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. Build The String

The program reads the two given strings into variables `a` and `b`. The function

```
string search(string a, string b)
```

finds the desired string `r` in the case when the substrings corresponding to `a` and `b` are located in `r` such that `a` precedes `b`. First, in the strings `a` and `b`, uppercase letters are replaced by their corresponding lowercase letters, and then by exhaustive search it is sequentially checked whether merging all prefixes of `a` with `b` yields a string that entirely contains `a`:

```
for(int i = 0; i < a.size(); i++) {
    string s = a.substr(0, i) + b;
    if(s.find(a) != -1)
        if(s.size() < r.size() or (s.size() == r.size() and s < r))
            r = s;
}
```

Thus, the string `r` becomes the shortest string that is also lexicographically smallest. Then the uppercase letters are restored in the required places.

The function `run()` calls the `search` function twice:

```
string r1 = search(a, b);
string r2 = search(b, a);
string r;
if(r1.size() == r2.size())
    r = min(r1, r2);
else if(r1.size() < r2.size())
    r = r1;
else
    r = r2;
```

```
cout << r << endl;
```

From the two results `r1` and `r2`, the lexicographically smaller one is chosen.

Problem C. Counting Robot's Paths

To count the valid paths, we can subtract the number of invalid paths from the total number of unrestricted paths.

0.1 Unrestricted Paths

Without any boundary constraint, a path from $(0, 0)$ to (n, n) consists of exactly n moves to the right (X) and n moves upwards (Y), for a total of $2n$ moves. The number of such paths is given by the binomial coefficient:

$$\text{Total Paths} = \binom{2n}{n}$$

0.2 Invalid Paths and the Reflection Principle

Let $d = a + 1$. The boundary line we must not touch is $y = x + d$.

If a path touches this boundary line, we locate the **first** point $P(x_0, y_0)$ where it touches $y = x + d$. From this point onward to the destination (n, n), we reflect the remaining segment of the path across the line $y = x + d$.

Under reflection across the line $y = x + d$:

- Any point (x, y) is mapped to $(y - d, x + d)$.
- The destination point (n, n) is mapped to the reflected destination:

$$(n - d, n + d) = (n - a - 1, n + a + 1)$$

Since the reflected path starts at $(0, 0)$ and ends at $(n - a - 1, n + a + 1)$, it consists of:

- Total steps: $(n - a - 1) + (n + a + 1) = 2n$ steps.
- Number of right (X) steps: $n - a - 1$.

There is a one-to-one correspondence (bijection) between any invalid path from $(0, 0)$ to (n, n) and any path from $(0, 0)$ to the reflected destination $(n - a - 1, n + a + 1)$. Therefore, the number of invalid paths is:

$$\text{Invalid Paths} = \binom{2n}{n - a - 1}$$

Note that if $n - a - 1 < 0$, it is impossible to touch the boundary, so the number of invalid paths is 0.

0.3 Final Formula

The number of valid paths modulo $10^9 + 7$ is:

$$\text{Ans} = \binom{2n}{n} - \binom{2n}{n - a - 1} \pmod{10^9 + 7}$$

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. Experimental Astrophysics

The precision of `long double` may be insufficient: direct cross-multiplication requires $6 \times (2 \times 10^9)^2 > 2^{64}$.

Let us transform the expression to sorting by $\frac{c_i}{a_i+b_i+c_i}$. Then cross-multiplication requires only $3 \times (2 \times 10^9)^2 < 2^{64}$.

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 $1 \le i < j$, $mx(i) < mn(j)$.

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.

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

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 $n \le 10^{18}$, the height $h \le 9$.

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.

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

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. Hit The Target!

If the thickness of the targets increases monotonically, a queue q can be used to store the thickness values of the targets and a variable s for the sum of the values in the queue. Targets are added to the queue sequentially, starting from the first. As t increases, the values in the queue increase monotonically. If after

removing the first element from the queue the sum s remains at least p, then that element is removed. This process is repeated as long as possible.

For each i, we maintain the minimum size of the queue. If $S < P$, then $f(i) = -1$, otherwise $f(i) = |Q|$ (where $|Q|$ is the number of elements in the queue). Time complexity: $O(N)$.

In the general case (if the thickness of the targets is distributed arbitrarily), we use a heap instead of a regular queue to work with arbitrary input order. Time complexity: $O(N \log N)$.

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 $O(N^3 M)$. 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 $a_2 \geq a_1$, we have $2^{b_1 - 1} \leq a_2 < 2^{b_1}$.

Consider a_3. Since $a_2 + a_3 = 2^{b_2}$ and $2^{b_1 - 1} \leq a_2 < 2^{b_1}$, we have $b_2 \geq b_1$. 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 $b_2 \geq b_1 + 1$. 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 $q \geq p + 1$. If $q = p + 1$, then by induction on $2^{p+1} - m$, the proposition holds. If $q \geq p + 2$, 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 $a_1, a_2, \ldots a_n$ 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 and Sequence

Note that a sequence is a subsequence of an arithmetic progression if and only if it is either strictly monotonically increasing, strictly monotonically decreasing, or all of its elements are equal.

Indeed, for any two elements of an arithmetic progression, the property $a_i - a_j = (i - j) \cdot d$ holds. Since the operation of taking a subsequence preserves the order of elements, for any $i > j$, $x_i - x_j$ must be equal to d multiplied by a positive integer. For $d > 0$ we obtain an increasing sequence, for $d < 0$ a decreasing one, and for $d = 0$ a constant one.

Suppose we have a monotonic sequence x_i. If it is decreasing, it is at least a subsequence of an arithmetic progression with $a_0 = x_0$ and $d = -1$ (this progression contains all integers less than x_0 in decreasing order). Moreover, in this case -1 is the answer (since even if there are other negative values of the common difference d, they are less than -1, and we are required to output the maximum value of d). If all of its elements are equal, it is a subsequence of the progression with $a_0 = x_0$ and $d = 0$ (an infinite sequence consisting of the same elements). If the sequence is increasing, it is at least a subsequence of an arithmetic progression with $a_0 = x_0$ and $d = 1$.

It remains only to note that the difference between any two elements of an arithmetic progression must be a multiple of d. Thus, in the case of an increasing sequence, the maximum value of d is equal to the greatest common divisor of all $n - 1$ differences of adjacent elements of the sequence x:

$$d = \gcd(x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1})$$

If the sequence is neither strictly monotonically decreasing nor strictly monotonically increasing, and contains at least two distinct elements, then the answer is ?.

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 $r^{A_1}xl^{B_1}$ and $r^{A_2}yl^{B_2}$ with $A_1 < A_2$. Cancel common parts, compare $xl^{B_1-B_2}$ 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 $x \le r$, $l < r$, 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 $l < r$, $l \le y$, 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 $r^A xl^{B_1}$ and $r^A yl^{B_2}$ with $B_1 > B_2$. Cancel common parts, compare $xl^{B_1-B_2}$ and y. The leftmost l in $xl^{B_1-B_2}$ has larger weight than y, so $xl^{B_1-B_2}$ 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.