RMI 2021, Day 1, English Editorial

RMI Scientific Committee

Thursday 16th December, 2021

Problem 1: Gardening

(Proposed by *Tamio-Vesa Nakajima*.)

Observation 1. *Consider a region that is $2 \times (2k+1)$ cells. Suppose that the first row contains only flowers of type t, and the second row contains t on the first and last cell. Call such a region a $(2k+1)$-cup. Such a region cannot exist for $k > 0$.*

Proof. This can be shown by induction. For $2k+1 = 3$, this can be checked immediately. For $2k+1 > 3$, note that in the second row of the $(2k+1)$-cup contains an odd sequence, say of length $2k'+1 < 2k+1$, containing one type of flower. If no third row exists, then clearly the situation is impossible. otherwise, the odd length sequence determines a $(2k'+1)$-cup, which cannot exist. □

Corollary 1. *Thus no $N \times M$ garden can exist when N or M are odd.*

Proof. Consider M odd. Create an $(N+2) \times (M+2)$ garden by adding a new type of flower around the old garden. Then the first two rows form a $(M+2)$-cup, which cannot exist as $M+2$ is odd. □

Corollary 2. *Thus for a garden to be possible, we must have N and M even.*

Thus, from here on assume N, M are even.

Observation 2. *Any $N \times M$ garden satisfies $K \leq NM/4$, and $K \neq NM/4 - 1$.*

Proof. Any type of flower must occupy at least 4 cells, and if it occupies more, it must occupy at least 12 cells. This can checked by exhaustion. Thus clearly we must use at most $NM/4$ flower types. Furthermore, if we were to remove flowers, one flower must take up more than 4 cells; it must thus take up at last 12 cells, and we can use at most $NM/4-2$ flower. So $K \neq NM/4-1$. □

Observation 3. *For any row in a garden, there exists a flower type appearing within that row that appears only at that row and below, or at that row and above.*

Proof. Say that a segment on this row is *closed* if it contains all the appearances within that row of all the flower types within that segment. We prove by induction that any closed segment contains a flower type that appears only at this row and above/below it. The base case is a segment of length 2, which is obvious: such a segment corresponds to a 2×2 square with only one flower type. Now consider longer segments. Suppose a flower type appeared only in one continuous subsegment of our closed segment. It then follows that the region bounded by that flower type does not intersect the row – which implies that the flower type only appears at the row and on one side of it, as required. Otherwise consider any flower type that appears as at

least two continuous subsegments of our segment. Then th interior of the region bounded by this flower type intersects our segment. Take any segment of the intersection of this interior with our segment – this is also a smaller closed subsegment. Thus we have completed our inductive argument, as this smaller closed subsegment must contain a flower type that appears at this row and only above/below it.

Note also that the entire row is a closed segment. Thus is must contain at least one color that appears only at the row and above/below it. □

Observation 4. *For any $N \times M$ garden, we have $K \geq \max(N/2, M/2)$.*

Proof. Without loss of generality let $M \leq N$. Thus we want to prove that $K = N/2$. Suppose $K < N/2$. Consider the top-most and the bottom-most row where each flower type appears. There are $2K < 2N$ such rows, so for at least one row all flowers that appear have appeared both above and below it. But this is impossible by the previous observation. So $K \geq N/2$. □

Observation 5. *For an $N \times N$ garden, $K \neq N/2 + 1$.*

Proof. Suppose $k = N/2 + 1$. Consider like in the previous observation the rows where each flower type first and last appears. Clearly for at least one row there are two flower types that either first or last appear at that row. Furthermore, because each row contains at least one flower that appears first/last at that row, we can deduce that exactly two rows have exactly two flowers that appear first/last at that row.

We now investigate the structure of the garden more fully. Suppose that for exactly the first R rows we have had exactly one flower type appear. Then no flowers can appear last at row $R+1$ unless $R = N/2$ (since the "flower type in the middle" is the only that can stop appearing, and it can only stop appearing when it has width 2, i.e. at $R = N/2$). Thus at row $R + 1$ we must have exactly two flower types appearing. If both of these had width greater than 2, then at the next row we would be forced to have again two flower types appearing, but this would mean that there are too many flowers, so this cannot happen. So one of the flower types has width two, and disappears on the next row. All other rows have exactly one appearance or disappearance; and in fact all the rows from $R + 2$ up to $N/2$ have one appearing flower, and all those left after $N/2$ have exactly one disappearing flower. We can thus deduce that $(N - 2R) \times (N - 2R)$ that is between rows and column $R + 1$ and $N - R$ has a 2×2 corner of one type, and otherwise is comprised of concentric flower types. Furthermore, it must have $N/2 + R$ colors. This is impossible. □

Observation 6. *For any even N, M, a garden exists for any K where $\max(N/2, M/2) \leq K \leq NM/4$, $K \neq NM/4 - 1$, and either $N \neq M$ or $K \neq N/2 - 1$.*

Proof. We provide a recursive procedure that achieves this:

1. If $K = NM/4$ then fill the entire garden with 2×2 areas with one flower type each.

2. Suppose $NM/4 - N - M + 1 \leq K < NM/4 - 1$. Then there exists a rectangle boundary of appropriate size, with one corner in one of the corners of the garden, that if planted with one flower type, and the rest of the garden filled with 2×2 areas with one flower type each, gives us a garden with K flower types.

3. Suppose $K = NM/4 - N - M$. Suppose without loss of generality that $N \leq M$. If $N \leq 4$ then $K \leq \max(N/2, M/2)$ which is impossible – so $N, M \geq 6$. First create a $N \times (M - 2)$ rectangle filled border with one flower type, with its corner in the corner of the garden; then within that rectangle create a 4×4 rectangle border, filled with one flower type. Fill the rest of the flower with 2×2 regions each with one flower type.

4. Suppose $K < NM/4 - N - M$. Then it can be proved that we can fill the border of the garden with one flower type, and recursively find the answer for $N - 2, M - 2, K - 1$.

With a careful implementation this can be done in $O(NM)$. □

Problem 2: Present

(Proposed by *Tamio-Vesa Nakajima*.)

Say that a set is gcd-closed if it contains the greatest common divisor of all pairs of its elements. Let $\mathcal{G}$ be the set of (finite) gcd-closed sets. The problem asks for the K-th gcd-closed set in *Laikan* order. More usually, *Laikan* order is called colexicographic order; if A is before B in Laikan/colexicographic order, write $A \preceq B$. Furthermore, we define an "opposite" of the gcd-closed sets. Call these gcd-independent sets: a set S is gcd-independent if and only if for all $x, y \in S$ we have $\gcd(x, y) \notin S$. Let $\mathcal{I}$ be the set of (finite) gcd-independent sets.

Note that for any collection of sets $\{S_i\}_{i \in I} \subseteq \mathcal{G}$ we can deduce that $S = \bigcap_{i \in I} S_i \in \mathcal{G}$. This is because if $x, y \in S$ then $x, y \in S_i$ for all $i \in I$, so $\gcd(x, y) \in S_i$ for all $i \in I$, so $\gcd(x, y) \in S$. Thus we can state the following definition.

Definition 1. For any set $S \subseteq \mathbb{N}$, let $\langle S \rangle \in \mathcal{G}$ be the smallest superset of S that is gcd-closed, i.e.

$$\langle S \rangle = \bigcap_{S \subseteq T \in \mathcal{G}} T.$$

We call this the gcd-closed set *generated* by S.

Observation 7. *For any $S \in \mathcal{G}$ there exists a unique $T \in \mathcal{I}$ such that $S = \langle T \rangle$.*

Proof. To find T, start with $T = \emptyset$. While $S \neq \emptyset$, remove $x = \max S$ from S, add x to T, then remove $\gcd(x, y)$ for all $y \in T$ from S. T remains gcd-independent throughout, and $\langle T \rangle = S$. Furthermore, each step is in a sense "forced" (the maximal element must always belong to T for instance), so T is unique. □

Observation 8. *For any $S, T \in \mathcal{I}$, we have $S \preceq T$ iff. $\langle S \rangle \preceq \langle T \rangle$.*

Proof. Left as an exercise; one direction can be shown using the procedure from the previous observation. □

These facts essentially show that $\langle \cdot \rangle$ is an order-preserving bijection between $\mathcal{I}$ and $\mathcal{G}$. Thus finding the K-th gcd-closed set is in fact equivalent to finding the K-th gcd-independent set, and generating a gcd-closed set from that. An interesting subproblem is finding the next gcd-independent set in colexicographic order.

Observation 9. *Given some $S \in \mathcal{I}$, the following procedure can compute the smallest $T \in \mathcal{I}$ that is greater than S (in colexicographic order):*

1. *Let x be the smallest integer* not *in $\langle S \rangle$.*
2. *Add x to S, and remove all elements less than x.*
3. *The resulting set is T.*

In order to make repeatedly computing the next gcd-independent set in colexicographic order efficient, one must also maintain the gcd-closed sets generated by these gcd-independent sets. To do this, maintain the elements of the current gcd-independent sets on a stack together with the elements in the generated set added because of the current elements. Removing small elements from S is then equivalent to popping the stack. To push x to the stack, we must find all the elements that are not currently in the generated set that must be added as a consequence of adding x. This can be done by iterating over the divisors d of x (that are not currently in the generated set) and checking if some element y is in the set for which $\gcd(x, y) = d$. All of this

can be made efficient by precomputing $\{y \mid \gcd(x, y) = d\}$ for all pairs (x, d) that can appear in the problem (it can be proved that $x, d \leq 40$). Furthermore, as an implementation note, it is much more efficient to represent sets as bitmasks.

We have now found a procedure that can, for a gcd-independent set, efficiently find the next gcd-independent set in colexicographic order. To solve our problem, precompute the 0-th, 10^7-th, $(2 \cdot 10^7)$-th, etc., gcd-independent sets, storing them in the source code for the problem. Then, on run-time, starting from one of the precomputed gcd-indpendent sets, iterate to the desired one. Recall of course that finding the K-th gcd-independent set is equivalent to finding the K-th gcd-closed set.

Problem 3: Speedrun

(Proposed by *Matei Tinca*.)

Subtask 1. For each node, we can just hold it's line in the adjacency matrix. That is, for node x, we store a string of length N, where the i-th bit is 1 if there is an edge between x and i. To walk through every node, we can just run a dfs algorithm, because for each node we have the list of its neighbours.

Subtask 2. Since the graph is a star, we have a central node which is connected to every other node. We can label every node with the index of the central node written in binary form. For instance, if the central node is 13, we can store 0000001101 in every node (although, if we hold the bits in reverse order, the implementation would be easier). Therefore, we know the structure of the entire tree from every node, so we can run a dfs algorithm to go through every node. Also, since we have 20 bits, we only use the first 10, which are enough, since they can store every number from 0 to $2^{10} - 1 = 1023$.

Subtask 3. Since the graph is a path, for each node we can store its neighbours, or 0 if the node is an endpoint. To go through every node, we go to the left as much as we can, and after hitting the endpoint, we can go to the right until we hit the other endpoint.

Subtask 4. In this subtask we must exploit the fact that we have a lot of possible calls and big labels. For this, we must observe that there are at most $\sqrt{N}$ nodes which have a degree higher than $\sqrt{N}$. So, for this subtask, for the nodes with less than $\sqrt{N}$ neighbours, we can just hold its adjacency list. This will use at most $\sqrt{N} \log N$ bits. For the other nodes, we don't need to store anything, and we can just try every other node. In the end, each node will use at most $\sqrt{N} \log N$, and we do at most $N\sqrt{N}$ wrong calls.

Subtask 5. In this solution, we use labels of size $3 \log N$ and we make no wrong calls. After rooting the tree in one node, for instance node 1, for each node, we can hold its father, one of its sons, and a link to one of it's "sibling". Therefore, for a node we can discover all of its neighbours: the father, one of the sons, and by going from "sibling" to "sibling" from that son, we can discover every son of that node. Keep in mind, that the sons of a node have to be linked in a cycle so that we can discover every node.

Subtask 5. In this solution, we use labels of size $2N$ and we make at most $2N$ wrong calls. We can run a depth-first search (DFS) from the root, and whenever we visit an unvisited node, we push it to a stack. After the DFS ends, the stack will contain the DFS-walk of the graph. For each node, we are going to hold its father and the next node to it in the DFS-walk (we are considering the DFS-walk to be cyclic, after the last element we have the first element of the stack). To walk

through the entire graph, when we are in a node, we try to go through the next node in the DFS-walk. If the function fails, we go to its father, and try again. If it doesn't work again, we go to its father and try again and do this until it works.

RMI 2021, Day 2, English Editorial

RMI Scientific Committee

Friday 17th December, 2021

Problem 1: NuM

(Proposed by *Alexandru Petrescu*.)

We will compute the number of gardens that do not satisfy the required conditions, then subtract that from $(2n)!$.

Let A_i denote the set of gardens (written as permutations of (i, j) where $i \in \{1, 2\}, j \in \{1, \ldots, n\}$) where the stones with index i are at a distance divisible by M. We are therefore interested in the size of $\bigcup_i A_i$. By the inclusion-exclusion principle this value is equal to

$$\sum_{S \subseteq 2^{\{1,\ldots,n\}}, S \neq \emptyset} (-1)^{|S|} \left| \bigcap_{i \in S} A_i \right|.$$

By symmetry, we know that $\bigcap_{i \in S} A_i = \bigcap_{i=1}^{|S|} A_i$. Thus, letting $f(i) = \cap_{j=1}^{i} A_j$ be the number of sequences where pairs with indices $1, \ldots, i$ are at distance divisible by M, this number is

$$\sum_{i=1}^{n} \binom{n}{i} (-1)^i f(i).$$

Let $d[i][j]$ be the number of ways of arranging the stones with numbers $1, \ldots, j$ at positions with remainder $1, \ldots, i$ modulo M, so that for each $x \in \{1, \ldots, j\}$ the stones with number x are at distance divisible by M. Observe that $f(i) = d[M][i](2n - 2i)!$ (we first place the stones with numbers $1, \ldots, i$ in $d[M][i]$, then place the remaining ones in $(2n - 2i)!$ ways). So the number we want to calculate is

$$\sum_{i=1}^{n} \binom{n}{i} (2n - 2i)!(-1)^i d[M][i].$$

How can we compute d? Let n_i be the number of integers in $\{1, \ldots, N\}$ with remainder i modulo M. We take the convention that numbers have remainders $1, \ldots, M$, not 0 in this editorial. By fixing the number of pairs of stones put in remainder class j modulo M, we find that for $i > 0$,

$$d[i][j] = \sum_{k=0}^{2k \leq n_i \wedge k \leq j} d[i-1][j-k] \frac{n_i!}{(n_i - 2k)!} \binom{k}{j}.$$

The coefficient $n_i!/(n_i - 2k)!$ is the number of ways of arranging $2k$ objects in n_i positions; whereas the binomial coefficient is necessary to choose which stone pairs are placed in this remainder class. Finally we note that $d[0][0] = 1, d[0][j] = 0$ for $j > 0$ (taking the convention that we consider the positions divisible by M at step M not step 0). This solution has complexity $O(N(n_1 + \ldots + n_M)) = O(N^2)$.

Problem 2: Paths

(Proposed by *Alexandra-Maria Udrișoiu*.)

Observation 1. Because the number of treats on each edge is non-negative, the optimal solution is obtained by choosing only vertices that are leaves.

Proof. Suppose the current solution contains a vertex x that is not a leaf and we replace it with any leaf y in its subtree that was not chosen in the solution. The path from the root to y will contain all the edges on the path from the root to x plus the edges on the path from x to y so the number of treats taken can only increase or stay the same.

If the number of leaves is smaller than K, then the answer is $\sum_{i=1}^{n-1} c_i$ (if all the leaves are chosen, each edge will be on at least a path). □

Definition 1. Let $treats(x, y)$ be the treats on the path from x to y where x is an ancestor of y.

Observation 2. For a fixed root, the optimal solution can be obtained by repeating the following procedure:

- Compute for every leaf x the number of treats $d(x, i)$ we can take by adding x to the current solution at step i. These edges will represent a path from an ancestor a_x of x to x - all edges on the path from the root to a_x were already visited and now we can visit the edges on the path from a_x to x.
- Add the leaf with the largest $d(x, i)$ to the solution. Let num_x be equal to d_x when x is added to the solution; i.e. $num_x = \text{treats}(a_x, x)$.

Proof. Let $leaf_i$ be the leaf added to the solution at step i.

Suppose the current solution contains $leaf_1, leaf_2, \ldots, leaf_{i-1}$, but does not contain $leaf_i$. We will show that we can always replace one of the remaining $K - i + 1$ leaves and obtain a better solution.

- No edges on the path from a_{leaf_i} to $leaf_i$ are visited. In this case, we can choose any leaf y. Because $d(y, i) \leq d(leaf_i, i)$ and the number of treats we lose by removing y cannot be greater than $d(y, i)$, the solution can only increase or remain the same.
- Otherwise, we will choose the leaf y for which the lca (let it be v) of $leaf_i$ and y is the lowest possible. The edges from the root to v will be visited no matter which one of them we choose. If we replace y with $leaf_i$, we will get $\text{treats}(v, leaf_i)$ treats and lose $\text{treats}(v, y)$. Because a_{leaf_i} is an ancestor of v (otherwise we would be in the first case), $d(leaf_i, i) = \text{treats}(a_{leaf_i}, leaf_i)$, $d(y, i) = \text{treats}(a_{leaf_i}, y)$ and we know that $d(y, i) \leq d(leaf_i, i)$, the soltion can only increase or remain the same. □

Simulating the procedure K times for each possible root will give us the time complexity $O(N^2K)$.

Definition 2. For each vertex v in the tree, let

- $down_v$ be the leaf in the subtree of v for each $\text{treats}(v, leaf)$ is maximum among all leaves in the subtree.
- up_v be the leaf that is not in the subtree of v for each $\text{treats}(v, leaf)$ is maximum among all such leaves.

Observation 3. For each leaf x, a_x will be its lowest ancestor for which $down_{ancestor}$ is not x or the root if $down_{root} = x$.

For each possible root, we can precalculate $down_v$ for all vertices using a dfs and then use the previous observation to compute num_x for all leaves. The answer will be the sum of the largest K values among all num_x. Depending on the method chosen to compute the answer, the time complexity will be $O(N^2)$ or $O(N^2 \log N)$

To further improve the solution, we need to find a way to update num_x faster instead of recalculating all of the for each root.

Observation 4. If we have calculated num_x for a root r and we want to change the root to q which is a child of r, only the paths for two leaves will change:

- If x is up_q, a_x will change from r to q, so num_x will be treats(q, x) (the edge from q to r will now be part of the path from the root to x).
- If y is $down_q$, a_y will also change from r to q, so num_y will be treats(q, y) (the edge from q to r will no longer be part of the path from the root to y).

Now, we will start with any vertex as the root r of the tree and precalculate up_v and $down_v$ for all vertices in the tree. Then, we will compute num_x for all leaves when the root is r and find the answer for r.

We will use a depth-first search to find the answer for all the other possible roots:

- When we enter vertex v, we will have to change the root from the father of v to v. Using the previous observation, we will update num_{up_x} and num_{down_x}.
- We need to find the sum of the K largest values. One way to do this is to keep the values of the current K chosen leaves in a max-heap and the other values in a min-heap. When we make an update, we will check if the maximum among the leaves that were not chosen is larger than the minimum among those that were chosen and swap these two.
- We will now calculate recursively the answer for the vertices in the subtree in v.
- After we have computed the answers for all vertices in the subtree of v, we need to change the root back from v to the father of v, so we need to update num_{up_x} and num_{down_x} again.

The time complexity will be $O(N \log N)$.

Problem 3: Weirdtree

(Proposed by *Tamio-Vesa Nakajima*.)

Brute force. This is a pretty straight-forward solution. At each step we do the following:

1. if it is a magic operation, we set in $O(1)$ the value at that position;
2. if it is a query, we add manually all the values from l to r in $O(r - l + 1) = O(n)$ worst case;
3. if it is a cut operation, we extract the leftmost maximum in $r - l + 1$ steps and decrease it by one.

Complexity: $O(NQ)$.

Subtask 2. This is another pretty straight-forward square root decomposition solution. Dividing the array in buckets of length $\sqrt{N}$, for each bucket we keep the following information:

1. Maximum value
2. Position of the maximum value
3. Sum of elements in the bucket

Then, the operations are performed in the following way:

1. if it is a magic operation, find the bucket that the element is in and, by iterating through the elements from that bucket, recompute the bucket's information.
2. if it is a query L R, manually add the elements $L \dots (\textit{beginning of some bucket } i) - 1$ and $(\textit{end of some bucket } j) + 1 \dots R$ and then iterating through bucket indices $i \dots j$, add the precomputed sums for each bucket. Obviously, if L and R are in the same bucket, manually adding the elements between L and R is enough.
3. if it is a cut operation, since $k = 1$, using a similar strategy as for the query, find the maximum and its position in the interval L R, let it be $\max_1$ at position pos_1. Then, this can be seen as the magic operation "Asuza changes the tree with index pos_1 so that it has height $\max_1$".

Complexity: $O(N + Q\sqrt{N})$. This is because we first do a precalculation to find each bucket's information and then, since each bucket is of length $\sqrt{N}$ and there are $\sqrt{N}$ different buckets, each operation will be $O(\sqrt{N})$.

Subtask 5. Here we have the important restriction that $l = 1$ and $r = n$. This means that every element is always inside the query. So, because the most "weird" operation is the one to decrease the maximal values, we can keep a set with pairs of (values, frequency at which they appear). Now, when we have to cut k times, we can extract and pop the current maximum node from the set. Now, let $\max_1$ be the current maximum value, and $\max_2$ be the second maximum value. Let fr_1 be the frequency of $\max_1$. If $\text{fr}_1(\max_1 - \max_2) \leq k$, then we know we can apply $\text{fr}_1(\max_1 - \max_2)$ and decrease all fr_1 values of $\max_1$ to the value of $\max_2$. This has the same effect with adding to the frequency of $\max_2$ (the current maximum node from the set) the value of fr_1. We still have to apply $k - \text{fr}_1(\max_1 - \max_2)$ operations. if $k < (\max_1 - \max_2)\text{fr}_1$, then we should decrease all $\max_1$ values with $[k/\text{fr}_1]$. Now we should decrease only the first $k\%\text{fr}_1$ values of $\max_1$. So, applying k cut operation in this case on $(\max_1, \text{fr}_1)$, has the effect of adding to the set 2 new nodes, $(\max_1 - [k/\text{fr}_1] - 1, k\%\text{fr}_1)$ and $(\max_1 - [k/\text{fr}_1], \text{fr}_1 - (k\%\text{fr}_1))$. Notice that by doing so, on each step we extract a number of nodes and add at most 2 back to the set. At the start we have maximum N nodes. A magic operation decreases one node's frequency and inserts at most one new node. So, we can have at most $N + 2Q$ different nodes in total. Now, each node can be popped exactly once, and we do $O(\textit{count of popped nodes steps})$, so we make at most $O(N + 2Q)$ pops from the set. Due to the complexity of insert/erase on a set, the final complexity is $O((N + 2Q)\log N)$.

The idea from subtask 5 isn't a bad one. We could guess that doing some algorithm that takes (number of different values that are affected by the current operation) steps isn't that bad. at the start, the number of different values is at most N. after a magic operation it increases by at most one. A cut operation on an interval also creates at most two new values. So, in total, we can have at most $N + 2Q$ different values over all operations. Because for each step on a cut operation, this number decreases by one, we can say that overall all the cut operations take at most $N + 2Q$ steps. Now, all we have to do is find an algorithm that can achieve (number of different values that are affected by the current operation) steps.

Subtask 6. This subtask can be solved with square root decomposition too. This time, however, for each bucket, the following information will be kept:

1. sum: The sum of elements in that bucket.
2. $\max_1$: The value of the maximum element in that bucket.
3. fr_1: How many times the $\max_1$ value appears In the bucket.
4. $\max_2$: The second-largest value that appears in that bucket.
5. lazy: The largest value that can actually appear in the bucket (the significance of this will be explained later).

Now, the operations will be processed in the following way:

1. If it is a magic operation, iterate through all elements in the given element's bucket and recompute the bucket's information. Keep in mind that the lazy plays an important role, as even if there is some tree (other than the currently modified one) that has a value higher than the lazy of the bucket, it must be seen as if its length is actually lazy, as it has been modified but not yet updated by our program.
2. If it is a query L R operation, compute the sum in a similar way to the previously described square root decomposition solution.
3. if it is a cutting operation, while there are still trees to be cut, compute the $\max_1$, fr_1 and $\max_2$ of the interval L R. Then, try to decrease all the $\max_1$ values as much as possible, either until the cutting operation ends, or until all the $\max_1$-s become $\max_2$-s. If all $\max_1$-s have become $\max_2$-s, then we know that the maximum value that we an have in the interval L R is actually $\max_2$ now. This is where the lazy for each bucket comes in; by iterating through the buckets between L and R (and obviously the prefix and suffix of the interval that must be iterated through separately, just like in the previous square root decomposition solution), we can set the lazy = $\max_2$. In the case where not all $\max_1$-s become $\max_2$-s, a similar approach is used, but some more particular cases must be taken into consideration.

The key of this solution is efficiently coding the updates on the buckets and making use of the lazy in order not to modify each element individually. This solution has complexity roughly $O(N + (N + Q)\sqrt{N})$, similarly to the previous square root decomposition solution, but now the complexity also depends on the number of distinct values in the array. The cutting operation has a good complexity in practice, as every time we compute the information in the interval in order to cut some subset of trees, the number of distinct values in the interval decreases, so the 'while' described above will not run for many steps.

Complete solution. *Variant 1.* We can also guess that probably the stuff that we kept track of at subtask 5 (that is, the first 2 maximum values and the frequency of the maximum) are still relevant. The whole problem suggests a Segment-Tree-like structure. So, right now we want to track the following things in a Segment-Tree node:

1. $\max_1$
2. $\max_2$
3. fr_1
4. sum

The composition of 2 nodes is straightforward. Magic and query operations are implemented trivially. For the cut operation, we can first take a query on the subsegment. Like in subtask 5, we can check if $\text{fr}_1(\max 1 - \max_2) \leq k$.

Case 1: $\text{fr}_1(\max_1 - \max_2) \leq k$. If yes, we should subtract from all the maximum positions $\max_1 - \max_2$. This suggests a lazy propagation, so we should also keep track of it. This propagation should only propagate on the maximum values. So, we only propagate from a node to a son if and only if they have the same maximum value ($\text{node}.\max_1 == \text{son}.\max_1$). There is, however, a problem. in the nodes which have the same $\max_2$ as the queries, we cannot just propagate lazily, because we don't know what the next different $\max_2$ will be. Since we cannot apply lazy, we have to go further down in the Segment-Tree. So, with these observations, so far the break and apply lazy conditions are as follows:

$$\textsc{Break}: (\text{node}.\text{left} > r) \vee (\text{node}.\text{right} < l) \vee (k = 0)$$
$$\textsc{Lazy}: (\text{node}.\text{left} \leq l) \wedge (r \leq \text{node}.\text{right}) \wedge (\text{node}.\max_1 - k > \text{node}.\max_2).$$

Case 2. $\text{fr}_1(\max_1 - \max_2) > k$. Like in case 1, we can still subtract from all maximum values k/fr_1. Now, we have to decrease only the first $k\%\,\text{fr}_1$ maximum values. In order to do that, on the range update operation we can also keep in mind the number of position we should subtract an "extra" 1 from the maximum. We can keep track of it by first applying the update on the left son and then on the right one. The break and lazy conditions change only slightly. In the following condition we use the *Iverson bracket*; so $[x < y]$ is 1 if $x < y$ and 0 otherwise.

$$\textsc{Break}: (\text{node}.\text{left} > r) \vee (\text{node}.\text{right} < l) \vee (k = 0 \wedge \mathit{extra} = 0)$$
$$\textsc{Lazy}: (\text{node}.\text{left} \leq l) \wedge (r \leq \text{node}.\text{right}) \wedge$$
$$(\text{node}.\max_1 - k - [\mathit{extra} > 0] > \text{node}.\max_2) \wedge (\mathit{extra} \geq \text{node}.\text{fr}_1)$$

Now the only question that remains is what the complexity of this is. Notice that in the range update operation, in the case when $\max_1 - k = \max_2$, we take some extra steps. But the following observation saves us: let us imagine the number of nodes visited by a normal Segment-tree operation (where the break and lazy conditions are not as heavily modified as ours). Now, in our case, for each extra step we take, we know that **the number of different values in the current subtree decreases** (because we merge the current maximum with the second maximum). So, overall, we take **at most the sum of the number of different values in each Segment-tree node** extra steps (which is $N \log N$ at the start). Now, a magic operation increases this sum by $\log N$ at most (it increases it by 1 for each node from the path from the root to the corresponding leaf). A query operation doesn't modify it at all, and an update range operation increases it by at most $\log N$ too (because it splits at most 1 value into 2). So, overall, this sum is at most $N \log N + Q \log N$. So, overall, we take at most $N \log N + Q \log N$ extra steps (compared to a normal Segment-tree) Recall that the normal complexity of a Segment-tree operation is $\log N$, and that by doing $O(\textit{the number of distinct values that are affected by cut})$ operations we make overall $N + 2Q$ steps, the overall complexity is $O((N + Q) \log N + N \log N + 2Q \log N) = O((N + Q) \log N)$.

Variant 2. As you might have noticed, the solution for *Subtask* 6 has a similar approach to the optimal solution, making use of square root sized buckets instead of a Segment Tree structure. Because of this, optimising that solution will also score 100 points, as it behaves very well in practice, too.