Osijek Day 2. Jagiellonian Editorial

A

Let $s_1, \ldots, s_n$ be input strings sorted by length, i.e. $|s_1| \leq \ldots \leq |s_n|$. Consider a pair of strings $\{s_i, s_j\}$ with $i < j$, so $|s_i| \leq |s_j|$. When do they form a k-suspicious pair? If $|s_j| \leq k$ then the situation is trivial.

If $|s_j| > k$, we can consider only replacements of size exactly k in s_j. Indeed if there exists a shorter replacement string, we can always extend it. Let's fix the start position p of the replacement string in s_j. We have a match if:

(a) prefixes match i.e. $s_i[1 \ldots p-1] = s_j[1 \ldots p-1]$;
(b) suffixes match i.e. $s_i[p+k \ldots] = s_j[p+k \ldots]$.

This gives the following solution idea. Fix the longer string s_j and position p. Count how many strings have lower index, matching prefix and matching suffix (using some data structure). Add this to answer.

Such solution is flawed as we might double count some pairs. If the shortest possible replacement string for a pair $\{s_i, s_j\}$ has length $k' < k$ then we will count such pair $k - k' + 1$ times. The solution to this is simple: subtract the count for $k + 1$.

It remains to describe how to do the counting efficiently. The conceptually easiest way is to map each word to a tuple (index, index in lex order, index in lex order of reversed words). Then each query becomes 3D range point counting. This can be solved using sweep-line and 2D segment tree in $O(S \log^2 S)$, where S is the sum of all string lengths. Such solution should pass if implemented efficiently.

To optimize the solution to $O(S \log S)$, we can solve queries for each possible prefix word separately. For a fixed prefix word, map all words with that prefix to a pair (index, index in lex order of reversed words). The total number of generated tuples over all prefixes sums to $O(S)$ and we execute $O(S)$ 2D queries.

In both solutions, one can build a trie for input words and a trie for reversed words to quickly generate tuples and queries.

B

Let's start off with a naive solution and slowly work our way up to the model solution.

For a given drawing D of some tree T that adheres to the rules mentioned in the problem statement let $W(D)$ and $H(D)$ denote the drawings width and height respectively. Let $T(v)$ denote the subtree of T rooted at v.

Let D be some drawing of T. Notice, that upon removing the root R of T from D, as well as all edges incident to it, D becomes two drawings D_1 and D_2 of subtrees of R's children. The bounding boxes of D_1 and D_2 by definition don't intersect, so as long as we enforce the subtrees to be drawn within them their drawings are completely independent. This leads to a natural dynamic programming solution in which for each node v we compute feasible drawings for different sized bounding boxes.

Notice, that it is enough to only consider drawings D of T such that both $W(D)$ and $H(D)$ are less than the number of nodes in T. Wider and higher drawings can always be compressed to fit these restrictions. Moreover if two drawings D_1 and D_2 of the same tree T exist, and $W(D_1) = W(D_2)$ as well as $H(D_1) \leq H(D_2)$ we can discard D_2 as it is never better.

This means, that for a given subtree $T(v)$, we only need to compute, for each possible width from 1 to $|T|$ the smallest required height of a drawing of $T(v)$ with this chosen width. We can merge the solutions of two subtrees $T(a)$ and $T(b)$ in $O(|T(A)||T(b)|)$ time which gives us a $O(n^2)$ solution.

To improve this solution, we can notice, that the areas of optimal drawings are always small. More precisely $O(|T|\log|T|)$ (if we always draw the smaller subtree directly under it's parent, we end up with a height of $O(\log|T|)$ and width $O(|T|)$. This bound is tight, as there exist trees requiring $\Omega(|T|\log|T|)$ area drawings.

This means that the smaller dimension of the optimal solution (and any intermediate solution of some subtree) must be $O(\sqrt{|T|\log|T|})$. If we narrow down our algorithm to only consider such states, we arrive at a $O(|T|\sqrt{|T|\log|T|})$ solution.

This solution also uses $O(|T|\sqrt{|T|\log|T|})$ memory. The authors implementations fit very comfortably under the memory limit, however it may be possible for other implementations to exceed it. In case of such issues one can optimize the memory usage to $O(\log|T|\sqrt{|T|\log|T|})$ by using heavy-light decomposition.

C

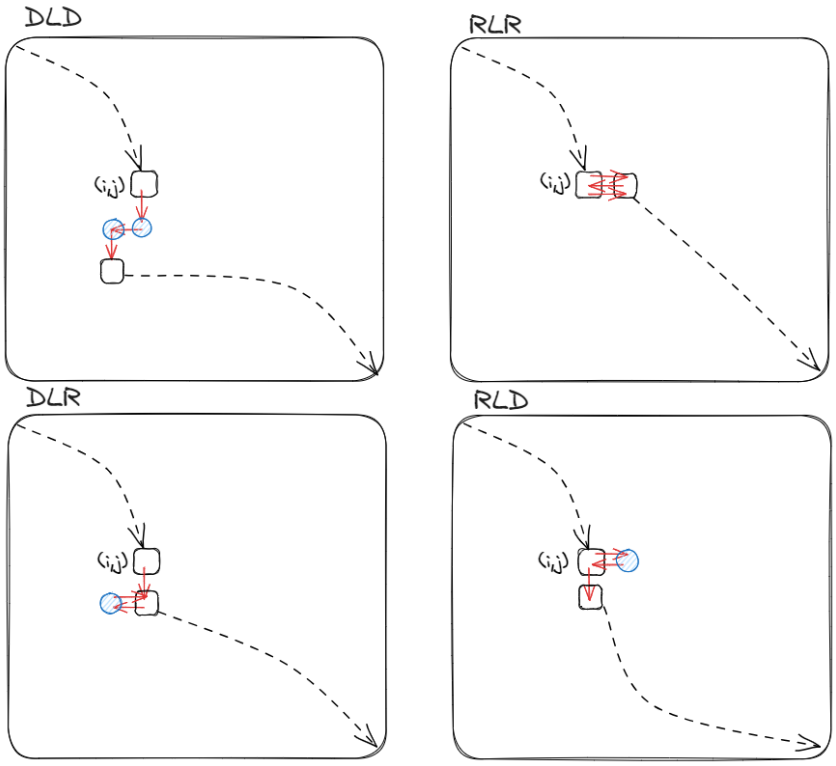
Let

- $A_{i,j}$ - maximum number of diamonds Jack can collect if he only goes right/down, from the top-left corner to (i,j)
- $B_{i,j}$ - maximum number of diamonds Jack can collect if he only goes right/down, from (i,j) to the bottom-right corner.

We can compute $A_{i,j}, B_{i,j}$ for all pairs (i,j) with simple dynamic programming in $O(nm)$.

Let's consider a cell (i,j) which Jack visits two steps before going left. There are four possibilities for the next three moves: RLR,RLD,DLR,DLD. In all four cases, we can compute the maximum number of diamonds Jack can collect as a sum of $A_{i,j}, B_{i',j'}$ and the content of a few other cells (see below).

The answer is the maximum over those values for all the pairs (i,j) and the $A_{n,m}$.



D

Let k be the number of the notes left in the hat when the Lucy is taking one. The values of the notes are a random subset of $\{1, \dots n\}$ of size k.

The expected value of the note with the highest value can be expanded as:

$$E[X_k] = P(X_k > 0) + P(X_k > 1) + \ldots + P(X_k > n)$$

.

$P(X_k \leq l)$ is the probability that the set of values contains only elements from $\{1, \dots l\}$, which is $\frac{\binom{l}{k}}{\binom{n}{k}}$. Since $P(X_k > l) = 1 - P(X_k \leq l)$, we can rewrite the expression above as:

$$E[X_k] = \sum_{i=0}^{n}[1 - P(X_k \leq l)] = n + 1 - \sum_{i=k}^{n} \frac{\binom{i}{k}}{\binom{n}{k}} = n + 1 - \frac{1}{\binom{n}{k}} \left(\sum_{i=k}^{n} \binom{i}{k} \right)$$

.

And then using the Hockey stick identity:

$$E[X_k] = n + 1 - \frac{1}{\binom{n}{k}} \left(\sum_{i=k}^{n} \binom{i}{k} \right) = n + 1 - \frac{\binom{n+1}{k+1}}{\binom{n}{k}} = n + 1 - \frac{n+1}{k+1}$$

.

As each of the k is equiprobable, the answer is

$$E[X] = \frac{1}{n} \left(\sum_{k=1}^{n} E[X_k] \right) = \frac{1}{n} \left[\sum_{k=1}^{n} \left(n + 1 - \frac{n+1}{k+1} \right) \right] = n+1-\frac{n+1}{n} \cdot \left(\sum_{k=1}^{n} \frac{1}{k+1} \right) = n+1-\frac{n+1}{n} \cdot (H_n - 1)$$

We have to compute this expression for many n's so we need to preprocess all the harmonic numbers up to $N = max_i(n_i)$ with trivial dynamic programming. Complexity: $O(N \log P)$

E

Let $S = a + b$. For all pairs $a < b$ we have a recursive relation: $E_{a,b} = 1 + \frac{1}{2} E_{2a,b-a}$ as with probability $\frac{1}{2}$ the process will end, and otherwise it will continue in state $(2a, b - a)$. The only special state is $\left(\frac{S}{2}, \frac{S}{2}\right)$, for which the answer is 1, as the process ends in one step no matter what.

We have a recursion with only one "final" state, so the expected values should depend on whether it is reachable. We will prove it with the following three lemmas.

Lemma 1: If $\left(\frac{S}{2}, \frac{S}{2}\right)$ in unreachable from the (a, b), then $E_{a,b} = 2$.

Informal proof: If it is unreachable, then we have an infinite recursion, which has to loop at some point. It is easy to see that all expected values on the loop are 2, and therefore by induction all the expected values on the "tails" are also 2.

Lemma 2: If $\left(\frac{S}{2}, \frac{S}{2}\right)$ is reachable from the (a, b) in k steps of recrusion then $E_{a,b} = 2 - 2^{-k}$.

Proof: trivial induction.

Lemma 3: *States, that reach the* $\left(\frac{S}{2}, \frac{S}{2}\right)$ *in* $k - 1$ *steps of recursion, are exactly all of the form* $\left(P \cdot \frac{S}{2^k}, Q \cdot \frac{S}{2^k}\right)$ *with* P, Q *odd.*

Proof: We will prove the lemma by induction. $k = 1$ is trivial

1. For the $\left(P \cdot \frac{S}{2^k}, Q \cdot \frac{S}{2^k}\right)$ the next step in the recursion is $\left(P \cdot \frac{S}{2^{k-1}}, (Q - P) \cdot \frac{S}{2^k}\right) = \left(P \cdot \frac{S}{2^{k-1}}, Q' \cdot \frac{S}{2^{k-1}}\right)$
2. For the $\left(P \cdot \frac{S}{2^{k-1}}, Q \cdot \frac{S}{2^{k-1}}\right)$ the only possible previous states are $\left(P \cdot \frac{S}{2^k}, Q' \cdot \frac{S}{2^k}\right)$ and $\left(P' \cdot \frac{S}{2^k}, Q \cdot \frac{S}{2^k}\right)$

From Lemma 3 it follows, that if $\left(\frac{S}{2}, \frac{S}{2}\right)$ is reachable from (a, b), we will reach it in at most $log_2(S)$ steps, so we can simply simulate the process for $log_2(S)$ steps and then print the answer based on Lemmas 1 and 2.

F

Let's assume for simplicity that:

(i) weights of edges are pairwise different;
(ii) differences between pairs of weights are pairwise different.

These properties ensure uniqueness of a solution, which simplifies the proof. They can be guaranteed by adding a formal ε^k to the k-th edge weight (i.e. ties are resolved using edge indices). We will later see it is not needed in the final solution.

Convexity

Let $k_{\min}$ and $k_{\max}$ be the minimum and maximum possible number of red edges in a spanning tree of G, respectively. Let $f(k)$ denote the minimum possible weight of a spanning tree that contains exactly k red edges. We claim that $f(k)$ is:

(i) defined for all $k \in \{k_{\min}, \ldots, k_{\max}\}$;
(ii) convex, i.e. $f(k) \leq (f(k-1) + f(k+1))/2$.

We now prove these claims; you can skip ahead if you don't care. We start with the following lemma:
Lemma 1. *Let S and T be spanning trees of G. For each edge $e \in S - T$, there exists an edge $f \in T - S$, such that both $S - e + f$ and $T - f + e$ are spanning trees of G.*

Proof. Let A and B be connected components of graph $S - e$. Let C be a cycle in graph $T + e$. Let $f \neq e$ be any edge in cycle C that connects vertices from A and B (it always exists, because e connects A and B). $S - e + f$ is a spanning tree, because edge f connects two components of $S - e$. $T - f + e$ is a spanning tree, because $T + e$ contains only one cycle and f lies on it. □

The above lemma is known in matroid theory as *Symmetric basis-exchange property*. Let $w(e)$ denote weight of an edge e and $w(T)$ denote weight of a spanning tree T. Now we can prove convexity:
Lemma 2. *Let S be a spanning tree of minimum weight that contains exactly $k-1$ red edges. Let T be a spanning tree of minimum weight that contains exactly $k+1$ red edges. There exists a spanning tree Q such that $w(Q) \leq (w(S) + w(T))/2$ that uses exactly k red edges.*

Proof. Let $e \in T - S$ be a red edge. Such edge always exists, because T has more red edges than S. By lemma 1, there exists an edge $f \in S - T$ such that both $T - e + f$ and $S - f + e$ are spanning trees. We claim that this edge must be blue. Suppose for the contrary that it is red. We have two cases:

(i) $w(e) < w(f)$: then $w(S - f + e) < w(S)$ and $S - f + e$ has $k - 1$ red edges – contradiction with minimum weight of S;
(ii) $w(e) > w(f)$: then $w(T - e + f) < w(T)$ and $T - e + f$ has $k + 1$ red edges – contradiction with minimum weight of T.

We have established that e is a red edge and f is a blue edge. Moreover, $w(S - f + e) + w(T - e + f) = w(S) + w(T)$, so one of these spanning trees has weight at most $(w(S) + w(T))/2$ and has exactly k red edges. □

Lemma 3. *There exists a spanning tree with k red edges for each $k \in \{k_{\min}, \ldots, k_{\max}\}$.*

Proof. Let $x < y$. If spanning trees with x and y red edges exist, then spanning trees with $x+1$ and $y-1$ red edges also exist. This can be proved similarly as in lemma 2. The lemma follows by induction. □

Alien's?

If we were asked to compute the answer for a single k, we could use the lambda optimization (also known as the "trick from Alien's"). If you don't know it, you should read about it before continuing.

Let λ be an integer parameter that we subtract from weight of each red edge. Consider MST of such modified graph. If $\lambda = -\infty$, we get MST for $k_{\min}$. If $\lambda = +\infty$, we get MST for $k_{\max}$. By binary searching value of λ, we can get MST for any k. Doing this independently for each possible k is unfortunately too slow.

Let's analyse how the MST changes for various λ. Consider how Kruskal's algorithm work. Let $r_1, \ldots, r_s$ and $b_1, \ldots, b_t$ be red and blue edges respectively, sorted by increasing weights. The sequence of edges considered by Kruskal's algorithm is the result of merging these two sequences. The relative order between red and blue edges depends here on λ.

There are a few options what can happen to a red edge r_i:

(a) it never belongs to an MST;
(b) it always belongs to an MST;
(c) there exists a λ_0 such that r_i is not included in MST for $\lambda < \lambda_0$ and it is included for $\lambda > \lambda_0$.

The λ_0 in case (c) corresponds to the situation when r_i swaps its position with some blue edge b_j in the sequence of edges considered by Kruskal's algorithm. This means $\lambda_0 = w(r_i) - w(b_j)$. We assumed that differences between weights are pairwise different, so nothing else changes in the sequence. We can deduce that always exactly one of r_i and b_j belongs to a solution. We will refer to such pairs of edges as *exchange pairs*.

Each red edge from case (c) belongs to exactly one exchange pair. The same holds for blue edges, so exchange pairs form a matching. If we could quickly find all exchange pairs, we could sort them by increasing weight differences and recover solution (thanks to convexity).

Finding exchange pairs

Suppose there are only edges of type (c) (handling cases (a) and (b) is left as exercise). In this case, the number of red and blue edges must be the same, because exchange pairs form perfect matching.

Let's use divide and conquer. Let's split the sequence of blue edges in the middle into two parts:

(i) the "left" part $b_1, \ldots, b_k$;
(ii) the "right" part $b_{k+1}, \ldots, b_t$.

We want to decide for each red edge if it is paired with blue edge on the left or on the right. We can do it by running Kruskal's algorithm on sequence of edges: $b_1, \ldots, b_k, r_1, \ldots, r_s$. The red edges included in the produced spanning tree must be paired with blue edges on the right.

Now we want to recursively solve the left and right half, but there is a problem: the above classification algorithm won't work if we just split the edge set. It is easy to see for the right part: we cannot forget about $b_1, \ldots, b_k$. To fix this, we can compress graph by contracting these edges. Then we can solve recursively on compressed graph with only "right" red and blue edges.

For the left side, we can do a similar thing. Here we need to contract the "right" red edges as they will always be included in MST.

We can implement both compression and Kruskal's test using DSU with rollbacks. The complexity of the procedure is then $O(n \log^2 n)$. Now notice that the algorithm depends only on the sorted order of edges. Thus the initial assumption about uniqueness of weights is not necessary.

G

Let's denote the number of refuellings we need when traveling from u to v by $r(u, v)$, and the distance between u, v by $d(u, v)$ We will start with a simple lemma:

Lemma $r(u, v) = r(v, u)$

Proof WLOG let's assume $r(u, v) < r(v, u)$. Let $a_1, ...a_k$ be the cities in which we refueled on the way from u to v. This means that all the consecutive cities in the sequence $u, a_1, ...a_k, v$ are in distance at most c from each other. By reversing the sequence, we prove that it is possible to travel from v to u with at most k refuellings.

Let's root the tree, and then for each vertex compute $refill(v)$ – what is the first vertex in which we will have to refill when traveling from the v to the root. We can do it with binary lifting, as the $refill(v)$ is in fact the last vertex on the path to root, that is in distance at most c (tank capacity) from the v.

Then we can then build the binary lifting data structure on the $refill(v)$ function. If v is ancestor of u we can easily answer:

1. what is the value of $r(u, v)$
2. what is the last city in which we had to refuel.

How do we compute $r(u, v)$ if the v isn't ancestor of u? Let $l = lca(u, v)$ and u', v' the last city in which we had to refuel when traveling from u to l and v to l resp.

We can use the fact that $d(u', v') < 2c$ to compute $r(u, v')$:

1. if $u' = v'$ then $r(u, v') = r(u, u')$
2. if $d(u', v') \leq c$ then $r(u, v') = r(u, u') + 1$
3. otherwise $r(u, v') = r(u, u') + 2$

Since when traveling from v to u we will refuel in the v' then $r(v, u) = r(v, v') + r(v', u) + 1 = r(v, v') + r(u, v') + 1$ (beware of corner case when $v = v'$).

Building all the data structures will take $O(n \log n)$, and we need $O(\log n)$ to answer each of the queries, so we end up with complexity $O((n + q) \log n)$

H

Let $L(v, u)$ be the longest path that starts at vertex v and the first edge that it goes through is (v, u). To compute $L(v, u)$ it would be enough to know all $L(u, w)$ such that $w \neq v$. We can recursively compute those values before computing $L(v, u)$. This recursion will not have any circular dependencies, due to the fact, that no orientation of the graph has a cycle. The rest of the problem boils down to finding an easy implementation. An easy and straightforward way is to keep a queue of w's for each v such that $L(w, v)$ is waiting to be computed. Whenever we compute a $L(v, u)$ we need to remove the appropriate w's from the queue and compute them.

Using the above approach we get an $O(n + m)$ solution. There exist alternative solutions that decompose the graph into trees of undirected edges and DAGs of directed edges and later use some dynamic programming, also in $O(n + m)$

I

Let $[1, A]$ range from which we generate numbers, and let's sort the sequence from the input.

For each a_j we'll compute $\sum_{i=1}^{n}(a_i \mod a_j)^2$ separately and sum up the answers.

For a fixed t, let $A_{j,t} = \{i : \lfloor \frac{a_i}{a_j} \rfloor = t\}$. Note that this set will be a range because we sorted the sequence. We can easily find the contribution to the sum above of the $i \in A_{j,t}$. If $A_{j,t}$ is a range $[p, q]$, then:

$$\sum_{i \in A_{j,t}} (a_i \mod a_j)^2 = \sum_{i \in A_{j,t}} (a_i - t \cdot a_j)^2 = \sum_{i=p}^{q} a_i^2 - 2a_j t \cdot \left(\sum_{i=p}^{q} a_i \right) + (a_j t)^2 \cdot \left(\sum_{i=p}^{q} a_i^0 \right)$$

We can find p and q with a binary search, and then given we precomputed prefix sums of the a_i^k for $k \in \{0, 1, 2\}$, we can compute the expression above in $O(1)$.

The only thing left is to notice that for a given a_j there are not many t's for which the $A_{j,t}$ isn't empty. The largest interesting t is $\frac{A}{a_j}$. The a_j was uniformly randomly chosen from $[1, A]$, so:

$$E\left[\frac{A}{a_j}\right] = \frac{1}{A} \cdot \left(\sum_{i=1}^{A} \frac{A}{i} \right) = \sum_{i=1}^{A} \frac{1}{i} = O(\log A)$$

and we end up with the expected runtime: $O(n \log n \log A)$.

J

Let's call $s_1, s_2 - s_1, \ldots, s_n - s_{n-1}$ the difference sequence of sequence s.

A non-negative sequence s is non-decreasing if and only if $D(s)$ is made up of non-negative numbers.

Similarly a non-negative sequence s is non-increasing if and only if $D(s)_1$ is non-negative, all other elements are non-positive and the sum of all elements is non-negative.

Notice, that summing up monotonic sequences of one type (either non-increasing or non-decreasing) always yields a monotonic sequence of the same type.

The sum of sequences is equal if and only if the sum of their difference sequences is equal.

Using the above facts, we can see that the problem is equivalent to the following: Given a difference sequence a' of some non-negative sequence determine if it can be expressed as the sum of the difference sequence of a non-negative, non-increasing sequence b' and the difference sequence of a non-negative non-decreasing sequence c'.

This is the same as determining if there exists a c' such that subtracting it from a' yields a non-negative sequence. WLOG we can assume that $c'_1 = a_1$ as this does not restrict b' nor the other elements of c' more than any other choice of c'_1. We can now see, that a valid c' exists if and only if the sum of negative entries in a' plus c'_1 is non-negative (by definition).

Adding an arithmetic progression with parameters s and d on a range $[p, q]$ of a is the same as adding s to a'_p, d to $a'_{p+1}, \ldots a'_q$ and $-(s + (q - p)d)$ to a'_{q+1}. Notice, that since $d, s \geq 1$ only the value added to a'_{q+1} will be negative.

To keep track of the sum of negative entries of a' let's keep two segment trees - a segment tree with negative entries of a' and a segment tree with positive entries. Every time a value switches its state from negative to positive or vice versa we can naively move it to the other segment tree. This will amortize to $O(n+q)$ moves, since each arithmetic sequence that we add only adds a negative number to one entry of a'. Checking if an update switches the state of an element we will use a tree that supports adding a value on a range and querying for the max / min in the whole tree. This results in a total time complexity of $O((n + q) \log n)$

K

Intuition The definition of *similar* pairs of subtrees may look scary, but one intuition behind it is the following: consider the input tree to represent some kind of an abstract expression (e.g. a lambda term), and projects to represent variables. A project being owned by a particular employee maps to a variable being local in a given subtree. The notion of similarity corresponds to two subtrees being the same modulo renaming local variables (in lambda calculus this is called α-equivalence). Thus, the problem can be (roughly) equivalently stated as: for each subtree in an expression, find the number of other subtrees that are equivalent to it modulo renaming of local variables.

Solution: rough idea It is easy to see that the notion of similarity is an equivalence relation (that is, it is transitive and symmetric), thus all the subtrees can be divided into equivalence classes. The idea is to compute a *hash* for each subtree, such that two subtrees are similar if they have the same hash (modulo hash collisions of course, which can be abundant if we use only a single hash with $MOD = \mathcal{O}(10^9)$, but two hashes with different MODs are enough). We can then simply count the number of occurrences of each hash. The solution mostly pertains to designing an appropriate hash.

Solution 1: "Editing" polynomial hashes The most straight-forward solution is to track a hash describing both the shape of a subtree and the identities of the variables (projects), but "edit" it when a given variable (project) becomes local (owned). Intuitively, while the variable is not local, the hash should record its exact ID (i.e. the p_i we are given in the input), but once it becomes local we should "forget" the exact value of p_i, and instead replace it with a *canonical identifier*, to ensure two subtrees that contain local variables "in the same places" get assigned the same hash. However, editing hashes is tricky; to do this we can use a very particular hash: encode a subtree as a sequence (describing both e.g. up/down edges, to record the shape, as well as values p_i, to record the variable IDs) and use a polynomial hash (i.e. $a_0 + B \cdot a_1 + B^2 \cdot a_2 + ...$). Then, given a hash representing a sequence, we can recompute the hash after a value at a particular index is changed (without having to actually store that sequence, as long as we can figure out the indices). It remains to fill in some remaining details, e.g. how to compute a canonical identifier to use for a local variable (hint: it has to uniquely "point" at where that variable appears; for that, it may be useful to e.g. hash some kind of an offset between the node where the variable became local and its first occurrence in the subtree below). This solution works in $\mathcal{O}(n \log n)$ time (or even $\mathcal{O}(n)$ with some care if we use hashmaps).

Solution 2: Smaller-to-larger There is also a different solution, which is based on the smaller-to-larger trick. This idea (but in the language of lambda expressions) is described in Hashing Modulo Alpha-Equivalence, although it has to be slightly generalized here (unlike lambda expressions, the trees in this problem could have nodes with more than 2 children). We quickly summarize it below.

The idea is to track two *kinds* of subtree hashes: a *structure hash* that describes the shape of the tree, including the identities of the variables (projects), but *only the ones already owned in a given subtree*, and a *variable map*, which maps identifiers of non-local variables (projects that are not yet owned) to hashes describing where these variables appear. To form a hash of a subtree, it is enough to combine the structure hash with the hash of the variable map. To hash a map, we need something that is invariant to the order of key-value pairs in it, and also can be "edited" in constant time when the map is changed; one option is to take a *bitwise XOR* of the hashes of entries (or really any invertible permutation-invariant combiner, e.g. a regular sum). When we merge hashes from children, we need to merge the variable maps; this can be done with smaller-to-larger. After filling in some remaining details, we get a solution working in $\mathcal{O}(n \log^2 n)$ time, which is enough to get OK.

L

First, for each point P_i let's sort all the other points in the clockwise order around it, and keep the result in vector V_i.

Then for each query let's extend the sides of the queried triangle into infinity as shown in the picture below. The plane around the triangle is now divided into three disjoint areas. Each of these areas is made up of points whose angle relative to the appropriate vertex P_i is in some fixed range. We can use binary search on vector V_i to find the number of the points in this range.

If in total there are $n - 3$ points in these areas, there is no point inside of the triangle, so the answer is YES, otherwise it is NO.

Complexity: $O(n^2 \log n)$ for preprocessing and then $O(\log n)$ per query.