18.405J/6.841J Advanced Complexity Theory February 4, 1997

Lecture 1

Lecturer: Daniel A. Spielman *Scribe: Henry Cohn*

1 Time Complexity Classes

Define

$$\text{TIME}(t(n)) = \{\text{languages } L : L \text{ is accepted by a Turing machine that halts within } t(n) \text{ steps}\}.$$

We won't specify the sort of Turing machine (e.g., one or two tape, or even a RAM or Kolmogorov machine). As long as we are consistent, it won't matter. Different definitions can change times by a polynomial factor, but that does not change the definitions of classes like P and NP.

Define

$$\text{P} = \bigcup_{k \geq 0} \text{TIME}(n^k).$$

Of course, the "P" stands for "polynomial time". We'll treat P as the class of things that can be computed reasonably/efficiently. Define

$$\text{EXPTIME} = \bigcup_{k \geq 0} \text{TIME}(2^{n^k}).$$

Obviously P $\subset$ EXPTIME. One of the few facts we'll prove in this course is that P $\neq$ EXPTIME. Most of the other statements we deal with will depend on unproved hypotheses. For example, we might prove that P $\neq$ NP implies that a certain problem is hard, but we generally won't be able to prove it unconditionally.

We can in fact prove the following theorem, which is much stronger than P $\neq$ EXPTIME.

Theorem 1 (Time Hierarchy Theorem) *If $f(n) = o(g(n)/\log g(n))$, then*

$$\mathit{TIME}(f(n)) \neq \mathit{TIME}(g(n)).$$

This theorem was proved in a prerequisite class. The proof is an application of diagonalization.

Define

$$\text{NTIME}(t(n)) = \{L : L \text{ is accepted by a non-deterministic Turing machine that always halts within } t(n) \text{ steps}\}.$$

Recall that a non-deterministic Turing machine has non-deterministic states, where it can branch. To see whether the machine accepts, follow all the branches and see whether any one of them accepts. Non-deterministic polynomial time is defined by

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

The fundamental questions we'd like to answer is whether P = NP. Unfortunately, we've got no idea how to answer it.

One useful way to think of languages in NP is as follows. A language L is in NP iff one can verify membership efficiently via short witnesses. More formally, that means that there is a language $L' \in \mathrm{P}$ and a polynomial $p(n)$ such that $x \in L$ iff there exists w such that $(x, w) \in L'$ and $|w| \leq p(|x|)$. Such a string w is called a witness.

2 Reductions, Completeness, and Hardness

A language A is *polynomial-time reducible* to a language B if there is a polynomial-time computable function f such that $x \in A$ iff $f(x) \in B$. We write $A \leq_P B$ in that case. The idea is that this condition implies that testing membership in A is no harder than testing membership in B. (This sort of reduction is also called mapping reduction, or Karp reduction.)

A language L is called *NP-complete* if $L \in \mathrm{NP}$ and for all $A \in \mathrm{NP}$, $A \leq_P L$. If any NP-complete problem is in P, then P = NP. Examples of NP-complete languages include circuit satisfiability (given a Boolean circuit with one output, can it be made 1?), SAT, and 3SAT.

Define

$$\mathrm{coNP} = \{L : \bar{L} \in \mathrm{NP}\}.$$

Here $\bar{L}$ is the complement of L. We do not think that NP = coNP.

The notion of reduction defined above is too limited for some situations. Turing reduction is a more general form of reduction, defined as follows. The language A is *polynomial-time Turing reducible* to B if there is a polynomial-time oracle Turing machine M that accepts A while using B as an oracle. We write $A \leq_T B$ in this case.

What this definition means is that the Turing machine can call a subroutine to ask whether strings are in B. More precisely, an *oracle Turing machine* has a special tape on which it can write queries, which are then answered in one step. (We call it an oracle because the oracle is allowed to give answers that could not have been computed efficiently, or even at all.)

A language L is defined to be *NP-hard* if $Ale_T L$ for all $A \in \mathrm{NP}$ (or, equivalently, for any one NP-complete language A). We can use a slight modification of this definition to talk about a function being NP-hard.

Notice that Turing reduction gives a perfectly good notion of hardness (in the sense that P = NP if any NP-hard language is in P), but it is not the same notion one gets from mapping reduction. For example, every coNP-complete language is NP-hard.

3 Implicit Circuit Value Problem

Since we know that EXPTIME $\neq$ P, to find a difficult problem we just need to find an EXPTIME-complete problem. A classic example is the *implicit circuit value problem*. The *circuit value problem* is to compute $C(x)$ where C is a given circuit and x an input; it can clearly be solved in polynomial time. The implicit circuit value problem is the same, except that C and x are defined only implicitly. Instead of being given C and x directly, one is given circuits D and Y that describe them as follows. Given input i, D outputs which gates in C are inputs to gate i and what type of gate it is. Given input i, Y outputs the i-th bit of x. Thus, D and Y implicitly describe an exponentially-large circuit C and its input x. Given exponential time, one can explicitly construct C and x and evaluate $C(x)$, so this problem is in EXPTIME. However, no better method comes to mind. In fact, one can imitate the proof of Cook's theorem to show that this problem is EXPTIME-complete. (Construct a huge tableau of the work tapes in an accepting computation, and make a circuit that accepts iff the computation is valid. The circuit will be immense, but it has a short implicit description.)

So now we have a problem that cannot be solved efficiently. However, it is striking that we do not even know whether P/poly contains EXPTIME. (Here, P/poly is languages recognized by polynomial-sized circuits, where we can use a different circuit for each input size. Clearly, P/poly properly contains P, but nobody knows just how large it is.)

4 The Polynomial Hierarchy

Some of the most important complexity classes we will deal with are those in the Polynomial Hierarchy. If B is a language, define

$$\mathrm{NP}^B = \{\text{languages } L \text{ such that there is a non-deterministic polynomial-time oracle Turing machine with oracle } B \text{ that accepts } L\}.$$

We write $\mathrm{NP}^{\mathrm{NP}}$ for NP^B where B is NP-complete. Clearly, $\mathrm{NP}^{\mathrm{NP}} = \mathrm{NP}^{\mathrm{coNP}}$. However, $\mathrm{coNP}^{\mathrm{NP}}$ is conjecturally different from $\mathrm{NP}^{\mathrm{NP}}$. It contains the minimum circuit problem (given a circuit, is there a smaller one computing the same function?). To see that it is in $\mathrm{coNP}^{\mathrm{NP}}$, notice that one can guess a smaller circuit and then accept iff there is always an input on which it differs from the original circuit (ask an NP oracle that question).

Let $\Sigma_1^P = \mathrm{NP}$ and $\Pi_1^P = \mathrm{coNP}$. Then continue by defining

$$\Sigma_{i+1}^P = \mathrm{NP}^{\Sigma_i^P}$$

and

$$\Pi_{i+1}^P = \text{co-}\Sigma_{i+1}^P = \mathrm{coNP}^{\Sigma_i^P}.$$

These complexity classes form the Polynomial Hierarchy (PH). It is conjectured that they are all different.