

Observation 1

Consider some valid labeling of the tree. We will define $l(x)$ as node x 's label. The labeling achieves the minimal lostness if and only if, after rooting the tree at the node with label 1, for every $1 \leq i \leq n$ other than the one for which $l(i) = 1$ holds,

$$l(\text{par}_i) < l(i)$$

where par_i represents the parent of the i -th node and the array v represents the labeling of the tree.

We can prove this by first observing what is the minimum value of each term in the expression we use to calculate the lostness. Namely, for a path between 2 nodes u, v with labels a, b respectively, the minimum amount the path itself contributes to the lostness will be $\max(a, b)$.

For each finished tree which satisfies Observation 1 (from now on we will call observation x O x) we see that every term in its lostness will be minimal. This can be proved inductively. Namely, we observe that the proposition holds for a tree of size 1 and for a tree of size n , we notice that the node with the label n will be a leaf, so the only paths containing it will be paths which have it as an endpoint. Afterwards we remove the largest labelled node and continue. If we imagine running this algorithm till the end, we see that for a path between nodes with label a, b , its value will be fixed precisely when the node $\max(a, b)$ is reached, meaning every term will be minimal.

For a tree which does not satisfy O1, while trying the same algorithm we discussed in the paragraph above, there will be a node which will at some point have the largest label and will not be a leaf. We now see that this will cause some terms to not be minimal.

Using this observation we can implement a $O(2^n)$ bitmask DP algorithm. While this may not seem promising for a proof of such length, it is not hard to extend it to $O(n^2)$. Remember the algorithm we used to prove O1, it will be very useful.

Observation 2

Consider a rooted tree of size n , rooted at the node with label 1 and without a node with label n . Consider all leaves of the tree. Now find the unlabeled leaf with the largest labelled ancestor, i.e. whose ancestor has the largest label. Let's call this leaf w . Without loss of generality, it is optimal to label w with n .

Suppose this is not the case and there exists a valid tree satisfying O1, where label n was not placed in w , rather it was placed in a leaf k . This means some other label was placed in its place, which was smaller than n . Now suppose the algorithm was ran and that it came to a point where a node was placed in w . We will run a *DFS* from n and stop when we arrive at a node which was labeled at the beginning or at a node which hasn't been assigned a label yet. Now consider the resulting component. Now we will place the node we placed now in w into the node with the smallest depth in this component. As for the other labels in the component besides n , we see that we can relabel the tree with them without violating O1. We place n . The resulting tree is valid, so we have proven O2.

From this rather convoluted proof we obtain an $O(n \log n)$ when label 1 is fixed. We calculate the largest labeled ancestor for each node with a *DFS* and then run the algorithm. In every step all we are required to do is maintain the leaf with the largest labeled ancestor. To figure out whether the answer is YES or NO, we just gotta print NO when we arrive at a label which is already fixed, but which is not a leaf.

But, the label 1 is not fixed in the general case. So we have to figure out more observations.

Observation 3

Let's call the smallest labeled node g . Root the tree at g . Run a *DFS* from g , stopping when a labeled node is reach. Now consider the component formed by doing so. In the final tree, the label 1 will be in one of those components. Moreover, without loss of generality, a neighbor of g will have the label 1.

First of all, no node outside of the component can have label 1. If this were not the case, then there would be a node with label greater than g 's label in between label 1 and g , which would be a contradiction by O1 and the definition of g , as that node will surely have a label greater than 1 and g 's label.

Now, we run the algorithm as we did before. When it's g 's turn to be deleted, it must be a leaf, so only one neighbor of g must remain. In what remains of its subtree, there are no labels, so we can put label 1 wherever we want to. We can just place it into the node that's a neighbor of g .

With this, we can solve Subtask 5. This is where we stopped during the problem presentation.

The full solution

We root the tree at g . First of all, we will run the algorithm we used to solve the previous subtasks, until the largest ancestor of any unlabeled leaf is g . Now we consider all the nodes in the component we defined earlier. Our goal is to remove all but one of the subtrees of g .

In general, a node can be removed when all but one of its neighbors is removed. In this case, because in this rooted tree a node's parent won't be removed before the node, except when g is removed, a node can be removed when its subtree in this rooted tree is removed. This condition can be simplified. Define $d(x)$ as the minimum labeled descendant of node x . Node x can be removed when $d(x)$ is removed.

If we only remove nodes considering this condition, it is possible to initially remove some non-leaf nodes. But this doesn't matter because there still exists some order of labeling them which does not do that. If we assume for some node a that its ancestor b was removed before it, then we can just swap them in the order of labeling.

We can prove this satisfies this condition with a bit of inequalities. Denote $l(x)$ as node x 's assigned label. Node a was labeled after b , so its label is smaller, therefore

$$l(a) \leq l(b)$$

Node b is a 's ancestor, so whatever labeled descendants a has, b has them too. From this we know that b will also have $d(a)$ among its descendants, by function d 's definition. We conclude that:

$$d(b) \leq d(a)$$

Because of O1 we can also conclude that

$$l(a) \leq d(a), \quad l(b) \leq d(b)$$

So if we swap $l(a)$ and $l(b)$, i.e. do $l(a) := l(b)$ and $l(b) := l(a)$ simultaneously, O1 will still hold because before the swap the following inequalities were true:

$$l(a) \leq l(b) \leq d(b)$$

$$l(b) \leq d(b) \leq d(a)$$

In the full solution, we pretty much have to try every neighbor of g to check if 1 can be in it. To be able to do this, we need to simplify this condition even further. To check if we can assign the remaining labels, we will first define the array a of length n . For every node x in g 's component we will add 1 to $a_{d(x)}$.

Now notice that while running the algorithm, even after the moment when the maximum labeled descendant of any leaf is g , if we continue running it past that there will still be some leaves whose maximum labeled descendant isn't g . We will continue the algorithm until running into g again and every time the maximum labeled descendant is g we will subtract 1 from a_s , where s is the current label which we are usually supposed to put somewhere while running the algorithm.

Now we will define array p as the suffix sum of a . Formally:

$$p_i = \sum_{j=i}^n a_j$$

Now notice that if $p_i < 0$ holds for any $1 \leq i \leq n$, the answer is NO. This is because we have nowhere to place label i while running the algorithm. Otherwise, if every $p_i \geq 0$, the answer is YES. This follows from what we have proved earlier.

To simplify this and to only have to check 1 index of p_i , we define another array m as the prefix minimum of p . Formally:

$$m_i = \min_{j=1}^i p_j$$

Now the answer depends solely on m_n .

All that remains is to try every neighbor of g to see if for one of them the answer is YES. After we have built the array p_i with all neighbors in consideration, we have to figure out how to remove a neighbor and its subtree in g 's component.

Then we notice that if we remove a node x then that will subtract 1 from all p_i such that

$$i \leq d(x).$$

This will have the same effect on the m array.

To check every neighbor's subtree, we first run a *DFS* from the neighbor we are currently trying and again stop when we reach some labeled node. In this *DFS*, we will remember all of the $d(x)$ we encountered and put them in a priority queue.

Now we will iterate from the greatest element in the priority queue to the smallest one and keep a value *total*. For every element y , we will add 1 to *total* and then afterwards check if

$$m_y - total \geq 0.$$

This is correct because by doing this we are checking every possible place where some element of m could've been changed. If we visualise what happened when we were removing nodes, then that is equivalent to there being multiple prefixes of m in which 1 was subtracted from them. We checked all of the prefixes where 1 was subtracted, so this algorithm is correct.

All of this yields a $O(n \log n)$ algorithm, where the log factor comes from using a priority queue.