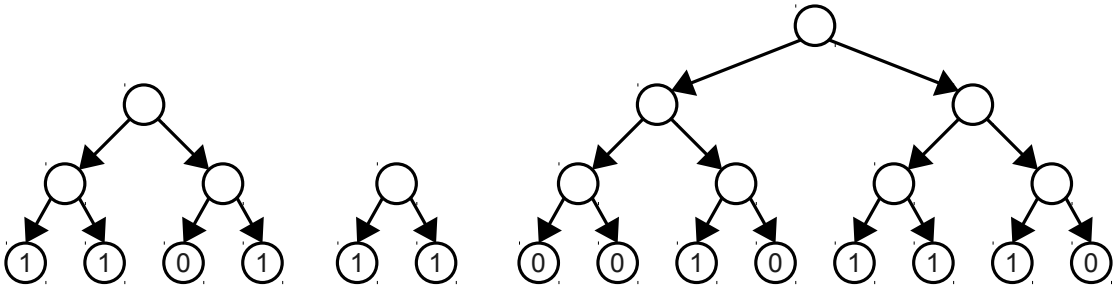
Problem Two: Evaluating NAND Trees

A *NAND tree* is a complete binary tree with the following properties:

- Each leaf node is labeled either 0 or 1.
- All internal nodes are *NAND gates*. A NAND gate is a logic gate that takes in two inputs and evaluates to 0 if both its inputs are 1 and to 1 if either input is 0.

We can *evaluate* a NAND tree by computing the value of the top-level NAND gate in the tree, which will evaluate either to 0 or to 1. (If the tree is a single leaf, the tree evaluates to the value of that leaf.) For example, the left and right trees below evaluate to 1; the middle tree evaluates to 0:



Here is a simple recursive algorithm for evaluating a NAND tree:

- If the tree is a single leaf node, return the value of that node.
- Otherwise, recursively evaluate the left and right subtrees, then apply the NAND operator to both of those values.

This algorithm takes $\Theta(n)$ time to evaluate a NAND tree with n leaf nodes. We can improve this algorithm using *short-circuiting*. If one subtree of node v evaluates to 0, then v must evaluate to 1 because 0 NAND 0 = 1 and 0 NAND 1 = 1. Therefore, we don't need to evaluate v's other subtree. This gives the following algorithm, which we'll call the *left-first algorithm*:

- If the tree is a single leaf node, return the value of that node.
- Otherwise:
  - Recursively evaluate the left subtree.
  - If it evaluates to 0, return 1.
  - Otherwise, recursively evaluate the right subtree.
  - If it evaluates to 0, return 1; otherwise return 0.

In many cases, the left-first algorithm runs faster than the $\Theta(n)$-time naïve algorithm. However, it is possible to construct NAND trees for which the left-first algorithm runs in time $\Theta(n)$.

i. **(8 Points)** Design an algorithm that creates a NAND tree T with $n = 2^k$ leaf nodes such that the left-first algorithm never short-circuits when evaluating T. Your algorithm should run in time polynomial in n. Then:

   - Describe your algorithm.
   - Prove that your algorithm produces a tree T with n leaves such that the left-first algorithm never short-circuits when evaluating T.
   - Prove your algorithm runs in time polynomial in n.

Since the left-first algorithm never short-circuits on inputs produced by your algorithm, the left-first algorithm has a worst-case runtime of $\Theta(n)$.

More generally, *any* deterministic algorithm for evaluating a NAND tree will have at least one input that causes it to run in $\Theta(n)$ time, but you don't need to prove this.

Despite the $\Theta(n)$ worst-case for deterministic evaluation algorithms, there is a simple *randomized* algorithm for evaluating NAND trees that, on expectation, does less than $\Theta(n)$ work. The idea is simple: use the same algorithm as above, but choose which subtree to evaluate first uniformly at random. We'll call this the *random-first algorithm*. More concretely:

- If the tree is a single leaf node, return the value of that node.
- Otherwise:
  - Choose one of the subtrees of the root at random and evaluate it.
  - If the value is 0, return 1.
  - Otherwise, recursively evaluate the other subtree.
  - If the value is 0, return 1; otherwise return 0.

To determine the runtime of the random-first algorithm, we will introduce *two* recurrence relations. Let $T_0(n)$ be the *expected* runtime of the random-first algorithm on a tree with n leaf nodes assuming the root evaluates to 0. Let $T_1(n)$ be the *expected* runtime of the random-first algorithm on a tree with n leaf nodes assuming the root evaluates to 1.

ii. **(6 Points)** Prove that the following recurrence relations for $T_0(n)$ and $T_1(n)$ are correct:

$$T_0(1) \leq \Theta(1)$$
$$T_0(n) \leq 2T_1(n / 2) + \Theta(1)$$

$$T_1(1) \leq \Theta(1)$$
$$T_1(n) \leq \tfrac{1}{2}T_1(n / 2) + T_0(n / 2) + \Theta(1)$$

iii. **(6 Points)** It turns out that $T_1(n) \leq T_0(n)$, though it's somewhat difficult to formally establish this. Using this fact, prove that $T_0(n) = O(n^\varepsilon)$ for some $\varepsilon < 1$. You can assume $n = 4^k$ for some natural number k. *(Hint: Write* $T_0(n)$ *in terms of itself.)*

Your result from (iii) proves that the random-first algorithm has expected sublinear runtime on *all* inputs, since $T_1(n) \leq T_0(n) = O(n^\varepsilon) = o(n)$. This is one of a few known problems where the best randomized algorithm is more efficient on expectation than the best deterministic algorithm in the worst case.

The last part of this problem explores this question: what happens if you try to evaluate a randomly-chosen NAND tree? The result is surprising.

Let's say a *random NAND tree* with $n = 2^k$ leaves is a NAND tree where each leaf is independently assigned a value of 0 or 1 uniformly at random.

iv. **(4 Points)** Let $P_0(n)$ denote the probability that a random NAND tree with n leaves evaluates to 0 and $P_1(n)$ denote the probability that a random NAND tree with n leaves evaluates to 1. Write recurrence relations for $P_0(n)$ and $P_1(n)$ and briefly explain why your recurrences are correct.

The recurrence relations you came up with in (iv) can't be solved using the techniques we've developed in this course, but you can easily write a short computer program to determine their values by writing out $n = 2^k$ and evaluating the recurrence for increasing values of k. If you do, you'll find that when $k \geq 15$, $P_0(n)$ is extremely close to 1 if k is even and $P_1(n)$ is extremely close to 1 if k is odd. Consequently, the algorithm "return the height of the tree modulo 2" returns the right answer with high probability in time $\Theta(\log n)$, even though it never actually evaluates the tree!