Lecture 12: The problem with Diagonalization, and Randomized Algorithms

Anup Rao

May 2, 2024

THE ONLY WAY WE KNOW how to prove lower bounds on the running time of Turing Machines is via diagonalization. Can we hope to show that $\mathbf{P} \neq \mathbf{NP}$ by some kind of diagonalization argument? In this lecture, we discuss an issue that is an obstacle to finding such a proof.

Definition 1 (Oracle Machines)**.** *Given a function $O : \{0,1\}^* \to \{0,1\}$, an* oracle-machine *is a Turing Machine that is allowed to use a special oracle tape to make queries to O. Each query to O takes unit time.*

We can define $\mathbf{P}^O$, $\mathbf{NP}^O$ as functions computable in poly time (resp nondeterministic poly time) with oracle access to O.

Then we have the following theorem:

Theorem 2. *There exists an oracle A such that $\mathbf{P}^A = \mathbf{NP}^A$, and an oracle B such that $\mathbf{P}^B \neq \mathbf{NP}^B$.*

The theorem gives a hint about one of the ways in which it will be hard to determine whether or not $\mathbf{P} = \mathbf{NP}$. Any such proof must not work in the *relativized* worlds where access to A, B is permitted. On the other hand, the kinds of proofs that we have seen using diagonalization *do relativize*—the same argument would work even if the machines have oracle access to some oracle O.

Proof Let A be the function that on input α, x outputs 1 if and only if $M_\alpha(x)$ outputs 1 in $2^{|x|}$ steps. Then $\mathbf{P}^A = \mathbf{EXP}$, since every exponential time computation can be simulated with access to A, and every query to A can be simulated in exponential time. Also $\mathbf{NP}^A = \mathbf{EXP}$, since in exponential time we can simulate all queries to A and simulate all nondeterministic choices.

To simulate a machine M_α, that runs in time 2^{n^c}, we first create a new machine M'_α that runs M_α on the first $n^{1/c}$ bits of its input. Then we call the oracle on $M_{\alpha'}(y)$, where y is the input of length n^c with x as the first $n^{1/c}$ bits of y.

The second part is more interesting. We shall define an oracle $B : \{0,1\}^* \to \{0,1\}$ and a function $f \in \mathbf{NP}^B$ such that $f \notin \mathbf{P}^B$. f is defined in terms of B as follows:

$$f(x) = \begin{cases} 1 & \text{if there exists } y \text{ such that } |y| = |x| \text{ and } B(y) = 1, \\ 0 & \text{else.} \end{cases}$$

We first show that $f \in \mathbf{NP}^B$: a non-deterministic machine can guess y of the same length as x, and make a single query to verify that $B(y) = 1$.

To define B, we shall use diagonalization. Let $M_1, M_2, \ldots, M_i, \ldots$, be an enumeration of all machines that query B, with the feature that every machine occurs infinitely often in the sequence. (Such an enumeration exists if we allow our programming language to have redundant lines). Our goal is to make sure that the $i'th$ machine fails to compute the correct value of $f(x)$ in time $2^{n/10}$, for some n where $n = |x|$. To do this we define the value of B gradually. We define the value of B in phases. After each phase, we shall have defined the value of B on a finite set of strings.

In Phase i, let t be so large that the value of B is not yet defined on each string of length t. Then run the i'th machine $M_i(1^t)$ for $2^{t/10}$ steps. Each time M_i queries a string of B whose value has not yet been defined, return 0 and define the value of B on that string to be 0. If M_i halts with value 1, then set B to be 0 on all strings of length t. If M_i halts with value 0, then pick a string y of length t that $M_i(1^t)$ did not query (note that such a string always exists since there are 2^t binary stings of length t and M_i did not take more than $2^{t/10}$ steps), and set $B(y) = 1$.

Set the value of B on strings that are not defined by the above process to be 0.

Suppose for the sake of contradiction that $f \in \mathbf{P}^B$. Then consider the machine M that computes f. Let i be the index such that the i'th machine in the enumeration is M and t be such that $M_i(1^t)$ was used to define B on strings of length t during the i'th phase. Since the machine occurs infinitely often, there is an i for which $2^{t/10}$ exceeds the running time of the machine. Clearly, $f(1^t) \neq M(1^t)$ and hence M does not compute f. ■

Randomized Algorithm review

Probability Spaces

A *probability space* is a set Ω such that every element $a \in \Omega$ is assigned a number $0 \leq \Pr[a] \leq 1$ (called the probability of a), and $\sum_{a \in \Omega} \Pr[a] = 1$.

An *event* in this space is a subset $E \subseteq \Omega$. The probability of the event is $\sum_{a \in E} \Pr[a]$. For example, imagine we toss a fair coin n times. Then the probability space consists of the 2^n possible outcomes of the coin tosses. If E is the event that the first k coin tosses are heads, this event has probability exactly 2^{-k}. Given two events E, E', we write $\Pr[E|E']$ to denote $\Pr[E \cap E'] / \Pr[E']$. This is the probability that E happens given that E' happens. We say that E, E' are independent if $\Pr[E \cap E'] = \Pr[E] \cdot \Pr[E']$. In other words, E, E' are independent if

$\Pr[E|E'] = \Pr[E]$.

A *real valued random variable* is a function $X : \Omega \to \mathbb{R}$. The number of heads in the coin tosses is a random variable. The expected value of a random variable X is defined as $\mathbb{E}[X] = \sum_{a \in \Omega} \Pr[a] \cdot X(a)$. The following lemma is a very useful fact about random variables.

Lemma 3 (Linearity of expectation)**.** *If X, Y are real random variables, then $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$.*

Proof

$$\begin{aligned}\mathbb{E}[X+Y] &= \sum_{a\in\Omega} \Pr[a] \cdot (X(a) + Y(a)) \\ &= \sum_{a\in\Omega} \Pr[a] \cdot Y(a) + \sum_{a\in\Omega} \Pr[a] \cdot X(a) \\ &= \mathbb{E}[X] + \mathbb{E}[Y].\end{aligned}$$

■

For example, let us calculate the expected number of runs of seeing 7 contiguous heads or tails in a 200 coin tosses. Let X_i be 1 if there are 7 heads or tails that start at the i'th position, and 0 otherwise. If $1 \leq i \leq 194$, then $\mathbb{E}[X_i] = \Pr[X_i = 1] = 2 \cdot 2^{-7} = 1/64$. If $i \geq 196$, then $X_i = 0$. On the other hand, the total number of such runs is $\sum_{i=1}^{194} X_i$. So by linearity of expectation, the expected number of such runs is $194/64 \approx 3.031$.

Here is an expectation basic magic trick: Tell your audience to generate two sequences of coin tosses—one generated using 200 flips of a coin, and the second generated by hand. You leave the room, and they write both sequences on a black board. Then you come back into the room and immediately point out the sequence that was generated by hand. The trick: a random sequence is very likely to have a run of 7 heads or tails, while people tend to not insert such a long run into a sequence that they think looks random.

In class, we discussed the waiting time to see the first heads. Suppose you keep tossing a fair coin until you see heads. Let T be the number of tosses you make. What is the expected value of T? The key observation is that if the first toss is a heads, you stop with $T = 1$. Otherwise, the rest of the experiment is exactly the same as the original random experiment. So, we get:

$$\begin{aligned}&\mathbb{E}[T] = (1/2) \cdot 1 + (1/2) \cdot (1 + \mathbb{E}[T]) \\ \Rightarrow\ &\mathbb{E}[T] \cdot (1 - 1/2) = 1 \\ \Rightarrow\ &\mathbb{E}[T] = 2.\end{aligned}$$

Randomized Algorithms

We shall give a few examples of problems where randomness helps to give very effective solutions.

2-SAT

A two SAT formula is a CNF formula where each clause has exactly 2-variables. Here we give a randomized algorithm that can find a satisfying assignment to such a formula, if one exists.

Input: A two sat formula ϕ
Result: A satisfying assignment for ϕ if one exists
Set $a = 0$ to be the n-bit all 0 string;
for $i = 1, 2, \ldots, 100n^2$ **do**
  if $\phi(a) = 1$ **then**
    Output a;
  end
  Let a_i, a_j be the variables of an arbitrary unsatisfied clause.
  Pick one of them at random and flip its value ;
end
Output "Formula is not satisfiable";

Algorithm 1: Algorithm for 2 SAT

If ϕ is not satisfiable, then clearly the algorithm has a correct output. Now suppose ϕ is satisfiable and b is a satisfying assignment, so $\phi(b) = 1$. We claim that the algorithm will find b (or some other satisfying assignment) within $100n^2$ steps with high probability. We shall discuss the analysis in the next lecture.