cyclic sequences; convexity is checked separately. Suppose Q is known and we need to find P such that $T(P) = Q$. Then

$$P_{i+1} = 2Q_i - P_i.$$

Once P_1 is fixed, all remaining vertices are determined uniquely. It is convenient to first construct

$$R_1 = 0, \qquad R_{i+1} = 2Q_i - R_i.$$

Then all candidates have the form

$$P_i = R_i + \varepsilon_i v,$$

where $v = P_1$.

If n is odd, the closing condition gives the unique value $v = R_{n+1}/2$.

If n is even, a preimage exists if and only if $S(Q) = 0$. Indeed, every image satisfies

$$S(T(P)) = \sum_i \varepsilon_i \frac{P_i + P_{i+1}}{2} = \frac{S(P)}{2} - \frac{S(P)}{2} = 0.$$

Conversely, the closing condition is $R_{n+1} = 0$, and the recurrence gives $R_{n+1} = -2S(Q)$. Thus, if $S(Q) = 0$, then $R_{n+1} = 0$ and v may be chosen arbitrarily.

Odd number of vertices

Both inverse operations are unique.

1. Restore the unique polygon B such that $T(B) = C$.

2. Check the local turns of B:

$$[B_{i+1} - B_i,\ B_{i+2} - B_{i+1}] > 0.$$

   If this check fails, there is no answer.

3. Restore the unique polygon A such that $T(A) = B$.

4. Similarly, check the local turns of A:

$$[A_{i+1} - A_i,\ A_{i+2} - A_{i+1}] > 0.$$

   If the check succeeds, output A; otherwise, output `NO`.

We prove below why checking local turns is sufficient for strict convexity.

Even number of vertices

First, restore the intermediate polygon B.

It is necessary for C to have a preimage, so first check $S(C) = 0$. If this does not hold, there is no answer. Otherwise, construct one preimage B^0 of polygon C with $B^0_1 = 0$. All preimages have the form

$$B_i = B^0_i + \varepsilon_i u.$$

For B to be an image of T, it must satisfy $S(B) = 0$ by the criterion above. Therefore, the suitable B is unique:

$$S(B^0) + nu = 0, \qquad u = -\frac{S(B^0)}{n}.$$

Construct this B and check its local turns. If not all of them are positive, there is no answer.

Now restore A. Construct one preimage A^0 of polygon B with $A^0_1 = 0$. All preimages have the form

$$A_i(v) = A^0_i + \varepsilon_i v.$$

Let

$$d_i = A^0_{i+1} - A^0_i.$$

Then side i of $A(v)$ is

$$e_i(v) = A_{i+1}(v) - A_i(v) = d_i - 2\varepsilon_i v.$$

The local left turn is described by the inequality

$$[e_i(v), e_{i+1}(v)] = [d_i, d_{i+1}] + 2\varepsilon_i[d_i + d_{i+1}, v] > 0.$$

This is a linear inequality in the two coordinates of v.

The statement guarantees that, if an answer exists, then there is an answer whose coordinates are integers divisible by 4 and whose absolute values do not exceed 10^9. For such an answer, every positive cross product of adjacent sides is divisible by 16, and hence is at least 16. Therefore, we can search for a solution of the more convenient closed system

$$[d_i, d_{i+1}] + 2\varepsilon_i[d_i + d_{i+1}, v] \geq 1.$$

The guaranteed answer satisfies even the right-hand side 16, so replacing it with 1 does not exclude that answer, but the risk of screwing up the precision is lower.

To make the intersection bounded, add a bounding box:

$$|v_x| \leq 10^9, \qquad |v_y| \leq 10^9.$$

Thus, we need to find any point in the intersection of $n + 4$ half-planes.

The usual $O(n \log n)$ half-plane intersection algorithm with floating-point line coefficients and intersection points is not accurate enough. To make it exact, find the set of lines forming the sides of the intersection using integer arithmetic, and only then compute their intersection points using floating-point arithmetic. Intermediate values do not fit into ‘int64‘, so ‘int128‘ is needed; fractions must be compared in $O(\log X)$ time using an algorithm similar to the Euclidean algorithm, where X bounds the numbers being compared. Ordinary cross-multiplication may not fit even into ‘int128‘.

Alternatively, use a simpler $O(n^2)$ algorithm. Iterate over the half-planes, assuming the boundary line of the current half-plane is a boundary line of the final intersection. Only one parameter remains on this line. Intersecting every other half-plane with this line yields a constraint of the form $t \geq l$ or $t \leq r$. If the intersection of all one-dimensional constraints is nonempty, take the corresponding point on the line. If no line yields a point, the intersection is empty.

If the intersection is empty, output `NO`. Otherwise, take the found v and output

$$A_i = A_i^0 + \varepsilon_i v.$$

Why local checks are sufficient

We need the following lemma. Let $Q = T(P)$, and suppose Q is strictly convex. Denote the sides of polygon P by

$$e_i = P_{i+1} - P_i.$$

If for every i,

$$[e_i, e_{i+1}] > 0,$$

then P is strictly convex.

Let us prove this. The condition $[e_i, e_{i+1}] > 0$ means that at every vertex of P, we turn left by an angle strictly between 0 and π. This alone does not yet guarantee convexity: in theory, the polygon could make several full turns.

Now use the fact that $Q = T(P)$. A side of Q is

$$Q_{i+1} - Q_i = \frac{e_i + e_{i+1}}{2}.$$

If vectors e_i and e_{i+1} form a left turn by an angle in $(0, \pi)$, then the vector $e_i + e_{i+1}$ points strictly between them. Hence, the direction of every side of Q is inserted between the directions of two adjacent sides of P.

Imagine walking along P and recording the directions of its sides, without resetting the angle after 2π. Between the direction of e_i and the next direction of e_{i+1}, there is always the direction of the corresponding side of Q. Therefore, the sides of P and the sides of Q make the same number of full turns.

But Q is strictly convex and its vertices are listed counterclockwise, so its sides make exactly one full turn. Consequently, the sides of P also make exactly one full turn. A polygon whose turns are all locally left and whose total turn is one full revolution is strictly convex.

The same argument also shows that T preserves strict convexity.

The application of the lemma is now immediate. For B, the image is the input strictly convex polygon C. Once B has passed the check, it is strictly convex; hence, for the restored A, it is also sufficient to check only local turns, because $T(A) = B$.

Output coordinate bound

We show that we can require

$$|A_{i,x}|, |A_{i,y}| \le 3 \cdot 10^9.$$

Let $L = 10^9$.

When n is odd, both inverse operations are unique. Therefore, if an answer exists, the algorithm restores the guaranteed answer, and its coordinates do not exceed L in absolute value.

Now let n be even, and suppose a guaranteed answer $\widetilde{A}$ exists with all coordinates bounded by L in absolute value. The algorithm chooses the unique B satisfying $T(B) = C$ and $S(B) = 0$. Since $S(T(\widetilde{A})) = 0$, this B equals $T(\widetilde{A})$.

The preimage A^0 is constructed with $A^0_1 = 0$. The guaranteed answer corresponds to the parameter $v^* = \widetilde{A}_1$, so

$$A^0_i = \widetilde{A}_i - \varepsilon_i v^*.$$

For either coordinate $\alpha \in \{x, y\}$,

$$|(A^0_i)_\alpha| \le |(\widetilde{A}_i)_\alpha| + |(v^*)_\alpha| \le 2L.$$

The found parameter lies inside the bounding box, so $|v_x|, |v_y| \le L$. Therefore,

$$|(A_i)_\alpha| \le |(A^0_i)_\alpha| + |v_\alpha| \le 3L.$$

Thus, all output coordinates do not exceed $3 \cdot 10^9$ in absolute value.

Complexity

Restoration and checks take $O(n)$. In the even case, we additionally need to find a point in the intersection of $O(n)$ half-planes in $O(n \log X)$ or $O(n^2)$ time.

Memory usage is $O(n)$.

Problem J. Jaded Juice

Let d_i be the total amount already poured from the i-th ordinary bottle, and let

$$S = \sum_i d_i.$$

As long as x is still unknown, no ordinary bottle is visible, so

$$x - d_i \geq l_i$$

for every i. The special bottle is also hidden, hence

$$x + S \leq w.$$

Therefore, all still possible values of x form the interval

$$A \leq x \leq w - S, \qquad A = \max_i (d_i + l_i).$$

Initially,

$$A = \max_i l_i.$$

If $x < A$, one of the ordinary bottles is already visible and x is known immediately, so only the branch $x \geq A$ matters.

The value of x becomes uniquely determined as soon as

$$A + S \geq w.$$

Suppose we choose bottle i. Since the smallest still possible value of x is A, this bottle is guaranteed to contain at least

$$A - d_i$$

units of juice. Pouring a smaller amount cannot provide more information, so we may always pour the maximum safe amount.

After this operation,

$$d_i := A, \qquad A := A + l_i.$$

Thus, the original problem becomes the following deterministic one:

— initially $A = \max l_i$ and all $d_i = 0$;

— in one operation, choose i, set $d_i := A$, and then set $A := A + l_i$;

— minimize the number of operations needed to obtain

$$A + \sum_i d_i \geq w.$$

Sort the labels in non-increasing order:

$$l_1 \geq l_2 \geq \ldots \geq l_n.$$

Only the last operation with a particular bottle affects its final value of d_i. Every earlier operation with this bottle only increases A by l_i. Therefore, all such repeated operations should use bottle 1, since l_1 is maximum, and they should be performed as early as possible.

The remaining operations use distinct bottles. If two such bottles with labels $l_i < l_j$ are used consecutively, using j before i is no worse: the final value of A is unchanged, while the second bottle receives a larger value of d. Hence these bottles should also be processed in non-increasing order of their labels.

Therefore, an optimal sequence has the form

$$\underbrace{1, 1, \ldots, 1}_{\alpha \text{ times}}, 2, 3, \ldots, k+1$$

for some $0 \le k < n$ and $\alpha \ge 1$. It contains $k + \alpha$ operations.

After this sequence, the value of $A + \sum d_i$ is

$$B_k + \alpha C_k,$$

where

$$B_k = (k+1)l_1 + kl_2 + (k-1)l_3 + \ldots + l_{k+1}$$

and

$$C_k = (k+2)l_1.$$

Thus, for a fixed k, the minimum possible value of α is

$$\alpha = \max\left(1, \left\lceil \frac{w - B_k}{C_k} \right\rceil\right).$$

The corresponding number of operations is $k + \alpha$, and we take the minimum over all k.

The values B_k can be maintained in linear time after sorting. Let

$$P_k = l_2 + l_3 + \ldots + l_{k+1}.$$

Starting with $B_0 = l_1$, we have

$$B_k = B_{k-1} + l_1 + P_k.$$

Once $\alpha = 1$ is sufficient for some k, we may stop: this gives $k + 1$ operations, while every larger value of k requires at least $k + 2$ operations.

The time complexity is

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

and the memory complexity is

$$O(n).$$

All intermediate calculations should use 128-bit integers.

Problem K. Kitchen Knives

There are $2^n - 1$ non-empty subsets. Therefore, a product must occur at least 2^{n-1} times to have probability strictly greater than 50%.

If all elements are equal to 1, every non-empty subset has product 1, so the answer is 1.

Otherwise, choose any index i such that $a_i > 1$. Pair every subset not containing i with the same subset after adding i. The products in each pair are different, because one of them is obtained from the other by multiplying by $a_i > 1$.

Hence, any fixed product occurs at most once in every pair, and therefore at most 2^{n-1} times. To occur with probability greater than 50%, it must occur exactly once in every pair.

Consider the pair consisting of the empty subset and the subset $\{i\}$. Their products are 1 and a_i. Since the empty subset is not counted, the only possible answer is

$$s = a_i.$$

Now consider an arbitrary subset of all indices except i, and let its product be p. The corresponding pair has products

$$p \qquad \text{and} \qquad p \cdot s.$$

For s to occur in this pair, we must have either $p = s$ or $p = 1$. Thus, every subset of the remaining elements must have product either 1 or s.

This is possible only when all remaining elements are equal to 1, except possibly one additional element equal to s. Indeed, any other value would already give an invalid singleton product, while two additional copies of s would give the product s^2.

Therefore, after sorting, the array must have one of the following forms:

$$[1, 1, \ldots, 1],$$

$$[1, 1, \ldots, 1, s],$$

or

$$[1, 1, \ldots, 1, s, s].$$

These conditions are also sufficient. With one copy of s, exactly 2^{n-1} subsets contain it. With two copies, exactly 2^{n-1} subsets contain exactly one of them. In both cases,

$$2^{n-1} > \frac{2^n - 1}{2}.$$

Thus, after sorting, we output the maximum element if:

— the second-largest element is 1; or

— the two largest elements are equal and every earlier element is 1.

Otherwise, the answer is -1.

The time complexity is $O(n \log n)$ (can easily achieve $O(n)$ but this was not needed) and the memory complexity is $O(n)$.

Problem L. Leave Triple & Link Tree

Create a bipartite graph. One side contains the vertices of the original graph, the other side contains the triples (triangles). For each triple, connect its triple-vertex to the three corresponding number-vertices. We obtain a bipartite graph in which all vertex degrees are equal to 3. If this graph is disconnected, then obviously no answer exists. Otherwise, we claim that an answer always exists, and we now give a constructive example. By Hall's theorem, this graph has a perfect matching. Let us take the edges of the matching and delete them; in each triangle, there will remain a connection between the two vertices that are not matched to the triangle-vertex. After deleting the matching, the degree of every vertex becomes 2, hence the graph is a set of vertex-disjoint cycles.

Return to the original graph after this deletion. It will also be a set of vertex-disjoint cycles.

Each original triple must have the following form: two adjacent vertices from the same cycle, and the third vertex anywhere. Moreover, each edge in the cycle graph corresponds to one such triple.

For each cycle C, define:

- $self(C)$: the number of triangles whose all three vertices lie in cycle C.
- $in(C)$: the number of triangles with one vertex in this cycle and two vertices in some other cycle.
- $out(C)$: the number of triangles with two vertices in this cycle and the third vertex in some other cycle.

Claim: $in(C) = out(C)$ for all cycles.

Proof: We show that $in(C) + self(C) = |C|$. Consider each vertex in cycle C. We claim that it must participate in exactly one triangle of type $self$ or in. This is true because originally the vertex belonged to three triples, and now it has two edges from the cycle, while the third triple is the remaining one (either in or $self$). Now we show that $out(C) + self(C) = |C|$. For this, consider each edge in cycle C. We claim that it must participate in exactly one triangle of type out or $self$, depending on whether the third vertex lies on the same cycle or not.

Thus $in(C) + self(C) = out(C) + self(C) \implies in(C) = out(C)$.

Build one more graph. The vertices of the new graph are the cycles of the old graph. Add a directed edge between two vertices if, in the corresponding cycles A and B in the old graph, there was a triangle of the form: two vertices from A and one vertex from B. In this graph, for each vertex v we have $in(v)$ in the new graph equal to $in(C)$ in the old graph, and $out(v)$ in the new graph equal to $out(C)$ in the old graph (where vertex v corresponds to cycle C). This graph is also weakly connected, because the original graph was connected and all our transformations preserve connectivity.

Claim: This directed graph is strongly connected.

Proof: Contract the strongly connected components. Consider a sink of this condensation graph. Since $in(v) = out(v)$ for every vertex, the numbers of incoming and outgoing edges of this SCC are also equal. Because it is a sink, the number of outgoing edges is 0, hence the number of incoming edges is also 0. Since the graph is weakly connected, this SCC is the only SCC in the graph, i.e. the graph is strongly connected.

Take any vertex in the graph and run DFS from it along the reversed edges. It will necessarily visit all vertices. Consider the DFS tree found this way. From each vertex, at most one edge goes out.

Return to the graph of vertex-disjoint cycles. The chosen edges correspond to some triples, and from each cycle we choose at most one triple whose edge belongs to this cycle. For each such triple, delete this edge and add any of the two remaining edges of the triple. Thus, from each cycle we remove at most one edge, and the new graph becomes connected (because the graph with "contracted"cycles was connected). From each triple, we still chose exactly one edge. Exactly one cycle remains from which we did not remove any edge (the root from which we started DFS). Delete any edge of this cycle, which corresponds to deleting one triple from the original array.

Final solution plan:

- delete the edges of a perfect matching in bipartite graph triple-vertices
- consider the set of cycles in the original graph
- build a directed graph whose vertices are the contracted cycles and whose edges are triangles between cycles
- run DFS from any vertex along reversed edges

- in the triangles corresponding to the edges of the DFS tree, replace the currently chosen edge by any other one in triangle
- delete any edge from the DFS root cycle

Problem M. Minecraft Spiderweb

Idea

We may assume that the web is a tree: if the union of its threads contains a cycle, deleting one edge of the cycle preserves connectivity of all terminals. There is also no reason for a thread to go outside the rectangle.

There is a strong structural theorem for rectilinear Steiner trees whose terminals lie on the boundary of a rectangle. An optimal tree belongs to one of the topologies considered below. We use this fact without proving it; a full proof is given by Cohoon, Richards, and Salowe, An Optimal Steiner Tree Algorithm for a Net Whose Terminals Are Constrained to a Rectangle (`https://sci-hub.ru/10.1109/43.45871`).

We find the best tree for every topology and take the minimum.

1. Boundary only

If there are no interior segments, the solution is the whole boundary with one gap between consecutive terminals removed. The terminals are given in boundary order, so all gap lengths, including the gap from the last terminal to the first, are obtained in one pass.

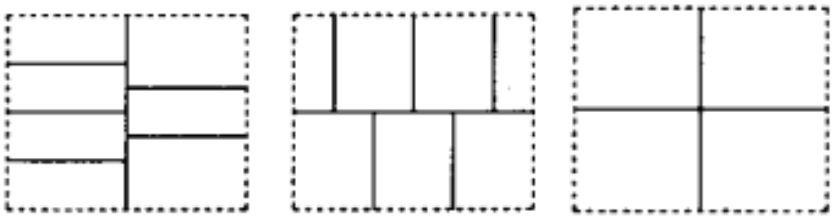
If the longest gap has length *gap*, the best cost of this type is

$$2(W + H) - gap.$$

Removing any one gap leaves a connected path through all terminals, hence it is best to remove the longest one. For one terminal the answer is 0.

2. One complete interior line

A complete interior line joins two opposite sides of the rectangle. First, assume it is vertical, $x = X$.



The structural proof also shows that segments incident to the complete line alternate. We do not use this fact: the algorithm below considers every way to attach boundary pieces to the line. Consequently, the cross topology is already covered by the same procedure and needs no separate algorithm.

DP for one boundary arc

The line $x = X$ is a connected backbone. It splits the boundary into a left and a right arc.

Consecutive terminals of an arc may be joined by boundary segments into one component, which is then attached to the backbone exactly once. Attaching it twice would create a cycle. The attachment point can be chosen to be a terminal of that component.

Let the terminals on the arc be $v_1, v_2, \ldots, v_k$ in this order, and let Δ_i be the arc distance from v_{i-1} to v_i. For a vertical backbone, connecting $v_i = (x_i, y_i)$ directly to it costs

$$c_i = |x_i - X|.$$

Let $con[i]$ be the minimum cost after the first i terminals if all their components are already connected to the backbone. Let $dis[i]$ be the same value when the component containing v_i is not connected yet. Initially,

$$con[1] = c_1, \qquad dis[1] = 0.$$

For $i \geq 2$,

$$\begin{aligned} con[i] &= \min\big(con[i-1] + \Delta_i,\ con[i-1] + c_i,\ dis[i-1] + \Delta_i + c_i\big), \\ dis[i] &= \min\big(con[i-1],\ dis[i-1] + \Delta_i\big). \end{aligned}$$

The three terms for *con* respectively extend an attached component, start a new attached component, and attach the previously unfinished component. The two terms for *dis* start or extend an unfinished component. The answer for the arc is $con[k]$; an empty arc costs 0.

Only the numbers c_i matter in this DP. They may be given by any function of the terminal and may equal ∞; this will be useful later.

Run this DP for both arcs and add the backbone length H. This gives the best answer for a fixed X.

It remains to go through $O(1)$ possible positions of X: it can be proven that X coincides with the coordinate of one of the end terminals on the top or bottom side. The horizontal backbone is handled in the same way after rotating the rectangle by 90°.

3. Several parallel complete lines

Assume that the complete lines are vertical.

Let $x = L$ and $x = R$ be the leftmost and rightmost complete lines. Their total cost is $2H$. Terminals left of L are attached to $x = L$ by the previous boundary-arc DP with $c_i = L - x_i$; terminals right of R are handled symmetrically with $c_i = x_i - R$. A stricter analysis of this topology gives a simpler construction outside the strip with at most one interior attachment. We do not need it: the same arc DP is already linear and is sufficient.

Inside $L \leq x \leq R$, terminals lie only on $y = 0$ and $y = H$.

DP inside the strip

Let

$$L = z_0 < z_1 < \ldots < z_m = R$$

be the two complete-line coordinates and all distinct terminal x-coordinates strictly between them. Terminals on L and R are already covered by the complete lines. After column z_i, store four values: $up[i]$, $down[i]$ for one component continuing through the upper or lower side; $both[i]$ for one component continuing through both sides; and $separate[i]$ for two still separate components.

$$up[0] = down[0] = both[0] = 0, \qquad separate[0] = \infty.$$

For an internal column z_i, $1 \le i < m$, define

$$\begin{aligned} \Delta &= z_i - z_{i-1}, \\ U &= up[i-1] + \Delta, \\ D &= down[i-1] + \Delta, \\ B &= both[i-1] + 2\Delta, \\ S &= separate[i-1] + 2\Delta \end{aligned}$$

— the costs of extending the segments to the current column — and

$$J = \min(U, D, B, S) + H$$

— the cost of adding the vertical connector between columns z_{i-1} and z_i. Then

$$\begin{aligned} up[i] &= \min(J, B, U), \\ down[i] &= \min(J, B, D), \\ both[i] &= \min(J, B), \\ separate[i] &= \min(U, D, S). \end{aligned}$$

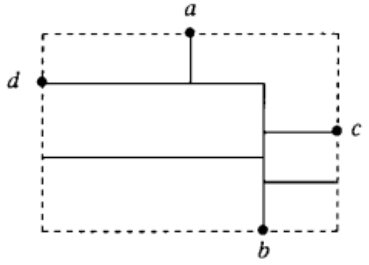
U means that only the upper segment was extended to z_i. Therefore, if the current column has a bottom terminal, omit U from the first formula; otherwise that terminal would not be covered. Symmetrically, if there is a top terminal, omit D from the second formula.

At $z_m = R$, only extend the four states, computing U, D, B, S by the same formulas, and do not add a new connector. The answer is $\min(U, D, B, S)$: $x = R$ is already paid for and joins the two components of *separate*.

As before, L and R can be slid to extreme terminals or to the boundary, so there are only constantly many pairs to check. Rotate the rectangle to handle horizontal complete lines.

4. A complete interior corner

The remaining case has an L-shaped backbone. One orientation is shown below.



Consider the backbone

$$(0, Y) \longrightarrow (X, Y) \longrightarrow (X, 0).$$

Its cost is $X + Y$. The boundary splits into two arcs. Reuse the *con/dis* DP: it does not depend on the form of the attachment cost c_i, so it is enough to define c_i as a function of the terminal.

For the inner arc, from $(0, Y)$ along the left and bottom sides to $(X, 0)$, set

$$c_{\text{in}}(x, y) = \min(X - x, Y - y).$$

On the outer arc, direct attachments are either vertical segments of length $H - Y$ from the top side to level Y, or horizontal segments of length $W - X$ from the right side to $x = X$. Therefore set

$$c_{\text{out}}(x, y) = \begin{cases} H - Y, & x \le X, \\ W - X, & y \le Y, \\ \infty, & \text{otherwise.} \end{cases}$$

The value ∞ does not forbid covering a terminal: it can still join a neighbouring terminal along the boundary. It only forbids starting a new component directly from this terminal.

Thus, for fixed X, Y, the candidate cost is

$$X + Y + (\text{inner arc answer}) + (\text{outer arc answer}).$$

The structural theorem imposes additional restrictions on alternation and on the number of segments incident to the sides of the corner. They are not used here: the arc DP permits a larger set of valid trees. This does not hurt correctness: every candidate constructed by the DP is feasible and therefore cannot be cheaper than the true optimum.

The coordinates X, Y are fixed by extreme terminals on the corresponding sides. Rotations handle all other orientations, and only constantly many candidates are considered.

Correctness and complexity

The structural theorem guarantees an optimum in one of the considered cases.

Each DP performs one linear pass over the terminals, and the number of backbone positions is constant. The time complexity is $O(n)$ and the memory complexity is $O(n)$.

Problem N. Native Niuean

Put

$$M = m + 2.$$

From the recurrence,

$$f_{i+1} = a_i f_i + f_{i-1},$$

we obtain

$$a_i f_i = f_{i+1} - f_{i-1}.$$

Summing over all positions makes all intermediate terms cancel:

$$\sum_{i=1}^{n} a_i f_i = f_n + f_{n+1} - f_0 - f_1 = f_n + f_{n+1} - 2.$$

Therefore, an array is valid exactly when

$$f_n + f_{n+1} \le M.$$

Now notice that the recurrence

$$f_{i+1} = a_i f_i + f_{i-1}$$

looks very similar to one step of the Euclidean algorithm: the larger number is represented as a positive multiple of the smaller one plus the previous number.

This suggests that the final pair

$$(f_n, f_{n+1})$$

may uniquely encode the whole array. We now prove this formally.

Given an array of positive integers, map it to

$$(x, y) = (f_n, f_{n+1}).$$

Note that, consecutive values are coprime, because

$$\gcd(f_{i+1}, f_i) = \gcd(a_i f_i + f_{i-1}, f_i) = \gcd(f_{i-1}, f_i).$$

Repeating this equality gives

$$\gcd(f_n, f_{n+1}) = \gcd(f_0, f_1) = 1.$$

Thus, every non-empty array produces a pair satisfying

$$1 \le x < y, \qquad \gcd(x, y) = 1.$$

We now construct the inverse mapping.

Suppose that a coprime pair $1 \le x < y$ is given. If it is the final pair of some array, then its last element must satisfy

$$y = a_n x + z,$$

where z is the previous value of the sequence. Since all f_i are positive, we need

$$1 \le z \le x.$$

Consequently, the only possible value of a_n is

$$a_n = \left\lfloor \frac{y-1}{x} \right\rfloor,$$

and then

$$z = y - a_n x.$$

The use of $y - 1$ instead of y guarantees that the remainder is positive.

We have

$$1 \le z \le x$$

and

$$\gcd(z, x) = \gcd(y, x) = 1.$$

If $x > 1$, then $z \neq x$, because otherwise x would divide both z and x. Hence

$$1 \le z < x.$$

Thus, unless $x = 1$, the pair strictly decreases:

$$(x, y) \longrightarrow (z, x).$$

If $x = 1$, coprimality and $1 \le z \le x$ imply $z = 1$, so we reach the initial pair

$$(1, 1).$$

Therefore, repeatedly applying this reverse Euclidean step eventually reaches $(1, 1)$. The obtained quotients, read in reverse order, form an array of positive integers.

Every reverse step is uniquely determined, and applying the original recurrence reconstructs the given pair. Hence this is a bijection:

$$\boxed{\text{non-empty arrays} \longleftrightarrow \{(x, y) : 1 \le x < y,\ \gcd(x, y) = 1\}.}$$

It is also convenient to extend this bijection by matching the empty array with the pair $(1, 1)$.

Under this bijection, the nativeness condition becomes

$$x + y \le M.$$

Let us count all arrays satisfying this condition, without yet considering the lexicographical requirement.

Fix

$$s = x + y.$$

Choosing x determines $y = s - x$, and

$$\gcd(x, y) = \gcd(x, s - x) = \gcd(x, s).$$

Therefore, the number of ordered positive coprime pairs with sum s is exactly

$$\varphi(s).$$

For $s \ge 3$, the map

$$(x, y) \longleftrightarrow (y, x)$$

is a fixed-point-free bijection on these pairs.

Indeed, a fixed point would have $x = y$. Since the pair is coprime, this would force $x = y = 1$, which is possible only for $s = 2$.

Hence, for every $s \geq 3$, exactly half of the $\varphi(s)$ pairs satisfy $x < y$. In particular, this proves that $\varphi(s)$ is even for every $s \geq 3$.

Thus, the total number of valid arrays is the integer

$$T = \sum_{s=3}^{M} \frac{\varphi(s)}{2}.$$

Equivalently, if

$$\Phi(M) = \sum_{s=1}^{M} \varphi(s),$$

then

$$T = \frac{\Phi(M) - 2}{2},$$

because

$$\varphi(1) = \varphi(2) = 1.$$

Now we need to impose the condition that the array is lexicographically smaller than its reverse.

Define

$$B(t) = \begin{pmatrix} t & 1 \\ 1 & 0 \end{pmatrix}, \qquad v = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The recurrence can be written as

$$\begin{pmatrix} f_{i+1} \\ f_i \end{pmatrix} = B(a_i) \begin{pmatrix} f_i \\ f_{i-1} \end{pmatrix}.$$

Therefore,

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = B(a_n)B(a_{n-1}) \cdots B(a_1)v.$$

Since the sum of the two coordinates equals the nativeness plus 2,

$$N(a) + 2 = v^{\mathsf{T}} B(a_n)B(a_{n-1}) \cdots B(a_1)v.$$

Every matrix $B(t)$ is symmetric. Taking the transpose of this scalar gives

$$N(a) + 2 = v^{\mathsf{T}} B(a_1)B(a_2) \cdots B(a_n)v.$$

The expression on the right is exactly the value corresponding to the reversed array. Hence

$$N(a) = N(\mathrm{rev}(a)).$$

Thus, reversal is an involution on the set of valid arrays.

Every non-palindromic array belongs to a pair

$$\{a, \mathrm{rev}(a)\}.$$

Exactly one of the two arrays is lexicographically smaller than the other. A palindromic array is equal to its reverse and therefore is never niuean.

If P denotes the number of valid palindromic arrays, then the required answer is

$$\frac{T - P}{2}.$$

It remains to count valid palindromes.

Consider first a palindrome of even length:

$$b_1, b_2, \ldots, b_r, b_r, \ldots, b_2, b_1.$$

Let

$$C = B(b_r) \cdots B(b_2)B(b_1).$$

By the array-to-pair bijection, the first half corresponds uniquely to a coprime pair

$$Cv = \begin{pmatrix} q \\ p \end{pmatrix}, \qquad 1 \le p < q.$$

The matrix product for the entire palindrome is

$$C^{\mathsf{T}}C.$$

Therefore,

$$N(a) + 2 = v^{\mathsf{T}}C^{\mathsf{T}}Cv = (Cv)^{\mathsf{T}}(Cv) = p^2 + q^2.$$

Conversely, every coprime pair $p < q$ uniquely determines the first half of the palindrome, and hence the whole palindrome.

Thus, we have another bijection:

$$\boxed{\text{even palindromes} \longleftrightarrow \{(p, q) : 1 \le p < q,\ \gcd(p, q) = 1\}.}$$

Under this bijection, the validity condition is

$$p^2 + q^2 \le M.$$

Define

$$R(X) = \sum_{\substack{y \ge 1 \\ y^2 < X}} \left\lfloor \sqrt{X - y^2} \right\rfloor.$$

This counts ordered positive pairs (x, y) satisfying

$$x^2 + y^2 \le X.$$

To keep only coprime pairs, use the identity

$$[\gcd(x, y) = 1] = \sum_{d|\gcd(x,y)} \mu(d).$$

For a fixed common divisor d, the map

$$(x, y) \longleftrightarrow \left(\frac{x}{d}, \frac{y}{d}\right)$$

is a bijection between pairs divisible by d inside the original circle and arbitrary positive pairs satisfying

$$x^2 + y^2 \le \left\lfloor \frac{M}{d^2} \right\rfloor.$$

Hence the number of ordered coprime pairs is

$$C_{\text{even}} = \sum_{d^2 \le M} \mu(d) R\left(\left\lfloor \frac{M}{d^2} \right\rfloor\right).$$

Swapping the coordinates pairs every point with $p \neq q$. The only coprime point on the diagonal is

$$(1, 1).$$

Therefore, the number of points with $p < q$, and thus the number of even palindromes, is

$$P_{\text{even}} = \frac{C_{\text{even}} - 1}{2}.$$

Now consider an odd palindrome:

$$b_1, \ldots, b_r, c, b_r, \ldots, b_1.$$

The half-array may now be empty. By the extended bijection, the half-array corresponds uniquely to a pair

$$1 \le p \le q, \qquad \gcd(p, q) = 1.$$

The equality $p = q$ occurs only for the empty half, which corresponds to $(1, 1)$.

Using the same matrix C as before, the product for the whole palindrome is

$$C^{\mathsf{T}} B(c) C.$$

Thus,

$$N(a) + 2 = \begin{pmatrix} q & p \end{pmatrix} B(c) \begin{pmatrix} q \\ p \end{pmatrix}.$$

Since

$$B(c)\begin{pmatrix} q \\ p \end{pmatrix} = \begin{pmatrix} cq+p \\ q \end{pmatrix},$$

we get

$$N(a) + 2 = cq^2 + 2pq.$$

Conversely, the pair (p, q) uniquely reconstructs the half-array, while c uniquely determines the middle element. Therefore,

$$\boxed{\text{odd palindromes} \longleftrightarrow \{(p, q, c) : 1 \le p \le q,\ \gcd(p, q) = 1,\ c \ge 1\}.}$$

The validity condition is

$$cq^2 + 2pq \le M.$$

Fix q and define

$$u = \left\lfloor \frac{M}{q^2} \right\rfloor.$$

Only values

$$q \le \sqrt{M}$$

need to be considered.

If

$$1 \le c \le u - 2,$$

then every valid $p \le q$ satisfies the inequality, because

$$cq^2 + 2pq \le (u-2)q^2 + 2q^2 = uq^2 \le M.$$

The number of integers p such that

$$1 \le p \le q, \qquad \gcd(p, q) = 1$$

is $\varphi(q)$. We use the convention $\varphi(1) = 1$.

Therefore, all values $c \le u - 2$ contribute

$$\varphi(q)\max(0, u-2).$$

Values $c \ge u + 1$ are impossible because already

$$cq^2 > M.$$

Thus, only

$$c = u - 1 \qquad \text{and} \qquad c = u$$

must be checked separately.

For a fixed c, the condition becomes

$$p \le \left\lfloor \frac{M - cq^2}{2q} \right\rfloor.$$

Define

$$G(q, X) = \#\{1 \le p \le X : \gcd(p, q) = 1\}.$$

Let the distinct prime divisors of q be $r_1, \ldots, r_k$. By inclusion-exclusion,

$$G(q, X) = \sum_{S \subseteq \{1,\ldots,k\}} (-1)^{|S|} \left\lfloor \frac{X}{\prod_{i \in S} r_i} \right\rfloor.$$

Indeed, we start with all integers from 1 to X and exclude those divisible by at least one prime divisor of q.

Hence,

$$P_{\text{odd}} = \sum_{q^2 \le M} \left(\varphi(q) \max(0, u - 2) + \sum_{\substack{c \in \{u-1, u\} \\ c \ge 1}} G\left(q, \left\lfloor \frac{M - cq^2}{2q} \right\rfloor\right) \right),$$

where

$$u = \left\lfloor \frac{M}{q^2} \right\rfloor.$$

The total number of palindromes is

$$P = P_{\text{even}} + P_{\text{odd}}.$$

Therefore, the final answer is

$$\boxed{\frac{T - P_{\text{even}} - P_{\text{odd}}}{2}}.$$

It remains to calculate

$$\Phi(M) = \sum_{x=1}^{M} \varphi(x).$$

Since M may be as large as 10^{10}, iterating over all values is impossible. We use the Min25 sieve.

Min25 sieve works in

$$O\left(\frac{M^{3/4}}{\log M}\right)$$

time.

The Möbius function, Euler's totient function, and one prime divisor of every number up to $\sqrt{M}$ are precomputed by a sieve.

The lattice-point sums require

$$O(\sqrt{M}\log M)$$

time, while the inclusion-exclusion calculations require at most

$$O(\sqrt{M}\log^2 M)$$

time.

Thus, the total time complexity is

$$O\left(\frac{M^{3/4}}{\log M}+\sqrt{M}\log^2 M\right),$$

and the memory complexity is

$$O(\sqrt{M}).$$