MITIT 2025-2026 Winter Contest Editorials: Advanced Individual Round

The MITIT Organizers

December 7, 2025

1 MIT and TIM

Problem Idea: Yifan Kang
Problem Preparation: Yifan Kang
Analysis By: Yifan Kang

Observation 1.1. *The operation can be viewed as swapping the outer non I characters around a fixed central* `I`*. Thus, the positions of all* `Is` *never change.*

For every index j with $S_j =$ `I` and non I neighbors, we draw an edge between $j-1$ and $j+1$. Then we can swap elements that are connected by edges. Thus, we can perform any permutation within the same connected component by composing swaps.

So, within each connected component, the number of `Ms` and `Ts` is fixed, but their order can be arbitrary.

Creating k copies of `MITIT` inside this component requires at least k `M`'s and $2k$ `T`'s. Thus,

$$k \le M, \quad k \le \left\lfloor \frac{T}{2} \right\rfloor .$$

So any component can produce at most

$$k_{\max} = \min\left(M, \left\lfloor \frac{T}{2} \right\rfloor\right)$$

occurrences.

Moreover, this is achievable because we can permute the `Ms` and `Ts` arbitrarily.

We can scan left to right in linear time to construct the graph, and count the number of `Ms` and `Ts` in each component takes linear time. The total time complexity is $O(N)$.

2 Snakey Beavers

Problem Idea: Ray Law
Problem Preparation: Bergüzar Yürüm
Analysis By: Ray Law

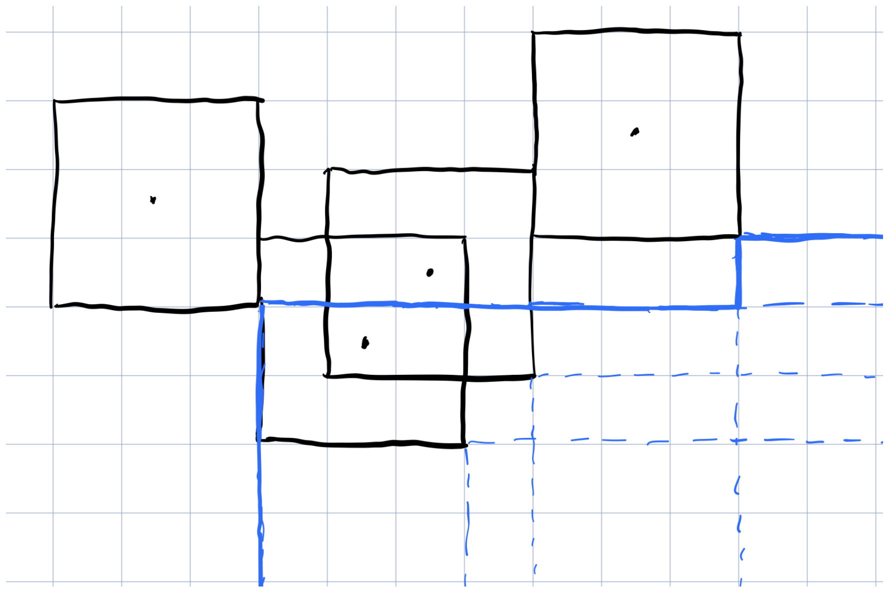
Define an inversion as two baby beavers i and j where $x_i < x_j$ and $y_i > y_j$. Define the separation of an inversion $f(i,j)$ as $\min(|x_i - x_j|, |y_i - y_j|)$.

Claim 2.1. *If m is the maximum separation over all inversions, the minimum time required is $\frac{m}{2}$.*

Proof. Let k be the minimum time required. Draw a square with side length $2k$ centered at each baby beaver. The baby beavers can form a snake if there is a path that moves only up and right and touches every square.

For every inversion (i,j), the squares must share an x or y-coordinate. Therefore, $k \geq \frac{f(i,j)}{2}$. This means $k \geq \frac{m}{2}$.

To construct a path with $k = \frac{m}{2}$, draw rays going down and to the right from the bottom right corner of every square. Take the leftmost and topmost of all $2N$ rays to get a path. This path does not go below any bottom right corner, so it does not go below any square. If the path went above any square, that would mean the maximum separation from the corresponding baby beaver to some other baby beaver is greater than m, contradiction. □



The problem reduces to finding the maximum separation of an inversion. For partial credit, check every pair of baby beavers to calculate m, the maximum separation over all inversions.

For full credit, we can use either binary search or two pointers.

Solution 1: Binary Search

Sort the baby beavers by x and binary search on the answer. Suppose we are testing the separation k. To find an inversion with separation $> k$, sweep the baby beavers by x. If i is the current baby beaver, maintain the maximum y_j for all j where $x_j < x_i - k$. If this maximum is greater than $y_i + k$, we found an inversion with separation $> k$.

Time Complexity: $O(N \log N)$

Solution 2: Two Pointers

Let (i, j) be an inversion. If there exists a baby beaver i' where $x_{i'} \leq x_i$ and $y_{i'} \geq y_i$, then $f(i', j) \geq f(i, j)$. Similarly, if there exists a baby beaver j' where $x_j \leq x_{j'}$ and $y_j \geq y_{j'}$, then $f(i, j') \geq f(i, j)$. This means that we only need to check a specific subset of inversions. If a is a list of baby beavers i where no $j \neq i$ satisfies $x_j \leq x_i$ and $y_j \geq y_i$, and b is a list of baby beavers i where no $j \neq i$ satisfies $x_j \geq x_i$ and $y_j \leq y_i$, then we only need to check inversions in

$$\{(u, v) \mid u \in a, v \in b\}.$$

Sort both a and b by x-coordinate.

Claim 2.2. *The function $g_i(j) = f(a_i, b_j)$ is unimodal over all valid j. It increases, reaches a maximum, then decreases.*

Proof. Recall the definition of f: $f(i, j) = \min(|x_i - x_j|, |y_i - y_j|)$. Since both a and b are increasing in both x and y coordinates, for a fixed i, $|x_{a_i} - x_{b_j}|$ increases over valid j as j increases, and $|y_{a_i} - y_{b_j}|$ decreases over valid j as j increases. Thus, $g_i(j)$ forms a V-shape and is unimodal. □

Claim 2.3. *Let $h(i) = \arg\max g_i(j)$, choosing the largest j in case of a tie. $h(i)$ is nondecreasing.*

Proof. From each baby beaver in a, draw a ray in the down-right direction. The maximum separation comes from the baby beaver in b closest to this ray. Therefore, as i increases, the ray moves up-right, so the closest baby beaver from b also moves up-right. This is equivalent to $h(i)$ being nondecreasing. □

Sort the baby beavers by x and use monotonic stacks to construct a and b. Then, use two pointers on a and b to find the maximum separation of an inversion.

Time Complexity: $O(N)$ after sorting.

3 Quantum Beaver

Problem Idea: Brian Xue
Problem Preparation: Brian Xue and Warren Bei
Analysis By: Brian Xue

Let's formalize the problem. For now, assume $n = 1$. A certain plan on day d_i will be the final plan for quantum value z if and only if $d_i \oplus z > d_j \oplus z$ for all $j \neq i$.

Claim 3.1. *Given integers x, y that satisfy $0 \leq x, y < 2^k$ for some k and $x \neq y$, the number of z in the range $[0, 2^k)$ such that $x \oplus z > y \oplus z$ is precisely 2^{k-1}.*

Proof. Consider the binary forms of x and y - add leading zeroes as necessary to make them both k digits. They will share some prefix, have some bit at which they are different, and then have a trailing suffix. Consider the following example for $k = 8$.

$$x = 01001001$$

$$y = 01011011$$

In the above example, the first 3 bits of x and y are the same. They diverge at the 4'th bit, and then have a few trailing bits. The effect of simultaneously XORing x and y is to choose some of the bits to flip. Regardless of which prefix bits we flip, the shared prefix will be the same for $x \oplus z$ and $y \oplus z$. Similarly, it is guaranteed that the 4th bit of $x \oplus z$ will be different from the 4th bit of $y \oplus z$.

Comparing two binary numbers is the same as lexographically comparing strings - looking for the first position at which the strings are different, and then comparing the values at that position. We know that the first position at which the strings are different will be the 4th position. To make $x \oplus z > y \oplus z$, we need the 4th bit of z to be exactly 1. The prefix will be the same, and the bits trailing after the first different bit do not matter.

To generalize, if b bits are shared, then the bit of z corresponding to 2^{k-1-b} must have value not equal to the same bit of x. The values of any of the other bits of z do not matter. Therefore, there are 2^{k-1} possible values of z. □

The above claim should be sufficient to solve the $n = 1$ and $q = 2$ subtask in $O(K)$ per test case. We now try to extend it to $q > 2$.

Claim 3.2. *You are given an integer v and a collection of integers $x = x_1, x_2, \cdots, x_q$ that satisfy $0 \leq v, x_i < 2^k$ for some k. It is also guaranteed v and all members of x are pairwise* *distinct. The number of integer z that satisfy $0 \leq z < 2^k$ and $z \oplus v > z \oplus x_i$ for all i can be computed in the following manner:*

For each $b \in [0, k)$, let y_b represent whether or not there exists an x_i that share a prefix of exactly b bits with z. Let Y be the total sum of all y_b. The number of z satisfying above condition is precisely 2^{k-Y}.

Proof. Consider applying the first claim to each individual element of x. In order for $v \oplus z > x_i \oplus z$, we need a specific bit of z to have value not equal to v. Note that, when multiple values of x_i all share b bits with v, all of the generated restrictions by claim 1 force the same bit of z to have a different value from v.

To combine all of the restrictions together, we need only to check whether each individual bit of z is restricted by some element of x_i. This value is represented by y_b, and the final count of satisfying z is precisely 2^{k-Y} - the number of ways to assign any values to unrestricted bits. $\square$

We can now solve the $n = 1$ subtask in $O(QK)$ per test case by constructing a trie of all d_i from the original problem. We can walk on the trie and enumerate the plans, and using the above claim, compute exactly the number of days for which that plan is the last plan.

Now we have to extend the solution from $n = 1$. We will attempt to use a sweep from tile $i = 1$ to tile $i = n$. We must maintain some data structure that supports the insertion of a plan, the deletion of a plan, and querying the current value of $f(z)$ assuming one tile is acted upon by all plans currently in the data structure. We can accumulate $f(z)$ for all tiles as the final answer.

Consider extending the earlier trie to be dynamic. Let's say we're at the node of the trie that represents range $[x, x + 2^{k'})$. Let our set of plans be all plans in this subtree. At this node, we should store $f(z)$ where we only consider $0 \leq z < 2^{k'}$ and the plans in this subtree. To merge two nodes, there are a few cases. If a node has no children, then the parent node doesn't matter. If a node has a single child

The final complexity of the solution is $O(N + QK)$.

Thank you to Warren Bei for coming up with this cleaner way of maintaining the trie. The original solution required lazy tags and, although asymptotically equal, was much slower.

4 Ditto Piano

Problem Idea: Ray Law
Problem Preparation: Ray Law
Analysis By: Ray Law

Subtask 1: $N \leq 1000$, $M = 0$

Let i and j be two distinct notes. i and j overlap if there is a time when both notes are being played simultaneously. This only occurs if the start time of each note comes before the end time of the other note. In other words, $a_i < b_j$ and $a_j < b_i$. If these notes overlap, they must be played with different fingers. Thus, after sorting the notes by key, we can greedily assign each note the smallest finger available to get the optimal finger assignment.

Let S_i be the set of notes j where $k_j < k_i$ and i and j overlap. Let $\texttt{dp}[i]$ be the smallest finger assignable to note i. Then $\texttt{dp}[i] = \max_{j \in S_i} \texttt{dp}[j] + 1$. The answer is the maximum $\texttt{dp}$ value found. There are $O(N)$ states and $O(N)$ transitions per state, so the total time complexity is $O(N^2)$.

Subtask 2: $N \leq 2 \cdot 10^5$, $M = 0$

This subtask can be solved in two ways: Optimizing the previous DP or by constructing a note dependency DAG.

Solution 1: DP Optimization

The bottleneck in the previous DP is that the number of previous conflicting notes (the size of S_i) grows linearly with i. Therefore, if we checked the time interval the current note was played instead of all the previous notes, our DP could be faster.

Sort the notes by key and compress the time intervals the notes are played on. Let $g_{i,t}$ be the greatest finger assigned to a note among the first i notes that is playing in the time inteval $[t, t+1]$. Now, we can rewrite the transition as:

$$\texttt{dp}[i] = \max_{t=a}^{b-1} g_{i-1,t}$$

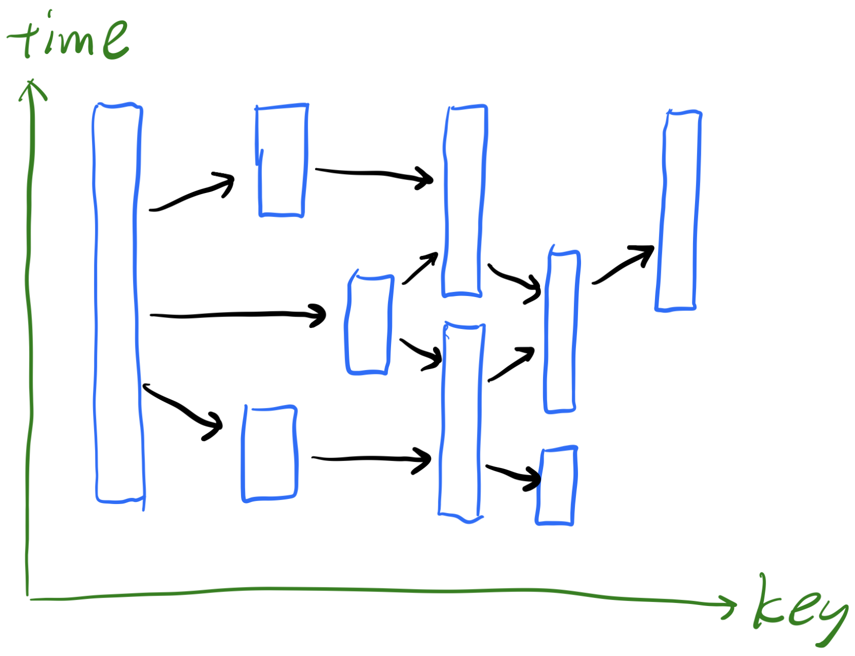
where note i is played from time a to time b.

In order to maintain g and calculate DP transitions as we sweep the notes, we need a data structure that supports range set updates (adding the finger assignment of a new note) and range max query. A lazy segment tree supports both in $O(\log N)$ time. This reduces the transitions to $O(\log N)$ per state, reducing the total time complexity to $O(N \log N)$.

Solution 2: Note Dependency DAG

Recall that notes i and j with $k_i < k_j$ overlap iff $a_i < b_j$ and $a_j < b_i$. Consider a note dependency DAG on the N notes, where an edge $i \to j$ is drawn for every pair of overlapping notes. The answer would be the longest path in the DAG, which can be computed with DP. However, there are $O(N^2)$ edges in the DAG because there are $O(N^2)$ pairs of overlapping notes. Constructing the DAG like this will not work.

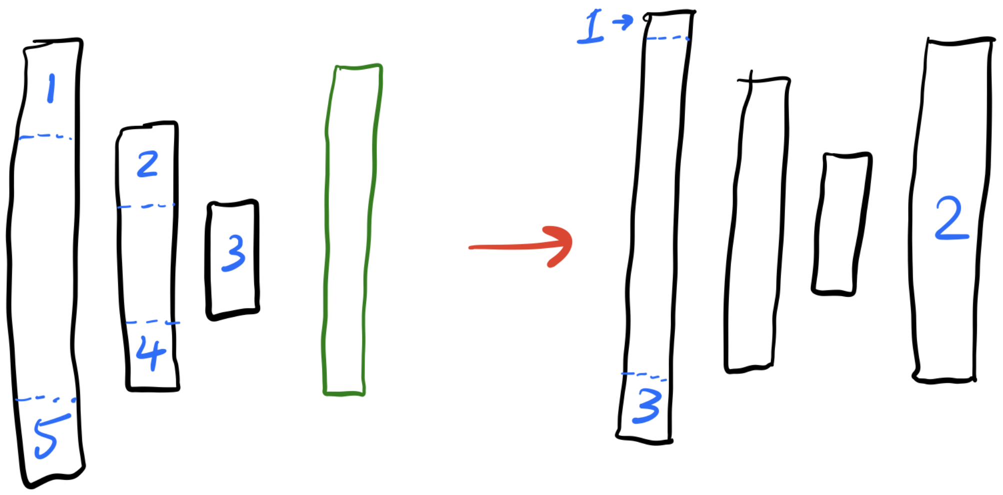
If the DAG contains three edges $i \to j$, $j \to k$, and $i \to k$, deleting the last edge will not change the longest path in the DAG. This means that we do not need to connect each note to every other note that overlaps with it. Only the notes that directly precede the current note need to be checked for overlap. An illustration of this reduced DAG is shown below:



The DAG now has $O(N)$ edges because it is planar.

To construct the DAG, plot the notes on the Cartesian plane, with the x axis being the key and the y axis being the time. Imagine you are at note i and you look left. You will see a bunch of segments, where each segment is a continuous section of a note that is not "covered" by a note with a later key. Maintain a set of segments as you sweep the notes. Each time you process a new notes, search the segment set to find the directly overlapping notes that come before it. Then, when you add the new note to the set of segments, some segments will be pushed off the set, and at most two at the borders will be cut into smaller segments.

An illustration is below. Before the green note, there are five segments, numbered from 1 to 5. When processing the green note, we search the set of segments and find that segments 1, 2, 3, and 4 overlap with it. We add an edge from each of the corresponding notes to the green note, then push the green note onto the set of segments. Segments 2, 3, and 4 get pushed off the set, and segment 1 gets cut.

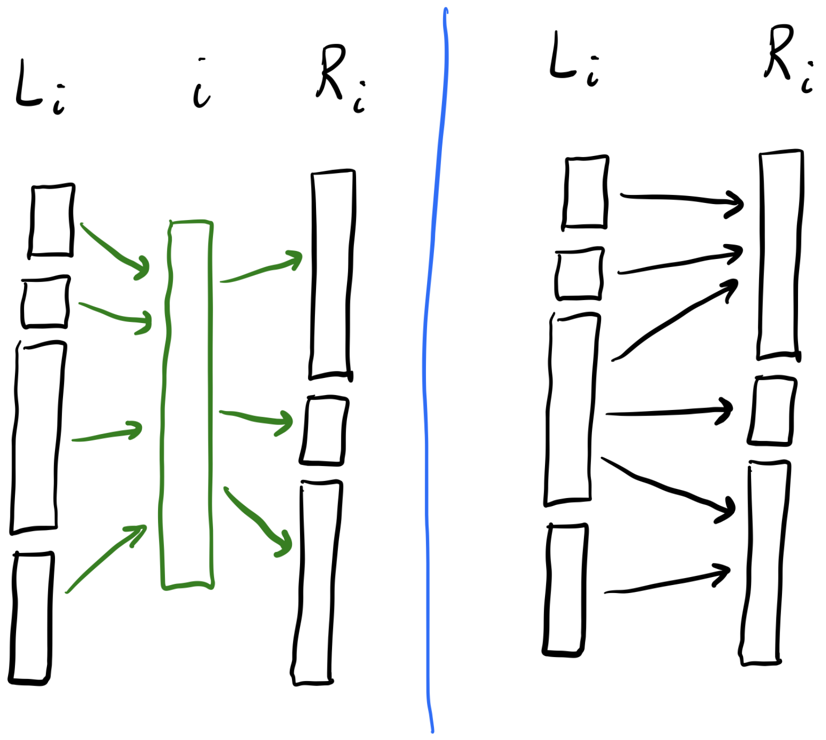


Constructing the DAG using the set of segments takes $O(N \log N)$ time, and finding the longest path takes $O(N)$ time. The total time complexity is $O(N \log N)$.

Full Solution: $N \leq 2 \cdot 10^5$, $M = 0$ **or** 1

Starting from the DAG from above, modify it so that after any note is deleted, all chains of overlapping notes are still captured. This can be done by adding a few more edges to the DAG.

Let L_i and R_i be the in-neighbors and out-neighbors of note i, respectively. If note i is deleted, only chains of notes that pass through note i will be affected. Each of these chains will pass through a note in L_i and a note in R_i, so it suffices to add overlapping edges between the two sets. Since no two notes in L_i overlap (and the same for R_i), we can add all overlapping edges using two pointers. For note i there are $O(|L_i| + |R_i|)$ additional edges, and summing over all i, the total number of additional edges is $O(N)$.



Now, we reduced the problem to finding the minimum possible longest path in a DAG if at most one node can be deleted. Let T be a topological order of the nodes (in our case, sorting notes by key suffices). For each node, use DP to calculate the longest path ending at that node, `ldist`, and the longest path starting at that node, `rdist`. The longest path without any deletions is $\max_i$ `ldist[i]`.

Now, we test each note deletion. For each node i, let J_i be the set of edges from a node u earlier in the topological sort ($T(u) < T(i)$) to a node v later in the topological sort ($T(i) < T(v)$). Let $V_i = \{\texttt{ldist[u] + rdist[v] + 1} : (u \to v) \in J_i\}$ be a multiset of longest path lengths through every edge in J_i. Maintain J_i and V_i as you iterate through the nodes in topological order. The upper bound on the answer if node i is deleted is $\max(\max \texttt{ldist}[j < i], \max \texttt{rdist}[j > i], \max V_i)$.

If you use a priority queue for J_i and a multiset for V_i, all node deletions can be checked in $O(N \log N)$ time. The total time complexity is $O(N \log N)$.

5 Busy Beaver's Water Network

Problem Idea: Cheng Jiang
Problem Preparation: Yifan Kang and Cheng Jiang

Analysis By: Cheng Jiang

Claim 5.1. *For a graph G that satisfies the condition, there exists a unique node u that has degree at least K in all spanning trees of G.*

Proof. Suppose that there are two possible nodes u, u'. Consider one spanning tree T_1 with $\deg(u) \geq K$ and one spanning tree T_2 with $\deg(u') \geq K$. Notice that for any two trees T_1, T_2, there exists a sequence of trees $T_1, T_{01}, T_{02}, \cdots, T_{0K}, T_2$ such that adjacent trees are only off by one edge. One such construction is simply seeking for an edge in T_2 that is not in T_1, adding it to T_1, and removing an edge in the cycle that is not in T_2.

Now, we find the first tree in these trees with $\deg(u) < K$. Such tree exist since in the end we have $\deg(u) \leq 2N - 2 - K - (N - 2) < K$. We must have $\deg(u) = K - 1$ since we are only off by one edge each time. Notice that any other node can have degree at most $2N - 2 - (K - 1) - (N - 2) < K$, contradiction. □

5.1 Solution

Let that node in the claim be u. We have that the minimum possible degree of u in a spanning tree is the number of connected components in the graph after deleting u. So it suffices to compute the number of connected graphs such that it leaves $\geq K$ connected components after deleting u. Clearly we can compute by a $O(N^2)$ DP (or $O(N \log N)$ using generating functions), the number of connected graphs with i ($1 \leq i \leq N$) nodes. We can multiply it by $2^i - 1$ to get the number of ways to connect it to the pivot node. Let this answer be f_i.

After having these computed, we can calculate the answer using an $O(N^3)$ DP by putting one connected component at one time, or by writing out the EGF of the sequence

$$F = \sum_{i=1}^{N} \frac{f_i}{i!}$$

and noting that the answer can be easily obtained by

$$[x^{N-1}] \sum_{i \geq K} \frac{F^i}{i!}$$

using polynomial composition ($O(N \log^2 N)$) or any other methods (Newton's Method, Lagrange's Inversion) in $O(N^2)$.