Problem Analysis

The 2026 ICPC Asia Pacific Championship

This is an analysis of some possible ways to solve the problems of The 2026 ICPC Asia Pacific Championship. Since the purpose of this analysis is mainly to give the general idea to solve each problem, we left several (implementation) details in the discussion for reader's exercise. If some of the terminology or algorithms mentioned below are not familiar to you, your favorite search engine should be able to help. If you find an error, please send an e-mail to icpc-apac-2026-judges@googlegroups.com about it.

Problem Title		Problem Author	Analysis Author
A	Compare Suffixes	Mitsuru Kusumoto	Mitsuru Kusumoto
B	Subtree Removal Game	Mitsuru Kusumoto	Shogo Murai
C	Upside Down Dijkstra	Pikatan Arya Bramajati	Pikatan Arya Bramajati
D	Christmas Tree Un-decoration	Albert Yulius Ramahalim	Albert Yulius Ramahalim
E	Parallel Sums	Pikatan Arya Bramajati	Pikatan Arya Bramajati
F	Minesweeper String	Brian Tsai	Brian Tsai
G	Extra Transition	Rama Aryasuta Pangestu	Shogo Murai
H	Reflect Sort	Pikatan Arya Bramajati	Pikatan Arya Bramajati
I	Growth Factor	Pikatan Arya Bramajati	Gyojun Youn
J	Worldwide Playlist	Albert Yulius Ramahalim	Albert Yulius Ramahalim
K	Time Display Stickers	Ashar Fuadi	Jonathan Irvin Gunawan
L	Onion	Mitsuru Kusumoto	Ke-Wei Huang
M	Deformed Balance	Mitsuru Kusumoto	Mitsuru Kusumoto

The contest judges would also like to thank the following for their valuable feedback to the tasks: Masaki Nishimoto, Ping-Hsuan Lin, Poyu Chou, Sian-Yang Tseng, Riku Kawasaki, and Yui Hosaka.

A Compare Suffixes

Step 1

We maintain the order of suffixes (also known as a suffix array) from the end of S. At the beginning, we determine whether $S(n) > S(n-1)$ or $S(n) < S(n-1)$. This can be done by a single query.

Suppose that we have now determined the order of k suffixes $S(n-k+1), S(n-k+2), \ldots, S(n)$:

$$S(p_{n-k+1}) < S(p_2) < \ldots < S(p_n)$$

for a sequence $(p_{n-k+1}, \ldots, p_n)$, which is a permutation of $(n-k+1, \ldots, n)$.

For each $i = n-k+1, \ldots, n-1$, consider the following question: can S_{p_i} and $S_{p_{i+1}}$ be the same character? If they are the same, the order of $S(p_i)$ and $S(p_{i+1})$ must be the same as that of $S(p_i+1)$ and $S(p_{i+1}+1)$. Since $S(p_i) < S(p_{i+1})$, this is equivalent to $S(p_i+1) < S(p_{i+1}+1)$. Thus, if $S(p_i+1) > S(p_{i+1}+1)$, then S_{p_i} and $S_{p_{i+1}}$ must be different characters. Since S consists of only lowercase letters, the number of such indices i is at most 25. We call such an index i a *splitting position*.

Step 2

We want to know which position the suffix $S(n-k)$ falls into the current ordering $S(p_1) < \ldots < S(p_k)$, using fewer queries. To do so, we split this ordering by splitting positions. By performing a kind of binary search, we can determine which sub-ordering $S(n-k)$ falls into. However, this can incur too many queries.

Assume that $S(p_x) < S(n-k) < S(p_{x+1})$ for some index x. We are interested in, after adding $S(n-k)$ in the ordering, a new splitting position happens to appear or not. Since n is much larger than 25, this does not happen in most cases.

If it does not, $S(p_x+1) < S(n-k+1)$ must hold. For each sub-ordering, say, $S(p_a) < \ldots < S(p_b)$, we can find the largest index $y \in [a, b]$ such that $S(p_y+1) < S(n-k+1)$. Let us say y is a *representative index* of that sub-ordering.

In the binary search we mentioned above, we use representative indices. After finding a representative index y such that $S(p_y) < S(n-k) < S(p_z)$ (where z is the representative index of next sub-ordering), we query to determine whether $S(n-k) < S(p_{y+1})$ holds. If it holds, we successfully determine the position of $S(n-k)$ with at most $\lceil \log_2(26) \rceil + 1 = 6$ queries.

If $S(n-k) < S(p_{y+1})$ does not hold, we simply do binary search over the current ordering $S(p_1) < \ldots < S(p_k)$ to determine the position of $S(n-k)$. This requires at most $\lceil \log_2(n) \rceil$ queries, and this case occurs at most 25 times.

Iterating $k = n-1$ down to 1 will yield the answer. Overall, the total query count is bounded by $6n + 25 \log_2(n) \leq 6260$. In general, for the number of the types of letters σ, the query count is $n(\log_2(\sigma) + 1) + \sigma \log_2(n)$. Computational time is $O(n^2)$.

B Subtree Removal Game

We can perform a binary search for the solution. For a fixed value v, let's consider the following game:

$Game(v)$: A leaf node holding an integer $i \leq v$ has label "F", and holding $i > v$ has label "S". If a node with "F" remains at the end of the game, the first player (minimizer) wins. If "S" remains, the second (maximizer) wins.

If we determine the value such that the winner of $Game(v)$ is the second player while that of $Game(v+1)$ is the first player, we can conclude that the answer is $v+1$. So, let's consider solving $Game(v)$ for some v. Note that we only need to take labels "F" and "S" into account now. In what follows, we denote the two players as "F" and "S" for simplicity ("F" for the first and "S" for the second).

For a subtree rooted at node x and a player p ("F" or "S"), define $Win(x, p)$ as the winner of a game that

- starts with player p, and
- starts with subtree x.

For a node x, let's define $D(x)$ as follows:

- If x is a leaf node labeled "F", $D(x) = 1$.
- If x is a leaf node labeled "S", $D(x) = -1$.
- If x is a non-leaf, let $S(x)$ is the sum of $D(y)$ over its child nodes y. Then, $D(x)$ is the *sign* of $S(x)$; that is, $D(x) = 1$ if $S(x) > 0$, $D(x) = 0$ if $S(x) = 0$, and $D(x) = -1$ if $S(x) < 0$.

We show the following argument by induction on the number of leaf nodes:

- If $D(x) > 0$, $Win(x, F) = Win(x, S) = F$,
- If $D(x) = 0$, $Win(x, F) = F$ and $Win(x, S) = S$, and
- If $D(x) < 0$, $Win(x, F) = Win(x, S) = S$.

Without loss of generality, we can assume that the root node has at least 2 children, or the tree consists only of the root node. For the latter case, the argument is trivial. We consider the former case below.

When $D(x) > 0$. If the root node x has at least one child y with $D(y) \leq 0$, the first player can choose y. Otherwise, the first player can select any (immediate) child of x. In either case, $D(x) > 0$ holds after the operation.

When $D(x) = 0$. We can find a path from $x_0 = x$ to x_k where

- $D(x_0) = ... = D(x_k) = 0$, and
- x_k has children y^+ and y^- with $D(y^+) > 0$ and $D(y^-) < 0$.

Then, the first player can choose y^-. After the operation $D(x) > 0$ holds.

When $D(x) < 0$. We note that the value of D changes by at most 1 after the operation. Therefore, $D(x) \leq 0$ holds after the operation.

From this argument, we can compute $Win(x, F)$ by recursion in $O(n)$. Overall time complexity is $O(n \log n)$.

C Upside Down Dijkstra

Since the wrong Dijkstra's algorithm always takes the current maximum distance, that means whenever a new vertex is processed, if it is adjacent to at least one unprocessed vertex, the next processed vertex must be one of those adjacent to the current vertex. If all of its adjacent vertices are already processed, then this vertex will not generate new entries to the heap and the next maximum distance comes from the next unprocessed "child" of its closest "ancestor". If observed more closely, notice that the order of vertices processed looks like an arbitrary naive DFS that does not go through the same vertices more than once. Each vertex in the DFS only unrecurse back to its source parent when all of its neighboring vertices are already processed.

The solution is to check whether or not S follows a possible naive DFS order, using a stack. A vertex can only be pushed into the stack if it is adjacent to the current top. The top of the stack can only be popped when it has already had all of its neighbors processed. The edge weights can be calculated based on the order of vertices pushed into the stack interacting with the current state of the stack.

Each iteration might have a worst case of $O(n)$ time complexity because of the need to iterate edges, so some non-trivial optimizations might be needed to make sure that the total time complexity is still within $O(n + m)$.

D Christmas Tree Un-decoration

First, consider the problem without updates. We can perform a DP on the tree. Define $\mathrm{dp}(u)$ as the minimum number of operations to remove all ornaments in the subtree rooted at vertex u. The transition is

$$\mathrm{dp}(u) = \max\left(a_u, \sum_{c \text{ child of } u} \mathrm{dp}(c)\right).$$

Thus, the answer can be computed in $O(n)$ time.

To handle updates, we use heavy-light decomposition. Define:

- b_u: the sum of $\mathrm{dp}(c)$ for all light children c of u.
- $h(u)$: the heavy child of u.

The transition becomes $\mathrm{dp}(u) = \max(a_u, b_u + \mathrm{dp}(h(u)))$.

Next, consider a chain in the heavy-light decomposition from its topmost vertex c_1 to a leaf c_k: $(c_1, c_2, \ldots, c_k)$. We have

$$\begin{aligned}\mathrm{dp}(c_1) &= \max(a_{c_1}, b_{c_1} + \mathrm{dp}(c_2)) \\ &= \max(a_{c_1}, b_{c_1} + a_{c_2}, b_{c_1} + b_{c_2} + \mathrm{dp}(c_3)) \\ &\vdots \\ &= \max_{i=1}^{k} \left(a_{c_i} + \sum_{j=1}^{i-1} b_{c_j} \right).\end{aligned}$$

Note that the only important value we need to compute is $\mathrm{dp}(c_1)$, since it either becomes the final answer (if $c_1 = 1$) or contributes to $b_{p_{c_1}}$ (if $c_1 \neq 1$). Therefore, we can maintain each term inside the $\max$ function above for each chain using a lazy segment tree.

To update the number of ornaments in vertex u_i to x_i, we update a_{u_i} in u_i's chain, then move up while updating the values of b for the chains' roots' parents. In this way, an update can be performed in $O(\log^2 n)$ time.

Therefore, the problem has been solved in $O(n \log n + q \log^2 n)$ time.

E Parallel Sums

To simplify, let's change the indexing of the sequences into 0-based, so now it's $A = (a_0, a_1, \ldots, a_{n-1})$ and $s_0, s_1, \ldots, s_{n-m}$ such that $s_i = a_i + a_{i+1} + \ldots + a_{i+m-1}$.

Consider the relationships between elements of A. Notice that $a_{i+m} - a_i = s_{i+1} - s_i$. Since the values of s_i are constant, then each element a_i is just $a_{i \bmod m}$ plus some constant value that can be obtained from several values of s_i.

From the values of s_i, construct a sequence $w_0, w_1, \ldots, w_{n-1}$ which indicates that $a_i = a_{i \bmod m} + w_i$. Then with this, satisfying the $n - m + 1$ requirements of s_i is equivalent to satisfying the following:

- $a_0 + a_1 + \ldots + a_{m-1} = s_0$
- $a_i = a_{i \bmod m} + w_i$ for all i.

Suppose we want to naively solve for a query with a pair (l, r). If $r - l + 1 < m$, then the answer can be arbitrarily small. Otherwise, we group each of $a_l, a_{l+1}, \ldots, a_r$ based on their indices modulo m. For elements at indices i with the same value of $i \bmod m$, only the maximum value of w_i matters. We calculate the maximum value of w_i for each group.

Suppose these maximum values are $y_0, y_1, \ldots, y_{m-1}$ for each group of indices modulo m. Then we need to find $a_0, a_1, \ldots, a_m$ such that $a_0 + a_1 + \ldots + a_{m-1} = s_0$ and the value of $\max(a_0 + y_0, a_1 + y_1, \ldots, a_{m-1} + y_{m-1})$ is minimized. It turns out that the minimum value of that is equal to $\left\lceil \frac{s_0 + y_0 + y_1 + \ldots + y_{m-1}}{m} \right\rceil$.

Now we need to be able to calculate that quickly. We have solutions for two cases, cases with $m \leq \sqrt{n}$ and cases with $m > \sqrt{n}$.

If $m \leq \sqrt{n}$, before doing the queries, we can build a sparse table for the values of w_i for each group of $i \bmod m$. Then, for each query, we do a range maximum query for each sparse table to get the values of y_i, then we sum them.

If $m > \sqrt{n}$, then each group has a small number of elements. For each possible pair (p_0, p_1) ($0 \leq p_0 \leq p_1 < \lceil \frac{n}{m} \rceil$):

1. We iterate each group and calculate the maximum value from the p_0-th to the p_1-th element of w_i of that group.
2. We make a prefix sum of length m from those maximum values.

Then, each query is just an $O(1)$ number of range sum queries in a few of those prefix sums.

Time complexity: $O((n+q)\sqrt{n})$

F Minesweeper String

For any given width w, we conceptually begin by assuming that each digit x (the number of mines in each cell) contributes $4x$ to the total sum. The actual total may decrease due to the following four situations:

1. Two digits x, y become vertically adjacent.

   If the positions of two digits differ by exactly w, they appear in two consecutive rows. In this case, the total decreases by $x + y$.

2. A digit x lies on the leftmost or rightmost column.

   If a digit's position p (0-based) satisfies

   $$p \bmod w = 0 \quad \text{or} \quad p \bmod w = w - 1,$$

   then the digit lies on the boundary of the grid, contributing a reduction of x.

   A special case might happen for the last digit, which will always decrease the actual total since the row is truncated to its right, even if it is not in the rightmost column.

3. A digit x lies in the first or last row.

   If a digit appears in the first or last row under width w, the total decreases by x. (Note that this may be counted twice.)

4. Two digits x, y become horizontally adjacent.

   If two digits are adjacent in the string, the total decreases by $x + y$ unless they are split across rows.

To compute all cases:

Case 1: Consider calculating the total reduction for each w. For a digit x at position p, it contributes a reduction of x if the digit at position $p - w$ is non-zero, or if the digit at position $p + w$ is non-zero (both can occur, resulting in a contribution of $2x$). Therefore, we can define two polynomials:

$$P(x) = \sum_{i=0}^{n-1} S_i \cdot x^i, \quad Q(x) = \sum_{i=0}^{n-1} [\![S_{n-i-1} \neq 0]\!] \cdot x^i,$$

The total reduction for w will be $[x^{n-1-w}](P(x)Q(x)) + [x^{n-1+w}](P(x)Q(x))$.

The product of P and Q can be computed using a single convolution in $O(n \log n)$.

Case 2: A digit at position p lies on a boundary column precisely for all divisors w of p and $p+1$. Thus, for each digit, we decrement the totals for all divisors of p and $p+1$. The total contribution for all w can be computed in $O(n \log n)$. Be careful with the special case of the last digit.

Case 3: A digit lies in the first or last row exactly when w falls within certain continuous intervals. These intervals can be accumulated using a difference array in linear time.

Case 4: For each adjacent pair of digits x, y in the string, initially decrease the total by $x + y$. Then, if their positions in the string are p and $p+1$, increment all divisors w of $p+1$. The total contribution for all w can be computed in $O(n \log n)$, and it is possible to merge this with Case 2.

The overall time complexity is $O(n \log n)$.

It is possible to modify the problem back to eight-direction minesweeper, but that might not be interesting.

G Extra Transition

We write a path from vertex a to b through zero or more vertex by $a \to^* b$ (which is equivalent to $a \to^* b$).

A graph G is a series-parallel graph with two terminals s and t if and only if G can be reduced to the graph with 2 vertices s and t with only one edge connecting them by repeatedly applying following operations:

- *series operation*: if a vertex v has exactly two incident edges to vertices x and y ($x \neq y$), remove v and its incident edges, then add an edge between x and y.
- *parallel operation*: if there are multiple edges between two vertices x and y, remove all of them, then add one edge between x and y.

Note that parallel edges are allowed during the operations.

We can see that a graph G is well-designed if and only if it is series-parallel. We show this later.

Let G be a series-parallel graph. Now we discuss the condition when addition of an edge between u and v keeps G series-parallel. In the definition of series operation, we may allow contracting several paths $x - v_1 - \cdots - v_k - y$ to $x - y$ (if all of $v_1, \ldots, v_k$ have degree of 2). Then, we let all of edges before contraction ($x - v_1$ and so on) be ones that have been initially existed or have been added in a parallel operation. Here, addition of an edge between u and v keeps G series-parallel if and only if there is an extended series operation involving u and v while contracting G into an edge.

- If such series operation exists, we can easily construct a sequence of operations for G added with an edge between u and v.

- Suppose G added with an edge between u and v is series-parallel. During contraction, vertices u and v can be removed (by series operations) only after the (initially added) edge between u and v is removed by a parallel operation. This means that, after applying operations, the original graph (before edge addition) could have a path $u - \cdots - v$ where any intermediate vertices have degree of 2.

We can check whether a graph is series-parallel in $O(m \log n)$ time. Along this check, we can also compute $\sum_{(i,j)\in S} w_i w_j$.

Proof of equivalence between well-designed and series-parallel graphs

First, we show that series-parallel graphs are well-designed by induction on the number of edges in G.

We have to first check that, if G is series-parallel, for any vertex v, there is a simple path from s to t through v. Since series-parallel graphs do not have isolated vertices, it suffices to show that, for any *edge* e, there is a simple path from s to t through e. This argument can be easily shown by induction.

If G consists of 2 vertices and an edge connecting them, G is clearly well-designed.

Otherwise, let G' be the graph obtained by applying a series operation to G. G' is also series-parallel and G' has strictly fewer edges than G, so G' should be well-designed. Let v be the vertex removed by the operation, and x and y be the vertices which are adjacent to v in G. For vertices p and q ($p \neq v, q \neq v$), if there are paths $s \to^* p \to^* q \to^* t$ and $sto^*q \to^* p \to^* t$ in G, we have paths of the same form in G' (possibly $x \to v \to y$ is contracted to $x \to y$). Also, suppose that, for a vertex p, there are paths $s \to^* p \to^* v \to^* t$ and $s \to^K v \to^* p \to^* t$ in G. Since v is adjacent only to x and y, we will have a path of form $s \to^* p \to^* x \to v \to y \to^* t$ or $s \to^* p \to^* y \to v \to x \to^* t$. The same applies to the latter path. Thus we will have paths $s \to^* p \to^* x \to^* t$ and $s \to^* x \to^* p \to^* t$, or $s \to^* p \to^* y \to^* t$ and $s \to^* y \to^* p \to^* t$. Either case contradicts to the fact that G' is well-designed. The same holds if G' is obtained by a parallel operation.

Now we show the inverse: if a graph G is well-designed, then it is series-parallel. If G is well-designed and G' can be obtained from G by applying a series or parallel operation, it is easy to see that G' is also well-designed. Thus, it suffices to show that an operation is applicable to G (unless G has only two vertices s and t and an edge connecting them). If G has a parallel edge, a parallel operation can be immediately applied. From now on, we assume that G is well-designed and has no parallel edge, and we show a series operation is applicable to G (that is, there is a vertex $v \neq s, t$ with degree of 2).

For two distinct vertices a and b (possibly s or t), if there exists a simple path $s \to^* a \to^* b \to^* t$, we write $a \Rightarrow b$. Note that $a \Rightarrow b$ and $b \Rightarrow a$ does not hold at the same time (because G is well-designed). Also this relationship is transitive: if $a \Rightarrow b$ and $b \Rightarrow c$, then $a \Rightarrow c$.

We can show that, for each edge e, there is a simple path from s to t through e. Thus, for any neighbor w of v, exactly one of $v \Rightarrow w$ and $w \Rightarrow v$ holds. Thus we can add directions to each (undirected) edge $\{u, v\}$: from u to v if $u \Rightarrow v$, or from v to u if $v \Rightarrow u$. TWe can see that this directed graph G' is a directed acyclic graph (DAG) with the source s and the terminal t. Then let T the *dominator tree* of G'. Also we let W be the connected component containing s of the graph $(V(G), E(G) \cap E(T))$. All edges adjacent to s are in T, so W is non-empty. Thus W will have at least one leaf.

If the only leaf in W is t, let $s = v_0, v_1, \ldots, v_{k-1}, v_k = t$ be the path from s to t in W. If v_i ($0 \leq i \leq k-1$)

has a neighbor $v'_{i+1} \neq v_{i+1}$ with $v_i \Rightarrow v'_{i+1}$ (note that v'_{i+1} cannot be one of $v_{i+2}, \ldots, v_k$, either), v'_{i+1} will be in W and thus W will have other leaf than t. From this, it follows that G is a path graph, which is clearly series-parallel.

Otherwise, let v be one of the *uppermost* leaf in W: there is no leaf v' in W such that $v' \Rightarrow v$. We show that the degree of v is two, thus a series operation is applicable.

Suppose there are two distinct neighbors w_1, w_2 of v with $v \Rightarrow w_1$ and $v \Rightarrow w_2$. Without loss of generality, we can assume $w_2 \Rightarrow w_1$ does not hold. Let p_1 be the parent of w_1 in T. Since p_1 should be an ancestor of v in T, we have a path $s \to^* p_1 \to^* w_1 \to v \to w_2 \to^* t$. This is simple because $w_2 \to^* t$ cannot contain w_1. We also have a path $s \to^* p_1 \to^* v \to w_1 \to^* t$, which contradicts the fact that G is well-designed. Therefore, there exists only one vertex w with $v \Rightarrow w$.

Since v is a leaf in W, its parent p is adjacent to v and $p \Rightarrow v$. Suppose another neighbor q of v satisfies $q \Rightarrow p$. By the definition of W, all path from s to v passes through p. Similarly, all path from s to q passes through p. We arbitrarily take one such path: $s, \ldots, p, m_1, \ldots, m_k = q$. Since p is in W and we have an edge between p and m_1, we have a leaf r in W as an descendant of m_1. r cannot be one of $m_1, \ldots, m_k$ because v is uppermost. Suppose the lowest ancestor of r in $m_1, \ldots, m_k$ is m_j. Then we have a path $s \to^* p \to v \to q \to^* m_j \to^* r \to^* t$ (we take the unique path $m_j \to^* r$ in W). This path is simple:

- r cannot be v because the parent of v is p.
- $m_j \to^* r$ cannot contain v; otherwise v cannot be a leaf.
- $r \to^* t$ cannot contain v because v is uppermost.

We also have a path $s \to^* p \to^* q \to v \to^* t$, which contradicts the fact that G is well-designed. Therefore, there exists only one vertex p with $p \Rightarrow v$.

Combining these two arguments, it follows that the degree of v is 2.

H Reflect Sort

Consider the attributes of the sequence that can't change no matter what operations you do. For each i ($1 \leq i \leq n-1$), the value of $|a_i - a_{i+1}|$ will always be the same.

That means, if the value of a_1 is fixed, you are forced to make $a_{i+1} = a_i + |a_i - a_{i+1}|$ for all $1 \leq i \leq n-1$. This makes the final value of a_n to be $a_1 + |a_1 - a_2| + |a_2 - a_3| + \ldots + |a_{n-1} - a_n|$. So to minimize a_n, you just need to minimize a_1.

Since a_1 is the minimum value in the end, you just need to make a_1 to be the smallest possible positive integer without worrying about the other elements.

Consider doing a prefix operation on $\{1, 2, \ldots, x\}$, and then doing another prefix operation on $\{1, 2, \ldots, x-1\}$. That increases the value of a_1 by $2 \times (a_{x+1} - a_x)$. Furthermore, for each value of $a_{x+1} - a_x$, you can always flip its sign without changing the value of a_1 by doing suffix operations. That means, the value of a_1 can be increased or decreased any number of times by any value $2 \times |a_x - a_{x+1}|$.

From number theory, it can be obtained that if the greatest common divisor of all values of $2 \times |a_x - a_{x+1}|$ is d, then the value of a_1 can be increased or decreased by any multiple of d. You want a_1 to have the minimum possible positive value. So if $a_1 \bmod d = 0$, you should make it to be d. Else, you should make it to be $a_1 \bmod d$.

Time complexity: $O(n \log \max a_i)$

I Growth Factor

The problem asks to find the number of integer sequences $(b_1, \ldots, b_n)$ such that $1 \le b_i \le a_i$ and $b_i \mid b_{i+1}$.

First, observe that the condition $b_i \mid b_{i+1}$ inherently implies that $b_i \le b_{i+1}$. Because the sequence b must be non-decreasing, any valid sequence must also satisfy $b_i \le a_{i+1}$, $b_i \le a_{i+2}$, and so on. Therefore, we can replace each a_i with $\min_{i \le j \le n} a_j$ to make the sequence a non-decreasing without altering the set of valid sequences for b. Let $M := \max a_i = a_n$.

Let $\mathrm{left}(v)$ be the minimum index i such that $a_i \ge v$. Let $f_v(l)$ be the number of valid prefixes of length l ending exactly with the value v.

If the prefix consists entirely of the value v up to index l, this is only valid if $v \le a_1$. Otherwise, the value v must be preceded by some proper factor u of v. Suppose the transition from u to v happens such that $b_j = u$ and $b_{j+1} = v$. For this to be valid, we must have $j \ge \max\{1, \mathrm{left}(v) - 1, \mathrm{left}(u)\}$. This imposes a lower bound on the transition index, and let this valid threshold be $L(u, v)$.

The transition can be written as:

$$f_v(l) = [v \le a_1] + \sum_{\substack{u \mid v \\ u < v}} \sum_{L(u,v) \le j < l} f_u(j)$$

Computing this directly would be too slow. However, we can represent $f_v(l)$ as a linear combination of binomial coefficients:

$$f_v(l) = \begin{cases} \displaystyle\sum_{k \ge 0} D_v[k] \binom{l}{k} & \text{if } v \le a_l \\ 0 & \text{otherwise} \end{cases}$$

This representation is extremely powerful because it allows us to utilize Fermat's identity. Specifically, for $L \le l - 1$, we have:

$$\sum_{j=L}^{l-1} \binom{j}{k} = \binom{l}{k+1} - \binom{L}{k+1}$$

Under the condition $v \le a_l$, substituting our polynomial representation into the transition yields:

$$\begin{aligned} \sum_{L(u,v) \le j < l} f_u(j) &= \sum_{L(u,v) \le j < l} [u \le a_j] \sum_{k \ge 0} D_u[k] \binom{j}{k} \\ &= \sum_{k \ge 0} D_u[k] \left(\binom{l}{k+1} - \binom{L(u,v)}{k+1} \right) \end{aligned}$$

From this, we can directly extract the updates for the coefficients D_v. For each proper factor u of v and for each k:

- Add $D_u[k]$ to $D_v[k+1]$.
- Subtract $D_u[k]\binom{L(u,v)}{k+1}$ from $D_v[0]$, since it acts as a constant term.

Because b_i must be a proper factor of b_{i+1} at each step, the value strictly increases by at least a factor of 2. Therefore, the maximum number of transitions is bounded by $\lfloor \log_2(M) \rfloor \leq 18$. This means we only need to maintain about 18 non-zero coefficients for each value v.

For each value u from 1 to M, we iterate over all its multiples v, and update the $\mathcal{O}(\log M)$ coefficients. Finally, the answer is the sum of $f_v(n)$ for all $1 \leq v \leq a_n$. The total time complexity will be $\mathcal{O}\left(n + M \log^2 M\right)$.

J Worldwide Playlist

Observe that the number of skips is equal to the number of songs played minus n.

To listen to the songs $b_1, \ldots, b_n$ in order, the total number of songs played is equal to $\mathrm{pos}_a(b_n) + kn$, where $\mathrm{pos}_a(x)$ denotes the position of song x in the permutation a, and k is the number of *wraps* around the playlist: the number of times a_1 is played after a_n.

We can see that a wrap occurs whenever the next desired song appears *earlier* in the permutation a than the current song. In other words, k is equal to the number of indices i such that $\mathrm{pos}_a(b_{i+1}) < \mathrm{pos}_a(b_i)$.

Next, let us handle updates. We maintain an array storing $\mathrm{pos}_a(i)$ for $1 \leq i \leq n$.

- To handle updates with $c = 2$, since swapping b_x and b_y affects only the comparisons involving these two positions, we can update the corresponding contributions to k in $O(1)$.
- To handle updates with $c = 1$, note that swapping a_x and a_y changes $\mathrm{pos}_a(i)$ for two values of i. We can recompute their contributions to k in $O(1)$ as well.

Therefore, the total time complexity is $O(n + q)$.

K Time Display Stickers

We can separate time displays into two different types:

1. time displays whose hour is between 00 and 09 (let's say these are time displays of type A), and
2. time displays whose hour is between 10 and 11 (let's say these are time displays of type B).

We want to check whether it is possible to create x time displays of type A and y time displays of type B. We can do this by greedily assigning stickers to positions with the fewest possible stickers to be put into first.

In other words, we can do the following steps:

	Possible stickers			
	`H`	`H`	`M`	`M`
Type A	0	$[0, 9]$	$[0, 5]$	$[0, 9]$
Type B	1	$[0, 1]$	$[0, 5]$	$[0, 9]$

1. Assign x stickers numbered 0 to the first `H` position for x time displays of type A.
2. Assign y stickers numbered 1 to the first `H` position for y time displays of type B.
3. Combine the remaining stickers numbered 0 and 1 together, and assign y of them to the second `H` position for y time displays of type B.
4. Combine the remaining stickers numbered 0 to 5 together, and assign $x+y$ of them to the first `M` position for $x + y$ time displays.
5. Combine the remaining stickers numbered 0 to 9 together, and assign $2x + y$ of them to the remaining positions.

If, at any step, we don't have enough number of required stickers, then it is not possible to create x time displays of type A and y time displays of type B.

To find the maximum value of $x + y$ among all valid values of x and y, we can try all possible values of x from 0 to n and use binary search to find the maximum valid value of y for the corresponding value of x.

This solution solves the problem in $O(n \log n)$.

L Onion

The key observation is that we can partition all points into $O(\sqrt{n})$ groups such that the points in each group are collinear.

Let $x_i = i$ and $y_i = (ai + b) \bmod n$ represent the coordinates of the i-th point. Let $r = \lceil \sqrt{n} \rceil$. We can split the range $[0, n-1]$ of coordinates into r intervals: $[0, r-1], [r, 2r-1], \ldots, [(r-1)r, n-1]$. By the pigeonhole principle, there exists two indices $0 \le j < k \le r$ such that y_j and y_k lie in the same interval.

Let $u = k - j = x_k - x_j$ and $v = y_k - y_j$. We have $u \le r$, $|v| \le r - 1$, and $v \equiv (ak + b) - (aj + b) \equiv au \pmod n$. Fix some $0 \le s \le u - 1$ and consider the sequence of points with indices $s, s+u, s+2u, \ldots$ ($s+tu$ for $0 \le t \le \lfloor \frac{n-1-s}{u} \rfloor$). We may write

$$\begin{aligned} y_{s+tu} &= (a(s + tu) + b) \bmod n \\ &= (as + b + atu) \bmod n \\ &= (y_s + tv) \bmod n \\ &= y_s + tv - n \left\lfloor \frac{y_s + tv}{n} \right\rfloor. \end{aligned}$$

For values of t with the same $\left\lfloor \frac{y_s+tv}{n} \right\rfloor$, the points (x_{s+tu}, y_{s+tu}) lie on the same line with slope $\frac{v}{u}$. If $|v| \neq 0$, as t increases, the value of $\left\lfloor \frac{y_s+tv}{n} \right\rfloor$ can only change when t changes by $\frac{n}{|v|}$. Therefore, the range of t can be partitioned into at most $\left\lfloor \frac{|v|}{u} \right\rfloor + 1$ intervals where the value of $\left\lfloor \frac{y_s+tv}{n} \right\rfloor$ is constant in each interval. In other words, the points with indices $s, s+u, s+2u, \ldots$ form at most $\left\lfloor \frac{|v|}{u} \right\rfloor + 1$ consecutive groups that are collinear. Summing this for s, we can partition all n points into a total of at most $|v| + u = O(\sqrt{n})$ groups.

We maintain the list of collinear point groups from top to bottom, each represented by left and right endpoints. Notice that only the endpoints of each group can be the vertices of the convex hull. Since there are only $O(\sqrt{n})$ candidates, the convex hull can be found in $O(\sqrt{n} \log n)$ time (which can be optimized to $O(\sqrt{n})$ if sorting is avoided by iterating points in a certain order, but this is not necessary). Then, we need to remove points on the convex hull boundary, which are

- all points in the topmost nonempty group,
- all points in the bottommost nonempty group, and
- the left or right endpoints of each group that lie on the boundary.

Each of these can be updated in $O(1)$ time for a total time complexity of $O(\sqrt{n})$.

To find k layers of convex hull, simply repeat the process k times. The total time complexity is $O(k\sqrt{n} \log n)$ or $O(k\sqrt{n})$.

M Deformed Balance

In what follows, we write an open parenthesis as `L` and a close parenthesis as `R`. The BNF formulation of the deformed strings can be defined as follows.

$$E ::= \texttt{R} \mid E\ \texttt{R}\ E \mid \texttt{L}\ E\ \texttt{L}.$$

Consider a slightly modified version of E, where the terminal symbol `L` is replaced by either '`{`' or '`}`':

$$E' ::= \texttt{R} \mid E'\ \texttt{R}\ E' \mid \{\ E'\ \}.$$

Suppose that a string S' conforms to E'. The following can be proven by induction.

- The string S' contains an odd number of `R`'s.
- Removing `R` from S' will result in a *balanced* sequence consisting of '`{`' and '`}`'.
- Two terminals '`{`' and '`}`' do not appear at adjacent positions; at least one `R` is always between them.
- The appearance of one terminal `R` flips the direction (opening or closing) of brackets `{}`.

Thus, S' has a form `{{{ ... {{{ R }}} ... }}} R {{{ ... }}} R }}} ... }}} R {{{ ... {{{ R }}} ...`. Two or more `R`'s may appear contiguously. Conversely, if a string S' has this form and the brackets `{}` are balanced, we can generate S' using E':

- `R` at odd occurrences should be generated by the first rule,
- `R` at even occurrences should be generated by the second, and
- each pair of corresponding '`{`' and '`}`' should be generated by the third.

Hence, if we want to determine whether a string consisting of `R` and `L` conforms to E, we can do so by replacing the `L`'s with either '`{`' or '`}`' and then checking if that conforms to E'.

For determining whether a string W has a deformed balance or not, we track two values x and y from the beginning, where x is the current nesting level for the parentheses `LR`, and y is the nesting level for the brackets `{}`. During the tracking $x, y \geq 0$ must be always hold. At the end of the string, $(x, y) = (1, 0)$ must hold for the string W to have a deformed balance.

The values of (x, y) change as follows:

- if the next character is `L`,
  - $(x + 1, y + 1)$ if $x + y = 0 \pmod 2$, or
  - $(x + 1, y - 1)$ otherwise
- if the next character is `R`, $(x - 1, y)$.

Now, let's move on to the original problem. For the input string S, we can prepend and append some characters to S to form $S' = X + S + Y$.

- The concatenation $X + S$ should result in a state (x', y') such that $x', y' \geq 0$.
  - After this, if $x' = 0$ and $y' = 0 \pmod 2$, $|Y| = 3 + 2y'$ is the shortest possible for S' to have a deformed balance.
  - Otherwise, $|Y| = x' + 2y' - 1$ is the required minimum.
- The only forms of X that need to be considered are one of the following:
  - `LLL... LLL`
  - `LLL... LLL R LLL... LLL`
  - `LLL... LLL R LLL... LLL R`
- Note that we don't need to consider a form like `LLLRLLLRLL` because the last `L`'s can be moved to the first part: `LLLLLRLLLR` is enough to consider.
- For the third case above, suppose that X has a form $L^f RL^g R$ for some $f > 0$ and $g \geq 0$ (with $f+g \geq 2$). The state after reading X is $(f + g - 2, f - g)$. For S' to have a deformed balance, $f + g - 2 + a \geq 0$ and $f - g + b \geq 0$ must hold, where a and b are constants that can be deduced from the input.
- The length of Y is $|Y| = (f + g - 2 + c) + 2(f - g + d) - 1$ for some c and d (unless $X + S$ ends up in the state $(0, 2y'')$ for some $y'' \geq 0$.)
- Thus, the total added length is $|X| + |Y| = 4f + (\text{const.})$. To find the answer, it suffices to iterate $f = 1, 2, \ldots$ and find the minimum f such that there exists g satisfying all the conditions above.

- The same arguments hold for the other cases. Note that the constants (a, b, c, d) may vary in the second and the third cases, due to the flip of brackets `{}`.

Total time complexity is $O(n)$.