CS 6810: Theory of Computing **Fall 2021**

Lecture 10: September 28, 2021

Lecturer: Eshan Chattopadhyay *Scribe: Jesse Goodman*

In which we introduce the polynomial hierarchy and alternating Turing machines.

1 The Polynomial Hierarchy

Background and motivation In the previous lecture, we introduced the complexity class $\mathsf{P/poly}$, which contains all languages computable by polynomial sized circuits. We saw that $\mathsf{P} \subseteq \mathsf{P/poly}$, and thus demonstrating a language $L \in \mathsf{NP}$ that does not have polynomial sized circuits would suffice to prove $\mathsf{P} \neq \mathsf{NP}$.

This approach towards proving $\mathsf{P} \neq \mathsf{NP}$ is tempting, as circuits are simple combinatorial objects that seem much easier to analyze than Turing machines. However, we saw in the last lecture that $\mathsf{P/poly}$ contains undecidable languages. Thus, one may be concerned that there may not be *any* languages in $\mathsf{NP} - \mathsf{P/poly}$, as this would make this approach towards proving $\mathsf{P} \neq \mathsf{NP}$ futile.

In this lecture, we introduce the *polynomial hierarchy*, a fundamental complexity class that will help assuage the above concerns. In particular, we will see (in the next lecture) that the polynomial hierarchy can be used to provide strong evidence that $\mathsf{NP} \not\subseteq \mathsf{P/poly}$, meaning that there is good reason to believe $\mathsf{P} \neq \mathsf{NP}$ can be proven using circuit lower bounds.

Beyond the above motivation, the polynomial hierarchy is a natural complexity class built from subclasses that very nicely generalize P, NP, and coNP. As a result, these subclasses are able to capture several natural problems which appear "just beyond the reach" of P, NP and coNP.[1] We now proceed to formally introduce the polynomial hierarchy.

The building blocks In order to build the polynomial hierarchy, the key idea is to provide two different ways to equip a complexity class with more computational power. More formally, recall that a complexity class $\mathcal{C}$ is simply a collection of languages $L \subseteq \{0,1\}^*$. The first way we will enhance $\mathcal{C}$ will be to equip it with an existential quantifier ($\exists$).

Definition 1.1 (Complexity class $\exists\mathcal{C}$)**.** *For any complexity class $\mathcal{C}$, the complexity class $\exists\mathcal{C}$ is defined as follows. For any language $L \subseteq \{0,1\}^*$, it holds that $L \in \exists\mathcal{C}$ if and only if there is a language $L' \in \mathcal{C}$ and polynomial q such that for all $x \in \{0,1\}^*$,*

$$x \in L \iff \exists y \in \{0,1\}^{q(|x|)} \text{ such that } (x,y) \in L'.$$

To help digest this definition, we make a few observations. First, note that the existential quantifier can only add power: more formally, we have $\mathcal{C} \subseteq \exists\mathcal{C}$, since one can always take $q = 0$. On the other hand, it is not too difficult to show that an additional existential quantifier *cannot* add even more power: that is, $\exists\exists\mathcal{C} = \exists\mathcal{C}$. Finally, to see why Definition 1.1 is natural, we observe that it provides the following clean characterization of NP.

Observation 1.2. $\exists\mathsf{P} = \mathsf{NP}$.

[1]We emphasize, however, that it not known whether these problems actually fall outside P, NP and coNP.

Proof. $L \in \exists\mathsf{P}$ iff there is a language $L' \in \mathsf{P}$ and a polynomial q such that $x \in L \iff \exists y \in \{0,1\}^{q(|x|)}$ such that $(x,y) \in L'$. Expanding out the definition of P, we get that $L \in \exists\mathsf{P}$ iff there is a polynomial q and poly-time Turing machine M (computing a language L') where $x \in L \iff \exists y \in \{0,1\}^{q(|x|)}$ such that $M(x,y) = 1$. This is exactly the verifier-based definition of NP. □

Indeed, as we believe $\mathsf{P} \subsetneq \mathsf{NP}$, this suggests that the existential quantifier may sometimes add *strictly* more power to a complexity class. This completes our discussion of the complexity class $\exists\mathcal{C}$. We now consider a second way to equip $\mathcal{C}$ with more computational power, using a for-all quantifier ($\forall$).

Definition 1.3 (Complexity class $\forall\mathcal{C}$)**.** *For any complexity class $\mathcal{C}$, the complexity class $\forall\mathcal{C}$ is defined as follows. For any language $L \subseteq \{0,1\}^*$, it holds that $L \in \forall\mathcal{C}$ if and only if there is a language $L' \in \mathcal{C}$ and polynomial q such that for all $x \in \{0,1\}^*$,*

$$x \in L \iff \forall y \in \{0,1\}^{q(|x|)} \text{ it holds that } (x,y) \in L'.$$

Observe that, as in the existential case, the for-all quantifier can only add power: indeed, we have $\mathcal{C} \subseteq \forall\mathcal{C}$, since we can always take $q = 0$. On the other hand, we again have that an additional quantifier *cannot* add even more power: that is, $\forall\forall\mathcal{C} = \forall\mathcal{C}$. Finally, we emphasize that Definition 1.3 is natural via the following clean characterization of coNP.

Observation 1.4. $\forall\mathsf{P} = \mathsf{coNP}$.

The proof of this observation is identical to the proof of Observation 1.2, using instead the verifier-based definition of coNP. As we believe $\mathsf{P} \subsetneq \mathsf{coNP}$, this suggests that the for-all quantifier may sometimes add *strictly* more power to a complexity class. This completes our discussion of the complexity class $\forall\mathcal{C}$.

Alternating quantifiers Above, we have seen that $\exists\mathsf{P}$ and $\forall\mathsf{P}$ exactly characterize NP and coNP, respectively. Given these observations, it is natural to ask about the complexity classes that are generated by prepending quantifiers to NP and coNP. Since we have seen that $\exists\mathcal{C} = \exists\exists\mathcal{C}$, we know via Observation 1.2 that $\exists\mathsf{NP} = \mathsf{NP}$. Similarly, we know via Observation 1.4 that $\forall\mathsf{coNP} = \mathsf{coNP}$. Indeed, repeating quantifiers is not so interesting, and it is more meaningful to ask about the classes $\forall\mathsf{NP} = \forall\exists\mathsf{P}$ and $\exists\mathsf{coNP} = \exists\forall\mathsf{P}$, which *alternate quanitifers*. We of course have $\forall\mathsf{NP} \supseteq \mathsf{NP}$ and $\exists\mathsf{coNP} \supseteq \mathsf{coNP}$, but are these containments strict?

This turns out to be a surprisingly deep question, for which we do not have an answer. However, the following language provides evidence that the answer may be *yes*:

$$\begin{aligned}\mathsf{minDNF} := \{\phi \in \{0,1\}^* : \phi &\text{ is a DNF formula such that}\\ &\text{no smaller DNF formula } \psi \text{ computes the same function as } \phi.\}\end{aligned}$$

It seems that minDNF lies "just beyond the reach" of NP: for any given formulas ϕ, ψ, it is easy to check $\phi \not\equiv \psi$ in NP by checking if there is an assignment z such that $\phi(z) \neq \psi(z)$. But making sure this holds *for all* smaller formulas ψ seems to require an additional (alternating) quantifier. Next, we show that such an alternating quantifier suffices.

Claim 1.5. $\mathsf{minDNF} \in \forall\mathsf{NP}$.

Proof. Since $\forall\mathsf{NP} = \forall\exists\mathsf{P}$, it suffices to find a language $L' \in \mathsf{P}$ and polynomial q such that

$$\phi \in \mathsf{minDNF} \iff \forall\psi \in \{0,1\}^{q(|\phi|)}, \exists z \in \{0,1\}^{q(|\phi|)} \text{ such that } (\phi,\psi,z) \in L'. \tag{1}$$

Consider the polynomial $q(m) = m - 1$ and the language L' of triples (ϕ, ψ, z) such that the following holds: ϕ, ψ are DNF formulas over the same set of variables, z is an assignment to those variables, and both $|\psi| < |\phi|$ and $\phi(z) \neq \psi(z)$ hold. Given an appropriate encoding of DNFs, it is straightforward to verify that Equation (1) holds. Furthermore, it is easy to evaluate DNFs in polynomial time. Thus $L' \in \mathsf{P}$, completing the proof. $\square$

The polynomial hierarchy Above, we demonstrated a language $L \in \forall\mathsf{NP}$ that is not known to be in NP. This suggests that the containment $\forall\exists\mathsf{P} \supseteq \exists\mathsf{P}$ may be strict, just like how the containment $\exists\mathsf{P} \supseteq \mathsf{P}$ may be strict - indeed, notice that the latter is exactly the question of whether $\mathsf{P} = \mathsf{NP}$. More generally, it is natural to ask how the power of P grows as we prepend more and more alternating quanitifers. Such motivates the definition of the *polynomial hierarchy*, which we are finally ready to formally define. (Below, we take the natural numbers $\mathbb{N}$ to include 0.)

Definition 1.6 (Polynomial hierarchy)**.** *First, we define the complexity classes $\Sigma_0 = \Pi_0 := \mathsf{P}$. Then, for every $i \in \mathbb{N}$, we define the classes $\Sigma_{i+1} := \exists\Pi_i$ and $\Pi_{i+1} = \forall\Sigma_i$. The* polynomial hierarchy PH *is then defined as follows.*

$$\mathsf{PH} := \bigcup_{i \in \mathbb{N}} \Sigma_i = \bigcup_{i \in \mathbb{N}} \Pi_i.$$

The following diagram (inspired by the Lecture 11 notes) gives a visual representation of the polynomial hierarchy, which may be easier to remember.

$$\begin{tikzcd}
 & & & & \vdots & & & & \\
\exists\Pi_2 & = & \Sigma_3 & & & & \Pi_3 & = & \forall\Sigma_2 \\
\exists\Pi_1 & = & \Sigma_2 \arrow[u] \arrow[urrrr] & & & & \Pi_2 \arrow[u] \arrow[ullll] & = & \forall\Sigma_1 \\
\exists\Pi_0 & = & \Sigma_1 \arrow[u] \arrow[urrrr] & & & & \Pi_1 \arrow[u] \arrow[ullll] & = & \forall\Sigma_0 \\
 & & \Sigma_0 & = & \mathsf{P} \arrow[ull] \arrow[urr] & = & \Pi_0 & &
\end{tikzcd}$$

In the diagram above, arrows indicate containment: these simply follow from our earlier observations that prepending an existential or for-all quantifier to a complexity class $\mathcal{C}$ can only make it more powerful. Furthermore, inspired by the above diagram, for every $i \in \mathbb{N}$ we call the complexity class $\Sigma_i \cup \Pi_i$ the i^{th} *level of the polynomial hierarchy.*

As can be seen, the only subclasses of the polynomial hierarchy for which a containment relationship is unknown are subclasses that appear in the same level. Indeed, resolving these containment relationships remain some of the most important open problems in theoretical computer science (see the next lecture for more details). Still, there is at least one relationship we *can* establish between classes that appear at the same level of the hierarchy.

Claim 1.7. *For all $i \in \mathbb{N}$, it holds that $\Sigma_i = \mathsf{co}\Pi_i$.*

Proof. We briefly remind the reader that for a complexity class $\mathcal{C}$, the class $\mathsf{co}\mathcal{C}$ contains a language L iff its complement $\overline{L}$ is in $\mathcal{C}$. We proceed by induction.

The base case $i = 0$ is straightforward: $\Sigma_0 = \Pi_0 = \mathsf{P}$, so we just need $L \in \mathsf{P} \iff \overline{L} \in \mathsf{P}$. This is clearly true, just by swapping the accept/reject states of the Turing machine computing L or $\overline{L}$.

For $i > 0$, we first prove $L \in \Sigma_i \implies L \in \mathsf{co}\Pi_i$. Towards this end, note $L \in \Sigma_i \implies L \in \exists\Pi_{i-1}$, and thus there is some language $L' \in \Pi_{i-1}$ and polynomial q such that

$$x \in L \iff \exists y \in \{0,1\}^{q(|x|)} \text{ such that } (x,y) \in L'.$$

Negating both sides of the implication, and using the definition of set complement, we get

$$x \in \overline{L} \iff \forall y \in \{0,1\}^{q(|x|)} \text{ it holds that } (x,y) \in \overline{L'}.$$

Since $\overline{L'} \in \mathsf{co}\Pi_{i-1}$, we get that $\overline{L} \in \forall\mathsf{co}\Pi_{i-1}$. By the induction hypothesis, $\overline{L} \in \forall\Sigma_{i-1}$, and we know $\forall\Sigma_{i-1} = \Pi_i$. Thus $\overline{L} \in \Pi_i$, or rather $L \in \mathsf{co}\Pi_i$, as desired.

We can prove $L \in \mathsf{co}\Pi_i \implies L \in \Sigma_i$ in a similar manner. Pick any $L \in \mathsf{co}\Pi_i$ and note $L \in \mathsf{co}\Pi_i \implies \overline{L} \in \Pi_i \implies \overline{L} \in \forall\Sigma_{i-1} \implies \overline{L} \in \forall\mathsf{co}\Pi_{i-1}$, where the last line follows by the induction hypothesis. Thus there is some $L' \in \mathsf{co}\Pi_{i-1}$ and polynomial q such that

$$x \in \overline{L} \iff \forall y \in \{0,1\}^{q(|x|)} \text{ it holds that } (x,y) \in L'.$$

Negating both sides of the implication, and using the definition of set complement, we get

$$x \in L \iff \exists y \in \{0,1\}^{q(|x|)} \text{ such that } (x,y) \in \overline{L'}.$$

Since $\overline{L'} \in \Pi_{i-1}$, we get that $L \in \exists\Pi_{i-1}$ and thus $L \in \Sigma_i$, as desired. □

Finally, whenever a new complexity class is introduced, asking about the *hardest* problems in that class can help us understand how it interacts with existing classes from the complexity landscape. In other words, now that we have introduced the polynomial hierarchy and the rich subclasses from which it is built, we would like to know if we can find problems that are complete for these classes. As it turns out, there are a very natural family of problems that do exactly this. We start by introducing them, below.

Definition 1.8 ($\Sigma_i\mathsf{SAT}$)**.** *For any fixed $i \in \mathbb{N}$, the language $\Sigma_i\mathsf{SAT} \subseteq \{0,1\}^*$ consists of all quantified Boolean formulas Φ of the form*

$$\exists y_1 \forall y_2 \ldots Q_i y_i \phi(y_1, y_2, \ldots, y_i),$$

where each y_i is a vector of Boolean variables, $Q_i = \exists$ if i is odd and $Q_i = \forall$ otherwise, ϕ is a Boolean formula, and the entire sentence Φ is true.

Definition 1.9 ($\Pi_i\mathsf{SAT}$)**.** *For any fixed $i \in \mathbb{N}$, the language $\Pi_i\mathsf{SAT} \subseteq \{0,1\}^*$ consists of all quantified Boolean formulas Φ of the form*

$$\forall y_1 \exists y_2 \ldots Q_i y_i \phi(y_1, y_2, \ldots, y_i),$$

where each y_i is a vector of Boolean variables, $Q_i = \forall$ if i is odd and $Q_i = \exists$ otherwise, ϕ is a Boolean formula, and the entire sentence Φ is true.

To conclude this section, we show that the above definitions provide complete problems for every subclass in the polynomial hierarchy.

Theorem 1.10. *For any $i \in \mathbb{N}$, the language $\Sigma_i\mathsf{SAT}$ is Σ_i-complete under polynomial time Karp reductions, and the language $\Pi_i\mathsf{SAT}$ is Π_i-complete under polynomial time Karp reductions.*

Proof. We only prove the result for $\Sigma_i\mathsf{SAT}$, as the proof for $\Pi_i\mathsf{SAT}$ is nearly identical. Towards this end, we need to prove $\Sigma_i\mathsf{SAT} \in \Sigma_i$, and that every $L \in \Sigma_i$ is poly-time Karp reducible to $\Sigma_i\mathsf{SAT}$.

To prove $\Sigma_i\mathsf{SAT} \in \Sigma_i$, we simply need a polynomial time TM M and polynomial q such that

$$\Phi \in \Sigma_i\mathsf{SAT} \iff \exists y'_1 \in \{0,1\}^{q(|\Phi|)} \forall y'_2 \in \{0,1\}^{q(|\Phi|)} \dots Q_i y'_i \in \{0,1\}^{q(|\Phi|)} M(\Phi, y'_1, y'_2, \dots, y'_i) = 1,$$

where $Q_i = \exists$ if i odd and $Q_i = \forall$ otherwise. Setting q to be the identity function, it is easy to construct such an M: simply have it evaluate ϕ (contained in Φ), plugging in each y'_i for y_i (and ignoring the $|y'_i| - |y_i| \geq 0$ extra bits at the end of y'_i).

To prove each $L \in \Sigma_i$ is poly-time Karp reducible to $\Sigma_i\mathsf{SAT}$, fix some $L \in \Sigma_i$ and recall there must be some poly-time TM M and polynomial q such that

$$x \in L \iff \exists y'_1 \in \{0,1\}^{q(|x|)} \forall y'_2 \in \{0,1\}^{q(|x|)} \dots Q_i y'_i \in \{0,1\}^{q(|x|)} M(x, y'_1, y'_2, \dots, y'_i) = 1,$$

where $Q_i = \exists$ if i is odd and $Q_i = \forall$ otherwise. By applying the Cook-Levin Theorem to M or its negation, there is a poly-time transformation f with input x that outputs a formula ϕ_x such that $M(x, y'_1, y'_2, \dots, y'_i) = 1$ if and only if $Q''_i y'' \phi_x(y'_1, y'_2, \dots, y'_i, y'')$ is true, where Q''_i matches the quantifier Q_i. Thus

$$x \in L \iff \exists y'_1 \forall y'_2 \dots Q_i y'_i Q''_i y'' \phi_x(x, y'_1, y'_2, \dots, y'_i, y'').$$

Merging Q_i and Q''_i turns the right-hand-side into an instance of $\Sigma_i\mathsf{SAT}$. Finally, we can bootstrap f to create a poly-time computable function f' that outputs the entire right-hand-side (instead of just ϕ_x), so that $x \in L \iff f'(x) \in \Sigma_i\mathsf{SAT}$, as desired. $\square$

2 Alternating Turing Machines

We now introduce a new type of Turing machine, which generalizes our previous definitions of Turing machines (see Lectures 1 and 2) and captures problems appearing in the polynomial hierarchy.

Definition 2.1 (Alternating Turing Machine (ATM))**.** *An* alternating Turing machine *(ATM) M is just like a standard Turing machine (as defined in Lecture 1), except for three key differences:*

- *Like an NDTM, an ATM has* two *transition functions δ_0, δ_1, instead of just one.*
- *Like an NDTM, an ATM has an additional special state q_{accept}, in addition to the standard special states $q_{\mathsf{start}}, q_{\mathsf{halt}}$.*
- *Unlike an NDTM, each state in the ATM (except for q_{halt} and q_{accept}) is labeled with the existential quantifier ($\exists$) or the for-all quantifier ($\forall$).*

Next, we must specify how an ATM computes, and provide a meaningful definition of runtime.

Definition 2.2 (Computation and runtime of ATMs)**.** *Given an input x, the* computation *of an ATM M starts at q_{start} and proceeds until reaching q_{halt} or q_{accept}, choosing at each step whether to transition according to δ_0 or δ_1 (just like an NDTM). The* runtime *of an ATM M on input x is at most T if M halts after at most T steps, no matter what sequence of transition functions were selected. The runtime of an ATM M is $T(n)$ if the runtime of M on any input x of length n is at most $T(n)$.*

Finally, we must define what it means for an ATM to *accept* an input string. While an NDTM is defined to accepted a string whenever *there exists* a sequence of transition function choices that lead to q_{accept}, the analogous definition for ATMs is slightly more nuanced.

Definition 2.3 (Accepting condition for ATMs)**.** *The ATM M* accepts *x if the following holds. Let $G_{M,x}$ be the configuration graph of M on input x.*[2] *For each node C_{accept} in $G_{M,x}$ representing a configuration in state q_{accept}, mark that node* accept. *Mark each node C_{reject} in state q_{halt} as* reject. *Then, for each unmarked node C with two marked children:*

- *If C is in a state labeled $\exists$, mark it as* accept *if at least one of its children is marked* accept*, and mark it* reject *otherwise.*
- *If C is in a state labeled $\forall$, mark it as* accept *if both of its children are marked* accept*, and mark it* reject *otherwise.*

Continue marking nodes in this manner until the node C_{start} in state q_{start} is marked. We say that M accepts x if and only if C_{start} is marked accept.

Now that we have a complete definition of alternating Turing machines, we are ready to discuss their relationship to the polynomial hierarchy. For this, we need to lift the standard definitions for time complexity to the alternating setting, which we do in the natural way.

Definition 2.4. *For any function $T : \mathbb{N} \to \mathbb{N}$ and language $L \subseteq \{0,1\}^*$, we say $L \in \mathsf{ATIME}(T(n))$ if there is a constant $C > 0$ and ATM M running in time $C \cdot T(n)$ that decides L.*

As expected, we can then define an alternating version of the class P.

Definition 2.5. *The complexity class AP of alternating polynomial time is defined as*

$$\mathsf{AP} := \bigcup_{C \in \mathbb{N}} \mathsf{ATIME}(n^C).$$

Now, to understand how alternating Turing machines are related to the polynomial hierarchy, we consider various restrictions to the former model. First, consider requiring that every state in the ATM be labeled with $\exists$ (see Definition 2.1). It is straightforward to show that such restricted ATMs are exactly NDTMs. Thus, if we define $\mathsf{AP}_{\exists}$ to be the class of languages decidable by such restricted ATMs running in polynomial time, we clearly get the following observation.

Observation 2.6. $\mathsf{AP}_{\exists} = \mathsf{NP}$.

Similarly, we can imagine requiring that every state in the ATM be labeled with $\forall$. In this case, we can define $\mathsf{AP}_{\forall}$ to be the class of languages decidable by these ATMs in polynomial time, and obtain the following.

Observation 2.7. $\mathsf{AP}_{\forall} = \mathsf{coNP}$.

More generally, we can establish a relationship between ATMs and every level of the polynomial hierarchy by restricting the number of *quantifier alternations* on any computation path of the ATM. Formally, we can say that an ATM M is a $(\exists, i)$-ATM if q_{start} is labeled $\exists$, and for any input x the following holds: let $G_{M,x}$ be the configuration graph of M on x. Then for every path from the starting configuration to a halting or accepting configuration, there are at most $i-1$ alternations (i.e., the quantifier label of the state changes at most $i-1$ times). If we let $\mathsf{AP}_{\exists,i}$ denote the class of languages decidable by $(\exists, i)$-ATMs in polynomial time, the following observation is relatively straightforward. (Below, we take the natural numbers to begin at 1.)

[2] $G_{M,x}$ is a directed graph where all non-leaf nodes have out-degree 2. Each node C corresponds to a configuration of the machine, and there is a directed edge (C, C') if the machine can transition from configuration C to configuration C' using either δ_0 or δ_1. See Lecture 6 notes for more detail.

Observation 2.8 (Observation 2.6, general version)**.** $\mathsf{AP}_{\exists,i} = \Sigma_i$*, for every* $i \in \mathbb{N}$*.*

Similarly, we can define $(\forall, i)$-ATMs and $\mathsf{AP}_{\forall,i}$ in the natural way, and observe the following.

Observation 2.9 (Observation 2.7, general version)**.** $\mathsf{AP}_{\forall,i} = \Pi_i$*, for every* $i \in \mathbb{N}$*.*

Thus, alternating Turing machines are closely connected to the polynomial hierarchy, in the sense that we can exactly characterize each subclass with appropriately restricted ATMs running in polynomial time.[3] To conclude this lecture, we show that alternating Turing machines characterize a completely different (deterministic) complexity class, in a result that is perhaps much more surprising.

Theorem 2.10 (Alternating time equals deterministic space)**.**

$$\mathsf{AP} = \mathsf{PSPACE}.$$

Before proving this theorem, we remark that $\mathsf{PH} \subseteq \mathsf{AP}$ since we saw above that each $\Sigma_i, \Pi_i \subseteq \mathsf{AP}$. However, it is not known whether $\mathsf{PH} \supseteq \mathsf{AP}$, the reason being that each language in PH is allowed to leverage a constant number of alternations, whereas languages in AP are allowed to make use of an *unbounded* number of alternations (i.e., the number of alternations can grow with input length). Indeed, we believe that $\mathsf{PH} \not\supseteq \mathsf{AP}$, as this would imply $\mathsf{PH} = \mathsf{PSPACE}$, which would lead to a collapse of the polynomial hierarchy (see Lecture 11 notes for more details). We now proceed to prove Theorem 2.10, concluding the lecture.

Proof of Theorem 2.10. We start by sketching $\mathsf{AP} \supseteq \mathsf{PSPACE}$. Towards this end, recall the language TQBF, which consists of all *true quantified Boolean formulas.* Such formula are of the form $\Phi = Q_1 x_1 Q_2 x_2 \dots Q_n x_n \phi(x_1, x_2, \dots, x_n)$, where each $Q_i \in \{\exists, \forall\}$, each x_i is a Boolean variable, ϕ is a Boolean formula, and $\Phi \in \mathsf{TQBF}$ if and only if Φ is true. Recall that TQBF is PSPACE-complete under polynomial-time Karp reductions, and thus it suffices to show that $\mathsf{TQBF} \in \mathsf{AP}$. This is almost immediate from definitions: to construct an ATM M that decides TQBF, simply label q_{start} with the quantifier Q_1, and then at every step i, have the transition function δ_0 fix x_i to 0, have δ_1 fix x_i to 1, and have both transition functions lead to a state labeled Q_{i+1} (where we define $Q_{n+1} = \exists$ for notational convenience). Then, at step $n+1$, use the fixings of $x_1, x_2, \dots, x_n$ to deterministically evaluate $\phi(x_1, x_2, \dots, x_n)$ in polynomial time.

We now sketch $\mathsf{AP} \subseteq \mathsf{PSPACE}$. This is similar to the proof we've seen for $\mathsf{TQBF} \in \mathsf{PSPACE}$ (from Lecture 7 notes). The key idea is to reuse space. More formally, let M be an ATM running in time n^C for some constant $C > 0$. We wish to construct a deterministic TM M' running in polynomial space, which simulates M. Towards this end, note that for any $\alpha \in \{0,1\}^*$ of length at most n^C, we can construct the deterministic Turing machines $\mathsf{ACCEPT}_\alpha, \mathsf{REJECT}_\alpha, \mathsf{EXISTS}_\alpha$, defined as follows. On input x, each of these TMs simulates M' on $|\alpha|$ steps, picking transition function δ_{α_i} at step i. ACCEPT_α outputs 1 iff an accepting state (q_{accept}) is reached, REJECT_α outputs 1 iff a rejecting state (q_{reject}) is reached, EXISTS_α outputs 1 iff a state labeled $\exists$ is reached.

Finally, we can define a deterministic Turing machine HELP, which takes input x, α, and is defined as follows. First, it accepts if $\mathsf{ACCEPT}_\alpha(x)$ outputs 1, and rejects if $\mathsf{REJECT}_\alpha(x)$ outputs 1. Otherwise, it checks $\mathsf{EXISTS}_\alpha(x)$. If the answer is 1, then it computes and returns $\mathsf{HELP}(x, \alpha \circ 0) \vee \mathsf{HELP}(x, \alpha \circ 1)$. Otherwise, it computes and returns $\mathsf{HELP}(x, \alpha \circ 0) \wedge \mathsf{HELP}(x, \alpha \circ 1)$.

Finally, we let ε denote the empty string, and define $M'(x) = \mathsf{HELP}(x, \varepsilon)$.

[3]Technically, we never described how to characterize P with ATMs. This can easily be done by forcing $\delta_0 = \delta_1$.

It is straightforward to show that M' decides the same language as M, and that M' can be implemented on a deterministic TM. To see why M' can be implemented in polynomial space, note that the two recursive calls in HELP can reuse the same space, simply by waiting for one of them to return before starting the other. All that needs to be remembered is the current state of α in the recursion stack, and the corresponding sequence of quantifiers (or rather, sequence of $\vee, \wedge$). Since $|\alpha| \leq n^C$, and since every other non-recursive function call clearly uses polynomial time and thus polynomial space, we get that M' runs in polynomial space, as desired. $\square$