CSE696 Lecture 2, Wed. Feb. 3: The Arithmetical and Polynomial Hierarchies.

(Un-)Computability Relative to an Oracle

Let $\mathscr{C}$ be a collection of oracle Turing machines M. Then for any language A, define

$$\mathscr{C}^A = \left\{L\left(M^A\right) : M \in \mathscr{C}\right\}.$$

For example, taking the collection of all oracle Turing machines,

$$\mathrm{RE}^A = \left\{L\left(M^A\right)\right\} = \text{ the set of all languages that can be accepted with oracle } A.$$

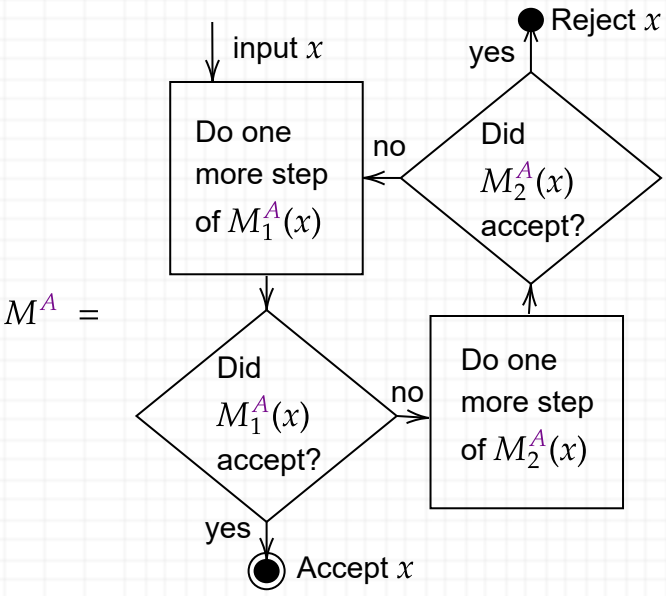
And $\mathrm{REC}^A = \left\{L\left(M^A\right) : M \textit{ is total with oracle } A\right\}$. Now here is a fun puzzle. Saying that M is total---i.e., halts for all inputs---with oracle A is a statement that depends on the particular oracle A. It does not come from M belonging to a collections of machines by themselves. The natural collection to use is that of OTMs that are total with *all* oracles---indeed, that have an associated *time clock* $t(n)$ that shuts them off after $t(|x|)$ steps independent of any answers from the oracle. Call this collection $\mathscr{T}$.

For any A, clearly $\mathscr{T}^A \subseteq \mathrm{REC}^A$ since every machine in $\mathscr{T}$ is of course total with oracle A. But are they equal? That is to say, if you have M and A such that M halts for all inputs when A is the oracle, can we replace M by M' such that $L\left(M'^A\right) = L\left(M^A\right)$ and M' is total with *all* oracles---indeed, has a computable running time bound apart from the oracle? To model this, identify all languages A with branches of the infinite binary tree $\mathcal{B}$. Now see if you can frame the problem in a way that leverages König's Lemma: every subtree of $\mathcal{B}$ that has no infinite branch is finite, and in particular, has finite depth.

The possibly-larger definition of REC^A suffices, in any event, for the following two theorems that were proved without an oracle in CSE596.

Theorem 1: For all oracles A, $\mathrm{RE}^A \cap co-\mathrm{RE}^A = \mathrm{REC}^A$.
Theorem 2: For all oracles A, $\mathrm{RE}^A \neq \mathrm{REC}^A$.

To prove Theorem 1, suppose $L \in \mathrm{RE}^A \cap co-\mathrm{RE}^A$. Then there are OTMs M_1, M_2 such that $L\left(M_1^A\right) = L$ and $L\left(M_2^A\right) = \sim L$. Build an OTM M to carry out the following routine---for any oracle A, not just the given one:



And to prove Theorem 2, define $D_A = \{\langle M \rangle : \langle M \rangle \notin L(M^A)\}$.

If D_A were in RE^A, then there would be an OTM Q such that $L(Q^A) = D_A$. But:

Q^A accepts $\langle Q \rangle \iff \langle Q \rangle \in D_A$
by definition of $L(Q^A) = D_A$,
$\iff \langle Q \rangle \notin D_A$ by definition of D_A.

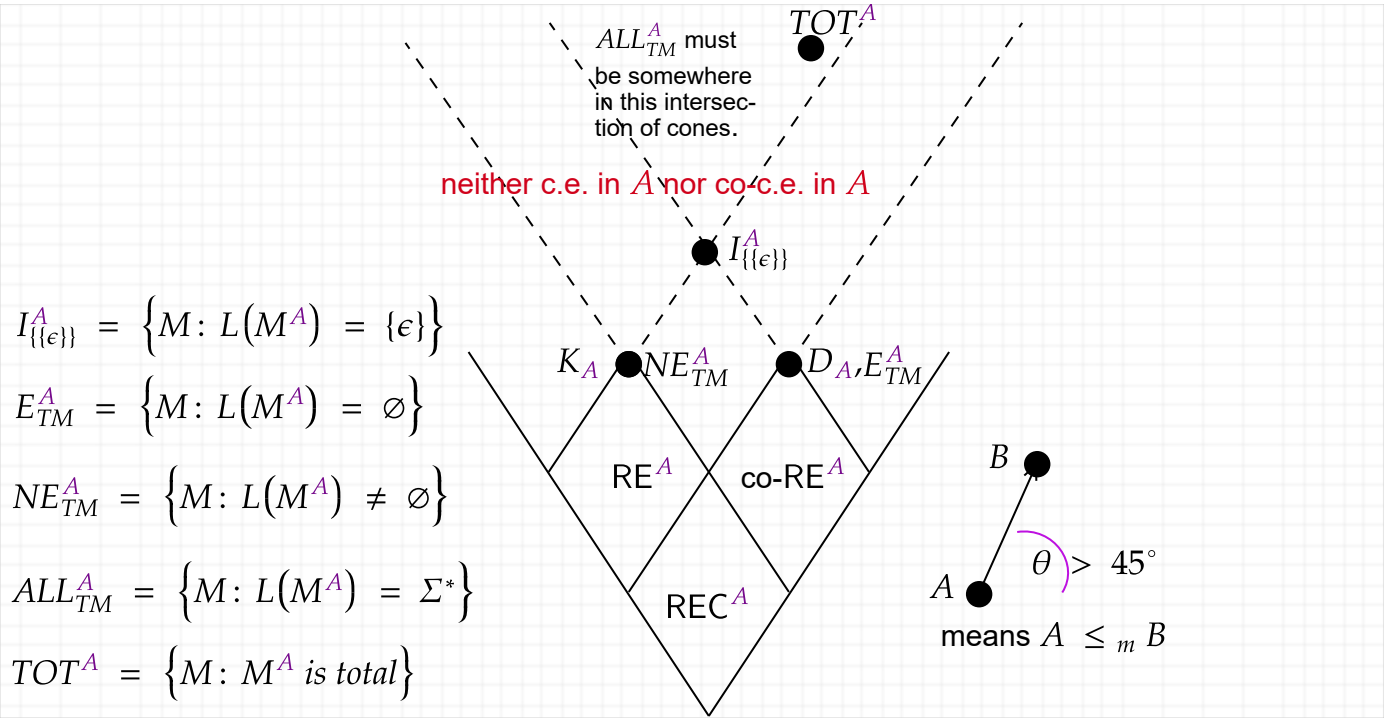
This contradiction shows $D_A \notin \text{RE}^A$, yet its complement (call it K_A) is $\{\langle M \rangle : \langle M \rangle \in L(M^A)\}$, which does belong to RE^A --- but not to REC^A. ☒

The point---which Turing realized in his original 1936 paper---is that these proofs are really the same as the original ones without the oracle. The pink As are not really *used* in the proof. They just "ride along." Another fact is that we can turn a universal Turing machine into a universal oracle Turing machine U such that for any oracle A, U^A on input $\langle M, w \rangle$ simulates $M^A(w)$. Furthermore:

Theorem 3: For all oracles A, $L(U^A)$ is complete for RE^A under reductions that are computable in linear time *without* using the oracle.

Proof: Given any language $L \in \text{RE}^A$, take an OTM M such that $L(M^A) = L$. To reduce L to $L(U^A)$, map $f(w) = \langle M, w \rangle$. Note that the reduction is "just syntax"---no oracle involved. ☒

So RE^A always has complete languages just like RE does without the oracle. Moreover, K_A is complete for RE^A via the same trick used to reduce the universal language A_{TM} to K without the oracle. We get exactly the same picture as before with extra "pink As" added:



Define $\mathscr{C}^{\mathsf{D}} = \{L(M^A) : M \in \mathscr{C},\ A \in \mathsf{D}\}$. For example, $\mathsf{RE}^{\mathsf{RE}} = \bigcup_{A \in \mathsf{RE}} \mathsf{RE}^A$, which by the completeness of K for RE (the ordinary non-oracle versions) equals RE^K. But this notation conceals a trap. In $\mathsf{RE}^{\mathsf{RE}}$ and especially in RE^A, the bottom RE is not a class of languages like the top RE is. It's a "class" of *machines*, and maybe should really be written $\mathscr{RE}^{\mathsf{RE}}$.

Likewise, when we write P^A and NP^A, the bottoms are not really the language classes P and NP. They are the collections of deterministic and nondeterministic oracle TMs, respectively, with polynomial time clocks bolted on. This is a major example of "abuse of notation" that is rampant in complexity theory and occasionally deceives. Pardon my French: *tout abus sera muni*.

The real significance of the *non-use* of the pink As is that the associated methods cannot resolve cases where they *do* matter. When A is the language TQBF of true quantified Boolean formulas, then $\mathsf{P}^A = \mathsf{NP}^A$. Hence methods that are ignorant of internal details of the oracle set can never prove $\mathsf{P} \neq \mathsf{NP}$ without the oracle. There are languages B such that $\mathsf{P}^B \neq \mathsf{NP}^B$, so $\mathsf{P} = \mathsf{NP}$ can never be proved without really being concrete about the *absence* of an oracle. The ability to "relativize" to oracles is thus the first barrier to resolving the $\mathsf{P} = ?\ \mathsf{NP}$ question.

Defining the Hierarchies

Nevertheless, the notation is useful to define both reductions and hierarchies.

Definition 2: For any languages A and B,

- A **Turing-reduces** to B, written $A \leq_T B$, if $A \in \mathsf{REC}^B$.
- A **polynomial-time Turing-reduces** to B, written $A \leq^p_T B$, if $A \in \mathsf{P}^B$.

Definition 3: $\sum_0^0 = \prod_0^0 = \text{REC}$, $\sum_1^0 = \text{RE}$, $\prod_1^0 = \text{co-RE}$, and for $k \geq 2$: $\sum_k^0 = \text{RE}^{\Sigma_{k-1}^0}$ and $\prod_k^0 = \text{co-}\sum_k^0$. Also $\text{AH} = \bigcup_k \sum_k^0$.

Definition 4: $\sum_0^p = \prod_0^p = \text{P}$, $\sum_1^p = \text{NP}$, $\prod_1^p = \text{co-NP}$, and for $k \geq 2$: $\sum_k^p = \text{NP}^{\Sigma_{k-1}^p}$ and $\prod_k^p = \text{co-}\sum_k^p$. Also $\text{PH} = \bigcup_k \sum_k^p$.

The term "arithmetical hierarchy" can refer either to the suite of these classes as a concept or to their union, which is the class AH. Likewise for "polynomial hierarchy" and PH. The superscript p stands for "polynomial". The superscript 0 (which is often omitted) stands for "first-order arithmetic" and/or the old-style notation $\mathbf{0}$ for REC as a "degree of unsolvability." Why are we talking about "arithmetic"? That's where much of the beauty and intellectual heft of the arithmetical hierarchy comes from. We will build on the next theorem proved in CSE596:

Theorem 4: A language L belongs to RE if and only if there is a decidable predicate $R(x, y)$ such that for all x, $x \in L \iff (\exists y)R(x, y)$.

Note also:

Theorem 4': A language L belongs to NP if and only if there are a polynomial-time decidable predicate $R(x, y)$ and a polynomial p such that for all x, $x \in L \iff (\exists y : |y| \leq p(|x|))R(x, y)$.

The only predicate that we need to consider in both cases is the **Kleene T-predicate** $T(M, x, \vec{c})$ $\equiv$ "$\vec{c}$ is an accepting computation of the (possibly nondeterministic) Turing machine M on input x." This is decidable in polynomial time---in fact, decidable in linear time by a machine that acts like a deterministic finite automaton with two heads. In the direction going forward from L, the machine M is fixed so we really get a predicate $T_M(x, y)$ where y encodes the computation $\vec{c}$. Furthermore, we could compact the statement in the NP case by stipulating that $R(x, y)$ be decidable in time polynomial in $|x|$ alone, so that a polynomial p forcing $|y| \leq p(|x|)$ would come from that. We could call the $R(x, y)$ predicate *polynomial p-decidable* in that case. But IMHO it is important to keep in mind that the major difference between the NP and RE cases is not the time to decide $R(x, y)$ but rather the length bound on y. For shorthand we can abbreviate $(\exists y : |y| \leq p(|x|))$ to $(\exists^p y)$ when the context for $|y| \leq p(|x|)$ is clear. Then we can also say:

- A language L belongs to co-RE if and only if there is a linear-time decidable predicate $R(x, y)$ such that for all x, $x \in L \iff (\forall y)R(x, y)$.
- A language L belongs to co-NP if and only if there is a linear-time decidable predicate $R(x, y)$ and a polynomial p such that for all x, $x \in L \iff (\forall^p y)R(x, y)$.

Arithmetic comes into play because the T-predicate can be encoded entirely numerically with M, x treated as numbers and $\vec{c}$ as the "tupling" of a list of numbers into one number. If the computation is the sequence of configurations $I_0, I_1, \ldots, I_t$ and these are already encoded as numbers, then we can consider $\vec{c} = 2^{I_0}3^{I_1} \cdots p_t^{I_t}$ using the first t odd primes. (As we sometimes say on the blog, 2 is a very odd prime.) There are more-compact ways to do pairing and tupling via polynomials $q(u, v)$ without exponentiating, but what's significant is that this does require multiplication. The legwork for this was already done before Turing's 1936 paper by Kurt Gödel in 1930-31 in the context of encoding *proofs* rather than *computations*, but the essence is much the same. It was Stephen Kleene who systematized the abstract formalization of computation. He wrote separately $U(\vec{c}) = v$ to specify the value output by the computation, but we can use any convenient variant notation such as $T(M, x, \vec{c}, v)$ or $T(M, x, t)$ just to say the computation halts within t steps. For the polynomial hierarchy we will still use predicates, but there we will find a backbone that uses no arithmetic at all, just propositional logic with $\wedge, \vee, \neg$ (or alternatively just NAND or NOR), growing from SAT to the language TQBF.

Definition 3. Call a predicate of the form $S(x) = (\exists y)R(x, y)$ with R (linear-time-) decidable a $\sum_1$*-predicate*, and $S'(x) = (\forall y)R(x, y)$ a $\prod_1$*-predicate*. And naturally enough, $\left(\exists^p y\right)R(x, y)$ is a $\sum_1^p$*-predicate* and $\left(\forall^p y\right)R(x, y)$ is a $\prod_1^p$*-predicate*.

Inductively, for $k \geq 2$, a $\sum_k$*-predicate* is one of the form $S(x) = (\exists y)R(x, y)$ where R is a $\prod_{k-1}$-predicate, and a $\prod_k$*-predicate* has the form $S'(x) = (\forall y)R'(x, y)$ where R' is a $\sum_{k-1}$-predicate. $\sum_k^p$*-predicates* and $\prod_k^p$*-predicates* are defined analogously.

Every $\sum_k$-predicate is the negation of a $\prod_k$-predicate and vice-versa. Any predicate $S(x)$ with the one free string/numeric variable x defines a language L_S via $L_S = \{x : S(x) \text{ is true}\}$. Note that when a particular string is substituted for x, the unary predicate becomes a logical **sentence**. Unrolling the definition, we see:

- A $\sum_2$-predicate has the form $S(x) = (\exists y)(\forall z)R(x, y, z)$ with R decidable (in linear time).
- A $\prod_2$-predicate has the form $S'(x) = (\forall y)(\exists z)R'(x, y, z)$ with R' decidable.
- A $\sum_3$-predicate has the form $S(x) = (\exists y)(\forall z)(\exists w)R(x, y, z, w)$ with R decidable.
- A $\prod_5^p$-predicate has the form $S'(x) = \left(\forall^p y\right)\left(\exists^p z\right)\left(\forall^p w\right)\left(\exists^p v\right)\left(\forall^p u\right)R$ with p a polynomial and $R(x, y, z, w, v, u)$ being polynomial-time (wlog. linear-time) decidable.

The number k reflects not the raw count of quantifiers but the number of times they **alternate** between $\exists$ and $\forall$. Using pairing and tupling, we can always "condense" quantifiers of the same kind, e.g. $(\exists t)(\exists u)(\exists v)R(\cdots)$ becomes $(\exists w)[w =: \langle t, u, v\rangle \wedge R(\cdots)]$.

Examples:

- M *never halts* $\equiv (\forall x)(\forall t)\neg T(M,x,t)$, so the language of (non-oracle) TMs that do not halt on any input [i.e., are such that for all inputs, the machine does not halt on that input] **is** co-c.e.
- M *always halts* $\equiv (\forall x)(\exists t)T(M,x,t)$, so the ALL_{TM} language belongs to $\prod_2^0$.
- $L(M)$ *belongs to* P: We can define a recursive enumeration $P_1, P_2, P_3, \ldots$ of polynomial-time bounded [oracle] Turing machines. Then $L(M) \in \mathrm{P} \equiv (\exists k)[L(M) = L(P_k)]$. This is a $\sum_3$-predicate because the equality of the languages of two Turing machines (even when both them are not necessarily halting) is definable by a $\prod_2$-predicate.

Now we can state:

Theorem (Kleene's Arithmetical Hierarchy Theorem): For all $k \geq 1$:

- A language L belongs to $\sum_k^0$ if and only if it equals L_S for some $\sum_k$-predicate S.
- Similarly, L belongs to $\prod_k^0$ iff it is defined by a $\prod_k$-predicate S'.
- The language V_k of **true** $\sum_k$-sentences of arithmetic is complete for $\sum_k^0$ under $\leq_m$ (indeed, under many-one reductions that are linear-time computable).

One immediate corollary of this and the separation $\mathrm{RE}^A \neq \mathrm{REC}^A$ in Theorem 2 involves a time inversion:

Corollary: The language V of all sentences of arithmetic that are true (in the standard model $\mathbb{N}$, that is) is hard for $\sum_k^0$ for all k, and hence does not belong to AH at all.

This was proved by the logician Alfred Tarski in 1933---after Gödel but before Turing. Compare:

- Using simple arithmetic, we can define real numbers that we cannot compute (Turing, 1936).
- The set of theorems of (Giuseppe Peano's formalization of) arithmetic is undecidable (Gödel, 1931). The set of theorems is, however, definable by a $\sum_1$-predicate.
- The set of true statements of (Peano's) arithmetic is not definable in arithmetic at all (Tarski).

In the polynomial world, we have an analogous statement:

Theorem ("Weak" Polynomial Hierarchy Theorem): For all $k \geq 1$:

- A language L belongs to $\sum_k^p$ if and only if it equals L_S for some $\sum_k^p$-predicate S.
- Similarly, L belongs to $\prod_k^p$ iff it is defined by a $\prod_k^p$-predicate S'.

- The language B_k of **true** Σ_k-sentences of propositional logic is complete for $\sum_k^p$ under $\leq_m^p$ (where the polynomial depends on the language being reduced to B_k).

Note that a Boolean *formula* $\phi(x_1, \ldots, x_n)$ is satisfiable if and only if the corresponding propositional *sentence* $(\exists x_1)(\exists x_2) \cdots (\exists x_n)\phi(x_1, \ldots, x_n)$ is true, i.e., belongs to B_1. Thus the PSPACE-complete language TQBF, which essentially equals $\cup_k B_k$, is analogous to V. This analogy promotes powerful *belief* that the polynomial hierarchy theorem is infinite and (hence) different from PSPACE, but we don't have such a "Strong Hierarchy Theorem" yet for PH.

Our notational setup enables us to do the $k = 2$ case while losing no generality.

Proof: The ($\Longleftarrow$) direction, from a Σ_2-predicate $S(x) = (\exists y)R(x, y)$ to an OTM M with the co-c.e. language L_R as oracle, is easy: we code M on input x to loop $y = 0, 1, 2, \ldots$ and accept if and when the oracle says yes to the query $\langle x, y \rangle$.

To do the ($\Longrightarrow$) direction, let M be an OTM with a co-c.e. oracle R, which we can (by induction) identify with a $\prod_1$-predicate $R(y)$. What makes the induction a little non-trivial is that we also use the $\sum_1$-representation of the negation R' of R. That is, we have $R(y) = (\forall w)Q(y, w)$ for some decidable predicate Q, so $R'(y) = (\exists w)Q'(y, w)$. Now we can define for any x:

$x \in L\left(M^R\right) \iff (\exists \vec{d})\Phi(\vec{d})$, where the string $\vec{d}$ unpacks not only into an accepting computation $\vec{c}$ that postulates answers to each oracle query, but also gives:

1. The queries $y_1, \ldots, y_k$ that are listed as being answered "yes" in $\vec{c}$, with the body of Φ including $R(y_1) \wedge \cdots \wedge R(y_k)$. This is where the second $(\forall \cdots)$ quantifier comes in.
2. The queries $z_1, \ldots, z_\ell$ listed in $\vec{c}$ as being answered "no", **together with** the witnesses $w_1, \ldots, w_\ell$ such that $Q'(z_1, w_1) \wedge \cdots \wedge Q'(z_\ell, w_\ell)$.

To complete putting this into strict Σ_2 form, we can use the identity

$$(\forall v_1)Q(y_1, v_1) \wedge (\forall v_2)Q(y_2, v_2) \wedge \cdots \wedge (\forall v_k)Q(y_k, v_k) \equiv (\forall v)[Q(y_1, v) \wedge \cdots \wedge Q(y_k, v)].$$

Even if the middle connector were $\vee$ not $\wedge$, we would still get a single universal **block** as $(\forall v_1, \ldots v_k)[Q(y_1, v_1) \vee \cdots \vee Q(y_k, v_k)]$. To do the induction for $k > 2$, we iterate similar aspects of the general algortihm for conversion to **prenex normal form**. That process also finishes the formal conversion into Σ_k-sentences for the third part. To finish this "handwave", the B_k case piggybacks onto the proof of the Cook-Levin Theorem for $k = 1$. $\boxtimes$