Singapore Informatics League 2026

Task Editorial (Final Contest)

September 2026

Contents

Task 1: `6 ate 8`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Jayden Tiew (Shadow1)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

An element can only eat the element directly to its right, and only if that element is exactly 2 larger, so the eater is always the left one of the pair. Eating does not change the eater's own value.

Subtask 1

Additional constraints: $a[1] \leq a[2] \leq \cdots \leq a[n]$

Cut the sequence between positions $j-1$ and j whenever $a[j]$ is neither equal to $a[j-1]$ nor equal to $a[j-1]+2$. Because the sequence is sorted, each block between cuts has the form $m, m+2, m+4, \ldots$, where each value appears one or more times and all copies of a value are next to each other. For example, a block might be $2, 2, 4, 6, 6, 6$.

If a block contains only one distinct value, nothing happens inside it. Otherwise, the copies of the largest value sit at the right end of the block, and the element just to their left is exactly 2 smaller, so it eats them one at a time. Then the copies of the second largest value are at the right end, and we repeat until only the copies of m are left.

The order matters. In the block $1, 3, 5$, if you let 1 eat 3 first, you are left with $1, 5$ and stuck. Eating from the largest value downwards instead gives $1, 3$ and then 1. This is why we always clear the right end of a block first.

Observation 1: The copies of m are never eaten.

Eating a copy of m needs a value $m-2$ directly to its left. The block contains no such value, and neither does the previous block. The last element of the previous block is at most $m-1$ and, by the cut rule, not $m-2$. Every value in that block is at most its last element and has the same parity as that element, so none of them equals $m-2$. Any $m-2$ lies at least two blocks to the left, and bringing it next to a copy of m means first clearing the entire previous block, including the copies of that block's own smallest value. The same argument applies to those copies. Following this to the left, we reach the first block, which has nothing to its left at all, so

none of these copies are ever eaten.

The answer for a test case is therefore the sum, over all blocks, of the number of copies of the block's smallest value. A single scan finds the cuts and the counts.

Time complexity: $\mathcal{O}(n)$

Subtask 2

Additional constraints: $n \leq 1000$ and $a[i] \leq 1000$ for all $1 \leq i \leq n$

Without sortedness the blocks are no longer laid out neatly, but the lesson of the $1, 3, 5$ example is enough to solve the general case. Eating the 3 early was a mistake because the 3 was the only element that could remove the 5, so it was still needed as a stepping stone.

An element is only needed as a stepping stone for elements to its right, so once everything to its right has been dealt with, eating it does no harm. This suggests a greedy that works from the right end of the sequence towards the left. Repeatedly find the rightmost pair where an element can eat its neighbour, and perform that eat.

Observation 2: Some optimal eating sequence begins with the rightmost available eat.

Let $(i, i+1)$ be the rightmost pair where an eat is possible. Nothing to the right of position i can eat, so the suffix starting at position $i+1$ can only lose elements from the left, to whatever sits at position i, which is $a[i]$ until $a[i]$ itself is eaten. After that, the element next to the suffix is the one that ate $a[i]$, with value $a[i]-2$, and it cannot eat $a[i+1] = a[i]+2$. Any later neighbour of the suffix is smaller still, so if $a[i]$ is eaten first, the suffix is frozen forever.

Any eating sequence therefore either never touches the suffix, or has $a[i]$ eat $a[i+1]$ as the first meal of $a[i]$. In the second case, moving that meal to the very front of the sequence invalidates nothing else. In the first case, the eat can be added at the front, and the sequence stays valid and removes one more element. Either way, some optimal sequence starts with the rightmost available eat, and by induction the greedy is optimal.

Each eat removes one element, so there are at most $n-1$ eats, and finding the rightmost available pair in the current sequence takes $\mathcal{O}(n)$ time.

Time complexity: $\mathcal{O}(n^2)$

Subtask 3

Additional constraints: No additional constraints

The same greedy can be run in a single pass from right to left. Keep the elements already dealt with on a stack, with the leftmost of them on top. For the current element x, pop the top of the stack while it equals $x + 2$, since x eats it. Then push x.

The loop stops at the right moment. The value $x + 2$ is the only value x can ever eat, so once the top is not $x + 2$, there is nothing more for x to do. The top of the stack is always the element immediately to the right of x in the current sequence, and no element on the stack can eat another, so while x is processed only x changes the stack, and the top is the only element x could eat next.

At the end, the answer for the test case is the size of the stack. Each element is pushed once and popped at most once.

In the fourth test case of the example, $a = [12, 6, 8, 10, 10, 7, 6, 8, 8]$, and the stack develops as follows:

Element processed	**Stack (top first)**
8	8
8	$8, 8$
6	6
7	$7, 6$
10	$10, 7, 6$
10	$10, 10, 7, 6$
8	$8, 7, 6$
6	$6, 7, 6$
12	$12, 6, 7, 6$

Four elements remain, which matches the expected answer for that test case.

Time complexity: $\mathcal{O}(n)$

Subtask	**Answer**
1	15497501
2	15683
3	8959153

Task 2: `Dog Walk`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Ernest Kiew (Shor the Duck)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

You can always stay still, and the dog bans only one direction per turn. If the direction you want is banned on the current turn, you can wait for a later turn that allows it, if there is one. The only cost is the turns spent waiting.

Subtask 1

Additional constraints: The grid contains no blocked cells

Let $c_v = |e_x - s_x|$ and $c_h = |e_y - s_y|$. With no blocked cells there is no reason to take a detour. Some optimal route makes exactly c_v vertical steps, all in one direction d_v, and c_h horizontal steps, all in one direction d_h. Any other step would cost a turn and later have to be undone, and waiting instead is never worse.

What remains is to choose the turns for these steps. They must fit into turns $1, 2, \ldots, T$, with at most one step per turn and no step in direction $d[i]$ on turn i. The answer is the smallest T for which this is possible.

There are three lower bounds on T:

- $T \geq c_v + c_h$, since each turn holds at most one step;
- $T \geq t_v$, where t_v is the index of the c_v-th turn with $d[i] \neq d_v$, since only those turns can hold a vertical step;
- $T \geq t_h$, where t_h is the index of the c_h-th turn with $d[i] \neq d_h$, for the same reason.

If $c_v = 0$, there are no vertical steps, d_v does not matter, and $t_v = 0$. Similarly, $t_h = 0$ if $c_h = 0$. In particular, if the start is the goal, the answer is 0, as in the second test case of the first example.

The smallest T that meets all three bounds can always be achieved. Among turns 1 to T, a turn that bans d_h can only be used for a vertical step, a turn that bans d_v can only be used for a horizontal step, and any other turn can be used for either. Give each direction its own turns first, and use the shared turns for whatever is left. The second and third bounds say that each direction has enough usable turns, and the first says that there are enough turns overall, so the shared turns can cover what is left of both directions.

The answer is therefore

$$T = \max(c_v + c_h,\ t_v,\ t_h),$$

or -1 if $T > n$. It is also -1 if fewer than c_v turns in the whole string have $d[i] \neq d_v$, or fewer than c_h have $d[i] \neq d_h$, because then t_v or t_h does not exist. Both indices can be found in one pass over d, and the grid does not even need to be stored.

In the first test case of the first example, we move 2 cells up and 4 cells right, and $d =$ `RURRRLUL`. The first bound is $2 + 4 = 6$. The 2nd turn that is not `U` is turn 3, and the 4th turn that is not `R` is turn 8. The answer is $\max(6, 3, 8) = 8$, which matches the example.

Time complexity: $\mathcal{O}(n)$

Subtask 2

Additional constraints: $n, h \cdot w \leq 1000$ for each test case, and $\sum(h \cdot w) \leq 20\,000$

Once there are blocked cells, we have to search the grid, and a plain BFS over the cells is not enough. In the third test case of the second example, the goal is 5 cells away, but the answer is 9. The only route of length 5 is `UUURU`, and its right step has to come after its three up steps. Those three steps need turns 1, 5 and 6, and from turn 5 onwards the dog always refuses to go right, so this route cannot be completed within n turns. A longer route can.

Since the bans depend on the turn number, the turn number has to be part of the state. Keep the set of cells that can be reached in t turns. To go from t to $t + 1$, try staying put and each of the three directions allowed on turn $t + 1$ from every cell in the set. The answer is the first t at which the goal is in the set, or -1 if it is still missing after n turns.

Equivalently, build a graph on the pairs (turn, cell), with an edge from (t, u) to $(t + 1, v)$ whenever moving from u to v is allowed on turn $t + 1$, and run a BFS from (turn 0, starting cell). Both versions visit at most $n \cdot h \cdot w \leq 10^6$ states per test case, each with at most 5 moves to try, and the bound on $\sum(h \cdot w)$ keeps the total within $2 \cdot 10^7$ states.

Time complexity: $\mathcal{O}(n \cdot h \cdot w)$

Subtask 3

Additional constraints: No additional constraints

Now $n \cdot h \cdot w$ can be as large as 10^{12}, so we can no longer keep the turn number in the state. Instead, we use the fact that reaching a cell earlier is never worse than reaching it later. The bans depend only on the turn number, so an early arrival can wait and then do whatever a later arrival would do. It is therefore enough to store one number per cell, dist$[u]$, the earliest turn count at which we can stand on cell u.

To step from u in direction z after k turns, wait for the first turn after turn k that allows z, as in the Note. Let nxt$[k][z]$ be the smallest $t > k$ with $d[t] \neq z$, or $n + 1$ if there is none. The step then lands on the neighbouring cell after exactly nxt$[k][z]$ turns.

The table can be filled for $k = 0, 1, \ldots, n$ in one backward pass over d. Keep four running values, all starting at $n + 1$. For $t = n$ down to 1, first store the current values as nxt$[t][\cdot]$, then set the three directions other than $d[t]$ to t. At the end, store the values as nxt$[0][\cdot]$. With four directions, this takes $\mathcal{O}(n)$ time.

Steps no longer all cost one turn, which rules out BFS. However, nxt$[k][z]$ never decreases as k increases, so leaving later never lets us arrive earlier. Because of this, Dijkstra's algorithm still works if we settle cells in increasing order of dist and relax each edge with nxt instead of adding a weight. A cell whose distance exceeds n is not relaxed, because nothing reached from it can arrive within n turns, and the table only covers $k \leq n$.

The answer is dist of the goal, or -1 if it is larger than n or the goal is never reached. Remember that a -1 still contributes -1 times the test case number to the output.

Since every distance that is ever set is an integer from 0 to $n + 1$, a bucket queue would remove the logarithm, but it is not needed here.

Time complexity: $\mathcal{O}(h \cdot w \log(h \cdot w) + n)$

Subtask	Answer
1	473546
2	56183
3	40615619

Task 3: `Gremlin Nob`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Ryan Goh (rgca)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

Suppose you play k cards, and let P_i be the total damage dealt by the first i of them. The Gremlin Nob attacks after every card except the killing one, and its i-th attack deals P_i damage. The total damage taken is therefore

$$P_1 + P_2 + \cdots + P_{k-1}.$$

Now count per card. The card played at position j appears in each of $P_j, \ldots, P_{k-1}$, so it is paid for $k - j$ times. Larger cards should therefore be played later. If every card of a chosen set is played, the damage taken is smallest when the set is played in ascending order, and we call this damage the charge of the set.

Observation 1: The answer is the minimum charge over all sets of cards with total damage at least h.

The charge is at least the damage actually taken when the set is played, so the minimum is never below the answer. Conversely, the cards played in an optimal play form a set with total damage at least h. Every card of that set is played, so by the Note the damage taken is at least its charge. The minimum is therefore never above the answer either. A valid set always exists, because $h \leq \sum a[i]$.

Subtask 1

Additional constraints: $1 \leq a[i] \leq 3$ for all $1 \leq i \leq n$

There are at most three distinct card values, and two cards of the same value are interchangeable, so a set is described completely by how many 1s, 2s and 3s it contains. Let c_1, c_2 and c_3 be the number of cards of each value. Try every triple (u, v, w) with $u \leq c_1$, $v \leq c_2$ and $w \leq c_3$, and discard the triples with $u + 2v + 3w < h$, since those do not kill the Gremlin Nob. For every remaining triple, the order is fixed (all the 1s, then all the 2s, then all the 3s), so the damage taken follows from the formula in the Note in $\mathcal{O}(n)$ time. The answer is the minimum over all triples.

There are $(c_1+1)(c_2+1)(c_3+1) \leq 16^3 = 4096$ triples, so no dynamic programming is needed.

Time complexity: $\mathcal{O}(n^4)$

Subtask 2

Additional constraints: $1 \leq n \leq 20$

The values $a[i]$ are large again, so cards can no longer be grouped by value. However, $2^{20} \approx 10^6$ is small enough to look at every set of cards directly.

Sort a in ascending order first. For each of the 2^n subsets, go through its cards in sorted order, which is the optimal order by the Note. Keep track of the damage dealt and the damage taken so far, and stop as soon as the damage dealt reaches h. If it never does, discard the subset. This is about $2 \cdot 10^7$ operations.

Time complexity: $\mathcal{O}(2^n \cdot n)$

Subtask 3

Additional constraints: No additional constraints

With $n \leq 45$, enumerating all 2^{45} subsets is too slow, which suggests meet in the middle.

Sort a in ascending order and split it into a low half L, the smallest $\lfloor n/2 \rfloor$ cards, and a high half H, the rest. Every card in L is at most every card in H. For any combined set, the optimal ascending order is therefore its L-part in ascending order followed by its H-part in ascending order.

Take a combined set and split it into its L-part and its H-part. The quantities involved are:

- X_L and X_H, the total damage dealt by each part;
- c_L and c_H, the damage taken from each part if that part alone were the whole fight, that is, the formula from the Note applied to that part on its own;
- sz, the number of cards in the H-part.

The combined set kills the Gremlin Nob exactly when $X_L + X_H \geq h$, and among all such pairs we want the one with the smallest total cost.

Let Q_j be the total damage of the first j cards of the L-part, so that $Q_p = X_L$, where p is the number of cards in the L-part. Let R_t be the total damage of the first t cards of the H-part. Since the L-part is played first, the total damage dealt after each card of the fight is, in order,

$$Q_1,\ \ldots,\ Q_p,\ X_L + R_1,\ \ldots,\ X_L + R_{\text{sz}},$$

and the damage taken is the sum of all of these except the last:

$$(Q_1 + \cdots + Q_p)\ +\ (\text{sz} - 1) \cdot X_L\ +\ (R_1 + \cdots + R_{\text{sz}-1}).$$

The last bracket is c_H. The first bracket is $c_L + X_L$. The formula for c_L leaves out $Q_p = X_L$, but here Q_p is paid, because the killing card is in the H-part. Adding up the three brackets gives

$$\text{cost} = c_L + c_H + \text{sz} \cdot X_L.$$

This assumes $\text{sz} \geq 1$. The case $\text{sz} = 0$ is the L-part killing on its own, which is handled separately below.

Once sz is fixed, the cost separates into two parts:

$$\text{cost} = \underbrace{\left(c_L + \text{sz} \cdot X_L\right)}_{\text{depends only on the } L\text{-part}} + \underbrace{c_H}_{\text{depends only on the } H\text{-part}}.$$

Without fixing sz, the H-part would affect the L-part's contribution through the multiplier, and the two halves could not be optimised independently. With sz fixed, the first bracket is a constant for a given L-subset, and all we need from the H side is the smallest c_H among H-subsets of size sz with $X_H \geq h - X_L$.

Enumerate all subsets of both halves as in Subtask 2. If a half-subset already reaches h on its own, its cost is immediately a candidate answer, and it is not stored. This covers the case $\text{sz} = 0$, and also every combined set that kills the Gremlin Nob during its L-part. It also covers every combined set whose H-part reaches h by itself, because that H-part alone costs at most c_H, which is at most $c_L + c_H + \text{sz} \cdot X_L$. A combined set that kills the Gremlin Nob partway through its H-part needs no special care, since by Observation 1 it may be charged in full like any other. All other half-subsets are stored, the L-subsets in one list and the H-subsets in the bucket indexed by their size sz.

Sort each H-bucket by X_H, then replace each entry's cost by the suffix minimum of the costs in that bucket. A lookup at $X_H \geq h - X_L$ then returns the cheapest valid entry directly.

Finally, go through every stored L-subset and every sz from 1 to $|H|$. Binary search bucket sz, if it is non-empty, for the first entry with $X_H \geq h - X_L$, read off its suffix minimum c_H, and take the minimum of $c_L + c_H + \text{sz} \cdot X_L$ over everything.

Enumerating the halves and sorting the buckets takes $\mathcal{O}(n \cdot 2^{n/2})$ time. The combining loop performs about $2^{22} \sum_{\text{sz}} \log_2 \binom{23}{\text{sz}} \approx 1.4 \cdot 10^9$ binary search steps, each a probe into the buckets, which is fine with no time limit to meet.

The L-subsets never actually need to be stored. They can be enumerated and queried on the fly, so only the H-buckets, about 130 megabytes, are kept in memory.

The binary searches can be removed altogether by processing the L-subsets in decreasing order of X_L. The threshold $h - X_L$ then only increases, and one moving pointer per bucket is enough. The price is that the L-subsets must be stored and sorted as well, which takes about 70 megabytes more. After sorting, the combining loop takes $\mathcal{O}(n \cdot 2^{n/2})$ time.

Costs reach about 10^{12}, so use a 64-bit integer type throughout.

Time complexity: $\mathcal{O}(n^2 \cdot 2^{n/2})$, or $\mathcal{O}(n \cdot 2^{n/2})$ with the moving pointers

Subtask	**Answer**
1	11650
2	557118549355
3	5073178661088

Task 4: `Reveille`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Ziv Lim (misalignedDiv)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

Waking a soldier by hand starts a chain. The soldier wakes a neighbour, that neighbour wakes one of their own neighbours, and so on. The chain stops when it reaches a soldier who points off the end of the row, or when it tries to wake somebody who is already awake. The task is therefore to choose where the chains start.

Subtask 1

Additional constraints: $tc = 4$ and in each test case, there is exactly one index i ($1 \leq i < n$) such that $s[i] \neq s[i+1]$

The string changes character exactly once, so it is either some `L`s followed by some `R`s, or some `R`s followed by some `L`s.

Consider first the shape `LL`$\ldots$`LRR`$\ldots$`R`. The last `L` and the first `R` are next to each other and point away from each other. Nobody can wake the last `L`. Their left neighbour, if any, is an `L` pointing further left, and their right neighbour is an `R` pointing further right. The last `L` must therefore be woken by hand, and by the same argument so must the first `R`. Waking these two wakes everybody, and the answer is always 2.

Now suppose the string has the shape `RR`$\ldots$`RLL`$\ldots$`L`, with a `R`s followed by b `L`s. Waking soldier 1 sends a chain rightwards through every `R` and one step into the `L`s, where it stops, because that `L` tries to wake somebody who is already awake. Waking soldier n does the mirror image. Two manual wake-ups are therefore always enough. It remains to decide when one is enough.

If $a \geq 2$ and $b \geq 2$, soldier 1's only neighbour is an `R` pointing away from them, so nobody can wake soldier 1. Likewise, soldier n's only neighbour is an `L` pointing away from them. Both have to be woken by hand, and the answer is 2.

Otherwise, $a \leq 1$ or $b \leq 1$, and one of the two chains above already covers the whole row. If $a \leq 1$, start at soldier n, and if $b \leq 1$, start at soldier 1. The answer is 1.

Together, the two cases give the following cost for a string of the shape `RR...RLL...L` with a `R`s and b `L`s:

$$\text{cost}(a, b) = \begin{cases} 2 & \text{if } a \geq 2 \text{ and } b \geq 2, \\ 1 & \text{otherwise.} \end{cases}$$

This rule also covers a string made entirely of `R`s or entirely of `L`s, which costs 1.

In the example, `RL` costs 1, `LR` costs 2, and `RRRRRRRRRRRLLLLL` costs 2, which agrees with both rules above.

Time complexity: $\mathcal{O}(n)$

Subtask 2

Additional constraints: $tc = 5000$ and $n \leq 14$

The string can now have any shape, but n is tiny, so we can try every set of soldiers to wake by hand. There are 2^n sets. For each set S, simulate the chains in $\mathcal{O}(n)$ and check whether every soldier ends up awake. Among the sets that wake everybody, take the one with the fewest manual wake-ups.

Over the 5000 test cases this is about $1.1 \cdot 10^9$ simple operations, which is fine here. Precomputing each soldier's chain as a bitmask, and obtaining the woken set of S from that of S without its lowest bit, reduces the cost to $\mathcal{O}(2^n)$ per test case.

Time complexity: $\mathcal{O}(2^n \cdot n)$, or $\mathcal{O}(2^n)$ with precomputed chains

Subtask 3

Additional constraints: $tc = 9$

Here n can be as large as $100\,000$. Instead of searching, we reduce a general string to the shape from Subtask 1.

The key observation is that an `L` immediately followed by an `R` acts as a wall. These two soldiers point away from each other, so no chain ever crosses between them, in either direction. We can therefore cut the string at every occurrence of `LR`, solve the pieces separately and add up their answers.

A piece contains no `L` immediately followed by an `R`, so no `L` comes before an `R` inside it. Every piece therefore has the shape `RR...RLL...L`, and the rule from Subtask 1 applies directly.

This gives a single left-to-right scan. Read a maximal run of `R`s of length a (empty only if this is the first piece), then a maximal run of `L`s of length b (empty only if this is the last piece). This is one piece. Add 2 to the answer if $a \geq 2$ and $b \geq 2$, and add 1 otherwise. Repeat from the current position until the string is exhausted. Each character is read once.

The answers are far below $10^9 + 7$, so the modulo in the output format never actually reduces anything, but apply it anyway.

Time complexity: $\mathcal{O}(n)$

Subtask	**Answer**
1	16
2	61534143
3	990536

Task 5: `Slimes`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Ernest Kiew (Shor the Duck)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

The problem involves two kinds of eating. The real timeline is the given schedule of events, which actually happen and change the row. The questions instead ask about hypothetical eating, in which one living slime is allowed to eat freely at a frozen moment. Hypothetical eating changes nothing, and the hypotheticals of two different slimes never interact.

In the real timeline, the eater of each event comes from the input, never from a size comparison. When the two slimes have equal size, both look like valid eaters, but only one of them carries the number that the answer is attributed to. Getting this backwards is invisible on data without ties, so any stress test should contain many equal adjacent sizes. Everywhere, including in the brute force, use the statement's convention that a slime eaten at second t is no longer alive at moment t.

Fix a moment and write the alive slimes, from left to right, as $c[1], c[2], \ldots, c[m]$, with prefix sums $Q[i] = c[1] + \cdots + c[i]$. Write V for the total mass, which never changes. Since $n \le 4 \cdot 10^6$ and $s[i] \le 10^9$, we have $V \le 4 \cdot 10^{15}$, so $\log_2 V \approx 52$. This is a different constant from $\log_2 n \approx 22$, and the two do different jobs below. A third constant, $\log_\varphi V \approx 75$, where φ is the golden ratio, bounds the number of phases in Observation 3. Keep the three apart in complexity calculations.

Observation 1: The set of slimes that a slime can eat does not depend on the order of its meals.

A slime's weight only ever increases, so a neighbour that is edible now is still edible later. Every maximal greedy sequence therefore eats the same set. This lets us talk about the reach $f(v)$ of a slime v, rather than one possible outcome.

Since v only ever eats at the two ends of the block it has consumed, $f(v)$ is an interval $[l, r]$. Once v has eaten all of $[l, r]$, its weight is $Q[r] - Q[l-1]$, and it is stuck on both sides, so

$$c[l-1] > Q[r] - Q[l-1] \qquad \text{and} \qquad c[r+1] > Q[r] - Q[l-1],$$

where a neighbour beyond the end of the row counts as $+\infty$. Call an interval with this property closed.

Observation 2: The reach $f(v)$ is the smallest closed interval containing v.

We have just seen that $f(v)$ is closed. Conversely, nothing inside a closed interval can ever escape it, since any slime in there accumulates at most $Q[r] - Q[l-1]$, which is strictly less than either neighbour. Hence $f(v)$ is contained in every closed interval that contains v.

The inequalities are strict because eating is allowed on ties, and a neighbour of size exactly $Q[r] - Q[l-1]$ gets eaten. Several of the arguments below fail if $>$ is replaced by $\geq$. The arguments also rely on every size being at least 1. A dead slime must therefore be removed from the structure rather than stored with size 0, since a zero is edible by everybody.

Extending the reach in one direction takes a single query. Suppose the eater has consumed $[l, r]$ and wants to keep going right. It can eat position $p = r + 1$ if and only if

$$c[p] \;\leq\; Q[p-1] - Q[l-1], \qquad \text{that is,} \qquad c[p] - Q[p-1] \;\leq\; -Q[l-1],$$

and the same test applies to each later p as it keeps going, because after eating p its weight is again everything from l to p. Define

$$\kappa[p] = c[p] - Q[p-1].$$

The rightward run from left endpoint l stops just before the first $p > r$ with $\kappa[p] > -Q[l-1]$.

The threshold depends only on the left endpoint l, and not on the eater's weight or on r. Eating rightwards accumulates everything to the right of l anyway, so a whole run of rightward meals is one query rather than one query per meal. The running maximum of κ over $[r+1, p]$ is non-decreasing in p, so finding the first index that exceeds a threshold is a binary search on a sparse table of range maxima, or a descent on a max segment tree. Either way it takes $\mathcal{O}(\log n)$ time.

Symmetrically, for leftward extension from right endpoint r, position $p = l - 1$ is edible if and only if $c[p] \leq Q[r] - Q[p]$. With

$$\lambda[p] = c[p] + Q[p],$$

the leftward run stops just after the last $p < l$ with $\lambda[p] > Q[r]$, and this threshold depends only on r.

To compute a reach, extend right until stuck, then left until stuck, then right again (the new mass may have unblocked it), and so on.

Observation 3: This alternation has at most about $\log_\varphi V \approx 75$ phases, where $\varphi \approx 1.618$ is the golden ratio.

Write W_i for the weight at the end of phase i. When phase i ended, the slime was blocked by a neighbour heavier than W_i. Phase $i + 2$ resumes in the same direction, so if it makes any progress at all, its first meal is that blocker, and $W_{i+2} > W_{i+1} + W_i$. The weight therefore more than doubles relative to the previous stop in the same direction, two phases earlier, but a single phase need not double anything. The recurrence is Fibonacci's, and since the weight

starts at 1 or more and never exceeds V, there are at most about $\log_\varphi V \approx 75$ phases. This is still $\mathcal{O}(\log V)$, so one reach query costs $\mathcal{O}(\log V \log n)$.

In terms of reaches, Alice asks whether $f(v)$ is the whole row. Bob asks, for each alive slime, for the number of other alive slimes inside $f(v)$, which is its length minus one. Bob's total for one moment is at most n^2 and fits in a signed 64-bit integer, but the sum over all moments is of order n^3 and does not. Reduce modulo $998\,244\,353$ as you go.

Subtask 1

Additional constraints: $1 \le n \le 50\,000$ and $1 \le \sum n \le 170\,000$

At this size we can answer both questions straight from the definition, and we can afford to throw the structure away and rebuild it at every moment. There are n moments with at most n alive slimes each, so there are about $n^2/2$ reach queries in total. At each moment, do the following.

1. Compact the alive slimes into the array $c[1..m]$ and compute Q, κ and λ.
2. Build a sparse table over κ and λ, or a segment tree, which builds bottom-up in $\mathcal{O}(m)$.
3. For each alive slime, run the alternation from the Note, binary searching for each blocker. Bob's contribution is the length of the reach minus one, and Alice's is 1 if the reach is all of $[1, m]$.

Because everything is rebuilt, this solution needs no point updates, lazy state or incremental bookkeeping. Apart from the reach computation from the Note, it shares none of the machinery of the other subtasks, so it is worth writing even after you have solved them, as an oracle to stress-test them against.

The total is $\mathcal{O}(n^2 \log V \log n)$ on paper. There are about $n^2/2 \approx 1.25 \cdot 10^9$ reach queries at $n = 50\,000$, but each costs far less than the bound suggests, since almost every query settles after two or three phases rather than about 75. With no time limit to meet, this is affordable.

Time complexity: $\mathcal{O}(n^2 \log V \log n)$

Subtask 2

Additional constraints: $q = 1$, $1 \le n \le 4 \cdot 10^6$, and $1 \le \sum n \le 8.1 \cdot 10^6$

Alice needs an answer for every slime at every moment, which is still $\Theta(n^2)$ questions. Unlike Bob's question, hers has a yes or no answer, and an answer that switches only once can be binary searched.

Observation 4: A merge never helps any slime that does not take part in it.

Let u and v merge into one slime, and let D be any other slime. Take any legal eating sequence of D after the merge and copy it verbatim, except at the step where D eats the merged slime. At that step, D has weight at least the size of the merged slime, which is the sum of the current sizes of u and v. Before the merge, D can therefore eat whichever of u and v is nearer, and then the other. Its weight afterwards is the same, so the rest of the sequence carries over. Everything achievable after a merge was achievable before it, so a slime that is never touched can only lose ability as time passes.

Observation 4 says nothing about the eater. When X eats Y, the weight of X jumps, and an ability that X had lost can come back. Nothing forces us to treat the slime after the merge as the same object as before, though. When X eats Y at second t, declare that X and Y both die and a new object Z is born, carrying the number of X. Every object is then untouched throughout its life, because by construction its life ends at the event that would break monotonicity. There are $2n-1$ objects in total, and for each of them, "can eat everyone" is true for a while and then false for the rest of its lifespan. At most one object with a given number is alive at any moment, so summing the good moments of each object onto its number counts nothing twice.

We can no longer rebuild the structure at every moment, so κ and λ have to be maintained through the events.

Observation 5: An alive slime always occupies a contiguous range of original indices.

This is true at the start, and a merge joins two ranges that are adjacent in the row. We can therefore work with the prefix sums P of the original sizes, which never change. For an alive slime occupying $[a, b]$, the keys κ and λ become

$$P[b] - 2P[a-1] \quad \text{stored at its start } a, \qquad\qquad 2P[b] - P[a-1] \quad \text{stored at its end } b,$$

with thresholds $-P[l-1]$ and $P[r]$ as before, where l and r are now the first and last original indices of the eater's consumed block. An event merges $[a, b]$ with $[b+1, c]$. Two entries die, the key at start $b+1$ and the key at end b, and two are rewritten, at start a and at end c. That is $\mathcal{O}(1)$ point updates per event, at $\mathcal{O}(\log n)$ each.

Each object needs one binary search over its own lifespan. The difficulty is that a probe "is object L good at moment t" needs the world as it stood at moment t, and different objects want very different moments. Run the binary searches in parallel instead of one at a time. In each round:

1. compute the current midpoint of every unresolved object and bucket the objects by it;

2. reset to moment 0 and sweep the timeline forward once, applying each event as the $\mathcal{O}(1)$ point updates above and answering every probe parked at that moment;
3. halve the candidate range of each object according to its probe result.

After $\mathcal{O}(\log n)$ rounds every switch point is known, and each object contributes (switch point) $-$ (birth) moments to its number. The timeline is only ever swept forwards, so the structure never has to be rewound. The search range of an object is its lifespan, not $[0, n)$, since asking about an object that does not exist is meaningless.

The cost is

$$\mathcal{O}(\log n) \times \big(\underbrace{n \log n}_{\text{event replays}} + \underbrace{n \log V \log n}_{\text{probes}} \big) = \mathcal{O}(n \log^2 n \log V).$$

As with the queries of Subtask 1, the probe term is far below its bound in practice. The extra $\log n$ is an artefact of parallel binary search, not of the problem, and the structure of Subtask 3 answers Alice without it.

Time complexity: $\mathcal{O}(n \log^2 n \log V)$

Subtask 3

Additional constraints: $q = 2$, $1 \le n \le 4 \cdot 10^6$, and $1 \le \sum n \le 8.1 \cdot 10^6$

Bob needs a number for every moment, so there is nothing to binary search, and we have to maintain the collection of all reaches as the timeline moves.

Observation 6: Two reaches are always nested or disjoint.

Suppose $f(X) = [2, 7]$ and $f(Y) = [5, 9]$ crossed (the numbers are only for illustration). Both are closed, so $c[8] > \text{sum}(2..7)$ and $c[4] > \text{sum}(5..9)$. If Y sits inside $[2, 7]$, it can never weigh more than $\text{sum}(2..7) < c[8]$, so it never reaches 9. If Y sits outside, at 8 or 9, then

$$c[4] > \text{sum}(5..9) \ge c[8] > \text{sum}(2..7) \ge c[4],$$

using only that all sizes are positive and that position 4 lies inside $[2, 7]$. Either way, we get a contradiction.

Call the distinct reach intervals nodes. The heaviest alive slime is at least as heavy as every neighbour it ever meets, so it can eat everyone, and the whole row is a node at every moment. By Observation 6, the nodes therefore form a tree, with the whole row as its root. From here on, a node is recorded as a range of original indices, which is well defined by Observation 5.

Observation 7: The tree has depth at most $\log_2 V \approx 52$, but it can have $\Theta(n)$ nodes.

If node B lies strictly inside node A, one of the blockers of B is heavier than $\text{sum}(B)$ and lies inside A, so $\text{sum}(A) > 2\,\text{sum}(B)$. The mass doubles at every level, so the depth is at most $\log_2 V \approx 52$. The number of nodes, however, can be $\Theta(n)$. For example, alternating sizes $10^9, 1, 10^9, 1, \ldots$ give every 1 its own reach. Walking an ancestor chain is therefore affordable, but anything that touches all nodes once per moment is not.

Many slimes share a reach, so store the distinct intervals, each with two counters:

- own, the number of alive slimes whose reach is exactly this interval;
- $inside$, the number of alive slimes lying within it.

For example, with huge slimes on both sides, the row $5, 5$ gives one node with $own = 2$ and $inside = 2$, while $100, 1$ gives an outer node with $own = 1$ and $inside = 2$, plus a separate node for the 1 alone. A slime with reach $[l, r]$ eats everything alive in $[l, r]$ except itself, so

$$\text{Bob's answer for one moment } = \sum_{\text{nodes}} own \cdot (inside - 1).$$

This must be kept as a running number and updated by deltas, never recomputed. Storing $inside$ explicitly is unnecessary, since a Fenwick tree over the alive slimes answers it on demand.

Run the timeline backwards. Forward in time, merges make reaches shrink, and shrinking is destructive. In the alive row $\ldots, 100, 3, 3, \ldots$, the two 3s form a node blocked by the 100. If the 100 eats the nearer 3, that node's interval is no longer even a union of alive slimes. Instead, start from the single slime at moment $n - 1$ and split one slime per step. The proof of Observation 4 never uses where the merge happens or which slime blocks what, so read backwards it shows that splitting only ever helps the bystanders. Reaches therefore only grow, and distinct nodes only ever fuse together. This is much easier to maintain than a structure that breaks apart.

Let slime C split into A, the eater, and B, the eaten. They are adjacent and together occupy the original range of C, so no range sum anywhere changes; only the sizes at the position of C do. A node is determined by its sum and its two blockers, so a node's interval can only change if C was one of its blockers. Four things happen.

(a) Nodes containing C survive, with $inside$ one larger. Their sums and blockers are untouched, so they are still closed. Their members are still valid too. A member that ate C had weight at least $\text{sum}(C)$ at that point, hence enough for each half in turn, so it reaches exactly as far as before. The contribution of each such node grows by its own, so the running total gains $\sum own$ over all nodes containing C. These nodes form the ancestor chain of the node of C, at most about 52 of them. Alternatively, keep a Fenwick tree with range add and point query that holds each node's own spread over its interval, and read it at the position of C in a single query. That is what the model solution does.

(b) A inherits $f(C)$ with no computation. A is at least as heavy as B and adjacent to it, so A can immediately eat B and become what C was, in the position of C. Hence $f(A) \supseteq f(C)$. Also, $f(C)$ is still a closed interval after the split, so nothing inside it escapes, and $f(A) \subseteq f(C)$. A therefore takes over the slot of C in the structure.

(c) B costs one reach computation from scratch. It is the only slime whose reach is not already known after the split.

(d) Nodes abutting C may now grow. These are the nodes ending immediately to the left of C and those starting immediately to the right, that is, the nodes whose blocker was C. Nodes sharing an endpoint are nested, so each side is a chain of at most about 52 nodes. Keep, for each boundary position, a list of the nodes ending there and a list of the nodes starting there. For each such node, re-run its alternation, starting towards C and resuming from its current span rather than from scratch. This is legitimate because splits only help, so the old span is still achievable. Then:

- if the interval is unchanged, there is nothing to do;
- if the new interval already exists as a node, fuse the two, adding this node's own into that one and retiring it;
- otherwise, move the node to its new interval.

In every case, the running total is corrected by $own \times (\text{new } inside - \text{old } inside)$, which takes two Fenwick queries.

Observation 8: The re-runs in (d) cost $\mathcal{O}(n \log V \log n)$ in total over the sweep.

Examining a chain node and finding that nothing changed costs $\mathcal{O}(1)$, and there are at most about 104 such nodes per split. The work that is not $\mathcal{O}(1)$ is a node actually growing, and the first step of a growth need not double anything. The node's blocker was C, and the node may eat only the nearer half B and then stop, blocked by A. In the row $10, 100$ with the 100 split into $B = 5$ and $A = 95$, the node holding just the 10 grows to cover $10, 5$. This is a real move (re-key the interval, update the Fenwick trees and make two $inside$ queries, $\mathcal{O}(\log n)$ in all) that takes the sum only from 10 to 15, and every node in the chain abutting C can do the same in the same split.

We therefore charge the cost in two parts. The first phase of each abutting node's re-run, together with its bookkeeping if it moved, is charged to the split. That is about 104 nodes at $\mathcal{O}(\log n)$ each, which is $\mathcal{O}(n \log V \log n)$ over the whole sweep, since $104 \approx 2 \log_2 V$.

Every later phase is charged to the node, and these phases do compound. Write W_0 for the node's sum before the re-run and W_i for its sum after phase i. Phase 2 heads away from C and eats the blocker on that side, which is heavier than W_0. In general, phase $i + 2$, if it

makes progress at all, begins by eating whatever stopped phase i, so $W_{i+2} > W_{i+1} + W_i$ as in Observation 3, and the sum at least doubles every two such phases. A node that grows, moves or fuses is the same node with the same budget, and new nodes are born only at the start of the sweep and, at most one per split, for B. At most n nodes ever exist, and each pays for $\mathcal{O}(\log V)$ phases over its life. With the reach computations for B added, the total is

$$\mathcal{O}(n \log V \log n).$$

Charging a fresh $\log V$ budget to every abutting node, as though each re-run were a reach computation from scratch, would overcount this by a factor of $\log V$. Charging only B would undercount, since the first phases are real work.

Besides the two segment trees and the two Fenwick trees, the structure needs a lookup keyed by interval, so that a newly computed interval can be recognised as an existing node and fused into it.

The same structure answers Alice's question. "X can eat everyone" means that $f(X)$ is the whole row, that is, X belongs to the root node. In reverse time, reaches only grow and the root is the largest possible reach, so root membership is absorbing. Once an object is in the root, it stays there until it stops existing, and when it splits, A inherits its node by (b). Each of the $2n-1$ objects therefore joins the root at most once, and there are only $\mathcal{O}(n)$ membership changes in total. When a node fuses into the root, walk its member list and stamp the join moment of each member. Keep member lists singly linked with tail pointers, so that fusing concatenates them in $\mathcal{O}(1)$. No lazy accumulators or small-to-large merging are needed. The good moments of an object are then the prefix of its lifespan from its birth to the moment it joined the root, and this prefix is settled when the object splits.

This answers $q = 1$ in $\mathcal{O}(n \log V \log n)$ as well, one $\log n$ better than the parallel binary search of Subtask 2, and the two methods make good cross-checks for each other.

Time complexity: $\mathcal{O}(n \log V \log n)$

Subtask	Answer
1	870563975
2	105981414
3	465971153

Task 6: `Snowwabbits`

Authored by: Brian Lee (penguin133)

Prepared by: Zhao Yaoqi (zyq1810)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

A spell can change a wabbit to any size you like, and what the wabbit used to be does not matter. Casting X spells is therefore the same as throwing away X wabbits of your choice and adding back X wabbits of any sizes. The answer is the smallest X for which this works.

From the bottom of the tower to the top, the size either stays the same or drops by exactly one at each step, and the top wabbit has size 1. A collection of wabbits therefore forms a proper snowman if and only if every size from 1 up to the biggest wabbit is present at least once. Repeated sizes cause no trouble, because wabbits of equal size can be stacked directly on each other.

After throwing away X wabbits, look at the ones that are kept, find the sizes below the biggest one that are missing, and add back one wabbit of each missing size. This works if and only if at most X sizes are missing. If fewer than X are missing, the spare wabbits can be given size 1.

Observation 1: For a fixed number of kept wabbits, it is best to keep the smallest ones.

Suppose c wabbits are kept, and let $D(M)$ be the number of distinct input sizes that are at most M. If the biggest kept wabbit has size M, at least $M - D(M)$ sizes are missing, and this bound never decreases as M grows. Keeping the c smallest wabbits gives the smallest possible M, and it meets the bound exactly, since every input size up to M is then present. No other choice of c wabbits leaves fewer sizes missing.

Subtask 1

Additional constraints: $a[i] = 2 \cdot i$ for all $1 \le i \le n$

In this subtask the sizes are $2, 4, 6, \ldots$, so every odd size is missing. By Observation 1, keep the L smallest wabbits. The biggest kept wabbit then has size $2L$, and the sizes $1, 3, \ldots, 2L-1$ are all missing. That is L sizes to fill, but only $n - L$ wabbits were thrown away, so we need $L \le n - L$, that is, $L \le \lfloor n/2 \rfloor$. The answer is $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$.

For example, when $n = 4$ the sizes are $2, 4, 6, 8$. Throw away the 6 and the 8, and add back a 1 and a 3, for a total of 2 spell casts.

Time complexity: $\mathcal{O}(n)$

Subtask 2

Additional constraints: All $a[i]$ are distinct

With no repeated sizes, every kept wabbit covers a size that no other kept wabbit covers.

Suppose the biggest kept wabbit has size v. The finished snowman must contain each of the sizes $1, 2, \ldots, v$, so it uses at least v wabbits, and there are only n of them. Hence a wabbit of size v can be kept only if $v \leq n$.

Conversely, throw away every wabbit larger than n and keep the other c wabbits. Their sizes are distinct and lie in $[1, n]$, so at most $n - c$ sizes below the biggest of them are missing, and the $n - c$ wabbits thrown away are enough to fill them.

The answer is therefore the number of wabbits of size greater than n. This needs only a count, not a sort.

In the second test case of the example, $n = 5$ and the sizes are $10, 20, 30, 40, 50$. All of them are bigger than 5, so all five wabbits are thrown away and replaced by a 1, 2, 3, 4 and 5.

Time complexity: $\mathcal{O}(n)$

Subtask 3

Additional constraints: No additional constraints

Repeated sizes are now allowed, and they break the test from Subtask 2. Keeping a repeat costs a wabbit without covering a new size, so $v \leq n$ is still necessary but no longer sufficient.

In the third test case of the example, there are five wabbits, all of size 4, and every size is below $n = 5$, yet the answer is 3. Keeping all five leaves nothing to throw away, and so nothing to add back as a 1, a 2 and a 3.

Sort the wabbits by size. By Observation 1, the kept wabbits form a prefix of the sorted list, and it remains to choose where to cut. Walk up the sorted list and keep the wabbits one at a time. At each step we know how many sizes below the current wabbit are missing, and how many wabbits still lie ahead, all of which would be thrown away. Keep going as long as the missing

sizes do not outnumber the wabbits ahead. When this fails, stop and throw away everything from that point onwards. Concretely, after keeping the first i wabbits of the sorted array a,

$$\text{missing} = a[i] - (\text{number of distinct sizes among them}), \qquad \text{ahead} = n - i,$$

and the i-th wabbit is kept if and only if missing $\leq n - i$.

This test only gets harder as we move right, since the number of missing sizes never shrinks and the number of wabbits ahead never grows. A single left-to-right scan therefore finds the cut-off point, and the answer is the number of wabbits at or after it. All of the running time is in the sort.

In the fourth test case of the example, the sizes are $5, 2, 2, 1, 4$, or $1, 2, 2, 4, 5$ after sorting. Keeping everything up to the 4 leaves one missing size, 3, and one wabbit ahead, which is fine. Keeping the 5 as well still leaves size 3 missing, but no wabbits remain ahead to throw away. We therefore cut before the 5, throw it away and add back a 3, for one spell cast.

Time complexity: $\mathcal{O}(n \log n)$

Subtask	**Answer**
1	2101795
2	1715181
3	1339852

Task 7: `Spring Sunlight`

Authored by: Tan Yi Kai (kai824) [Guest problemsetter]

Prepared by: Siew Jing Hong (siewjh)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

The answer is the median of some batch, so it is one of the values in a. The condition "the overall quality is at least X" only gets easier as X decreases, since a partition with overall quality at least X also has overall quality at least X' for every $X' < X$. We can therefore binary search for the answer over the sorted distinct values of a. Each step asks whether the array can be split into k batches, each with median at least X.

Observation 1: A batch has median at least X if and only if at least half of its elements are at least X.

Consider a batch of size s that contains c elements that are at least X. Its median is the m-th largest element, where m is the smallest integer with $2m \geq s$. The m-th largest element is at least X exactly when at least m elements are at least X, that is, when $c \geq m$. Since m is the smallest integer with $2m \geq s$, this is equivalent to $2c \geq s$. No case analysis on the parity of s is needed.

This suggests marking every element with $+1$ if it is at least X and with -1 otherwise. Call a batch good if the sum of its marks is at least 0. By Observation 1, a batch is good if and only if its median is at least X.

Subtask 1

Additional constraints: $k = 2$

With $k = 2$ there is only one boundary to place. Sweep the boundary from left to right while maintaining the sum of the marks in the left batch. The sum of the right batch is the total sum minus the left sum. Accept X if some boundary makes both sums at least 0. Each check takes $\mathcal{O}(n)$ time.

Alternatively, the binary search can be skipped. Sweep the same boundary while maintaining the median of each side with an ordered multiset per side, and take the largest value of $\min(\text{left median}, \text{right median})$ over all boundaries.

Time complexity: $\mathcal{O}(n \log n)$

Subtask 2

Additional constraints: $1 \leq k \leq n \leq 500$

The binary search and the ± 1 marks carry over unchanged, and only the check has to be generalised from 2 batches to k batches.

Observation 2: If the array can be split into $f \geq k$ good batches, it can also be split into exactly k good batches.

Concatenating two adjacent good batches gives a batch whose sum is the sum of two non-negative numbers, so it is good too. Starting from f good batches, we can merge adjacent batches one pair at a time until k remain, and all of them stay good. It is therefore enough to compute the maximum number f of good batches that the array can be split into, and check whether $f \geq k$.

To compute f, let $P[0], P[1], \ldots, P[n]$ be the prefix sums of the marks, with $P[0] = 0$. Cut positions $0 = c_0 < c_1 < \cdots < c_f = n$ produce batches with sums $P[c_1] - P[c_0], P[c_2] - P[c_1], \ldots$, so every batch is good if and only if

$$P[c_0] \leq P[c_1] \leq \cdots \leq P[c_f].$$

Hence f is one less than the length of the longest non-decreasing subsequence of P that is forced to start at index 0 and end at index n.

This can be computed with dynamic programming. Let $\text{dp}[i]$ be the maximum number of good batches covering the first i elements, so that $\text{dp}[0] = 0$ and

$$\text{dp}[i] = 1 + \max\{\, \text{dp}[j] \;:\; j < i,\ 0 \leq P[j] \leq P[i] \,\}.$$

The condition $P[j] \geq 0$ says that index j can be reached, because a chain of good batches from index 0 to index j exists if and only if $P[j] \geq 0$. Along any chain P is non-decreasing, so $P[j] \geq P[0] = 0$, and conversely, if $P[j] \geq 0$, the first j elements alone form a good batch. No j qualifies if and only if $P[i] < 0$, and then i cannot be reached either, so $\text{dp}[i]$ is left undefined. An undefined $\text{dp}[n]$ means that the check fails. Fill the table for $i = 1, \ldots, n$ in this order. The check passes if $\text{dp}[n] \geq k$.

With $n \leq 500$, the inner maximum can be computed by scanning all $j < i$, so each check takes $\mathcal{O}(n^2)$ time.

In the second test case of the example, $a = [2, 3, 4, 4, 5, 5, 4]$ and $k = 3$. For $X = 4$, the marks are $[-1, -1, +1, +1, +1, +1, +1]$ and $P = [0, -1, -2, -1, 0, 1, 2, 3]$. The subsequence

at indices $0, 4, 5, 6, 7$ has values $0, 0, 1, 2, 3$, which gives $f = 4 \geq 3$, so $X = 4$ is feasible. For $X = 5$, the marks are $[-1, -1, -1, -1, +1, +1, -1]$ and $P[7] = -3 < P[0]$, so not even a single good batch covering the whole array exists, and $X = 5$ is infeasible. The answer is therefore 4.

Time complexity: $\mathcal{O}(n^2 \log n)$

Subtask 3

Additional constraints: No additional constraints

What Subtask 2 computes is a longest non-decreasing subsequence, and running the standard $\mathcal{O}(n \log n)$ algorithm for it inside the binary search, for $\mathcal{O}(n \log^2 n)$ in total, is already enough here. You may use any standard implementation.

This problem has extra structure, though, which makes an $\mathcal{O}(n)$ check possible. The standard algorithm maintains, for each length L, the value $g[L]$, defined as the smallest value on which a chain of L good batches starting at index 0 can end, over the indices processed so far. Initially only $g[0] = P[0] = 0$ is set, for the forced start at index 0. Merging two adjacent batches of a chain of $L + 1$ batches gives a chain of L batches ending on the same value, so g is non-decreasing in L. (Also $g[0] = 0 \leq g[1]$, since a single good batch ends on a non-negative value.)

To process index i, find the largest L with $g[L] \leq P[i]$. This is the best predecessor, so $\text{dp}[i] = L + 1$. Then write $P[i]$ into $g[\text{dp}[i]]$, which was previously either unset or strictly larger than $P[i]$. If not even $L = 0$ qualifies, that is, if $P[i] < 0$, then index i is unreachable and nothing is written, as in Subtask 2. Only reachable indices write, so every stored value is at least 0.

Since every mark is ± 1, consecutive prefix sums differ by 1. Instead of searching g from scratch at every index, we can therefore carry the answer forward. Maintain:

- $\text{cnt}[v]$, the number of lengths L with $g[L] = v$. Only $v \geq 0$ is ever stored, and a read at a negative v must return 0.
- S, the number of lengths L with $g[L] \leq P[i]$, where i is the index processed most recently, counting the entry written at i itself.

Since g is non-decreasing, the lengths counted by S are $L = 0, 1, \ldots, S - 1$. After index 0, we have $g[0] = 0$, $\text{cnt}[0] = 1$ and $S = 1$.

Now process $i = 1, 2, \ldots, n$ in order.

1. If P moved up by one, the counted range gains the value $P[i]$, so add cnt$[P[i]]$ to S. If P moved down by one, it loses the value $P[i-1]$, so subtract cnt$[P[i-1]]$ from S.
2. Now S counts the lengths with $g[L] \leq P[i]$, so dp$[i] = S$. If $S = 0$, index i is unreachable, and we move on to the next index.
3. Otherwise, write $P[i]$ into $g[S]$. If that entry was set before, decrement cnt at its old value. Then increment cnt$[P[i]]$ and increase S by one.

The old value was strictly greater than $P[i]$, which is why that length was not counted in S, and the new value is counted. Hence S again means what it should, and no other correction is needed. The check passes if dp$[n] \geq k$, that is, if the value of S read at $i = n$, before the write, is at least k.

Every step now takes $\mathcal{O}(1)$ time, so each check takes $\mathcal{O}(n)$, and the overall running time is $\mathcal{O}(n \log n)$, as in Subtask 1.

Time complexity: $\mathcal{O}(n \log n)$

Subtask	Answer
1	28194945047
2	16341821245
3	13874398697

Task 8: `StringStringString`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Zhao Yaoqi (zyq1810)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

Y is a concatenation of copies of y, so the score is the number of copies times $|y|$, and the number of copies is maximised by plain greedy matching. Scan X from left to right with a pointer into y. Advance the pointer whenever the current character of X matches the character it points at. Whenever the pointer runs off the end of y, count one completed copy and reset the pointer to the start of y.

By a standard exchange argument, matching every character as early as possible is optimal. You do not need to prove this in the contest.

Positions in S and T are 0-indexed throughout. The answer needs a 64-bit integer type. A single score can already exceed 2^{31}, since X has up to $K \cdot |S| \leq 5 \cdot 10^{12}$ characters, and the grand total runs over $|S| + |T|$ prefixes and all test cases, and reaches roughly $5.3 \cdot 10^{15}$ on the final subtask.

Subtask 1

Additional constraints: $1 \leq K \leq 500$ and $1 \leq |S|, |T| \leq 500$

Here X has at most $K \cdot |S| = 250\,000$ characters, so we can run the greedy over all of X once for each of the $|S| + |T|$ games. That is under $2.5 \cdot 10^8$ character comparisons per test case, and under $2.5 \cdot 10^9$ over all test cases, which is fine here.

Time complexity: $\mathcal{O}(K \cdot |S| \cdot (|S| + |T|))$

Subtask 2

Additional constraints: $1 \leq |S|, |T| \leq 500$

K can now be as large as 10^9, and X is too long to scan even once. We can still afford to rebuild everything from scratch for each of the $|S| + |T|$ prefixes, so the only new difficulty is the size

of K. The fix is to process X one whole copy of x at a time instead of one character at a time.

For Duck A, $x = S_m$ and $y = T$. The state of the greedy is a single number, the position of the pointer in T. For each j from 0 to $|T| - 1$, define

- $\mathrm{nxt}[j]$, the pointer position after feeding one whole copy of x with the pointer starting at j;
- $g[j]$, the number of characters of T matched while doing so.

Here the pointer wraps from $|T|$ back to 0, and one copy of x may complete several copies of T, so $g[j]$ can exceed $|T|$. Both arrays are computed together by walking x once while carrying $|T|$ independent pointers, in $\mathcal{O}(|x| \cdot |T|)$ time.

The array nxt is a functional graph on $|T|$ nodes with weighted edges, and we want the total weight of the walk of exactly K steps starting at node 0. Walk from node 0, marking nodes as visited, until a node repeats. This splits the walk into a tail of t steps with total weight A, and a cycle of c steps with total weight B. If $K \le t$, the answer can be read straight off the tail. Otherwise, the total number of matched characters is

$$W = A + \left\lfloor \frac{K - t}{c} \right\rfloor \cdot B + (\text{weight of the first } (K - t) \bmod c \text{ steps of the cycle}),$$

where the last term is found by walking the leftover steps one at a time. The walk takes $\mathcal{O}(|T|)$ time per prefix.

Note that W counts matched characters of T, not completed copies, and the walk usually stops partway through a copy of T. The score is $\lfloor W/|T| \rfloor \cdot |T|$, and this rounding must happen once, at the very end, and not once per copy of x.

For Duck B, $x = S$ and $y = T_m$. The repeating string is now S, so the nodes must be positions in S instead. Say we are at position p of S when the last character consumed was $S[p]$, so that the next available character is $S[(p + 1) \bmod |S|]$. From position p, greedily matching one whole copy of y consumes some $d[p] \ge 1$ characters of X and leaves us at position $(p + d[p]) \bmod |S|$. We start at $p = |S| - 1$ having consumed 0 characters, so that matching begins at index 0 of X.

To build $d[\cdot]$, we need to know, for a position p of S and a character c, how many characters must be consumed to reach the next occurrence of c. Call this $\mathrm{nxt}_S[p][c]$, counting cyclically and always at least 1. It is precomputed once per test case in $\mathcal{O}(26 \cdot |S|)$ time. With it, matching one copy of y takes $|y|$ table lookups rather than a character-by-character scan of X, and computing $d[\cdot]$ from scratch for one prefix costs $\mathcal{O}(|S| \cdot |T|)$. Scanning instead makes each lookup $\mathcal{O}(|S|)$ in the worst case, which pushes this subtask to around $3 \cdot 10^{10}$ operations per test case, several hundred times more for no benefit.

To build nxt_S, sweep backwards over S while maintaining, for each character, the smallest index strictly greater than the current one at which it occurs. If no such index exists, the next occurrence wraps around to the first occurrence of that character in S, and the distance gains $|S|$. Make one forward pass first to record the first occurrence of every character, and use it as the initial value for the backward sweep. Then take the distance modulo $|S|$, with 0 mapped to $|S|$. The distance is 0 exactly when the only occurrence of the character is at p itself, and the next occurrence is then one full lap away. A character that does not occur in S at all is marked dead.

This is again a weighted functional graph, now on the positions of S, but the question asked of it is the reverse. Instead of being given the number of steps and asked for the total weight, we are given a budget of $K \cdot |S|$ characters and asked for the largest number of steps. The tail and the cycle are found as before. Every weight is at least 1, so the cumulative cost strictly increases along the walk, the number of affordable steps is well defined, and a full cycle costs $B \geq c \geq 1$.

Do not forget that the budget may run out during the tail. With $K = 1$, the budget is only $|S|$ characters, while one step already costs at least $|T_m|$. For any prefix longer than $|S|$ the answer is therefore zero steps, and the walk never leaves the starting node. The walk goes as follows.

1. Walk the tail step by step, checking each time that the next step is still affordable. If the budget runs out, stop here.
2. Otherwise, take $\lfloor \text{remaining}/B \rfloor$ full cycles.
3. Then walk out the leftover steps one at a time for as long as they are affordable. There are fewer than c of them, since the remaining budget is now below B.

The score is the number of steps times $|T_m|$. If T_m contains a character that does not occur in S, no copy of y can ever be matched, and the score is 0 for this prefix and for every longer one.

In total, Duck A's half costs $\mathcal{O}(|S|^2 \cdot |T|)$ and Duck B's half costs $\mathcal{O}(|S| \cdot |T|^2)$, and both are at most about $1.25 \cdot 10^8$ here. If you would rather not think about cycles, binary lifting over the transition works just as well for this subtask, at the cost of a $\log K$ factor.

Time complexity: $\mathcal{O}(|S|^2 \cdot |T| + |S| \cdot |T|^2)$

Subtask 3

Additional constraints: No additional constraints

The cost of Subtask 2 comes from rebuilding the transition table for every prefix, not from the walk. The prefixes are nested, since S_{m+1} is S_m with one character appended, so we can update the table instead of rebuilding it.

For Duck A, the state kept for start position j is the current pointer $\mathrm{pos}[j]$ into T, together with $g[j]$. When the next character of S is appended, compare it with $T[\mathrm{pos}[j]]$. If they match, $\mathrm{pos}[j]$ advances by one, wrapping from $|T|$ back to 0; otherwise nothing happens. Separately, $g[j]$ is incremented on every match, including the ones that cause a wrap, because g counts matched characters of T, not completed copies of T. (Counting completed copies instead of characters also works, with a multiplication by $|T|$ at the end in place of $\lfloor W/|T| \rfloor \cdot |T|$. Flooring the character count once per copy of x does not work, because it discards the partially matched copy of T carried from one copy of x to the next.) This is $\mathcal{O}(1)$ per start position per appended character, so producing all $|S|$ tables costs $\mathcal{O}(|S| \cdot |T|)$ in total.

For Duck B, the state kept for start position p is the running cost $d[p]$, from which the current position $(p + d[p]) \bmod |S|$ is recovered. Appending the next character c of T adds $\mathrm{nxt}_S[(p + d[p]) \bmod |S|][c]$ to $d[p]$, using the table from Subtask 2, or marks $d[p]$ dead if c does not occur in S. Again this is $\mathcal{O}(1)$ per start position per appended character.

The walk is still $\mathcal{O}(|T|)$ per prefix for Duck A and $\mathcal{O}(|S|)$ per prefix for Duck B, so the task takes $\mathcal{O}(|S| \cdot |T|)$ per test case, around $2.5 \cdot 10^7$ operations. Binary lifting does not help here. Its tables would have to be rebuilt for every prefix, which multiplies all of the above by $\log K$ for no benefit over the walk.

Time complexity: $\mathcal{O}(|S| \cdot |T|)$

Subtask	Answer
1	21210195
2	36846713838068
3	5291509919272790

Task 9: `Topsy Turvy`

Authored by: Brian Lee (penguin133)

Prepared by: Seah E-Ket (JustKitkat)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

Alice's minions are never attacked, and Bob's minions never attack. The health of Alice's minions and the attack power of Bob's minions therefore never matter on their own, and $b[i]$ and $c[i]$ are only relevant as values that a swap could produce. The battle is decided by Alice's attack powers a and Bob's health points d alone, and a swap acts on only one of these numbers:

- swapping Alice's minion i replaces $a[i]$ by $b[i]$;
- swapping Bob's minion i replaces $d[i]$ by $c[i]$.

Swapping a minion twice returns it to its original state, so every minion is swapped at most once, and the answer is the number of minions we choose to swap. A swap is only worth doing if it raises an attack ($b[i] > a[i]$) or lowers a health value ($c[i] < d[i]$), and we call such a swap useful. Every other swap leaves the situation unchanged or makes it worse, so we discard it.

Alice has to kill all n of Bob's minions with her n minions, one each. She therefore needs a perfect matching in which every attacker's power is at least its target's health. After sorting both lists, this is possible if and only if the j-th smallest attack power is at least the j-th smallest health for every j.

Observation 1: Alice wins if and only if, for every threshold u,

$$\#\{i : a[i] \le u\} \le \#\{i : d[i] \le u\}.$$

If p attacks are at most u but fewer than p healths are, one of those weak attackers must face a minion whose health exceeds u, and it cannot kill that minion. Conversely, take u just below the j-th smallest health. At most $j-1$ healths are at most u, hence at most $j-1$ attacks are, and the j-th smallest attack is at least the j-th smallest health.

Define

$$\text{excess}(u) = \#\{i : a[i] \le u\} - \#\{i : d[i] \le u\}.$$

Alice wins exactly when $\text{excess}(u) \le 0$ for every u.

Observation 2: Every useful swap lowers excess by 1 on one interval and leaves it unchanged everywhere else.

Swapping Alice's minion i, where $b[i] > a[i]$, removes $a[i]$ from the multiset of attack powers and inserts $b[i]$. This lowers $\#\{j : a[j] \leq u\}$ by one for exactly those u with $a[i] \leq u < b[i]$, so the swap lowers excess by 1 on the interval $[a[i], b[i])$. Swapping Bob's minion i, where $c[i] < d[i]$, raises $\#\{j : d[j] \leq u\}$ by one for the u with $c[i] \leq u < d[i]$, so it lowers excess by 1 on the interval $[c[i], d[i])$.

Every useful swap is therefore an interval of cost 1, whichever side it comes from, and we can collect all of them into a single list of at most $2n$ intervals. The problem reduces to choosing the fewest of these intervals so that every point u is covered by at least excess(u) chosen intervals. Both excess and the coverage are constant between consecutive values that appear in the input, since excess changes only at a and d values and the coverage only at interval endpoints. It is therefore enough to check the at most $4n$ distinct input values.

Subtask 1

Additional constraints: $n \leq 1000$

This is the standard interval covering greedy. Sweep u from small to large. Whenever the coverage at u is short of excess(u), repeatedly take, among the intervals that contain u and have not been taken yet, the one that reaches furthest to the right. If at some point no unused interval contains u, Alice cannot win, and the answer is -1.

You do not need to prove this in the contest, but the exchange argument is short. At the leftmost point where we are short, every interval that could help starts at or before u. These intervals differ only in how far to the right they reach, and reaching further is never worse.

With $n \leq 1000$, the greedy can be implemented literally. At each of the $\mathcal{O}(n)$ coordinates, scan the interval list to find the best available interval.

In the third test case of the example, the only useful swaps are Alice's first minion, which gives the interval $[1, 4)$, and Bob's second minion, which gives $[4, 6)$. The function excess is 1 on $[1, 2)$ and on $[5, 6)$ and 0 elsewhere, so both intervals are needed and the answer is 2.

Time complexity: $\mathcal{O}(n^2)$

Subtask 2

Additional constraints: $c[i] = d[i]$ for all $1 \leq i \leq n$

Every swap on Bob's side is now a no-op, so the interval list contains only the Alice family $[a[i], b[i])$, and everything else from Subtask 1 carries over. However, n can now reach 10^6, so the sweep needs proper data structures instead of rescanning the interval list.

Sort the coordinates and maintain the following during the sweep.

- An event list with $+1$ at every $a[i]$ and -1 at every $d[i]$. Sorting it and accumulating during the sweep gives excess(u) at the current coordinate. All events at u, of both signs, are applied before u is checked, since both counts are inclusive.
- A max-heap `avail` of the right endpoints of intervals whose left endpoint has been passed and which have not been chosen. An interval $[l, r)$ is pushed when the sweep reaches l, and any entry with $r \leq u$ is discarded, since it no longer covers u.
- A min-heap `used` of the right endpoints of the chosen intervals. After popping the entries with $r \leq u$, its size is the number of chosen intervals that still cover u.

At each coordinate, do the following in order.

1. Expire the chosen intervals that have ended.
2. Apply the events at this coordinate to update excess.
3. Push the intervals that start here.
4. While the coverage is short of excess, pop the largest right endpoint from `avail` and take that interval. If `avail` runs empty while we are still short, report -1.

The coordinates to visit are the event positions, the left endpoints of the intervals and the expiry times of the chosen intervals. We can take the smallest of the three fronts at each step, or visit every distinct input value in order. Every interval and every event is pushed and popped a constant number of times.

Time complexity: $\mathcal{O}(n \log n)$

Subtask 3

Additional constraints: No additional constraints

The sweep from Subtask 2 still works once Bob's swaps are added back. A Bob minion with $c[i] < d[i]$ contributes the interval $[c[i], d[i])$, which has the same shape and cost as an Alice interval. Put both families into one list and run the sweep unchanged.

Time complexity: $\mathcal{O}(n \log n)$

Subtask	Answer
1	33825
2	21967553
3	36708776

Task 10: `Xiaoyang Ranking`

Authored by: Xin Anxiaoyang (Xiaoyang)

Prepared by: Xin Anxiaoyang (Xiaoyang)

Editorial written by: Ernest Kiew (Shor the Duck) (heavily AI-assisted)

Note

Most of what the zanes said does not matter. For each xiaoyang x, let $\text{lo}[x]$ and $\text{hi}[x]$ be the smallest and the largest rank that any zane gave x. If you place x at rank t, the largest offence caused by x is

$$\max(t - \text{lo}[x],\ \text{hi}[x] - t),$$

because $|t - s|$ is largest when s is one of the two extreme opinions. Every opinion strictly between $\text{lo}[x]$ and $\text{hi}[x]$ can be thrown away, and lo and hi can be computed in $\mathcal{O}(nm)$ while reading the input.

Now fix a candidate outrage k. The offence above is at most k exactly when

$$\text{hi}[x] - k \le t \le \text{lo}[x] + k.$$

A ranking therefore has outrage at most k if and only if every xiaoyang x is placed at a rank between $L[x]$ and $R[x]$, where

$$L[x] = \max(1,\ \text{hi}[x] - k), \qquad R[x] = \min(n,\ \text{lo}[x] + k)$$

are the first and last rank that x may take, clipped to the valid ranks. Each xiaoyang is now an interval and each rank is a point, and we have to give every interval a different point inside it.

Subtask 1

Additional constraints: mode $= 1$ in every test case and $1 \le n \le 1\,000\,000$

In this subtask we only need the minimum outrage k, not the ranking itself.

Increasing k only widens the intervals, so if some k works, every larger k works too, and we can binary search on k. The upper end of the search is safe, since at $k = n - 1$ every interval is all of $[1, n]$.

To check one value of k, think of rank t as a time slot, $L[x]$ as a release time and $R[x]$ as a deadline. Each xiaoyang is then a job of length one, and all n jobs have to be scheduled into the

n slots. This is the classic setting for the earliest deadline first greedy. Sweep $t = 1, 2, \ldots, n$. At slot t, insert every x with $L[x] = t$ into a min-priority queue keyed by $R[x]$, then pop the smallest deadline. The value k fails if the queue is ever empty or if the popped deadline is already in the past. An empty queue at slot t means that the remaining $n - t + 1$ xiaoyangs would have to share the $n - t$ later slots.

The optimality of earliest deadline first is standard, and you do not need to prove it in the contest. It follows from a short exchange argument. Suppose a valid assignment agrees with the greedy on ranks $1, \ldots, t-1$, but puts some other xiaoyang y at rank t and the greedy's pick x at a later rank t' (so y was in the queue too). Then $t' \le R[x] \le R[y]$, so y can take rank t' instead and the assignment stays valid.

The intervals do not need to be sorted again for every check. $L[x]$ grows with hi$[x]$, so a single sort by hi gives the release order for every k. Alternatively, bucket the xiaoyangs by $L[x]$ inside each check, which takes $\mathcal{O}(n)$. One check costs $\mathcal{O}(n \log n)$, so the binary search costs $\mathcal{O}(n \log^2 n)$.

Time complexity: $\mathcal{O}(nm + n \log^2 n)$

Subtask 2

Additional constraints: mode $= 2$ in every test case and $1 \le n \le 500$

Now we have to output the lexicographically smallest ranking with the minimum outrage. First run Subtask 1 to find k, and fix the intervals $[L[x], R[x]]$ for that k.

Fill the ranks $t = 1, 2, \ldots, n$ in order. At rank t, go through the unplaced xiaoyangs in increasing order of number, and take the first one that may sit at rank t and leaves everyone else matchable to the remaining ranks $t + 1, \ldots, n$. This is correct because any valid ranking that agrees with our earlier choices must put someone who satisfies both conditions at rank t. Nobody smaller than our pick can appear there, and our pick can be completed to a valid ranking.

Checking whether the rest is still solvable is the same interval matching as before, with the free ranks now being $t + 1, \ldots, n$. The checker from Subtask 1 answers it if the sweep starts at slot $t + 1$ and every unplaced xiaoyang is released no earlier than that.

There are n ranks, up to n candidates per rank and an $\mathcal{O}(n \log n)$ check for each candidate, which gives $\mathcal{O}(n^3 \log n)$. This is fine for $n \le 500$ but too slow for $n = 10^6$.

Time complexity: $\mathcal{O}(nm + n^3 \log n)$

Subtask 3

Additional constraints: mode $= 2$ in every test case and $1 \le n \le 1\,000\,000$

The greedy from Subtask 2 is still correct, but we cannot afford to test the candidates one at a time. Instead, we describe all valid candidates for rank t directly.

Consider the bipartite graph with the xiaoyangs on one side and the ranks on the other, and an edge whenever the rank lies inside the xiaoyang's interval. We are asking whether this graph has a perfect matching. By Hall's theorem, it does if and only if every group of xiaoyangs can use at least as many ranks between them as the group has members. There are 2^n groups, but because every xiaoyang's allowed ranks form an interval, the condition becomes much easier to check.

Observation 1: Non-empty intervals can be given distinct points if and only if, for every window of ranks, the number of intervals lying entirely inside that window is at most the length of the window.

For one direction, intervals inside a window have nowhere else to go, so there cannot be more of them than there are ranks in the window.

For the other direction, take any group of xiaoyangs and look at the ranks their intervals cover. These ranks form a few maximal runs of consecutive ranks. An interval is contiguous, so it cannot straddle a gap between two runs, and each interval of the group lies entirely inside exactly one run. By the window condition, each run contains at most as many of the group's intervals as it has ranks. Summing over the runs shows that the group can use at least as many ranks as it has members, which is what Hall's theorem requires.

We keep the invariant that the current state is solvable. It holds at $t = 1$ because k is feasible, and Observation 2 below shows that placing any valid candidate keeps it. Cut every unplaced xiaoyang's interval down to the free ranks $t, \dots, n$, so that lying inside $[t, b]$ just means $R[x] \le b$. By Observation 1, every window of free ranks satisfies the window condition, and in particular so does every window starting at t. These windows decide who may take rank t. For each $b \ge t$, define

$$\text{slack}(b) = (b - t + 1) - \#\{\text{unplaced } x : R[x] \le b\},$$

the number of free ranks in $[t, b]$ minus the number of xiaoyangs that must be placed by rank b. The window condition for $[t, b]$ says that $\text{slack}(b) \ge 0$. Since the state is solvable, this holds for every b. Call b tight if $\text{slack}(b) = 0$, and let b^* be the smallest tight b. It always exists, because $\text{slack}(n) = 0$.

Observation 2: The xiaoyangs that may take rank t are exactly the unplaced ones with $L[x] \le t$ and $R[x] \le b^*$.

Nobody else can take rank t. The xiaoyangs with $R[x] \le b^*$ fill the ranks $t, \dots, b^*$ with nothing

to spare, so rank t must go to one of them.

Conversely, placing any of them keeps the state solvable, which is the step the invariant relies on. Suppose we place such an x at rank t, and compare each new window $[t+1, b]$ with the old window $[t, b]$. The new window has one free rank fewer. If the old window had slack to spare, it absorbs the loss. If the old window was tight, then $b \geq b^* \geq R[x]$, so x was one of the xiaoyangs counted in it, and removing x cancels the lost rank. Windows starting at $t+2$ or later keep all their ranks and all their members, because $L[x] \leq t$ means that x was never inside them. Every window still satisfies the condition. The cut intervals also stay non-empty, because no unplaced $y \neq x$ has $R[y] = t$. Otherwise $\text{slack}(t) = 0$, so $b^* = t$, and y would be the only candidate. By Observation 1, the new state is solvable.

There is always at least one candidate, since some valid ranking agrees with our earlier choices, and whoever it puts at rank t qualifies. Also, no unplaced xiaoyang can have $R[x] < t$, so a xiaoyang is available as soon as $L[x] \leq t$.

The algorithm therefore processes $t = 1, \ldots, n$ in order. At each step it finds b^* and places the smallest-numbered available xiaoyang with $R[x] \leq b^*$.

To find b^*, keep a segment tree over the ranks storing

$$f(b) = b - \#\{\text{unplaced } x : R[x] \leq b\},$$

which is $\text{slack}(b)$ plus the constant $t-1$. At the start every xiaoyang is unplaced, so the initial values come from a prefix count of the values $R[x]$. Since slack is never negative and is zero somewhere, the minimum of f over $[t, n]$ is always exactly $t-1$, and b^* is the leftmost position where this minimum is reached. The tree therefore needs range add, and a query for the minimum together with its leftmost position. Placing a xiaoyang lowers the count for every $b \geq R[x]$, which is a range add of $+1$ on $[R[x], n]$. The suffix query can be avoided altogether. When t advances, add a huge constant at position $t-1$, so that expired ranks can never be the minimum, and b^* becomes the leftmost minimum of the whole tree.

To find the candidate, keep a second segment tree over the xiaoyang numbers. It stores $R[x]$ for each available unplaced xiaoyang and $+\infty$ for everyone else, with the minimum at every node. The leftmost position with value at most b^* is found by the standard descent, which goes left whenever the left child's minimum is at most b^* and right otherwise. Bucket the xiaoyangs by $L[x]$ so that each one is switched on at the right step, and switch the chosen one off after placing it.

There are $\mathcal{O}(n)$ updates in total and two $\mathcal{O}(\log n)$ queries per step, so building the ranking takes $\mathcal{O}(n \log n)$ on top of the binary search for k. The reported value can be far too large for a 32-bit integer type, so use a 64-bit one.

Time complexity: $\mathcal{O}(nm + n \log^2 n)$

Subtask	Answer
1	18333451
2	3194556751
3	4430667721355402774