Solution description, InfO(1)Cup 2024

InfO(1)Cup 2024 Scientific Committee

February 2024

Lattice

Author: Lucian Badea

We are given a set of points $S = \{(x_1, y_1), \ldots, (x_P, y_P)\}$ and an integer K. We are asked to find for how many integers n we have that $n \mid x_i$ and $n \mid y_i$ for at least K indices $i \in \{1, \ldots, P\}$.

We first observe the following $n \mid x_i$ and $n \mid y_i$ if and only if $n \mid \gcd(x_i, y_i)$, where $\gcd(a, b)$ is the greatest common divisor of a and b. Thus, replacing (x_i, y_i) with $\gcd(x_i, y_i) = g_i$, we reduce our problem to the following: given a set $S = \{z_1, \ldots, z_}$ and an integer K, for how many n do we have $n \mid z_i$ for at least K indices $i \in \{1, \ldots, P\}$.

How can we do this efficiently? First, create an array $A[z] = \#\{i \mid z_i = z\}$ i.e. $A[z]$ is the number of elements z_i equal to z. Now, fix some integer $n \in \{1, \ldots, V\}$, where $V = \max_i z_i$, and consider all the multiples of n. An integer n is counted towards the solution if $\sum_{m=1}^{\lfloor V/n \rfloor} A[mn] \geq K$. To compute the sum on the left requires at most V/n iterations, so overall this algorithm requires $\sum_{n=1}^{V} V/n = O(V \log V)$ iterations overall. This is sufficient to solve the task.

Gregor and maximum length

Author: Rares Iordache Mihai

We are given a sequence $a_1, \ldots, a_n \in \{1, \ldots, V\}$, and, for a set $S \subseteq \{1, \ldots, V\}$ we can eliminate all a_i such that $a_i \in S$. We want to do this so that the resulting sequence is non-decreasing and of maximum length.

This problem is equivalent to the following one: find the maximal non-decreasing subsequence of $a_1, \ldots, a_n$ which, if a value a belongs to the subsequence, then all other values equal to a must belong to the subsequence. We call a subsequence with this

property *full*. Our algorithm will thus be a modification of the algorithm for finding the longest non-decreasing subsequence. Let f_x be the number of indices i for which $a_i = x$.

Let us step through the elements in $a_1, \ldots, a_n$. When we are at step i, we will maintain a set $P = \{(x_1, l_1), \ldots, (x_k, l_k)\}$ which represents the maximal full non-decreasing subsequences which terminate at or before element a_i. A pair (x, l) represents a full non-decreasing subsequence terminating at x with length l. Observe that if $x_i \geq x_j$ but $l_i \leq l_j$ then (x_j, l_j) is always pointless (it is worse that (x_i, l_i) since $l_i \leq l_j$, but also it is harder to continue this sequence since $x_i \geq x_j$). Since we care about maximising the length of the subsequence, we will maintain the property that $x_1 \leq x_2 \leq \ldots \leq x_k$ and $l_1 \leq l_2 \leq \ldots \leq l_k$.

Suppose that at step i we are at the leftmost element equal to a_i. Then we can see, for all the full non-decreasing subsequences ending with elements equal to a_i, what will be the longest possible subsequence up to this point. It must be a subsequence represented by (x, l) with $x < a_i$; furthermore, it is optimum to take the maximum such x, since this corresponds to the maximum l. Thus, find the maximum t such that $x_t \leq a_i$, observe that the longest possible full non-decreasing subsequence ending at an element equal to a_i must be of length $l_t + f_{a_i}$. We also store the fact that the second-to-last element in such a sequence is x_t — let this fact be denoted by $p[a_i] = x_t$.

Now, when we arrive at the element a_j which is the rightmost element equal to a_i, we must introduce this subsequence to the set P. Let $x = a_i$ be the last element in this subsequence, and $l = l_t + f_{a_i}$ be the length of it. Firstly, we observe that (x, l) cannot be added into P if the largest $x' \leq x$ for which $(x', l') \in P$ has $l' \geq l$ — since then (x', l') is better than (x, l). Otherwise, (x, l) must be added into P, and it might erase several elements (x'', l''). Indeed, we must repeatedly find the smallest $x'' \geq x$, and check if $l'' \leq l$, in which case we must erase (x'', l'') from P. At the end of the algorithm, we get the list $P = \{(x_1, l_1), \ldots, (x_k, l_k)\}$ and observe that the solution is given by l_k. Furthermore, by going from x_k to $p[x_k]$ to $p[p[x_k]]$ and so on, we can find the elements which we must not erase.

Now, let us turn to the data structure P. We can see that P holds pairs (x, l); we interpret x as the key, and l as the value. P must be able to retrieve the value corresponding to a key, must be able to support inserting and deleting elements by key, and must also be able to find the smallest key greater than some given key, and the largest key smaller than some given key. We observe that the data structure `std::map<int, int>` does precisely this, in $\log n$ time.

How many operations must be done on P? We observe that we must do $O(n)$ key searches, and $O(n)$ insertions into P. The number of deletions is not obvious from the algorithm — but observe that it must be at most the number of insertions (one cannot delete more elements than one has inserted!) so it is also $O(n)$. It follows that this algorithm takes $O(n \log n)$ time.

XorSecv

Author: Iulian Arsenoiu, Tudor Muşat, Tamio Vesa-Nakajima

We are given an array $a_1, \ldots, a_n$ and an integer P. We must compute

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=0}^{j-i} (a_{i+k} \operatorname{xor} k)^P.$$

We introduce the *Iverson bracket*. For some statement ϕ, if ϕ is true then $[\phi] = 1$, otherwise $[\phi] = 0$. So, for example, $[0 < 1] = 1$ but $[1 < 0] = 0$.

Thus, we can rewrite the previous formula as follows.

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=0}^{n} (a_{i+k} \operatorname{xor} k)^P [k \le j - i] = \sum_{k=0}^{n} \sum_{i=1}^{n} (a_{i+k} \operatorname{xor} k)^P \sum_{j=i}^{n} [k + i \le j]$$
$$= \sum_{k=0}^{n} \sum_{i=1}^{n} (a_{i+k} \operatorname{xor} k)^P \max(n - k - i + 1, 0) = \sum_{k=0}^{n} \sum_{i=1}^{n-k} (a_{i+k} \operatorname{xor} k)^P (n - k - i + 1).$$

We now introduce a change of variables. Let $t = i + k$. Then the previous expression is equal to

$$\sum_{k=0}^{n} \sum_{t=k+1}^{n} (a_t \operatorname{xor} k)^P (n - t + 1) = \sum_{k=0}^{n} \sum_{t=1}^{n} (a_t \operatorname{xor} k)^P (n - t + 1)[t \ge k + 1]$$
$$= \sum_{t=1}^{n} \sum_{k=0}^{n} (a_t \operatorname{xor} k)^P (n - t + 1)[t \ge k + 1] = \sum_{t=1}^{n} (n - t + 1) \sum_{k=0}^{t-1} (a_t \operatorname{xor} k)^P$$

Hence it is sufficient to compute the following function

$$f(x, t) = \sum_{k=0}^{t-1} (x \operatorname{xor} k)^P,$$

as the solution is then just $\sum_{t=1}^{n} (n - t + 1) f(a_t, t)$.

Now, consider the set $S(x, t) = \{x \operatorname{xor} k \mid 0 \le k < t\}$. We have that $f(x, t) = \sum_{y \in S(x,t)} y^P$. Thus it would be helpful for us to first understand the structure of $S(x, t)$ better.

Structure of $S(x, t)$. Suppose that $t = b + 2^a$, where $2^{a+1} \mid b$. In other words 2^a is the least significant 1-bit of t. Then we observe that $S(x, t)$ is equal to

$$S(x, b) \cup \{x \operatorname{xor} (b + i) \mid i \in \{0, \ldots, 2^a - 1\}\}.$$

But note now that $x \text{ XOR } (b+i)$ where $i \in \{0, \dots, 2^a - 1\}$ can be made equal to any intgeri which coincides with $x \text{ XOR } b$ for all except the last a bits, in exactly one way. Hence

$$S(x, b + 2^a) = S(x, b) \cup \{x \text{ XOR } b, \dots, (x \text{ XOR } b) + 2^a - 1\}.$$

It follows that each $S(x, t)$ is the union of at most $\log n$ different disjoint intervals (as $t \le n$). This is the key property we will use.

Computing $f(x, t)$. Let $T[n] = 1^P + \cdots + n^P$. Observe that $x^P + \cdots + y^P = T[y] - T[x-1]$ (if $T[0] = 0$). We observe that $S(x, t)$ is the disjoint union of $\log n$ different intervals; since we want to compute $\sum_{y \in S(x,t)} y^P$, we can use this decomposition of $S(x, t)$ into intervals to split this sum into the sum of those $\log n$ intervals, each of which can be computed using T.

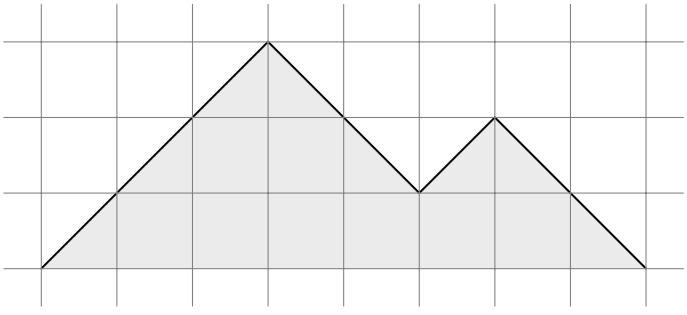
We then use this function f to compute the result as explained above. We note that the total time complexity is $O(n \log n)$.

Bracket wheel

Author: Predescu Sebastian

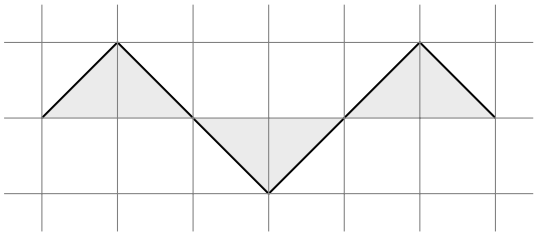
In this problem, we are given a sequence of brackets $s_1, \dots, s_n$, and we must swap at most k pairs of brackets so that as many cyclic shifts of the resulting sequence form a properly balanced sequence of brackets. To solve this problem we first devote some time to understanding bracket sequences.

Structure of sequences of brackets. Consider for instance the sequence of brackets (((())())), which is properly balanced. We can alternately represent this as a "mountain", in the following way:

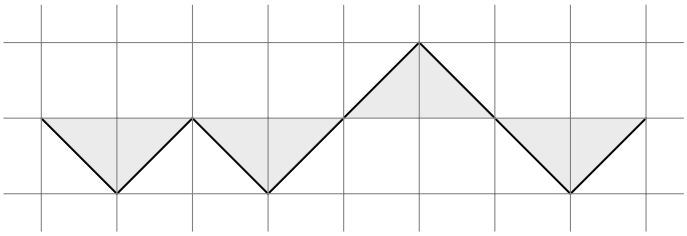


We draw the mountain from left to right; every time we reach a (we go up one step, and every time we reach a) we go down one step. With this representation, when is a

sequence of brackets properly balanced? Firstly, it must go from height 0 to height 0 (which is guaranteed by the fact that $s_1, \dots, s_n$ has n open and closed brackets respectively). Second, the mountain must never go below 0. For example ())(() is not properly balanced:



Now, we are ready to understand the number of cyclic shifts that are properly balanced. Given a sequence of brackets that contains the same number of closed and open brackets, the number of cyclic shifts that are balanced is equal to the number of times the mountain passes through the minimum height. For instance, for)()(())(, we have the following mountain.



The mountain passes through the minimum 3 times, and hence there are 3 balanced cyclic shifts, i.e. (())()(), ()(())() and ()()(()).

We finally observe that we can initially rotate the sequence so that the mountain never passes below 0. We do this by first computing the heights of the mountain, then rotating the sequence to begin at one of the minimal heights. We do this, so we assume the mountain of $s_1, \dots, s_n$ never passes below 0.

Understanding swaps. We now consider what a swap does to our sequences. There are two cases. Firstly, if $n = 2$ then clearly the result is always 1, since there are only two possible bracket sequence i.e. () and)(, and the result is always 1. So assume $n > 2$. Now, a swap can either swap two equal parentheses (in which case nothing happens), or it can swap two different parentheses.

Now, note that the effect of the operation is the same as turning one (into a), and one) into a (. Thus, we can reach a sequence $t_1, \dots, t_n$ from $s_1, \dots, s_n$ by swapping pairs of brackets if and only if we can reach it by turning x open parentheses to closed parentheses, and x closed parentheses to open parentheses, for some $0 \leq x \leq k$.

Algorithm. We are now ready to describe our algorithm. It will use the dynamic programming strategy. Let $d[i][m][x][y]$ denote the maximum numbers our mountain can go through the minimum, if we consider the first i parentheses, the current minimum is m, we have turned x open brackets to closed brackets, and y closed brackets to open brackets. We only need this information for $i \in \{0, \dots, n\}$, $m \in \{-2k, \dots, 0\}$, $x, y \in \{0, \dots, k\}$. Observe that the base case is $d[0][0][0][0] = 1$; we can then set $d[0][\star][\star][\star] = -\infty$ for all other cells with $i = 0$.

We will present our algorithm using a "forward update"-style dynamic programming strategy. This means that for every state (i, m, x, y), we compute which states we can reach, and update those states accordingly. Thus suppose we have reached state (i, m, x, y). We need to update two potential cases: if we change parenthesis $i+1$ or not. Suppose $\Delta x = 1$ if $s_{i+1} =$ (and we change it to), and 0 otherwise; likewise let $\Delta y = 1$ if $s_{i+1} =$) and we change it to (. We will then reach a state $(i+1, m', x', y')$ with $x' = x + \Delta x, y' = y + \Delta y$. What will be m'? We have two cases.

We change the minimum. Suppose $h[i]$ is the height of point i in the mountain for $s_1, \dots, s_n$ *without changing anything*. The height at the new state will be $h[i] - 2x' + 2y'$. If $m > h[i] - 2x' + 2y'$ then the new minimum will become $h[i] - 2x' + 2y'$, so we need to update the $d[i+1][h[i] - 2x' + 2y'][x'][y]$ with value 1.

We do not change the minimum. Suppose now that $h[i] - 2x' + 2y' \geq m$. Then we need to update $d[i+1][m][x'][y']$ with either $d[i][m][x][y]$ or $d[i][m][x][y]+1$, with the +1 being applied if $h[i] - 2x' + 2y' = m$ i.e. the minimum is hit.

Applying the described algorithm, we get an $O(nk^3)$ dynamic programming algorithm. By saving only two "lines" of the array d i.e. $d[i][\star][\star][\star]$ and $d[i+1][\star][\star][\star]$, we also get $O(k^3)$ memory, which greatly optimises both the time and memory complexity.

Ping-Pong

Author: Lucian Badea

In this problem, we are given three integers, a, b and c. We must calculate the number of different configurations of ping-pong games that can be played, so that in the end, the first player has a wins, the second player has b wins, and the third player has c wins. After every game, the winner stays at the table and plays against the person that was initially watching.

Interpreting the problem. First of all, it is important to find a way of interpreting the problem, so that it becomes a string counting problem, rather than what it initially

was. For simplicity, we will count the number of configurations in which each player watches the first game. For instance, we assume that *Alice* does not play in the first game. From her point of view, any game configuration can be seen as a string, each character of the string being equal to W (a win), L (a loss) or S (a match which she spectated). Then, these strings have some properties: Firstly, since Alice watches the first game, all of these strings must begin with an S. Secondly, after any L, the string can either end, or there must be an S.

Therefore, any string representing a configuration can be of one of the following two forms:

$$S\underbrace{WW\ldots W}_{L_1}LS\underbrace{WW\ldots W}_{L_2}L\ldots S\underbrace{WW\ldots W}_{L_{nr}}L$$

or

$$S\underbrace{WW\ldots W}_{L_1}LS\underbrace{WW\ldots W}_{L_2}L\ldots S\underbrace{WW\ldots W}_{L_{nr-1}}LS\underbrace{WW\ldots W}_{L_{nr}}$$

where $0 \leq L_i$ for all $1 \leq i \leq nr$. We call a sequence comprising an S, several W and an L a *complete* sequence, since the outcomes of the games in the sequence do not influence any game outside of the sequence, and at the end of a complete sequence, the player watching is the same as at the beginning of the sequence.

Understanding how complete sequences affect points. It is easy to see that $a = L_1 + L_2 + \cdots + L_{nr}$. However, a complete sequence can decode two possible scenarios: when Bob wins the first game of the sequence, or when Charlie wins the first game of the sequence. Depending on the length of the sequence, the score of Bob and Charlie can change as follows:

The length of the sequence is even. Then, the player who wins the first game also wins the last one, therefore his score is increased by 2, while the other player's score remains unchanged.

The length of the sequence if odd. Then, the player who loses the first game wins the last one, therefore both players' scores are increased by 1.

Counting the number of scenarios. Firstly, the first type of string that represents a configuration can be seen as a concatenation of multiple complete sequences. The second type of string can be seen as a string of the first type of length $a + b + c + 1$ instead of $a + b + c$, from which the last character is removed. Depending on the parity of the length of the last complete sequence of this string, the number of points gained by Bob and Charlie also changes. If the length is even, then the player who wins the first game of the last sequence loses a point. Otherwise, the other player loses a point. We make the following notations (from the first player's perspective):

- $E(x, y, z)$ is the number of scenarios where the first player wins x games, the second player wins y games, the third player wins z games, the first game of the last complete sequence is won by the third player, and the last complete sequence is even
- $O(x, y, z)$ is the number of scenarios where the first player wins x games, the second player wins y games, the third player wins z games, the first game of the last complete sequence is won by the third player, and the last complete sequence is odd

Then, the total number of scenarios (where Alice spectates the first game) can be divided into:

- $O(a, b, c) + O(a, c, b) + E(a, b, c) + E(a, c, b)$: the number of configurations encoded by strings of the first type
- $O(a, b, c + 1) + O(a, c, b + 1)$: the number of configurations encoded by strings of the second type, which end in an odd-length sequence
- $E(a, b, c + 1) + E(a, c, b + 1)$: the number of configurations encoded by strings of the second type, which end in an even-length sequence

We are left with calculating $O(x, y, z)$ and $E(x, y, z)$. Let k be the number of odd-length complete sequences in the string. Let nr_y be the number of even-length complete sequences which add 2 points to the score of the second player, and nr_z be the number of even length complete sequences which add 2 points to the score of the third player. Then, $nr_y = \frac{y-k}{2}$ and $nr_z = \frac{z-k}{2}$. Moreover, nr, the total number of complete sequences, is equal to $k + nr_y + nr_z$. It is important to note that when either one of this numbers is not an integer, the values of $O(x, y, z)$ and $E(x, y, z)$ are 0. For a fixed k, the number of different strings is equal to the number of ways of assigning even lengths to all complete sequences, multiplied by the number of ways to choose k sequences out of them and add 1 to their length. For $O(x, y, z)$, this is equal to

$$\binom{\frac{x-k}{2} + nr - 1}{nr - 1} \times \binom{nr - 1}{k - 1}$$

(since one of the odd sequences is always the last one), and for $E(x, y, z)$, it is equal to

$$\binom{\frac{x-k}{2} + nr - 1}{nr - 1} \times \binom{nr - 1}{k}$$

(since the last sequence is always even). Then, we must select the even sequences which add 2 to the score of each player, and the winner of the first match of all odd sequences. For $O(x, y, z)$, this is equal to $\binom{nr_y+nr_z}{nr_z} \times 2^{k-1}$, and for $E(x, y, z)$ it is equal to $\binom{nr_y+nr_z-1}{nr_z-1} \times 2^k$.

Therefore, letting

$$M_O = \{x \in \mathbb{N} | 1 \le x \le \min(a, b, c), 2|(a-k)\},$$
$$M_E = \{x \in \mathbb{N} | 0 \le x \le \min(a, b, c-1), 2|(a-k)\},$$

we have the following formulas:

$$O(x,y,z) = \prod_{k \in M_O} \binom{\frac{x-k}{2} + nr - 1}{nr-1} \times \binom{nr-1}{k-1} \times \binom{nr_y + nr_z}{nr_z} \times 2^{k-1}$$
$$E(x,y,z) = \prod_{k \in M_E} \binom{\frac{x-k}{2} + nr - 1}{nr-1} \times \binom{nr-1}{k} \times \binom{nr_y + nr_z - 1}{nr_z - 1} \times 2^{k}$$

We can also similarly calculate the number of configurations in which Bob and Charlie don't play in the first match.

Maxstack

Author: Iulian Arsenoiu

Interpreting the problem. First of all, this problem can be entirely solved without tree knowledge. However, it is easier to understand the solution when thinking about the array of operations as a tree. (We note that trees are included within the syllabus.) The tree is constructed as follows: Initially, there is only one node in the tree, node 0. Each push operation adds a new child to the node we are currently in, assigns the value added to the new edge (from now on, we will call it the edge of the node), and then sends us to the new node, while a pop operation sends us to the parent of the node we are currently in. Furthermore, by $s(x)$ we denote the operations in the array from the one which adds x, to the one which sends us to its parent.

Solution for $C = 1$. Every query (l, r) can be seen as starting from one of the nodes in the tree and simulating the operations of $s(l, r)$ on that node. After each operation, the maximum value on the path from our current node to the starting node is added to the result. Let val_{node} be the result of $s(node)$. Let up_{node} be the first node on the path from $node$ to 0 with the property that the value on the edge of up_{node} is strictly greater than the value on the edge of $node$. Then the value of the edge of node $node$ can only affect the val of the nodes on the path between $node$ and up_{node} (not including up_{node}). Let $nrch_{node}$ be the number of children of $node$, and nr_{node} be the number of times the value of the edge of $node$ is the maximal value in the stack, when performing the operations of $s(node)$. Then, $nr_{node} = 1 + nrch_{node} + \sum_i^{up[i]=node} nr_i$. The value of val for all nodes on the path

between *node* and up_{node} must be increased with nr_{node} multiplied by the value of the edge of *node*. This can be done using *Difference arrays*, or *Binary indexed trees*. Finding the values of *up* can be either done with a *Range minimum query*, or with a stack which, when entering a node, removes all values smaller than the value of the edge of that node, and when it returns to the node, adds them again.

A finally empty query is equivalent to calculating the value of *val* of a certain node. A normal query is equivalent to calculating the sum of the values of *val* for several consecutive nodes with the same parent. Depending on the implementation, the time complexity can either be $O(N \log_2 N)$ or $O(N)$.

Understanding the queries for $C = 2$. We define a series of nodes as a maximal set of nodes which have the same parent. Then, let $S_1, S_2, \ldots, S_{k_{node}}$ be the children of a node. For a query (l, r), let l_{node} be the first node in S so that $s(S_{l_{node}})$ is included in (l, r), and r_{node} be the last node in S with this property. If no such node exists, we consider l_{node} to be 1 and r_{node} to be 0. Then, the answer to the query is equal to:

$$\sum_{node=1}^{n} \sum_{i=l_{node}}^{r_{node}} \sum_{j=i}^{r_{node}} \sum_{p=i}^{j} val_{S_p}$$

Subtasks 5 & 6. For these two subtasks of $C = 2$, we are guaranteed that the query sequence is correct. For a query (l, r), let *top* be the parent of the node added when performing the operation l. Then, except for the series of node *top*, all series are either completely included in the query, or not included at all. Let sum_{node} and $psum_{node}$ be:

$$sum_{node} = \sum_{i=1}^{k_{node}} \sum_{j=i}^{k_{node}} \sum_{p=i}^{j} val_{S_p}$$

$$psum_{node} = sum_{node} + \sum_{i}^{parent(i)=node} psum_i$$

The values of *sum* and *psum* can easily be calculated using a *Depth-first search* and some mathematical formulas. Then, for Subtask 5, it is enough to output $psum_{top}$. For Subtask 6, the result is equal to:

$$\sum_{i=l_{node}}^{r_{node}} psum_i + \sum_{i=l_{node}}^{r_{node}} \sum_{j=i}^{r_{node}} \sum_{p=i}^{j} val_{S_p}$$

This can be calculated using some more complex difference arrays and mathematical formulas.

Subtasks 7 & 8 For Subtask 7, we can go through all series and apply the formulas found before using a brute-force algorithm. Please note that going through all series, and for each series, going through all of its nodes is fast enough, since a node can only be part of a single series.

For Subtask 8, we must further optimise the brute-force algorithm. We will divide the series into two categories: light - series having less than C nodes, and heavy - series having more than C nodes. To solve the queries for the heavy series, it is enough to go through all heavy series (note that there cannot be more than $\frac{N}{C}$ such series) and use the same difference arrays as in Subtask 6. For the light series, we must solve the queries offline. For each series, we will use a data structure (for example a square root decomposition or a segment tree), and each time we get to a new node, we update all the nodes in the series accordingly (note that we must update at most C nodes). Depending on the data structure used, the optimal choice of C can either lead to a time complexity of $O(N\sqrt{N} + Q\sqrt{N})$ or $O(N\sqrt{N \log_2 N} + Q\sqrt{N \log_2 N})$.

Solution for $C = 2$. The last observation, which leads to a complete solution, is that due to the restriction for $C = 2$, it is guaranteed that for a query (l, r), except for the series of node *top*, the nodes that are completely included in the query are a prefix of the series. Please note that a prefix can also mean the whole series. So, we must treat the *top* node separately, as we did for Subtask 6. For the other nodes, we will add their values to the query answers offline, using a *Line Sweeping Algorithm*. For a node, let l_{node} be the index of the first operation in $s(node)$. We will process the array from left to right. Every time we get to a pop operation, it means that the current prefix of a series has just been extended. We will use a binary indexed tree, and for the series of a node *node*, we will update the value with which its prefix contributes to the answers on the position l_{node}. When getting to position *pos*, we will answer to all queries (l, r) for which $r = pos$. The answer of a query (l, r) is equal to the sum of the numbers in the range $[l, r]$ in the binary indexed tree. The total time complexity of this algorithm it $O(N \log_2 N + Q \log_2 N)$.

Taking the problem further. After reading this editorial, it might not be surprising to find out that this problem can be solved without the restrictions on the queries provided in the statement. For $C = 1$, it is enough to check whether the sequence is correct or not, otherwise the answer is 0. For $C = 2$, if we remove the restriction, we are left with the idea that for each query, except for the series of node *top*, the nodes that are completely included in the query are either a prefix or a suffix of the series. However, we must change the definition of node *top*, as being the *Lowest Common Ancestor* of all the nodes in the query. The case for the suffixes is similar to the one for the prefixes, therefore the problem can be still solved in $O(N \log_2 N + Q \log_2 N)$ total time complexity.