Problem A. Ancestors And Queries

Idea, code, and data by myee. Solution by myee. Expected difficulty: Medium.

The target complexity is $O(n \log n)$ (considering n, m, q to be of the same order), but solutions with small constants and an extra log factor are accepted. The main solution is based on offline divide-and-conquer + virtual tree.

Given a rooted tree with n nodes (rooted at 1), a sequence of length m with values in $[1, n]$, and q queries: $\max_{1 \le i \le r} \mathrm{LCA}(i, u)$. Constraints: $1 \le n, m, q \le 5 \times 10^5$, 5 seconds / 1GB.

Consider building a segment tree over the sequence. The answer for each query is determined by the maximum over $O(\log n)$ nodes in the segment tree. Therefore, we only need to compute the answer for each node in the segment tree. In this context, consider building the virtual tree for all points in the interval. Then each query becomes: query the "maximum node id on the upward chain in the virtual tree if the query point is also added to it".

Process offline. When building the virtual tree, also incorporate the query points. Record which points can be built solely based on points within the interval, and which points have a descendant within the interval. Then perform DP to compute, for each point, the maximum id and the last id on its upward chain among such points. This allows answering each query in $O(1)$.

For simplicity, we can always include the root node in the virtual tree and treat it as a point within the interval, since the problem guarantees the root id is 1.

Using two-way separation and two-way merging, we can ensure the query sequence is in DFS order, thus allowing us to build each virtual tree in linear time using $O(1)$ LCA queries (e.g., via an ST table).

Thus, the problem can be solved in $O((n + q) \log n)$.

Problem B. Borders And Center

Given a string s of length n, answer q queries: the sum of the number of borders for substrings centered at a given position.

$$n \le 10^6, \quad q \le 20.$$

Consider how to compute the answer for a given k in $O(n)$. Transform the problem: each pair of equal substrings $s[l_1 \dots r_1], s[l_2 \dots r_2]$ contributes 1 to the center $k = (l_1 + r_2)/2$. We also refer to $(l_1 + r_2)/2$ as the center of this pair of substrings.

Then, reflect s about k and mix the new string with the original. Specifically, let $t_i = s_{2k-i}$ be the reflected string, with indices from $2k - n$ to $2k - 1$. Its intersection with the original string s (indices 1 to n) is from L to R. The mixed string is $s_L t_L s_{L+1} t_{L+1} s_{L+2} t_{L+2} \dots s_R t_R$.

We find that in this mixed string, each even palindrome whose right endpoint is a character in t corresponds one-to-one with a pair of equal substrings of s centered at k. Therefore, the answer is simply the number of such even palindromes, which can be computed using Manacher's algorithm.

Problem C. Covering Game

There is an $n \times n$ grid with n black cells, the rest are white. Any two black cells are in different rows and columns.

For any $k \in \mathbb{Z}_+$, you have infinitely many $1 \times k$ small rectangles (horizontal strips) and infinitely many $k \times 1$ small rectangles (vertical strips). Horizontal and vertical strips are collectively called strips.

Specifically, a 1×1 small rectangle can be either a horizontal or a vertical strip, but they are considered different states (i.e., placed horizontally or vertically).

You need to cover all white cells with your strips without overlap, and you cannot cover black cells. Count the number of essentially different covering schemes that use exactly $2n - 2$ strips. Two coverings are considered essentially different if there exists a white cell where the strip covering it in the first scheme is different from the strip covering it in the second scheme. Output the answer modulo $10^9 + 7$.

Consider a covering. First define a term: direction. For a horizontal strip, we say it points to the black cell in its row. For a vertical strip, we say it points to the black cell in its column.

We then place arrows on each white cell: $\leftrightarrow$ if covered by a horizontal strip, $\uparrow$ if covered by a vertical strip.

First prove a lemma: To satisfy the covering requirement, at least $2n-2$ strips are needed. The main idea is to consider how many strips point to each black cell.

Case 1: There exists a black cell not pointed to by any strip.

As shown in the figure, the arrows for white cells in the same row and column as this black cell are determined, and the corresponding strips are all distinct, so there are at least $2n-2$ strips.

Case 2: Every black cell is pointed to by at least one strip, and there are two black cells each pointed to by only one strip (not necessarily the same strip).

As shown in the figure, assume two black cells are each pointed to by only one strip. Consider all white cells within the two "crosses" formed by these two black cells. First, look at a white cell that is in the same row/column as both black cells, e.g., the red cell in the upper right. Suppose its arrow is $\leftrightarrow$. Then the blue cell in the lower left must have arrow $\leftrightarrow$, and other arrows in the figure can be completely determined. (The red and blue arrows here are unrelated to the colors in the problem statement.)

Let there be u columns between these two black cells. Count the minimum number of strips needed. Vertical strips need at least $n-2-u$. Ignoring the u columns between the black cells, horizontal strips need at least n. However, the u black cells in the middle columns will split horizontal strips that might originally span these columns, requiring at least u additional horizontal strips. Total: at least $(n-2-u)+n+u=2n-2$ strips.

Case 3: Every black cell is pointed to by at least one strip, and at most one black cell is pointed to by only one strip (i.e., other black cells are pointed to by at least two strips).

Note that each strip points to exactly one black cell. Simple double counting shows at least $2n-1$ strips are needed.

The lemma is proven. Now return to the original problem.

Since the problem requires using exactly $2n-2$ strips, which is the minimum, we only need to consider cases where equality is achieved in Case 1 and Case 2. That is, all strips must be obtained by extending from the marked $\leftrightarrow$ and $\uparrow$ arrows.

From Case 1, the number of schemes can be divided into four parts: upper left, lower left, upper right, lower right. Multiply the scheme counts for these four regions. For Case 2, the middle u columns no longer need consideration; simply extend horizontal strips to cover fully. The two sides can again be reduced to multiplying the scheme counts of the four regions (upper left, lower left, upper right, lower right). Therefore, we only need to consider the scheme count for one black cell and its upper left/lower left/upper right/lower right region.

(However, in Case 2, the red arrow could also be $\uparrow$; those schemes must also be counted.)

Consider the upper right part. First, note that if two black cells are arranged in a "top-left to bottom-right" pattern (like the red cells in the figure), then there is a white cell (marked with $\times$ in the figure) that cannot be covered by extending from the marked black cells, making a covering with $2n-2$ strips impossible.

Therefore, the black cells in this part must follow a "bottom-left to top-right" trend. That is, for any two black cells (i_1, j_1), (i_2, j_2), if $i_1 > i_2$, then necessarily $j_1 < j_2$, where $(1,1)$ is the top-left corner of the $n \times n$ grid.

Checking this trend is simple: use a set and sweep from right to left, maintaining the predecessor, because the condition "satisfies this trend" can be passed from the predecessor.

Now assume the black cells indeed satisfy the "bottom-left to top-right" trend. Draw a valid covering. Consider the boundaries between horizontal and vertical strips; they exactly form a path from bottom-

left to top-right. This path touches the upper-right corners of all black cells and each step goes either up or right (see the red line in the figure). Similarly, any such path from bottom-left to top-right that touches the upper-right corners of all black cells and moves only up or right corresponds to an extension method.

Note: This is not a one-to-one correspondence, because if you reach the bottom-left corner of a black cell, the two ways to reach its upper-right corner are equivalent in terms of strip extension. However, counting this number of schemes has a strong recurrence, so we only need to consider the case of two black cells.

In the figure, suppose we need to go from point O to the upper-right corner. The step before reaching the destination could be to A or B, but $C \to A$ and $C \to B$ are the same, so we must subtract the number of ways from O to C.

In summary, for this local part, the number of strip extension schemes is: (ways from O to A) plus (ways from O to B) minus (ways from O to C).

Each small part is essentially a binomial coefficient of the form $\binom{a+b}{a}$. Preprocess these. For a cluster of black cells with the "bottom-left to top-right" trend, compute the scheme count between adjacent black cells (in the "bottom-left to top-right" sense) and multiply them all. This counting can be done together with maintaining the predecessor using a set.

Finally, computing the total number of schemes is straightforward: maintain prefix sums and compute directly.

Time complexity: $O(n \log n)$.

0.1 Linear Approach from Tester

(Replace set with a doubly linked list to maintain predecessor and successor.)

We require the four regions each form a monotonic chain. Direct partitioning is difficult. Consider the following problem:

For $1 \sim i$, find the position j with the largest p_j but $p_j < p_i$ (i.e., $j = \arg\max\{p_j \mid p_j < p_i\}$), and the position k with the smallest p_k but $p_k > p_i$ (i.e., $k = \arg\min\{p_k \mid p_k > p_i\}$). Define $f_i = f_j + 1$, $g_i = g_k + 1$. Here f and g can be understood as some kind of greedy increasing subsequence length. Then the prefix up to i is monotonic if and only if: $f_i + g_i - 1 = i$. Also, the number of partitioning schemes for the prefix can be easily derived from j and k via recurrence.

Maintaining j, k forward might require data structures and likely a log factor. Going backward is simple: only predecessor/successor queries and deletions are needed, which a doubly linked list can handle in linear time.

Given the main difficulty of this problem lies in finding a countable method, the log solutions are not intentionally constrained later.

Problem D. DAGPower

Given a DAG $G = (V, E)$. Let the set of nodes with indegree zero be $S \subseteq V$, and an initial partition $S = S_y \cup S_z$ is provided. Given the value a_i for each node, find an optimal partition S'_y, S'_z such that the sum of a_i for nodes connected to both S'_y and S'_z, as well as nodes differing from the original partition, is minimized.

This can be understood as: each partition itself has a cost, modifying the partition also incurs a corresponding cost, and the goal is to minimize the total cost.

$$1 \le N = |V| \le 5000, \quad 1 \le M = |E| \le 50,000$$

It is not difficult to think of the min-cut model: assigning a node to one side incurs a certain cost; if two nodes do not belong to the same side, an additional cost is incurred, and the goal is to minimize the total cost.

For a node i in S, assigning it to the same side as the initial partition costs 0, while assigning it to the opposite side costs a_i. The graph construction for this part is $O(N)$.

Next, consider the cost between each pair of nodes. If constructing the graph directly, one can consider changing the cost to: if i is connected to both S'_y and S'_z, then incur a cost of $2a_i$; otherwise, incur a cost of a_i. This way, we only need to compute the cost based on whether each set is connected to i, which is easier to model. However, the resulting graph would have $O(N^2)$ edges, which is unacceptable for this problem.

To control the complexity, consider directly utilizing the structure of the original graph for flow.

Split the original graph into two copies, connect S_y to one copy, and connect the other copy to S_z (the side connected to the source may require the reverse graph; assume it is S_y). Then, if a node v can connect to both $s \in S'_y$ and $t \in S'_z$, there will be a path source $\to s \to v \to t \to$ sink in the flow model. Connect the two copies of v with an edge of capacity a_v to represent the corresponding cost.

Note the details: If the modification cost is calculated directly as before, there might be insufficient flow in the middle. To resolve this, add a large constant to the edges connecting S_y and S_z directly to the source and sink, and connect the corresponding nodes of the same S in the two copies with an edge of capacity $+\infty$ (other construction methods may not require this).

The standard solution modeled as above runs in less than 0.4s, and the time limit should be quite lenient, but it is not guaranteed that other similar modeling approaches will necessarily pass.

Given several computational tasks, there may be dependencies between them. The goal is to split these tasks between two servers to minimize the total completion time.

If modeled directly, the answer is trivial (assign all tasks to the same server). If each server must have at least one task, the min-cut can also be used, but the data range needs to be reduced, which would allow a quadratic graph construction to pass.

Problem E. Edges of LCP Weight

For a string s of length n consisting only of lowercase letters a to z, consider an undirected weighted graph with n nodes. For every two distinct nodes i, j, there is an edge with weight $\mathrm{LCP}(suf_i, suf_j)$, where $suf_i = s[i:n]$. Define the value of string s as the sum of edge weights in its maximum spanning tree.

Find any string of length n (consisting only of a to z) with the k-th smallest value.

$$1 \le n, \sum n \le 4 \times 10^5, \quad 1 \le k \le \min(26^n, 10^{15}).$$

Since $26^{12} > 10^{15}$, when $n \ge 12$, the number of strings with the smallest value exceeds 10^{15}, so we only need to construct a string with the smallest value.

It's not hard to see that for a given string, sorting its suffixes and connecting adjacent suffixes yields a maximum spanning tree. Its total edge weight is the sum of the height array, denote it as x (can be proven via Kruskal's algorithm and properties of the height array).

By a classic result, x equals $\frac{n\cdot(n+1)}{2}$ minus the number of distinct substrings. Therefore, we only need to construct a string maximizing the number of distinct substrings.

Consider each length separately. The upper bound for distinct substrings is $\sum_{i=1}^{n} \min(26^i, n-i+1)$. Use the following method to construct an infinite string s such that for any n, the prefix of length n of s is the desired string:

1. Let $s = ab\ldots z$.
2. For $i = 2, 3, \ldots$, append letters to s such that every string of length i consisting of lowercase letters appears exactly once in s (De Bruijn sequence). This can be achieved by finding an Eulerian path.

In implementation, stop when s reaches length n. Time complexity: $O(26n)$.

For $n \leq 11$, since $B_{11} = 678570$, we can brute-force precompute the number and answer for all essentially different strings. For queries, use binary search. Naive brute-force preprocessing time complexity is $O(B_n \cdot n^2 \log n)$, acceptable.

Problem F. Find The XOR Sum

Generate an infinite tree by the following rules:

1. Root node is 1.
2. Every positive integer appears in the tree.
3. For node i, let $mn(i)$ be the smallest child id and $mx(i)$ the largest child id. Then $mx(i) - mn(i) = i - 1$, and for any $1 \leq i < j$, $mx(i) < mn(j)$.

Now keep only the first n nodes. Each node has a weight (initially 0). Perform q operations of two types:

1. Bitwise OR c to all node weights in the subtree of x.
2. Query the XOR sum of all node weights in the subtree of x.

$$1 \leq x \leq n \leq 10^{18}, \quad 1 \leq q \leq 10^6, \quad 1 \leq c < 2^{60}.$$

First analyze the tree's properties. Easy to prove by induction: $mn(i) = 2 + \frac{i(i-1)}{2}$, $mx(i) = 1 + \frac{i(i+1)}{2}$. From this, we can also derive $fa(i) = \lfloor \sqrt{2i-2} + 0.5 \rfloor$. Moreover, as depth increases, node ids grow quadratically. It can be found that for $n \leq 10^{18}$, the height $h \leq 9$.

Consider offline storing all nodes that appear and discretize them. Using the small tree height, we can brute-force climb up to find parent for each point, building a tree with $O(q)$ nodes.

For each node, maintain: subtree XOR sum, a modification tag, and parity of the number of positions where each bit is still 0 within the subtree. Initially, the parity depends on the parity of the subtree size.

Before each operation, brute-force traverse all ancestor nodes to push down modification tags. For a modification operation, first compute which bits are newly set compared to the tag. Using the parity of zero-bit counts, compute the new subtree XOR sum. Also propagate the change in XOR sum up to all ancestors. Then compute the new parity and propagate parity changes up to all ancestors. For a query operation, directly output the subtree XOR sum. All operations can be implemented using bit operations without bit-by-bit processing.

Preprocessing and discretization: $O(q \log q)$. Maintaining operations: $O(qh)$. Actually, if ignoring the small height property and directly using a segment tree, complexity $O(q \log q)$ is also acceptable.

If not maintaining parity of zero counts and using an $O(qh \log V)$ approach, it will likely TLE. A few teams optimized it to pass. An $O(q(h + \log V))$ approach might be borderline but should pass.

Special note: When implementing discretization or similar processes, it's better to use `.find` on maps rather than direct key access. Direct access inserts a key if it doesn't exist, greatly increasing time and space constants.

Problem G. Get Two Strings From One

Given a binary string of length n, partition it into two subsequences such that the sum of their values (when interpreted as binary numbers) is minimized.

$$\sum n \leq 5 \times 10^5.$$

Intuition: try to minimize the effective lengths (number of significant bits) of both subsequences.

Consider the following greedy: For some prefix, assign all zeroes to one subsequence and all ones to the other subsequence. Then assign the entire suffix to the subsequence containing the zeroes.

Using exchange arguments, it can be proven that this greedy achieves the minimum, and it occurs when the difference in effective lengths of the two subsequences is $O(1)$.

In fact, the split position is uniquely determined: the position where the length of the subsequence containing ones first becomes greater than or equal to the length of the suffix. Whether enumerating $O(1)$ possible positions or directly finding the unique position, time complexity is $O(n)$.

Problem H. Help For Future Navigators

There are m points. For an energy allocation scheme $a_{1\ldots n}$, its weight is $\prod_{1\le i,j\le n} A_{ij}(a_i, a_j)$, where $A_{ij}(x, y)$ depends only on the order relation between x and y ($>$, $=$, $<$).

Now there are q queries. Each query gives n, asking for the sum of weights over all schemes distributing n points of energy among the m points.

Consider a fixed sequence c. There are $\binom{n}{c_1,c_2,\ldots,c_m}$ ways to obtain it. Using Lucas's theorem, when taking modulo p, this multinomial coefficient equals the product of digit-wise multinomial coefficients when treating c_i and n in base p: $\left(\prod_{i=1}^{k}\binom{c_{1,i},c_{2,i},\ldots,c_{m,i}}\right) \bmod p$. For each digit, if there is a carry when adding c_i in base p to get n, the corresponding digit-wise multinomial coefficient becomes 0, contributing nothing. Therefore, we only need to count schemes where no carry occurs, i.e., $n_i = \sum_{j=1}^{m} c_{i,j}$ for each digit i.

First consider the simpler case $n < p$. Notice A, B only impose weight based on the order relations among the m elements. So for a fixed order relation pattern M, its weight is a fixed value $D(M)$. For a fixed M, compute the sum $W(M) = \sum n! \prod_{i=1}^{m} \frac{1}{c_i!}$ over all $(c_1, \ldots, c_m)$ consistent with M. The contribution is $D(M)W(M)$.

Observe that many $W(M)$ have the same value. When swapping any two elements in M, $W(M)$ does not change. Consider, for each M, contributing at a new graph M' where the m elements are sorted according to M. Then $W(M) = W(M')$, and M''s order relations are of the form $c_1\ op_1\ c_2, \ldots, op_{m-1}\ c_m$, where each op_d is either $=$ or $<$. There are essentially 2^{m-1} distinct M'.

For each M', compute the sum of $D(M)$ over all M that map to it.

For the op sequence corresponding to M', breaking at each $<$ yields length sequence $l_1, \ldots, l_d$. In the original M, $l_1, \ldots, l_d$ correspond to different sets $S_1, \ldots, S_w$, with $|S_x| = l_x$. Consider DP: let $dp_{S,x}$ denote, after considering sets $S = \cup_{d\le x} S_d$, the total weight sum for the sizes $|S_1|, |S_2|, \ldots, |S_x|$ corresponding to operator sequence T. Transition: enumerate the pattern of S_{x+1}. When computing $D(M)$, we need $\prod_{i=1}^{w}\prod_{j=i+1}^{w} A_{S_i,S_j} \prod_{i=1}^{w} B_{S_i}$, which can be incorporated during DP. Complexity: $\sum_{S\subseteq[m]} 2^{m-|S|} 2^{|S|-1} = 2^{2m-1} = O(4^m)$.

Now compute the answer for each state M'. Each state corresponds to a partition. For each $1 \le m' \le m$, compute the weight sum $F_{S,s}$ for a set of size m' with partition S and sum s. Transition: enumerate the new partition and the value for a new element, achieving $\sum_{i=1}^{m} 2^{i-1} p^2 = O(2^m p^2)$. Due to strict monotonicity constraints, the enumeration of the new element's value should be placed in the outermost loop.

Now consider $n \ge p$ case. After applying Lucas, the main difficulty is that order relations across different digits can influence each other.

Consider determining from the highest digit downwards. Let the current digit be i. The order relations determined by higher digits are M, corresponding to operators op.

Determine for each point what to fill in digit i from left to right, thus determining the new order relations M' after filling digit i. If we can precompute the transition count from (M, M') for each possible sum t at this digit (i.e., build a transition graph), we can perform the overall DP.

For each pair (M, M'), note that segments separated by $<$ in op are independent. The problem reduces to counting the number of ways to change a segment of all $=$ into the segment's order relations in M'. This can be precomputed. Finally, combining independent segments for (M, M') is just convolution. This

can be optimized using a search tree.

Precompute the transition graph of states. For each query, run DP on this graph, achieving $O(qG(m)\log_p n)$.

Total complexity $O(4^m + F(m)p^2 + qG(m)\log_p n)$. Here:

$$F(m) = [x^m]\prod_{i=1}^{m}\left(\frac{1}{1-x^i}\right)^{2^{i-1}},$$
$$G(m) = [x^m]\prod_{i=1}^{m}\prod_{j=0}^{i-1}\left(\frac{1}{1-2^j x^i}\right)^{\binom{i-1}{j}}.$$

We have $F(12) = 17547$, $G(12) = 806227$. A loose bound is $F(m) = O(2^m\lambda(m))$, $G(m) = O(3^m\lambda(m))$, but this bound is not tight.

If using NTT and subset convolution, can achieve $O(4^m + 2^m p\log^2 p + F(m)p\log p + qF(m)m^2\log_p n)$, but in this problem, it's not faster than naive convolution.

Problem I. Incident in the Ruins

Given an $N \times M$ incomplete 0-1 matrix s. Fill the missing bits such that rows are in non-decreasing order, and the last row's value is minimized.

$$1 \le N, M \le 200.$$

First consider minimizing the last row.

Greedy approach: maintain the minimum possible s_i. When processing a new row, find the leftmost position j where it doesn't match the minimum. From j leftwards, find the first modifiable position k, flip it (considering carry), then greedily determine bits to the right of k.

To ensure minimization for subsequent counting, directly overwrite the input s_N with the computed minimum.

Next, consider counting.

Note: For a filling scheme satisfying the order, there exists a split point k in the first bit such that:

- For all $i \le k$, the first bit of s_i is 0.
- For all $i > k$, the first bit of s_i is 1.
- The remaining bits for $[1, k]$ and $(k, N]$ are independent.

After splitting by k, the remaining bits also satisfy similar properties. Design DP: $f(i, l, r)$ = number of ways to fill bits $[i, M]$ given that rows l to r have identical first $i-1$ bits. Transition: cases where all rows have 0 at bit i, all have 1, or there exists a split point. Need to check validity for each case (whether the corresponding bits can all be set to 0 or 1).

If validity checking is implemented efficiently, complexity is $O(N^3M)$. Due to small DP constant, it can pass.

Note: For $f(i, \cdot, \cdot)$, when enumerating split point k, there exists a minimal $l_{\min}$ such that for $l_{\min} \le j \le k$, s_j can have bit i as 0. Similarly, there exists a maximal $r_{\max}$ such that for $k < j \le r_{\max}$, bit i can be 1. So we can first enumerate k, then l, r.

In practice, the standard solution enumerating k first runs in about 0.3s, while the normal order takes 0.9s.

Problem J. Just Check The Possibility

Answer T queries. Each query gives two positive integers n, m. Ask whether there exists a positive integer sequence $a_1, a_2, \ldots, a_n$ such that $\exists 1 \le j \le n, a_j = m$, and there exists a non-negative integer t satisfying

$$\prod_{i=1}^{n}(a_i + a_{i+1}) = 2^{2t+1},$$

where $a_{n+1} = a_1$. If constructible, output `YES`, else `NO`.

$$1 \le T \le 10^6, \quad 1 \le n \le 2 \times 10^6, \quad 1 \le m \le 2^{62} - 1.$$

Lemma 1: When n is even, there does NOT exist $a_1, a_2, \ldots, a_n$ such that $\prod_{i=1}^{n}(a_i + a_{i+1}) = 2^{2t+1} (t \in \mathbb{N})$.

Proof: By induction on n. $n = 2$ is obvious. Assume holds for $n = k - 2$. For $n = k$:

Proof by contradiction. Suppose such a sequence of length k exists: $\prod_{i=1}^{k}(a_i + a_{i+1}) = 2^{2t+1}$.

Then clearly each $a_i + a_{i+1}$ must be a positive power of 2. Let $a_i + a_{i+1} = 2^{b_i}$.

Lemma 2: If all a_i are required to be odd, then when n is odd, there does NOT exist $a_1, a_2, \ldots, a_n$ such that $\prod_{i=1}^{n}(a_i + a_{i+1}) = 2^{2t} (t \in \mathbb{N})$.

Proof similar to Lemma 1, omitted.

Lemma 3: If all a_i are required to be odd, then when n is odd, if there exists $a_1, a_2, \ldots, a_n$ such that $\prod_{i=1}^{n}(a_i + a_{i+1}) = 2^{2t+1} (t \in \mathbb{N})$, then $\min_{i=1}^{n}\{a_i\} = 1$.

Proof: Assume $a_1 = \min_{i=1}^{n}\{a_i\}$. If $a_1 \neq 1$, we use induction on n to show impossibility. The induction hypothesis is clear.

Proof by contradiction. Suppose $a_1, a_2, \ldots, a_k$ exists with $\prod_{i=1}^{k}(a_i + a_{i+1}) = 2^{2t+1} (t \in \mathbb{N})$. First, under modulo k indexing, there is no index j such that $a_j = a_{j+2}$. Otherwise, removing a_j, a_{j+1} yields a sequence of length $k - 2$ satisfying the condition, contradicting the induction hypothesis!

Let $a_i + a_{i+1} = 2^{b_i}$. Then $a_1 + a_2 = 2^{b_1}$. Since $a_2 \ge a_1$, we have $2^{b_1 - 1} \le a_2 < 2^{b_1}$.

Consider a_3. Since $a_2 + a_3 = 2^{b_2}$ and $2^{b_1 - 1} \le a_2 < 2^{b_1}$, we have $b_2 \ge b_1$. But if $b_2 = b_1$, then $a_1 + a_2 = a_2 + a_3$, implying $a_1 = a_3$, contradicting the previous small conclusion. So $b_2 \ge b_1 + 1$. Then $2^{b_1} < a_3 < 2^{b_2}$, so $a_2 < a_3$, and $2^{b_2 - 1} < a_3 < 2^{b_2}$.

Continuing, we deduce: a_i are strictly increasing. Then $a_1 < a_2 < \cdots < a_k < a_1$, contradiction! So such sequence does not exist for $n = k$.

For each positive odd m, construct a sequence $d_{m,1}, d_{m,2}, \ldots, d_{m,l_m}$ such that $d_{m,1} = 1$, $d_{m,l_m} = m$, and $\forall i \in [1, l_m - 1]$, $\exists t \in \mathbb{N}$, $d_{m,i} + d_{m,i+1} = 2^t$. Construction rule:

For $m = 1$, set $l_1 = 1$, $d_{1,1} = 1$.

For $m \neq 1$, let $2^p < m < 2^{p+1} (p \in \mathbb{N})$. Set $l_m = l_{2^{p+1} - m} + 1$, $d_{m,i} = d_{2^{p+1} - m, i}$ for $i = 1, 2, \ldots, l_{2^{p+1} - m}$, and $d_{m,l_m} = m$. By the second principle of induction, such a sequence meets the requirement.

Define $\{a_i\} = \{d_{m,1}, d_{m,2}, \ldots, d_{m,l_m - 1}, d_{m,l_m}, d_{m,l_m - 1}, \ldots, d_{m,2}, d_{m,1}\}$. From the definition of d_m sequence, we have $\prod_{i=1}^{2l_m - 1}(a_i + a_{i+1}) = 2^{2t+1} (t \in \mathbb{N})$.

We claim: This $\{a_i\}$ sequence is the shortest sequence containing m that satisfies the condition.

By Lemma 3, the sequence must contain 1. Assume $a_1 = 1$.

We only need to show that d_m is the shortest sequence starting with 1, ending with m, and having adjacent sums as powers of 2.

Consider the second-to-last number c. Let $2^p < m < 2^{p+1}$, $m + c = 2^q$. Then $q \ge p + 1$. If $q = p + 1$, then by induction on $2^{p+1} - m$, the proposition holds. If $q \ge p + 2$, then $c > m$. Analogous to the proof of strict monotonic increase in Lemma 3, the sequence would be strictly decreasing, contradiction!

The final solution process:

Write $m = 2^f \cdot g$, where g is odd.

If n is even, by Lemma 1, output NO.

Otherwise, n is odd. If a construction exists, then $a_1, a_2, \dots a_n$ must all have the same exponent of 2 in their prime factorization, namely f. If f is odd, let $a'_i = a_i/2^f$. Then $\prod_{i=1}^{n}(a'_i + a'_{i+1}) = 2^{2t'}(t' \in \mathbb{N})$. By Lemma 2, output NO.

If f is even, recursively construct d_g for g. If $n < 2l_g - 1$, construction impossible. If $n \geq 2l_g - 1$, set the first $2l_g - 1$ numbers as $d_{g,i} \times 2^f$, and fill the remaining numbers with 2^f. This construction works.

Overall time complexity: $O(T \log m)$.

Problem K. King of Monkeys and Two Sets

Given n, m, L, a, b and sets A, B of sizes n, m respectively, satisfying $0, L \in A, B$. Find the smallest set C such that:

- $0, L \in C$.
- For $A \cap C$ sorted increasingly, the difference between consecutive elements $\leq a$.
- For $B \cap C$ sorted increasingly, the difference between consecutive elements $\leq b$.

$$2 \leq n, m \leq 10^6, \quad 1 \leq L \leq 10^{18}.$$

Consider DP over elements in $A \cap B$. Let $A \cap B = \{0 = x_0 < x_1 < \ldots < x_k = L\}$. Define f_k as the answer for subproblem with $L \leftarrow x_k$, $A \leftarrow A \cap [0, x_k]$, $B \leftarrow B \cap [0, x_k]$. Then $f_0 = 1$,

$$f_i = \min_{0 \leq j < i} \{f_j + d_A(x_j, x_i) + d_B(x_j, x_i) - 1\},$$

where $d_A(x, y)$ is the size minus 1 of a subset of A containing x, y with consecutive differences $\leq a$; similarly for d_B.

Characterize $d_A(x, y)$. Define $\mathrm{nxt}_A(x)$ as the largest element in A not exceeding $x + a$. Connect each x in A (except L) to $\mathrm{nxt}_A(x)$, forming a tree T_A rooted at L. Note that

$$\varepsilon_A(x, y) := d_A(x, y) - \mathrm{dep}_A(x) + \mathrm{dep}_A(y) \in \{0, 1\}.$$

Then let $g_i = f_i + \mathrm{dep}_A(x_i) + \mathrm{dep}_B(x_i)$. We have

$$g_i = \min_{0 \leq j < i} \{g_j + \varepsilon_A(x_j, x_i) + \varepsilon_B(x_j, x_i) - 1\}.$$

Consider the DFS order id_A of T_A when traversing children subtrees from small to large starting from the root. Note $\epsilon_A(x, y) = [\mathrm{id}_A(x) < \mathrm{id}_A(y)]$. After computing g_i, we can place it at point $(\mathrm{id}_A(x_i), \mathrm{id}_B(x_i))$ and use a 2D data structure / CDQ divide-and-conquer to maintain, achieving $O(n \log^2 n)$. Also note the optimal g_i is at most the current minimum g value $+1$. So we can maintain two data structures for these two possible values. Queries are of the form: "Is there a point in region $x > x_0, y > y_0$?" Use segment tree to maintain interval max. Complexity: $O(n \log n)$.

Problem L. Luxuries and Aesthetics

To simplify the following proof, for sufficiently small $\epsilon > 0$, replace a_i with $a_i + i\epsilon$. This makes all elements of sequence a distinct, avoiding messy discussions. When ϵ is small enough, the effect on the answer is a polynomial in ϵ, so this replacement is valid. Also, initially sort a in ascending order; this does not affect the final result. In the following, assume $l < r$; the case $l = r$ is simple.

Conclusion 1: There exists an optimal solution (achieving maximum) where each group forms a contiguous interval.

Proof: Let b_x denote the group assigned to index x. If there exist three indices $x < y < z$ such that $b_x = b_z \neq b_y$, let group b_x have length l_1 and sum s_1, group b_y have length l_2 and sum s_2. Let $s = s_1 + s_2$. The function $f(x) = \frac{x^2}{l_1^2} + \frac{(s-x)^2}{l_2^2}$ is a convex function. If $f(s_1)$ is on the left side of the valley, move y from b_y to b_z and z from b_z to b_y, decreasing s_1, thus increasing $f(s_1)$. If $f(s_1)$ is on the right side, move x from b_x to b_y and y from b_y to b_x, increasing s_1, thus increasing $f(s_1)$. Therefore, there exists an optimal solution without such $x < y < z$ with $b_x = b_z \neq b_y$. If a group does not form a contiguous interval, let its indices in increasing order be $c_1, c_2, \ldots, c_l$. Then there exists $c_d + 1 < c_{d+1}$. Take $x = c_d, y = c_d + 1, z = c_{d+1}$, obtaining such a triple, contradiction. So each group forms a contiguous interval.

Conclusion 2: There exists an optimal solution where groups form contiguous intervals, and when the interval length bounds are $[l, r]$, the group length sequence $l_1, l_2, \ldots$ has the form $r^A x l^B$ (i.e., A groups of length r, then one group of length x, then B groups of length l).

Proof: For adjacent group lengths (l_x, l_{x+1}), let the sum for group x be s_x and for group $x+1$ be s_{x+1}. Their contribution is $\frac{s_x^2}{l_x^2} + \frac{s_{x+1}^2}{l_{x+1}^2}$. Consider moving the first element a_d from group $x+1$ to group x. Since $a_d > \frac{s_x}{l_x}$ and $a_d \leq \frac{s_{x+1}}{l_{x+1}}$, the left term increases and the right term does not decrease. So if there exists (l_x, l_{x+1}) with $l_x + 1 \leq r$ and $l_{x+1} - 1 \geq l$, changing it to $(l_x + 1, l_{x+1} - 1)$ increases the answer. Thus, an optimal solution must satisfy for any adjacent pair (l_x, l_{x+1}), either $l_x = r$ or $l_{x+1} = l$. It's easy to see no x can satisfy both $l_{x+1} = l$ and $l_{x+1} = r$. So those x with $l_x = r$ form a prefix, and those with $l_{x+1} = l$ form a suffix, resulting in the $r^A x l^B$ form.

Conclusion 3: There exists an optimal solution satisfying Conclusion 2, and it is the one with the smallest A among those satisfying Conclusion 2. If multiple have the same minimal A, choose the one with the largest B.

Proof: First show smaller A is better. Compare $r^{A_1} x l^{B_1}$ and $r^{A_2} y l^{B_2}$ with $A_1 < A_2$. Cancel common parts, compare $x l^{B_1 - B_2}$ and $r^{A_2 - A_1} y$. If $A_2 - A_1 > B_1 - B_2$, there exists a matching $x \to r$, $(l \to r)^{B_1 - B_2}$ with $x \leq r$, $l < r$, so sums cannot be equal. If $A_2 - A_1 = B_1 - B_2$, there exists matching $x \to r$, $(l \to r)^{B_1 - B_2}$, $l \to y$, similarly sums cannot be equal. So $A_2 - A_1 < B_1 - B_2$. Then we can give a matching $(l \to r)^{A_2 - A_1}$, $l \to y$, where $l < r$, $l \leq y$, and each part on the left is larger than the corresponding on the right, and there are extra segments on the left, so left is larger.

Next, show when A fixed, larger B is better. Compare $r^A x l^{B_1}$ and $r^A y l^{B_2}$ with $B_1 > B_2$. Cancel common parts, compare $x l^{B_1 - B_2}$ and y. The leftmost l in $x l^{B_1 - B_2}$ has larger weight than y, so $x l^{B_1 - B_2}$ is larger than y.

Based on these conclusions, preprocess prefix sums in $O(n \log n)$ using harmonic series. For queries, compute in $O(1)$ the minimal achievable A, and the maximal achievable B given minimal A. Overall complexity $O(n \log n + q)$.

According to the above conclusions, preprocess prefix sums in $O(n \log n)$ using harmonic series. For queries, compute in $O(1)$ the minimal achievable A, and the maximal achievable B given minimal A, achieving $O(n \log n + q)$ complexity.