

# CSE 203A: Randomized Algorithms

Spring 2026

**Lecture No. 7:** Application of Yao's techniques & Hashing

**Date:** April 13th, 2026

**Instructor:** Daniel M. Kane

**Scribe:** Cody Lepp

## Lecture Overview

During this lecture, we corrected a lemma used in randomized routing (last lecture) while providing an intuitive proof. Following, we introduced the problem of NAND tree evaluation in the context of the theory of games and determined upper and lower bounds on query complexity while applying Yao's minimax theorem for finding the lower bound. Lastly, the motivation for hashing was established within the context of creating dictionary data structures.

## 1 Previously

We recall Yao's minimax principle in the context of zero-error (ZPP) query complexity. This principle provides a fundamental connection between randomized and deterministic algorithms in the query model.

**Theorem 1.1** (Yao's Minimax Principle). *For any query problem, the minimum expected number of queries required by a zero-error randomized algorithm on the worst-case input is equal to the maximum, over all input distributions  $D$ , of the expected number of queries required by the optimal deterministic algorithm on inputs drawn from  $D$ . Formally,*

$$\min_A \max_x \mathbb{E}[\text{queries of } \mathcal{A} \text{ on } x] = \max_D \min_A \mathbb{E}_{x \sim D}[\text{queries of deterministic } A \text{ on } x],$$

where  $\mathcal{A}$  ranges over randomized (zero-error) algorithms and  $A$  ranges over deterministic algorithms.

This theorem implies that the worst-case expected performance of the best randomized algorithm is characterized by the average-case performance of the best deterministic algorithm under the hardest input distribution.

Consequently, Yao's principle provides a powerful framework for proving lower bounds on randomized query complexity. To establish such a lower bound, it suffices to:

1. Construct a distribution  $D$  over inputs that is difficult for deterministic algorithms, and
2. Show that every deterministic algorithm must incur a large expected number of queries when inputs are drawn from  $D$ .

Thus, proving lower bounds for randomized algorithms reduces to identifying and analyzing *hard input distributions*.

## 2 Randomized routing revisited

We consider a routing problem in which each packet  $i$  is assigned a path from its source to its destination  $f(i)$ , where  $f$  is chosen randomly. Packets traverse their paths edge by edge, and at each time step, at most one packet may traverse a given edge.

**Lemma 2.1.** *For any packet  $i$ , the total time required to reach its destination is at most the length of its path plus the number of other packets whose paths share at least one edge with the path of packet  $i$ .*

*Proof.* Fix a packet  $i$  and let its path consist of edges  $e_1, e_2, \dots, e_m$ . In the absence of interference, packet  $i$  would traverse one edge per time step and arrive at time  $m$ .

To account for delays, define the *delay* of packet  $i$  after traversing edge  $e_t$  to be the quantity  $\ell$  such that it reaches the end of  $e_t$  at time  $t + \ell$ . Thus,  $\ell$  measures the cumulative delay incurred up to that point.

The key structural observation is that if two paths share edges, then they overlap on a contiguous interval of edges. Therefore, any packet that interferes with packet  $i$  must do so along such an interval.

We now argue that each unit of delay experienced by packet  $i$  can be attributed to a distinct interfering packet. Suppose that packet  $i$  incurs delay  $\ell + 1$  upon attempting to traverse some edge  $e_t$ . This can only occur if another packet occupies  $e_t$  at that time. Let  $p$  be such a packet. By construction, packet  $p$  must also be traversing the path segment containing  $e_t$ , and it must have incurred delay at least  $\ell$  at that point. In particular, we may charge this additional unit of delay of packet  $i$  to packet  $p$ .

Since each unit of delay is caused by a distinct packet that shares an edge with the path of  $i$ , the total delay of packet  $i$  is at most the number of such packets. Therefore, the total time to deliver packet  $i$  is bounded by

$$(\text{path length}) + (\text{number of packets sharing an edge with the path}),$$

as claimed. □

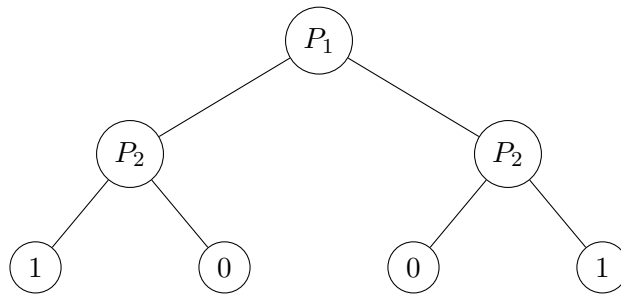
## 3 NAND Tree Evaluation

Game tree evaluation arises naturally from a two-player game in which players alternate making binary decisions over  $n$  rounds. Each root-to-leaf path corresponds to a sequence of moves, and each leaf specifies an outcome. The goal is to determine which player has a winning strategy under optimal play.

This problem can be modeled as the evaluation of a complete binary tree of height  $n$ , where each leaf is labeled with a bit in  $\{0, 1\}$  and each internal node computes the NAND of its two children:

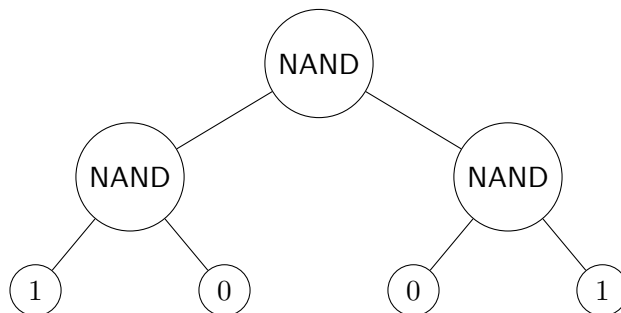
$$\text{NAND}(x, y) = \neg(x \wedge y).$$

Each node is labeled with 1 if and only if the current player has a winning strategy from that position. The value at the root determines the winner. The task is to evaluate the root by querying leaf values.

**Example: Game Tree (Depth 2)**

In this game tree:

- The root corresponds to Player 1's move.
- The second level corresponds to Player 2's responses.
- Leaves represent outcomes (e.g., 1 = win for Player 1, 0 = loss).

**Equivalent NAND Tree**

In the NAND tree:

- Internal nodes compute  $\text{NAND}(x, y) = \neg(x \wedge y)$ .
- Leaves correspond to the same outcomes as in the game tree.
- The root value determines whether the first player has a winning strategy.

**Deterministic Complexity**

**Theorem 3.1.** *Any deterministic algorithm for evaluating a NAND tree of height  $n$  requires  $2^n$  queries in the worst case.*

*Proof.* We argue via an adversary strategy. The adversary assigns leaf values adaptively in response to the algorithm's queries so as to keep the value of the root undetermined for as long as possible.

Observe that an internal node is determined only once both of its children have been evaluated. Thus, unless all leaves in the subtree of a node are queried, the adversary can always assign values to make either outcome consistent with the queries so far.

By maintaining this invariant recursively from the leaves upward, the adversary ensures that the root remains undetermined until all  $2^n$  leaves have been queried. Hence, any deterministic algorithm must query all leaves.  $\square$

## Nondeterministic Complexity

**Theorem 3.2.** *The nondeterministic query complexity of NAND tree evaluation is  $2^{n/2}$ .*

*Proof.* To certify that the root evaluates to 1, it suffices to exhibit a winning strategy for one player. This corresponds to fixing that player's choices along the tree. The opposing player's moves then determine the path, and there are  $2^{n/2}$  possible sequences of such moves (since each player moves on alternating levels).

Thus, verifying that the chosen strategy succeeds against all opponent responses requires checking  $2^{n/2}$  leaves. This yields a nondeterministic complexity of  $2^{n/2}$ .  $\square$

## Randomized Upper Bound

We now consider a randomized algorithm that evaluates the tree recursively by exploring children in a random order and short-circuiting when possible:

```
eval(node):
    if leaf:
        return query(node)
    choose a random child c
    v = eval(c)
    if v == 1:
        return eval(other child)
    else:
        return 1
```

We analyze the expected number of queries performed by this algorithm.

Let:

- $A_n$  denote the expected number of queries to evaluate a node of height  $n$  with value 0,
- $B_n$  denote the expected number of queries to evaluate a node of height  $n$  with value 1.

The base case is  $A_0 = B_0 = 1$ .

**Case 1: Node value 0.** Both children must evaluate to 1, so

$$A_n = B_{n-1} + B_{n-1} = 2B_{n-1}.$$

**Case 2: Node value 1.** At least one child evaluates to 0. We pick a child uniformly at random:

- With probability  $1/2$ , we pick the child with value 0, costing  $A_{n-1}$  queries.
- With probability  $1/2$ , we pick the child with value 1, after which we must evaluate both children.

Thus,

$$\begin{aligned} B_n &\leq \frac{1}{2}A_{n-1} + \frac{1}{2}(B_{n-1} + A_{n-1}) \\ &= A_{n-1} + \frac{1}{2}B_{n-1}. \end{aligned}$$

Combining the recurrences,

$$A_n = 2B_{n-1}, \quad B_n \leq A_{n-1} + \frac{1}{2}B_{n-1}.$$

Substituting and unrolling yields

$$T(n) \leq 3T(n-2),$$

which implies

$$T(n) \leq 3^{n/2}.$$

**Theorem 3.3.** *The randomized query complexity of NAND tree evaluation is*

$$O(3^{n/2}) = O((\sqrt{3})^n).$$

This bound lies strictly between the deterministic complexity  $2^n$  and the nondeterministic complexity  $2^{n/2}$ .

### Lower Bounds via Yao's Principle

To obtain lower bounds for randomized algorithms, we apply Yao's minimax principle. This reduces the problem to constructing a distribution over inputs for which every deterministic algorithm has large expected query complexity.

A natural candidate distribution is obtained by assigning each leaf independently to be 1 with probability  $p$ , where  $p$  satisfies the fixed-point equation

$$p = 1 - p^2.$$

This choice ensures that the distribution of values is balanced across levels of the tree.

**Lemma 3.4.** *Under the independent leaf distribution, the optimal deterministic algorithm evaluates the tree via a depth-first search with short-circuiting.*

*Proof.* Since the leaf values are independent, there is no global structure to exploit. Thus, the best strategy is to recursively evaluate subtrees and terminate early whenever the value of a node is determined. This corresponds precisely to a depth-first search with pruning.  $\square$

However, this distribution does not yield a tight lower bound. The independence assumption allows deterministic algorithms to perform relatively efficiently using local decisions. To obtain stronger lower bounds, one must consider more sophisticated, correlated input distributions, whose analysis is significantly more involved.

## 4 Introduction to Hashing

We are motivated by the design of a dictionary data structure that supports INSERT and LOOKUP operations in  $O(1)$  time. Let  $U$  denote a universe of possible keys with  $|U| = N$ , and let  $S \subseteq U$  be the set of stored elements with  $|S| = n$ , where typically  $n \ll N$ .

Our goal is to store the elements of  $S$  (possibly with associated data) in a way that allows efficient retrieval. A natural approach is to use an array-based representation. Specifically, we define a function

$$f : U \rightarrow \{1, 2, \dots, m\},$$

and maintain an array  $A$  of size  $m$ , where each element  $s \in S$  is stored at location  $A[f(s)]$ .

In the ideal case, we would like  $f$  to be injective on  $S$ , so that no two distinct elements  $x, y \in S$  satisfy  $f(x) = f(y)$ . Such events, known as *collisions*, complicate lookup operations, as multiple elements may map to the same location.

However, constructing a deterministic function  $f$  that avoids collisions for all possible subsets  $S \subseteq U$  is infeasible unless  $m \geq |U|$ , which is prohibitively large in most applications. Consequently, we relax this requirement and instead allow a small number of collisions.

The central idea of hashing is to choose the function  $f$  at random from a suitable family of functions. By doing so, we can ensure that, with high probability (over the choice of  $f$ ), the number of collisions among elements of  $S$  is small. This enables the design of data structures that achieve constant-time operations in expectation, even though collisions may occur.

## References

- [1] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.