**CMPUT 675: Computational Complexity Theory** **Winter 2019**

# Lecture 5 (Jan 22): Ladner's Theorem and Oracles

*Lecturer: Zachary Friggstad* *Scribe: Jason Cannon*

In this lecture, we will wrap up our discussion of **NP**. However, the focus will be on its more high-level implications. Beginning with some basic definitions, we will motivate a proof that uses a technique called *padding*. We will then discuss the landscape of **NP** and the problems therein, showing that under the assumption that $\mathbf{P} \neq \mathbf{NP}$, we can find problems which are not decidable in polynomial-time and are not **NP**-complete. The lecture will then pivot to discuss *oracle* machines, where we will prove results that indicate a need for more than *relativising* facts if we ever hope to tackle the infamous problem of **P** versus **NP**.

## 5.1 EXP and NEXP

Previously, we have encountered the complexity class $\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c})$. In a similar fashion, we can define its non-deterministic variation:

**Definition 1** *Let* $\mathbf{NEXP} = \bigcup_{c \geq 1} \mathbf{NTIME}\left(2^{n^c}\right)$, *this is the exponential-time analog of* **NP**.

Consider that **NTIME** is defined by NDTM Turing machines. Therefore, a language is in **NEXP** if there is a verifier containing two transition functions which always run in exponential time, regardless of which transition function was followed. The input is in the language if and only if some branch of the process will lead it to accept. Let us now prove a result that draws a parallel between polynomial and exponential time classes.

**Theorem 1** *If* $\mathbf{EXP} \neq \mathbf{NEXP}$, *then* $\mathbf{P} \neq \mathbf{NP}$.

In the following proof, we will use a technique called *padding*. The idea is to "pad" every string in the language with a sequence of useless characters. In this example we will pad the string with 1's. This expands the language to "fit" in the complexity class which we are discussing. We will often use this technique when working with languages in complexity classes that need to be scaled up, for example.

**Proof.** We will prove by contrapositive, that is, assume that $\mathbf{P} = \mathbf{NP}$ and show that $\mathbf{EXP} = \mathbf{NEXP}$. Obviously, $\mathbf{EXP} \subseteq \mathbf{NEXP}$ as we can view each language in **EXP** as a NDTM using two identical transition functions. We now have to show that any language in **NEXP** is contained within **EXP**. Let $L \in \mathbf{NEXP}$ be decidable by a NDTM $M$ in time $\leq 2^{n^c}$. We then consider the language

$$L_{\text{pad}} = \left\{ \langle x, 1^{2^{|x|^c}} \rangle : x \in L \right\}$$

Intuitively, $L_{\text{pad}}$ is the language $L$ with exponentially many 1's padded to it.

**Claim 1** $L_{\text{pad}} \in \mathbf{NP}$.

**Proof.** Given $y$, we first check if there is a string $x$ such that $y = \langle x, 1^{2^{|x|^c}} \rangle$. If not, reject $y$. Otherwise, non-deterministically simulate $M$ by guessing transitions to determine if $x \in L$. Although $M$ runs in exponential time, it is only on a section of the input that looks logarithmic with respect to the size. Hence, the running time is $O(2^{|x|^c})$, which is polynomial in $y$ because the input string is exponential in length compared to $x$. ∎

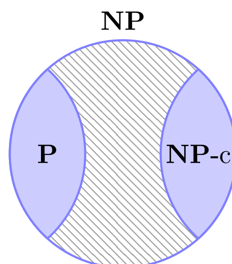
Therefore, it follows that $L_{\text{pad}} \in \mathbf{P}$ (recall that we assumed $\mathbf{P} = \mathbf{NP}$) and as such is decidable in poly($|y|$) time by some deterministic machine $\widehat{M}$. Hence, because $L_{\text{pad}} \in \mathbf{P}$ it follows that $L \in \mathbf{EXP}$: to determine if a given input $x$ is in $L$, we will pad the input as such, where $y = \langle x, 1^{\text{poly}(|y|)} \rangle$ and simulate $\widehat{M}(y)$. The running time of $\widehat{M}(y)$ is polynomial in $|y|$ where $|y|$ is exponential in the input size $|x|$. Thus, the running time is exponential in $|x|$.

## 5.2 NP-Intermediate Problems

Now let us suppose that $\mathbf{P} \neq \mathbf{NP}$. We would then have a separation into two disjoint sets: the problems that are in $\mathbf{P}$ and the problems that are $\mathbf{NP}$-complete. The question is, are there any problems that are in $\mathbf{NP}$ that do not exist in either of these sets? Under our assumption, we will show that there exists a "gray area" of sorts, where there are problems that are not in $\mathbf{P}$, nor are they $\mathbf{NP}$-complete. We will call these $\mathbf{NP}$-intermediate problems.

For example, some real problems exist that could live in this gray area. Consider the problem of *graph isomorphism*. We know this problem to clearly live in $\mathbf{NP}$ as we can provide a bijection between two graphs. However, we have been unable to find a polynomial-time algorithm or a polynomial-time reduction from an existing $\mathbf{NP}$-hard problem. Recently, the notion that this is an intermediate problem has been challenged as there is now a **quasi**-polynomial time algorithm for deciding if two graphs are isomorphic: an algorithm with running time $O(n^{\log^c n})$ for some constant $c$ [B16].

Another example that could be an $\mathbf{NP}$-intermediate problem is *integer factorization*. That is, does there exist a prime factor within some given range?




The complexity landscape if $\mathbf{P} \neq \mathbf{NP}$.

**Theorem 2 (Ladner's Theorem 1975)** *Suppose that $\mathbf{P} \neq \mathbf{NP}$, then there exists a language $L \in \mathbf{NP} - \mathbf{P}$ that is not $\mathbf{NP}$-complete.*

**Proof.** For any function $H : \mathbb{N} \to \mathbb{N}$, we define the language $\text{SAT}_H = \left\{ \langle \phi, 1^{|\phi|^{H(\phi)}} \rangle : \phi \in \text{SAT} \right\}$. That is, $\text{SAT}_H$ is SAT with some additional padding. We will now define the function $H : \mathbb{N} \to \mathbb{N}$ as follows: $H(n) =$ minimum number $i < \log_2 \log_2 n$ such that $M_i$ decides if $x \in \text{SAT}_H$ or not, in at most $1 + i \cdot |x|^i$ steps for each $|x| < \log_2 n$. If there is no such number $i$ then $H(n) = \log_2 \log_2 n$. This is a recursive definition, it relies only upon checking smaller strings of length at most $\log_2 n$.[1,2]

[1] Although this function may not be defined for small $i \in \mathbb{N}$, we could use a variation with the same asymptotic growth rate such as $\lceil \log_2(\lceil \log_2(n+2) \rceil) \rceil$. Furthermore, $H(n)$ is not well-defined for $x$ of length 0, as we would need to transition at least one step; so we could add 1 to $H(n)$ if needed.

[2] Consider $M_i$ to be the Turing machine represented by the binary expansion of $i$. We assume that every Turing machine appears infinitely often due to padding of useless characters that do not change the functionality of machine.

- $\text{SAT}_H \in \mathbf{NP}$. Since $H(n)$ is polynomial time computable (proven as a question in the homework), we can efficiently check that a given input is of the form $\langle \phi, 1^{|\phi|^{H(|\phi|)}} \rangle$. We can then check that $\phi \in \text{SAT}$ as we will non-deterministically attempt to guess a satisfying assignment.

Before showing the next two steps that prove $\text{SAT}_H \notin \mathbf{P}$ and is not $\mathbf{NP}$-complete, we have to establish some properties.

**Claim 2** $\text{SAT}_H \in \mathbf{P} \implies H(n) \in O(1)$

That is, if our language is indeed in $\mathbf{P}$ then $H(n)$ is bounded by some constant that it never exceeds.

**Proof.** As $\text{SAT}_H \in \mathbf{P}$ there exists some Turing machine $M$ that decides $\text{SAT}_H$ in $\leq c \cdot n^c$ steps. Recall that we have assumed that every Turing machine is enumerated infinitely often, so we can pick some $i > c$ such that $M_i = M$. Hence, we have that for $n > 2^{2^i}, H(n) \leq i$. Therefore, $H(n) \in O(1)$. ■

The second property we will prove is almost a strong negation of the previous one. That is,

**Claim 3** $\text{SAT}_H \notin \mathbf{P} \implies \forall i, H(n) = i$ *for only finitely many* $n$.

**Proof.** Suppose not. That is, there exists some $i$ such that $H(n) = i$ for infinitely many $n$. We claim that $M_i$ decides $\text{SAT}_H$ in $i \cdot |x|^i$ time. To see this, consider any $x \in \{0,1\}^*$ and pick $n > 2^{|x|}$ such that $H(n) = i$. By definition of $H(n)$, this means $M_i$ should correctly decide if $x \in \text{SAT}_H$ in $i \cdot |x|^i$ time. Thus, $\text{SAT}_H \in \mathbf{P}$. ■

We will now show the two required steps: that $\text{SAT}_H$ (1) is not in $\mathbf{P}$ and (2) is not $\mathbf{NP}$-complete using these two properties.

- $\text{SAT}_H \notin \mathbf{P}$. Suppose not. That is, $\text{SAT}_H \in \mathbf{P}$ and by our first property we have $H(n) \in O(1)$. Consider that $\text{SAT}_H$ is bounded by some constant, which implies that $\text{SAT}_H$ is SAT with polynomial padding of 1's. Hence, a polynomial-time algorithm to decide $\text{SAT}_H$ can then be used to solve SAT in polynomial time. Note that this argument is reminiscent to the padding argument we saw before. In this case we could simply pad SAT with polynomial 1's and run a polynomial-time algorithm that solves $\text{SAT}_H$. This implies that $\text{SAT} \in \mathbf{P}$, contradicting our assumption $\mathbf{P} \neq \mathbf{NP}$.

- $\text{SAT}_H$ is not $\mathbf{NP}$-complete. Suppose not. That is, $\text{SAT}_H$ is $\mathbf{NP}$-complete which implies there exists a polynomial-time reduction $f$ reducing SAT to $\text{SAT}_H$, $f : \phi \to \langle \psi, 1^{|\psi|^{H(\psi)}} \rangle$. The idea is that we can apply $f$ and remove the useless padding from the instance of $\text{SAT}_H$. Additionally, we have just shown that $\text{SAT}_H \notin \mathbf{P}$. By our second property, we know that $\forall i, H(n) = i$ for only finitely many $i$. Hence, it follows that $|\psi| < |\phi|$ for large enough $|\phi|$. This implies that we could take an instance of SAT and use a polynomial-time reduction to reduce to a smaller instance of SAT. As such, we could recursively apply this reduction until we have achieved some desired constant size and solve any instance in constant time. Thus, $\text{SAT} \in \mathbf{P}$ which contradicts our assumption $\mathbf{P} \neq \mathbf{NP}$ and the fact that SAT is $\mathbf{NP}$-complete.

Therefore, it follows that $\text{SAT}_H \in \mathbf{NP} - \mathbf{P}$ and is not $\mathbf{NP}$-complete. Under the assumption that $\mathbf{P} \neq \mathbf{NP}$, we have constructed our first $\mathbf{NP}$-intermediate problem. Ultimately, we have separated $\mathbf{P}$ and $\mathbf{NP}$-complete problems by creating a class of problems that exist in the "gray area" between them. ■

Observe a key argument in this proof: if there is a polynomial-time reduction from SAT to itself that maps instances of size $n$ to instances of size $o(n)$, then $\mathbf{P} = \mathbf{NP}$.

## 5.3 Oracle Machines

An *oracle* Turing machine is a Turing machine $M$ with access to a special read-write tape called an *oracle tape*. Additionally, we also associate to the machine a language $O \subseteq \{0,1\}^*$. The Turing machine $M$ is similar to a deterministic Turing machine, but with 3 additional states $q_{\text{query}}, q_{\text{yes}}$, and $q_{\text{no}}$. If at any point during the execution of a program, $M$ enters the state $q_{\text{query}}$, then the machine transitions to the state $q_{\text{yes}}$ if the contents of the oracle tape belong to the language $O$. Otherwise, the machine transitions to the state $q_{\text{no}}$. Furthermore, it is important to note that regardless of the language $O$, the query counts only as a single computational step. We can define *non-deterministic* oracle Turing machines in a similar manner.

We will now define some new complexity classes relating to oracles:

**Definition 2** *Analogous to* $\mathbf{P}$ *and* $\mathbf{NP}$*, we have that for every* $O \subseteq \{0,1\}^*$, $\mathbf{P}^O$ *is the set containing all languages decidable by a polynomial-time deterministic oracle Turing Machine with oracle access to* $O$*. It follows that* $\mathbf{NP}^O$ *is the set containing every language that can be decided by a polynomial-time nondeterministic Turing machine with oracle access to* $O$*.*

### 5.3.1 Relativising Proofs

Intriguingly, almost all of the proofs we have covered so far still hold when applied to polynomial oracle Turing machines! For example, the diagonalization proofs used to prove results about space and time complexity still work verbatim, as does the padding argument covered in the beginning of the notes. We can also still simulate oracle machines as long as we are working with one fixed oracle language. As such, we call proofs that can hold in the presence of oracles *relativising* proofs. Similarly, if the proof does not hold in the presence of oracles, we say that it is a *nonrelativising* proof. For example, one of the only nonrelativising proofs that we have covered thus far is the *Cook-Levin* theorem. This is because given an oracle machine, the concept of "encoding" (using clauses and variables to capture the computation) is not well-defined.

### 5.3.2 Limitations of Techniques

In this section, we will demonstrate to some extent the limits of the techniques we have learned. We will show that the basic concepts of padding, diagonalization, and so on by themselves are not enough to separate $\mathbf{P}$ versus $\mathbf{NP}$, unless they take advantage of some nonrelativising fact. In order to separate $\mathbf{P}$ and $\mathbf{NP}$, we need more tools than we have had access to so far. *Cook-Levin*'s theorem could potentially provide some insight, as we could use it to focus on a single problem such as SAT and show that SAT $\notin \mathbf{P}$. However, we simply cannot consider these two complexity classes at a high level and hope to give rise to any kind of contradiction. A consequence of this theorem is that we cannot simply solve $\mathbf{P}$ versus $\mathbf{NP}$ using a relativising proof.

**Theorem 3 (Baker, Gill, Solovay 1975)** *There exist oracles* $A, B \subseteq \{0,1\}^*$ *such that* $\mathbf{P}^A = \mathbf{NP}^A$ *and* $\mathbf{P}^B \neq \mathbf{NP}^B$.

First, we will let $A$ be the language EXPCOM $= \{\langle \alpha, x, 1^t \rangle \mid \text{the DTM } M_\alpha \text{ accepts } x \text{ in at most } 2^t \text{ steps}\}$. This in an exponential-time complete language. It is essentially the analog of TMSAT, seen previously, except for its deterministic exponential time. EXPCOM is the language of all triples $\langle \alpha, x, 1^t \rangle$, where $\alpha$ is encodes a DTM, $x$ is some proposed input to that machine, and $t$ is a sequence of 1's encoding some unary value $t$. We will now prove the following claim:

**Claim 4** $\mathbf{P}^A = \mathbf{NP}^A = \mathbf{EXP}$ *for some oracle* $A$*.*

That is, $\mathbf{P}^A$ and $\mathbf{NP}^A$ are the same. In fact, the class of languages that can be decided by Turing machines of either category is exactly those that can be deterministically decided without oracles by an exponential time algorithm.

**Proof.** Clearly, $\mathbf{P}^A \subseteq \mathbf{NP}^A$ is obvious, for the same reason that $\mathbf{P}$ is contained in $\mathbf{NP}$. Every deterministic oracle verifier machine looks like a non-deterministic oracle machine with two identical transition functions.

**Claim 5** $\mathbf{NP}^A \subseteq \mathbf{EXP}$

Let $L \in \mathbf{NP}^A$ be decided by NDTM $M$ with oracle access to $A$ and running time $O(n^c)$. As $L$ is still in $\mathbf{NP}$, it must still make a polynomial-time number of transitions. We want to show that any language that can be decided by a polynomial-time non-deterministic verifier with oracle access to $A$ (an exponential-time complete language) can be decided in deterministic exponential-time. Note, when $M$ queries $A$, it is done in $O(n^c)$ time. Recall from the definition of $A$ that $t$ is polynomial-time bounded. Thus, each oracle query can be deterministically simulated in $2^{O(n^c)}$ time and the number of non-deterministic branches is also at most $\leq 2^{O(n^c)}$. Hence, we simply enumerate all branches and simulate the oracle in $2^{O(n^c)}$ time. Consider the following carefully: we are multiplying these two running times. For each of the exponentially-many branches, we are doing exponential-time computation. When multiplying them, we will just "hide" things in the $O$. Therefore, $L \in \mathbf{EXP}$.

**Claim 6** $\mathbf{EXP} \subseteq \mathbf{P}^A$

Finally, we will show that $\mathbf{EXP}$ languages are contained in $\mathbf{P}^A$. Fortunately, the proof is simpler. Consider that we know $A$ to be exponential-time complete. So, any given input $x$ decided by some Turing machine $M_\alpha$ can be decided by a polynomial-time oracle machine. Given $x$, add $\alpha$ to it as polynomial size padding to construct an instance that can be solved by the oracle machine.

Let $L \in \mathbf{EXP}$ be decidable by $M_\alpha$ in time $2^{n^c}$. To decide if $x \in L$ with a polynomial-time oracle machine, we must:

- write $\langle \alpha, x, 1^{n^c} \rangle$ (which takes poly $|x|$ time)
- query $A$, output answer.

As we have $\mathbf{P}^A \subseteq \mathbf{NP}^A \subseteq \mathbf{EXP} \subseteq \mathbf{P}^A$ it follows that $\mathbf{P}^A = \mathbf{NP}^A = \mathbf{EXP}$, as required. ■

### 5.3.3 Construction of $B$

Let us now demonstrate an interesting language. Using this language and an oracle, we will show that in fact we can separate $\mathbf{NP}$ from nearly exponential-time, not just $\mathbf{P}$ from $\mathbf{NP}$. Let us construct an oracle $B$ that will separate these two iteratively. First, for an arbitrary language $B \subseteq \{0,1\}^*$ let us consider the unary encoding $U_B = \{1^n : \exists x \in B \text{ with } |x| = n\}$ (which is more like a unary indicator). So this language contains strings of only 1's with relation to the length of $x$ in the original language $B$. Hence, for any $B$, it follows that $U_B \in \mathbf{NP}^B$. The verifier can check that the input consists of only 1's and if so, attempts to non-deterministically guess the form of $x$ and query the oracle. This holds even if we use a really weird language that is undecidable.

**Claim 7** $\mathbf{P}^B \neq \mathbf{NP}^B$ *for some oracle* $B$.

**Proof.** To prove this claim, we will carefully and iteratively construct a language $B$ over a series of steps. Our construction will ensure that a newly defined unary language $U_B$ is not in $\mathbf{P}^B$. We will build $B$ in stages, referring to each as **Stage i**, where **Stage 0** is the initial stage.

In essence, we will simulate an oracle Turing machine that will query the language we are partially building. If it queries something we have already answered, then we will consistently give the same answer. Otherwise, we will define an answer. Here is the formal construction:

(a) *Constructing $B$ in stages.* Initially, we construct the following two sets:
$B_Y^0, B_N^0 := \varnothing$
(with the invariant being that after each step the $B^i$'s constructed are finite)

(b) *Building $B_Y^0, B_N^0$, inductively.* Use the following algorithm:
- Let $n_i = 1 + \max\{|x| : \left(x \in B_Y^{<i} \cup B_N^{<i}\right)\}$
- Run $M_i$ on $1^{n_i}$ for $\leq 2^{\frac{n_i}{10}}$ steps
  - If $M_i$ queries with some $x \in B_Y^{\leq i} \cup B_N^{\leq i}$ (union of all previous sets up to $i$ itself), answer the query consistently.
  - If $M_i$ queries with $x \notin B_Y^{\leq i} \cup B_N^{\leq i}$ , answer "No" and add $x$ to $B_N^i$.
- If $M_i$ halts and accepts $1^{n_i}$ in $\leq 2^{\frac{n_i}{10}}$ steps, add all $\{0,1\}^n$ to $B_N^i$.
- Otherwise, pick some $x \notin \{0,1\}^{n_i} - (B_Y^i \cup B_N^i)$ and add $x$ to $B_Y^i$ (such $x$ exists as we halted in less than $2^{n_i}$, so there is some string that we did not query).

Let $B = \bigcup_{j\geq 0} B_Y^j$ and recall the definition of the unary language $U_B = \{1^n : \exists x \in B, |x| = n\}$. Our goal is to show that $U_B \notin \mathbf{P}^B$.

That is, show that no polynomial-time oracle machine with access to this language $B$ could have decided $U_B$. Also note that the proof is going to use a sort of "diagonal-type" argument once again. Therefore, our claim is:

**Claim 8** $U_B \in \mathbf{NP}^B - \mathbf{P}^B$

**Proof.** We will prove the above claim as follows: We have already seen previously that for any language $B$ it holds that $U_B \in \mathbf{NP}^B$. Again, given an input it will verify if it consists only of 1's. If so, it will non-deterministically guess $x$ and ask the oracle if $x$ is in the language. As such, we must prove:

**Claim 9** $U_B \notin \mathbf{P}^B$

Suppose not. That is, $U_B \in \mathbf{P}^B$. Let $M$ be a polynomial-time Turing machine with oracle $B$ that decides $U_B$ in $p(n)$ time, where $p(n)$ is a polynomial. We have assumed we can enumerate our Turing machines such that each machine appears infinitely often. Consider that the $n_i$'s are increasing throughout the enumeration (because they are always larger than previously queried, and at each step, something was added). Let's pick some $i$ such that $2^{\frac{n_i}{10}} > p(n_i)$. We know such $i$ exists because the $n_i$'s are increasing exponentially and will eventually surpass the polynomial. Consider that as $M_i$ decides the language, regardless of the input, $M_i$ will halt before $2^{n_i}$ steps because it runs in time $p(n)$. The question is whether it accepts or not. We have:

- $M_i(1^{n_i})$ accepts if and only if $\exists x \in \{0,1\}^{n_i} \cap B$, by definition of $U_B$.
- $M_i(1^{n_i})$ accepts it and only if $\forall x \in \{0,1\}^{n_i}, x \notin B$, by construction of $B_n^i$.

Intuitively, by definition of the language $M_i$ accepts if some element of the required length exists in the set. However, if it accepts, by construction the set would have been empty.

Hence, this is a contraction and it follows that $U_B \notin \mathbf{P}^B$. ∎

Therefore, we have shown that there exist oracles $A, B$ such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$. ∎

## References

AB09 S. Arora and B. Barak, Computational Complexity: A Modern Approach, *Cambridge University Press, New York, NY, USA*, 2009.

B16 L. Babai, Graph Isomorphism in Quasipolynomial Time, in proceedings of ACM SIGACT Symposium on Theory of Computing, 684–697, 2016.