Exercise 1 (NAND-tree evaluation). Let us consider a complete binary tree of height $h$, whose internal nodes are labeled NAND and whose $2^h$ leaves are assigned unknown binary values. The value of the tree is recursively defined as the NAND (where $x$ nand $y = \neg(x \land y)$) of the values of its two subtrees. We want to compute the value of tree while inspecting as few leaves as possible.

Question 1.1) Compute the value of the tree in Fig. 1. Show that it is not necessary to inspect all the leaves to compute this value.

Answer. The value of the tree is 1 as shown below.

Note that inspecting only the 2 greyed leaves is enough to evaluate the root!

Question 1.2) Show that for every deterministic algorithm, there exists a assignment of the leaves that forces the algorithm to inspect all the leaves.
Hint. Show by recurrence on \( h \) that an adversary can assign values of the leaves as they are inspected by the algorithm in such a way that the value of the tree depends on the value of the last leaf inspected.

Answer. We proceed by recurrence on \( h \) to show that there is an adversarial strategy that assigns values to the leaves as they are inspected so that the value on the tree depends on the value of the last leaf inspected.

For \( h = 0 \), the tree consists in a single leaf and its value is the value of leaf. Assume now that the property is true for trees of height \( h - 1 \) and consider a tree \( T \) of height \( h \). We apply by recurrence hypothesis the adversarial strategy in both trees \( T_1 \) and \( T_2 \) until the last leaf of them is inspected.

Without loss of generality, \( T_1 \) is the first to have its last leaf inspected, we then choose the value of this leaf so that \( T_1 \) evaluates to 1, the value of \( T \) is then the negation of the value of \( T_2 \) which depends on the value of the last leaf inspected in \( T_2 \) by recurrence hypothesis, which is also the last leaf inspected in \( T \).

Now, since the value of the tree depends on the value of the last leaf inspected for the instance built by the adversary, the deterministic algorithm has to inspect all the leaves to determine the value of the tree for this instance.

We consider the following simple randomized algorithm, which evaluates each subtree recursively in random order (either left then right or right then left) and evaluates the second one only if the value is not yet determined:

\[
\text{Algorithm 1 Randomized NAND-tree evaluation}
\]

\[
\text{Procedure NANDTreeEvaluation}(T)
\]

\[
\text{if } T \text{ is a leave of value } x \text{ then}
\]

\[
\hspace{1cm} \text{return } x
\]

\[
\text{else}
\]

\[
\hspace{1cm} \text{Let } r \text{ be a uniform random bit.}
\]

\[
\hspace{1cm} \text{Let } (T_1, T_2) = \begin{cases} (T.\text{left}, T.\text{right}) & \text{if } r = 0 \\ (T.\text{right}, T.\text{left}) & \text{else} \end{cases}
\]

\[
\hspace{1cm} \text{if NANDTreeEvaluation}(T_1) = 0 \text{ then}
\]

\[
\hspace{2cm} \text{return } 1
\]

\[
\hspace{1cm} \text{else}
\]

\[
\hspace{2cm} \text{return not NANDTreeEvaluation}(T_2)
\]

\[
\text{}\hspace{-1cm} \Rightarrow \text{ Question 1.3) May this algorithm fail?}
\]

Answer. \( \Rightarrow \) Algorithm only uses that \( 0 \text{ NAND } x = 1 \) and \( 1 \text{ NAND } x = \text{ not } x \) to skip unnecessary calculations. It thus always outputs the correct answer. \( \Rightarrow \)

Given an assignment \( \sigma : \{\text{leaves of } T\} \rightarrow \{0, 1\} \), we denote by \( T(\sigma) \) the resulting value of \( T \). In order to evaluate how many leaves are inspected on expectation for the worst assignment \( \sigma \), it is convenient to introduce two functions: \( Q_0(h) \) and \( Q_1(h) \), where \( Q_v(h) \) is the worst possible expected number of leaves inspected by Algorithm on a tree of height \( h \) over all assignments \( \sigma \) such that \( T(\sigma) = v \). The expected number of leaves inspected by the algorithm for the worst assignment \( \sigma \) is then \( \max(Q_0(h), Q_1(h)) \).

\[
\text{\Rightarrow Question 1.4) Show that for all } h \geq 1, \ Q_1(h) = Q_0(h - 1) + \frac{1}{2}Q_1(h - 1) \text{ and } Q_0(h) = 2Q_1(h - 1).
\]

\[
\Rightarrow \text{ Hint. Remark that the algorithm take advantage of the fact that: if the tree } T \text{ evaluates to 1 then at least one of its subtrees evaluates to 0 which is favorable; and if } T \text{ evaluates to 0, then both of its subtrees evaluate to 1 which are both favorable.}
\]

Answer. \( \Rightarrow \) If the tree evaluates to 1, then at least one of its subtrees evaluates to 0. If this subtree is evaluated first, then the algorithm does not evaluate
the other. If both its subtrees evaluate to 0, then only one subtree will be evaluated. It follows that the worst case is when one of its subtrees evaluates to 0 and the other to 1, in which case the algorithm evaluate with probability 1/2 only the 0-subtree, and both 1- and 0-subtree also with probability 1/2. Since both subtrees do not share leaves, they can thus both be chosen to be worst cases for the algorithm and: \[ Q_1(h) = \frac{1}{2}Q_0(h-1) + \frac{1}{2}(Q_1(h-1) + Q_0(h-1)) = Q_0(h-1) + \frac{1}{2}Q_1(h-1). \]

Now, if the tree evaluates to 0, then both of its subtrees evaluates to 1 and both are evaluated by the algorithm. Again, as the subtrees do not share leaves, they can be chosen as worst cases of height \( h - 1 \) for the algorithm and thus: \[ Q_0(h) = 2Q_1(h-1). \]

Then, for all \( h \geq 2 \), \( Q_1(h) = \frac{1}{2}Q_1(h-1) + 2Q_1(h-2) \). Since \( Q_1(0) = Q_0(0) = 1, \) and \( Q_1(1) = 3/2 \), it follows that (admitted): \( Q_1(h) = \frac{33 + 5\sqrt{33}}{66} \alpha^h + \frac{33 - 5\sqrt{33}}{66} \beta^h = \Theta(\alpha^h), \) where \( \alpha = \frac{1 + \sqrt{33}}{4} \approx 1.686 \) and \( \beta = \frac{1 - \sqrt{33}}{4} \approx -1.186. \) If \( N = 2^h \) denote the number of leaves, the randomized algorithm inspects thus \( \Theta(N^{1/2} \alpha) = \Theta(N^{0.753\ldots}) \ll N \) leaves on expectation in the worst case. The randomized algorithm is then much more efficient than the deterministic algorithm in the worst case.

We will now use Yao’s principle to prove that this algorithm is indeed to optimal among the randomized algorithms that evaluate the trees recursively (which is in fact optimal for random trees — admitted). Recall that Yao’s principle states that “the expected value of a randomized algorithm on its worst instance equals the worst possible expected value of the best deterministic algorithm for the worst distribution of instances.”

In order to get a lower bound that matches our algorithm, we need to force the deterministic algorithm to look half the time at both children of a node that evaluates to 1. So, rather than choosing the values independently, we build our instance from the top down. We start by flipping a coin and setting the root to 0 or 1 with equal probability. Then we descend the tree, determining the values of the nodes at each level. If a node gets value 0, we set both its children to 1; if a node gets value 1, we set one of its children to 1 and the other to 0, flipping a fair coin to decide which is which. Then the truth values of the nodes are correlated in such a way that the easy case never happens.

**Question 1.5** Show that any deterministic algorithm that evaluates the tree recursively (i.e., which computes the value of one subtree before inspecting the leaves of the other subtree) inspects at least \( \Omega(\alpha^h) \) leaves on expectation for this distribution of instances.

**Answer:** Consider a deterministic recursive algorithm. Now suppose that we have a deterministic algorithm that evaluates the tree recursively, and suppose it is working on a node of value 1. No matter how it decides which child to evaluate first, our instances are chosen so that the first child has value 1 with probability 1/2. Therefore, it is forced to evaluate the other child with probability 1/2, just as for randomized algorithm.

Combining this with the fact that we always need to evaluate both children of a node of value 0, we see that the expected running time of the deterministic algorithm is governed by exactly the same equations as that of the randomized algorithm. The solution is again \( \alpha^j \approx (N^{0.753\ldots}) \), giving a lower bound that matches the randomized algorithm exactly. The only difference is that this probability now comes from randomness in the instances, instead of randomness in the algorithm’s behavior.

Yao’s principle allows to conclude that the randomized algorithm is optimal --- almost.

The problem is that we limited our analysis to algorithms that proceed recursively from the root down to the leaves, evaluating entire subtrees before looking anywhere else. For random trees, the optimal algorithm is indeed of this type, but we omit this (much more difficult) part of the proof. In fact, for some other types of trees (such as Majority ternary trees) we can get a better algorithm by skipping a generation, sampling several grandchildren in order to decide which children to evaluate first.