Problem A. Alice, Bob, and two arrays.

Let's write both arrays explicitly (we will get rid of the representation of arrays as segments). In $dp[x][y]$, we will store the winning status of the game if we have removed the first x characters from a and the first y characters from b.

We will iterate over the character that the first player will place and recalculate dp. This solution works in $O(NMk)$.

Now we need to eliminate the iteration over the new character. We will iterate in decreasing order, first x, then y. Notice that when we decrease y by 1, only the transitions for one color change, so we can simply maintain the count of colors that have transitions to losing positions. This allows us to obtain a solution in $O(NM)$.

We will number the indices of the arrays starting from one.

Notice that if we made a move, we will end up in position (x, y), where $a_x = b_y$. Let's calculate dp only for such positions.

We will fix the segment of identical characters that contains x, and the segment of identical characters that contains y. Let these segments be $[lx, rx]$ and $[ly, ry]$. Both of these segments consist of identical characters, which we will call t.

We want to find dp for all pairs (x, y), where x lies in the segment $[lx, rx]$, and y lies in the segment $[ly, ry]$. Notice that for all these pairs, transitions by characters other than t lead to the same positions. If among these transitions there is a losing position, then all positions of interest to us are winning, as there is a transition from all of them to a losing position.

Otherwise, if there is no transition to a losing position, then there is no point in transitioning by characters other than t. At the same time, we can freely make transitions by character t until we find ourselves in a new block (i.e., until we exit the segments $[lx, rx]$, $[ly, ry]$).

The last observation is that if we transition by a new character (i.e., not the one we are currently on), we will end up at the beginning of a block. That is, if we find ourselves in position (x', y'), then x' is at the beginning of the segment of identical characters, and y' is at the beginning of the segment of identical characters. Therefore, let's store dp only for such positions (i.e., those that are at the beginning of a block).

1. Using these observations, we will proceed as follows.
2. We will store dp only for positions at the beginnings of blocks.
3. We will calculate dp for these positions by iterating from the end to the beginning.
4. To find the answer for other (x, y), we will iterate over transitions by "other"characters.
5. If there is a transition to a losing position, then the position (x, y) is winning.
6. Otherwise, we transition by character a_x until we find ourselves in a new block, then we return to step 4.

This works in $O(n^3k)$.

Using the same optimization as for the solution in $O(NM)$, we can eliminate the iteration over the character and obtain a solution in $O(n^3)$.

A solution in $O(n^3)$ with proper optimization can already score 100 points, but the authors have a solution in $O(n^2 \log)$. Let's explain the general idea of the solution.

Consider all pairs (x, y) with color t. If we only consider transitions by character t, then all pairs will break into "diagonals,"so that when transitioning by character t, we move to the next pair on the diagonal.

We want to be able to jump along the diagonal for each character until we find ourselves in a block from which there is a transition to a losing position (a transition by another character). To do this, we will

simply store the last pair on this diagonal from which there is a transition to a losing position by another character.

This can be done implicitly using a data structure in $O(n^2 \log)$, however, such a solution has too large a constant.

Notice that the number of diagonals of interest to us is $O(nm + q)$, so we can simply compress the coordinates and write a data structure from the bottom up; such a solution will score 100 points.

Problem B. The arithmetic exercise

Subgroup 1.

Let all x_i be equal to x. In the first subgroup, it is proposed to apply changes to the zero elements of the array at each step, if possible. If there are no zero cells, we will apply changes to the very last element of the array. Then the last cell will alternate between the values $x, -x, x, ...$

Subgroup 2.

There are 2^m different ways to make a sequence of changes, as each of the m operations can be applied either to the first or the second element a. We will enumerate all possible sequences of changes and calculate the sum after their application. Among all such sums, we will take the maximum. This solution works in $O(m \cdot 2^m)$.

Subgroup 3.

We will use dynamic programming. Let $dp[pref][i][j] = true/false$ — can we process the first $pref$ changes so that $a_1 = i, a_2 = j$ holds. Note that i and j can be negative. It is clear that a_1 and a_2 cannot exceed the absolute value of the sum of the values x_i. The sum of all x_i does not exceed $10m$, so we can store values in the range from -500 to 500.

Subgroup 4.

We will improve the dynamics from the previous group. Notice that for a fixed value of a_1 on the prefix, it makes sense to recalculate only through the minimum or maximum available a_2. Therefore, we will store the explicit value of only one of the elements of the array. We will introduce $dp_{min}[pref][i]$ and $dp_{max}[pref][i]$, which store the minimum and maximum achievable value of a_2 on the prefix if $a_1 = i$.

Note.

Let's make a remark that will help us significantly advance towards the solution. It is clear that each element of the sequence x_i will enter the final sum either with a «+» sign or with a «-» sign. Let's divide all m elements into n sequences, depending on which position of the array a the change was applied to. Some of the sequences may be empty. In each sequence, the signs of the elements alternate, and the last element always has a "+"sign. Therefore, if we replace the signs with $+1$ and -1 respectively, the sum on each suffix of such arrays will be either 1 or 0. We will combine all sequences back into one and look at the sum of the signs on an arbitrary suffix. It will be in the range $[0; n]$. Now we can use this as a criterion to check the correctness of the sequence of signs.

Subgroup 5.

We will iterate over the sign for each x_i and check that the described criterion holds for each suffix. The solution works in $O(m \cdot 2^m)$.

Subgroup 6 and 7.

We will again use dynamic programming. Let $dp[i][j]$ be the maximum sum that can be obtained on suffix i, if the current balance of signs equals j. Then $dp[i][j] = max(dp[i+1][j-1] + x_i, dp[i+1][j+1] - x_i)$. This solution works in $O(nm)$.

Subgroup 8.

In this subgroup, it is proposed to act greedily, adhering to the balance criterion. For example, we can go from the end of the sequence x_i. At each step, we will try to take a new element and, if necessary, discard some element that was taken earlier. Since there are at most two different values, it makes sense

to discard only the minimum. We will maintain a queue of minima that have been taken with a "+"sign and that can still be discarded.

Subgroup 9 **and** 10**.**

There are several alternative approaches. Let's consider a few of them.

1. We will improve the solution for the previous subgroup. We will go from the end, storing the current arrangement of signs for the elements of the suffix. In the segment tree, we will maintain elements for which we can change the sign to the opposite. When we move to a new element, we try to assign it a «+» sign (if this is not possible, we change the sign of the minimum element with a «+» sign for which this can be done) and see how the sum changes as a result of this action. Similarly, we try to assign a «-» sign and see how the sum changes. Among the two options, we choose the one that yields the highest sum.

2. We will sort the sequence x_i in descending order of absolute value. We will go through the sorted values and at each step try to assign the desired sign to the element (if the number is positive — «+» else «-»), if possible. Otherwise, we will assign the sign that we are forced to assign. To understand whether we can assign a particular sign, we will maintain a segment tree, at each position of which the current balance of the corresponding suffix is stored. If we want to assign the next sign, we need to look at the minimum and maximum balance that is already achieved before it and check whether the remaining signs can be arranged so that the balance criterion is not violated.

3. Let's recall the solution for subgroups 6 and 7, which uses dynamic programming. Notice that the function is convex separately for the two parities, so we can use the slope trick.

Problem C. Dreaming is not harmful

First, let's make a general remark that will help us in solving all the subtasks. Consider the order in which the vertices will be removed in the original tree, and for simplicity, we will call it the removal order.

Let's consider an arbitrary vertex v and denote by S_v the set of vertices that come before v in the removal order. Now, let's define the optimal vertex for nullification. Note that it is pointless to nullify a vertex on the path from v to the root, as this can only worsen the position of v in the removal order. When nullifying a vertex c, which is not on the path to the root, the position of v in the removal order will decrease by the number of vertices from S_v in the subtree of vertex c.

Thus, we need to find a subtree that does not contain vertex v, with the maximum number of vertices that come before v in the removal order.

In the future, when we refer to the sum in the subtree, we will mean the number of vertices from S_v in that subtree.

- **Subtask 1. No more than two bamboos (10 points)**
  In this subtask, the tree consists of no more than two bamboos hanging from the root. In this case, the removal order can be found using the two-pointer method. The optimal vertex for nullification is the root of one of the bamboos. To answer the queries, it is sufficient to traverse the removal order and maintain the sum in both bamboos.

- **Subtask 2. Arbitrary number of bamboos (6 points)**
  The solution is similar to subtask 1. To find the removal order, we need to efficiently merge several lists, which can be done using a data structure, such as a priority queue or a set. Now, while traversing the removal order, we will maintain the sum in each bamboo and additionally track two bamboos with the maximum sum. To answer a query, we will choose the best one that does not contain the query vertex.

Now we will learn to simulate the process in the general case. We will maintain a set of the root's children in a data structure. In the next iteration, we will remove the vertex with the maximum value from it and add its children. Depending on the chosen data structure, the asymptotic complexity will be $O(n \log n)$ or $O(n^2)$. A priority queue or a set would be suitable for an efficient implementation.

- **Subtask 3. Any simulation (8 points)**
  In this subtask, it is required to write any correct simulation. To answer a query, we will iterate over the vertex for nullification and run the simulation. We will obtain a solution no worse than $O(n^4)$.

- **Subtask 4.** $O(n^2 \log n)$ **(13 points)**
  We will iterate over the vertex for nullification, run the simulation in $O(n \log n)$, and update the answer for each vertex with its number in the resulting removal order.

- **Subtask 5.** $O(n \log n + nq)$ **(11 points)**
  We will go through the removal order and maintain a set of visited vertices. To answer a query, we will calculate the sums in the subtrees and choose the maximum that does not contain the query vertex.

- **Subtask 6. Balanced binary tree (9 points)**
  We will go through the removal order and maintain the sum in each subtree. This information can be recalculated in $O(\log n)$ when considering the next vertex. Note that the optimal vertex for nullification is adjacent to the path from the query vertex to the root. To answer a query, we will consider all such vertices and obtain a solution in $O(n \log n)$.

- **Subtask 7.** $O(n \log n + nh)$ **(11 points)**
  We will act similarly to subtask 6. In addition to the sum in the subtrees, we will also maintain two children with the maximum sum for each vertex. Recalculating such information when adding a vertex and finding the subtree adjacent to the path to the root with the maximum sum can be done in $O(h)$.

- **Subtask 8. Min-heap (14 points)**
  It can be noted that in this case, each subtree adjacent to the path from the query vertex to the root either goes entirely to the query vertex in the removal order or entirely after it.

  We will take advantage of this and pre-calculate the sizes of the subtrees, then we will traverse the tree in depth. We will maintain the size of the optimal subtree adjacent to the path. When at the next vertex, we will sort its children by value. Children with larger values will entirely go in the removal order before children with smaller values. We will sequentially start a depth-first traversal from them, updating the value of the optimal subtree with the size of the children before them.

  As a result of such a traversal, we can also write down the removal order in linear time. We will obtain a solution in $O(n)$.

- **Subtask 9. Full solution** $O(n \log^2 n)$ **(18 points)**
  Similarly to subtask 6, we will go through the removal order and maintain sums in the subtrees. For this, we will use the Heavy-Light Decomposition (HLD) data structure. When considering the next vertex, we will add 1 on the path to the root. To answer a query, we will subtract ∞ on the path to the root (to avoid considering subtrees containing the query vertex), take the maximum in the tree, and then cancel the subtraction.

Problem D. Cute Subsequences

Let $i_1, \ldots, i_k$ be the indices that maximize the value for each chosen subsequence in the optimal answer. Note that the answer itself is then equal to $a_{i_1} + \ldots + a_{i_k} + max(i_1, \ldots, i_k)$.

This means that if we fix $max(i_1, \ldots, i_k) = x$, we want to maximize the sum of the $k-1$ elements of the array to the left of the x-th element. To do this, we can move left while maintaining a multiset that stores the current $k-1$ maximum elements and keeping track of their sum. Then the answer will be the maximum over all x, $x + a_x + sum_x$, where sum_x is the sum of the $k-1$ maximum elements to the left of x.

Problem E. Strong Connectivity Strikes Back

Subgroup 1

For each subset of edges, we will check if the Strongly Connected Components (SCCs) change when the edges in it are reversed. A good subset does not contain subsets for which the SCCs do not change. For each subset, we will check if it is good by enumerating all its subsets. Using Tarjan's or Kosaraju's algorithm to find SCCs, we obtain a solution in $O(3^m + 2^m \cdot m)$.

Subgroup 2

For each set, we need to check if it contains one of the "bad"subsets. We will use dynamic programming over subsets:

$$\text{is\_good}[S] = \neg\text{can\_reverse}[S] \wedge \bigwedge_{e \in S} \text{is\_good}[S \setminus \{e\}]$$

In total, we obtain a solution in $O(2^m \cdot m)$.

Subgroups 3, 4 (Acyclic Graph)

Consider some edge $e = (u, v)$. If there is no other path from u to v, e can be reversed, and the graph will remain acyclic. Thus, e cannot be in a good set.

Now suppose there is another path from u to v. Then we can restore the direction of e from the directions of the edges on this path, meaning the direction on e can be erased. The set of all such edges e is good. This set is the only maximal one.

To check for the existence of a path that does not go through an edge, we remove the edge from the graph and perform a breadth-first search. In total, we obtain a solution in $O(m^2)$.

Subgroup 5 (Hamiltonian Cycle)

We will call all edges that do not lie on the Hamiltonian cycle *direct*. Since reversing them does not violate the strong connectivity of the graph, they cannot be in a good set.

Direct edges cover certain segments of the cycle. Consider the intersection of several such segments (possibly disconnected, as the segments are cyclic). The claim is that all edges of the cycle that belong to this intersection can be reversed, and the graph will remain strongly connected.

Thus, at least one edge of the cycle in the intersection of segments must not lie in a good set. Note that if we can take another segment into our intersection and it decreases, we will obtain a stronger condition. Moreover, the "minimal"intersections that cannot be reduced further do not intersect.

We will select one edge from each minimal intersection and take the set of all edges except the direct ones and the selected ones. Why will it be good? Since the intersections are minimal, the edges in the graph form chains between vertices of degree 2. Since there should be no sinks and sources in the final graph, the directions on all edges of the chain are restored unambiguously.

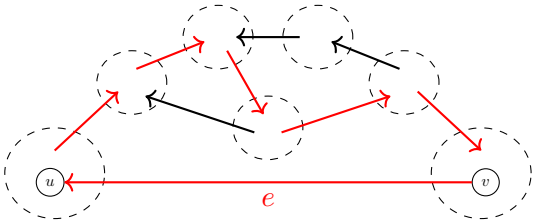
The resulting set will be a maximal good one. The number of ways is the product of the sizes of the minimal intersections.

Subgroup 6 (Strongly Connected Graph)

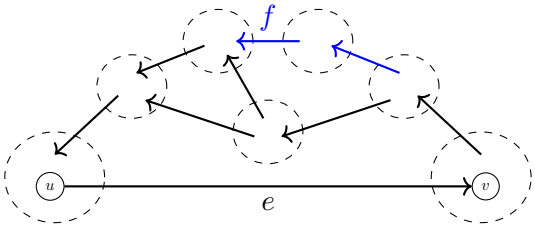
We generalize the idea of "minimal"sets that should not fully lie in a good one to the general case.

Consider an edge $e = (u, v)$. Again, if removing it keeps the graph strongly connected, e cannot be in a good set. Otherwise, reversing e would split the graph into several SCCs. We will denote the set of edges belonging to the condensation as $S(e)$ ($e \in S(e)$).

Suppose we have reversed e and want to reverse some subset of edges so that the graph remains strongly connected. We must reverse some subset $S(e)$ to create a path from u to v.



Note that if $f \in S(e)$, then $S(f) \subseteq S(e)$. If $S(f) = S(e)$ for all $f \in S(e)$, then the edges in $S(e)$ form a cycle in the condensation. We will call such $S(e)$ *necessary*—this is equivalent to the "minimal"set from the previous solution.



Consider some set of edges that can be reversed while keeping the graph strongly connected. Let e be an edge from this set with the minimal $S(e)$. Then $S(e)$ is necessary. Indeed, suppose it is not, then there exists an edge $f \in S(e)$ such that there is a path from v to u in the condensation graph that does not go through f. Then $S(f) \subset S(e)$; a contradiction.

Since $S(e)$ forms a cycle, $S(e)$ lies entirely within the considered set. Therefore, any set of edges that can be reversed without violating strong connectivity contains some necessary set.

Moreover, we can reverse $S(e)$ itself to keep the graph strongly connected. Thus, A is good $\iff$ A does not contain any necessary $S(e)$.

To achieve this, we need to not take one edge from each necessary set into A. Since they do not intersect, this can be done in any way. The maximal good set consists of all edges except those that do not affect strong connectivity and those selected from necessary sets. The number of ways is the product of the sizes of the necessary sets.

Complete Solution

We will solve the problem separately for each SCC and for the condensation graph. In the condensation graph, there may be several edges connecting two SCCs; we must not take any one of them into the good set. We will multiply the answer by their count.

The final solution works in $O(m^2)$.

Partial Score Solution

From the complete solution, an interesting fact follows: any good set lies within some maximal good set. This allows us to write the following greedy solution:

Suppose we have already chosen some good set. Now we want to understand if we can add another edge to it. Under what condition can we not do this? When it is possible to reverse our edge and some other subset of already erased edges while maintaining strong connectivity.

As mentioned above, if after reversing our edge (u, v) there is no longer a path from u to v, we need to reverse some edges from the already taken set to create this path. We can check if this is possible by

searching for a path from u to v in the graph where we have added the reversed edges for each erased edge.

If such a path does not exist, then the edge (u, v) can be erased. Why, if such a path exists, can we not do this? It is claimed that after reversing the edge (u, v) and the edges on the found path, the graph will remain strongly connected. Indeed, we obtained a cycle from this path and the reversed edge (v, u). For all edges that we reversed on this path, reachability between the ends of the edge remained, as all vertices on the cycle are reachable from each other.

By considering all edges of the graph in any order, we will construct some maximal good set.

Problem F. Best Runner

First, let's analyze the subgroups where the main idea from the general solution is not used:

Subgroup 2: The first runner will always win: he must move one lane to the left after each lane (as long as this is possible). This way, he will run the most lanes, as he will have run on the shortest lanes.

Subgroup 3: This can be solved using dynamic programming. $dp[i][j]$ (where i is the current lane of the runner, and j is the remaining time) equals the maximum number of lanes that can be run starting from such a situation.

Subgroup 5: The runner who starts on the lane with the minimum length always wins, as he can run the maximum number of times on the shortest lane.

For the other subgroups, we need to understand what the optimal route of the runner will look like. Let's say he starts on lane i.

- He must finish on the shortest lane he has been on (let's denote it as j). If he stayed on the shortest lane and ran on it until the end of the time, he would have run at least as many lanes as in any other scenario.
- He must also transition from i to j as quickly as possible, and then only run on j. This way, he will run the maximum number of lanes among all scenarios.

Let's say we have a runner starting on lane i and ending on lane j. We define the number of lanes he will run. Without loss of generality, let $i < j$. Then he will spend $a_i + a_{i+1} + \ldots + a_{j-1}$ time to reach lane j, and then he will run on j for the remaining time. Thus, he will run $j - i + \lfloor \frac{T-(a_i+a_{i+1}+\ldots+a_{j-1})}{a_j} \rfloor$ lanes (if the time from lane i to lane j does not exceed T). If we maintain prefix sums of the array a, we can compute the number of lanes in $O(1)$.

This is enough to solve **subgroup** 1: for each starting position, iterate through all ending positions and find the maximum number of lanes.

Subgroup 4: here we need to iterate only through those lanes that are strictly shorter than all lanes between the starting lane and it as ending positions. Since the lengths of the lanes do not exceed 20, for each starting position, we will consider no more than 20 ending lanes to the left and right. To quickly find the nearest lane to the left (or right) that is shorter than the current one, we can precompute them in advance.

Subgroup 6: we just need to apply a small trick. We will instead find for each ending position j the runner who will run the maximum number of lanes and finish on lane j. It can be shown that this will either be the nearest runner to the left of the lane, or the nearest runner to the right (or a runner starting on this lane, if such exists) — other runners will take more time to reach j (if j is the optimal ending position). Thus, we need to iterate through no more than 2 starting lanes for each ending lane.

Problem G. Card Flip

Let there be k cards left on the table, all of which are single-sided or flipped double-sided. Then players can only perform the first action, and thus the winner is determined unambiguously. If k is odd, the first player wins; otherwise, the second player wins.

Now suppose there are single-sided and flipped cards on the table, as well as one unflipped double-sided card, which the current player will act upon. It is noted that in such a position, the current player wins. Indeed, the current player can choose the move that will leave an even number of cards, meaning the current player can make a move that will guarantee his victory. Thus, the player who can take the last double-sided card wins.

The previous observation allows us to simplify the problem. Let x be the largest number on the front side of the double-sided cards. The player who plays with card x wins. Then:

1. We can remove all single-sided cards with numbers greater than x. Their presence does not affect which player will act with card x.
2. We can replace all double-sided cards that have a number on the back side greater than x with single-sided cards. Note that regardless of whether this card is removed or flipped, it will not affect which player will take card x. Thus, we can consider it simply as a single-sided card with the number on the front side of the original double-sided card.
3. After the two previous changes, when the turn reaches card x, no other cards will remain, meaning the current player will definitely discard it to ensure his victory. Therefore, we can replace this card with a single-sided card with the number x.

This simplification of the problem does not change the winner of the game, but the number of double-sided cards on the table decreases by at least 1.

Since the number of double-sided cards decreases after each simplification, applying this operation a finite number of times will yield a position containing only single-sided cards. Then, using the first observation, we can determine the winner of this game.

It is noted that to simplify the problem, it is necessary to iterate through all cards, i.e. perform $O(n+m)$ actions. Since after this the number of double-sided cards will decrease by at least 1, the total number of simplifications will not exceed $O(n)$. Thus, we have obtained a solution with a complexity of $O(n(n+m))$.

Let's try to speed up the previous solution. Instead of honestly performing the next simplification, we will simply search for the double-sided card that will be the last in the new game. It is noted that this card has the following property: the number on its back side is less than the current x, and among such cards, it has the largest number on the front side. Thus, by sorting the double-sided cards by number on the front side in $O(n\log(n))$, we can iterate through them in decreasing order of number, and at the moment when the condition on the number on the back side is met, we will update the value of x, thus moving to the next card. By performing this action, we will find the number x — the number on the front side of the double-sided card that will be the last in the last simplification. By looking at the parity of the number of double-sided cards with numbers not greater than x and single-sided cards with numbers less than x, we will determine the winner of the game. The complexity of this solution is $O(n\log(n)+m)$. Using count sort, we can achieve a complexity of $O(n+m)$, but this was not required in this problem.

Problem H. Order Statistics

In subtask 1, it is sufficient to model the process described in the statement in $(mn\log n)$ time by sorting the array m times and decreasing the last k elements by 1.

In subtasks 2-4, $k=1$, meaning that in each of the m operations, the maximum element in the array is decreased by 1. Let's assume that the largest element in the array after m operations is x. Then we have

$$m \le \max(a_1-x,0)+\ldots+\max(a_n-x,0)=T(x)$$

It is also easy to notice that the largest value in the array after m operations is equal to the smallest x that satisfies this condition. We will find this x using binary search. After $T(x)$ operations, all elements of the array that were greater than x will become equal to x, while the others will remain unchanged. We know that the maximum in the array will not be less than this, so the remaining $m-T(x)$ operations will

decrease the elements equal to x. We know the number of remaining operations, so we can simply decrease any $m - T(x)$ elements equal to x by 1 to obtain the final array. This solution works in $O(n \log \max a_i)$ time and passes subtask 2. To pass subtasks 3 and 4, we need to learn how to quickly compute the value of $T(x)$ for a given x. In subtask 3, there are no changes, so it is sufficient to use the precomputed prefix sums on the sorted array a, while in subtask 4, to maintain changes, we can use a segment tree, a Fenwick tree, or a Cartesian tree, resulting in a final asymptotic complexity of $O(n \log m \log n)$.

In subtask 5, $k = 2$. We will sort the array a. While a_n is greater than the maximum of $a_1, a_2, \ldots, a_{n-1}$, the problem is equivalent to the case of $k = 1$, since at each step a_n and the maximum of $a_1, a_2, \ldots, a_{n-1}$ are decreased. We will find, using binary search and the solution for $k = 1$, the moment when a_n becomes equal to the maximum of the prefix $a_1, a_2, \ldots, a_{n-1}$. This part of the solution works in $O(n \log^2 C)$ time. We will sort the array again, now $a_n = a_{n-1}$. From this moment on, the sorted array after each operation will look as follows: some unchanged prefix of the array, followed by a suffix consisting of numbers x or $x + 1$ for some x, with the suffix length being at least 2. For simplicity, let's say that the suffix length is even; the odd case is handled more or less similarly. If L is the length of the suffix, and the minimum value in the suffix is greater than the maximum in the prefix, then in $L/2$ operations, all elements in the suffix will be decreased by 1. At the moment when x becomes equal to the maximum in the prefix, the length of the suffix will increase by 1. The total number of suffix expansions will not exceed n. If we handle all expansions carefully, this part of the solution will work in $O(n)$ time. Thus, we obtain a solution in $O(n \log^2 C)$.

In subtask 6, $m \leq 10^6$. We will learn to process one operation in $O(\log n)$ time while keeping the array sorted. Let's assume the array is sorted before the operation, and the number x is at position $n - k + 1$. Let the number of elements in the array greater than x be k_1. Then in this operation, all elements of the array greater than x, as well as $k - k_1$ elements equal to x, will be decreased. To maintain the sorted order of the array, we need to choose $k - k_1$ of the elements equal to x that are the leftmost in the array. We can find the number of elements greater than x, the first element not less than x, and decrease all elements in the segment by 1 using a segment tree in $O(\log n)$ time. In total, we obtain a solution in $O(m \log n)$ time.

In the remaining subtasks, the main idea of the complete solution is used in one way or another.

The main idea: let's imagine that in each operation we decrease not the k largest elements, but any k elements of our choice. We will also assume that the operations are applied in such a way as to maintain the sorted order of the array (note that this requirement does not restrict the operations if we consider them as operations on an unordered set of elements of the array). Let b be the array obtained by applying the operation from the statement m times, and c be the array obtained by applying any sequence of m operations, where each operation decreases k arbitrary elements by 1. We will understand how the array b, which we want to find, stands out among all possible arrays c. Let's assume that the arrays b and c are sorted. Then it is claimed that $b_1 + \ldots + b_i \geq c_1 + \ldots + c_i$ for all i from 1 to n, meaning that the operations from the statement "maximize"the resulting array in some sense. It also follows that the array $b_1, b_2 \ldots, b_n$ is the lexicographically maximum array that can be obtained by such operations. Proof: consider a sequence of operations that leads to the maximum possible sum $c_1 + \ldots + c_t$. Let's look at the last bad operation (the one that decreased not the k largest elements by 1). Then if elements i, j are such that before this operation $a_i < a_j$ ($i < j$, since the operations preserve the order of elements), and the i-th element was decreased by 1, while the j-th was not. Let's assume that after applying all operations, the array became equal to $b_1, b_2 \ldots, b_n$. If $b_i < b_j$, swapping the i-th and j-th elements in the considered bad operation will yield a final value that is at least $b_1 + b_2 + \ldots + b_t$. If $b_i = b_j$, then after the bad operation, there was at least one operation that decreased a_j and did not decrease a_i. Swapping a_i and a_j in both operations will yield the same array b, but will increase the sum of the elements that were decreased by 1 in the bad operation. The number of operations and the sum of elements are finite, so this process will eventually end, and we will obtain a sequence consisting of good operations that maximizes the sum $c_1 + \ldots + c_t$.

We have understood that b maximizes the prefix sums among all arrays c obtained by m operations of subtracting 1 from k elements. The next task allows us to easily describe all suitable arrays c, for which we will search for this maximum.

Let's recall a classic problem: how to check if in the array $d_1, d_2, \ldots, d_n \geq 0$ we can decrease k different (but not necessarily distinct in value) elements m times, so that all elements of the array remain non-negative. The answer: this is possible if and only if $\min(d_1, m) + \ldots + \min(d_n, m) \geq mk$. The proof is left as an exercise.

Using the described solution to the classic problem, we can check whether it is possible to obtain from a an array c after performing m operations such that $c_i \geq b_i$ for all i. Then, by performing binary search sequentially on b_1, then on b_2 with fixed b_1, and so on, we will find the maximum lexicographically array that can be obtained from a in m operations, which will indeed be the answer. Depending on the efficiency of the implementation, we can achieve a solution in $O(n^2 \log m)$, $O(n^2)$, $O(n \log m)$, or $O(n)$ time, passing subtasks 7 and possibly 8.

We will learn to calculate the maximum possible prefix sum $a_1 + a_2 + \ldots + a_i$ that can be obtained after m operations. Let's assume that we have decreased the element a_i to x. Then the elements $a_1, a_2, \ldots, a_{i-1}$ must be decreased at least to x, and the elements $a_{i+1}, \ldots, a_n$ can be decreased no lower than x, because the order of elements in the array must be preserved. Let's consider two quantities $f(x)$ — the minimum number by which we must decrease the elements in the prefix up to the i-th, and $g(x)$ — the maximum number by which we can decrease the elements in the suffix starting from the $(i+1)$-th, in order to maintain the non-decreasing order of elements in the array. Since in total, after m operations, all elements will be decreased by mk, the total decrease in the prefix will be at least $\max(f(x), mk - g(x))$. It is easy to see that this estimate is achievable, so in the case when we decrease a_i to x, the maximum possible prefix sum is $a_1 + a_2 + \ldots + a_i - \max(f(x), mk - g(x))$. We can find the maximum of such a function over x using binary search, as the function $f(x)$ is non-increasing, while the function $mk - g(x)$ is non-decreasing. Without queries to change the functions $f(x)$ and $g(x)$, we can compute them using prefix sums on the sorted array, which passes subtasks 9 and 10. To handle update queries, we can use a segment tree, a Fenwick tree, or a Cartesian tree, which passes subtask 11 and, depending on the efficiency of the implementation, subtask 12. The time complexity of the solution is $O(n \log m \log n$.