Problem A. Ant and Sticks

Let

$$D = |AB| = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}.$$

If $M \geq 3$, the answer is -1. Indeed, three sticks can form an arbitrarily small triangle around A (or around B), so the ant can be blocked completely.

If $M = 2$, the answer is

$$D + 2W.$$

For the lower bound, place the two sticks as a very narrow V near A. Any path from A to B must first go around one arm, which costs almost W, and then still travel almost $D + W$ to reach B. So, $D + 2W$ is achievable in the limit.

For the upper bound, consider the planar drawing formed by the segment AB and the two sticks, splitting at intersections if needed. Take an ε-neighborhood of this drawing. Its outer boundary is a simple closed curve. Removing the two tiny arcs near A and B, it splits into two boundary paths joining points $O(\varepsilon)$-close to A and B. After adding these $O(\varepsilon)$ endpoint corrections, both become valid paths from A to B.

Every piece of the drawing is traced by the outer boundary at most twice, so the total length of these two paths is at most

$$2(D + 2W) + O(\varepsilon).$$

Hence one of them has length at most

$$D + 2W + O(\varepsilon).$$

Letting $\varepsilon \to 0$, we get $L \leq D + 2W$. Together with the lower bound, this gives $L = D + 2W$.

If $M = 1$, let the stick endpoints be P, Q. If the stick does not intersect $[A, B]$, then $[A, B]$ itself is still a valid path, so the shortest path is just D. Therefore, an optimal placement must intersect $[A, B]$. In that case, any path from A to B must go around one endpoint of the stick, so the shortest possible length is

$$\min(|AP| + |PB|,\ |AQ| + |QB|).$$

Now consider any $L > D$ and the ellipse

$$\mathcal{E}_L = \{X : |AX| + |XB| \leq L\}.$$

If one endpoint of the stick lies inside $\mathcal{E}_L$, then going around that endpoint gives a path of length at most L. So, if we want the shortest path to be greater than L, both endpoints must lie on or outside this ellipse.

Also, $[A, B] \subset \mathcal{E}_L$, so a single segment can block all paths inside $\mathcal{E}_L$ only if its supporting line cuts a chord of the ellipse whose line intersects AB. Therefore, the minimum stick length needed for threshold L is exactly the minimum length of such a chord.

A standard fact about ellipses says that the shortest such chord is the latus rectum, and its length is

$$\frac{L^2 - D^2}{L}.$$

Therefore

$$W \geq \frac{L^2 - D^2}{L}.$$

At the optimum we have equality, so

$$L^2 - WL - D^2 = 0,$$

which gives

$$L = \frac{W + \sqrt{W^2 + 4D^2}}{2}.$$

This bound is also achievable in the limit by placing the stick perpendicular to AB and moving it arbitrarily close to A or to B.

Therefore, the value of the answer is

$$\begin{cases} -1, & M \geq 3, \\ D + 2W, & M = 2, \\ \dfrac{W + \sqrt{W^2 + 4D^2}}{2}, & M = 1. \end{cases}$$

Problem B. Palindrome and Permutation

We want to make A a palindrome, but each element from 1 to N can only be used once.

So, if there is an index i such that $A_i \neq A_{N+1-i}$, the only possible way to make this opposite pair equal is to use one of the values A_i or A_{N+1-i}, but we cannot use both.

When there are several pairs of values like this, it is often helpful to look at them as edges of a graph, which is exactly what we will do.

Consider an undirected graph on the vertex set $\{1, \dots, N\}$, with one edge between A_i and A_{N+1-i} for each i such that $A_i \neq A_{N+1-i}$. Note that this graph might have multiple edges between the same pair of vertices.

For each edge, we need to make a choice on whether to use A_i or A_{N+1-i} to satisfy the opposite pair $(i, N+1-i)$. This can be thought of as orienting the edge in the appropriate direction; say we orient the edge towards the value used to satisfy the pair. Further, since each element from 1 to N can be used at most once, we have an additional constraint that after deciding on the orientation, each vertex must have an indegree of at most 1.

Let us now look at some connected component of the graph.

1. If the component is a tree, it is always possible to orient the edges appropriately.
2. If the component contains exactly one cycle, so it is a tree plus one edge, it is still always possible to orient the edges appropriately.
3. For any larger component, it is never possible to orient the edges appropriately, since the constraint on indegree at most 1 means the number of edges oriented within a component cannot exceed its size.

So, if any connected component of the graph we created has more edges than vertices, no permutation is valid, and the answer is simply

$$N! \cdot (N+1) = (N+1)!.$$

This leaves us with the case where each component is either a tree, or a tree plus one edge.

The minimum number of operations needed can then be analyzed as follows:

- If the component is a tree plus one edge, then every vertex in the component must have an indegree of exactly 1. This means we must see every value of the component before we reach a palindrome.
- If the component is just a tree, then we can satisfy all edges without ever seeing one of the values. So, we are able to skip the last occurring value in each tree component.

Thus, if we want $f(A, P) = k$, we must have the following:

1. Every value that lies in a non-tree component must appear at or before index k in P.
2. For every tree component, at most one value can appear after index k in P.

Note that such a configuration guarantees $f(A, P) \leq k$ rather than exactly equal to k, but that can be worked around easily enough in the end by subtracting out the configurations with value $< k$.

Finally, we need to be able to actually count the number of configurations with k fixed.

Observe that once k is fixed, what we are looking for is equivalent to this: pick $(N-k)$ of the tree components, and for each of them set aside exactly one vertex which will appear after index k. Everything else will appear before index k, including all vertices belonging to non-tree components.

Once the elements at indices $\leq k$ and $> k$ have been decided, they can be freely reordered, so that gives

$$k! \cdot (N-k)!$$

configurations.

As for actually computing the number of ways of picking one vertex each from $(N - k)$ trees: the constraints allow $O(N^2)$, so a simple dynamic programming of the form

$$dp[i][j] = \text{number of ways of picking } j \text{ vertices from the first } i \text{ tree components}$$

will do the job. It may be noted that due to the nature of the transitions, it is possible to optimize this solution to $O(N \log N)$ using NTT, but that was not required to get AC.

Finally, one potential issue that may come to mind is dealing with isolated vertices of the graph, since technically whenever we process an isolated vertex, we are not really assigning a direction to any edge. However, it can be proved that there always will be a way to perform the assignment that never actually causes any issues, since we can just write the values of such isolated vertices into whichever is the final index we are planning to operate on, so that the value just gets overwritten in the end.

Problem C. Score Queries

For a subarray $A_l, A_{l+1}, \ldots, A_r$ and $l < i < r$, index i contributes to score$(A_l, \ldots, A_r)$ iff

$$\min(A_l, \ldots, A_{i-1}) + \min(A_{i+1}, \ldots, A_r) < 2A_i.$$

So a query $[L, R]$ asks for the number of triples (l, i, r) with

$$L \le l < i < r \le R$$

satisfying this inequality.

For a fixed center i, consider the following four arrays.

- previous_smaller$[i]$, the nearest position left of i with value $< A_i$,
- next_smaller$[i]$, the nearest position right of i with value $< A_i$,
- previous_smaller_or_equal$[i]$, the nearest position left of i with value $\le A_i$,
- next_smaller_or_equal$[i]$, the nearest position right of i with value $\le A_i$.

If $l \le$ previous_smaller$[i]$ and $r \ge$ next_smaller$[i]$, then both side minima are already $< A_i$, so we immediately get one rectangle of valid pairs.

$$[1, \text{previous\_smaller}[i]] \times [\text{next\_smaller}[i], N].$$

Now suppose $l >$ previous_smaller$[i]$. Let j be the rightmost minimum of $[l, i-1]$. Then $A_j \ge A_i$, and there is no position in (j, i) with value $\le A_j$, so

$$\text{next\_smaller\_or\_equal}[j] = i.$$

And if next_smaller_or_equal$[j] = i$ and $A_j \ge A_i$, then j is the rightmost minimum exactly for

$$\text{previous\_smaller}[j] + 1 \le l \le j.$$

For those l, the left minimum is fixed to A_j, so we need

$$\min(A_{i+1}, \ldots, A_r) < 2A_i - A_j.$$

If s is the first position to the right of i with $A_s < 2A_i - A_j$, this gives the rectangle

$$[\text{previous\_smaller}[j] + 1, j] \times [s, N].$$

The case $r <$ next_smaller$[i]$ is symmetric. If j is the leftmost minimum of $[i+1, r]$, then

$$\text{previous\_smaller\_or\_equal}[j] = i,$$

and if previous_smaller_or_equal$[j] = i$, then j is that minimum exactly for

$$j \le r \le \text{next\_smaller}[j] - 1.$$

If t is the last position to the left of i with $A_t < 2A_i - A_j$, this gives the rectangle

$$[1, t] \times [j, \text{next\_smaller}[j] - 1].$$

These three types cover all valid pairs. Indeed, if

$$l > \text{previous\_smaller}[i] \qquad \text{and} \qquad r < \text{next\_smaller}[i],$$

then both side minima are at least A_i, so the inequality cannot hold.

So, for a fixed center i, every valid pair (l, r) belongs to exactly one of the three types. In Type 2, the index j is uniquely determined as the rightmost minimum of $[l, i-1]$. In Type 3, the index j is uniquely determined as the leftmost minimum of $[i+1, r]$. Therefore, each valid pair (l, r) is counted exactly once.

Hence, every contribution comes from one of the rectangles

- $[1, \text{previous\_smaller}[i]] \times [\text{next\_smaller}[i], N]$,
- $[\text{previous\_smaller}[j] + 1, j] \times [s, N]$ for some j with $\text{next\_smaller\_or\_equal}[j] = i$,
- $[1, t] \times [j, \text{next\_smaller}[j] - 1]$ for some j with $\text{previous\_smaller\_or\_equal}[j] = i$.

There are only $O(N)$ such rectangles.

- Type 1 gives at most one rectangle for each center i.
- Type 2 gives at most one rectangle for each index j, because once j is fixed, the center is forced to be $\text{next\_smaller\_or\_equal}[j]$.
- Type 3 gives at most one rectangle for each index j, because once j is fixed, the center is forced to be $\text{previous\_smaller\_or\_equal}[j]$.

For Type 2 and Type 3 we still need one boundary. We need the first or last position whose value is $< K$, where

$$K = 2A_i - A_j.$$

We can easily find all such positions in $O(N \log N)$ total.

Now, we have $O(N)$ rectangles $[a, b] \times [c, d]$ in the plane of pairs (l, r). For a query $[L, R]$, we want the number of points in the rectangle with

$$l \geq L, \qquad r \leq R.$$

For fixed R, let

$$g(R) = \max(0, \min(R, d) - c + 1),$$

which is exactly the number of allowed choices of r inside $[c, d]$.

Then one rectangle contributes

$$0 \quad \text{if } L > b,$$

$$((b + 1) - L)\, g(R) \quad \text{if } a \leq L \leq b,$$

$$(b - a + 1)\, g(R) \quad \text{if } L < a.$$

So when we sweep L from N down to 1, each rectangle has three states.

- inactive while $L > b$,
- active while $a \leq L \leq b$,
- full while $L < a$.

We can have three Fenwick-based range-add / prefix-query structures over the R-axis.

- active_count stores how many active rectangles cover each R,
- active_offset stores the sum of $(b + 1)$ over active rectangles covering each R,
- full_contrib stores the sum of $(b - a + 1)$ over full rectangles covering each R.

When a rectangle becomes active at $L = b$, add 1 to active_count and add $b+1$ to active_offset on the interval $[c, d]$.

When it becomes full at $L = a - 1$, remove it from the two active structures and add $b - a + 1$ to full_contrib on $[c, d]$.

Adding a value v on $[c, d]$ contributes

$$v \cdot \max(0, \min(R, d) - c + 1)$$

Therefore the answer to query $[L, R]$ is

$$\text{full\_contrib.pref}(R) + \text{active\_offset.pref}(R) - L \cdot \text{active\_count.pref}(R).$$

Each rectangle changes state only a constant number of times, and each query is answered once. So, the sweep runs in $O((N + Q) \log N)$.

Problem D. Merging Maximum

Each operation merges three consecutive elements into their maximum.

This gives us the following:

Observation: Suppose we perform several merge operations on P, and end up with the array $[B_1, B_2, \ldots, B_k]$. Then, each B_i will correspond to the maximum element of some **odd-length** subarray of P, and these subarrays partition P.

Formally, these will exist ranges $[l_1, r_1], \ldots, [l_k, r_k]$ such that:

- $r_i + 1 = l_{i+1}$ for each $1 \le i < k$
- $l_1 = 1$ and $r_k = N$
- $r_i - l_i + 1$ is odd for each i
- $B_i = \max(P_{l_i}, \ldots, P_{r_i})$ for each i

That is, every reachable array can be obtained by taking a partition of P into odd-length subarrays, and then replacing each subarray with its maximum.

This leads to a natural DP solution. Let $dp[i]$ be the number of sequences obtainable from the prefix of length i of P, and compute $dp[i]$ by considering every odd-length subarray of the form $[j, i]$. This would simply result in $dp[i] = dp[i-1] + dp[i-3] + dp[i-5] + \ldots$ However, this is clearly not entirely correct, because it counts *partitions* and not the resulting arrays - but it is possible that two different partitions result in the same final array when maximums are taken (for example, with $P = [3, 1, 2, 4]$, the array $[3, 4]$ can be reached in two ways).

To get around this, we canonicalize reachable sequences. That is, when trying to transition using $j \le i$ while computing $dp[i]$, we only count sequences for which $[j, i]$ is the *shortest* possible final subarray in the partition. Once $[j, i]$ is fixed, there are $dp[j-1]$ sequences before it. However, to ensure that $[j, i]$ is shortest, we need to take care not to count those sequences that can be obtained by choosing $[j+2, i]$ instead, since that would be a shorter final subarray.

First, note that this is only possible if $[j, i]$ and $[j+2, i]$ have the same maximum element. In that case, a sequence counted using $[j, i]$ is overcounted exactly when the prefix before the last block can also be extended through $j+1$, so that the last block may instead start at $j+2$. Therefore, the sequences we need to subtract are precisely those obtainable from a prefix ending at $j-1$ whose last element is $\ge \max(P_j, P_{j+1})$. Those are exactly the sequences that can also be viewed as ending at $j+1$.

Since this correction term is independent of i, it's possible to compute and store it for each index while computing the DP. Concretely, for each prefix ending at t we maintain how many canonical sequences end there with each possible last value. Then the correction for $j = t+1$ is just the suffix sum over values at least $\max(P_j, P_{j+1})$. The final answer is $dp[N]$. The time complexity is $O(N^2)$.

Problem E. Optimal Splitting

Suppose $|X| \geq |Y|$. After padding Y with a's until $|Y| = |X|$, the answer is

$$Z_i = \max(X_i, Y_i).$$

So, we only need the lexicographically smallest feasible string Z.

Now, suppose the prefix $Z_1 Z_2 \ldots Z_i$ is already fixed. For every $0 \leq j \leq i$, let $dp[i][j] = 1$ mean that the first $i + j$ characters of S can be split into two subsequences L and R such that

- $|L| = i$ and $|R| = j$,
- for every $1 \leq t \leq j$, we have $\max(L_t, R_t) = Z_t$,
- for every $j < t \leq i$, we have $L_t = Z_t$.

So, the first j positions are already matched, while $L_{j+1} \ldots L_i$ is still unmatched. Initially only $dp[0][0]$ is reachable.

If $dp[i][j] = 1$, the next unused character is S_{i+j+1}. As long as $j < i$ and

$$S_{i+j+1} \leq Z_{j+1},$$

we may put this character into R and move to $dp[i][j+1]$. Hence, for a fixed i, we keep extending the same row with this move.

So, every legal way to postpone the creation of Z_{i+1} is already represented by some reachable state in row i.

Once this postponement stops, the next unused character must go to L, so it becomes Z_{i+1}. Therefore,

$$Z_{i+1} = \min\{\, S_{i+j+1} \mid dp[i][j] = 1,\ i + j < N \,\}.$$

No smaller value is possible, because every feasible completion must create the next answer character from some reachable state in this row. This minimum is feasible as well. Take a state attaining it, put that character into L, and put every remaining unused character of S into L too. The characters already placed in R stay within the required bounds, and the missing suffix of R is padded with a's.

After fixing Z_{i+1}, we can compute row $i + 1$ directly. First mark every j with $dp[i][j] = 1$ and

$$S_{i+j+1} = Z_{i+1},$$

because these are exactly the states that create the new answer character. Then continue the same left-to-right scan. While $j < i + 1$ and

$$S_{(i+1)+j+1} \leq Z_{j+1},$$

the character may be appended to R, so we move to $j + 1$. This gives exactly the reachable states for the next prefix.

As soon as some reachable state satisfies

$$i + j = N,$$

all characters of S have been assigned, so the current prefix is already the full answer. Padding the shorter subsequence with a's does not change any maximum.

The time complexity is $O(N^2)$.

Problem F. Restricted Removals

Let's consider some B. Let the zero positions of B be

$$z_1 < z_2 < \cdots < z_Z,$$

where $Z = N - K$, and set $z_{Z+1} = N + 1$.

For each r, the interval $[z_r, z_{r+1}-1]$ is one block. The prefix before z_1 is protected forever. Since B is applied to current positions, every time Bob chooses the current index z_r, he removes the leftmost remaining element of that block. So, from each block he can take only a prefix. Also, by processing the blocks from right to left, Bob can realize any chosen prefix length in every block.

Hence, for this fixed B, Bob's best score is

$$\sum_{r=1}^{Z} g(z_r, z_{r+1} - 1), \qquad g(l, r) = \max\left(0, \max_{l \le x \le r} \sum_{i=l}^{x} A_i\right).$$

So Alice is choosing a partition of some suffix of A into exactly Z nonempty consecutive blocks, and the answer is the minimum possible sum of these block costs.

Let

$$S_0 = 0, \qquad S_j = \sum_{i=1}^{j} A_i.$$

For a block $[i + 1, j]$,

$$g(i + 1, j) = \max_{i \le h \le j} (S_h - S_i),$$

where $h = i$ means taking nothing from the block.

Now, define

$$dp[t][j]$$

to be the minimum total cost of partitioning some suffix of $A_1, \ldots, A_j$ into exactly t blocks. Then $dp[0][j] = 0$, and for $t \ge 1$,

$$dp[t][j] = \min_{0 \le i < j} \left(dp[t-1][i] + \max_{i \le h \le j} (S_h - S_i)\right).$$

The answer is $dp[Z][N]$.

For fixed t and split position i, the contribution to $dp[t][j]$ is

$$(dp[t-1][i] - S_i) + M_i(j), \qquad M_i(j) = \max_{i \le h \le j} S_h.$$

When j increases, $M_i(j)$ either stays the same or becomes S_j. So, many split positions share the same current maximum, and we can group them by this value. For each group, it is enough to store the minimum value of $dp[t-1][i] - S_i$.

When S_j arrives, all suffix groups whose maximum is at most S_j merge into one group with maximum S_j. Prefix minima of

$$\text{group maximum} + \text{best base}$$

then give $dp[t][j]$ in $O(1)$.

Hence, the total time complexity is $O(N^2)$.

Problem G. Sequence Domination

Let

$$d_i = A_i - B_i.$$

For a fixed value $d_i = t$, the number of pairs (A_i, B_i) with $A_i - B_i = t$ is exactly

$$M - |t|.$$

So the answer is the weighted count of all valid difference arrays d, where the weight of one array is

$$\prod_{i=1}^{N}(M - |d_i|).$$

Thus we only need to characterize which arrays d are valid.

Claim. The difference array d is valid if and only if for every k,

$$c_k = d_k + d_{k-1} + 2d_{k-2} + 4d_{k-3} + \cdots \geq 0.$$

To prove this, it is convenient to parametrize super decreasing sequences by independent nonnegative variables. For a super decreasing sequence V, define

$$U_i = V_i - \sum_{j=i+1}^{N} V_j.$$

Then $U_i \geq 0$ for every i, and every nonnegative sequence U defines a unique super decreasing sequence V by

$$V_N = U_N, \qquad V_i = U_i + V_{i+1} + \cdots + V_N.$$

Also,

$$V_i = U_i + \sum_{k=i+1}^{N} 2^{k-i-1}U_k,$$

so in

$$\sum_{i=1}^{N} d_i V_i$$

the coefficient of U_k is exactly c_k. Hence

$$\sum_{i=1}^{N} d_i V_i = \sum_{k=1}^{N} c_k U_k.$$

Since the U_k can be any nonnegative integers independently, this is nonnegative for all super decreasing V if and only if every c_k is nonnegative.

Now, define

$$w_1 = 0, \qquad w_{i+1} = d_i + 2w_i.$$

Then

$$w_i = d_{i-1} + 2d_{i-2} + 4d_{i-3} + \cdots,$$

so the claim says exactly that when we choose d_i, it must satisfy

$$d_i + w_i \geq 0.$$

Let

$$S = M - 1.$$

Since every difference satisfies $-S \le d_i \le S$, once $w_i \ge S$ we always have

$$w_{i+1} = d_i + 2w_i \ge -S + 2S = S.$$

So all states at least S can be merged into one saturated state.

We now process positions from left to right. Let

$$dp[s]$$

be the weighted count after processing the first $i-1$ positions. For $0 \le s < S$, it means that the current value of w_i is exactly s. For $s = S$, it means that the current value of w_i is at least S.

Initially $w_1 = 0$, so

$$dp[0] = 1.$$

If the new exact state is $t < S$, then

$$t = d + 2s \implies d = t - 2s.$$

Also, the condition $d + s \ge 0$ becomes $t - s \ge 0$, so $s \le t$. Therefore

$$ndp[t] = \sum_{s=0}^{t} dp[s] \cdot (M - |t - 2s|).$$

Let

$$h = \left\lfloor \frac{t}{2} \right\rfloor.$$

The weight $M - |t - 2s|$ is linear in s on each side of h, namely $(M - t) + 2s$ for $s \le h$ and $(M + t) - 2s$ for $s > h$. So, with prefix sums of $dp[s]$ and $s \cdot dp[s]$, each exact state is computed in $O(1)$ time.

The saturated state is handled separately. From an already saturated state, every difference is valid, and the total weight is

$$\sum_{d=-(M-1)}^{M-1} (M - |d|) = M^2,$$

so this contributes $M^2 \cdot dp[S]$.

From an exact state $s < S$, we move to saturation when

$$d + 2s \ge S \iff d \ge S - 2s,$$

so the contribution is

$$dp[s] \cdot \sum_{d=S-2s}^{S} (M - |d|),$$

and these values can be precomputed.

So, each layer costs $O(M)$, and the total time complexity is $O(NM)$.

Problem H. Maximum Difference

Let the distinct values currently present in the array A be

$$v_1 < v_2 < \cdots < v_k.$$

We call the cut between v_i and v_{i+1} bad if some element at least v_{i+1} appears before some element at most v_i.

The key observation is

$$\text{score}(A) = \max_{1 \le i < k,\ \text{cut } i \text{ bad}} (v_{i+1} - v_i),$$

with value 0 if there is no bad cut.

Indeed, if $v_{i+1} - v_i > D$, then an operation of cost at most D cannot touch both sides of that cut, so a bad cut of that size can never be fixed.

Now, let D be the maximum bad gap. Any cut with gap $> D$ is already good, so those cuts split the present values into groups that already appear in the correct relative order. Inside one group, consecutive present values differ by at most D. By repeatedly sorting all occurrences of two consecutive present values, we can sort the whole group using only operations of cost at most D. Since different groups were already ordered correctly, the whole array becomes non-decreasing.

So, after each update we only need to know the maximum bad gap.

Now, define for every $x \in [1, N]$,

$$\text{crossings}(x) = \#\{\, i \mid 1 \le i < N,\ A_i > x \ge A_{i+1} \,\}.$$

If we replace each element by 1 when it is $> x$, and by 0 otherwise, then the cut at x is bad exactly when this binary sequence contains a 1 before a 0, equivalently an adjacent pair $1, 0$. Hence

$$\text{the cut at } x \text{ is bad} \iff \text{crossings}(x) > 0.$$

If $u < v$ are consecutive present values, then no value in (u, v) appears, so $\text{crossings}(x)$ is constant on the whole interval $[u, v-1]$. Therefore, the answer is the maximum length of a block $[u, w]$ on the value axis such that

- u is present,
- every value in $(u, w]$ is absent,
- $\text{crossings}(u) > 0$.

If the next present value after u is v, then this block is exactly $[u, v-1]$, and its length is $v-u$, which is exactly the gap from the first observation.

This reduction is useful because updates are local.

- every adjacent descent $A_i > A_{i+1}$ adds 1 to $\text{crossings}(x)$ for all $x \in [A_{i+1}, A_i - 1]$,
- after changing one position p, only the pairs $(p-1, p)$ and $(p, p+1)$ can change.

So, one update causes at most four range additions on crossings, plus two point changes in value presence.

To maintain the answer, we use a segment tree on the value axis $[1, N]$. For each node I, we store only the information needed to reconstruct the best block inside I.

- whether I contains any present value,
- the minimum crossing count on I,
- the length of the longest absent prefix of I,
- the rightmost block ending at the right boundary of I, namely its length and its crossing count,
- the longest block contained in I,

- the longest block in I whose starting crossing count is strictly larger than the minimum crossing count of I.

When merging two children L and R, every relevant block falls into one of three types.

- entirely inside L,
- entirely inside R,
- the rightmost block of L, extended through the absent prefix of R.

That is enough to recompute all node information in $O(1)$. A lazy range add only shifts crossing counts, so it changes only the fields that depend on crossings, not the stored lengths.

Each update takes $O(\log N)$, so the total time complexity is

$$O((N + M) \log N).$$

Problem I. Point Mirror

For $N = 1$, the answer is simply whether $A_1 = B_1$.
Let's assume $N \geq 2$, and define

$$g(X) = \gcd_{1 \leq i < j \leq N} |X_i - X_j|.$$

If one configuration is reachable from another, two invariants must hold.
First, $g(A)$ is invariant. A swap does nothing. After

$$A_i \leftarrow 2A_j - A_i,$$

we have

$$(2A_j - A_i) - A_j = -(A_i - A_j),$$
$$(2A_j - A_i) - A_k = 2(A_j - A_k) - (A_i - A_k),$$

so every new difference is divisible by the old gcd. Since the mirror operation is invertible, the gcd is unchanged.
Second, if $g = g(A)$, then the multiset of residues modulo $2g$ is invariant, because

$$2A_j - A_i \equiv A_i \pmod{2g}.$$

So a mirror keeps the moved point in the same residue class modulo $2g$, and a swap only permutes these residue classes.
Let's assume these invariants match for A and B.
If $g(A) \neq g(B)$, the answer is `NO`.
If $g(A) = g(B) = 0$, then every coordinate in each configuration is the same. Mirror does nothing, so only swaps remain, and we only need the sorted multisets to match.
From now on let $g = g(A) = g(B) > 0$. Subtract a common constant from every coordinate and divide by g. Reachability does not change, the gcd becomes 1, and the invariant modulo $2g$ becomes just the parity multiset.
If $N = 2$, the two points already differ by 1. From $(p, p+1)$, one mirror and possibly a swap gives $(p+1, p+2)$ or $(p-1, p)$, so any adjacent pair with the same parity multiset is reachable.
Now, let $N \geq 3$. After sorting the current coordinates, let

$$X_1 \leq X_2 \leq \cdots \leq X_N,$$

and let $d_k = X_{k+1} - X_k$. Their gcd is 1.
On three consecutive points

$$x,\ x+u,\ x+u+v,$$

mirroring the left one across the middle one changes the adjacent gaps (u, v) into

$$(\min(u, v), |u - v|).$$

This is one Euclidean algorithm step, so repeating it eventually creates an adjacent gap equal to 1.
Now, suppose we have two points p and $p+1$ with difference 1. Then

$$x \xrightarrow{\text{mirror across } p} 2p - x \xrightarrow{\text{mirror across } p+1} x + 2,$$

and reversing the order gives $x - 2$. So, after normalization, every point can be moved to p or $p+1$ according to its parity. Hence, every normalized configuration can be reduced to the

same canonical form, where some points are at p and the rest are at $p+1$, with multiplicities determined by the parity counts.

Using the same two anchors, we can also move each non-anchor point from that canonical form to any target configuration with the same parity multiset. So, for $N \geq 2$, two configurations are reachable from each other if and only if

$$g(A) = g(B) = g$$

and the multisets

$$\{A_i \bmod 2g\}_{i=1}^{N} \quad \text{and} \quad \{B_i \bmod 2g\}_{i=1}^{N}$$

are equal.

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

Problem J. Counting Is Fun

Let's consider some non-empty subsequence B of A and sort its values as

$$C_1 \le C_2 \le \cdots \le C_M.$$

The score depends only on the multiset of chosen values, so this loses nothing.

Claim 1.

$$\text{score}(B) = \max_{1 \le i \le M} (C_i + M - i).$$

For the lower bound, we can fix i. The elements $C_i, C_{i+1}, \ldots, C_M$ each need at least C_i decrements, so all of their last operations happen in rounds at least C_i. These last operations must be distinct, because in the last operation for each element its current value is 1. So, among these $M - i + 1$ elements, the latest last round is at least

$$C_i + (M - i).$$

For the upper bound, let

$$T = \max_{1 \le i \le M} (C_i + M - i), \qquad f_i = T - (M - i).$$

We can decrease C_i in rounds

$$f_i - C_i + 1, f_i - C_i + 2, \ldots, f_i.$$

This is valid because $f_i \ge C_i$ and $f_1 < f_2 < \cdots < f_M$. In round t, if the i-th element is chosen, its current value is $f_i - t + 1$, so the chosen values in that round are pairwise distinct. Thus we can finish in exactly T rounds.

This also gives a useful observation. If a subsequence S has score x, and we add one more value y with

$$\max(S) \le y,$$

then the new score becomes

$$\max(x + 1, y).$$

Let's sort the whole array.

$$A_1 \le A_2 \le \cdots \le A_N.$$

This does not affect the answer.

Now, let

$$g_0 = 0, \qquad g_i = \max(A_i, g_{i-1} + 1).$$

Then g_i is the score of the prefix $A_1, \ldots, A_i$, and the sequence g is strictly increasing.

Claim 2. Every score of a subsequence of $A_1, \ldots, A_i$ belongs to

$$\{g_0, g_1, \cdots, g_i\}.$$

We prove this by induction on i. If a subsequence does not use A_i, we are done. Otherwise, remove A_i. By induction the remaining score is some g_j, and the previous observation says that after adding A_i back the new score is

$$\max(A_i, g_j + 1).$$

Let p_i be the first index with $g_{p_i} \ge A_i$. Since $g_i \ge A_i$, such an index exists. Also $g_{p_i - 1} < A_i$ and $A_{p_i} \le A_i$, so

$$g_{p_i} = \max(A_{p_i}, g_{p_i - 1} + 1) = A_i.$$

If $j < p_i$, then $g_j < A_i$, so the new score is g_{p_i}. If $j \geq p_i$, then $A_{j+1} \leq A_i < g_j + 1$, so $g_{j+1} = g_j + 1$ and the new score is g_{j+1}. So, no score outside $\{g_0, \ldots, g_i\}$ can appear.
This leads to a natural dynamic programming solution. Now, let

$$dp[i][t]$$

be the number of subsequences of the prefix of length i whose score is exactly t. Then

$$dp[i+1][t] \mathrel{+}= dp[i][t], \qquad dp[i+1][\max(t+1, A_{i+1})] \mathrel{+}= dp[i][t].$$

By Claim 2, after processing the first i elements there are only $O(i)$ reachable scores. So, one layer can be kept in a map from score to count. This gives an $O(N^2 \log N)$ solution, which passes comfortably. If we want $O(N^2)$, the same reachable states can be stored explicitly, for example as a vector of pairs.
Hence, the final answer is

$$\sum_t dp[N][t] \cdot t \pmod{998244353}.$$

Problem K. Prefix MEX Equation

First we must understand when the pair of arrays (X, Y) can have equal MEX-es (not even prefix MEX arrays; just $\text{MEX}(X) = \text{MEX}(Y)$) after some pointwise swaps. The following can be proved.

Lemma.

Let $\text{freq}[x]$ denote the number of times x appears in X and Y combined. Let m be the smallest non-negative integer such that $\text{freq}[m] < 2$. Then, $\text{MEX}(X) = \text{MEX}(Y)$ after some swaps of X_i with Y_i if and only if $\text{freq}[m] = 0$.

Equivalently, the smallest element that appears exactly once must be larger than the MEX of the union of arrays.

For X and Y to be able to have equal prefix MEX arrays after some swaps then, one necessary condition is that the above criterion must hold for every prefix of the arrays. It can be shown that this is sufficient as well (to construct a valid sequence of swaps, consider the sequence of indices where the prefix MEX is forced to change, from left to right, and try to fix swaps between each of them).

We now try to count the number of ranges for which this condition holds on each prefix. Let's fix L, the left endpoint of the range. The set of valid right endpoints R will clearly form a contiguous range starting at L, so to count them it suffices to find the maximum valid R.

Define $\text{first}[x]$ to be the smallest index $i \geq L$ such that either $A_i = x$ or $B_i = x$. Define $\text{second}[x]$ to be the second-smallest index $i \geq L$ containing x. Note that $\text{first}[x] = \text{second}[x]$ is allowed if $A_i = B_i = x$, since we care about overall frequency.

Observe that the range $[L, R]$ will be "killed" by a value x if and only if $\text{first}[x] \leq R < \text{second}[x]$, and also each of $\text{first}[0], \text{first}[1], \ldots, \text{first}[x-1]$ must be $\leq R$. That is, x must have only one occurrence in $[L, R]$, and also the MEX must be $> x$.

This means the value x can "kill" a right endpoint R only if $\max(\text{first}[0], \ldots, \text{first}[x]) < \text{second}[x]$. In particular, all we need to do is find the smallest x that satisfies this criterion: this will give us the first endpoint that's invalid, which is what we want.

This can be maintained using, for example, a segment tree. Store the values of $\max(\text{first}[0], \ldots, \text{first}[x]) - \text{second}[x]$ in a segment tree built over x; we want to find the leftmost x for which this is negative (which is doable by storing range maximums).

As for updates, when moving the left endpoint from L to $L+1$, the only things that change are $\text{first}[x]$ and $\text{second}[x]$ for $x = A_i, B_i$. One way to process these easily is to store $\max(\text{first}[0], \ldots, \text{first}[x])$ separately from $\text{second}[x]$, and then also store their difference. Updating $\text{first}[x]$ then corresponds to a range-set operation on the first term, and updating $\text{second}[x]$ corresponds to a point update.

Problem L. Alice and Bob

Bob's pairs $(X_1, Y_1), \ldots, (X_N, Y_N)$ form a perfect matching M on the $2N$ vertices. Consider the tree rooted at vertex 1, and let depth$[v]$ be the depth of v. Then for any pair (p, q),

$$\text{dist}(p, q) = \text{depth}[p] + \text{depth}[q] - 2 \cdot \text{depth}[\text{lca}(p, q)],$$

where $\text{lca}(p, q)$ is the lowest common ancestor of p and q. Therefore

$$\sum_{(p,q)\in M} \text{dist}(p, q) = \sum_{x=1}^{2N} \text{depth}[x] - 2 \sum_{(p,q)\in M} \text{depth}[\text{lca}(p, q)].$$

So every attainable total has parity

$$P = \sum_{x=1}^{2N} \text{depth}[x] \pmod 2,$$

which depends only on the tree.

We can also look at the contribution of each edge. This gives the minimum and maximum attainable totals, and in fact every value between them with the same parity as the minimum is attainable. We only need the parity statement here.

If $K \not\equiv P \pmod 2$, then K is already impossible, so Alice can delete and re-add the same edge.

It remains to flip the parity P. We can take any leaf x, let p be its parent, and replace the edge (x, p) by an edge from x to any other neighbor of p. This keeps the graph a tree, and only depth$[x]$ changes, by exactly 1. Therefore, P flips. After the modification, every attainable total has parity $1 - P \not\equiv K \pmod 2$, so Bob can no longer obtain a total distance of K.

The time complexity is $O(N)$.

Problem M. Vertex Separation

Clearly, for a separable pair to exist the graph must contain a cycle, since in a tree there is a unique simple path between any pair of vertices. So, we focus on graphs that contain cycles for now.

Observation: If a connected graph contains a cycle of length ≥ 4, then every vertex is separable.
Proof: Let the cycle be $(v_1, \ldots, v_k)$ in order. For any vertex v, take the shortest path from v to a cycle vertex v_i; let this path be P.
Observe that the paths $P \to v_{i+1}$ and P followed by the longer way around the cycle from v_i to v_{i+1} differ in length by at least 2.
Thus, the *interesting* graphs are quite restricted: they can only have cycles of length 3 (a.k.a triangles). Further, note that any two triangles cannot share an edge, since if they do, a cycle of length 4 will be formed. They can, however, share a vertex.

Observation: If a connected graph has its only cycles be edge-disjoint triangles, we can say the following:

- If the graph contains at most one triangle, no vertex is separable.
- If the graph contains ≥ 2 triangles, the separable vertices can be determined as follows:
  - Compress each triangle into a single new vertex, with edges to its three corresponding vertices. The graph created this way will be a tree.
  - v is separable if and only if there exists a path in this tree that contains at least two triangle vertices and has v as one endpoint. In particular, at least two vertices corresponding to each triangle will be separable.

Now, consider any spanning tree of the graph. Each extra edge added to the spanning tree will define a cycle. Since the triangles must be edge-disjoint, their count is thus uniquely determined to be $M - (N - 1)$. Let $t = M - (N - 1)$ be the number of triangles we have.

Each triangle uses up two edges of the spanning tree, so we must have $2t \leq N - 1$ to even be able to form t edge-disjoint triangles. A minimal construction is now to create the triangles $(1, 2, 3), (1, 4, 5), (1, 6, 7), \ldots, (1, 2t, 2t + 1)$, which has $2t$ separable vertices and 1 non-separable vertex (that being vertex 1). As for the remaining vertices: observe that attaching a vertex directly to 1 will make the new vertex non-separable, while attaching it to 2 will make it separable. Thus, the remaining $N-(2t+1)$ vertices can freely be separable or not, so as long as $2t \leq K < N$ we can find an appropriate solution using these t triangles. Note that as discussed above, the only edge case here is $t \leq 1$, where only $K = 0$ has a solution.

That leaves the case of $K = N$, i.e. we want every vertex to be separable. For this, the graph must contain a cycle of length at least 4, so if $N \leq 3$ or $M = N-1$ no solution exists. Otherwise a solution always does: a simple construction is to take the cycle $(1, 2, \ldots, N)$ and then simply keep adding edges till M edges are reached.