Zuha Aqib 26106

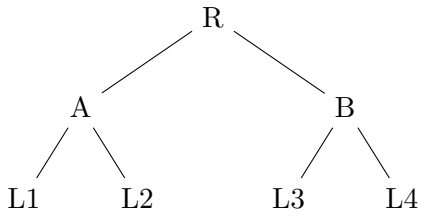
Question 1

We are given a complete binary tree where each internal node performs a `NAND` operation on its two children, and the leaves contain Boolean values (either 0 or 1). The value of the root is computed by evaluating the tree bottom-up using the `NAND` operation.

Our task is to find the smallest number of leaf values that must be revealed (or read) to determine the value of the root node. The question also tells us how a minimal proof works: if a node evaluates to **True** (i.e., 1), then it's enough to know that at least one of its children was 0; but if a node evaluates to **False** (i.e., 0), then we must know that both of its children were 1, since that is the only way for `NAND` to produce a 0.

A	B	A `NAND` B
0	0	1
0	1	1
1	0	1
1	1	0

To understand what's going on, it helps to look at a small example. Suppose the depth of the tree is $k = 2$. Then the tree has $2^2 = 4$ leaves. It would look like this:



Here, L1, L2, L3, and L4 are the leaves. Node A computes `NAND`(L1, L2), and node B computes `NAND`(L3, L4). The root R then computes `NAND`(A, B).

Let's say we are given values for the leaves and want to determine the value of the root. We might think we need all four leaves, but that's not always true. For example, suppose:

$$\text{L1} = 1, \quad \text{L2} = 1 \Rightarrow \text{A} = \texttt{NAND}(1, 1) = 0$$

$$\text{L3} = 0 \Rightarrow \text{B} = \texttt{NAND}(0, \_) = 1 \quad \text{(we don't even need L4 since one 0 is enough)}$$

$$\text{R} = \texttt{NAND}(0, 1) = 1$$

In this case, we only needed to check L1, L2, and L3. That's three leaves, not all four.

But what if we try to do even less? Try this:

$$\text{L1} = 0, \quad \text{L3} = 0$$

Then A = NAND(0, _) = 1, and B = NAND(0, _) = 1. Since both A and B are 1, then R = NAND(1,1) = 0. That means we were able to determine the value of the root just by looking at two leaves: L1 and L3.

This is the kind of thing we want: the smallest number of leaves that are strictly necessary to compute the root value.

Here's the general idea. Because of how NAND works, proving that a node is 1 (i.e., True) only requires you to show that one of its children is 0 — it doesn't matter what the other child is. But if a node is 0, then you must show that both children are 1, since NAND(1,1) is the only way to get 0.

Now suppose we have a bigger tree, say of depth $k = 4$. Then there are $2^4 = 16$ leaves. If we were to evaluate the entire tree, we'd need to look at all 16 values. But we don't always need to. Just like in the smaller example, we can try to be clever and stop early.

The trick is that the **proof tree**, i.e., the part of the tree that we actually need to explore to determine the value of the root, forms a smaller subtree of the full tree. We only visit nodes when absolutely necessary.

Let's suppose the root of the tree is 0. That means both its children must be 1. And for each of those children, to prove they are 1, we only need to find a single 0 child. This pattern continues. Every time we need to prove a node is 1, we look at just one child — the one that is 0. But if we need to prove a node is 0, we must look at both of its children, each of which must be 1.

So the worst case, the part that costs us more, is proving a node is 0, because we're forced to explore more. But proving a node is 1 only needs a "witness" — a single 0 somewhere below.

Now, here's the key idea: because of this alternating structure (sometimes proving both children, sometimes only one), we only end up branching fully every **two levels**. That is, if the original tree has depth k, then the structure of the proof tree — the part we actually traverse — ends up having depth $k/2$.

For example, at the root (depth 0), suppose it's 0. Then we look at both children (depth 1), and for each child, we only need one 0 grandchild (depth 2). Each of those grandchildren that is 0 would need both of its children to be 1, and so on. We alternate between "look at both children" and "look at just one."

Because of this pattern, the size of the proof tree (number of leaves we actually visit) ends up being:

$$\text{Number of leaves in a full binary tree of depth } \frac{k}{2} = 2^{k/2}$$

This gives us the final result. Even though the full tree has 2^k leaves, the minimum number of leaves we need to observe in order to determine the value of the root is exactly:

$$\boxed{2^{k/2}}$$

as long as k is even.

To summarize, the number of leaves in the minimal proof comes from how often we are forced to check both children of a node (when it's False), and when we can get away with just one child

(when it's True). Because of this pattern, we only fully branch out every two levels. That means the proof tree is a full binary tree of half the depth, i.e., depth $k/2$, and hence has $2^{k/2}$ leaves.