Problem A. Simple Game

Problem idea: Geunyoung Jang (azberjibiou)

Preparation: Geunyoung Jang (azberjibiou)

First solver: 구재현 (Onsite, 1 min.), aeren (Open, 2 min.)

Total solved participant: 22 (Onsite), 57 (Open)

Subtask 1 (50 points)

Consecutive numbers are coprime. For each operation pair two adjacent integers.

Subtask 2 (50 points)

Assume $gcd(a, b) \neq 1$. Every element in the sequence can be expressed as $a + bx$ where x is an integer between 0 and $2n - 1$. Since $gcd(a, b)$ divides a and b, $gcd(a, b)$ divides $a + bx$, which means that $gcd(a, b)$ divides every element in the sequence. So you cannot proceed the operation.

Now let's think when $gcd(a, b) = 1$. By euclidean algorithm, $gcd(a + ib, a + (i + 1)b) = gcd(a + ib, b) = gcd(a, b) = 1$. This means that any two adjacent elements in the sequence are coprime. For ith operation, you need to choose $a + 2ib$ and $a + (2i + 1)b$. Note that the range of number can exceed int range.

Problem B. Clique partition

Problem idea: Junhyuk Song (SongC)

Preparation: Junhyuk Song (SongC)

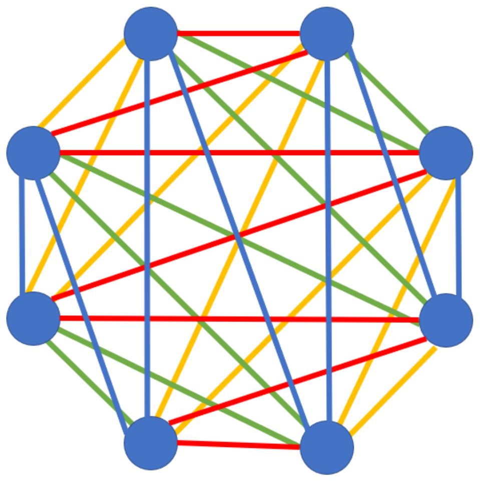
First solver: 장준하 *(Onsite, 15 min.), ncy09 (Open, 12 min.)*

Total solved participant: 14 (Onsite), 14 (Open)

Subtask 1 (20 points)

Consider the minimum of the answer for N. There are $\frac{N(N-1)}{2}$ edges and a tree has $N-1$ edges. So, if we can decompose a clique into $\lceil \frac{N}{2} \rceil$ trees it must be the minimum. Since $N \leq 8$ holds, we can do some paperwork and calculate answers by hands for every possible N.

Subtask 2 (80 points)



In fact, if N is an even number, we can decompose the clique to trees with out any duplication of edges. We can even split it into paths instead of general trees. The diagram at the top shows the way of decomposition. In odd case, just solve for $N-1$ and add a star graph around the node N.

Problem C. Monochrome Tree

Problem idea: Geunyoung Jang (azberjibiou)

Preparation: Geunyoung Jang (azberjibiou)

First solver: 구재현 *(Onsite, 10 min.), ainta (Open, 13 min.)*

Total solved participant: 20 (Onsite), 20 (Open)

Subtask 1 (20 points)

Try for every possible coloring. There are total 2^N colorings and solving the score takes $O(N^2)$ time so the answer can be solved in $O(2^N N^2)$ time.

Subtask 2 (20 points)

Let $dp[i][j]$ be the minimum score of a subtree of i where j vertices are colored black.

Then, we have the following recursive rule:

- Base case: For a leaf v, $dp[v][0] = dp[v][1] = 0$.
- Merge: For two children w_1, w_2 of v, we will shrink them to obtain one child w equivalent to both children. $dp[w][i] = min(dp[w_1][j] + dp[w_2][i-j])$
- Extend: For a vertex v with a single child w, we will add vertex v as a parent. $dp[v][i] = max(dp[w][i-1] + i - 1, dp[w][i])$

Implementing this will yield $O(N^3)$ complexity. Since merging two children with subtree size of A and B takes $O(AB)$ time, we can use tree optimization technique to get an $O(N^2)$ solution, but it is not required.

Subtask 3 (60 points)

Let's denote a good pair as a black vertex pair where one is an ancestor of another.

Lemma 1. *If there is vertex x which is colored white and its parent is colored black, exchanging color makes the score equal or smaller.*

Proof. Denote parent of vertex x as y. Let's count the good pairs where one of the vertices is x or y.

Number of good pairs where x or y is a descendant does not change.

Let z be a black vertex. For z to be a descendent of x, z should be in a subtree of x. For z to be a descendent of y, z should be in a subtree of y. Since subtree of y includes subtree of x, number of good pairs where x is an ancestor is smaller or equal than the number of good pairs where y is an ancestor is an ancestor.

□

Now we know that there exists an optimal solution where for every black vertex, every descendent is a black vertex. Let's count number of good pairs by fixing the ancestor. The number of good pairs will be sum of (subtree size -1) for every black vertex. The answer is $c_k =$ (sum of smallest k subtree size $-k$).

Problem D. Merging branches

Problem idea: *Geunyoung Jang* (*azberjibiou*)

Preparation: *Geunyoung Jang* (*azberjibiou*)

First solver: 구재현 (*Onsite*, *25 min.*), *ainta* (*Open*, *20 min.*)

Total solved participant: *9* (*Onsite*), *7* (*Open*)

We will use word extend as same meaning with widen. We define i-th branch and $i+1$-th branch is connected if after extending each branches, i-th branch and $i+1$-th branch intersects. Note that we can merge all branches if and only if i-th branch and $i+1$-th branch is connected for all $1 \le i < N$ after branch extension.

Subtask 1 (15 points)

Let's try to solve a problem whether we can merge all branches with cost K. Let's look from left to right. Extending leftmost branch with length K to the right side should be optimal. Assume that we have extended first to $i-1$-th branches optimally. Let's think about how we should extend the i-th branch.

If $l_i - r'_{i-1} > K$, we cannot merge all branches. If $l_i \le r'_{i-1}$, extending length K to the right side is optimal, which means $l'_i = l_i$, $r'_i = r_i + K$. Otherwise, extending i-th branch left by distance between $i-1$-th branch and i-th branch and extending right as far as possible is optimal. This means $l'_i = r'_{i-1}, r'_i = r_i + K - (l_i - l'_i)$.

This kind of greedy approach can be done in $O(N)$ time.

If $K = 20$, we can merge all branches by simply extending every branches with length 20 to the right. Since $l_i – r_{i-1} \le 20$, every adjacent branch intersects.

For each query, brute-forcing K from 1 to 20 gives us an answer with time complexity $O(20N)$. The total time complexity will be $O(20NQ)$.

Subtask 2 (15 points)

We use the same greedy approach in subtask 1. Instead of using brute force on K, we can use binary search on K. Since the answer is bounded below $max(A_i)$, the answer will be $O(NQlogX)$ where $X = max(A_i) = 10^9$.

Subtask 3 (70 points)

Denote a_i as the distance between ith branch and $i+1$th branch. Strictly speaking, $a_i = l_{i+1} - r_i$.

Lemma 1. *We can merge all branches with cost K if and only if for all* $1 \le l \le r \le N-1$,

$$\sum_{i=l}^{r} a_i \le (r-l+2)K$$

Proof. ($\rightarrow$) Denote x_i as the length extended left from i-th branch. Then $K - x_i$ is the length extended right from i-th branch.

Equivalent condition for merging all branches are the following.

- $0 \le x_i \le K$

- $K - x_i + x_{i+1} \geq a_i$

Sum up the second inequality for $l \leq i \leq r$. We obtain $(r-l+1)K - x_l + x_{r+1} \geq \sum_{i=l}^{r} a_i$. By using $0 \leq x_i \leq K$, we obtain $(r-l+2)K = (r-l+1)K - 0 + K \geq (r-l+1)K - x_l + x_{r+1} \geq \sum_{i=l}^{r} a_i$.

($\leftarrow$) Let's prove by contradiction. By using the greedy algorithm at subtask 1, assume that we cannot merge between p-th branch and $p+1$-th branch. This means that when we determined from x_1 to x_p, $a_p > (K - x_p) + K = 2K - x_p$.

Let s be a maximum integer among integers where $x_s = 0$. Since $x_1 = 0$, there exists such integer s. For $s \leq i < p$, we connected ith branch and $i+1$th branch. Also none of x_i is zero. So $K - x_i + x_{i+1} = a_i$ holds. By summing up, we get $(p-s)K + x_p = \sum_{i=s}^{p-1} a_i$ (Notice that $x_s = 0$). By assumption, we have $(p-s+2)K \geq \sum_{i=s}^{p} a_i$. By subtracting two (in)equality, we get $2K - x_p \geq a_p$, which is a contradiction.

$\square$

$\sum_{i=l}^{r} a_i \leq (r-l+2)K$ is equivalent to $\frac{\sum_{i=l}^{r} a_i}{r-l+2} \leq K$. Denote $f(l,r) = \frac{\sum_{i=l}^{r} a_i}{r-l+2}$.

Let $dp[s][e]$ =minimum cost to merge branches from s-th to e-th ($s < e$). Then we have the following recursive rule.

- Base case: $dp[s][s+1] = f(s,s)$
- Recursive case:

$$dp[s][e] = \min_{s \leq l \leq r < e} f(l,r) = \min(\min_{s \leq l \leq r < e-1} f(l,r), \min_{s+1 \leq l \leq r < e} f(l,r), f(s, e-1))$$

By dynamic programming, we can solve $dp[s][e]$ for all $1 \leq s \leq e \leq N-1$ in $O(1)$. By this, we can solve each query in $O(1)$. Since there are $O(N^2)$ dp states, we can solve the answer in $O(N^2 + Q)$.

Problem E. Binary sequence and queries

Problem idea: *Geunyoung Jang* (*azberjibiou*)

Preparation: *Geunyoung Jang* (*azberjibiou*)

First solver: 최은수 (*Onsite, 76 min.*), *arnold518* (*Open, 79 min.*)

Total solved team: *8* (*Onsite*), *6* (*Open*)

Subtask 1 (10 points)

For query of type 1, simply change a_p into t. For every query of type 2, brute force all possible segments $[s, e]$ where $l \le s \le e \le r$. Since there are $\frac{(r-l+1)(r-l+2)}{2}$ possible segments and for each segment we can solve longest consecutive 0s' and 1s' in $O(l-r)$, we can solve each query in $O(N^3)$, which leads to $O(QN^3)$ solution.

Subtask 2 (20 points)

If $x = 0$ and $y = 0$, there is no such segment. Let's think when $x \neq 0$ or $y \neq 0$. Define each $f(s, e), g(s, e)$ be the longest consecutive 0s' and 1s' of segment $[s, e]$.

Lemma 1. *For* $s_1 \le s_2 \le e_2 \le e_1$, $f(s_1, e_1) \ge f(s_2, e_2)$ *and* $g(s_1, e_1) \ge g(s_2, e_2)$.

Proof. Trivial. □

Lemma 2. *If two segment* $[s_1, e_1]$ *and* $[s_2, e_2]$ *differ by at most* 1, $f(s_1, e_1)$ *and* $f(s_2, e_2)$ *differ by at most* 1.

Proof. Trivial. □

Lemma 3. *Let longest consecutive 0s' and 1s' of segment* $[l, r]$ *be* x_0, y_0. *There exists a segment* $[s, e]$ *that satisfies the given conditions if and only if* $x \le x_0$ *and* $y \le y_0$.

Proof. ($\leftarrow$) Since $l \le s \le e \le r$, Lemma 1 leads to $x = f(s, e) \le f(l, r) = x_0$, $y = g(s, e) \le g(l, r) = y_0$.

($\rightarrow$) Let k_1 be the minimum k such that $f(l, k) \ge x$ and $g(l, k) \ge y$. Since $f(l, r) = x_0 \ge x$ and $g(l, r) = y_0 \ge y$, there exists such k. We can easily show that if $a_{k_1} = 0$, $f(l, k_1) = x$ and if $a_{k_1} = 1$, $g(l, k_1) = y$ by minimality of k_1 and Lemma 2.

WLOG $a_{k_1} = 0$ and $f(l, k_1) = x$. This means that suffix of $a_l a_{l+1} ... a_{k_1}$ has x consecutive zeros. This leads to $f(k_1 - x + 1, k_1) = x$, $g(k_1 - x + 1, k_1) = 0$.

Let k_2 be the maximum k where $f(k, k_1) \ge x$ and $g(k, k_1) \ge y$. For $k \le k_1 - x + 1$, $f(k, k_1) = x$. This shows that $f(k_2, k_1) = x$. Since $g(k_1 - x + 1, k_1) = 0$ and $g(l, k_1) = y_0 \ge y$, $g(k_2, k_1) = y$ by Lemma 2.

We have found a segment $[k_2, k_1]$ where $f(k_2, k_1) = x$ and $g(k_2, k_1) = y$.

□

$f(l, r)$ and $g(l, r)$ can be solved in $O(r - l)$, to decide if a segment that satisfies the given condition exists. If there exists such segment, we can find k_1 and k_2 by sweeping k_1 from l to r, then sweeping k_2 from k_1 to l. This can be done in $O(r - l)$. Since every query can be solved in $O(N)$, the whole problem can be solved in $O(QN)$.

Subtask 3(30 points)

First, find k_1 for all queries by offline query. Then, find k_2 for all queries by offline query. Separate the query into x and y. Then, sort each query by x and y. Sweeping using ordered set will end in an $O((N + Q)logN)$ solution.

Subtask 4(40 points)

Let's define a segment tree where node for $[l, r]$ stores the following information.

- $f(l, r)$, $g(l, r)$
- number of consecutive same element at left / right
- length of interval

Since we can merge two adjacent nodes, we can make a segment tree. Making a segment tree takes $O(NlogN)$ time. For each query, solving $f(l, r)$ and $g(l, r)$ costs $O(logN)$ time. We can solve k_1 and k_2 in subtask 2 by using binary search. Since each value requires $O(logN)$ time, we can solve k_1 and k_2 in $O(log^2N)$ time.

Overall we can solve the problem in $O(NlogN + Qlog^2N)$ time. Using binary search in segment tree will reduce the time complexity into $O(NlogN + QlogN)$.

Problem F. Tsunami

Problem idea: Hongyoon Mun (mhy908)

Preparation: Hongyoon Mun (mhy908)

First solver: N/A (Onsite), arnold518 (Open, 183 min.)

Total solved team: 0 (Onsite), 1 (Open)

Subtask 1 (11 points)

Make a lattice graph of size XY. Edges layed horizontally at $y = i$ will have weight c_i, and edges layed vertically will have weight calculated with the arrangement of obstacles. By applying dijkstra. the time complexity will be $O(XYlogXY)$.

Subtask 2 (7 points)

Construct a DP.

Let $DP[i][j]$ be minimum time to go to point (i, j).

Then $DP[i][j] = min_{0<=i'<=X+1}(DP[i'][j-1] + cross(i', j) + c_{j-1} \times |j - j'|)$, where $cross(i', j)$ is the time crossing an obstacle placed at (i', j).

If (i, j) is an evacuation spot, Also consider the r value of that point.

By naively calculating the above DP, we get $O(X^2Y)$ which is not an improvement compared with Subtask 1. But we can apply simple optimizations to this DP.

Note that most of the DP values don't change as i increases. It only changes when there is an event at that particular coordinate.

When there is an evacuation spot at (i, j) which has value r, we can consider $r + c_j \times |i - i'|$ for a new candidate of $DP[i'][j + 1]$. If there is an obstacle, we just use the original equation. With this optimization, we get $O(XY + NX + MX^2)$.

Subtask 3 (10 points)

We now need to make further optimizations for the obstacle case.

The reason why it takes $O(X^2)$ time is because we tried all possible starting x-coords and finishing x-coords for moving horizontally. Instead, we can just traverse the 1 dimensional array $DP[*][j]$ for fixed j and maintain prefix/suffix minimums of $DP[i][j] - c_j \times i/DP[i][j] + c_j \times i$ respectively. With these values, we can consider horizontal moving at obstacle coordinates achieving $O((Y + N + M)X)$ time complexity.

Another way to solve this subtask is observing that there is always an answer that only move horizontally at evacuation points or the endpoints of obstacles. We can prove this easilly with the fact that c is a nondecreasing sequence.

Therefore, we can just think that there is a new evacuation point at endpoints(endpoints will be $s - 1$, $e + 1$ if the obstacle is ranged at $[s, e]$) of obstacles having r value of $DP[s - 1][j]$ and $DP[e + 1][j]$, and modify the original DP equation to $DP[i][j] = DP[i][j-1] + cross(i, j-1)$.

Subtask 4 (23 points)

In this subtask, we don't have obstacles. Therefore, we only need to consider horizontal movement at given evacuation points.

Note that at this subtask, all optimal paths start at some evacuation spot, move horizontally, and then move vertically until we reach $y = Y$. More specifically, The answer at (x, Y) is $min_{1<=i<=N}(r_i + c_{q_i} \times |p_i - x|)$. If we split the absolute value term into 2 pieces we get a minimum of $min_{1<=i<=N, x<=p_i}(r_i + c_{q_i} \times (p_i - x))$ and $min_{1<=i<=N, p_i<x}(r_i + c_{q_i} \times (x - p_i))$.

The above expressions can be solved by a convex hull structure, such as a li-chao tree with a time complexity of $O(NlogX)$.

Subtask 5 (25 points)

In this subtask, all c values are identical. We can solve this by maintaining the lower hull of line segments (may be discontinuous) either having a slope of either c_1 or $-c_1$. We won't provide further explanation, but we can handle the structure using data structures such as segment tree etc. The time complexity can be either $O(MlogX)$ or $O(Mlog^2X)$.

Subtask 6 (24 points)

In this subtask, there is only 1 evacuation spot. This subtask was originally designed for other creative solutions, but we couldn't find solutions that can solve only this paricular subtask.

Subtask 7 (13 points)

For the full task, we need a data structure to handle

1. range addition

2. range line insertion

and maintain the minimum y-values for all x-values.

There are 2 possible ways to construct this structure.

A straight-forward way is to modify a li-chao tree to handle those instructions. Range line insertion is easy, just find $O(logN)$ appropriate subtrees and do original line insertions on those subtrees. Time complexity will be $O(log^2N)$.

Range addition can be implemented with lazy propogation, but additional updates need to be made. We define a "flush" mechanism as updating a line written in a node to both to the left subtree and right subtree, and filling the original node with infinity. This will take $O(logN)$ time, but flushing does not change the minimum values for all x's.

Therefore if we combine flushing and lazy propogation, we can solve the full problem in $O((N+M)log^2X)$. Further explanation is written at https://codeforces.com/blog/entry/86731.

Another approach needs additional observation that when we update a line, no more than one continuous range gets covered by that line. One can prove it using that c is a nondecreasing sequence. Therefore if we already know each x-coordinates' minimum y values, we can apply binary search to find the appropriate range, and cover the range by the given linear function. Now, this can be implemented by lazy segment tree in $O((N+M)log^2X)$. This approach can be improved by taking out the $logX$ term in binary search using internal binary search in the segment tree leading to $O((N+M)logX)$, but not required to solve the full task.

Problem G. Remix

Problem idea: Issa Hazem

Preparation: Issa Hazem

First solver: 최은수 (Onsite, 166 min.), ainta (Open, 62 min.)

Total solved team: 4 (Onsite), 4 (Open)

Notice how any new number we make can only be smaller than or equal to $\max(t)$, so the optimal answer can't be larger than the $\max(s)$. Let e be the subset we do the final operation on, then the final number we end up with is equal to $\max(e) - \min(e)$. We can make this value greater by either increasing the value of $\max(e)$ or decreasing the value of $\min(e)$. But as we have seen any operation done on a subset may only decrease its max value, so the max value, optimally, will be a number from the original set. So our only option is to try and decrease the value of $\min(e)$.

Now we just need to solve the following problem: we have a multiset s, and we can choose any subset t and replace it with one element equal to $\max(t) - \min(t)$. Minimize the minimum element in the final set. Note that we don't have to end up with one integer like in the original problem and therefore we may ignore the scheme of choosing a subset altogether and only focus on choosing the two integral values, $\min(t)$ and $\max(t)$, a subset of size two if you must.

Let's think of all the non-empty subsets of s, and let c_i be the sum of elements of the i-th subset. The lower bound for the value of $\min(e)$ is the lowest value for $|c_i - c_j|$ where $i \neq j$. And it turns out that there is a construction to achieve this value. Let A and B be the i-th and j-th subsets for which $|c_i - c_j|$ is minimum, if there's any element that occurs in both sets remove it from both and it will not affect $|c_i - c_j|$. Now do the following until one of the sets becomes empty:

1. Choose any element x from A, and choose any element y from B.
2. Do an operation with the subset $\{x, y\}$.
3. Erase x and y from A and B respectively.
4. Insert $x - y$ to A if x is greater than y, and insert $y - x$ to B otherwise.

Notice how $|\sum A - \sum B|$ is equal to $|c_i - c_j|$ after each operation. In the end, we will be left with exactly one integer with the value $|c_i - c_j|$ in one of the sets. Note that we must be left with exactly one integer, not two or more because otherwise, it would contradict that $|c_i - c_j|$ is minimum as we could have removed the extra element(s) and achieved a smaller difference.

To find c_i and c_j, we can use knapsack dp but this only works for sufficiently small values of n and s_i. To achieve a better result, we need to notice that the absolute minimum value for $|c_i - c_j|$ is 0 which only happens when $c_i = c_j$. Since the values in the multiset are bounded by some maximum value, let's say M, we can show that any multiset of size larger than some value C must contain two disjoint subsets with equal sum. In a multiset of size C, there are a total of 2^C subsets and the maximum sum for any subset is CM. As a result, if $2^C > CM$, there must exist two disjoint subsets having equal sum by the pigeonhole principle. So we just need to choose the minimum C satisfying the mentioned equation then compute all the subset sums of any C elements in $O(2^C)$.

Time Complexity: $O(n + 2^C)$ where C is the minimum multiset size such that there exist two subsets with equal sum ($C \leq 22$ for $s_i \leq 10^5$).

Problem H. Good night

Problem idea: *Junhyuk Song* (*SongC*)

Preparation: *Junhyuk Song* (*SongC*)

First solver: *N/A* (*Onsite*), *N/A* (*Open*)

Total solved team: *0* (*Onsite*), *0* (*Open*)

Subtask 1 (11 points)

To construct a naive polynomial algorithm, we should observe some properties of streetlights.

- If a streetlight goes out and doesn't turn back on immediately, it will remain off permanently.
- Over time, the area of illumination along the line can only decrease.
- If any streetlights don't go out for a time interval of T, then that state will be maintained permanently.

The proofs have been left out, to be completed as an exercise by the reader. By these facts, we can construct an $O(N^3)$ simulation algorithm. For each period, examine every light to determine if it turns back on immediately after going out. Testing a single light requires $O(N)$ time, so the time complexity for each period is $O(N^2)$. Based on the third property, at least one light will go out during each period, except for the last one. So there are at most N periods and the overall complexity of simulation is $O(N^3)$.

Subtask 2 (12 points)

We can further optimize the simulation algorithm from subtask 1. By using a segment tree, we can efficiently maintain the state of the line. Specifically, the tree will store the number of lights illuminating each segment of the line. To check whether a position is lighted by streetlight at a specific time, we can perform queries on the segment tree. If the number of lights that go out at that time is denoted as m, then the number of required queries is $O(m)$. Notably, the cumulative number of lights going out in a period is $O(N)$. As a result, the complexity of the simulation is improved to $O(N^2 logN)$.

Subtask 3 (25 points)

By the constraint, there are no two lights going out at the same time. In our simulation, checking each light individually is inefficient. Instead, we should focus on the lights which are remained off permanently when it goes out. If we have the set of such lights denoted as S, the following method works correctly:

1. Pop the light that goes out earliest from S.
2. Remove it and update the line's state.
3. Update S by adding lights which newly satisfy condition due to elimination of a light.
4. Repeat 1-3 until S is empty.

Notice that we don't have to delete any light on update at (3) since the area of illumination can only decrease. If we can maintain S with priority queue and find new lights added to S fast enough, the whole problem can be solved efficiently.

To detect new lights, consider the condition for x to be in S. It is equivalent to the following: There exists a point lying on the path between origin and x which is either not illuminated by any light or is solely illuminated by x.

Now we can optimize the algorithm with a segment tree. Construct a minimum segment tree with lazy propagation. Begin by recording the number of lights illuminating each segment, then determine the initial S by just checking for every lights. Proceed as mentioned:

1. Remove a light from S, then decrement the corresponding segment in the tree using lazy propagation.
2. Check the tree's minimum value. If it's zero, a point remains unilluminated. All lights behind this point can be added to S. (To optimize, pinpoint the leftmost zero or ignore previously detected points. Ignoring can be done by assigning a sufficiently large value to the point.)
3. Similarly, detect points illuminated by just one light and assess them. To ascertain the exact index of a light, construct another tree storing the sum of light indices. With only one light illuminating a point, its identification is possible through a point query on this secondary tree.

Since the light count for a specific point decreases, every unit segments are detected at most twice. The algorithm requires $O(N)$ segment tree queries and the overall time complexity is $O(NlogN)$.

Subtask 4 (52 points)

$O(Nlog^2N)$ **Solution**

Let's try to apply the solution of subtask 3. The only difference between subtask 3 and the full task is existence of multiple lights which go out simultaneously. Most parts of the algorithm are reusable, but we have to revise the condition to be included in S. For the full task, let's try to modify the condition like the followings.

- There exists a point lying on the path between origin and x which is not illuminated by any light.
- There exists a point lying on the path between origin and x which is only illuminated by lights whose value of A is same with x.

This isn't a strict equivalence, as there might be a light x meeting the second criterion but could potentially be turned on later. Think of scenarios where a light, sharing the same A value as x, re-lights post-extinguish, lighting the route to x. Pinpointing a strict equivalent condition for S is challenging. However, if we can identify at least one of the fastest extinguishing lights from S, the problem becomes solvable. (Consider Dijkstra algorithm)

Modify the second criterion to: There exists a point on the path between the origin and x which is illuminated only by lights sharing x's A value and are **behind** x.

Upon any light extinguishing, at least one light satisfies this modified condition. Focus on the leftmost light extinguishing simultaneously.

The problem can now be solved using a complex data structure. Most parts of the algorithm mirror subtask 3. After each deactivation, we now need to pinpoint lights meeting the modified conditions. When focusing a line's unit segment, consider events triggering updates to the PQ:

1. The number of distinct A values illuminating the segment becomes one.
2. The segment, already lit by a singular A value, sees a shift in the leftmost illuminating light's position.
3. All illuminating lights are removed.

For a segment tree storing lights, each node should contain a set of pairs representing lights illuminating the entire segment in the form (A_i, X_i). During initialization, assign pairs to nodes for all lights, with each light setting $O(logN)$ pairs.

For light elimination, tree maintenance is simplified by erasing $O(logN)$ pairs from sets. Our focus now shifts to updates. Nodes with more than two distinct A_i, values will always remain lit, regardless of the number of lights extinguishing simultaneously. Thus, such nodes need no attention. An update for a leaf node demands $O(logN)$ within our structure. (Just an application of a point query on a segment tree)

For case 1 and 3, they happen only one time for a node. So, just use point update for all leaves under it. Then total number of update is $O(NlogN)$ and total time complexity is $O(Nlog^2N)$.

For case 2, we will focus on each node in the same way to case 1. Denote the minimum of X_i for a node as v. As time progresses, v ascends and v affects to lights which are further left than v. So if v becomes to v_1 to v_0, we can just update for each nodes in $(v_0, v_1]$. The whole complexity about this case is bounded to $O(Nlog^2N)$.

Finally, we can implement all operations we need in amortized $O(Nlog^2N)$.

$O(NlogN)$ **Solution by Sjimed**

Let's think about that the point x is whether reachable or not, at the time t. We may assume light $1, 2, ..., n$ is not turned off right before t and $A_1 \equiv A_2 \equiv ... \equiv A_k \equiv t \pmod T$ $(1 \le k \le n)$. We just want to compute the minimum $x > 0$ such that x is not reachable at t.

In this perspective, the light $i \le k$ indicates that if we can reach $\max(L_i, X_i)$, we also can reach R_i. For the lights that $i > k$, we can reach R_i if we can reach L_i. Therefore, the problem could be changed to finding the minimum x which is not covered by the intervals: $[\max(L_i, X_i), R_i]$ for $1 \le i \le k$, and $[L_i, R_i]$ for $k < i \le n$.

So basically the light i covers $[L_i, R_i]$, but $[\max(L_i, X_i), R_i]$ if $A_i \equiv t \pmod T$(the interval $[L_i, \max(L_i, X_i))$ is erased). If currently there is only one $0 \le t < T$ such that $\exists i$, $A_i = t$ and $x \in [L_i, \max(L_i, X_i))$, x would be light off when the time $\pmod T$ becomes A_i.

Therefore, if we could manage the number of times that cover the point, we could manage the set S.

Currently, our programming task is:

1. Count the number of the times that $[L_i, \max(L_i, X_i))$ covers the point and find the minimum
2. If there exists x that the count value is at most 1 and there is no turned-on light that $x \in [\max(L_i, X_i), R_i]$, update x in S and erase x.

By using the minimum segment tree with lazy propagation, we can get the point that the count value is less or equal to 1. We also can eliminate all $[\max(L_i, X_i), R_i]$'s of turned-on lights and delete a point after putting into S simply adding 2(this makes all the points in the interval never come out).

Since all lights are turned off according to the order of X_i among the lights having the same A_is, we can pre-compute the changing intervals using data structures such as std set. All lights are turned off at most once, so the total update query of the segment tree is $O(N)$.

Finally, we just manage the queries some light is actually turned off or some point is not reachable anymore, using a priority queue.

Total time complexity of the solution is $O(N \log N)$