International Informatics Olympiad in Teams
International Round
Editorial

The scientific committee

Tuesday, 26th May, 2026

Problem Copou Grand Prix

Proposed by: Alexandru Gheoghieș

First, let us analyze a simplified version of the problem where we can assign only two distinct point values (e.g., W and 0) to the available rankings. In this scenario, the optimal strategy is to find a prefix i that maximizes the score difference between Alice and Bob. We define this difference for a given prefix i as:

$$\sum_{A_j \leq i} 1 - \sum_{B_j \leq i} 1$$

Let S_{max} denote this maximum prefix difference. It can be shown that the maximum achievable score difference is $W \cdot S_{max}$. This result is obtained by assigning the maximum weight W to the highest-ranking positions (up to the prefix that yields S_{max}) and 0 to all remaining positions.

The key takeaway is a greedy principle: to maximize the score difference, the highest-ranking positions must receive the maximum possible points, while the lowest-ranking ones must receive the minimum. We can generalize this approach to solve the full problem.

For each position i, we compute two values:

- $PrefixScore_i$: The maximum possible score obtained from the prefix up to i.
- $SuffixScore_i$: The minimum possible score obtained from the suffix starting after i.

The final answer is then the maximum sum of these two values across all valid split points:

$$\max_{1 \leq i < m} (PrefixScore_i + SuffixScore_{i+1})$$

To efficiently manage the large range of possible rankings, we observe that the only relevant positions are those where either Alice or Bob finished at least once. By storing these relevant positions in an ordered map structure, we can compute the $PrefixScore$ array by iterating through the map in increasing order, and the $SuffixScore$ array by iterating in decreasing order.

By processing these prefix and suffix arrays efficiently, the overall time complexity of the solution becomes $O(N \log M)$, where N is the total number of recorded finishes and M is the number of distinct relevant positions.

Problem Drawing

Proposed by: Alin-Gabriel Răileanu

For the following discussion, let $W_{total} = \sum_{e \in E} w(e)$ be the sum of all edge weights in the input tree $T = (V, E)$.

1. Reducing the problem to edge-disjoint paths

We will prove the following fundamental theorem, which forms the basis of all solutions:

> **Reduction Theorem.** The minimum cost of a drawing using at most K chalk lifts is exactly:
>
> $$2 \cdot W_{total} - \max\{w(P_1) + w(P_2) + \cdots + w(P_{K+1})\}$$
>
> where the maximum is taken over all possible choices of at most $K + 1$ simple edge-disjoint paths in the tree T.

Step 1: Euler condition

A drawing with at most K chalk lifts consists of at most $K + 1$ continuous walks. For every edge $e \in E$, let $t(e) \geq 1$ denote the total number of times edge e is traversed by these walks.

Construct the connected multigraph $M = (V, E_M)$ where every edge $e \in E$ appears exactly $t(e)$ times.

By a generalization of Euler's theorem, a connected multigraph can be decomposed into at most P open walks if and only if it has at most $2P$ vertices of odd degree.

Therefore, we obtain the first structural constraint: **In any valid drawing, the multigraph M has at most $2(K + 1)$ odd-degree vertices.**

Step 2: In an optimal drawing, $t(e) \in \{1, 2\}$

Assume by contradiction that there exists an edge $e' = (u, v)$ such that $t(e') \geq 3$ in an optimal drawing.

If we remove two copies of e' from M, the edge remains covered at least once, so connectivity is preserved. This operation decreases the degrees of both u and v by 2.

Hence, the operation **does not change the parity** of any vertex in M, meaning the new multigraph still satisfies the condition of having at most $2(K + 1)$ odd vertices.

However, the total cost strictly decreases, since we save $2 \cdot w(e')$.

This contradicts optimality.

Therefore, in every optimal drawing, $t(e) \in \{1, 2\}$ for every edge $e \in E$.

Step 3: The parity invariant

Define the set of "saved" edges (traversed exactly once) as $S = \{e \in E \mid t(e) = 1\}$.

All remaining edges are traversed exactly twice.

The total cost becomes

$$C = \sum_{e \in S} w(e) + 2 \sum_{e \notin S} w(e) = 2 \cdot W_{total} - \sum_{e \in S} w(e).$$

Thus, minimizing the drawing cost is exactly equivalent to **maximizing the total weight of** S.

Now consider the parity of a vertex v in the multigraph M:

$$\deg_M(v) = \deg_S(v) + 2 \cdot \big|\{e \notin S \mid v \in e\}\big|.$$

Therefore,

$$\deg_M(v) \equiv \deg_S(v) \pmod 2.$$

Invariant: The parity of every vertex in M is identical to its parity in the subgraph (V, S).

Replacing this into the constraint from Step 1, we deduce that the subgraph formed by the saved edges (V, S) can have at most $2(K + 1)$ odd-degree vertices.

Step 4: Decomposition of bounded forests

Since (V, S) is a subgraph of a tree, it contains no cycles, hence it is a forest.

A classical property of forests states:

> Any forest with at most $2P$ odd-degree vertices can be decomposed into at most P edge-disjoint simple paths.

Indeed, every connected component of the forest is a tree, and a tree with exactly $2t$ odd-degree vertices can be decomposed into exactly t edge-disjoint paths. Summing over all connected components yields the result.

Since (V, S) has at most $2(K + 1)$ odd vertices, it follows that the saved edges can always be represented as the union of at most $K + 1$ edge-disjoint simple paths.

Thus, maximizing the total saved weight is exactly equivalent to choosing at most $K + 1$ edge-disjoint paths of maximum total weight.

Step 5: Sufficiency (reverse construction)

Conversely, suppose we choose $m \leq K + 1$ edge-disjoint paths.

Their union forms a subgraph S whose only odd-degree vertices are the path endpoints. Therefore, S has at most $2m \leq 2(K + 1)$ odd-degree vertices.

Now construct the multigraph M by setting $t(e) = 1$ for $e \in S$ and $t(e) = 2$ otherwise.

Then M has at most $2(K+1)$ odd vertices. By the generalized Euler theorem, M can be drawn using at most $K + 1$ walks, i.e. at most K chalk lifts.

The reduction is complete.

2. Computing the reduction in $\mathcal{O}(N \cdot K)$

Having reduced the problem to finding at most $K + 1$ edge-disjoint paths with maximum total weight, we can solve the medium subtasks using two classical approaches.

Method 2.1: Knapsack DP on tree

Choose an arbitrary root.

Define $dp[u][p][j]$ as the maximum total saved weight inside the subtree of u, using j fully closed paths.

The state $p \in \{0, 1\}$ indicates whether there exists an open path starting from u that may later be extended toward the parent of u.

Transitions are performed by merging the current subtree of u with a new child c, deciding whether edge (u, c) is saved or doubled.

If both states contain open paths ($p = 1$), they may be merged through edge (u, c) into a closed path.

This DP can be computed in $\mathcal{O}(N \cdot K)$ using the technique presented here: Part 7.

Method 2.2: Greedy algorithm

An alternative approach is to iteratively select the best path.

At every step $1 \dots K + 1$:

- Find the maximum weighted diameter in the residual graph.
- Add its weight to the answer.
- Multiply by -1 the weights of all edges belonging to this diameter.

This approach runs in $\mathcal{O}(K \cdot N)$ and can be justified using Max-Flow arguments.

From this point there are two possible continuations: optimizing the DP solution or optimizing the greedy solution using advanced data structures. Below we continue with the first approach.

3. The $\mathcal{O}(N \log W)$ optimization using Alien's Trick

For $N, K \leq 200\,000$, the $\mathcal{O}(N \cdot K)$ approach becomes too slow.

Proof of concavity

Let $f(x)$ denote the maximum total saved weight obtainable using exactly x paths.

From the nature of the greedy process discussed above, at each step we choose the edges with the largest remaining positive contribution in the residual graph. Since penalties (negative edges) accumulate over time, the marginal gain decreases at every step.

Formally,

$$f(x) - f(x-1) \geq f(x+1) - f(x),$$

which means that the function $f(x)$ is **concave**.

Applying Lagrangian Relaxation (Alien's Trick)

Due to concavity, we may eliminate the restriction of using exactly $K + 1$ paths by instead introducing a **penalty** $\lambda \geq 0$ for every path used.

Define the relaxed problem:

$$h(\lambda) = \max_{x \geq 0} \big(f(x) - \lambda \cdot x\big).$$

For a fixed λ, we can compute $h(\lambda)$ in a single linear DFS traversal in $\mathcal{O}(N)$.

The new DP no longer requires the state j (number of paths). It only stores:

$$dp[u][p] = (\text{penalized\_cost}, \text{number\_of\_paths}).$$

Whenever we merge two endpoints into a completed path or start a new path, we subtract λ from the profit and increment the path counter.

To recover exactly $f(K+1)$, we binary search the optimal penalty value $\lambda' \in [0, W_{total}]$.

- If the DP chooses more than $K+1$ paths, then λ is too small (increase it).
- If the DP chooses fewer than $K+1$ paths, then λ is too large (decrease it).

Once we find a value λ' such that one optimal configuration uses $c' \geq K+1$ paths, we recover the original value by undoing the penalty:

$$f(K+1) = h(\lambda') + \lambda' \cdot (K+1).$$

Final complexity:

The binary search requires $\mathcal{O}(\log W_{total})$ iterations.

Each iteration performs a linear DP in $\mathcal{O}(N)$ time and memory.

Thus, the total complexity becomes $\mathcal{O}(N \log W_{total})$, which comfortably fits within the constraints.

Subtask 1

Examples.

Subtask 2 and 3

Compute the diameter (or weighted diameter) using standard linear tree algorithms.

The final answer is:

$$2 \cdot W_{total} - \text{diameter}(T).$$

Subtask 4 (Every node has degree at most 2)

The tree is simply a path.

Since at least one continuous walk is always allowed, we may save the entire path using a single traversal.

Therefore, all $N-1$ edges are traversed exactly once, and the final drawing cost becomes exactly W_{total}.

Complexity: $\mathcal{O}(N)$.

Subtask 5 (All $N-1$ edges are incident to node 1)

The tree is a star.

Any simple path in this configuration uses either:

- one edge (starting from the center and ending in a leaf), or
- two edges (connecting two leaves through the center).

With at most $K+1$ edge-disjoint paths, we may therefore select at most $2(K+1)$ edges.

The solution consists of sorting the edges in decreasing order of weight and summing the first $\min(N-1, 2K+2)$ weights.

The final cost is obtained by subtracting this value from $2 \cdot W_{total}$.

Complexity: $\mathcal{O}(N \log N)$.

Subtask 6 ($N \leq 200$)

This subtask targets the dynamic programming approach described above, implemented naively in $O(N^2 \cdot K)$.

Subtask 7 ($N \cdot K \leq 10^7$)

This subtask targets the optimized $O(N \cdot K)$ dynamic programming solution.

Subtask 8 (Every simple path has at most 100 edges)

To optimize the greedy solution, one may use a difficult combination of HLD + Treaps / Segment Trees.

This subtask helps because most edge cases in that optimization become easier to handle.

Subtask 9 ($N \leq 7 \cdot 10^4$)

For this subtask, the full data structure solution from Subtask 8 can be implemented in $O((N + K) \log^2 N)$.

Subtask 10 (No additional constraints, $N, K \leq 200\,000$)

This subtask requires the full Alien's Trick solution in $O(N \log W_{total})$, or an optimized version of the data structure solution.

Thinking problem

Using the hints from this editorial, find a data-structure optimization for the greedy algorithm.

More precisely, solve as efficiently as possible the following problem:

Given a tree, process Q operations of two types:

- Update $u - v$: negate the weight of every edge on the path between these two nodes.
- Query: find the weighted diameter of the tree.

Problem Prahova Valley

Proposed by: Andrei-Cristian Ivan

First, let's analyze the algorithm. If we take a closer look, we can see that each position R_i represents the subarray $[i, R_i)$, where every element is divisible by V_i. Since a valid solution is guaranteed, we can also deduce that if we take two different subarrays $[i, R_i)$ and $[j, R_j)$, either one is completely contained within the other, or they are entirely disjoint. Therefore, there are no partial overlaps.

How do we construct a lexicographically minimal array V? We will iterate from 1 to N through the sequence and maintain a stack of R values. Before processing the i-th element, we must pop elements from the stack until $R_{\text{stack top}} > i$ to keep only the active *restrictions*. We have to analyze the following cases:

- $R_i = N + 1$: In this case, $V_i = 1$ because it must divide every element in $V_{i+1}, \ldots, V_N$, and 1 ensures minimality.
- $R_i = R_{\text{stack top}}$: In this case, $V_i = V_{\text{stack top}}$.
- $R_i < R_{\text{stack top}}$: In this case, it is clear that assigning $V_{\text{stack top}}$ would violate the bounds of this restriction, so we must multiply the value by another integer.

For the subtask where $R_i = i + 1$, we can see that a valid solution is to alternate between 2 and 3. We will apply a similar approach to the general case. For an interval $[i, R_i)$, we will use the same construction as the intervals containing it, and we will alternate between using 2 and 3 for intervals that are on the same level of inclusion.

Problem Wrong Modulo

Proposed by: Alexandru Gheoghieș

Let's analyze the equation given in the task.

The modulo operation can be formally defined using the floor function ($\lfloor \cdot \rfloor$):

$$A \pmod{B} = A - B \cdot \left\lfloor \frac{A}{B} \right\rfloor$$

If we apply this definition to our expression (where $A = a \cdot x$ and $B = a \cdot y$), we have:

$$(a \cdot x) \pmod{a \cdot y} = (a \cdot x) - (a \cdot y) \cdot \left\lfloor \frac{a \cdot x}{a \cdot y} \right\rfloor$$

Since $a \neq 0$, the fraction inside the floor function simplifies ($\frac{a \cdot x}{a \cdot y} = \frac{x}{y}$), and the equation becomes:

$$(a \cdot x) \pmod{a \cdot y} = (a \cdot x) - (a \cdot y) \cdot \left\lfloor \frac{x}{y} \right\rfloor$$

Now we factor out a:

$$(a \cdot x) \pmod{a \cdot y} = a \cdot \left(x - y \cdot \left\lfloor \frac{x}{y} \right\rfloor \right)$$

The expression in the parentheses is exactly the definition for $x \pmod{y}$. Therefore:

$$(a \cdot x) \pmod{a \cdot y} = a \cdot (x \pmod{y})$$

Substituting this back into the initial equation, the condition becomes:

$$a \cdot (x \pmod{y}) = x \pmod{y}$$

Let $r = x \pmod{y}$. The relation becomes:

$$a \cdot r = r \implies r(a - 1) = 0$$

Case $a = 1$ If $a = 1$, the equation becomes $r \cdot 0 = 0$, which holds true for any remainder r. Therefore, any pair (x, y) with $1 \leq x, y \leq N$ satisfies the condition. The answer for this case is simply N^2.

Case $a > 1$ If $a > 1$, then $a - 1 > 0$, meaning the equation $r(a-1) = 0$ is satisfied if and only if $r = 0$. This means $x \pmod{y} = 0$, so x must be a multiple of y.

For a fixed y, the number of multiples $x \leq N$ is $\lfloor \frac{N}{y} \rfloor$. The total number of valid pairs is the sum over all possible values of y:

$$\sum_{y=1}^{N} \left\lfloor \frac{N}{y} \right\rfloor$$

Optimal Solution: $O(\sqrt{N})$ For Subtask 2, a naive $O(N)$ computation for each test case would be sufficient. However, for Subtask 5, we have $N \leq 10^9$ and $T \leq 1000$. An $O(N)$ approach would require up to 10^{12} operations, exceeding the time limit.

To optimize this, we analyze the behavior of the function $f(y) = \lfloor \frac{N}{y} \rfloor$. Notice that as y increases, the value of $f(y)$ decreases, but it stays constant for long intervals. For example, for all $y \in (\lfloor \frac{N}{2} \rfloor, N]$, the value of $\lfloor \frac{N}{y} \rfloor$ is exactly 1.

In fact, the expression $\lfloor \frac{N}{y} \rfloor$ takes at most $2\sqrt{N}$ distinct values:

- For $y \leq \sqrt{N}$, y can take at most $\sqrt{N}$ integer values, yielding at most $\sqrt{N}$ distinct results.
- For $y > \sqrt{N}$, the result satisfies $\lfloor \frac{N}{y} \rfloor \leq \lfloor \frac{N}{\sqrt{N}} \rfloor = \sqrt{N}$. Thus, there are at most $\sqrt{N}$ possible distinct results for the fraction.

Since there are very few distinct values, we can group the terms in our sum by these identical values. We iterate through the sequence starting with $L = 1$. Let $V = \lfloor \frac{N}{L} \rfloor$. We want to find the largest index $R \geq L$ such that $\lfloor \frac{N}{R} \rfloor = V$.

From the inequality $\lfloor \frac{N}{R} \rfloor = V$, we deduce that $\frac{N}{R} \geq V$, which implies $R \leq \lfloor \frac{N}{V} \rfloor$. Therefore, the upper bound of the current block that yields the same quotient V is exactly:

$$R = \left\lfloor \frac{N}{V} \right\rfloor = \left\lfloor \frac{N}{\lfloor \frac{N}{L} \rfloor} \right\rfloor$$

For all y in the interval $[L, R]$, the value of $\lfloor \frac{N}{y} \rfloor$ is exactly V. The contribution of this entire interval to our total sum is simply the number of elements in the interval multiplied by V:

$$(R - L + 1) \times V$$

After adding this contribution to our total sum, we can safely skip the entire interval and set the start of our next block to $L = R + 1$. We repeat this process, jumping from block to block, until L exceeds N.

Final Complexity Because the algorithm processes the range in blocks and evaluates exactly one block for each distinct value of $\lfloor \frac{N}{y} \rfloor$, it will run at most $2\sqrt{N}$ times per test case. The time complexity per testcase drops from $O(N)$ to $O(\sqrt{N})$. For T test cases, the overall time complexity is $O(T\sqrt{N})$, which roughly requires $1000 \times 2\sqrt{10^9} \approx 6.3 \times 10^7$ operations, comfortably executing within standard time limits. The space complexity is $O(1)$.

Problem From Bucharest 2 Piatra Neamț

Proposed by: Haba Matei Ionuț and Răzvan Alexandru Rotaru

First, let's observe what the task asks for. The problem requires us to find the number of pairs (u, v) such that there is a directed path from u to v and a directed path from v to u. By definition, two vertices satisfy this condition if and only if they belong to the same *Strongly Connected Component* (SCC).

If an SCC contains exactly K original cities, it will contribute K^2 valid pairs to our total answer. The total answer is simply the sum of K^2 over all SCCs in the final network.

Subtask 1: Exemple

Subtask 2: $N, Q \leq 500$ For these subtasks, we can naively simulate each construction phase by individually iterating through the elements in the interval $[l, r]$ and adding the corresponding edges to an adjacency list. Because N and Q are very small, the maximum number of added edges is $O(Q \cdot N)$. After constructing the graph, we can find the SCCs using standard algorithms like Tarjan's or Kosaraju's, or even by running a BFS/DFS from every single node. The overall time complexity would be $O(N(N + M) + Q \cdot N)$, which comfortably passes the time limit.

Subtask 3: $Q = 0$ In this subtask, there are no construction phases. The graph is static and identical to the original road network. We can directly run an SCC algorithm (Tarjan or Kosaraju) on the initial N vertices and M edges. The time complexity is $O(N + M)$.

Subtask 4: All construction phases are of type = 3 An operation of type 3 effectively creates a fully connected clique among the cities in the interval $[l, r]$. Because all cities in this interval can now mutually reach each other, they are guaranteed to belong to the exact same SCC. Therefore, we can process these queries offline using a Disjoint Set Union (DSU) data structure, merging adjacent elements in $[l, r]$ to compress them into "super-cities". After compressing the intervals, we can run Tarjan's algorithm on the reduced graph.

Subtask 5: $r - l \leq 40$ **for types 1 and 2** Here, the intervals for type 1 and 2 construction phases are very small (at most 40 cities). We can still afford to naively add edges one by one for these operations because each query will add at most 40 edges. The total number of edges added by queries will not exceed $40 \cdot Q$. Combined with the DSU compression logic for type 3 queries (if any), the total number of edges in the graph remains linear with respect to the constraints, avoiding the need for heavy data structures.

Subtask 6: Full Solution (Segment Tree for Graphs) First of all we begin by solving the third type queries in a similar manner to the fourth subtask so we are left only with updates of type 2 and 3s For the final subtask, adding $O(N)$ edges per operation would yield up to $O(Q \cdot N)$ edges, requiring too much memory and time. We need a mechanism to add edges to/from entire intervals in logarithmic time. We can achieve this using a well-known technique involving two segment trees built over the N cities:

- **In-Tree**: A segment tree where edges are directed from parents to children. The leaves correspond to the actual N cities. This tree efficiently distributes incoming paths to intervals.
- **Out-Tree**: A segment tree where edges are directed from children to parents. The leaves are the same N cities. This tree efficiently collects outgoing paths from intervals.

Using this structure, we process the construction phases dynamically:

- **Type 1 (**$1\ x\ l\ r$**)**: We add a directed edge from the leaf x to the $O(\log N)$ canonical nodes in the **In-Tree** that perfectly cover the interval $[l, r]$. Because In-Tree edges flow downwards, x can reach all leaves in $[l, r]$.
- **Type 2 (**$2\ x\ l\ r$**)**: We add a directed edge from the $O(\log N)$ canonical nodes in the **Out-Tree** covering $[l, r]$ to the leaf x. Because Out-Tree edges flow upwards from the leaves, all cities in $[l, r]$ can reach x.
- **Type 3 (**$3\ l\ r$**)**: We model this by creating a single unique **dummy node** for each such operation. We add edges from the **Out-Tree** nodes of $[l, r]$ to the dummy node, and from the dummy node to the **In-Tree** nodes of $[l, r]$. This allows any city in $[l, r]$ to reach the dummy node and then travel to any other city in $[l, r]$. Because the dummy node is isolated otherwise, no false paths are created outside the interval.

Final Complexity After modeling all updates through the In-Tree and Out-Tree, we simply run an SCC algorithm (like Tarjan's or Kosaraju's algorithm) on the generated graph. The number of vertices in our extended graph is bounded by $O(N + Q)$, and the number of edges is bounded by $O(M + N + Q \log N)$. The SCC algorithm takes $O(V + E)$ time, yielding an overall time complexity of $O(M + N + Q \log N)$.

Butea

Proposed by: Tiberiu-Stefan Cozma

Given k **black nodes** $(S_1, S_2, \ldots, S_k)$, our goal is to choose exactly k nodes $(T_1, T_2, \ldots, T_k)$ to minimize the total sum of distances between all pairs of S and T.

This is mathematically equivalent to choosing the k nodes with the smallest individual cost, where the cost of a node u is the sum of its distances to all black nodes:

$$C(u) = \sum_{i=1}^{k} \text{dist}(u, S_i)$$

Calculating $C(u)$ for every single node in the tree is too slow. To build an efficient algorithm, we can rely on two key structural properties of the tree.

Lemma 1: The optimal nodes form a connected component

,

Proof: Let r be the node with the absolute minimum cost. If we root the tree at r and move to any child v, the cost change depends entirely on how many black nodes are inside v's subtree.

Unless v's subtree contains more than half of all the black nodes, moving to v will strictly increase the cost. Because the number of black nodes in a subtree only decreases as you move further away from the root, once the cost starts increasing, it can never decrease again. Therefore, any set of optimal (lowest-cost) nodes must be connected to r.

Lemma 2: Every optimal node lies on the virtual tree induced by the black nodes

Proof

- **Outside the virtual tree:** If a node u is outside the virtual tree, its subtree contains zero black nodes. Moving from u toward the virtual tree strictly decreases the distance to *all* black nodes simultaneously, lowering the cost. Thus, u cannot be optimal.
- **On a compressed edge:** If a node lies on a long, unbranching path between two virtual tree nodes, it is always optimal to slide toward whichever endpoint has the "heavier" weight of black nodes behind it.

The Solution

Based on these two lemmas, the strategy is intuitive: find the absolute best starting node, and systematically expand outward.

To do this efficiently, we use a **Centroid Decomposition** data structure, which allows us to insert/remove black nodes and query $C(u)$ in $O(\log n)$ or $O(\log^2 n)$ time.

Algorithm Steps per Query:

1. **Initialize:** Load all k black nodes into the centroid structure.
2. **Find the Root:** Enumerate the nodes in the *virtual tree* of the black nodes to find the one with the absolute minimum cost.
3. **Expand:** Starting from this minimum-cost node, perform a BFS-like expansion. Use a priority queue to always visit the neighboring node with the lowest $C(u)$.
4. **Finish:** Stop the search once you have collected exactly k nodes.
5. **Cleanup:** Clear the centroid structure by removing the black nodes.

Because Lemmas 1 and 2 guarantee we only explore the strictly relevant region of the tree, and the original tree is binary (meaning node degrees are small), the total time complexity per query safely reduces to: $O\left(k \log^2 n\right)$

Little Square

Proposed by: Andrei Lețu

Subtask 1: $P = 0$

We have no restriction on where to place the pieces. Each piece can be rotated in one of four orientations (apart from the small 1×1 piece, which is the same any way you rotate it). We have $N - 1$ pieces that can be arranged in 4 orientations each, so the answer is 4^{N-1}.

Subtask 2: $N \leq 8$

We can generate all boards through backtracking and check which ones satisfy the restrictions.

Subtask 3: one small piece at the top

Since the small piece is fixed at the top, $Y - 1$ of the other $N - 1$ small pieces must be placed below and to the left of the small piece, and the rest must be placed below and to the right. So the answer is C($N - 1$, $Y - 1$), where C(n, k) is n choose k.

Subtask 4: one small piece somewhere on the board

Similar to the previous subtask, the small piece si fixed. Each time we placed one of the other pieces it will either cover the small piece above or below, and either to the left or the right, independently. Among the $N-1$ other pieces, we must choose $X-1$ that cover the small piece above (the rest will cover it below), and $Y-1$ to cover it to the left (the rest will cover it to the right). The choice for each piece is made independently. The answer is
$\mathrm{C}(N-1,\, Y-1) \cdot \mathrm{C}(N-1,\, X-1)$.

Subtask 5: one piece on the board

If the piece has side length of 1, this problem is reduced to the previous subtask. Otherwise we have to consider each of the 4 orientations and check that the piece remains inside the board. For each orientation we can compute the numbers of ways to place the bigger pieces around it similar to subtask 4 and the number of ways to place the smaller pieces inside it similar to subtask 1, and multiply the results.

Subtask 6: full

We can sort the pieces by side length. The smallest piece could have up to 4 orientations, the rest will have their opening pointed toward the smallest fixed piece. We can start with the biggest fixed piece, compute the number of ways to arrange the pieces larger than it, and then multiply that with the arrangements for the smaller pieces. To get the number for the smaller pieces we can consider the square bounded by the current piece and recursively solve the problem for the smaller bounded region, considering only the remaining pieces.

Pear Trees

Proposed by: Andrei Lețu

Since we need to compute the answer modulo an odd number, this might affect the parity during our operations. To avoid this we either need to keep track of what parity each number has, or we can compute everything modulo $2 \cdot mod$ to avoid messing up our parities.

Subtask 1: $N, Q \leq 1000$

We can solve this in $\mathrm{O}(N \cdot Q)$ by looping through the whole interval and computing the result from left to right.

Subtask 2: $P_i \leq 1$

All trees have either 1 or 0 flowers. Let's consider 3 cases:

1. Both trees have 0 flowers, nothing changes after pollination.
2. Both trees have 1 flower, nothing changes after pollination.
3. One tree has 0 flowers and one tree has 1 flower, then after pollination both trees will have 1 flower each.

We can notice that the number of flowers in a tree will never go above 1. If there are no trees that have flowers in the interval then the result will be 0, otherwise, the tree that has a flower will propagate its flower all the way to the right. We can precompute for each position where the closest flower is in the left direction, and answer the queries by checking whether the closest flower is in the interval. Time complexity is $O(N + Q)$.

Subtask 3: $N \leq 1000$

We can precompute the result for all possible intervals by fixintg the left endpoint and looping through the right endpoints. Time complexity is $O(N^2)$.

Subtask 4: the number of flowers in each tree has the same parity

Since the multiplicative branch of our pollination operation preserves parity, all queries are reduced to computing the product on an interval. This can be done by using a prefix product array. Time complexity is $O(N + Q)$.

Subtask 5: full

Consider a fixed interval $[L, R]$, with the bees currently being in position $L-1$, and let x be the number of flowers in tree $L-1$ currently. Since we are only doing multiplications and additions, the number of flowers in tree R will be of the form $a \cdot x + b$, where a and b are constant coefficients that depend on the parity of x. We can keep this information in a data structure, for example the nodes of a segment tree. Time complexity is $O(N + Q \cdot \ln N)$.

Buses from Bragadiru

Proposed by: Andrei Chertes

Each stop i carries two values, $depart_i$ and $arrive_i$, and the cost of hopping directly from i to j is $C(i, j) = depart_i \cdot arrive_j$. A trip of length k is a walk

$$u_{i_1} \to u_{i_2} \to \cdots \to u_{i_{k+1}}$$

through $k+1$ stops, where repetitions are allowed, and its cost is the product of its k edge costs. A query `3 L R k` asks for the sum, over *all* trips of length k whose stops lie entirely in $[L, R]$, of the trip cost, taken modulo $10^9 + 7$. The two update types add a value on a whole range of *depart* or of *arrive*.

Throughout, fix a query interval $[L, R]$ and let $m = R - L + 1$. Restricting to the stops of $[L, R]$, let C be the $m \times m$ matrix with $C[i][j] = depart_i \cdot arrive_j$.

Subtask 1 ($N, Q \leq 1000$, $k \leq 20$): naive dynamic programming.

Let $f_t[v]$ be the sum of costs of all trips of length t ending at stop v (staying inside $[L, R]$). A trip of length 0 is a single stop of cost 1, so $f_0[v] = 1$ for every v, and

$$f_t[v] = \sum_{u \in [L,R]} f_{t-1}[u] \cdot C(u, v).$$

The answer to the query is $\sum_v f_k[v]$. Each transition is a matrix–vector product costing $O(m^2)$, so one query costs $O(m^2 k)$. This is fast enough only when both N and k are small.

Subtask 2 ($N, Q \le 100$, large k): it is a matrix power.

The recurrence above is exactly the multiplication of the vector f_{t-1} by the matrix C. Hence $f_k = C^k f_0$, and since f_0 is the all–ones vector and we sum all entries of f_k, the answer equals $\mathbf{1}^\top C^k \mathbf{1}$, i.e. the sum of all entries of C^k. We compute C^k with binary exponentiation in $O(m^3 \log k)$ per query. This handles arbitrarily large k, but the cubic cost in m restricts it to small N.

Subtask 3 ($N, Q \le 5000$, $k \le 100$): a cheaper step.

We do not actually need the whole matrix. Returning to the dynamic programming, notice that

$$f_t[v] = \sum_u f_{t-1}[u] \cdot depart_u \cdot arrive_v = arrive_v \cdot \Big(\sum_u f_{t-1}[u] \cdot depart_u\Big),$$

so each step costs $O(m)$ instead of $O(m^2)$: compute the scalar $S = \sum_u f_{t-1}[u]\, depart_u$ once, then set $f_t[v] = arrive_v \cdot S$. Iterating k steps gives $O(mk)$ per query, which passes when N is large but k is small. For large k it is still too slow, since it iterates k times.

Subtask 4 ($k = 1$): edge case.

For $k = 1$ a trip is a single edge $u \to v$, so the answer is simply

$$\sum_{u \in [L,R]} \sum_{v \in [L,R]} depart_u \cdot arrive_v = \Big(\sum_u depart_u\Big)\Big(\sum_v arrive_v\Big),$$

two range sums and one multiplication. No repeated multiplication is needed, which is why this case can be handled without any of the machinery above.

The full solution

Method 1: C is a rank-one matrix. Write $\mathbf{d} = (depart_i)_{i \in [L,R]}$ and $\mathbf{a} = (arrive_i)_{i \in [L,R]}$. Then

$$C = \mathbf{d}\,\mathbf{a}^\top, \qquad C[i][j] = depart_i \cdot arrive_j,$$

that is, C is the outer product of two vectors and therefore has rank one. For such a matrix

$$C^2 = \mathbf{d}\,\mathbf{a}^\top\,\mathbf{d}\,\mathbf{a}^\top = \mathbf{d}\,(\mathbf{a}^\top \mathbf{d})\,\mathbf{a}^\top = (\mathbf{a}^\top \mathbf{d})\,C,$$

and by induction

$$C^k = (\mathbf{a}^\top \mathbf{d})^{\,k-1}\, C = \mathrm{Tr}(C)^{\,k-1}\, C,$$

because $\mathbf{a}^\top \mathbf{d} = \sum_i depart_i \cdot arrive_i = \sum_i C[i][i] = \mathrm{Tr}(C)$. Summing all entries of C^k yields

$$\boxed{\text{answer} = \Big(\sum_i depart_i\Big)\Big(\sum_i arrive_i\Big)\ \mathrm{Tr}(C)^{\,k-1}, \qquad \mathrm{Tr}(C) = \sum_i depart_i \cdot arrive_i}$$

with all sums taken over $i \in [L, R]$.

Method 2: expand a single chain.

The same formula falls out with no linear algebra. Take any chain $u_{i_1} \to u_{i_2} \to \cdots \to u_{i_{k+1}}$; its cost is

$$\prod_{j=1}^{k} C(u_{i_j}, u_{i_{j+1}}) = \prod_{j=1}^{k} depart_{i_j}\, arrive_{i_{j+1}} = depart_{i_1} \Big(\prod_{j=2}^{k} arrive_{i_j}\, depart_{i_j}\Big)\, arrive_{i_{k+1}},$$

where we merely regrouped consecutive factors: every interior stop i_j (for $2 \le j \le k$) contributes both an $arrive_{i_j}$, as the head of one edge, and a $depart_{i_j}$, as the tail of the next. Now sum over all chains, i.e. over all independent choices of $i_1, \ldots, i_{k+1} \in [L, R]$. Since the indices are independent, the sum factors into a product of independent sums:

$$\sum_{\text{chains}} \text{cost} = \Big(\sum_{i_1} depart_{i_1}\Big)\Big(\sum_{i_{k+1}} arrive_{i_{k+1}}\Big) \prod_{j=2}^{k} \Big(\sum_{i_j} arrive_{i_j}\, depart_{i_j}\Big),$$

which is exactly $(\sum depart)(\sum arrive)\big(\sum arrive_i\, depart_i\big)^{k-1}$ — the same result, with the common factor $\sum arrive_i\, depart_i$ appearing $k-1$ times.

Computing the three range sums.

The answer depends only on three quantities over $[L, R]$:

$$S_d = \sum_i depart_i, \qquad S_a = \sum_i arrive_i, \qquad S_{da} = \sum_i depart_i\, arrive_i,$$

after which answer $= S_d \cdot S_a \cdot S_{da}^{k-1}$, where S_{da}^{k-1} *must* be evaluated by binary exponentiation in $O(\log k)$ — looping k times is the trap that fails for large k (see Subtask 3). It remains to maintain S_d, S_a, S_{da} on a range under the updates.

- **No updates (prefix sums).** If *depart* and *arrive* never change, precompute prefix sums of $depart_i$, of $arrive_i$ and of $depart_i\, arrive_i$. Each query then answers in $O(\log k)$.
- **One update type (simple segment tree).** Suppose only *depart* receives range-add updates (the *arrive* case is symmetric). Then S_a is still a static prefix sum, while S_d needs range-add / range-sum, a standard lazy segment tree. For S_{da}, adding v to $depart_i$ for all $i \in [l, r]$ increases $\sum depart_i\, arrive_i$ by $v \cdot \sum_{i \in [l,r]} arrive_i$, so a single lazy tag on *depart* combined with the static $\sum arrive$ stored in each node maintains S_{da}.
- **Both update types (the full segment tree).** Now both *depart* and *arrive* get range-add updates. Each node stores, for its segment, the three sums S_d, S_a, S_{da} together with its length len, and carries two lazy tags $lazy_d$ and $lazy_a$. Adding v to every *depart* in the segment updates

  $$S_{da} \mathrel{+}= v \cdot S_a, \qquad S_d \mathrel{+}= v \cdot \text{len},$$

  and adding w to every *arrive* updates

  $$S_{da} \mathrel{+}= w \cdot S_d, \qquad S_a \mathrel{+}= w \cdot \text{len}.$$

  The only subtlety is the cross term: when a node carries both tags, applying the *depart* tag first and the *arrive* tag second — each reading the partial sums already updated by the previous one — yields the correct S_{da}, since the extra $v \cdot w \cdot \text{len}$ contribution is absorbed automatically. Every update and query then runs in $O(\log N)$.

Putting it together, each event costs $O(\log N)$ for the range sums plus $O(\log k)$ for the exponentiation, for an overall complexity of $O((N+Q)\log N + Q \log k)$.

Scientific Committee

The problems given to this competition have been authored and prepared by:

- Ștefan Cosmin Dăscălescu — IIOT International Scientific Committee Representative
- Andrei Chertes — National University of Science and Technology Politehnica Bucharest, Faculty of Automatics and Computers
- Tiberiu Cozma — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Alexandru Gheorghieș — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Matei Ionuț Haba — National University of Science and Technology Politehnica Bucharest, Faculty of Automatics and Computers
- Andrei-Cristian Ivan — National University of Science and Technology Politehnica Bucharest, Faculty of Automatics and Computers
- Andrei Lețu — Babeș-Bolyai University of Cluj-Napoca, Faculty of Mathematics and Computer Science
- Alin-Gabriel Răileanu — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Răzvan-Alexandru Rotaru — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Emanuela Cerchez — "Emil Racoviță" National College, Iași
- Adrian Panaete — "A.T. Laurian" National College, Botoșani
- Marinel Șerban — "Emil Racoviță" National College, Iași
- Szabó Zoltan — Mureș County School Inspectorate
- Luca-Ștefan Camburu — École Polytechnique, Palaiseau - Paris
- Andrei Mihai Ciobanu — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Andrei Enăchescu — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Matei Ionescu — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Cristian Luchian — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Gheorghe Manolache — National College of Computer Science, Piatra-Neamț
- Robert-Sebastian Moldovan — Babeș-Bolyai University of Cluj-Napoca, Faculty of Mathematics and Computer Science
- Bogdan-Ioan Popa — Just 4 Kids Theoretical High School, Bucharest
- Raul Ardelean — Babeș-Bolyai University of Cluj-Napoca, Faculty of Mathematics and Computer Science
- Vlad-Mihai Bogdan — University of Bucharest, Faculty of Mathematics and Computer Science

- Alex-Robert David — Babeș-Bolyai University of Cluj-Napoca, Faculty of Mathematics and Computer Science
- Paul Diac — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Daniel Gheorghe — University of Bucharest, Faculty of Mathematics and Computer Science
- Petruț-Rareș Gheorghieș — National University of Science and Technology Politehnica Bucharest, Faculty of Automatics and Computers
- Alexandra-Ioana Moroz — Babeș-Bolyai University of Cluj-Napoca, Faculty of Mathematics and Computer Science
- Ștefan-Claudiu Neagu — University of Bucharest, Faculty of Mathematics and Computer Science
- Ștefan-Alexandru Nuță — National University of Science and Technology Politehnica Bucharest, Faculty of Automatics and Computers
- Eduard-Lucian Pîrtac — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Robert-Andrei Popa — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Cristian Tanase — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- Gabriel Turbincă — "Alexandru Ioan Cuza" University of Iași, Faculty of Computer Science
- David Vișan — Southern University of Denmark, Sønderborg