Problem 1A. Bored of Long Statements?

Author: **Feysa Bohdan**
Task preparation: **Feysa Bohdan**
Editorial: **Feysa Bohdan**

Block 1: $a_1 = a_2 = \cdots = a_n$;

If $a_1 = a_2 = \cdots = a_n$, for every permutation of a $f(a)$ is equal to 0. Thus, the answer is always 0 in this block.

Block 2: $n \leq 9$;

For $n \leq 9$ we can iterate through every permutation of a and manually calculate $f(a)$ with $O(n)$ time complexity. Final time complexity is $O(n! \cdot n)$.

Block 3: $n \leq 500$;

For $n \leq 500$ we can see that the for every fixed triplet i, j, k $(i < j < k)$ it is optimal for those elements in a to be sorted by ascending order $(a_i \leq a_j \leq a_k)$ or to be sorted by descending order $(a_i \geq a_j \geq a_k)$.

Proof:

Suppose that we want to minimize

$$(x - y)^2 + (y - z)^2$$

$$(x - y)^2 + (y - z)^2 = x^2 + y^2 - 2xy + y^2 + z^2 - 2yz$$

We need to maximize $2xy + 2yz$.

For any given x, y that $x + y = A$, the maximal value of xy is equal to $\frac{A^2}{4}$

Proof: For any real q:

$$\frac{A^2}{4} \geq (\frac{A}{2} - q)(\frac{A}{2} + q) = \frac{A^2}{4} - q^2$$

In other words, $x - y$ should be minimal.

So, the to maximize $2xy + 2yz$ the condition $(x \leq y \leq z)$ should be true.

There are at most ten different a_i in the array;

For this block, we have to combine the ideas of previous blocks. Total time complexity for a test in which there are k different values in a is $O(k! \cdot k)$.

Block 4: $n \leq 10\,000$;

We should take the fact from block 3 and rephrase it:

There has to be no local maximum or minimum elements.

We should start from the second element and make following actions for every j $(2 \leq j \leq n)$:

- if $a_{j-1} > a_j$ we swap elements on positions j and $j - 1$.

We repeat this algorithm until there are no local maximum and minimum elements. It can be proven that the worst time complexity of this algorithm is $O(n^2)$.

Block 5: no additional restrictions.

The algorithm from block 5 results in a sorted array, so by using an efficient sorting algorithm we can achieve time complexity of $O(n \cdot log(n))$.

Problem 1B. Ridiculous Queries

Author: **Tymkovych Oleksandr**
Task preparation: **Tymkovych Oleksandr, Tsitsei Pavlo**
Editorial: **Tymkovych Oleksandr**

Block 1: $p = [1, 2, \ldots, n]$.

In this block we may notice, that $p_i^k = p_i$, hence the answer for each query is $\max\limits_{1 \le i \le n}(p_i + i)$.

Block 2: $q \le 10$ and $1 \le k \le 100$.

In this block the intended idea was to find p_i^k in $O(k)$ for specific i. The total complexity will be $O(q \cdot n \cdot k)$.

Block 3: $1 \le k \le 100$.

In this block, we may precalculate p_i^k for each $1 \le i \le n$ in $O(n \cdot k)$. Then, precalculate the answer for each k and answer the queries in $O(1)$.

Block 4: for each pair $1 \le i, j \le n$ there exist k that $p_i^k = j$.

In this block the idea was to look at the permutation as a graph. The permutation forms a single cycle of length n. Hence, we may replace k with $k \bmod n$ and do the same, as in block 3.

Block 5: $n \le 20$.

Now, we may process each cycle independently. In each cycle we may replace k with $k \bmod \ell$ where ℓ is the length of the current cycle. Then, the idea is the same as in block 2. The total complexity is $O(q \cdot n^2)$.

Block 6: $q \le 10^4$.

For each cycle in the permutation, we may precalculate the maximum for each k in $O(n^2)$. Then, answer each query in $O(n)$.

Block 7: no additional restrictions.

At first, we do the same precalculation as in the block 6. Also, we may group the cycles with the same length, taking the maximum value for each $k \bmod \ell$.

What is the upper bound for the number of distinct cycle lengths in a permutation length n? The length of the permutation is equal to the sum of cycle lengths. Let us consider the sum of distinct cycle lengths. Suppose these lengths are $1, 2, \ldots, x$. Then, the sum is $1 + 2 + \cdots + x = \frac{x(x+1)}{2}$. For $x \approx \sqrt{n}$ the sum is already greater than n. This proves, that there are at most around $\sqrt{n}$ distinct cycle lengths.

Hence, we may answer each query in $O(\sqrt{n})$. The total complexity is $O(n^2 + q\sqrt{n})$.

Problem 1C. And Again, Queries

Author: **Tymkovych Oleksandr**
Task preparation: **Tymkovych Oleksandr, Feysa Bohdan**
Editorial: **Tymkovych Oleksandr**

Block 1: $q = 1$, $l = 1$ and $r = n$.

In this block we have to find the answer for the whole array. Greedy idea works! We may try to take as many elements as we can to the certain segment.

Block 2: $k = 1$.

The constraints in this block mean that there is exactly one distinct element in each segment. We may answer the queries using a segment tree on the given array. For each node, let us store the answer for the segment, the first element in the segment, and the last element in the segment. For merging two nodes, you may compare the last element of the left segment and the first element of the right segment.

Block 3: $l = 1$ for all queries.

We may use the idea from block 1 and store the answer for each prefix. Then, answer the queries in $O(1)$.

Block 4: $a_i \leq 10^5$, $1 \leq n \leq 10^5$.

Let us calculate for each index $1 \leq i \leq n$ the largest j ($i \leq j \leq n$) such that there are at most k distinct elements between $a_i, a_{i+1}, \ldots, a_j$. Let us call that j as x_i. It can be done $O(n \log n \log A)$ using Fenwick tree with binary search.

Now, the answer for the query is the number of jumps of type $i \to x_i$ starting from l and finishing in r. Here, we may use binary jumps technique. $up_{i,k}$ is the position we will find ourselves after 2^k jumps starting from i. It can be precalculated in $O(n \log n)$. Also, it is easy to answer the queries using this precalculation in $O(\log n)$ for each query.

The total complexity is $O(n \log n \log A + q \log n)$.

Block 5: $a_i \leq 10^5$, $1 \leq n \leq 10^5$.

In this block, the intended idea is the same as in block 4, but the precalculation of x_i can be done in $O(n \log A)$ using two pointers or segment tree.

Block 6: $a_i \leq 10^5$, $1 \leq n \leq 10^5$.

The idea is the same as in block 5. You may compress the integers of the array to get the precalculation in $O(n \log n)$.

Problem 1D. Simple task?

Author: **Tsitsei Pavlo**
Task preparation: **Tsitsei Pavlo**
Editorial: **Tsitsei Pavlo**

Notions:

Let's denote $[a, b, c]$ as number of substrings of odd length with $s_l = a$, $s_m = b$, $s_r = c$. So answer is $[0, 0, 0]+[1, 1, 1]$. Let's also denote $[all]$ as all substrings of odd length.

Block 1: There are no zeros in the string:

Here, all substrings of odd length are good substring. So we can iterate through the string and easily count the number of such substrings. Solution works in $O(n)$.

Block 2: $n \leq 100$:

We iterate through all possible triples of l, r, k and check if $k = \frac{l+r}{2}$, its length is odd, $l < r$ and $s_l = s_r = s_k$. Solution works in $O(n^3)$.

Block 3: $n \leq 10^3$:

Let's extend the solution for the second block. If we fix only l, r and $k = \frac{l+r}{2}$. Then we just check if substring has odd length, $l < r$ and $s_l = s_r = s_k$. Solution works in $O(n^2)$.

Block 4: The string contains at most 10^3 ones:

Answer is $[0, 0, 0] + [1, 1, 1] = [all] - [1, 0, 0] - [0, 1, 0] - [0, 0, 1] - [1, 0, 1] - [1, 1, 0] - [0, 1, 1]$. We can easily find $[all]$ (first block). Let's fix only one "1" and find the number of substrings with this. It will be $[1, 0, 0] + [0, 1, 0] + [0, 0, 1] + 2 \cdot [1, 0, 1] + 2 \cdot [1, 1, 0] + 2 \cdot [0, 1, 1] + 3 \cdot [1, 1, 1]$. We can find $[1, 0, 1]$, $[1, 1, 0]$, $[0, 1, 1]$, $[1, 1, 1]$ in $O(n^2)$ just by fixing two "1". So we can reduce equation to $[1, 0, 0] + [0, 1, 0] + [0, 0, 1] + [1, 0, 1] + [1, 1, 0] + [0, 1, 1]$ and subtract from $[all]$ and get answer. Total complexity $O(n^2)$.

Block 5: Without additional constraints:

We can fix some two bound and find number of substrings with it in $O(n)$. But we cannot specify a third element. So if we write down some possible options, we can get such variables: $[1, 0, 1] + [1, 1, 1]$ (fix left and right bounds to "1") $[0, 1, 1] + [1, 1, 1]$ (fix middle and right bounds to "1") $[1, 1, 0] + [1, 1, 1]$ (fix left and middle bounds to "1") $[0, 0, 0] + [1, 0, 0]$ (fix middle and right bounds to "0") $[0, 0, 0] + [0, 1, 0]$ (fix left and right bounds to "0") $[0, 0, 0] + [0, 0, 1]$ (fix left and middle bounds to "0"). It is easy to see that the sum of all is $[all] + 2 * ([0, 0, 0] + [1, 1, 1])$. So we just subtract $[all]$, divide by 2 and here is the answer. Solution works in $O(n)$.

Problem 2A. A Little Party Never Hurt Anybody

Author: **Tymkovych Oleksandr**
Task preparation: **Tymkovych Oleksandr**
Editorial: **Tymkovych Oleksandr**

Formally, the statement said that the party is bad when there exist an element on the segment that is a submask of any other element on the segment. To check, if the number x is a submask of y we may check if x OR $y = y$ where OR is a bitwise operation.

Block 1: $n \leq 40$.

In this block, we may try to check every segment. In each segment, we try to find a bad person. The complexity is $O(n^4)$.

Block 2: $n \leq 100$.

Notice that the bad person is always with a minimal value, since it must be a submask of all other elements. Hence, we may track a possible bad person. The complexity is $O(n^3)$.

Block 3: $a_i \in \{0, 1\}$.

Notice that here, in each segment, there exists a bad person.

Block 3: no additional restrictions.

We know that a bad person possibly is a minimum on the segment. Also, we know that it must be a submask of all other elements on the segment. The observation is that the minimum must be equal to the bitwise AND of all elements on the segment.

So we may track the minimum and the bitwise AND for each segment. The complexity is $O(n^2)$.

Problem 2B. A Little Cheating Never Hurt Anybody

Author: **Tsitsei Pavlo**
Task preparation: **Tsitsei Pavlo**
Editorial: **Tsitsei Pavlo**

Block 1: $k = 0$:

Here we cannot perform any number of operation, so we need to find the maximum subarray. We can find it using prefix sum. We just perform prefix sum, so we reduce the problem to finding i, j such that $j < i$ and $a_i - a_j$ is maximum. If we fix i then j is an element such that a_j is minimum on prefix from 0 to i. It can be done using set in STL or regular heap. Solution work in $O(n \cdot log(n))$.

Block 2: $k = 1$:

Here, we can perform only one operation. So we find the left most maximum subarray and perform operation on the right most element. It is the best element, because it considers all elements that intersect with this maximum subarray, so if all subarray intersects, than answer will be reduced by 1. Solution works in $O(n \cdot log(n))$ because uses two times algorithm from the first block.

Block 3: $p_i \geq 0$:

Here, the maximum subarray is all array, so we just perform all operations on positive integers while they exist. If there is more, then we see that maximum subarray is some element, because $p_i \leq 0$. So we just need to minimize the maximum element, which we can do by reducing each element by 1. So answer will be $\frac{n}{k}$ where k is remaining number of operations. Solution works in $O(n)$.

Block 4: $n \leq 1\,000$:

Here we will assume that we don't know how to find maximum subarray in $O(n \cdot log(n))$, but in $O(n^2)$ just by iterating through all subarrays instead. We can use binary search for the answer. If we can get maximum subarray sum less or equal x, then we obviously can get it less or equal y when $x < y$. Also, if we cannot get the maximum subarray sum less or equal to x, we cannot get it less or equal than y for $y < x$. So this function is monotonic, and we need to find such x that we can make maximum subarray less or equal to x, but we cannot do it for $x - 1$. Let's fix some m and try to make all subarrays sum less or equal to m. To make it we iterate through i from left to right and find the maximum subarray for his element, and if it is bigger than m, then we reduce element i to such value, that this segment will have the sum less or equal than m. We count the number of operation that we should make and if it is bigger than k than obviously answer is bigger, and less otherwise. This algorithm work with the same proof as it was in second block. Solution works in $O(n^2 \cdot log(n))$.

Block 5: $n \leq 10^5$:

If we use solution from block 4 and use the algorithm for finding maximum subarray from the first block, we would get solution that work in $O(n \cdot log^2(n))$.

Block 6: without additional constraints:

Here we need to optimize solution. Let's optimize finding maximum subarray. As we said, we need to find minimum element on prefix $[1, i]$ for all i. It can be done in linear time using the same algorithm as prefix sum but minimum instead. This algorithm will work in $O(n)$, so solution works in $O(n \cdot log(n))$

Problem 2C. A Little Bit of Cryptography Never Hurt Anybody

Автор задачі: **Feysa Bohdan**
Задачу підготував: **Feysa Bohdan**
Розбір написав: **Feysa Bohdan**

- Observation 1.

  The total number of pyramidal permutations of length n is equal to 2^{n-1}.

  Proof:

  Suppose that you start with a sequence of n zeros. In each step i you will be adding number i to some position. There are always only 2 possible positions for number i when every smaller one is already positioned somewhere in the sequence.

  Proof:

  If after placing i in the sequence, both its adjacent elements are still zeros, the position of i becomes a local minimum. (And by definition it is not a pyramidal permutation anymore, because there are at least 2 local maximums. (by rephrasing the definition, there has to be at most one maximum.))

  Only the last element n has only one possible placement.

  So there are $\underbrace{2 \cdot 2 \cdot \cdots \cdot 2 \cdot 2}_{\text{n-1 times}} \cdot 1 = 2^{n-1}$ possible permutations.

- Observation 2:

  Each of 2^{n-1} permutations may be represented via bitmask of length $n-1$.

  Proof:

  From observation 1 we know that for each element that is less than n there are exactly 2 plausible positions. After creation of the permutation, we do the following:

  Set all bits in the bitmask to 1.

  - for every number j that is placed in a position smaller than the placement of number n, we set the j bit from the front to be equal to 0.

  As an example:

  $[1, 3, 5, 4, 2] \to [0, 1, 0, 1]$;

  $[1, 5, 4, 3, 2] \to [1, 0, 0, 0]$;

- Observation 3:

  for each permutation, its bitmask is also equal to the number of pyramidal permutations that are strictly smaller than it.

Block 1: $n \leq 10$:

In this block, we could check every permutation and find whether is pyramidal or not. After that, we could find the number of those that are smaller or equal to p manually. Time complexity $O(n! \cdot n)$

Block 2: $n \leq 30$:

We can use binary search to find the lexicographically biggest permutation smaller or equal to the given one. We will be using the observations $1, 2, 3$ to do that.

Block 3: $n \leq 60$:

We will be using the idea of block 2, but with slightly bigger constraints and output by modulo.

Block 4: $n \leq 500$:

We will manually construct the permutation that is smaller or equal to p, after that we will convert it in the binary string and finally into the desired answer. Permutation is constructed as follows:

For some prefix of length k the biggest pyramidal permutation lexicographically smaller or equal to p than will be equal to prefix of p. We will try to construct a permutation for every k ($0 \leq k \leq n$). If the first k elements are equal, the $k+1$-th element has to be smaller than p_{k+1}. We can store the numbers that were not used in the set. After that starting from the $k+2$-th element we will take the biggest that was not used and use it up. If the created permutation is not bigger than p, we will save it for later. There are several conditions when permutation with first k elements equal to the ones used in p is impossible to build:

- there is at least one index $j \leq k$ that $p_j < p_{j-1}$.
- $k+1 < n$ and when we are at position $k+1$ there is no number that is smaller than p_{k+1}.
- the permutation constructed for fixed k is lexicographically bigger than p.

After every possible permutation is built, we will choose lexicographically the biggest one. We will use observation 2 and transform it into a bitmask. After that, using observation 3 we will calculate the answer and output it via desired modulo.

Final time complexity: $O(n^2 \cdot log(n))$ or $O(n^2)$ depending on implementation.

Total memory used: $O(n^2)$ for storing optimal permutations for every k.

Block 5: $n \leq 10^4$:

We will be using the idea of previous block. If we take the $O(n^2)$ solution from the previous block, we will only need to optimize memory. For memory optimization, we will not be storing every permutation, only the biggest one. When the new possible permutation is made, we check whether it is bigger than the current maximum, if it is, we delete the previous one and store the new one.

Block 6: $n \leq 10^5$:

The final observation for optimization is that the possible permutation created for fixed k will **always** be bigger than the one constructed for $k-1$. (if both permutations are possible to create)

So the task is to find the maximal possible k for which we can create a pyramidal permutation.

This can be achieved by tracking the first index m of p where:

- if $i <$ position of number n and $p_i < p_{i-1}$.

- if $i \geq$ position of number n and $p_i > p_{i-1}$.

if there is no index m that satisfies the conditions above, p is pyramidal and we should directly go to observations 2 and 3 to get the answer.

If the desired m is found, there are two possibilities:

- $m <$ position of number n. We need to find the biggest $j \leq m$ that $p_j > p_{j-1} + 1$.
- $m \geq$ position of number n. We need to find the biggest $j \leq$(position of number n in p) that $p_j > p_{j-1} + 1$.

the index j that we found would be equal to $k + 1$, so we can construct the solution for fixed k in $O(n \cdot log(n))$ time. Basic implementation is to store every used value on the prefix in a set.

After that we take constructed permutation and using observations $2, 3$ find the answer.

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

Block 7: without additional constraints:

The final block requires you to optimize it further by getting rid of *set* structure and precomputing powers of 2 required to determine the answer.

Time complexity $O(n)$.

Problem 2D. A Little Bit of Formal Statements Never Hurt Anybody

Author: **Tsitsei Pavlo**
Task preparation: **Tsitsei Pavlo**
Editorial: **Tsitsei Pavlo**

Block 1: $n \leq 100$:

Let element a_i be maximum on subarray. Then we can go in the left to fix left end and then go in the right to fix right end. Solution works in $O(n^3)$.

Block 2: $n \leq 1\,000$:

Fix the left end of the subarray and iterate in the right for the right end. Also, fix index of maximum on the segment in parallel. Using prefix sum, we can check if segment is good in constant time, so solution works in $O(n^2)$.

Block 3: $a_1 < a_2 < \cdots < a_n$:

Let's extend the solution for the first block. If a_i is maximum, then i is the right end of subarray where a_i is maximum, because $a_{i+1} > a_i$. Then we already have $O(n^2)$ algorithm. We need to find the number of (l, r) where $a_l + \cdots + a_k = a_k + \cdots + a_r$ is true. We know that $k = r$, so $a_l + \cdots + a_r = a_r \implies a_l + \cdots + a_{r-1} = 0$. So the task is to find the number of subarrays with sum of all elements equal to 0 and $r < n$, which can be easily solved in $O(n)$ using hash table. Also, we need to add n to the answer, because $a_l = a_r$ is always a good subarray. So solution work in $O(n)$.

Block 4: there exists i $(1 \leq i \leq n)$, such that $a_1 < \cdots < a_i$ and $a_i > \cdots > a_n$:

Let's extend the solution for the third block. So we know how to solve task for $a_1 < \cdots < a_i$ $(l, r \leq i)$ and by symmetry for $a_i > \cdots > a_n$ $(l, r \geq i)$. So we can fix a_i and look for $l < i$ and $r > i$. We can iterate from $i - 1$ to 1 and fix l, memorize sum from l to i $(a_l + \cdots + a_i)$. Then iterate from $i + 1$ to n to fix r. Then we can easily find the number of l such that $a_i + \cdots + a_r = a_i + \cdots + a_l$. Using hash table, solution work in $O(n)$.

Block 5: $n \leq 10^5$:

Let's extend the solution for the first block. We can find first bigger on the left and right side using stack in $O(n)$. For element i we have l_i and r_i where l_i is first left bigger element and r_i is first right bigger element. If $i - l_i \leq r_i - i$, then we can iterate from i to l_i and fix l (symmetrical for r) and we will have sum $a_l + \cdots + a_i$. So we need to find a number of r such that $i \leq r < r_i$ and $a_i + \cdots + a_r = a_l + \cdots + a_i$. Let b be the prefix sum of a. Then $a_i + \cdots + a_r = a_l + \cdots + a_i$ is the same as $b_r - b_{i-1} = b_i - b_{l-1}$. So $b_r = b_i + b_{i-1} - b_{l-1}$. Let h be a hash table of arrays. Then we can memorize i in h_{b_i}. Using this, when we fix l we know what b_r is equal and using binary search find a number of such indexes r in range $[i, r_i)$. In fact, this solution works in $O(n \cdot log^2(n))$, because binary search in hash table works in $log(n)$ and brute force works in at most $n \cdot log(n)$.

Proof:

We will use math induction on number of elements. For $n = 1$ or $n = 0$, brute force will work in at most $n \cdot log(n)$. Consider $n > 1$. Let k be the index of the maximum element in the array. By induction, answer in subarray $[1, k-1]$ is at most $(k-1) \cdot log(k-1)$ and for $[k+1, n]$ is at most $(n-k-1) \cdot log(n-k-1)$. Also, $min(k-1, n-k) \leq \frac{n}{2}$, so solution work in $(k-1) \cdot log(k-1) + (n-k-1) \cdot log(n-k-1) + \frac{n}{2}$. It is the sum of one increasing and one decreasing function, so maximal value is when $k = \frac{n}{2}$ and we get $(k-1) \cdot log(k-1) + (n-k-1) \cdot log(n-k-1) + \frac{n}{2} \leq \frac{n}{2} \cdot log(\frac{n}{2}) + \frac{n}{2} \cdot log(\frac{n}{2}) + \frac{n}{2} = n \cdot (log(\frac{n}{2}) + \frac{1}{2}) = n \cdot (log(\frac{n}{2}) + log(\sqrt{2})) \leq n \cdot log(n)$.

1 Block 6: array consists of random integers $(-10^9 \leq a_i \leq 10^9)$:

Here we can use divide and conquerer method to solve this block in $n \cdot log(n)$. Firstly, let k be the index of the maximum element in the array. Then we can iterate through the right side and memorize in hash table the number of each prefix sum. After that, iterate through left side and using formula $b_r = b_i + b_{i-1} - b_{l-1}$ we can sum up all answers for this maximum in $O(n)$. Then we only need to find answers for subarrays $[1, k]$ and $[k, n]$ which we can do recursively. Solution works in $O(n \cdot log(n))$ on random tests.

Proof:

When tests are random, the probability that maximum is on segment $[l, r]$ is $\frac{r-l+1}{n}$. So if we assign $l = \frac{n}{4}$ and $r = \frac{3 \cdot n}{4}$, then probability is $\frac{1}{2}$. So, probability that maximum is near the center is $\frac{1}{2}$ and it means that in average 1 out of 2 times will decrease size of segment in 2 times, so solution decreases exponentially and, hence, have logarithmic states, so it works in $O(n \cdot log(n))$ in average.

Block 7: no additional restrictions:

Here is 2 solution, both work in $O(n \cdot log(n))$:

First:

Let's use divide-and-conquer. We have segment from l to r. Let $m = \lfloor \frac{l+r}{2} \rfloor$, and we count the number of segments that satisfy the statement and intersect point m. Let's assume that maximum is somewhere on the right side from m. While we iterate through the right side of the segment, we can uniquely determine which value should be maximum. If we currently at index r, then we know that maximum is somewhere on segment $[m, r]$ so it can only be maximum value there. Let k be the index of maximum element. Now we need to find the number of l such that $b_{l-1} = b_k - b_r + b_{k-1}$. Also, maximum on segment $[l, i]$ should be less than a_k. It can be done using two pointers. We can memorize all indexes on segment $[l, i]$ in hash table and while maximum on segment $[l, i]$ is less than a_k we can memorize b_l and then decrease it. Also, we need to take answers from recursive calls on the left and right side.

Second:

We can extend solution for the 6 block. There is $O(n \cdot log^2(n))$ because of the binary search, so we will get rid of it. So we need to find number of r in range $[i, r_i)$ with fixed value. In other words, we are given at most $n \cdot log(n)$ queries of type find number of numbers x on the segment $[l, r]$. We can use divide-and-conquer on queries. So for fixed l_i, r_i we have $m = \lfloor \frac{l_i + r_i}{2} \rfloor$ where each query goes through m. We can divide query $x, [l, r]$ to queries $x, [l, m]$ and $x, [m+1, r]$. We have fixed m so we can just iterate and memorize in hash table answers in parallel, so we would get solution in $O(n \cdot log(n))$