CS 154 NOTES

ARUN DEBRAY
MARCH 13, 2014

These notes were taken in Stanford's CS 154 class in Winter 2014, taught by Ryan Williams. I live-TEXed them using `vim`, and as such there may be typos; please send questions, comments, complaints, and corrections to a.debray@math.utexas.edu. Thanks to Rebecca Wang for catching a few errors.

CONTENTS

Part 1. Finite Automata: Very Simple Models

1. Deterministic Finite Automata: 1/7/14

"When the going gets tough, the tough make a new definition."

This class is about formal models of computation, which makes it a blend of philosophy, mathematics, and engineering. What is computation? What can and cannot be computed? What can be *efficiently* computed (whatever that means in context)?

Though it's a theory class, there are excellent reasons to care: it leads to new perspectives on computing, and theory often drives practice. For example, the theory of quantum computers is relatively well-known, even though the best quantum computer today can only factor numbers as large as 21. Finally, math is good for you! Just like vegetables. And, like the vegetables in my fridge, the course content is timeless. Advances in technology come and go, but theorems are forever.

The course falls into three parts: finite automata, which are simple and relatively well-understood models; computability theory, which is much more powerful but not quite as well understood; and complexity theory, which provides more applied theory for understanding how long it takes to solve some problems, but isn't as well-understood.

This is a proofs class. One great way to do a proof is to hit each of the three levels:

- A short phrase (two to four words) that gives a hint of what trick will be used to prove the claim.
- The second level: a short, one-paragraph description of the ideas.
- The last level: the complete proof.

This is how Sipser wrote his book, and we are encouraged to write our solutions. For example:

Claim. Suppose $A \subseteq \{1,\dots,2n\}$ and $|A| = n+1$. Then, there are two numbers in A such that one divides the other.

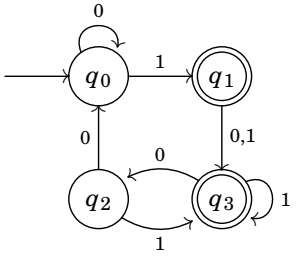
Proof. The pigeonhole principle states that if one has m pigeons and puts them in n holes. and $m > n$, then there is at least one hole with more than one pigeon in it.[1] Another useful observation is that every $a \in \mathbb{Z}$ can be written as $a = 2^k m$, where m is odd and $k \in \mathbb{Z}$.

On the second level: using the pigeonhole principle, we can show there is an odd m and $a_1 \neq a_2$ in A such that $a_1 = 2^{k_1} m$ and $a_2 = 2^{k_2} m$, so one must divide the other.

More formally: write each element in A of the form $a = 2^k m$, where $m \in \{1,\dots,2n\}$ is odd. Since $|A| = n+1$, then there must be two distinct numbers in A with the same odd part, since there are only n such odd numbers. Thus, $a_1, a_2 \in A$ have the same odd part, so $a_1 = 2^{k_1} m$ and $a_2 = 2^{k_2} m$, and therefore one divides the other. ☒

Note that in this class, the homeworks tend to require the most complete level of proof, but the class won't go into the goriest details. In any case, the high-level concepts are more important, though the details are not unimportant.

Moving into content, a deterministic finite automaton is a set of states with transitions between them. For example:



One can then accept or reject strings. Formally:

Definition 1.1. A language over a set (an alphabet) Σ is a set of strings over Σ, i.e. a subset of Σ^* (the set of all strings over Σ).

It's useful to think of a language as a function from $\Sigma^* \to \{0,1\}$, where the set of the language is the strings that output a 1.

Definition 1.2. A deterministic finite automaton (DFA) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states, e.g. $Q = \{q_0, q_1, q_2, q_3\}$ in the above example;
- Σ is the alphabet, which is also a finite set;
- $\delta : Q \times \Sigma \to Q$ is the transition function;
- $q_0 \in Q$ is the start state; and
- $F \subseteq Q$ is the set of accept states (final states).

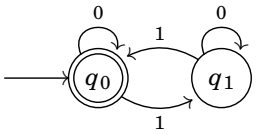
Then, let $w_1,\dots,w_n \in \Sigma$ and $\vec{w} = w_1 \dots w_n \in \Sigma^*$. Then, M accepts $\vec{w}$ if there exist $r_0,\dots,r_n \in Q$ such that:

- $r_0 = q_0$,
- $\delta(r_i, w_{i+1}) = r_{i+1}$, and
- $r_n \in F$.

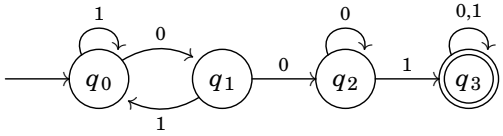
Finally, the language accepted by M is $L(M) = \{s \in \Sigma^* \mid M \text{ accepts } s\}$.

[1] I personally prefer the formulation in which one drills n holes in m pigeons, so that at least one pigeon has more than one hole in it...

For example the automaton



accepts a string from $\{0,1\}^*$ iff it has an even number of 1s, and the following DFA accepts exactly the strings containing a sequence 001:



Here, at q_1, the automaton has seen a 0, at q_2 it's seen 00, and at q_3, it's seen a 001 (so it wins).

Definition 1.3. A language A is regular if there exists a DFA M such that $L(M) = A$, i.e. there is some DFA that recognizes it.

Theorem 1.4 (Union Theorem for Regular Languages)**.** *If L_1 and L_2 are regular languages over the same alphabet Σ, then $L_1 \cup L_2$ is also a regular language.*

Proof. The proof idea is to run both M_1 and M_2 at the same time. Specifically, given a DFA $M_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$ that recognizes L_1 and a DFA M_2 for L_2 with analogous notation, construct a DFA as follows:

- $Q = Q_1 \times Q_2$, i.e. pairs of states from M_1 and M_2.
- Σ is the same.
- $q_0 = (q_0^1, q_0^2)$.
- $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ or } q_2 \in F_2\}$ (i.e. all pairs of states such that at least one is a final state).
- $\delta((q_1, q_2), \sigma) = (\delta(q_1, \sigma), \delta(q_2, \sigma))$.

Then, you can check that this DFA accepts a string iff one of M_1 or M_2 accepts it. ⊠

Theorem 1.5 (Intersection Theorem for Regular Languages)**.** *If L_1 and L_2 are regular languages over the same alphabet Σ, then $L_1 \cap L_2$ is a regular language.*

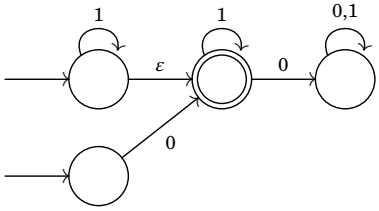
The proof is the same as the proof for Theorem 1.4, except that the automaton has different accept states: (q_1, q_2) is an accept state iff q_1 is an accept state of M_1 *and* q_2 is an accept state of M_2.

There are several other operations that we can do on regular languages: suppose A and B are regular languages.

- The complement $\neg A = \{w \in \Sigma^* \mid w \notin A\}$ is regular.
- The reverse of a regular language $A^R = \{w_1 \dots w_k \mid w_k \dots w_1 \in A, w_i \in \Sigma\}$ is regular.
- Concatenation of regular languages yields a regular language $A \cdot B = \{st \mid s \in A, t \in B\}$.
- The star operator preserves regularity: $A^* = \{s_1 \dots s_k \mid k \geq 0 \text{ and each } s_i \in A\}$. Notice that the empty string ε is in A^*.

Most of these aren't too hard to prove, but the reverse property isn't at all obvious. Intuitively, given a DFA M that recognizes L, we want to construct a DFA M^R that recognizes L^R. If M accepts a string w, then w describes a directed path in M from the start state to an accept state. Intuitively, one would want to reverse all of the arrows and switch the start and accept states, but having multiple start states is a bit of a problem, and there might be ambiguity in the reversed transition "function." Well, we can certainly define it, and then say that this machine accepts a string if there is some path that reaches some accept state from some start state. This looks like a nondeterminstic finite automaton (NFA), which will be useful, but the fact that this implies regularity is not obvious either.

Here is an NFA:



This NFA accepts the language $\{0^i 1^j \mid i,j \geq 0\}$. Observe that undefined transitions are OK (though they lead to rejection), and ε-transitions (where it just doesn't read anything, or reads the empty string) are also OK.

Definition 1.6. A non-determinsitic finite automaton (NFA) is a 5-tuple $N = (Q, \Sigma, \delta, Q_0, F)$, where

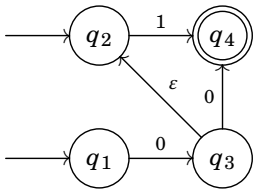
- Q is the finite set of states;
- Σ is a finite alphabet;
- $\delta : Q \times \Sigma_\varepsilon \to 2^Q$ is the transition function, which this time is on $\Sigma_\varepsilon = \Sigma \sqcup \varepsilon$ (where ε is the empty string) and sends this to a set of possible states (2^Q denotes the set of subsets of Q);
- $Q_0 \subseteq Q$ is the set of start states; and
- $F \subseteq Q$ is the set of accept states.

Then, N accepts a $w \in \Sigma^*$ if it can be written as $w_1 \dots w_n$ where $w_i \in \Sigma_\varepsilon$ (i.e. one might add some empty strings) and there exist $r_0, \dots, r_n \in Q$ such that:

- $r_0 \in Q_0$,
- $r_{i+1} \in \delta(r_i, w_{i+1})$ for $i = 0, \dots, n-1$, and
- $r_n \in F$.

Then, $L(N)$ is the language recognized by N, the set of strings accepted by it (as before).

As an example of this formalism, in the automaton N below,



$N = (Q, \Sigma, \delta, Q_0, F)$, where $Q = \{q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, $Q_0 = \{q_1, q_2\}$, $F = \{q_4\}$, and $\delta(q_3, 1) = \emptyset$ and $\delta(q_3, 0) = \{q_4\}$.

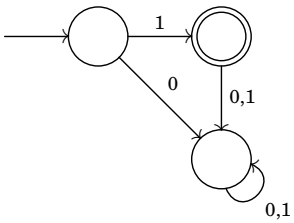
The deterministic model of computation is simple: follow the algorithm, then accept or reject. The nondeterministic method tries many possible paths, and accepts if any one of them does. Sometimes, this makes no difference, but in others, the result is provably different, but in both cases it will be useful.

2. Nondeterminism, Finite Automata, and Regular Expressions: 1/9/14

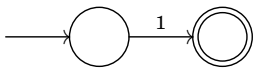
We were in the middle of determining whether the reverse of a regular language is regular. On specific languages, one doesn't have too much difficulty; for example, one can construct DFAs for the language consisting of strings beginning with a 1 and for the language consisting of strings ending with a 1. In general, though, can every right-to-left DFA be replaced with a normal DFA?

To understand this, we introduced the notion of an NFA. One wrinkle is that the definition provided allows multiple start states, but the textbook only allows one. This is easily shown to be equivalent: one can add a new start state and add ε-transitions to each former start state, to get an equivalent NFA with only one start state.

In general, NFAs are simpler than DFA. Here is a minimal DFA that recognizes the language {1}:



However, the minimal NFA (while it is in general harder to find) is simpler:



However, it turns out that this makes no difference!

Theorem 2.1. *For every NFA N, there exists a DFA M such that $L(M) = L(N)$.*

Corollary 2.2. *A language L is regular iff there exists an NFA N such that $L = L(N)$.*

Corollary 2.3. *A language L is regular iff its reverse L^{R} is regular.*

Proof of Theorem 2.1. There's no need to go into the incredibly gory details, but the meat of the proof is given here. The idea is that to determine if an NFA accepts, one keeps track of a set of states that could exist at the given time. Thus, one could use a DFA to simulate an NFA, where each state of the DFA is a collection of states of the NFA representing a slice of computation in parallel.

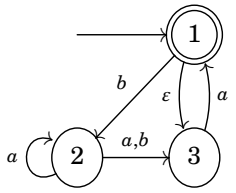
Thus, given some NFA $N = (Q, \Sigma, \delta, Q_0, F)$, output a DFA ($M = (Q', \Sigma, \delta', q_0', F')$, where $Q' = 2^Q$ (i.e. the set of subsets of Q), and $\delta' : Q' \times \Sigma \to Q'$ is given by

$$\delta'(R, \sigma) = \bigcup_{r \in R} \varepsilon(\delta(r, \sigma)),$$

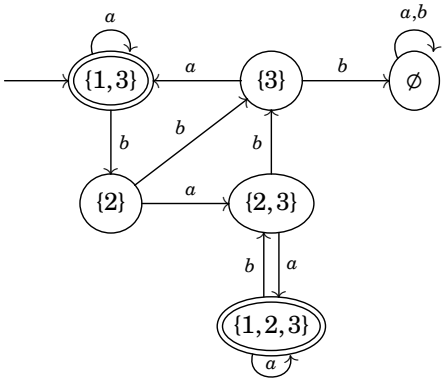
where this notation denotes the *ε-closure* of a set; for a set $S \subseteq Q$, the ε-closure of S, denoted $\varepsilon(S)$, is the set of states $q \in Q$ reachable from some $s \in S$ by taking 0 or more ε-transitions. This allows the DFA to deterministically handle ε-transitions.

Then, the start state is $q_0' = \varepsilon(Q_0)$, to account for ε-transitions from the start state(s), and $F' = \{R \in Q' \mid f \in R \text{ for some } f \in F\}$. These are the sets of states (of M) that contain any accept state, since acceptance on a NFA was defined to include any possible path accepting. ⊠

As an illustration, here is how this algorithm looks on the following NFA:



Then, $\varepsilon(\{1\}) = \{1, 3\}$. Then, the resulting DFA is



Notice that the empty set is present, and that $\{1\}$ and $\{1, 2\}$ are unreachable from the start state (so they could be placed on the graph, but they're pretty useless).

A useful rule is that NFAs can make proofs much easier. For example, for the union theorem, one could just glom the two DFAs together to get an NFA with two start states (that is disconnected, but that's OK). As another example, to show that the concatenation of two regular languages is regular, one can connect the accept states of an NFA M_1 that recognizes the first language to the start states of an NFA that recognizes the second language (and then make them both regular states, and so on).

Regular languages are also closed under the star operator, which treats a language as an alphabet, in some sense: the strings made up of compositions of substrings of a language A.

If L is regular, then let M be a DFA with $L(M) = L$. Then, one can just add ε-transitions from every accept state to every start state, creating an NFA N that recognizes L^*. Additionally, since $\varepsilon \in L^*$, it's necessary to make a single start state that always accepts, before seeing anything else (add ε-transitions to the previously existing start states).

Formally, given some DFA $M = (Q, \Sigma, \delta, q_1, F)$, produce an NFA $N = (Q', \Sigma, \delta', \{q_0\}, F')$, where $Q' = Q \cup \{q_0\}$ and $F' = F \cup \{q_0\}$. Then,

$$\delta'(q, a) = \begin{cases} \{\delta(q, a)\}, & \text{if } q \in Q \text{ and } a \neq \varepsilon \\ \{q_1\}, & \text{if } q \in F \text{ and } a = \varepsilon. \end{cases}$$

Now, we can show that $L(N) = L^*$.

- $L(N) \supseteq L^*$: suppose that $w = w_1 \dots w_k \in L^*$, where $w_i \in L$ for each i. Then, by induction on k, we can show that N accepts w.

  The base cases are $k = 0$, where $w = \varepsilon$, which was handled by the start state, and $k = 1$, where $w \in L$, which is also pretty clear.

  Inductive step: suppose N accepts all strings $v = v_1 \dots v_k \in L^*$ where $v_i \in L$. Then let $u = u_1 \dots u_k u_{k+1} \in L^*$, so that $u_j \in L$. Since N accepts $u_1 \dots u_k$ by induction and M accepts u_{k+1}, then one can quickly check that N also accepts all of u.
- $L(N) \subseteq L^*$. Assume w is accepted by N, so that we want to show that $w \in L^*$. If $w = \varepsilon$, then clearly $w \in L^*$, and now proceed by induction on the length of w; suppose that for all u of length at most k, N accepts u implies $u \in L^*$.

  Let w be accepts by N and have length $k+1$. Then, write $w = uv$, where v is the substring read after the last ε-transition. Then, $u \in L(N)$, since the last state before an ε-transition is an accept state. Then, u is a strictly smaller state than w, so $u \in L^*$ by the inductive hypothesis, and since the first thing after an ε-transition is a start state, then $v \in L(M)$. Thus, $w = uv \in L^*$, just by the definition of L^*.

Now, rather than viewing computation as a machine, regular expressions allow one to view computation as a set of rules.

Definition 2.4. Let Σ be an alphabet. Then, regular expressions over Σ can be defined inductively:

- For all $\sigma \in \Sigma$, σ is a regular expression, as well as ε and $\emptyset$.
- If R_1 and R_2 are regular expressions, then $R_1 R_2$ and $R_1 + R_2$ are both regular expressions, as is R_1^*.

A regular expression is associated to a language as follows: σ represents the language $\{\sigma\}$ and ε represents $\{\varepsilon\}$. Then, $\emptyset$ represents $\emptyset$. Then, if R_1 recognizes L_1 and R_2 recognizes L_2, then $R_1 R_2$ (sometimes written $R_1 \cdot R_2$) represents the concatenation $L_1 \cdot L_2$, $R_1 + R_2$ represents $L_1 \cup L_2$, and R_1^* represents L^*.

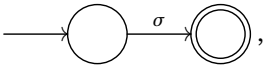
For example, if $\Sigma = \{0, 1\}$, then $0^* 1 0^*$ represents the language consisting of strings consisting of a single 1, and $\emptyset^*$ recognizes the language $\{\varepsilon\}$. The language of strings with length at least 3 and whose third symbol is a zero is given by the expression $(0+1)(0+1)0(0+1)^*$. The language $\{w \mid$ every odd position in w is a $1\}$ is represented by $(1(0+1))^*(1+\varepsilon)$. Notice that some tricks need to be done to include ε and such, and in general there are edge cases.

More interestingly, let $D = \{w \mid w$ has an equal number of occurrences of 01 and $10\}$.[2] This merits the impressive regex $1 + 0 + \varepsilon + 0(0+1)^*0 + 1(0+1)^*1$. This is because, excluding edge cases, a $w \in D$ is such that either w starts with a 0 and ends with a 0, or w starts with a 1 and ends with a 1. This is because sending $00 \mapsto 0$ and $11 \mapsto 1$ on a string preserves the property that the number of 01 substrings is equal to the number of 10 substrings, and this reduces it to a string of the form $1010\dots01$ or similar. Then, the three edge cases are those too short to contain a 01 or a 10, so they are 0, 1, and ε.

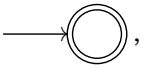
Proposition 2.5. *L can be represented by a regular expression iff L is regular.*

Proof. In the forward direction, given a regular expression, one will obtain an NFA N such that R represents $L(N)$. Proceed by induction on the number of symbols in R.

The base cases are $R = \sigma$, given by



$R = \varepsilon$, given by



and $R = \emptyset$ given by an NFA which accepts nothing.

In the general case, where R has length $k > 1$, suppose that a regular expression of length $\leq k$ represents a regular language. Then, it must be possible to decompose R in one of the following ways: $R = R_! + R_2$, $R = R_1 R_2$, or $R = R_1^*$. But the union, concatenation, and star closure of regular languages must be regular, and R_1 and R_2 must have length at most k, so they correspond to regular languages, and thus R does as well.

The other direction will be given in the next lecture.

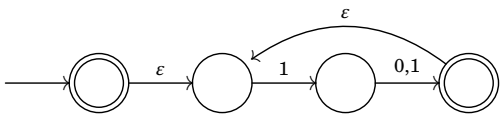
[2] These substrings are allowed to overlap.

3. Finite Automata vs. Regular Expressions, Non-Regular Languages: 1/14/14

"My name is Hans —" "And I'm Franz —" "... and we're here to pump you up!"

Looking at problems by how complex their descriptions are is a prevailing theme in computer science. Regular expressions are among the simpler ones. One piece of terminology is that a string $w \in \Sigma^*$ is accepted by a regular expression R, or matches R if it is in $L(R)$.

Last time, it was proven that if L is given by a regular expression, then L is regular. An example of the given algorithm gives the following NFA for the regular expression $(1(0+1))^*$:



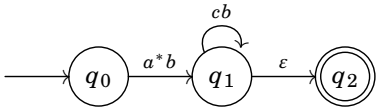
For the proof in the reverse direction, it's necessary to introduce a more general class of automata.

Definition 3.1. A generalized nondeterministic finite automaton (GNFA) is a 5-tuple $(G = Q, \Sigma, R, q_{\text{start}}, q_{\text{accept}})$, which is an NFA whose arcs are labeled by regular expressions:

- Q is the finite set of states, Σ is the finite alphabet as above, and q_{start} is the single start state and q_{accept} the single accept state, as per usual or close to it, and
- R is a transition function $Q \times Q \to \mathscr{R}$, where $\mathscr{R}$ is the set of regular expressions over Σ; that is, each arc is labeled by a regular expression.

G accepts a string w if there is a sequence $q_0 = q_{\text{start}}, q_1, \ldots, q_{n-1}, q_n = q_{\text{accept}}$ and strings $x_1, \ldots, x_n$ such that x_i matches $R(q_i, q_{i+1})$ for all i and $w = x_1 \ldots x_n$. Then, $L(G)$ is the set of strings G accepts, as normal.

For example, here is a GNFA over $\{a, b, c\}$:

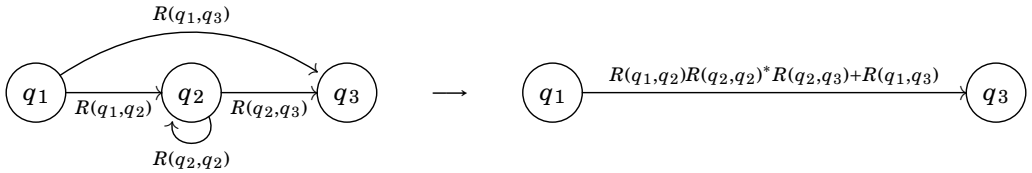


This GNFA recognizes the language matched by the regular expression $a^*b(cb)^*$.

Now, given some NFA, one can construct a sequence of GNFAs that lead to a regular expression. This algorithm can be called Convert(G) for a given GNFA G.

(1) First, add start and accept states and ε-transitions to (respectively from) the old start and accept states.
(2) Then, while the machine has more than two states:
   - Pick some state q, and remove it. Then, replace all paths through it $q_1 \to q \to q_2$ with arrows $q_1 \to q_2$ with regular expressions corresponding to the removed arrows. Specifically,
(3) Finally, the GNFA is just two states, with the transition given by a regular expression $R(q_1, q_2)$. Return this regular expression.

The ripping-out process looks like this, though it can become complicated.



Using this convention, the algorithm defines the transition function when a state q_{rip} has been removed as $R'(q_i, q_j) = R(q_i, q_{\text{rip}})R(q_{\text{rip}}, q_{\text{rip}})^*R(q_{\text{rip}}, q_j) + R(q_i, q_j)$.

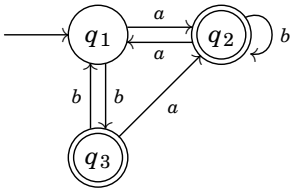
Then, if G' is the GNFA returned after ripping out q_{rip} and updating the transition function in the way defined above, then $L(G') = L(G)$. This is because either the sequence of states didn't include q_{rip}, in which case of course it still works, and if it did, then it has to match $R(q_i, q_{\text{rip}})R(q_{\text{rip}}, q_{\text{rip}})^*R(q_{\text{rip}}, q_j)$ there, which allows it to work.

Theorem 3.2. *More interestingly, $L(R) = L(G)$, where $R =$ Convert(G).*

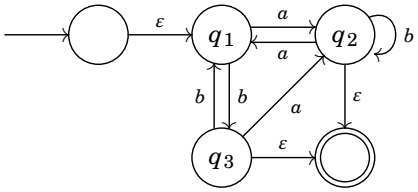
Proof. Proceed by induction on k, the number of states in G. In the case $k = 2$, this is pretty clear.

In the inductive step, suppose G has k states, and G' be the first GNFA created by ripping something out. Then, as mentioned above, $L(G) = L(G')$, and by induction $L(G') = L(R)$, so $L(G) = L(R)$ too. ⊠

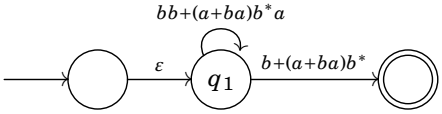
As an example, consider the following automaton:



After adding start and accept states, this looks like the following:



Then ripping out q_2 and q_3, this becomes



The resulting regular expression is rather ugly, but it does the job: $(bb+(a+ba)b^*a)^*b+(a+ba)b^*$.

In summary, there are several rather different ways of viewing the same class of languages or computation: through DFAs, NFAs, regular languages, and regular expressions. However, not all languages are regular. Intuitively, a DFA has a finite amount of memory, given by the set of states, but languages can be infinite. Alternatively, the set of DFAs is countable, but the set of all possible languages is not countable, so almost all languages must be non-regular.

However, things get nuanced: let $C = \{w \mid w$ has an equal number of 1s and 0s$\}$, and $D = \{w \mid w$ has an equal number of substrings 01 and 10$\}$. We saw last time in class that D is regular, given by an icky but still regular expression, and yet C is not! Intuitively, one needs to know the difference between the number of 0s seen and the number of 1s seen. This seems to require an infinite number of states, but of course this isn't rigorous. A better technique is given by the pumping lemma.

Lemma 3.3 (Pumping)**.** *Let L be an infinite regular language. Then, there exists a $p \in \mathbb{N}$ (i.e. $p > 0$) such that for all strings $w \in L$ with $|w| > p$, there is a way to write $w = xyz$, where:*

(1) $|y| > 0$ *(that is, $y \neq \varepsilon$),*
(2) $|xy| \leq p$*, and*
(3) *For all $i \geq 0$, $xy^iz \in L$.*

This is called the pumping lemma because given this decomposition, one can "pump" more copies of y into the string.

Proof of Lemma 3.3. Let M be a DFA that recognizes L, and P be the number of states in M. Then, if $w \in L$ is such that $|w| \geq p$, then there must be a path $q_0, \ldots, q_{|w|}$ for w, so by the pigeonhole principle, there exist i, j such that $q_i = q_j$, because there are $p+1$ states and p options. Without loss of generality, assume q_i happens before q_j.

Thus, let x be the part of w before q_i, y be that between q_i and q_j, and z be that after q_j. Then, y is nonempty, because there's at least one transition between q_i and q_j; the pigeonhole principle guarantees $j \leq p$, so $|xy| \leq p$, and since y comes from a loop in the state graph, xy^iz traces over that loop i times, and is still a viable way through the NFA, and is thus accepted. ⊠

Now, one can show that an (infinite) language is not regular by using the contrapositive: that for every $p \in \mathbb{N}$, there exists a $w \in L$ such that $|w| > p$ and the conditions don't hold.

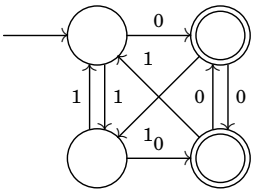
For example, one can show that $B = \{0^n1^n \mid n \geq 0\}$ isn't a regular language. Suppose B is regular, and since B is infinite, then let P be the pumping length for B. Then, let $w = 0^P1^P$. If B is regular, then there is a way to write $w = xyz$, $|y| > 0$, and so on. But since $|xy| < P$ and $w = 0^P1^P$, then y is all zeros. But then, $xyyz$ has more 0s than 1s, which means $xyyz \notin L$, violating the conditions of the pumping lemma.

Analogously, if C is as above the language with an equal number of 0s and 1s, then one can do something similar to show C isn't regular: suppose C is regular, and let P be its pumping length (since C is infinite). Let $w = 0^P 1^P$, and do the same thing as above; $xyyz$ has more 0s than 1s, and this forces the same contradiction. In essence, if one starts with a string within the language, and ends up with a string outside of the language, *no matter the partition into substrings*, then the language cannot be regular.

For another example, let $B = \{0^{n^2} \mid n \geq 0\}$, and suppose B is regular. If the pumping length is P, then let $w = 0^{p^2}$. Then, $w = xyz$ with $0 < |y| \leq p$. Then, $xyyz = 0^{p^2+|y|}$, but $p^2 + |y|$ isn't a square, because $p^2 < p^2 + |y| \leq p^2 + p < p^2 + 2p + 1 = (p+1)^2$.

4. Minimizing DFAs: 1/16/14

Today's question is whether a given DFA has a minimal number of states. For example, look at this one:

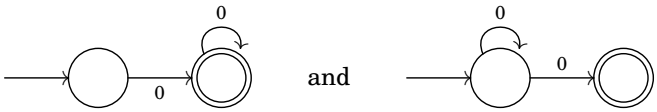


(4.1)

It seems like there is a DFA that accepts the same language with fewer states.

Theorem 4.2. *For every regular language L, there exists a unique*[3] *minimal-state DFA M^* such that $L = L(M^*)$, and moreover, there an efficient algorithm which, given a DFA M, will output M^*.*

This result is really nice, and its analogues generally don't hold for other models of computation. For one quick example, it's easy to construct distinct minimal-state NFAs for a given regular language, such as



Moreover, the question of an efficient algorithm is *open*.

For the proof, it will be useful to extend the transition function δ to $\Delta : Q \times \Sigma^* \to Q$, in which:

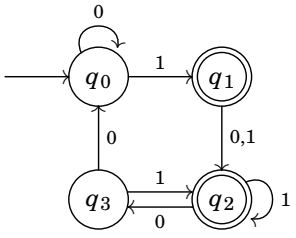
- $\Delta(q, \varepsilon) = q$,
- $\Delta(q, \sigma) = \delta(q, \sigma)$ for $\sigma \in \Sigma$, and
- $\Delta(q, \sigma_1 \ldots \sigma_{k+1}) = \delta(\Delta(q, \sigma_1 \ldots \sigma_k), \sigma_{k+1})$.

In other words, where does M end up after starting at q and reading $\sigma_1 \ldots \sigma_{k+1}$?

Definition 4.3. A $w \in \Sigma^*$ distinguishes states q_1 and q_2 iff exactly one of $\Delta(q_1, w)$ and $\Delta(q_2, w)$ is a final state. Two states p and q are distinguishable if there exists a $w \in \Sigma^*$ such that w distinguishes p and q, and are considered indistinguishable if for all $w \in \Sigma^*$, $\Delta(p, w) \in F$ iff $\Delta(q, w) \in F$.

In other words, one can run M on input w, and the result of the string can indicate whether it started at q_1 or q_2. Intuitively, indistinguishable states are redundant.

Look at the following automaton:



Then, ε distinguishes q_0 and q_1 (and any accept state from a non-accept state), 10 distinguishes q_0 and q_3, and 0 distinguishes q_1 and q_2.

Now, for some fixed DFA $M = (Q, \Sigma, \delta, q_0, F)$ and $p, q, r \in Q$, then write $p \sim q$ if p and q are indistinguishable, and $p \not\sim q$ otherwise. This is an equivalence relation, i.e.:

[3] This is understood to be up to isomorphism, i.e. relabelling the states.

- $p \sim p$, i.e. $\sim$ is reflexive. This is because $\Delta(p,w) \in F$ iff $\Delta(p,w) \in F$, which is silly but true.
- $p \sim q$ implies $q \sim p$, i.e. it's symmetric. This is because indistinguishability is an if-and-only-if criterion.
- $\sim$ is transitive: if $p \sim q$ and $q \sim r$, then $p \sim r$. This is because for all $w \in \Sigma^*$, $\Delta(p,w) \in F$ iff $\Delta(q,w) \in F$ iff $\Delta(r,w) \in F$, so $\Delta(p,w) \in F$ iff $\Delta(r,w) \in F$.

Thus, $\sim$ partitions the states of Q into disjoint sets called equivalence classes. Write $[q] = \{p \mid p \sim q\}$ for some state q.

Looking back at the automaton (4.1), there are two equivalence classes: those that accept and those that reject. It will be possible to construct a minimal DFA by contracting the DFA into its equivalence classes. More formally, this will be done according to the algorithm MINIMIZE-DFA, which inputs a DFA M and outputs a minimal DFA M_{MIN} such that $L(M) = L(M_{\text{MIN}})$, M_{MIN} has no inaccessible states, and M_{MIN} is irreducible (which will be shown to be equivalent to all states of M_{MIN} being distinguishable). Furthermore, there is an efficient dynamic-programming algorithm that can discover this M_{MIN} which outputs a set $D_M = \{(p,q) \mid p,q \in Q \text{ and } p \neq q\}$ of distinguishable states and $\text{EQUIV}_M = \{[q] \mid q \in Q\}$.

This algorithm can be considered as table-filling algorithm, where one can consider only the entries below the diagonal (since $\sim$ is an equivalence relation). Then:

- Base case: for all (p,q) such that p accepts and q rejects, we know $p \neq q$,
- Then, iterate: if there are states p,q and a symbol $\sigma \in \Sigma$ such that $\delta(p,\sigma) = p'$ and $\delta(q,\sigma) = q'$ and $p' \neq q'$, then $p \neq q$.

Repeat the iterative step until no more can be added. This must terminate, because there are a finite number of states and a finite number of symbols, so either the rule stops applying or everything is marked as inequivalent, which takes polynomial time.

Claim. (p,q) are marked as distinguishable by the table-filling algorithm iff indeed $p \neq q$.

Proof. Proceed by induction on the number of iterations in the algorithm: if (p,q) are marked as distiguishable at the start, then one is in F and the other isn't, so ε distinguishes them.

More generally, suppose that (p,q) are marked as distinguishable at some point. Then, by induction, there are states (p',q') that were marked as distinguishable and therefore are actually distinguishable by some string w. Without loss of generality, suppose $\Delta(p,w) \in F$ and $\Delta(q,w) \notin F$ (if not, then just switch them), and since p and q were marked, then there exists a $\sigma \in \Sigma$ such that $p' = \delta(p,\sigma)$ and $q' = \delta(q,\sigma)$. Then, σw distinguishes p and q, so $p \neq q$.

Conversely, suppose that (p,q) isn't marked by the algorithm; then, we want to show that $p \sim q$. Well, argue by contradiction, and suppose that $p \neq q$. Call such a pair a "bad pair." Then, there is a string w with $|w| > 0$ such that $\Delta(p,w) \in F$ and $\Delta(q,w) \notin F$ (or maybe switching p and q; it's the same thing).

Of all such bad pairs, let (p,q) be the pait with the shortest-length distinguishing string w. Then, $w = \sigma w'$ where $\sigma \in \Sigma$, so let $p' = \delta(p,\sigma)$ and $q' = \delta(q,\sigma)$. Then, (p',q') is a bad pair:

- Clearly, (p',q') can't have been marked by the algorithm, because then its next step would catch (p,q), which is a bad pair and thus isn't caught.
- Also, since $|w| > 0$, then w' is a string that can be used to distinguish p' and q': $\Delta(p,w) = \Delta(q',w')$ and $\Delta(q,w) = \Delta(q',w')$, so they are distinguished because p and q are.

Thus, (p',q') are distiguished by w', but $\left|w'\right| < |w|$, and w was supposed to be minimal, so the necessary contradiction is found. ⊠

This claim justifies using the notation "distinguishable" for the states marked by the table-filling algorithm.

Now, we can actually get down to minimizing DFAs. The algorithm is as follows:

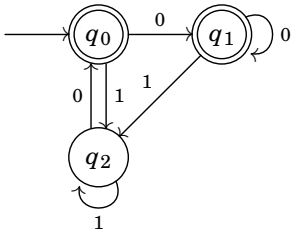
(1) First, remove all inaccessible states of M.
(2) Apply the table-filling algorithm to get the set EQUIV_M of equivalence classes for M,
(3) Define a DFA $M_{\text{MIN}} = (Q_{\text{MIN}}, \Sigma, \delta_{\text{MIN}}, q_{0,\text{MIN}}, F_{\text{MIN}})$ given by:
    - $Q_{\text{MIN}} = \text{EQUIV}_M$,
    - $q_{0,\text{MIN}} = [q_0]$,
    - $F_{\text{MIN}} = \{[q] \mid q \in F\}$, and
    - $\delta_{\text{MIN}}([q],\sigma) = [\delta(q,\sigma)]$.

    The start and accept states are just the equivalence classes of start and accept states. However, since we're working on equivalence classes, it's important to check whether δ_{MIN} is even well-defined.

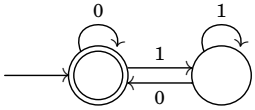
However, because all states in $[q]$ are indistinguishable, then the results after applying σ must also be indistinguishable (otherwise you could distinguish the stuff in $[q]$).

Claim. Then, $L(M_{\text{MIN}}) = L(M)$.

For example, suppose we want to minimize the following automaton. All states are accessible, so the first step of pruning the zero accessible states is uninteresting.

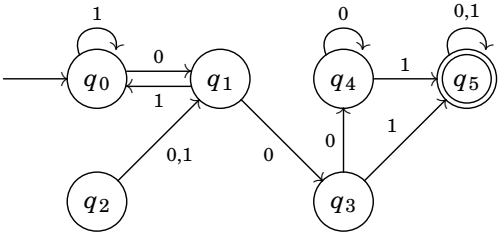


Then, once one writes down the equivalence classes, the result is



because the states q_0 and q_1 from the first automaton are equivalent.

For a more convoluted example, consider



q_2 isn't reachable, so it can be removed first. Now, q_3 and q_4 are indistinguishable, but the rest can all be seen to be distinguishable. Then, one can replace them with a common node.

Now, we know that M_{MIN} recognizes the same language, so why is it minimal? Suppose that $L(M') = L(M_{\text{MIN}})$ and M' is irreducible. Then, the proof will show that there is an isomorphism between M' and M_{MIN}.

Corollary 4.4. *M is minimal iff it has no inaccessible states and is irreducible.*

Proof. Let $M^{\min}$ be minimal for M. Then, $L(M) = L(M^{\min})$, has no inaccessible states, and is irreducible, and by the above claim (which will be proven next lecture), they are isomorphic. ⊠

5. The Myhill-Nerode Theorem and Streaming Algorithms: 1/21/14

The proof of Theorem 4.2 was mostly given last lecture, but relies on the following claim. This implies the uniqueness of the output of the algorithm as the minimal DFA for a regular language.

Claim. Suppose $L(M') = L(M_{\text{MIN}})$ and M' is irreducible and has no inaccessible states. Then, there is an isomorphism (i.e. a relabeling of the states) between M' and M_{MIN}.

Proof. The proof recursively constructs a map from the states of M_{MIN} to those of M'. Start by sending $q_{0,\text{MIN}} \mapsto q'_0$. Then, repeatedly suppose that $p \mapsto p'$ and $p \xrightarrow{\sigma} q$ and $p' \xrightarrow{\sigma} q'$. Then, send $q \mapsto q'$.

Then, one needs to show that this map is defined everywhere, that it is well-defined, and that it is a bijection. Then, from the recursive construction it's pretty clear that it preserves transitions (i.e. commutes with the transition functions δ).

First, why is it defined everywhere? For all states q of M_{MIN}, there is a string w such that $\Delta_{\text{MIN}}(q_{0,\text{MIN}}, w) = q$, because M_{MIN} is minimal; any inaccessible states were removed. Then, if $q' = \Delta(q'_0, w)$, then $q \mapsto q'$. This can be formally shown by induction on $|w|$, and it's particularly easier because of the recursive setup of the algorithm.

Next, it's well-defined. Suppose there are states q' and q'' such that $q \mapsto q'$ and $q \mapsto q''$. Then, we can show that q' and q'' are indistinguishable (and therefore that they must be equal, since M' is irreducible). Suppose u

and v are strings such that u sends $q_0' \to q'$ and v sends it to q''. Then, $u, v : q_{0,\text{MIN}} \to q$. But by induction on the lengths of u and v, one can show that they cannot be distinguished by q, but then, q' and q'' cannot distinguish u and v either, because $L(M) = L(M_{\text{MIN}})$, but this is a problem, because it implies that q is distinguishable from itself.

Now, why is the map onto? The goal is to show that for every state q' of M', there is a state q of M_{MIN} such that $q \mapsto q'$. Well, since M' also has no inaccessible states, then for every q' there is a w such that M' run on w halts at q'; then, let q be the state M_{MIN} halts on after running on w; then, $q \mapsto q'$ (which is formally shown by induction on $|w|$). The map is one-to-one because if $p, q \mapsto q'$, then p and q are distinguishable if they aren't equal, but their distinguishing string would also distinguish q' and q', which is a slight problem. ⊠

The Myhill-Nerode theorem is an interesting characterization of regular languages. One can define an equivalence relation on strings given some language: if $L \subseteq \sigma^*$ and $x, y \in \Sigma^*$, then x and y are indistinguishable to L if for all $z \in \Sigma^*$, $xz \in L$ iff $yz \in L$. One writes this as $x \equiv_L y$.

Claim. $\equiv_L$ is in fact an equivalence relation for any language L.

Proof. These follow from properties of "iff:" X iff X; X iff Y is equivalent to Y iff X; and if X iff Y and Y iff Z, then X iff Z. Then, plug into the definition of $\equiv_L$. ⊠

Theorem 5.1 (Myhill-Nerode)**.** *A language L is regular iff the number of equivalence classes of Σ^* under $\equiv_L$ is finite.*

This powerful statement gives a complete characterization of regular languages. But it also has nothing to do with the machine construction of regular languages that was their original definition, which is interesting.

Proof of Theorem 5.1. Suppose L is regular, and let $M = (Q, \Sigma, \delta, q_0, F)$ be a minimal DFA for L. Define an equivalence relation $x \sim_M y$ for $\Delta(q_0, x) = \Delta(q_0, y)$; that is, M reaches the same state when it reads in x and y.

Claim. $\sim_M$ is an equivalence relation with $|Q|$ equivalence classes.

Proof. The number of classes is pretty clear: it's bounded above by the number of states in M, but since M is minimal, then every state is accessible, so there must be $|Q|$ states.

As for why it's an equivalence relation, this is basically the same thing with the iff as for $\equiv_L$. ⊠

Next, if $x \sim_M y$, then $x \equiv_L y$, which is basically just checking the definitions: $x \sim_M y$ implies that for all $z \in \Sigma^*$, xz and yz reach the same state of M, so $xz \in L$ iff $yz \in L$, since $L = L(M)$, and this is exactly the definition of $x \equiv_L y$. Thus, there are at most $|Q|$ equivalence classes of $\equiv_L$, and in particular this number must be finite.

The specific claim that $x \sim_M y \implies X \equiv_L y$ shows that there are at most as many classes in $\equiv_L$ as in $\sim_M$ is because if one takes one x_i from each equivalence class of $\equiv_L$, then $x_i \not\equiv_L x_j$ if $i \neq j$, and therefore $x_i \not\sim_M x_j$, so we have at least as many equivalence classes of $\sim_M$.

In the reverse direction, we can build a DFA from the set of equivalence classes of $\equiv_L$. Specifically, suppose $\equiv_L$ has k equivalence classes. Then, let $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is the set of equivalence classes of $\equiv_L$.
- $q_0 = [\varepsilon] = \{y \mid y \equiv_L \varepsilon\}$.
- $\delta([x], \sigma) = [x\sigma]$.
- $F = \{[x] \mid x \in L\}$.

Then, it happens (though one has to check this) that M accepts x iff $x \in L$. Part of this involves demonstrating that δ is well-defined. ⊠

Now, with this theorem, one has another way of proving a language is or isn't regular. One can show that a language isn't regular by exhibiting infinitely many strings $w_1, w_2, \ldots$ that are distinguishable to L, i.e. for each pair w_i and w_j there is a $z_{ij} \in \Sigma^*$ such that exactly one of $w_i z_{ij}$ and $w_j z_{ij}$ is in L. This set of w_i is called a distinguishing set of L. Thus, every language L has either a DFA or a distinguishing set.

Theorem 5.2. $L = \{0^n 1^n \mid n \geq 0\}$ *is not regular.*

Proof. This was already proven using the Pumping Lemma, but consider the set $S = \{0^n \mid n \geq 1\}$. This is a distinguishing set for L, because for any $0^m, 0^n \in S$ with $m \neq n$, let $z = 1^n$. Then, $0^m 1^m \in L$, but $0^m 1^n \notin L$. Thus, every pair of strings in S are distinguishable to L, but S is infinite, so L cannot be regular. ⊠

That was pretty slick. This theorem can often be easier for proving a language isn't regular than the pumping lemma.

This relates to another question: how can one check if two regular expressions are equivalent? One has algorithms for converting regular expressions into DFAs, then DFAs into minimal DFAs, and then constructing an isomorphism; thus, one can combine these algorithms. However, minimizing a regular expression is considerably more complex, unless P = NP.

Streaming Algorithms. The intuition behind a streaming algorithm is that one must be able to respond to a data stream in real time. Thus, such an algorithm reads one character of input at a time, but cannot access past data, and has some small amount of memory and time to make computations.

For example, a streaming algorithm to recognize $L = \{x \mid x \text{ has more 1s than 0s}\}$ is as follows:

- Initialize $C := 0$ and $B := 0$, and let c be the bit we're currently looking at.
- Then, if $C = 0$, then set $B := x$ and $C := 1$.
- If $C \neq 0$ and $B = x$, then increment C.
- If $C \neq 0$ and $B \neq x$, then decrement C.
- When the stream stops, accept iff $B = 1$ and $C > 0$.

The idea is that B is the current majority value and C is how much more common it is. This requires $O(\log n)$ memory, which is pretty nice.

Streaming algorithms differ from DFAs in several ways:

(1) Streaming algorithms can output more than just a single bit (i.e. can do more than just accept or reject).
(2) The memory or space used by a streaming algorithm can increase with the length of the string, e.g. $O(\log n)$ or $O(\log\log n)$.
(3) Streaming algorithms can make multiple passes over the data, or be randomized.

Theorem 5.3. *Suppose a language L is recognized by a DFA with $|Q| \leq 2^p$ states. Then, L is computable by a streaming algorithm that uses at most p bits of space.*

Proof. The algorithm can essentially directly simulate the DFA directly; since there are 2^p states or fewer, then only p bits are necessary to hold the state. Then, it reads the state and the next character to determine whether or not to accept. ⊠

Definition 5.4. For any $n \in \mathbb{N}$, let $L_n = L \cap \Sigma^n$, i.e. the set of strings of a language L of length exactly n.

Theorem 5.5. *Suppose L is computable by a streaming algorithm A using $f(n)$ bits of space. Then, for all n, L_n is recognized by a DFA with at most $2^{f(n)}$ states.*

Proof. Create a DFA M with one state for every possible configuration of the memory of A. Then, M simulates A, where the state transitions are given by how A updates its memory given some input in the stream. ⊠

Notice that we don't care about how fast streaming algorithms run, though most of the examples given, not to mention all of those used in the real world, are reasonably time-efficient. The key is the strict conditions on space complexity.

6. Streaming Algorithms: 1/23/14

Streaming algorithms are important both in theory and in practice; if there is to be any theory of "big data," it will rely heavily on them. The trick is that you can't know when the stream stops; instead, the solution has to be updated to be correct at each point. We saw that a DFA with at most 2^p states can be simulated with a streaming algorithm with p bits of memory, and in a sort of converse, if a language L can be computed by a streaming algorithm with $f(n)$ bits of space, then $L_n = L \cap \Sigma_n$ (which is a finite and therefore language) is recognized by a DFA with at most $2^{f(n))}$ states; this DFA is created by simulating the streaming algorithm.

These two results allow the theory of finite automata to be applied to streaming algorithms. For example, if L is the language of strings over $\{0,1\}$ that have more 1s than 0s, is there a streaming algorithm for L that uses strictly less than $\log_2 n$ space? Not really; the algorithm given in the last lecture is optimal.

Theorem 6.1. *Every streaming algorithm for L needs at least $(\log_2 n) - 1$ bits of space.*

Proof. For convenience, let n be even (the proof also works for n odd, but it's a little ickier), and let $L_n = \{0,1\}^n \subseteq L$. Then, we will give a set S_n of $n/2+1$ strings which are pairwise distinguishable in L_n. Then, by the Myhill-Nerode theorem, every DFA recognizing L_n requires at least $n/2+1$ states, and therefore that every streaming algorithm for L requires $(\log_2 n)-1$ bits of space.

In greater detail, let $S_n = \{0^{n/2-i}1^i \mid i = 0,\dots,n/2\}$, and we want to show that every pair of strings $x, y \in S$ are distinguishable by L_n (that is, $x \not\equiv_{L_n} y$). Then, $x = 0^{n/2-k}1^k$ and $y = 0^{n/2-j}1^j$, where without loss of generality we suppose that $k > j$.

Then, $z = 0^{k-1}1^{n/2-(k-1)}$ distinguishes x and y in L_n, because xz has $n/2-1$ zeroes and $n/2+1$ ones, so $xz \in L_n$, and yz has $n/2+(k+j-1)$ zeroes and $n/2-(k-j-1)$ ones, but $k-j-1 \geq 0$, so $yz \notin L_n$.

Thus, all pairs of strings in S_n are distinguishable in L_n, so $\equiv_{L_n}$ has at least $|S_n|$ equivalence classes. Thus, every DFA recognizing L_n must have at least $|S_n|$ states. Thus, by Theorem 5.3, every streaming algorithm for L requires at least $\log_s |S_n|$ bits of space. Since $|S_n| = n/2+1$, then this completes the proof. ⊠

There's something interesting going on here: DFAs are used to say something about a model of computation strictly more powerful than themselves, in some sense of bootstrapping.

The number of distinct elements problem accepts as input some $x \in \{1,\dots,2^k\}^*$, where $2^k \geq |x|^2$ and computes the number of distinct elements appearing in x. Then, there is a streaming algorithm for this problem in $O(kn)$ space (where $n = |x|$), since this allows one to store all of the input. But this suggests a better solution might exist — morally, it's not even a streaming algorithm. But it's harder to do better!

Theorem 6.2. *Every streaming algorithm for the distinct elements problem requires $\Omega(kn)$ space.*

Proof. This problem will be solved by direct reasoning, rather than passing to a DFA, because the output is more than one bit.

Define $x, y \in \Sigma^*$ to be DE-distinguishable if there exists some $z \in \Sigma^*$ such that xz and yz contain a different number of distinct elements.

Lemma 6.3. *Let $S \subset \Sigma^*$ be such that every pair $x, y \in S$ is DE-distinguishable. Then, every streaming algorithm for the distinct elements problem needs at least $\log_2|S|$ bits of space.*

Proof. By the pigeonhole principle, if an algorithm A uses strictly less than $\log_2|S|$ bits, then there are distinct $x, y \in S$ that lead A to the same memory state. Then, for any z, xy and yz look identical, so to speak, to A and aren't distinguished, so $x \sim y$. ⊠

Lemma 6.4. *There is a DE-distinguishable set S of size $2^{\Omega(kn)}$.*

Proof. For each subset $T \subseteq \Sigma$ of size $n/2$, define x_T to be any concatenation of strings in T. Then, if T and T' are distinct, then x_T and $x_{T'}$ are distinguishable, because $x_T x_T$ has $n/2$ distinct elements, and $x_T x_{T'}$ has strictly more than $n/2$ elements, because they must differ somewhere.

Now, if $2^k > n^2$, we want to show that there are $2^{\Omega(kn)}$ such subsets, since there are $\binom{|\Sigma|}{n/2}$ choices, but since $|\Sigma| = 2^k$, then using the fact that $\binom{m}{c} \geq (m/c)^c$, which is true because

$$\binom{m}{c} = \frac{m!}{c!(m-c)!} = \frac{m\cdot(m-1)\cdots(m-c+1)}{c(c-1)\dots(2)(1)} \geq \frac{m^c}{c^c},$$

because when you subtract the same small amount from the numerator and the denominator, the fraction actually gets larger. Thus, $\binom{|\Sigma|}{n/2} \geq \binom{2^{kn/2}}{(n/2)^{n/2}} = 2^{kn/2 - n/2\log_2(n/2)} = 2^{\Omega(kn)}$ because $2^k > n^2$. ⊠

Now, throw these two lemmas together: since there is set of pairwise distinguishable elements of size $2^{\Omega(kn)}$ elements, then every streaming algorithm for this problem needs at least $\Omega(kn)$ memory. ⊠

Randomness actually makes this better; if the algorithm is allowed to make random choices during its process, there's a much better algorithm that can approximate the solution to about 0.1% error using some pretty clever hashing and $O(k+\log n)$ space.

What if the alphabet has more than two characters? A generalization of this algorithm is to find the top several items in some larger alphabet. This is a very common algorithm used in the real world. Formally, given a k and a string $x = x_1 \dots x_n \in \Sigma^n$, output the set $S = \{\sigma \in \Sigma \mid \sigma \text{ occurs more than } n/k \text{ times in } x\}$. Thus, by counting, $|S| \leq k$.

Theorem 6.5. *There is a two-pass streaming algorithm for this problem using $O(k(\log|\Sigma| + \log n))$ space.*

Proof. On the first pass, initialize a set $T \subseteq \Sigma \times \mathbb{N}$, which is initially empty. This will associate each symbol of the alphabet with a count of how many times it appeared, so that when reading (σ, m), remove (σ, m) from T and add $(\sigma, m+1)$ to it. Alternatively, if $|T| < k-1$ and $(\sigma, m) \notin T$ for any m, then there's nothing to remove, so add $(\sigma, 1)$ to the set. Otherwise, there is a symbol we haven't seen before, so decrement each counter (adding $(\sigma', m'-1)$ and removing each (σ', m), and removing any symbols with count zero).

Then, T contains all σ occurring more than n/k times in x. Then, in the second pass, count all occurrences of all σ' appearing in T to determine which ones occur at least n/k times. The key realization is that for all $(\sigma, m) \in T$, the number of times that σ appears in the stream is at least m, and that for every σ, its corresponding counter is decremented at most n/k times (because each time, some number of symbols are removed). Thus, all counts are within n/k of the true count, and in particular if σ appears more than n/k elements, then it exists in T at the end of the first pass. ⊠

This algorithm is simple, yet counterintuitive.

Returning to the language of strings with more 1s than 0s, we say a solution that uses $(\log_2 n) + 1$ bits of space, but a lower bound is $(\log_2 n) - 1$ space. How much space is needed exactly? Does it depend on whether n is odd or even?

Part 2. Computability Theory: Very Powerful Models

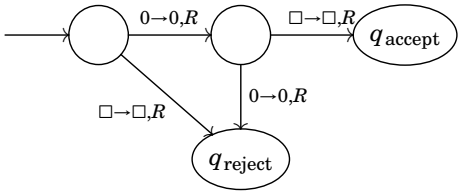
7. Turing Machines: 1/28/14

"When faced with a warehouse with miles of rewritable tape, Alan Turing did not despair. In fact, he invented computer science."

Now the course turns to a similarly simple model as a DFA, but this time it's very powerful.

A Turing machine has an infinite, rewritable tape and is considered to be "looking at" some particular symbol on the tape. It has some state q_0, reads (and possibly writes) to the tape, and then moves to the left or write. If it tries to move left from the left side of the tape, it is understood to remain at the leftmost part of the tape.

Here's a state diagram for a Turing machine:



Here, □ represents a blank symbol on the tape.

This Turing machine decides the language $\Sigma = \{0\}$. This is distinct from the notion of a Turing machine that recognizes Σ, i.e. one that halts and accepts 0, and infinite loops on other things.

Here's an example of a Turing machine that *decides* the language $L = \{w\#w \mid w \in \{0,1\}^*\}$. Notice that this language isn't regular, which one can show by Myhill-Nerode; all of the strings 0^i for $i > 0$ are pairwise distinguishable: 0^i and 0^j by $\#0^i$.

It is common to give a description of a Turing machine in pseudocode:

(1) If there is no # on the tape, or more than one #, reject.
(2) While there is a bit to the left of #,
   - Replace the first bit with X, and check if the first bit b to the right of the # is identical. If not, reject.
   - Replace that bit b with an X as well.
(3) If there is a bit to the right of #, reject; if else, accept.

It's also conventional to move everything over one index on the tape to the right, so that it's clear to the Turing machine when it has reached the leftmost edge of the input. Thus, once it sees a blank, it knows its at the end of the input, or the beginning.

Of course, it's necessary to have a formal definition for a Turing machine.

Definition 7.1. A Turing machine is a 7-tuple $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- Q is a finite set of states,
- Σ is the input alphabet, where $\square \notin \Sigma$ is the blank state.

- Γ is the tape alphabet, often different from the input alphabet, where $\square \in \Gamma$ and $\Sigma \subseteq \Gamma$.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.
- $q_0 \in Q$ is the start state,
- q_{accept} is the accept state, and
- q_{reject} is the reject state.

This means that one could store the configuration of a given Turing machine as a finite string in $(Q \cup \Gamma)^*$; for example, if the non-blank tape looks like 1101000110 and the Turing machine is on the sixth bit of tape in state q_7, then one would write $11010q_700110 \in \{Q \cup \Gamma\}^*$. Because the input is finite, then at any finite point in time, there are only finitely many non-blank symbols.

Definition 7.2. Let C_1 and C_2 be configurations of a Turing machine M. Then, C_1 yields C_2 if M is in configuration C_2 after running M in configuration C_1 for one step.

For example, if $\delta(q_1, b) = (q_2, c, L)$, then aaq_1bb yields aq_2acb. If $\delta(q_1, a) = (q_2, c, R)$, then $cabq_1a$ yields $cabcq_2\square$ (the Turing machine reads past the end of the input).

Definition 7.3. Let $w \in \Sigma^*$ and M be a Turing machine. Then, M accepts w if there are configurations $C_0, C_1, \ldots, C_k$ such that

- $C_0 = q_0 w$,
- C_i yields C_{i+1} for $i = 0, \ldots, k-1$, and
- C_k contains the accept state q_{accept}.

One can more specifically say that M accepts w in k steps.

Definition 7.4.

- A Turing machine recognizes a language L if it accepts exactly those strings in L. A language is called recursively enumerable (r.e.) or recognizable if there exists a Turing machine that recognizes L.
- A Turing machine decides a language L if it accepts those strings in L and rejects those strings not in L. A language is called decidable or recursive if there exists a Turing machine that decides L.

Recall that the complement of a language $L \subseteq \Sigma^*$ is $\neg L = \Sigma^* \setminus L$.

Theorem 7.5. *A language $L \subseteq \Sigma^*$ is decidable iff both L and $\neg L$ are recognizable.*

Proof. The forward direction is fairly clear, so look at the reverse direction. Given a Turing machine M_1 that recognizes L and a Turing machine M_2 that recognizes $\neg L$, how ought one to build a new machine M that decides L?

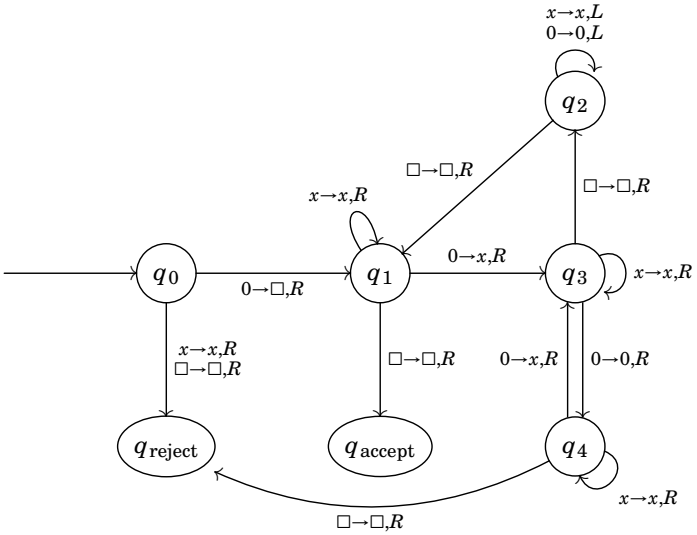
This can be done by simulating $M_1(x)$ on one tape, and $M_2(x)$ on another. Exactly one will accept; if M_1 accepts, then accept, and if M_2 accepts, then reject. This turns out to be equivalent to the one-tape model by interleaving the two tapes on one tape, but the exact details are worth puzzling out. ⊠

Consider the language $\{0^{n^2} \mid n \geq 0\}$. This is not at all regular, but it can be decided by a Turing machine! The decider will be given by pseudocode and then an actual diagram; in problem sets the state diagram isn't necessary, but it'll provide intuition for what sorts of pseudocode correspond to legal Turing machine state transitions.

(1) Sweep from left to right, and then cross out every other 0.
(2) If in step 1, and the tape had only one 0, then accept.
(3) If in step 1 there was an odd number of 0s (greater than 1), reject.
(4) Otherwise, move the head back to the first input symbol.
(5) Then, go to step 1.

Why does this work? Each time step 1 happens, the number of zeros is halved. Every number can be written as $n = k2^i$, where k is odd, and n is a power of 2 iff $k = 1$; this algorithm just determines the value of k.

Here's what the state diagram looks like:



One can also define multitape Turing machines, which can look at several different tapes and is focused at one point on each tape. However, this isn't more powerful (though it makes some things easier to prove).

Theorem 7.6. *Every multitape Turing machine corresponds to some single-state Turing machine.*

Proof. The details are messy (see Sipser), but the gist of it is to copy the inputs from the k tapes onto a single tape, separated by # marks. Then, where the Turing machine is on each state is represented by a dot: double the tape alphabet and then add a dot wherever the machine was. For example, 0 and 1 versus $\dot{0}$ and $\dot{1}$. Then, when the Turing machine moves, all the dots have to be read and updated... this is not the most elegant or efficient solution, but it makes everything work. One has a similarly messy construction given by interleaving the multiple tapes, too. ⊠

It's also possible to talk about nondeterministic Turing machines. The low-level details of this will be left to the reader, but it should be inuitively clear: there are multiple possible positions for each state-symbol pair. This *also* is no more powerful.

Theorem 7.7. *Every nondeterministic Turing machine M can be transformed into a single-tape Turing machine M that recognizes the language $L(N)$ (i.e. the language recognized by N).*

Proof idea. Pick a natural ordering on all strings in $\{Q \cup \Gamma \cup \#\}^*$.

Then, for all strings $D \in \{Q \cup \Gamma \cup \#\}^*$ in that ordering, check if $D = C_0 \# \cdots \# C_k$, where $C_0, \ldots, C_k$ is one of the accepting computation histories for w. If so, accept. ⊠

The fact that one can encode Turing machines as bit strings means that one can feed a Turing machine to another as input. One way to do this is to start with 0^n, indicating n states, then 10^m (i.e. m tape symbols), 10^k (the first k are input symbols), then 10^s indicates the start state, 10^t the accept state, 10^r the reject state, and 10^u the blank symbol. Then another 1. Then, $((p,i),(q,j,L)) = 0^p 10^i 10^q 10^j 10$ and $(p,i),(q,j,R)) = 0^p 10^i 10^q 10^j 100$ and so on. This is one way, not the most efficient; there exist other ways. But the point is that such an encoding exists.

Similarly, one can encode DFAs, NFAs, and any string $w \in \Sigma^*$ as but strings. For example, if $x \in \Sigma^*$, define its binary encoding to be $b_\Sigma(x)$. Then, pairs of strings can be encoded as $(x.y) := 0^{|b_\Sigma(x)|} 1 b_\Sigma(x) b\Sigma(y)$, from which the information from x and y can be retrieved, so to speak.

Using this, one can define some problems/languages:

- $A_{\text{DFA}} = \{(M,w) \mid M \text{ is a DFA that accepts } w\}$,
- $A_{\text{NFA}} = \{(M,w) \mid M \text{ is an NFA that accepts } w\}$, and
- $A_{\text{TM}} = \{(M,w) \mid M \text{ is a Turing machine that recognizes } w\}$.

The representations of DFAs and NFAs for these languages can be specified precisely, but the exact specification doesn't affect the complexity of the problem; one cares just that such a representation exists.

Theorem 7.8 (Universal Turing Machine)**.** *There exists a Turing machine U which takes as input the code of an arbitrary Turing machine M and an input string w, such that U accepts (M,w) iff M accepts w.*

This is an important, fundamental property of Turing machines! Notice that DFAs and NFAs don't have this property: A_{DFA} and A_{NFA} aren't regular, but A_{TM} is recognizable.

The idea behind the proof of Theorem 7.8 is to make a multi-state Turing machine which reads in (M,w) on one state, then has a tape that represents what M looks like when it is running,m another representing the transition function of M, and a final tape consisting of the current state of M (holding that number, where the states are numbered).

One interesting property of Turing machines is their granularity: you can extend them a lot (multi-tape, nondeterminism, and so on) and still get the same set of recognizable or decidable languages. Efficiency is unimportant.

The Church-Turing Thesis. *Everyone's intuitive notion of algorithms are captured by the notion of Turing machines.*

This is *not* a theorem; it is a falsifiable scientific hypothesis. Some people argue against it in the context of AI, but it's been thoroughly tested since the notion of Turing machines was first formulated decades ago.[4] Some serious people have written serious papers to try to prove the Church-Turing thesis, but always run into weird issues in the real world, such as quantum mechanics or black holes or such.

8. Recognizability, Decidability, and Reductions: 1/30/14

Recall the definition of a Turing machine, and the idea that a configuration of a Turing machine can be encoded into a string, which allows one to define a configuration yielding another configuration, or accepting a string, and so on.

Recall a Turing machine *recognizes* a language if the strings it accepts are exactly those in the language; it *decides* the language if it rejects all strings not in the language (rather than running indefinitely). Then, languages are called recognizable or decidable if there exists a Turing machine that recognizes or decides them, respectively.

Theorem 8.1. *A_{DFA}, the acceptance problem for DFAs, is decidable.*

Proof. A DFA is a special case of a Turing machine, and from the construction of the universal Turing machine U, one can run U on a DFA M and string w, and output the answer. Thus, U decides A_{DFA}. ☒

Theorem 8.2. *A_{NFA}, the acceptance problem for NFAs, is also decidable.*

Proof. A Turing machine can convert an NFA into a DFA, and then use the fact that A_{DFA} is decidable. ☒

Interestingly, however, A_{TM} is recognizable but not decidable! The universal Turing machine U recognizes it, but won't necessarily stop if the Turing machine it simulates doesn't stop.

More alarmingly, there exist non-recognizable languages, which will be proven in a moment. But then, assuming the Church-Turing thesis, this means there are problems that no computing device can solve! One can prove this by showing there is no surjection from the set of Turing machines to the set of languages over $\{0,1\}$. This means there are, in some sense, more problems to solve than programs to solve them.

Recall that a function $f : A \to B$ is onto (or surjective) iff for every $b \in B$, there exists an $a \in A$ such that $f(a) = b$. In some sense, the domain covers the entire codomain.

Theorem 8.3. *For any set L, let 2^L denote its power set; then, there is no onto function $f : L \to 2^L$.*

Proof. Suppose there exists an onto function $f : L \to 2^L$. Then, define $S = \{x \in L \mid x \notin f(x)\} \in 2^L$. Since f is onto, then there must be a $y \in L$ such that $f(y) = S$. Then, is $y \in S$? If it s, then $y \notin f(y) = S$, so it must not be. But then, $y \notin S$, which by the definition of S, implies $y \in f(y) = S$. ☒

Alternatively, one can avoid the contradiction by just letting $f : L \to 2^L$ be an arbitrary function, and define S as before. Then, for any $x \in L$, if $x \in S$, then $x \notin f(x)$, and if $x \notin S$, then $x \in f(x)$. But in either case, $f(x) \neq S$, so f is not onto.

Another way of phrasing this is that for every set L, its power set has strictly larger cardinality.

[4] Not to mention that the people trying to discredit it have over time looked more and more like crackpots.

Suppose that all languages over $\{0,1\}$ were recognizable. Then, the function f : {Turing machines} → {languages} is onto. Every Turing machine can be given by a bitstring, so the set of Turing machines can be thought of as a subset of $\{0,1\}^* = L$. But the set of languages is the set of subsets of $\{0,1\}^*$, i.e 2^L. But there's no onto function $L \to 2^L$, but therefore there can't be one from the set of Turing machines to 2^L either.

Perhaps this seems very abstract, but remember the Church-Turing thesis: take crucial part of this proof was that Turing machines are given by finite descriptions, just like programs in the real world, but problems aren't limited by this. Thus, this has implications on the real world.

This proof is sadly nonconstructive, but an example of an unrecognizable language will be given soon.

Russell's Paradox. In the early 1900s, logicians tried to define consistent foundations for mathematics. Suppose there is a set X, a universe of all possible sets. Then, people generally accepted the following axiom.

Frege's Axiom. *Let $f : X \to \{0,1\}$. Then, $\{S \in X \mid f(S) = 1\}$ is a set.*

Then, let $F = \{S \in X \mid S \notin S\}$, which seems reasonable. But then, if $F \in F$, then $F \notin F$, and if $F \notin F$, then $F \in F$! This is no good; the logical system is inconsistent.

For another lens on countability, one has the following theorem. This states that the real numbers are uncountable: they cannot be indexed by the integers. Here, integers are analogous to $\{0,1\}^*$, and real numbers to its power set.

Theorem 8.4. *There is no onto function $\mathbb{Z}^+ \to (0,1)$ (positive integers to real numbers).*

Proof. Suppose f is such a function, given as

$$1 \to 0.28347279\ldots$$
$$2 \to 0.88388384\ldots$$
$$3 \to 0.77635284\ldots$$
$$4 \to 0.11111111\ldots$$
$$5 \to 0.12345678\ldots$$

and so on. But then, define $r \in (0,1)$ such that its n^{th} digit is 1 if the n^{th} digit of $f(n)$ is not 1, and 2 if it is. Then, $f(n) \neq r$ for all n. ⊠

Here, $r = 0.11121\ldots$

This proof was discovered by Cantor, and he ended up going crazy after producing these and similar counterintuitive statements in set theory. It's known as Cantor's diagonal argument.

Note that there *is* a bijection between $\mathbb{Z}^+$ and $\mathbb{Z}^+ \times \mathbb{Z}^+$, given by counting $(1,1)$, $(1,2)$, $(2,1)$, $(2,2)$, $(3,1)$, and so on. There's even a bijection $\mathbb{Z}^+ \to \mathbb{Q}$, given by 1/1, 1/2, 2/1, 1/3, 3/2, 2/3, 3/1, 1/4, 4/3, and so on. There are several ways to do this, but this one is interesting in that it doesn't repeat any (which is irrelevant for the proof, but makes it more elegant).

Returning to undecidability, it's time to actually produce a language that is not recognizable.

Theorem 8.5. *A_{TM} is recognizable, but* not *decidable.*

Corollary 8.6. *Thus, $\neg A_{\text{TM}}$ is not recognizable.*

This is because we saw last lecture that if L and $\neg L$ are both recognizable, then L is decidable.

Definition 8.7. A language is co-recognizable (co-r.e.) if its complement is recognizable.

Corollary 8.8. *By this definition, A_{TM} is not co-recognizable.*

Proof of Theorem 8.5. Suppose H is a machine that decides A_{TM}, i.e. $H((M,w))$ accepts iff M accepts w. Then, let D be the Turing machine that outputs the opposite of H.

Run D on itself; then, if D accepts, then it rejects; and if it rejects, then it accepts. This is clearly impossible, so such an H cannot exist. ⊠

This is another tricky proof, but it's really a diagonalization argument in disguise: for any two Turing machines M_i and M_j, one can determine whether M_i accepts on input M_j. But then, one can define D to be the machine that does the opposite of the diagonal $M_i(M_i)$, and then D can't be anywhere in the table, because it would force a paradox.

Alternatively, here's a proof without contradictions.

Proof. Let H be a machine that recognizes A_{TM}. Since the universal machine is an example, then such an H exists. Then, $H((M,w))$ accepts if M accepts w, and loops or rejects if M does either. Then, let $D_H(M)$ do the opposite of whatever H does on input (M,M).

Thus, $D_H(D_H)$ rejects if D_H accepts D_H, and accepts if D_H rejects D_H... but this time, it's OK, because the third option is that D_H loops forever. Thus, we have $(D_H,D_H) \notin A_{\text{TM}}$, but H runs forever on the input (D_H,D_H), so H cannot decide A_{TM}. ⊠

Let HALT_{TM} denote the halting problem: $\text{HALT}_{\text{TM}} = \{(M,w) \mid M \text{ is a Turing machine that halts on input } w\}$.

Theorem 8.9. HALT_{TM} *is undecidable.*

Proof. The key idea of the proof is to prove by contradiction by reducing the problem: if H decides HALT_{TM}, then construct an M' that decides A_{TM}.

Suppose H is a Turing machine that decides HALT_{TM}. Then, let $M'(M,w)$ run $H(M,w)$; if H rejects, then reject, and if H accepts, then run a universal Turing machine M on w until it halts. If M accepts, then accept, and if M rejects, then reject. ⊠

This is a very high-level specification of a Turing machine, but that's perfectly all right.

This is an example of a common pattern: one can often prove a language L is undecidable by proving that if L is decidable, then so is A_{TM}. This says that L is as least as hard as A_{TM}, so to speak, and one could say $L \leq_{\text{m}} A_{\text{TM}}$.

Mapping Reductions.

Definition 8.10. $f : \Sigma^* \to \Sigma^*$ is a computable function if there exists a Turing machine M that, when run on any input $w \in \Sigma^*$, halts with just $f(w)$ written on its tape.

Definition 8.11. If A and B are languages over an alphabet Σ, then A is mapping-reducible to B, denoted $A \leq_{\text{m}} B$, if there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that $f(a) \in B$ iff $a \in A$. This f is called a mapping reduction from A to B.

Theorem 8.12. *If $A \leq_{\text{m}} B$ and $B \leq_{\text{m}} C$, then $A \leq_{\text{m}} C$.*

Proof. Let f be the computable function reducing A to B and g be that from B to C. Then, $g \circ f$ is computable, and reduces A to C. ⊠

Theorem 8.13. *If $A \leq_{\text{m}} B$ and B is decidable, then A is decidable.*

Proof. Let M decide B and f be a mapping reduction from A to B. Then, let M' be the Turing machine that computes $f(w)$ on input w, and then run M on it and output its answer. Then, one can reason that M' decides A. ⊠

Theorem 8.14. *If $A \leq_{\text{m}} B$ and B is recognizable, then A is recognizable.*

The proof is essentially the same as for Theorem 8.13, except that M is a recognizer and that perhaps it doesn't halt, in which case M' doesn't either.

Corollary 8.15.

- *If $A \leq_{\text{m}} B$ and A is undecidable, then B is undecidable.*
- *If $A \leq_{\text{m}} B$ and A is unrecognizable, then B is unrecognizable.*

Notice that the proof of Theorem 8.9 boils down to showing that $A_{\text{TM}} \leq_{\text{m}} \text{HALT}_{\text{TM}}$. It is also true that $\text{HALT}_{\text{TM}} \leq_{\text{m}} A_{\text{TM}}$, because one can define $f(M,w)$ to construct M', which accepts if $M(w)$ halts and otherwise loops forever, Then, $f(M,w) = (M',w)$.

Definition 8.16. $\text{EMPTY}_{\text{TM}} = \{M \mid M \text{ is a Turing machine and } L(M) \neq \emptyset\}$.

Theorem 8.17. EMPTY_{TM} *is undecidable.*

Proof. Assume by contradiction that there is a Turing machine E that decides EMPTY_{TM}. Then, E can be used to get a decider D for A_{TM}; specifically, let D on input (M,w) build a Turing machine M' which on input x does the following: if $x = w$, then run $M(w)$ and else reject. Then, run $E(M')$ and accept iff E rejects.

If M accepts w, then $L(M') = \{w\}$, and in particular is nonempty. But if M doesn't accept w, then $L(M') = \emptyset$. ⊠

Theorem 8.18. *In fact,* EMPTY_{TM} *is unrecognizable.*

Proof. In the proof of Theorem 8.17, the proof just constructs a reduction $\neg A_{\text{TM}} \leq_{\text{m}} \text{EMPTY}_{\text{TM}}$, so since $\neg A_{\text{TM}}$ is not recognizable, then neither is EMPTY_{TM}. ⊠

The regularity problem for Turing machines is $\text{REGULAR}_{\text{TM}} = \{M \mid M \text{ is a Turing machine and } L(M) \text{ is regular}\}$. It might be nice to be able to reduce a machine to a DFA, but this cannot be done in general.

Theorem 8.19. $\text{REGULAR}_{\text{TM}}$ *is unrecognizable.*

Proof. The proof will construct a reduction $\neg A_{\text{TM}} \leq_{\text{m}} \text{REGULAR}_{\text{TM}}$. Let $f(M,w)$ output a Turing machine M' which on input x runs $M(w)$ if $x = 0^n1^n$, and else rejects. Thus, if $(M,w) \in A_{\text{TM}}$, then $f(M,w)$ accepts $\{0^n1^n\}$, which isn't regular. But if $(M,w) \notin A_{\text{TM}}$, then $f(M,w) = M'$ accepts the empty set, which is regular. Thus, distinguishing regular languages from nonregular languages would allow one to solve $\neg A_{\text{TM}}$. ⊠

9. Reductions, Undecidability, and the Post Correspondence Problem: 2/4/14

> *"Though the Turing machine couldn't decide between the two options, at least it recognized a good thing when it saw it."*

Last time, we talked about uncomputability: some problems are uncomputable, even with any computational mode. Thanks to the Church-Turing thesis, this corresponds to our actual inability to solve these problems with algorithms. A concrete example of an undecidable problem is A_{TM}, the Turing machine decision problem. Even though it's recognizable, as a result of the existence of the universal Turing machine, $\neg A_{\text{TM}}$ isn't recognizable. This was proven by showing that for any machine H that recognizes A_{TM}, there is a Turing machine D_H on which H runs forever, and thus cannot decide.

Now that one concrete instance of an undecidable problem exists, it can be used to show other problems are undecidable; for example, the halting problem can be reduced to A_{TM} and thus is also undecidable. Formally, $A_{\text{TM}} \leq_{\text{m}} \text{HALT}_{\text{TM}}$, i.e. there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that for every $w \in \Sigma^*$, $w \in A_{\text{TM}}$ iff $f(w) \in \text{HALT}_{\text{TM}}$. This is called a mapping reduction, or sometimes a many-one reduction (there's no reason f has to be injective). This is useful because if $A \leq_{\text{m}} B$ and B is decidable (resp. recognizable), then A is decidable (resp. recognizable), and therefore if A is undecidable (resp. unrecognizable), then B is undecidable (resp. unrecognizable).

We saw that $A_{\text{TM}} \leq_{\text{m}} \text{HALT}_{\text{TM}}$ and therefore that $\neg A_{\text{TM}} \leq_{\text{m}} \neg \text{HALT}_{\text{TM}}$. This is a general principle: if $A \leq_{\text{m}} B$, then $\neg A \leq_{\text{m}} \neg B$, because the same computable function f ca be used. Another interesting question is as to whether $\neg A \leq_{\text{m}} A$; this is true if A is decidable. But $\neg \text{HALT}_{\text{TM}} \not\leq_{\text{m}} \text{HALT}_{\text{TM}}$, because HALT_{TM} is recognizable but not decidable, so $\neg \text{HALT}_{\text{TM}}$ isn't recognizable. But a reduction would imply it were recognizable, which is wrong, so no such reduction exists.

If $A \leq_{\text{m}} B$ and $B \leq_{\text{m}} A$, one might call A and B equivalent, and write $A \equiv_{\text{m}} B$. For example, we have see that $\text{HALT}_{\text{TM}} \equiv_{\text{m}} A_{\text{TM}}$. In some sense, this means that if a wizard gave you the choice of being able to magically solve one or the other, the choice is irrelevant; each will lead to a solution to the other. Intuitively, both problems have the same difficulty.

Since it's easy to minimize DFAs, one might try to do the same with Turing machines. Such an algorithm would be useful for program optimization. However, it's not that nice.

Theorem 9.1. EQ_{TM} *is unrecognizable.*

Proof. The proof idea is to show $\text{EMPTY}_{\text{TM}} \leq_{\text{m}} \text{EQ}_{\text{TM}}$.

Let $M_\emptyset$ be a "dummy" Turing machine which accepts nothing (i.e. there's no path from the start state to the accept state). Then, for any Turing machine M, define $f(M) = (M, M_\emptyset)$, which is pretty clearly computable. But then, $f(M) \in \text{EQ}_{\text{TM}}$ iff $M \in \text{EMPTY}_{\text{TM}}$, so $\text{EMPTY}_{\text{TM}} \leq_{\text{m}} \text{EQ}_{\text{TM}}$, and since EMPTY_{TM} is unrecognizable, then EQ_{TM} must be too. ⊠

Not all undecidable problems deal with arcana of Turing machines running on each other. Consider Post's Correspondence Problem[5] (PCP)[6], which is as follows. There are a bunch of dominoes with strings from Σ^* on

[5] Oh... I thought all this time it was the Post Correspondence Problem, i.e. correspondence by mail. Instead, it was apparently invented by a researcher named Post.

[6] No, this is not a drug. But it's pretty trippy, and legal to boot!

them on the top and the bottom, e.g.

$$\left[\frac{aaa}{a}\right] \quad \left[\frac{a}{c}\right] \quad \left[\frac{a}{aa}\right] \quad \left[\frac{c}{a}\right].$$

The goal is to choose from these dominoes, with repetition, so that when they're lined up, the top and bottom strings are the same. For example, here one would win with

$$\left[\frac{aaa}{a}\right]\left[\frac{a}{aa}\right]\left[\frac{a}{aa}\right].$$

In the case

$$\left[\frac{b}{ca}\right] \quad \left[\frac{a}{ab}\right] \quad \left[\frac{ca}{a}\right] \quad \left[\frac{abc}{c}\right],$$

there are multiple ways to win, including

$$\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right].$$

This sounds like a standard interview question, doesn't it? How would one write a program to solve it, or even to determine whether a win exists? Some observations can be made: the first domino must have the first letters of its top and bottom string must be identical, and similarly for the last domino and the last letters. Secondly, if all of the dominoes have a longer top than bottom (or vice versa, for all of them), then it's clear that no win will happen. Finally, if a domino has the same string on the top and bottom, it is the solution.

Thus, the problem is: given a set of domino types, is there a match? Let PCP be the language of sets of dominoes that have a match.

Theorem 9.2. PCP *is undecidable.*

This is a surprise: there's no clear relation of PCP to computation! But since Turing machines are extremely simple models of computation, they are relatively easier to model with, say, dominoes.

First, let the FPCP problem be nearly the same of PCP, except that the set of dominoes contains a distinguished first domino type, and all solutions must start with this domino. Let FPCP be the language of sets of dominoes with such a solution.

Theorem 9.3. FPCP *is undeciable.*

Proof. The proof will be by a reduction from A_{TM} to FPCP.

Recall the notion of a computation history for a Turing machine M on input w, which is a sequence of configurations $C_0, \dots, C_k$ such that C_0 is the starting configuration of the machine, each C_i yields C_{i+1} on the next character of w, and C_k is an accept state. Then, the goal is to produce an instance P of FPCP where the solutions generate strings of the form $C_0 C_1 \cdots C_k$. This takes seven steps:

(1) The first domino in P is $[\#/\#q_0 w\#]$ (i.e. the starting configuration of the Turing machine).
(2) If $\delta(q,a) = (p,b,R)$, the add the domino $[qa/bp]$.
(3) If $\delta(q,a) = (p,b,L)$, then add $[cqa/pcb]$ for all $c \in \Gamma$. These two classes of dominoes capture the transitions of a Turing machine, because after this domino has been added, the next domino must start with the next configuration given the input w, even encoding the tape!
(4) Add $[a/a]$ for all $a \in \Gamma$.
(5) Add $[\#/\#]$ and $[\#/\square\#]$, for filling in the configurations.
(6) For all $a \in \Gamma$ (including $\square$), add $[aq_{\text{acc}}/q_{\text{acc}}]$ and $[q_{\text{acc}}a/q_{\text{acc}}]$. This forces valid configurations to end on an accept state.
(7) The final domino is $[q_{\text{acc}}\#/\#]$.

Thus, one obtains a large but finite number of dominoes.

This means that $A_{\text{TM}} \leq_{\text{m}} \text{FPCP}$, so FPCP is undecidable. ⊠

This is useful for solving the more general PCP problem.

Proof of Theorem 9.2. The goal is to reduce $\text{FPCP} \leq_{\text{m}} \text{PCP}$.

For $u = u_1 u_2 \dots u_n$, where $u_i \in \Gamma \cup Q \cup \{\#\}$, define

$$\star u = *u_1 * u_2 * \cdots * u_n$$
$$u\star = u_1 * u_2 * \cdots * u_n *$$
$$\star u \star = *u_1 * u_2 * u_3 * \cdots * u_n *$$

Suppose that one has an FPCP instance of dominoes $[t_1/b_1],[t_2/b_2],\ldots,[t_n/b_n]$. Then, create a PCP problem with the dominoes $[\star t_1/\star b_1\star]$ and $[\star t_i/b_i\star]$ for all $i=1,\ldots,n$, along with an extra domino $[*\diamond/\diamond]$. The idea is that this requires one to start with $[\star t_1/\star b_1\star]$, but then continue with the rest of the game as normal. Thus, FPCP $\leq_{\mathrm{m}}$ PCP, so PCP is undecidable. ⊠

What we've shown is that given an (M,w) one can compute an instance of PCP that has a match iff M accepts w, so domino solitare is another universal model of computation. Instead of Turing machines or even computers, you could use dominoes.

10. Oracles, Rice's Theorem, the Recursion Theorem, and the Fixed-Point Theorem: 2/6/14

There's a lot of depth to computability theory that haven't really happened yet; it's not just about things being decidable, recognizable, or whatever. The connection to the Church-Turing thesis makes these have interesting implications, and can be very general. For example, the recursion theorem demonstrates how a Turing machine can access its own code. This makes proofs of undecidability shorter (albeit trippier), and also enables one to create quines in sufficiently powerful languages.

Definition 10.1. A decidable predicate $R(x,y)$ is a proposition about the input strings x and y that is implemented by some Turing machine M. That is, for all x and y, $R(x,y)$ is true implies that $M(x,y)$ accepts, and $R(x,y)$ is false implies that $M(x,y)$ rejects.

One can think of R as a function $\Sigma^*\times\Sigma^*\to\{0,1\}$. Examples include $R(x,y)$ which checks whether xy has at most 100 0s, or $R(N,y)$ which reports whether N is a Turing machine that halts on y within 99 steps. Notice how this relates to decidable languages. But it also offers a connection with recognizability:

Theorem 10.2. *A language A is recognizable iff there is a decidable predicate $R(x,y)$ such that $A=\{x\mid \exists y \text{ such that } R(x,y)\}$.*

Proof. Suppose A is of the format described above. Then, a Turing machine on input x can run $R(x,y)$ on all y (after enumerating them). If $R(x,y)$ is true for some y, then this machine will reach it, halt, and accept.

Conversely, suppose A is recognizable, and let M recognize it Then, define $R(x,y)$ to be true if M accepts x in y steps (where y is interpreted as a positive integer). Thus, M accepts x iff there is some number y of steps such that $R(x,y)$ is true. Thus, A can be written in the specified form. ⊠

Oracles. This will be a new model of computation, which formalizes the joke told last lecture about the wizard.

The idea is that an oracle Turing machine is able to magically decide whether a string is in some set. Of course, this is much more powerful than anything else we've seen.

Definition 10.3. More formally, an oracle Turing machine M is a Turing machine equipped with a set $B\subseteq\Gamma^*$ to which M may "ask" membership queries on a special oracle tape, and receives an answer in one step. (Formally, M enters a special state $q_?$, and if the string y on the oracle tape is in B, it goes to q_{YES}, and if not, it branches to q_{NO}.)

The upshot is that in pseudocode descriptions of an oracle Turing machine, one can test if $x\in B$ for some string x obtained from the input, and use it as a branch instruction.

The power of this construction is that it doesn't depend on B being decidable or recognizable. But this means that the notions of recognizability and decidability change slightly.

Definition 10.4.

- A is recognizable *with respect to* B if there is an oracle Turing machine equipped with B that recognizes A.
- A is decidable *with respect to* B if there is an oracle Turing machine equipped with B that decides A.

For example, A_{TM} is decidable with respect to $\mathrm{HALT}_{\mathrm{TM}}$, because on input (M,w), one can decide whether M accepts w by rejecting if (M,w) doesn't halt (which uses the oracle for $\mathrm{HALT}_{\mathrm{TM}}$), and running M if it does, and accepting if it does.

Theorem 10.5. $\mathrm{HALT}_{\mathrm{TM}}$ *is decidable in* A_{TM}.

If A is decidable in B, then one says that A Turing reduces to B, written $A\leq_{\mathrm{T}}B$. This is akin to mapping reduction, but is much more powerful: based on the answers to questions in B, one can ask more questions.

Theorem 10.6. *If $A \leq_m B$, then $A \leq_T B$.*

Proof. If $A \leq_m B$, then there is a computable function $f : \Sigma^* \to \Sigma^*$, such that $w \in A$ iff $f(w) \in B$. Thus, an oracle for B can be used to decide A, by computing $f(w)$ and testing if it's in B. ⊠

Theorem 10.7. $\neg \text{HALT}_{\text{TM}} \leq_T \text{HALT}_{\text{TM}}$.

Proof. Construct a Turing machine D that, on input (M, w), reject if $(M, w) \in \text{HALT}_{\text{TM}}$, and otherwise accept. ⊠

However, we saw last lecture that $\neg \text{HALT}_{\text{TM}} \not\leq_m \text{HALT}_{\text{TM}}$, so Turing reducibility is strictly more powerful than mapping reducibility.

Though this seems pretty awesome magical, there are still limits to powerful oracles. Even if we could solve the halting problem, there are problems that cannot be solved with Turing machines. For example, consider SUPERHALT, the language of (M, x) where M is an oracle Turing machine for the halting problem and M halts on x. The proof of this is essentially the same as that for the halting problem, and the key idea is that the oracle is for HALT_{TM}, not for SUPERHALT; then, the same diagonalization still holds.

This means that there's an infinite hierarchy of undecidable problems: let $\text{SUPERHALT}^0 = \text{HALT}_{\text{TM}}$ and $\text{SUPERHALT}^1 = \text{SUPERHALT}$; then, define SUPERHALT^n inductively as the set of $(M.x)$ where M halts on x, where M is an oracle Turing machine for SUPERHALT^{n-1}. Then, SUPERHALT^m is undecidable in SUPERHALT^n whenever $m > n$.

Exercise 10.8. Show that $\text{REVERSE} = \{N \mid N \text{ accepts } w^R \text{ if it accepts } w\}$ is undecidable.

Solution. Given a machine D for deciding REVERSE, one can decide A_{TM}: on input (M, w), generate the Turing machine M_w, which accepts on input 01 and on input 10 runs $M(w)$. Then, $D(M_w)$ indicates whether M accepts w, which is a problem. ⊠

Exercise 10.9. Consider the language of (M, w) such that M tries to move its head off of the left end of the input at least once when run on input w. Is it decidable?

Exercise 10.10. Consider the language of (M, w), where M run on input w moves left at least once.

These are undecidable and decidable, respectively, which is counterintuitive. But here's why:

Solution to Exercise 10.9. The proof will reduce from A_{TM} to this language: on input (M, w) make a Turing machine N that shifts w over one cell, marks a special symbol on the leftmost cell, and then simulates $M(w)$ on the tape. Then, if M moves to the marked cell but hasn't yet accepted, then N moves back to the right, and if M accepts, then N tries to move past the end. Then, (M, w) is in A_{TM} iff (N, w) isn't. ⊠

Solution to Exercise 10.10. To explicitly decide it, on input (M, w), run the machine for $|Q_M| + |w| + 1$ steps; accept if it's moved to the left, and reject otherwise.

In this sense, if it hasn't moved to the left, it won't; this is because when it's passed $|w|$ pieces of tape, then it's only looking at blanks, and then after a further $|Q_M| + 1$ steps, it's transitioned into at least one state twice by the pigeonhole principle, so it's trapped in a loop (since it always reads blanks), so it will keep repeating, and therefore never move left. ⊠

This leads to the Rice's theorem, which is an extremely powerful way to analyze the decidability of languages.

Theorem 10.11 (Rice)**.** *Let L be a language over Turing machine descriptions M. Then, suppose L satisfies the following two properties.*

(1) *L is nontrivial, i.e. there exist Turing machines M_{YES} and M_{NO} such that $M_{\text{YES}} \in L$ and $M_{\text{NO}} \notin L$.*
(2) *L is semantic, i.e. for all Turing machines M_1 and M_2 such that $L(M_1) = L(M_2)$, then $M_1 \in L$ iff $M_2 \in L$.*

Then, L is undecidable.

Basically every undecidable language we've seen so far is an easy consequence of applying Rice's theorem in the right way.

Another way to perceive this is to realize a property of Turing machines to be a function $P\{\text{Turing machines}\} \to \{0, 1\}$. Then, P is called nontrivial if there exists a M_{YES} such that $P(M_{\text{YES}}) = 1$ and an M_{NO} such that $P(M_{\text{NO}}) = 0$, and P is called semantic if whenever $L(M_1) = L(M_2)$, then $P(M_1) = P(M_2)$. Then, Rice's theorem states that the language $L_p\{M \mid P(M) = 1\}$ is undecidable.

The notion of a semantic property, one that depends only on the language of a given Turing machine, is hard to intuitively understand, so here are some examples.

- M accepts 0.
- For all w, $M(w)$ accepts iff $M(w^{\mathrm{R}})$ accepts (the reverse of the string).
- $L(M) = \{0\}$.
- $L(M)$ is empty.
- $L(M)$ is regular.
- M accepts exactly 163 strings.

In all of these cases, the languages for which these properties are true are undecidable!

Here are some properties which aren't semantic.

- M halts and rejects 0.
- M tries to move its head off the left end of the tape on input 0.
- M never moves its head left on input 0.
- M has exactly 154 states.
- M halts on all input.

Notice that they do not depend on $L(M)$.

Proof of Theorem 10.11. The goal is to reduce A_{TM} to the given language L. Let $M_\emptyset$ be a Turing machine that never halts, which as seen before can be constructed.

Suppose first that $M_\emptyset \notin L$, and choose an $M_{\mathrm{YES}} \in L$ (which must exist, because L is nontrivial). Then, reduce from A_{TM}: on input (M, w), output the Turing machine $M_w(x)$ which accepts if M accepts w and M_{YES} accepts x, and rejects otherwise. Then, $L(M_w) = L(M_{\mathrm{YES}}) \in L$. Conversely, if M doesn't accept w, then $L(M_w) = \emptyset = L(M_)$, but $M \notin L$, so $M_w \notin L$. Thus, this is a reduction.

In the other case, where $M_\emptyset \in L$, reduce instead $\neg A_{\mathrm{TM}}$ to L, by producing an $M_{\mathrm{NO}} \notin L$ (since L is nontrivial), and M_w accepts iff M accepts w and M_{NO} accepts x. This is a reduction by almost exactly the same proof. In this case we know the slightly stronger statement that L isn't recognizable. ⊠

Self-Reference and the Recursion Theorem. This part of the lecture, about Turing machines that think about themselves, is probably the trippiest part of the class.

Lemma 10.12. *There is a computable function $q : \Sigma^* \to \Sigma^*$ such that for any string w, $q(w)$ is the description of a Turing machine P_w that one very input, prints out w and accepts.*

This is easily shown: just make the Turing machine which reads w, and writes the Turing machine that ignores its input and writes w.

Getting weirder:

Theorem 10.13. *There is a Turing machine which prints its own code.*

Proof. First, define a Turing machine B which, on input M, produces the Turing machine which: on input w, calculate $M(P_M(w))$, which ignores w and feeds M to itself.

Now, consider the Turing machine $B(P_B)$. On input w, it prints $M(P_M)$, where $M = B$, so it prints $B(P_B)$. Whoa. ⊠

Another way of looking at this is that there is a program which prints its own description or source, or a sentence which describes itself, e.g.

"Print this sentence."

Another one, which corresponds to the machine B in the proof, is: "Print two copies of the following, the second in quotes: 'Print two copies of the following, the second in quotes.' "

There's yet another way to approach this, using lambda calculus. The self-printing function is $(\lambda x.\, x\, x)(\lambda x.\, x\, x)$, i.e. evaluates the second argument on the first, but it is its own second argument.

This leads to the really weird recursion theorem:

Theorem 10.14 (Recursion)**.** *For every Turing machine T computing a function $t : \Sigma^* \to \Sigma^* \to \Sigma^*$, there is a Turing machine R computing a function $r : \Sigma^* \to \Sigma^*$ given by $r(w) = t(R, w)$.*

In other words, R computes on itself, so one can think of T as obtaining its own description! Thus, Turing machine pseudocode is allowed to just obtain its own description and use it. Notice that the Church-Turing thesis means that this has implications to AI and free will...

Proof of Theorem 10.14. Suppose T computes $t(a,b)$. Let B be the Turing machine that, on input N (a Turing machine that takes 2 inputs), runs $N(P_N(w),w)$ on input w. Then, let M be defined as $T(B(N),w)$. Finally, define $R = T(B(P_M(w)),w)$. Thus, $w \xrightarrow{S_M} M \xrightarrow{B} S \to T(S,w)$.

So, what's S? S is what happens when (M,w) is fed into B, so S ends up just computing $T(B(P_M(w)),w)$! So $S = R$ and therefore R satisfies the required property. ⊠

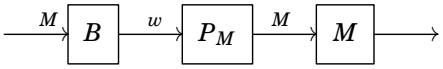
Puzzle: can you decide (M_1,w_1), (M_2,w_2), and $(M_3.w_3)$ (i.e. whether all three halt) with only two calls to an oracle to HALT_{TM}?

11. Self-Reference and the Foundations of Mathematics: 2/11/14

"I promise to never again squeeze humor out of self-reference." - xkcd

This lecture has an ambitious title, but indeed is an ambitious lecture. If one embraces the Church-Turing thesis that everything computable is represented by a Turing machine, then computability theory should say something about the foundations of mathematics.

One helpful way to understand statements like the Recursion Theorem is to draw pictures: For example, the Turing machine which prints its own description looks like



in the notation used last lecture. It's counterintuitive, but fiddle with it enough and it ought to make sense. But then, it can be used to allow Turing machines to access their own descriptions.

This often leads to faster proofs, e.g. that A_{TM} is undecidable:

Proof. Suppose A_{TM} is decidable, and let H be a decider for it. Then, let B be the machine that, on input w, obtains its own description B, runs H on (B,w), and then flips the output. This leads to a paradox. ⊠

This can be seen as a formalization or resolution of the free will paradox: no single deterministic machine can determine what an arbitrary Turing machine can do, by way of doing the opposite of what it reported.

Theorem 11.1 (The Fixed-Point Theorem)**.** *Let $t : \Sigma^* \to \Sigma^*$ be computable. Then, there exists a Turing machine F such that $t(F)$ outputs the description of a Turing machine G such that $L(F) = L(G)$.*

Proof. Here's the pseudocode for F: on input w,

(1) Run $t(F)$ and obtain an output string G. Then, interpret G as a description of a Turing machine.
(2) Run G on input w, and accept iff G does. ⊠

This can be used to given an alternate proof of Rice's theorem:

Proof of Theorem 10.11. Suppose one could decide a nontrivial semantic L. Then, $t(M)$, which outputs a Turing machine $M_{\text{NO}} \notin L$ if $M \in L$ and a Turing machine $M_{\text{YES}} \in L$ if $M \notin L$, is a computable function. Thus, it must have a fixed point M, but if $M \in L$, then $t(M) = M_{\text{NO}} \notin L$, but $L(M) = L(M_{\text{NO}})$, so L can't be semantic (and again with the same idea for $M \notin L$). ⊠

Now, for the ambitious part of the lecture: what can computability theory say about the foundations of mathematics itself? This will be relatively informal.

Definition 11.2. A formal system describes a formal language for writing finite mathematical statements in which:

- there is a definition of what statements are "true," and
- there is a notion of a proof of a statement.

Example 11.3. Every Turing machine M defines a formal system $\mathscr{F}$, where the mathematical statements are elements of Σ^*. Then, the string w represents the statement that M accepts w. Then, the true statements in $\mathscr{F}$ are exactly $L(M)$, and a proof that M accepts w is defined to be an accepting computation history for w. ◁

Definition 11.4.

- A formal system $\mathscr{F}$ is consistent (or sound) if no false statement in $\mathscr{F}$ has a valid proof in $\mathscr{F}$ (i.e. provability implies truth).

- A formal system $\mathscr{F}$ is complete if every true statement in $\mathscr{F}$ has a valid proof in $\mathscr{F}$ (i.e. truth implies provability).

These are nice notions to have for formal systems, and there are certainly many that are both consistent and complete, but we also want formal systems that has implications on mathematics.

Definition 11.5. A formal system $\mathscr{F}$ is interesting if

(1) Any mathematical statement about computation can be described as a statement of $\mathscr{F}$. That is, given (M,w), there is an $S_{M,w} \in \mathscr{F}$ such that $S_{M,w}$ is true in $\mathscr{F}$ iff M accepts w.
(2) Proofs are convincing, i.e. it should be possible to check that a proof of a theorem is correct. In other words, the set $\{(S,P) \mid P \text{ is a proof of } S \text{ in } \mathscr{F}\}$ is decidable.
(3) If $S \in \mathscr{F}$ and there is a proof of S describable in terms of computation, then there is a proof of S in $\mathscr{F}$. This means the system is expressive enough to understand computation histories and the like. In other words, if M accepts w, then there is a proof P in $\mathscr{F}$ of $S_{M,w}$.

There are plenty of uninteresting systems, such as the first-order theory of numbers with only addition; no notion of prime numbers, etc. exist. But it's consistent and complete. But there are also interesting systems.

The theorems described below, which place some limitations on mathematics, are powerful and have had huge implications on the mathematics of the past century. Their proofs are quite tricky, so sketches will be given.

Theorem 11.6 (Gödel 1931)**.** *For every consistent and interesting system $\mathscr{F}$, $\mathscr{F}$ is* incomplete*: there are mathematical statements in $\mathscr{F}$ that are true, but cannot be proven in $\mathscr{F}$.*

It's not immediately clear how something can be known to be true if it isn't provable; but the idea is that a system has a set of true statements and a set of proofs and the two don't match.

It would also be nice to be able to prove that mathematics is consistent, but...

Theorem 11.7 (Gödel 1931)**.** *The consistency of $\mathscr{F}$ cannot be proven in $\mathscr{F}$.*

Theorem 11.8 (Church-Turing 1936)**.** *The problem of checking whether a given statement in $\mathscr{F}$ has a proof is undecidable.*

Given a statement and a proof, it's decidable, but with only a statement there's in general nothing we can do.

Proof sketch of Theorem 11.6. Let $S_{M,w} \in \mathscr{F}$ be true iff M accepts w. Then, define a Turing machine $G(x)$ which:

(1) First, uses the Recursion Theorem to obtain its own description.
(2) Constructs a statement $S' = \neg S_{G,\varepsilon}$.
(3) Searches for a proof of S' in $\mathscr{F}$ over all finite-length strings. Accept if such a proof is found.

S' effectively says there is no proof of S' in $\mathscr{F}$, which can't be provable in $\mathscr{F}$, but is thus true. If such a proof exists, that G doesn't accept the empty string, then G does accept ε, which is a problem. ⊠

Notice that we must be looking in from some other system... this is quite a diversion to get into, and is outside of the scope of this lecture.

Proof of Theorem 11.7. Suppose it is possible to prove that $\mathscr{F}$ is consistent within $\mathscr{F}$. Then, let $S^{(=)} \neg S_{G,\varepsilon}$ as before, which is true, but has no proof in $\mathscr{F}$ (since G doesn't accept ε).

Thus, since $\mathscr{F}$ is consistent, then there's a proof in $\mathscr{F}$ that $\neg S_{G,\varepsilon}$ is true: if $S_{G,\varepsilon}$ is true, then there is a proof in $\mathscr{F}$ of $\neg S_{G,\varepsilon}$, and therefore, since $\mathscr{F}$ is consistent, then $\neg S_{G,\varepsilon}$ is true. But $S_{G,\varepsilon}$ and $\neg S_{G,\varepsilon}$ cannot both be true, so there's a contradiction, and therefore $\neg S_{G,\varepsilon}$ is true (i.e. the initial assumption was false).

But this is a contradiction, since it was just seen in the previous proof to be undecidable! ⊠

Finally, we can turn undecidability to mathematics.

Proof of Theorem 11.8. Let $\text{PROVABLE}_{\mathscr{F}}$ be the set of S such that $\mathscr{F}$ has a proof for S or for $\neg S$. Suppose $\text{PROVABLE}_{\mathscr{F}}$ is decidable with a decider P. Then, one can decide A_{TM} as follows:

- On input (M,w), run P on $S_{M,w}$.
- If P accepts, then a proof exists, so examine all possible proofs in $\mathscr{F}$ and accept when a proof of $S_{M,w}$ is found.

- If P accepts and a proof of $\neg S_{M,w}$ is found, then reject. (This doesn't conflict with the above, because $\mathscr{F}$ is consistent.)
- If P rejects, then reject. This is reasonable because if M accepts w, then there is always a proof of that. ⊠

12. A Universal Theory of Data Compression: Kolmogorov Complexity: 2/18/14

"For every technical topic, there is a relevant xkcd."

Kolmogorov complexity is a very general theory of data compression. As the Church-Turing thesis claims that everyone's intuitive notion of algorithms corresponds to Turing machines, one might want to formulate a universal theory of information. In particular, can one quantify how much information is contained in a string?

Consider two strings $A = 0101010101010101010101010101010101$ and $B = 1100100111011101011010010110010011$. Intuitively, A doesn't contain as much information as B. In some sense, A can be compressed into a smaller space than B, which intuits that it contains less information.

Thus, the thesis is that the amount of information contained in a string is the length of the shortest way to describe it. But how should strings be described? So far, the answer has usually been Turing machines.

Definition 12.1. If $x \in \{0,1\}^*$, then the shortest description of x, denoted $d(x)$, is the lexicographically shortest string $\langle M, w\rangle$ such that $M(w)$ halts with only x on its tape.

Note that different representations of Turing machines might give different specific examples of lexicographically shortest strings. The notation $\langle M, w\rangle$ represents a pairing of M and w, i.e. some sort of concatenation.

Theorem 12.2. *There is a one-to-one computable function $\langle\cdot,\cdot\rangle$ and computable functions $\pi_1, \pi_2 : \Sigma^* \to \Sigma^*$ such that $z = \langle M, w\rangle$ iff $\pi_1(z) = M$ and $\pi_2(z) = w$.*

Proof. As an example, let $Z(x_1 \cdots x_k) = 0x_1 0x_2 0 \cdots 0x_k 1$. Then, $\langle M, w\rangle = Z(M)w$ is a pairing, and one can write Turing machines which calculate π_1 and π_2. ⊠

Notice that in the above example, $|\langle M, w\rangle| = 2|M| + |w| + 1$. It would be perfect to have a pairing which has minimal size $|M| + |w|$, but this isn't possible in real life. It is possible to create better pairings; for example, let $b(n)$ represent the binary encoding of an $n \in \mathbb{N}$, and let Z be as above. Then, define $\langle M, w\rangle = Z(b(|M|))Mw$. This concatenates M and w, and tacks on the length of M, so that one can reconstruct the original information later. This asymptotically takes less space, since it's logarithmic.

For example, to encode $\langle 10110, 101\rangle$, $b(|10110|) = 101$, so the pairing is $\langle 10110, 101\rangle = 010001110110101$. Now, $|\langle M, w\rangle| \leq 2\log|M| + |M| + |w| + 1$. It's possible to still do better, with a $\log\log|M|$ term, and so on.

Definition 12.3. The Kolmogorov complexity of x (or the K-complexity of x) is $K(x) = |d(x)|$, where $d(x)$ is the lexicographically shortest description of x as above. This is given in the context of some fixed pairing function.

This is in some sense the maximal data compression for x. It's also quite abstract; examples will develop more naturally from properties of the Kolmogorov complexity.

Theorem 12.4. *There exists a fixed c such that for all $x \in \{0,1\}^*$, $K(x) \leq |x| + c$.*

Intuitively, this says that the amount of information in x isn't much more than $|x|$.

Proof of Theorem 12.4. Define a Turing machine M which simply halts on any input w. Then, on any string x, $M(x)$ halts with x on its tape, so if $c = 2|M| + 1$, then

$$K(x) \leq |\langle M, x\rangle| \leq 2|M| + |x| + 1 \leq |x| + c.$$ ⊠

This is just a rough upper bound; it might be possible to make it smaller.

The intuition for the following theorem is that the information in xx isn't that much more than that of x.

Theorem 12.5. *There is a fixed constant c such that for all $x \in \{0,1\}^*$, $K(xx) \leq K(x) + c$.*

Proof. Let N be the Turing machine which on input $\langle M, w\rangle$, computes $M(w)$ and prints it twice (i.e. $s = M(w)$, and it prints s). Then, if $\langle M, w\rangle$ is the shortest description of x, then $\langle N, \langle M, w\rangle\rangle$ is a description of xx, so

$$\begin{aligned} K(xx) &\leq |\langle N, \langle M, x\rangle\rangle| \leq 2|N| + |\langle M, x\rangle| + 1 \\ &\leq 2|N| + K(x) + 1 \leq c + K(x) \end{aligned}$$

where $c = 2|N| + 1$. ⊠

This generalizes: there isn't all that much more information in x^n than in x.

Theorem 12.6. *There is a fixed constant c such that for all $n \geq 2$ and $x \in \{0,1\}^*$, $K(x^n) \leq K(x) + c \log n$.*

Proof. Let N be the Turing machine which on input $\langle n, M, w\rangle$, assigns $x = M(w)$ and prints x n times, and let $\langle M, w\rangle$ be the shortest description for x. Then, $K(x^n) \leq K(\langle N, \langle n, M, w\rangle\rangle) \leq 2|N| + d \log n + K(x) \leq c \log n + K(x)$ for some constants c and d. ⊠

In general, it's not really possible to do better than $\log n$ here. But this allows us to understand $K((01)^n)$, with the example string given earlier: here, $K((01)^n) \leq c + d \log n$, which is much nicer than average.

One disadvantage of basing everything on Turing machines is that representations might be shorter in other languages. But this can also be formally addressed.

Definition 12.7. An interpreter is a semi-computable[7] function $p : \Sigma^* \to \Sigma^*$. Intuitively, it takes programs as input and (may) print their outputs.

Definition 12.8. Then, the shortest description of $x \in \{0,1\}^*$ relative to p is the shortest description of any string w such that $p(w) = x$, and is denoted $d_p(x)$. The Kolmogorov complexity relative to p is $K_p(x) = |d_p(x)|$.

Theorem 12.9. *For every interpreter p, there is a $c \in \mathbb{Z}$ such that for all $x \in \{0,1\}^*$, $K(x) \leq K_p(x) + c$.*

This intuitively says that interpreters might be faster, but not much more so asymptotically.

Proof of Theorem 12.9. Let M be the Turing machine which on input w, simulates $p(w)$ and writes its output to the tape. Then, $\langle M, d_p(x)\rangle$ is a description of x, and

$$K(x) \leq |\langle M, d_p(x)\rangle| \leq 2|M| + K_p(x) + 1 \leq c + K_p(x).$$ ⊠

This really is a universal notion, up to an additive constant. There's no reason we had to start with Turing machines; the results are always the same.

Theorem 12.10. *There exist incompressible strings of every length; for all n, there is an $x \in \{0,1\}^n$ such that $K(x) \geq n$.*

Proof. There are 2^n binary strings of length n, and $2^n - 1$ binary descriptions of length strictly less than n. And since strings to descriptions must be bijective, then there must be at least one n-bit string which doesn't have a description of length less than n. ⊠

Though it seems like this is a major flaw in this particular data compression algorithm, it's actually a very general statement about data compression: every compression algorithm must have some incompressible strings.

Worse off, a randomly chosen string is highly likely to be incompressible. In the intuition of the following theorem, n is large and c is small.

Theorem 12.11. *For all $n \in \mathbb{N}$ and $c \geq 1$, $\Pr_{x \in \{0,1\}^n}[K(x) \geq n - c] \geq 1 - 1/2^c$.*

Proof. There are 2^n string of length n, and $2^{n-c} - 1$ descriptions of length less than $n - c$. Hence, the probability that a random x satisfies $K(x) < n - c$ is at most $(2^{n-c} - 1)/2^n < 1/2^c$. ⊠

The fact that there exist incompressible strings is surprisingly useful, even outside of this corner of theory. For example, consider the language $L' = \{ww \mid w \in \{0,1\}^*\}$, which is not regular. There's a proof of this using Kolmogorov complexity: suppose L' is regular and D is a DFA such that $L(D) = L'$. Then, choose a string $x \in \{0,1\}^n$ such that $K(x) \geq n$. After reading in x, D is in state q_x.

Define a Turing machine M, which on input (D, q, n) tries all possible paths p of length n through D from state q; if p ends in an accept state, it prints the string that caused it to accept from q. But then, $\langle M, \langle D, q_x, n\rangle\rangle$ is a description of x, because D accepts xx, and if $y \neq x$, then xy isn't rejected. But its length is at most $c_1 + c_2 \log n$, because D and M have finite, constant descriptions that don't depend on x, and n takes just $\log n$ bits. But this means $K(x) = O(\log n)$, which is a contradiction, because x was chosen to be incompressible. Thus, L' is not regular.

Of course, you can use Myhill-Nerode to show this, but this proof is an example of a fundamentally different style of argument.

[7] This means that it is given by a Turing machine which may compute the answer, and may not halt.

Exercise 12.12. Give short algorithms for generating the following strings:

(1) 0100011011000001010011100101110111000 0
(2) 1235813213455891442333776109 87
(3) 1262412072050404032036288036288 00.

Solution. The first string just counts upwards from 1 in binary; the second concatenates the first several Fibonacci numbers; and the last concatenates the first ten factorials. These lead to efficient algorithms.

Well, this seems kind of hard in general. There's a formal way to think about this.[8] Does there exist an algorithm to perform optimal compression, and can algorithms tell us if a given string is compressible?

Theorem 12.13. *The language* COMPRESS $= \{(x,c) \mid K(x) \le c\}$ *is undecidable.*

This proof is different from most of the proofs we've seen before for undecidability: if it were decidable, then inutitively, one could construct an algorithm that prints the shortest incompressible string of length n, and then such a string could be succinctly described by the code of the algorithm.

Barry's Paradox. *The smallest integer that cannot be described in fewer than thirteen words.*

But this is a description on twelve words! Self-reference is mystifying.

Proof of Theorem 12.13. Suppose that COMPRESS is decidable. Then, let M be the Turing machine which, on input $x \in \{0,1\}^*$, interprets x as a number N. Then, for all $y \in \{0,1\}^n$, considered in lexicographical order, if $(y,N) \in$ COMPRESS, then print y and halt.

This algorithm finds the shortest string that y' such that $K(y') > n$. But then, $\langle M,x\rangle$ describes y', and $|\langle M,x\rangle| \le c + \log N$, so $N < K(y') \le c + 2\log N$, which is a problem for sufficiently large N. ⊠

The proof encodes Barry's paradox, in some sense.

This can be used to give (yet) another proof that A_{TM} is undecidable, by reducing from COMPRESS. Given a pair (x,c), construct a Turing machine $M_{x,c}$, which, on input w:

- For all pairs $\langle M',w'\rangle$ with $|\langle M',w'\rangle| \le c$, simulate each M' on w' in sequence.
- If some M' halts and prints x, then accept.

Then, $K(x) \le c$ iff $M_{x,c}$ accepts ε (or any string, really, because M ignores its input),

Kolmogorov complexity is still a very active topic, e.g. in Bauwens et al., *Short lists with short concepts in short time*, in the IEEE Conference on Computational Complexity, 2013, it is shown that the following problem *is* computable: given an input x, print a list of $|x|^2$ strings such that at least one string y on the list encodes a description of x, and where y has length at most $K(x)+c$. The fact that one has a list of strings allows one to approximate the Kolmogorov complexity, even though the exact problem is undecidable.

More on Formal Systems. Kolmogorov complexity leads to another view of the formal-systems approach to the foundation of mathematics.

Theorem 12.14. *For every interesting, consistent formal system $\mathcal{F}$, there exists a t such that, for all x, $K(x) > t$ is an unprovable statement within $\mathcal{F}$.*

Proof. Define a Turing machine which reads its input as an integer: M, which on input k searches over all strings x and proofs P for a proof P in $\mathcal{F}$ that $K(x) > k$. Then, output x if it is found.

If $M(k)$ halts, then it must print some output x'. Then, $K(x') = K(\langle M,k\rangle) \le c + |k| \le c + \log k$ for some c. But since $\mathcal{F}$ is consistent, then $K(x')k$ is true, but $k < c + \log k$ doesn't hold for large enough k, so choose a t for which this doesn't hold. Then, $M(t)$ cannot halt, so there is no proof that $K(x) > t$. ⊠

The theorem statement is even more absurd when one realizes that almost all strings of sufficiently large length are incompressible. But that's just life. Also, this leads to a huge number of randomly generated statements with no proof, but that are true with very high probability, a far reach from Gödel's abstruse statements.

[8] Mentioned in lecture today: http://xkcd.com/1155/.

Part 3. Complexity Theory: The Modern Models

13. Time Complexity: 2/20/14

"This is the way real math is done: first you do the proof, then you figure out what the theorem says."

Complexity theory asks not just what can or can't be computed in principle, but also what can and can't be computed with limited resources, e.g. time and space. Clearly, this is a bit more practical than these abstract notions of decidability and undecidability; something may be abstractly solvable, but still take forever to complete.

Complexity theory is safely enriched with many, many important open problems, and nobody has any idea how to solve them.

Asymptotics and Time Complexity. If $f, g : \mathbb{N} \to \mathbb{R}^+$, then recall that $f(n) = O(g(n))$ if there exist positive integers c and n_0 such that for every $n \geq n_0$, $f(n) \leq cg(n)$, i.e. as n gets large enough, f is bounded above by a fixed multiple of g. In the case where $f = O(g)$, one says that g is an *upper bound* of f.

For example, $2n^{4.1} + 200284n^4 + 2 = O(n^{4.1})$, and $3n\log_2 n + 5n\log_2\log_2 n = O(n\log_2 n)$, and $n\log_{10} n^{78} = O(n\log n)$, because $\log(n^{78}) = 78\log n$, and the constant can be swallowed up. In fact, since the change of base rule for logarithms involves scaling by a constant factor, the base of a logarithm is irrelevant with respect to this: $O(\log n) = O(\log_2 n) = O(\log_{10} n)$, so the base is usually unspecified.

Time complexity is measured by counting the number of steps it takes for a Turing machine to halt. Since we want the Turing machine to halt on all inputs, then we will only look at decidable problems right now.

Definition 13.1. Let M be a Turing machine that halts on all of its inputs. Then, the *running time* or *time complexity* of M is the function $T : \mathbb{N} \to \mathbb{N}$, where $T(n)$ is the maximum number of steps taken by M over all inputs of length n.

Definition 13.2. The time complexity class of a function $t(n)$ is TIME($t(n)$), the set of languages L' for which there is a Turing machine M that decides L' and has time complexity $O(t(n)$.

Equivalently, one would say that M has time complexity at most $c \cdot t(n)$, or that M has $O(t(n))$ running time. For example, the standard algorithm for deciding $A = \{0^k 1^k \mid k \geq 0\}$, which moves back and forth to the midpoint, is in TIME(n^2). If $\text{REGULAR}_{\text{TM}}$ denotes the set of regular languages, then $\text{REGULAR}_{\text{TM}} \subseteq \text{TIME}(n)$, because a DFA necessarily runs in the number of steps equal to the size of the string.

It's also possible to place $A \in \text{TIME}(n\log n)$. This is given by a Turing machine M which, on input w,

- If w isn't of the form 0^*1^*, reject. (This takes time $O(n)$.)
- Then, repeat the following until all bits of w are crossed out:
  - If the parity of 0s and 1s are different (i.e. the numbers mod 2), then reject.
  - Cross out every other 0 and every other 1.
- Then, accept.

Knowing the parity at every step allows one to know exactly how many 0s and 1s there are, akin to reading the bits in the binary description of the number. Then, crossing out every other 0 or 1 pushes it to the next-order bit.

Can one do better than $n\log n$? It can be proven that a single-tape Turing machine cannot decide A in time less than $O(n\log n)$.

Exercise 13.3 (Extra credit)**.** Suppose that $f(n) = o(n\log n)$ (i.e. is strictly bounded above).[9] Then, show that TIME($f(n)$) contains only regular languages!

Note that this doesn't imply the nonexistence of $n\log\log n$ algorithms; this is just single-tape Turing machines, so more interesting algorithms still exist. But this collapse of time complexity is very interesting.

Theorem 13.4. *$A = \{0^k 1^k \mid k \geq 0\}$ can be decided in time $O(n)$ with a two-tape Turing machine.*

Proof idea. Scan all 0s and copy them to the second tape. Then, scan all 1s, and for each 1 scanned, cross off a 0 from the second tape. Then, if the numbers disagree, then reject; else, accept. ⊠

[9] More precisely, $f(n) = o(g(n))$ iff $\lim_{n\to\infty} f(n)/g(n) = 0$.

Notice a difference with computability theory: multiple-tape Turing machines make no difference as to whether a language is decidable, but produce different results in complexity theory.

Theorem 13.5. *Let $t : \mathbb{N} \to \mathbb{N}$ satisfy $t(n) \geq n$ for all n. Then, every $t(n)$-time multi-tape Turing machine has an equivalent $O(t(n)^2)$-time single-tape Turing machine (that is, they decide the same language).*

This follows by tracing out the complexity of their proof of (decidability) equivalence: every step of the multitape machine corresponds to updating all of the previous data on the single tape, causing $T(n)^2$ steps.

Universal Turing machines are also efficient (i.e. take quadratic time).

Theorem 13.6. *There is a (single-tape) Turing machine U which takes as input (M, w), where M is a Turing machine and w is an input string, and accepts (M, w) in $O(|M|^2 t^2)$ steps iff M accepts w in t steps.*

Proof sketch. Once again, this follows by tracing the proof of the existence of the universal Turing machine for computability (i.e. the recognizer for A_{TM}): we had a two-tape Turing machine representing the inputs and the states, and so in t steps U updates $O(|M|t)$ steps to update both of these: $|M|$ is intuitively there because we need to update the configuration. ⊠

This paves the way to something called the Time Hierarchy Theorem, which intuits that with more time to compute, one can solve strictly more problems. As referenced by Exercise 13.3, there are gaps.

Theorem 13.7 (Time Hierarchy)**.** *For "reasonable" $f, g : \mathbb{N} \to \mathbb{N}$ where $f(n) = O(g(n)^{1/2})$, $\text{TIME}(f(n)) \subsetneq \text{TIME}(g(n))$.*

The exact notion of "reasonable" will be explained within the proof, which is easier than stating it beforehand.

Proof. Define a Turing machine N as follows: if $w \neq M10^k$ for some Turing machine M and integer k, reject. Then, let U be the universal Turing machine, and run it on input (M, w) for $f(|w|)$ steps, which takes $O(f(|w|)^2)$ time. Then, accept if M has not accepted w; otherwise, reject.

Note that the above argument depended on the following three assumptions:

(1) $g(n) \geq n$, so that the multi-tape Turing machine can be converted into a single-tape Turing machine.
(2) $f(|w|^2) \leq O(g(n))$.
(3) The value $f(n)$ can be efficiently computed, i.e. in $O(g(n))$ time. The number may be large, but there's a nice algorithm for computing it. Most functions anyone would care about satisfy this.

This is what is meant by "reasonable."

Now, we have seen that $L(N) \in \text{TIME}(g(n))$, and it remains to show it's not in $\text{TIME}(f(n))$. What happens when we feed N to itself? Specifically, suppose $N(N10^k)$ runs in time $O(f(n))$. Then, it outputs the opposite value to what it should if it halts, so it can't halt in time $f(n)$. ⊠

The conditions can be improved slightly. There is a much more difficult proof (e.g. in CS 254) of the following.

Theorem 13.8. *For "reasonable" f and g where $f(n) = O(g(n)/\log^2 g(n))$, $\text{TIME}(f(n)) \subsetneq \text{TIME}(g(n))$.*

Thus, there is an infinite hierarchy of decidable, but increasingly time-consuming problems. One important question is whether there exist natural problems (as opposed to the contrived one cooked up for the proof) that sit at every level of the time hierarchy. One example of a natural problem is 3-Sum, which given an $S \subseteq \mathbb{N}$, asks if there exist $a, b, c \in S$ such that $a + b = c$. This has an $O(n^2)$ solution that is popular for job interviews, but it's open as to whether this problem has a better solution (e.g. linear time). Yet this is a useful problem.

The notion of an efficient computation in complexity theory is a problem solvable in polynomial time:

$$\mathbf{P} = \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k).$$

This leads to a notion of intuition about efficient algorithms.

The Extended Church-Turing Thesis. *Everyone's intuitive notion of* efficient *algorithms corresponds to the formalism of polynomial-time Turing machines.*

This is much more controversial (or as another way of thinking about it, open) thesis: randomized algorithms might be efficient, but it's open as to whether they can be efficiently deterministically simulated. Additionally, $\text{TIME}(n^{1000000}) \subseteq \mathbf{P}$, but most people wouldn't call these efficient. Finally, do quantum algorithms that cannot be efficiently encoded as classical algorithms quality as efficient? These algorithms could undermine cryptography as we know it.

Definition 13.9. A nondeterminstic Turing machine is a Turing machine that can intuitively choose between several state transitions at each step, and accepts a string if any of these choices leads to an accept state. Just like with NFAs, the formal definition is very similar to that of a deterministic Turing machine, except that the transition function is again represented by a function δ from the states to sets of state–tape-direction triples, where each element of the set is one possible transition.

Then, one can define accepting computation histories in the same manner: where each configuration can yield the next one (even if it could also yield other ones), and they lead to acceptance. Then, the Turing machine accepts a string if such a computation history exists.

Finally, N has time complexity $T(n)$ if for all n, all inputs of length n, and all histories, N halts in $T(n)$ time. If one views the possible paths through these choices as a tree, then this says that the maximal depth of the tree is $T(n)$.

Definition 13.10. $\text{NTIME}(t(n))$ is the set of languages decided by an $O(t(n))$-time nondeterministic Turing machine.

Since determinism is a special case of nondeterminism, then $\text{TIME}(t(n)) \subseteq \text{NTIME}(t(n))$. But the reverse inclusion is open for most $t(n)$: it's known that $\text{TIME}(t(n)) \subsetneq \text{NTIME}(t(n))$, but in general this question is open.

A Boolean formula is a function on booleans using $\neg$, $\wedge$, $\vee$, and parentheses. A satisfying assignment for such a formula is an assignment to the variables that makes it true (when evaluated in the normal manner). For example, $\phi = a \wedge b \wedge c \wedge \neg d$ is satisfiable (take $a = b = c = 1$ and $d = 0$). But $\neg(x \vee y) \wedge x$ is not satisfiable. The set of satisfiable Boolean formulas is called SAT.

A restriction of this problem: a 3cnf-formula has a form like

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_2 \vee x_5) \wedge (x_3 \vee \neg x_2 \vee \neg x_1).$$

The inside groupings, separated by $\vee$ symbols, are called literals; the greater structures (triples of literals) are called clauses, and are separated by $\wedge$. Then, it is possible to reduce some formulas into 3cnf-form. 3SAT denotes the set of satisfiable 3cnf-formulas.

Theorem 13.11. $\text{3SAT} \in \text{NTIME}(n^2)$.

Proof. On input ϕ, first check if the formula is in 3cnf-form, which takes $O(n^2)$ time. Then, for each variable v in ϕ, nondeterministically substitute 0 or 1 in place of v, Then, evaluate the formula for these choices of the variables, and accept if it's true. ⊠

Thus, $\text{3SAT} \in \mathbf{NP}$, where

$$\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k).$$

Theorem 13.12. *$L \in \mathbf{NP}$ iff there is a constant k and polynomial-time Turing machine V such that*

$$L = \{x \mid \text{ there exists a } y \in \Sigma^* \text{ such that } |y| \leq |x|^k \text{ and } V(x,y) \text{ accepts}\}. \tag{13.13}$$

14. More on P versus NP and the Cook-Levin Theorem: 2/25/14

At this point, the words "problem" and "language" really mean the same thing: the set of solutions to a problem forms a language, and so on. More modern terminology tends to favor the former, though.

Recalling Theorem 13.12, contrast this with the fact that a language A is recognizable iff there exists a deciding Turing machine $V(x,y)$ such that $A = \{x \in \Sigma^* \mid \text{there exists a } y \in \Sigma^* \text{ such that } V(x,y) \text{ accepts}\}$. The change was that for **NP**, V is required to run in polynomial time (which also restricts the length of the input). In some sense, we're looking at **P** within **NP** just like recognizability within decidability; **P** and recognizability are problems for which verification is feasible, and decidability and **NP** are those problems for which solutions can be feasibly found. Much of complexity theory has been guided by this analogy.[10]

Proof sketch of Theorem 13.12. Suppose L can be written in the form (13.13); then, a nondeterministic algorithm to decide L is to nondeterministically guess a y and then run $V(x,y)$. Since this computation takes polynomial time, $L \in \mathbf{NP}$.

Conversely, if $L \in \mathbf{NP}$, then there exists a nondeterministic polynomial-time Turing machine that decides L. Then, define $V(x,y)$ to accept iff y encodes an accepting computation history of N on x. ⊠

[10] The analogy isn't perfect; the diagonalization proof that there are recognizable, but undecidable, languages doesn't map across the analogy.

Thus, we can say that $L \in \mathbf{NP}$ iff there are polynomial-length proofs for membership in L (these proofs come from computation histories). One can look at 3SAT and SAT, which are hugely important (on the order of multiple Turing awards given for progress made, even in practical areas).

Another way to think of **NP** is that it is easy to verify "yes" answers, but not necessarily "no" answers (just like recognizability). Thus, SAT $\in$ **NP**, because if $\pi \in$ SAT, there is a short proof given the correct variables to satisfy the expression, and verifying that proof can be done quickly. Proving that something is not satisfiable, though, is harder; it doesn't seem possible to do better than trying all 2^n variable combinations, which takes exponential time. So SAT $\in$ **NP**.

The Hamiltonian path problem is another example, coming from the very large class of problems of finding something small in something larger. It's very useful in lots of settings, e.g. printing circuit boards. The problem itself is to, given a graph, determine whether a Hamiltonian path (i.e. a path that traverses every node exactly once) exists in it.

Making this more formal, assume some reasonable encoding of graphs (e.g. the adjacency matrix); then, define HAMPATH $= \{(G,s,t) \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$.

Theorem 14.1. HAMPATH $\in$ **NP**.

Intuitively, given a graph and a purported Hamiltonian path, it is possible to verify that it actually is a Hamiltonian path in polynomial time.

Another graph-theoretic problem is the k-clique problem. A k-clique on a graph is a complete subgraph of k nodes (e.g. five friends on a social network who all know each other). This problem can be formalized as CLIQUE $= \{(G,k) \mid G$ is an undirected graph with a k-clique$\}$.

Theorem 14.2. CLIQUE $\in$ **NP**.

Proof. A k-clique in G is certainly a proof that $(G,k) \in$ CLIQUE, and given a subset S of k nodes from G, it's possible to efficiently check that all possible edges are present in G between the nodes of S. ⊠

Notice that there's no formalism of Turing machines in these proofs; it's perfectly fine to use these higher-level methods to prove things.

Once again: **P** is the set of problems that can be efficiently solved, and **NP** is the set of problems where solutions can be efficiently verified.

So, the question on everyone's tongues: is **P** = **NP**? Heuristically, this would mean that problem solving can be automated in some sense; every problem that can be efficiently verified can be efficiently solved. This is (as is unlikely to be a surprise) an open question. Clearly, at least **P** $\subseteq$ **NP**. But if you can get the other direction, it's a Millennium Prize problem, meaning a correct solution is worth $\$1,000,000$. It might even be possible to use it to check the existence of short proofs of the other open problems...

Notice that for finite automata, these are equal (i.e. verification and problem-solving are the same), and for Turing machines, recognzability and decidability are known to be different notions. So complexity theory lies somewhere between automata theory and computability theory.

The general academic consensus is that it's probably the case that **P** $\neq$ **NP**. This is because if **P** = **NP**, then:

- Mathematicians might be out of a job; while provability is in general undecidable, short proofs (no more than quadratic or cubic in the length of the theorem) are a question of complexity theory rather than computability theory. Verifying a short proof is a polynomial-time problem, so it's in **NP**, so if **P** = **NP**, then one could construct an algorithm to generate short proofs (assuming the constant multiplier isn't completely atrocious). Note that if the proof is sufficiently nonconstructive, it might not be possible to extract an algorithm.
- Cryptography as we know it would be impossible; specifically, a one-way function is a function which is easy to compute but hard to invert (i.e. given a y, it's hard to find a x such that $f(x) = y$, but given an x, it's easy to calculate $f(x)$). Clearly, this requires **P** $\neq$ **NP**.
- There are grander conjectures of implications to efficiently and globally optimizing every aspect of daily life, though these seem farther afield.

It's also possible that this question is undecidable, and people have asked this, but right now the techniques complexity theorists have just don't work for it. Maybe in 40 years someone should try again.

Polynomial-Time Reducibility.

Definition 14.3. $f : \Sigma^* \to \Sigma^*$ is a polynomial-time computable function if there is a polynomial-time Turing machine M such that for every input w, M halts with $f(w)$ on its tape.

This looks just like the previous definitions of reducibility, replaced with the condition on M.

Definition 14.4. A language A is polynomial-time reducible to another language B, denoted $A \leq_{\mathrm{P}} B$, if there is a polynomial-time computable $f : \Sigma^* \to \Sigma^*$ such that $w \in A$ iff $f(w) \in B$. Then, one says f is a polynomial-time reduction from A to B.

In the above case, there is a k such that $|f(w)| \leq |w|^k$.

Theorem 14.5. *If $A \leq_{\mathrm{P}} B$ and $B \leq_{\mathrm{P}} C$, then $A \leq_{\mathrm{P}} C$.*

The proof is a little more involved, but boils down to the fact that the composition of two polynomials is a polynomial.

Theorem 14.6. *If $A \leq_{\mathrm{P}} B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.*

Proof. let M_B be a polynomial-time Turing machine that decides B and f be a polynomial-time reduction from A to V. Then, build a polynomial-time decider for A that computes f on its input w, and then runs B on $f(w)$. ⊠

Theorem 14.7. *If $A \leq_{\mathrm{P}} B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.*

The proof is the same as the proof for Theorem 14.6, but with nondeterministic polynomial-time deciders.

One way to understand this class of languages better is to understand its maximal elements under $\leq_{\mathrm{P}}$.

Definition 14.8.

- If every $A \in \mathbf{NP}$ is polynomial-time reducible to B (i.e. $A \leq_{\mathrm{P}} B$), then B is called **NP**-hard.
- If $B \in \mathbf{NP}$ and B is **NP**-hard, then B is called **NP**-complete.

In computability theory, we saw that for every recognizable language L, $L \leq_{\mathrm{m}} A_{\mathrm{TM}}$; this can be said as A_{TM} is complete for all recognizable languages.

Note that if L is **NP**-complete and $L \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$, and if $L \notin \mathbf{P}$, then $\mathbf{P} \neq \mathbf{NP}$, of course. Thus, the whole problem boils down to these **NP**-complete problems, and makes this whole theory more compelling.

Suppose L is **NP**-complete and $\mathbf{P} \neq \mathbf{NP}$; then, L isn't decidable in time n^k for every k.

The following theorem seems specific, but is very useful for other problems in complexity theory or elsewhere, e.g. by reducing a general problem (e.g. bug finding) to a SAT problem, and then using a SAT solver to deal with it.

Theorem 14.9 (Cook-Levin)**.** SAT *and* 3SAT *are* **NP***-complete.*

Proof. First, 3SAT $\in \mathbf{NP}$, because a satisfying assignment is a "proof" that a solution exists.

Then, why is every language $A \in \mathbf{NP}$ polynomial-time reducible to 3SAT? For any $A \in \mathbf{NP}$, let N be a nondeterministic Turing machine deciding A in n^k time. The idea is to encode the entire tree of possible paths of execution of N as a 3cnf-formula with length polynomial in n, but the naïve way of doing this (encoding the entire tree) has exponential size.[11]

Definition 14.10.

- If $L(N) \in \mathrm{NTIME}(n^k)$, then a tableau for N on w is an $n^k \times n^k$ table whose rows are configurations of some possible computation history of N on w. Since one can use at most n^k space in n^k time, then having only n^k columns (each the next square of tape) is reasonable.
- A tableau is accepting if some row of the tableau has an accept state. In this case, all rows after that row are taken to preserve this accept state.
- Each of the $(n^k)^2$ entries in the tableau is called a cell; the cell in row i and column j is denoted cell$[i, j]$, and is the j^{th} symbol in the i^{th} configuration.

Then, the idea is to, given w, construct a 3cnf formula ϕ describing all of the logical constraints that every accepting tableau for N on w must satisfy; this ϕ will be satisfiable iff there is an accepting tableau for N on w. (There are lots of proof of this theorem, but this one seems easier to understand, and is in Sipser besides.)

[11] Notice that this strategy worked for automata: you could encode all possible paths of an NFA into a DFA.

Let $C = Q \cup \Gamma \cup \{\#\}$. Then, for a given tableau, for $1 \le i, j \le n^k$ and $s \in C$, let $x_{i,j,s}$ be the Boolean variable that cell$[i,j] = s$. These $x_{i,j,s}$ will be the variables for the ϕ that we wish to construct, i.e. cell$[i,j] = s$ iff $x_{i,j,s} = 1$.

ϕ will be given by four subformulas: $\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$, given as:

- ϕ_{cell} checks that for all i and j, there is exactly one $s \in C$ such that $x_{i,j,s} = 1$.
- ϕ_{start} checks that the first row of the table is the start configuration of N on input w.
- ϕ_{accept} checks that the last row of the table has an accept state.
- ϕ_{move} checks that every row is a configuration that legally follows from the previous configuration.

More explicitly,

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left(\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \ne t}} (\neg x_{i,j,s} \vee \neg x_{i,j,t}) \right) \right).$$

These aren't technically 3cnf, but can be easily converted into them (clearly, they're general satisfiability problems).

The last, hardest one is ϕ_{move}, which needs to show that every 2×3 "window" of cells in the tableau is legal, i.e. consistent with the transition functions of N. This is sufficient because in a single step, a Turing machine can only move a small distance (i.e. 1).

For example, if $\delta(q_1, a) = \{(q_1, b, R)\}$ and $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$, then

a	q_1	b
q_2	a	c

is a legal window.

The proof will be continued next lecture.

15. **NP**-Complete Problems: 2/27/14

> *"**NP**-complete problems are like the zombies of computer science. They're everywhere, and they're really hard to kill."*

...though we're still looking at the proof that 3SAT is **NP**-complete, let alone anything else. We needed to construct ϕ_{move}, by looking at every 2×3 window and checking that it's consistent with the transition function. Then, the key lemma is that if it's consistent locally, then it's consistent globally: if every window is legal and the top row is the start configuraton, then each row is a configuration that yields the next row.

This is proven by induction on the number rows: if one row doesn't yield the rest, then there must be some illegal window in the last row.

Finally, we can define ϕ_{move} as

$$\phi_{\text{move}} = \bigwedge_{\substack{1 \le i \le n^k - 1 \\ 1 \le j \le n^k - 2}} W(i,j),$$

where $W(i,j)$ is the formula that the window whose upper left corner is at position (i,j) is legal, i.e.

$$W(i,j) = \bigvee_{\text{legal windows } (a_1, \ldots, a_6)} (x_{i,j,a_1} \wedge \cdots \wedge x_{i+1,j+2,a_6}),$$

but this isn't in 3cnf form, so you have to take its complement, which turns the outside $\wedge$ into a $\vee$, which is OK.

It's also necessary to convert longer strings into shorter ones: $a_1 \vee a_2 \vee \cdots \vee a_t$ can be written in 3cnf by introducing extra variables z_i. Once this is done, there are $O(n^k)$ or $O(n^{2k})$ clauses in each of these formulas, so ϕ has $O(n^{2k})$ clauses and was generated efficiently from A. ⊠

An alternate proof defines a problem CIRCUIT-SAT, which is easier to reduce to SAT, but can also be shown to be **NP**-complete.

Is it possible that 3SAT can be solved in $O(n)$ time on a multitape Turing machine? It seems unlikely, because this would not just imply $\mathbf{P} = \mathbf{NP}$, but that there would be a "dream machine" that cranks out short proofs of theorems, quickly optimize a lot of things, and in general produce quality work from a formula for recognizing it.

Nonetheless, this is an open question!

There are thousands of **NP**-complete problems, within every topic of CS and even within the other sciences too, and if one were determined in either way, then all of them would. These are huge within theoretical computer science.

Corollary 15.1. SAT $\in$ **P** *iff* **P** = **NP**.

Given some problem $\Pi \in$ **NP**, one now has a simpler recipe to show that languages are **NP**-complete:

(1) Take a problem Σ known to be **NP**-hard (i.e. 3SAT).
(2) Construct a polynomial-time reduction $\Sigma \leq_{\mathrm{P}} \Pi$.

For example, we defined CLIQUE, the clique problem on graphs.

Theorem 15.2. CLIQUE *is* **NP**-*complete.*

Proof. This proof will show 3SAT $\leq_{\mathrm{P}}$ CLIQUE, i.e. efficiently transform a 3cnf formula into a graph with the specific clique properties.

Let k be the number of clauses of ϕ, and construct a graph G with k groups of 3 nodes each; intuitively, the i^{th} group of nodes corresponds to the i^{th} clause C_i of ϕ, and join nodes v and v' by an edge if they aren't in the same group and they don't correspond to opposite literals (i.e. x_i and $\neg x_i$).

Given an SAT assignment A of ϕ, then for every clause C there is at least one literal set true by A (since the entire thing needs to come out as true). For every clause C, let v_C be a vertex in G corresponding to a literal of C satisfied by A.

Then, there is a k-clique $S = \{v_C : C \in \phi\}$, because if $(v_C, v_{C'}) \notin E$, then v_C and v'_C label inconsistent literals, but that means that A can't satisfy both of them, which is a contradiction.

Hence, S is a k-clique, so $(G,k) \in$ CLIQUE.

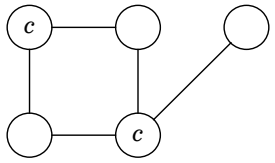
In the reverse direction, suppose $(G,k) \in$ CLIQUE was given by ϕ, so the goal is to construct a satisfying assignment A for ϕ. Let S be an m-clique of G; then, S contains one variable from each group (since there are no edges between two things of the same group).

Now, for each variable x of ϕ, make an assignment A where $x = 1$ iff there is a vertex $v \in S$ with label x. Thus, A satisfies at least one literal in the i^{th} clause of ϕ, so it is a satisfying assignment. ⊠

Related to the clique problem is the independent-set problem IS, which, given a graph $G = (V,E)$ and an integer k, is there an independent set of size k in the graph, i.e. a subgraph with no vertices of size at least k? Clearly, CLIQUE $\leq_{\mathrm{P}}$ IS, by the algorithm that takes in a graph and produces its complement, so IS is **NP**-complete.

Definition 15.3. A vertex cover of a graph G is a set of nodes C such that every edge borders some node in C.

For example, in the following graph, the nodes marked c form a vertex cover:



These have applications to networks of all kinds. Clearly, the entire graph is its own vertex cover, but we want a more minimal example. Then, VERTEX-COVER is the problem that, given a graph G and an integer k, determines whether there is a vertex cover of size $|G| - k$. This is a minimization problem, in the sense that the other graph problems here have been maximization problems.

Then, IS $\leq_{\mathrm{P}}$ VERTEX-COVER:

Claim. For every graph $G = (V,E)$ and subset $S \subset V$, S is an independent set iff $V \setminus S$ is a vertex cover.

Proof. S is an independent set iff for all $u, v \in S$, $(u,v) \notin E$, so if $(u,v) \in E$, then one of u or v is not in S. Thus, $V \setminus S$ is a vertex cover of size $|G| - k$. ⊠

Now, one can create the polynomial-time reduction $(G,k) \mapsto (G, |G| - k)$, so VERTEX-COVER is **NP**-complete.

But there are also **NP**-complete problems out of number theory, which look completely different and illustrate how deep the notion of **NP**-completeness is. The subset-sum problem is given a set $S = \{a_1, \ldots, a_n\} \subset \mathbb{N}$ and a $t \in \mathbb{N}$. Then, the question is whether there exists an $S' \subseteq S$ such that

$$t = \sum_{i \in S'} a_i = t.$$

Then, SUBSET-SUM is the problem as to whether such a S' exists, given S and t.

The following result was proven in CS 161, using dynamic programming.

Theorem 15.4. *There is an algorithmm polynomial in n and t, that solves* SUBSET-SUM.

Yet since t can be specified in $\log t$ bits, this isn't polynomial in the length of the input!

It will be possible to reduce VERTEX-COVER $\leq_P$ SUBSET-SUM by reducing a graph to a specific set of numbers. Given (G,k), let $E = \{e_0,\dots,e_{m-1}\}$ and $V = \{1,\dots,n\}$. Then, the subset-sum instance (S,t) will have $|S| = m+n$, given by two clases of numbers:

- The "edge numbers:" if $e_j \in E$, then add $b_j = 4^j$ in S.
- The "vertex numbers:" for each $i \in V$, put

$$a_i = 4^m + \sum_{j : i \in e_j} 4^j$$

in S.

Then, let

$$t = k \cdot 4^m + \sum_{j=0}^{m-1} (2 \cdot 4^j).$$

Claim. If $(G,k) \in$ VERTEX-COVER, then $(S,t) \in$ SUBSET-SUM.

This is because if $C \subseteq V$ is a vertex cover with k vertices, then let

$$S' = \{a_i : i \in C\} \cup \{b_j : |e_j \cap C| = 1\} \subseteq S.$$

Then, the sum of the numbers in S' will be equal to t. The way to intuit this is to think of these numbers being vectors in "base 4,"[12] i.e. $4^j = b_j \sim (0,\dots,0,1,0\dots,0)$, and a_i is also a binary vector with a 1 in the j^{th} slot wherever vertex i is contained in edge j (and a 1 in the top slot). Then, $t = (k,2,2,\dots,2)$ (so not quite in base 4, but nearly so).

Then, for each component of the vector, the sum over a vertex cover C is exactly 2 for the edges, because for any edge, either e_j is hit once by C, in which the value is 2, or it's hit twice by C, in which case you still get 2. Then, throwing in the vertices implies that the subset-sum solution exists.

In the opposite direction, if $(S,t) \in$ SUBSET-SUM, I want to show that $(G,k) \in$ VERTEX-COVER. But suppose $C \subseteq V$ and $F \subseteq$ satisfy

$$\sum_{i \in C} a_i + \sum_{e_j \in F} b_j = t;$$

then, C is a vertex cover of size k. Once again, this follows because the numbers can be thought of as vertices in base 4, so C must have size k, because each vertex contributes a 1 to the first component, but the first component of t is k.

There's a whole wing of people on the 4^{th} floor of Gates who study the subset-sum problem (cryptographers), but, oddly enough, they don't really care about solving vertex-cover, even though the problems are the same from this viewpoint.

If G is a graph and s and t are nodes of G, we can ask the shortest (longest) path problem: is there a simple path of length less than k (resp. more than k) from s to t? Using Dijkstra's algorithm, the shortest-path problem is in **P**, but the longest-path problem ends up in **NP**.

16. **NP**-Complete Problems, Part II: 3/4/14

Related to the subset-sum problem is something in economics called the knapsack problem, in which one is given a cost budget C, a profit target P, and a set $S = \{(p_1,c_1),\dots,(p_n,c_n)\}$ of pairs of positive integers.[13] Then, the goal is to choose items out of S such that their costs (first item of the pair) are no more than C, but their profits (second entry) are higher than P. In some sense, what's the greatest utility I can get out of stuffing heavy items into a knapsack?

Theorem 16.1. KNAPSACK *is* **NP**-*complete.*

[12] Or maybe in $\mathbb{Z}/4\mathbb{Z}$?

[13] Sometimes, one can take fractional amounts of each item, in which case this becomes a linear programming problem.

Proof. This will be pretty easy after reducing from SUBSET-SUM. First, KNAPSACK $\in$ **NP**, because it has a pretty reasonable polynomial-time verifier (in the same was as everything else we've done these few days).

Then, it will be possible to reduce SUBSET-SUM $\leq_P$ KNAPSACK as follows: given an instance $(S = \{a_1,\dots,a_n\},t)$ of SUBSET-SUM, create an instance of KNAPSACK by setting $(p_i,c_i) \coloneqq (a_i,a_i)$ for all i, and then $S' = \{(p_1,c_1),\dots,(p,c_n)\}$. Finally, let $P = C = t$. ⊠

Another somewhat economic problem is the partition problem: given a set $S = \{a_1,\dots,a_n\} \subset \mathbb{N}$, is there an $S' \subseteq S$ such that

$$\sum_{i\in S'} a_i = \sum_{a\in S\setminus S'} a_i?$$

In other words, PARTITION is the problem as to whether there is a way to partition S "fairly" into two parts.

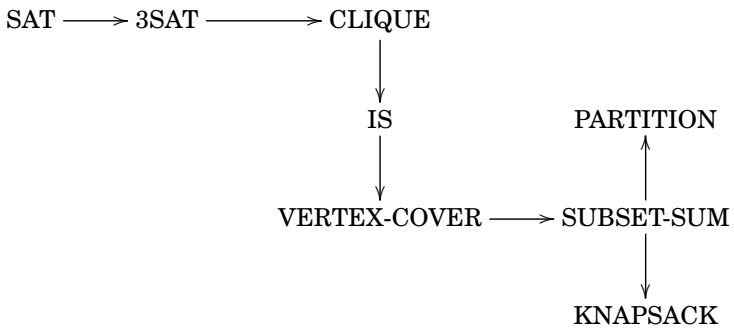
Theorem 16.2. PARTITION *is* **NP**-*complete.*

Proof. This seems intuitively easier than SUBSET-SUM, since there's no fixed target to reach. However, a reduction still exists. Clearly, there's a polynomial-time verifier for this problem, but we can reduce SUBSET-SUM $\leq_P$ PARTITION: given a set $S = \{a_1,\dots,a_n\} \subset \mathbb{N}$ and a target t, output $T = \{a_1,\dots,a_n,2A-t,A+t\}$, where $A = \sum_{i\in S} a_i$.

Now, we want to show that there's a solution to $(S,t) \in$ SUBSET-SUM iff there's a partition of T. The sum of all of the numbers in T is $4A$, so T has a partition iff there exists a $T' \subseteq T$ summing to $2A$. Thus, intersecting it with $\{a_1,\dots,a_n\}$ yields a solution to the subset-sum problem, because exactly one of $\{2A-t.A+t\} \in T'$ (or else there would be too much there). Thus, what remains, i.e. $T' \setminus \{2A-t\}$ is a subset of S if that's in T', and if $A+t \in T'$, then instead take $T \setminus T'$, which has the same sum (since T' is a solution of T) and contains only $2A-t$ and elements of S.

In the reverse direction, if $(S,t) \in$ SUBSET-SUM, then one can take a subset S' that sums to t, and add $A-t$ to it to get a partition. ⊠

Right now, we have the following chain of reductions, where $A \to B$ indicates that $A \leq_P B$.



Another problem is the bin-packing problem: given a set $S = \{a_1,\dots,a_n\}$ of positive integers, a bin capacity B, and a target $K \in \mathbb{N}$, can one partition S into K subsets such that each subset sums to at most B? One can imagine lots of applications of this to shippiing and to optimization. Sometimes, this is formally denoted BINPACKING.

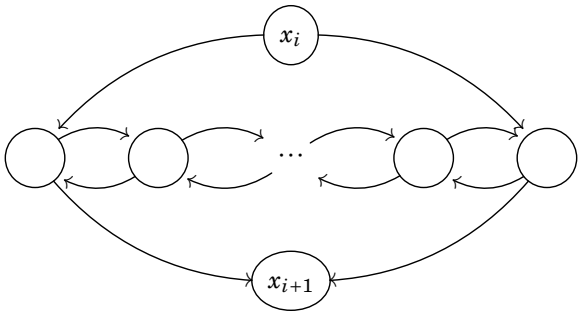
Claim. PARTITION $\leq_P$ BINPACKING, and in particular the latter is **NP**-complete.

Proof. Clearly, bin packing is in **NP**, since it can be verified in polynomial time. But given an instance $S = \{a_1,\dots,a_n\}$ of the partition problem, it can be worded as a bin packing problem by creating two bins. Then, a solution for one exists iff there is a solution for the other. ⊠

Recall that last lecture we looked at the shortest-path and longest=path problems; the former was shown to be in **P** by Dijkstra's algorithm. This can be reduced from HAMPATH (since if there is a Hamiltonian path, it's the longest), which we already saw in Theorem 14.1 was in **NP**.

Theorem 16.3. HAMPATH *is* **NP**-*complete.*

Proof. The proof will reduce 3SAT $\leq_P$ HAMPATH. Specifically, let $F = C_1 \wedge \cdots \wedge C_n$ be a 3cnf formula, with each $c_i = x_i \vee x_j \vee x_k$. Then, we can construct graphs G_i that look like this:



Specifically, there are $3m+1$ nodes in the second row, corresponding to allowable paths. Then, let G be the union of these graphs (joining the node x_i from G_{i-1} to x_i in G_i).

Add some *more* edges to G: specifically, add a node for each of $c_1, \ldots, c_m$, and in the second row, consider each pair of arrows from left to right to correspond to $C_1, \ldots, C_m$, connecting (say) nodes $c_{j,i}$ and $c_{j,f}$ for each j. Then, if clause j contains x_j, add arrows from $c_{j,i} \to C_j \to c_{j,f}$, and if it contains $\neg x_j$, add arrows $c_{j,f} \to C_j \to c_{f,i}$; thus, in some sense, truth represents heading to the right along the second row, and falsity to the left.

Then, F is satisfiable iff G has a Hamiltonian path; now that the graph is set up, this is some amount of tracing through the logic. ⊠

Eew... incredibly, this this is the nicest reduction the professor knows of, from any **NP**-complete problem. Maybe it's possible to do better. It at least has the property that it's agnostic to the variables. Don't think in terms of "let's suppose I have a solution."

Thus, the longest-path problem is also **NP**-hard.

coNP and Friends. There's another class co**NP** = $\{L \mid \neg L \in \mathbf{NP}\}$. This also looks like nondeterminism, but with the following twist: a co-nondeterministic machine has multiple paths of computation, but the machine accepts if all paths reach an accept state, and rejects if at least one path rejects.

For the same reason that $\mathbf{P} \subseteq \mathbf{NP}$, $\mathbf{P} \subseteq$ co**NP**: every deterministic machine can be seen as a co-nondeterministic one. However, **NP** and co**NP** are a little more interesting: it's believed they aren't equal, but this is a very open problem. Of course, $\mathbf{P} =$ co**P**, since a deterministic decider for A can be turned into one for $\neg A$ by flipping the accept and reject states. One can draw Venn diagrams relating these sets, though there are so many aspects of these diagrams that are open problems.

Definition 16.4. A language is co**NP**-complete if:

(1) $B \in$ co**NP**, and
(2) For every $A \in$ co**NP**, $A \leq_P B$ (i.e. B is co**NP**-hard).

Let's look at $\neg$SAT (sometimes called UNSAT), the set of Boolean formulas ϕ such that no variable assignment satisfies ϕ.

Theorem 16.5. $\neg$SAT *is* co**NP**-*complete.*

Proof. Since SAT $\in$ **NP**, then $\neg$SAT $\in$ co**NP**. To show that it's co**NP**-hard, use the Cook-Levin theorem.

Let $A \in$ co**NP**. Then, on input w, transform w into a formula ϕ using the reducer for $\neg A \leq_P$ SAT (guaranteed by Cook-Levin); then, $w \in A$ iff $\phi \in \neg$SAT. ⊠

We also have the set of tautologies: TAUT = $\{\phi \mid \neg\phi \in \neg\text{SAT}\}$ (i.e. things satisfied by every assignment of variables). TAUT is co**NP**-complete, by a similar line of reasoning to the above theorem.

Another worthwhile question is whether $\mathbf{P} = \mathbf{NP} \cap$ co**NP**. In computability theory, the analogous statement (i.e. a language is decidable iff it is both recognizable and co-recognizable) is true. The idea is that if this is true, then short proofs of positive and negative answers imply it's possible to solve these kinds of problems quickly.

Of course, this is an open problem, and one direction would imply $\mathbf{P} = \mathbf{NP}$. However, there are interesting problems in $\mathbf{NP} \cap$ co**NP**, e.g. prime factorization! One can formalize this into a decision problem by asking for the set of integers (m, n) (each greater than 1) such that there is a prime factor p of m greater than n. But if this factoring problem is in **P**, then most public-key cryptography would be breakable, e.g. RSA.

However, we can say that factoring is in $\mathbf{NP} \cap \text{co}\mathbf{NP}$. Defining PRIMES to be the set of prime numbers, it's rather nontrivial and interesting to show the following.

Theorem 16.6 (Pratt)**.** PRIMES $\in \mathbf{NP} \cap \text{co}\mathbf{NP}$.

In fact, PRIMES $\in \mathbf{P}$, which was open for a while, and shown relatively recently.

Theorem 16.7. FACTORING $\in \mathbf{NP} \cap \text{co}\mathbf{NP}$.

Proof. The idea is that it's easy to check the prime factorization of a number: given a factorization $p_1^{e_1} \cdots p_k^{e_k}$ of m, one can efficiently check that each p_i is prime, and then whether they multiply together to make m. ⊠

Note that several of these problems, including prime factorization, rely on quantum computers, which aren't strictly deterministic, nor quite nondeterministic. This class of problems is called **BQP**, and there are lots of open questions about it too.

17. Polytime and Oracles, Space Complexity: 3/6/14

"Let's talk about something a little more down-to-Earth, space complexity."

Last time, we saw that TAUT $\in \text{co}\mathbf{NP}$, so if a formula *isn't* a tautology (a tautology is a formula that is always satisfiable for every assignment), then there is a short proof (i.e. an assignment that doesn't work). However, whether the complement is true (do there exist short proofs that something's a tautology?) is open, i.e. whether $\text{co}\mathbf{NP} = \mathbf{NP}$. Thus, TAUT is co**NP**-complete, as we saw last time. You can play this game with lots of co**NP** problems.

We saw yet another open question, as to whether $\mathbf{P} = \mathbf{NP} \cap \text{co}\mathbf{NP}$. If this is true, then there are unfortunate implications for the RSA algorithm for cryptography.

NP-complete problems lead to co**NP**-problems, e.g. ¬SAT, ¬HAMPATH. But nobody knows if there are any problems that are ($\mathbf{NP} \cap \text{co}\mathbf{NP}$)-complete! Intuitively, **P**, **NP**, co**NP**, and so on can all be defined with specific machine models. For every machine, we can get an encoding, thanks to the framework of computability theory. But $\mathbf{NP} \cap \text{co}\mathbf{NP}$ is just the intersection of two complexity classes, so there's no easy description that we know of for these sorts of problems.

Oracle Turing Machines. Recall from computability theory we have the notion of oracle Turing machines, which can magically answer queries to some problem; then, one asks what else one could solve with this power. The key to using them here is that such oracle Turing machines can ask their oracles in polynomial time (in fact, in one step).

Definition 17.1. For some language B, the complexity class $\mathbf{P}^B$ is the set of languages that can be decided by a polynomial-time Turing machine with an oracle for B.

For example, we might want to know what can be solved in polynomial time with a SAT solver, which would be $\mathbf{P}^{\text{SAT}}$, and $\mathbf{P}^{\mathbf{NP}}$ is the class of language decidable by some polynomial-time Turing machine, with an oracle for some $B \in \mathbf{NP}$. But of course, since SAT is **NP**-complete, then $\mathbf{P}^{\text{SAT}} = \mathbf{P}^{\mathbf{NP}}$: using the Cook-Levin theorem, for any $B \in \mathbf{NP}$, an oracle for B can be simulated in polynomial time by an oracle for SAT. Maybe the polynomial is slightly larger, but it's still polynomial.

If $B \in \mathbf{P}$, then $\mathbf{P}^B = \mathbf{P}$, because an oracle for B can be straight-up simulated in polynomial time.

More interesting question: is $\mathbf{NP} \subseteq \mathbf{P}^{\mathbf{NP}}$? This is also true: for any $L \in \mathbf{NP}$, one can create an oracle for L, and then just ask it. We also have $\text{co}\mathbf{NP} \subseteq \mathbf{P}^{\mathbf{NP}}$, for the same reason: the oracle decides problems in **NP**, and therefore also decides co**NP**: for any $L \in \text{co}\mathbf{NP}$, one can create an oracle for $\neg L$, and then flip its answer. This corresponds to the notion that the complements of recognizable languages can be decided by an oracle for the halting problem, even though the halting problem isn't co-recognizable.

In general, this implies that $\mathbf{P}^{\mathbf{NP}} = \mathbf{P}^{\text{co}\mathbf{NP}}$, even though we don't know if $\mathbf{NP} = \text{co}\mathbf{NP}$! This complexity class has the very natural definition of the class of problems decidable in polynomial time if SAT solvers work, and this is something which causes it to appear in a reasonable number of applications.[14]

A typical problem in $\mathbf{P}^{\mathbf{NP}}$ looks like FIRST-SAT, the set of pairs (ϕ, i) such that $\phi \in$ SAT and the i^{th} bit of the lexicographically first[15] satisfiable assignment of ϕ is 1.

[14] One might ask whether $\mathbf{P}^{\mathbf{NP}} = \mathbf{NP} \cup \text{co}\mathbf{NP}$. This is an open problem, but the answer is believed to be no.

[15] Viewing the assignment as a binary string, with one bit for each variable, then this is the dictionary order $0^n < 0^{n-1}1 < \cdots < 1^n$.

One can compute the lexicographically first satisfying assignment using only polynomially many calls to a SAT oracle, in a manner similar to this week's homework (i.e. reducing the problem, one variable at a time). But notice that this problem isn't obviously in either **NP** or co**NP**; it isn't thought to be, but it might.

In the same manner, one can define $\mathbf{NP}^B$ and $\text{co}\mathbf{NP}^B$, which are what you would expect (languages decidable by a nondeterministic or co-nondeterministic Turing machine with an oracle for B in polynomial time, respectively). Now we can start asking some interesting-looking questions: is $\mathbf{NP} = \mathbf{NP}^{\mathbf{NP}}$? How about $\text{co}\mathbf{NP}^{\mathbf{NP}} = \mathbf{NP}^{\mathbf{NP}}$? These are both open questions, even if one assumes that $\mathbf{P} \neq \mathbf{NP}$, but the general belief is that both of these are untrue. In fact, there are heuristic arguments that something is unlikely by reducing it to one of these equalities!

In the same way as before, $\mathbf{NP}^{\mathbf{NP}} = \mathbf{NP}^{\text{co}\mathbf{NP}}$, because an oracle for an $L \in \mathbf{NP}$ can be turned into an oracle for $\neg L \in \text{co}\mathbf{NP}$ and vice versa in polynomial time.

There are very natural problems in some of these seemingly abstract complexity classes. For example, here's one in $\text{co}\mathbf{NP}^{\mathbf{NP}}$.

Definition 17.2.

- Two Boolean formulas ϕ and ψ under the same number of variables are equivalent if they have the same value on every assignment to the variables, e.g. x and $x \vee x$.
- A Boolean formula is minimal if every equivalent formula has at least as many symbols as it (i.e. $\wedge$, $\vee$, and $\neg$).

This is a very useful problem in lots of fields. Let EQUIV $= \{(\phi, \psi) \mid \phi \text{ and } \psi \text{ are equivalent}\}$, and MIN-FORMULA denote the set of minimal formulas.

Theorem 17.3. MIN-FORMULA $\in \text{co}\mathbf{NP}^{\mathbf{NP}}$.

Proof. Notice that EQUIV $\in \text{co}\mathbf{NP}$, because if two formulas aren't equivalent, then there's a short proof of that. In fact, it's co**NP**-hard, because it can be reduced to TAUT (is ϕ equivalent to the trivial tautology?). Thus, EQUIV can be decided by an oracle for SAT.

Here's a $\text{co}\mathbf{NP}^{\text{EQUIV}}$ Turing machine that efficiently decides MIN-FORMULA, given an input formula ϕ: for every formula ψ such that $|\psi| < |\phi|$, determine whether $(\phi, \psi) \in$ EQUIV. If so, reject; then, after testing all of these, accept. ⊠

However, we believe this problem isn't in **NP** or co**NP**, so it (probably) lies within interesting intersections of complexity classes.

Note that the time hierarchy still exists when one adds in an oracle. This suggests that oracles can be used to learn about the relationship between **P** and **NP**, but it's not as useful as one might hope:

Theorem 17.4.

(1) *There is an oracle B for which* $\mathbf{P}^B = \mathbf{NP}^B$.
(2) *There is an oracle A for which* $\mathbf{P}^A \neq \mathbf{NP}^A$.

(1) is pretty easy: just take B to be an oracle for the halting problem. But the second part is much trickier; see Sipser, §9.2.

Space Complexity. Space complexity seems less natural than time complexity,[16] though it has an interesting theory and useful applications, even touching on some aspects of game theory.

Definition 17.5.

- Let M be a deterministic Turing machine that halts on all inputs. Then, the space complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the index of the furthest tape cell reached by M on any input of length n.
- Let M be a *non*deterministic Turing machine that halts on all inputs. Then, the space complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the index of the furthest tape cell reached by M on any input of length n and any computation path through M.

Basically, what's the largest amount of space that this algorithm can take?

Definition 17.6.

[16] Though I suppose if you're traveling at relativistic speeds they're the same thing.

- SPACE($s(n)$) is the set of languages L decided by a Turing machine with $O(s(n))$ space complexity.
- NSPACE($s(n)$) is the set of languages L decided by a nondeterministic Turing machine with $O(s(n))$ space complexity.

The idea is that SPACE($s(n)$) formalizes the class of problems solvable by computers with bounded memory. This sounds very much like problems we face in the real world.

One important question is how time and space relate to each other in computing. Going from time to space isn't too bad, but SPACE(n^2) problems could take much longer than n^2 time to run, because, in some sense, you can reuse space, but not time.

Even SPACE(n) is very powerful; for example, 3SAT $\in$ SPACE(n), because one can try every single assignment within linear space, and thus decide 3SAT (albeit in exponential time).

If M is a halting, non-deterministic Turing machine which on an input x uses S space on all paths, then the number of time steps is at most the total number of configurations, because if a configuration repeats, then the machine loops. But since a configuration can be written in $O(s(n))$ bits, then this takes time at most $2^{O(S)}$. In other words,

$$\text{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(2^{c \cdot s(n)}).$$

In fact, it's also the case that

$$\text{NSPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(2^{c \cdot s(n)}).$$

This can be shown by viewing it as a graph problem, where the nodes are the configurations of a given Turing machine M on its given input x, and an edge from $u \to v$ implies that the configuration u can yield configuration v in one step. Now, this is just a graph reachability problem of size $2^{O(S)}$, which can be solved in $O(n^2)$ time.

Definition 17.7. The notions of efficient computation and verification are

$$\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k) \qquad \text{and} \qquad \text{NPSPACE} = \bigcup_{k \in \mathbb{N}} \text{NSPACE}(n^k).$$

Notice that $\mathbf{P} \subseteq \text{PSPACE}$, and that $\mathbf{NP} \subseteq \text{PSPACE}$ as well, because we saw that 3SAT $\in$ PSPACE. In fact, every complexity class we've talked about so far is contained in PSPACE, even the exotic ones like co$\mathbf{NP}^{\mathbf{NP}}$.

Things do get weirder: the complexity class RLOGSPACE of problems that can be solved by a randomized algorithm with high probability in logarithmic space is believed to be equal to SPACE($\log n$), though it's an open problem. This is more generally the case: we have randomized classes R$\mathbf{P}$ versus $\mathbf{P}$ and so on, and they seem like they should be equal, but we aren't certain.

18. Space Complexity, Savitch's Theorem, and PSPACE: 3/11/14

"Every time I hear about it, the thesis gets shorter."

Recall that we were talking about the space complexity of a Turing machine M, the function $f(n)$ that on n returns the largest amount of space (i.e. tape) M takes when running on all inputs of length n. We also saw that computations done in $S(n)$ space take at most $2^{O(S(n))}$ time, because of a bound on the number of configurations, and that in a slightly more complex proof, this is also true of nondeterministic computations.

It's true that $\mathbf{NP}^{\mathbf{NP}} \subseteq \text{PSPACE}$, because an oracle for $\mathbf{NP}$ can be simulated in polynomial space (even if it takes a lot of time); instead of nondeterministically guessing something, loop through all possible choices (more concretely, this could involve reducing via SAT or something, and testing all possible assignments, which is deterministic, uses exponential time, and uses polynomial space). Thus, this takes polynomial space, and the original (non-oracle) part can be simulated in the same way. In general, reasoning about these can be more tricky; for example, $\mathbf{P} \neq \text{SPACE}(O(n))$, but this proof is somewhat involved.

Since a deterministic Turing machine M that halts on all inputs using at most $s(n)$ space has an upper bound on running time of $s(n)|Q||\Gamma|^{s(n)} = 2^{O(s(n))}$, then it makes sense to define

$$\text{EXPTIME} = \bigcup_{k \in \mathbb{N}} O\left(2^{n^k}\right).$$

Then, it's not too bad to show (sort of like we did above) that PSPACE $\subseteq$ EXPTIME and NPSPACE $\subseteq$ EXPTIME. Thus, we know that $\mathbf{P} \subseteq \mathbf{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$, and by the Time Hierarchy theorem, $\mathbf{P} \neq \text{EXPTIME}$, since $\mathbf{P} \subsetneq \text{TIME}(2^n)$. But which of the three inclusions is proper? That's also open. Or is more than one true? Proving any of these would win a Turing award...

Similar to the Time Hierarchy theorem, there's a space hierarchy, but it's much more powerful; we saw that $\text{TIME}(n \log\log n) = \text{TIME}(O(n))$ (which is the class of regular languages), but $\text{SPACE}(n\log\log n) \supsetneq \text{SPACE}(O(n))$.

Theorem 18.1 (Space Hierarchy)**.** *For* any *unbounded function* $a : \mathbb{N} \to \mathbb{N}$ *and "reasonable" functions* $s : \mathbb{N} \to \mathbb{N}$, $\text{SPACE}(s(n)) \subsetneq \text{SPACE}(a(n)s(n))$.

Proof idea. This is basically the same proof as of Theorem 13.7, using diagonalization and the fact that the halting problem is unrecognizable.

One really surprising result shows that PSPACE = NPSPACE, among other things.

Theorem 18.2 (Savitch)**.** *For functions* $s(n)$ *where* $s(n) \geq n$, $\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^2)$.

Proof. Given a nondeterministic Turing machine N with space complexity $s(n)$, one might construct a deterministic machine M that tries every possible branch of $s(n)$. But this is too much data to keep track of, as there are $2^{2^{s(n)}}$ branches to keep track of, so one needs at least that much space. The intuition here is to look at breadth-first and depth-first searching algorithms. So we'll need a different algorithm.

Given configurations C_1 and C_2 of an $s(n)$-space (nondeterministic) Turing machine N, and a number $t = 2^k$, the goal is to determine whether N can get from C_1 to C_2 within t steps. This procedure is called $\text{SIM}(C_1, C_2, t)$.

- If $t = 1$, then accept if $C_1 = C_2$, or C_1 yields C_2 within one step, which takes space $O(s(n))$.
- More interesting is the inductive step, which guesses the midpoint of the path between C_1 and C_2. Specifically, for all configurations C_m using $s(n)$ space, run $\text{SIM}(C_1, C_m, t/2)$ and $\text{SIM}(C_m, C_2, t/2)$, and accept if they both accept.
- If no such C_m leads to existence, then reject.

This algorithm has $O(\log t)$ levels of recursion, and each level of recursion takes $O(s(n))$ space. But we only need to keep track of the one path through everything, so failed guesses for the midpoint can be overwritten. Thus, the total space needed is $O(s(n)\log t)$ space.

Now, let $d > 0$ be the number such that the maximum running time of $N(w)$ on any computation path is at most $2^{ds(|w|)}$, and let C_{acc} be the unique accepting configuration on inputs of length $|w|$.[17]. Now, simulate N by M, which on input w runs $\text{SIM}(q_0 w, C_{\text{acc}}, 2^{ds(|w|)})$ and accepts iff it does. This can be calculated to take $O(s(n)^2)$ space, since $\log t = O(S(n)^2)$. ⊠

Another surprising result is the existence of PSPACE-complete problems. These offer general strategies for solving problems in PSPACE, and intuitively are a lot broader than **NP**-complete problems and such.

Definition 18.3. A language B is PSPACE-complete if:

(1) $B \in \text{PSPACE}$, and
(2) For any $A \in \text{PSPACE}$, $A \leq_{\text{P}} B$.

Definition 18.4. A fully quantified Boolean formula is a Boolean formula where every variable is quantified (i.e. there is an $\forall$ or an $\exists$ for each variable) These formulas are either true or false, and are much more general than SAT or $\neg$SAT e.g. $\varphi \in \text{SAT}$ iff $\exists x_1 \cdots \exists x_n[\varphi]$, and $\varphi \in \neg\text{SAT}$ iff $\forall x_1 \cdots \forall x_n[\neg\varphi]$ is true.

For example, some more such fully quantified Boolean formulas are $\exists x \exists y[x \vee \neg y]$, $\forall x[x \vee \neg x]$, and $\forall x[x]$.
Let TQBF denote the set of true fully quantified Boolean formulas.

Theorem 18.5 (Meyer-Stockmeyer)**.** TQBF *is* PSPACE*-complete.*

Proof. That TQBF is in PSPACE isn't yet obvious, so consider the following algorithm QBF-SOLVER:

(1) If ϕ has no quantifiers, then it is an expression with only constants, so it can be straightforwardly evaluated. Accept iff ϕ evaluates to 1.
(2) If $\phi = \exists x \psi$ (it's existentially quantified), recursively call this algorithm on ψ twice, first with x set to 0, and second with x set to 1. Accept iff *at least* one of these calls accepts.
(3) If $\phi = \forall x \psi$ (it's universally quantified), then recursively call this algorithm twice, first with x set to 0, and second with x set to 1. Then, accept iff *both* of these calls accept.

[17] This hasn't really been discussed yet, but can be done without loss of generality by, for example, clearing the tape before accepting.

What's the space complexity of this algorithm? In the same way as for the proof of Savitch's theorem, it's possible to reuse space, because the algorithm only needs to store one path through the tree of recursion at once, so if $|\phi| = K$, then this takes $O(K \cdot n) \leq O(K^2)$ (where there are n variables).

Now, why is TQBF PSPACE-hard? This will use tableaux in the same way as the Cook-Levin theorem, but will need to be modified in order to work properly: the algorithm used above creates a tableau that has width n^k and height $2^{O(n^k)}$, which s *a priori* too large. However, a cleverly designed $\phi \in$ TQBF will do the trick, i.e. be true iff M accepts w.

Let $s(n) = n^k$ and $b \geq 0$ be an integer. Using two collections of $s(n)$ Boolean variables denoted c and d representing two configurations and an integer $t \geq 0$, construct the quantified Boolean formula $\phi_{c,d,t}$ which is true iff M starting in configuration c reaches configuration d in at most t steps. Then, set $\phi = \phi_{\text{start},c_{\text{acc}},h}$, where c_{acc} is the unique accept state and h is the exponential time bound.

Similarly to Savitch's theorem, $\phi_{c,d,t}$ claims that there is a c' such that $\phi_{c,c',t/2}$ and $\phi_{c',d,t/2}$ are both true. In the base case, $\phi_{c,d,1}$ means that $c = d$ or d follows from c in a single step of M; the former is expressed by saying each of the $bs(n)$ variables representing c are equal to the corresponding ones in d. To show that d follows from c in a single step of M, use 2×3 windows in the same way as in the Cook-Levin theorem, and write a cnf formula.

Now, we can construct $\phi_{c,d,t}$ recursively for $t > 1$:

$$\phi_{c,d,t} = \exists m[\phi_{c,m,t/2} \wedge \phi_{c,m,t/2}].$$

Here, $M = \exists x_1 \exists x_2 \cdots \exists x_S$ where $S = bn^k$. But this is too long; every level of recursion cuts t in half, but doubles the size of the formula! So the length will be $O(t)$, which is exponential.

So far, the proof has only used existential quantifiers, so maybe we should introduce a $\forall$ in there somewhere. Instead, modify the formula to be

$$\phi_{c,d,t} = \exists m \forall x, y[((x,y) = (c,m) \vee (x,y) = (m,d)) \implies \phi_{x,y,t/2}].$$

This folds the two recursive copies of $\phi_{c,d,t}$ into one. Then, $\text{size}(s,t) = O(S(n)) + \text{size}(s,t/2)$, so the total size is bounded above by $O(s(n)\log t)$ (the number of calls is logarithmic in t). This is much shorter, only quadratic in the space bound! ⊠

19. PSPACE-completeness and Randomized Complexity: 3/13/14

There's a general intuition that **NP** corresponds to finding optimal strategies in hard solitare games, and PSPACE corresponds to finding optimal strategies in two-player games. Accordingly, a recent paper shows that Candy Crush is **NP**-hard,[18] and for formalizations of many popular two-player games, it's PSPACE-complete to decide who has a winning strategy on a game board.

For example, TQBF can be interpreted as a game between two players, E and A. Given a fully quantified Boolean formula, such as $\forall x \exists y[(x \vee y) \wedge (\neg x \vee \neg y)]$, E and A alternate choosing values for variables, where E chooses for existential quantifiers and A for universal quantifiers. Then, E wins if the result is an assignment, and A wins if it's not valid.

Theorem 19.1. *If* FG *is the set of formulas in which E has a winning strategy, then* FG *is* PSPACE*-complete.*

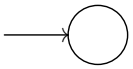
Proof. FG = TQBF, because if there's an assignment to all of the existentially quantified variables such that all choices of universally quantified variables work, then that formula is in TQBF, and this choice is a winning strategy for E. ⊠

Another game is the geography game, where two players take turns naming cities from anywhere in the world, but each city chosen must begin with the same letter the previous city ended with, and cities can't be repeated. For example, one might go from Austin to Newark to Kalamazoo to Opelika and so on. The game ends when one player can't think of another choice of words. This looks like a nice Hamiltonian path problem.

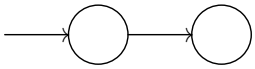
The generalized geography problem considers a directed graph and a start state a; Player 1 moves from here to some other state, and they Player 2 makes another move on the graph, and so on, subject to the constraint

[18] Walsh, Toby. "Candy Crush is **NP**-hard." http://arxiv.org/pdf/1403.1911v1.pdf. 11 March 2014.

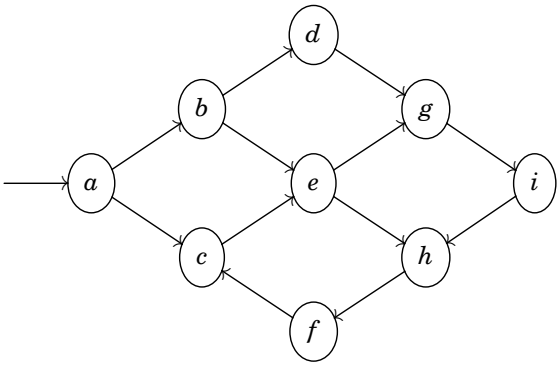
that they cannot revisit previously visited nodes. Then, game ends when one player has no moves, and thus loses. For example, in the following graph, Player 2 has a winning strategy of not doing anything.



Here's a graph where Player 1 wins:



Now things can get more interesting:



Who has a winning strategy here?

Theorem 19.2. GG*, the set of generalized-geography games in which Player 1 has a winning strategy, is* PSPACE*-complete.*

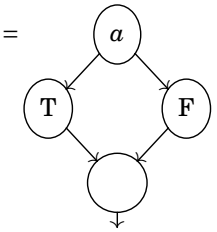
Proof. First, why is it even in PSPACE? Here's an algorithm M accepting a graph G and a start node a.

- If node a has no outgoing edges, then reject.
- For all nodes $a_1,\dots,a_k$ pointed to by a, recursively call $M(G\setminus\{a\},a_i)$.
- If all of the calls accept, then reject; otherwise, accept (since a turn flips the roles of Players 1 and 2; they're symmetric except for who goes first).

If there are n nodes, one needs $O(n^2)$ space to store each stack frame, but there are at most $O(n)$ stack frames, and thus this takes polynomial space. This is interesting, given that writing down a complete winning strategy could take exponential space (since it would involve lots of branching, based on whatever player 2 does).

To show that GGis PSPACE-hard, we will show that FG $\leq_P$ GG by transforming a formula ϕ into a pointed graph (i.e. graph with start node) (G,a) such that E has a winning strategy in ϕ iff player 1 does in (G,a).

The idea is that, given some formula, without loss of generality write it as a bunch of cnf clauses separated by $\wedge$. Then, build a graph that starts like this:

$G(x_1) =$ 

corresponding to x_1 (the left side corresponds to setting it to true, and the right side to false). Then, define $G(x_2),\dots,G(x_k)$ similarly, and connect $G(x_i)\to G(x_{i+1})$. Then, from $G(x_k)$ lead out to a node c, which is connected to nodes $c_1,\dots,c_n$ corresponding to each clause. Then, each clause has arrows to each of its literals (e.g. if $C_1 = x_1\vee\neg x_2\vee x_3$, there are arrows to nodes (C_1,x_1), $(C_1,\neg x_2)$, and (C_1,x_3), so different clauses have different nodes). Then, from each node (C_i,x_i), add a node to the T node of $G(x_i)$, and for $(C_i,\neg x_i)$, add an arrow to the F node of $G(x_i)$. Tracing through this spaghetti graph,[19] it can be shown that E can satisfy the formula iff Player 1 has a winning strategy on this graph. ⊠

[19] Not a technical term.

Here's an interesting question: is chess PSPACE-complete? The crux is, of course, how you define chess as a class of problems. Conventional 8×8 chess has a winning strategy that can be determined in cosntant time... for some incredibly painful constant. But generalized versions of chess, checkers, and Go can be shown to be PSPACE-hard.

Randomized Complexity. Randomized algorithms are useful to understand, since they happen a lot in practice and are useful, but so little is known about them theoretically.

Definition 19.3. A probabilistic Turing machine is a nondeterministic Turing machine where each nondeterministic step is called a "coin flip," and is set up such that each nondeterministic step has only two legal next moves (heads or tails, in some sense).

Then, this Turing machine accepts if the path of coin flips leads it to an accept state... which means the same probabilistic Turing machine might accept and reject the same input on different trials, and the question is how probable these are, not whether they are black-and-white true or false. The probability of reaching a state after k coin flips is 2^{-k}.

These are very useful objects to study: often, there is a simpler probabilistic randomized algorithm for a problem than a deterministic algorithm for that problem, and often it's more efficient. However, it's completely open as to whether randomized algorithms can be used to solve problems much faster than deterministic ones in general.

Definition 19.4. The probability that a randomized Turing machine machine M accepts a string w is

$$\Pr[M \text{ accepts } w] = \sum_{H \in \mathcal{H}} \Pr[M(w) \text{ has history } H],$$

where $\mathcal{H}$ is the set of accepting computation histories of M on w.

Theorem 19.5. *$A \in$ **NP** iff there exists a nondeterministic polynomial-time Turing machine M such that $\Pr[M \text{ accepts } w] > 0$ for all $w \in A$ and is zero for $w \notin A$.*

Definition 19.6. A probabilistic Turing machine M recognizes a language A with error ε if for all $w \in A$, $\Pr[M \text{ accepts } w] \geq 1 - \varepsilon$ and for $w \notin A$, $\Pr[M \text{ doesn't accept } w] \geq 1 - \varepsilon$.

Lemma 19.7 (Error Reduction)**.** *Let $0 < \varepsilon < 1/2$ and $f : \mathbb{N} \to \mathbb{N}$ be a function (often taken to be a polynomial). If M_1 is a randomized Turing machine that has error ε and runs in $t(n)$ time, the there is an equivalent randomized Turing machine M_2 such that $M - 2$ has error $2^{-f(n)}$ and runs in $O(f(n)t(n))$ time.*

Proof. Good old Monte Carlo simulation: M_2 runs M_1 for $11f(n)$ times, and then chooses the majority response. ⊠

Definition 19.8. BPP denotes the complexity class of languages L that can be recognized by a probabilistic polynomial-time Turing machine with error at most 1/3.

By the lemma, though, any number between 0 and 1/2 would work, and the same class BPP would be obtained.

Definition 19.9. An arithmetic formula is a formula in some finite number of variables, and symbols $+$, $-$, and $*$, so that one can make polynomials. Then, ZERO-POLY denotes the set of formulas that are identically zero, e.g. $(x-y)\cdot(x+y) - x\cdot x - y\cdot y - 2\cdot x\cdot y$, which can be notationally simplified, of course, into $(x+y)^2 - x^2 - y^2 - 2xy$. There's a rich history of polynomial identities in mathematics (somehow involving Gauss, to nobody's surprise), and these are also useful for verification of sufficiently restricted programs.

This can be thought of as an arithmetic analogue of SAT.

Let $p \in \mathbb{Z}[x]$ (i.e. a polynomial over $\mathbb{Z}$ in one variable), but such that we can't see the coefficients, and can only query it. However, we can test whether $p = 0$ by querying $d + 1$ values, because this is enough to uniquely determine such a polynomial. So how does this generalize to $\mathbb{Z}[x_1, \ldots, x_n]$? If a $p(x_1, \ldots, x_n) \in \mathbb{Z}[x_1, \ldots, x_n]$ is a product of m single-variable polynomials, each of which has at most t terms, then expanding out everything would take t^m time, which is icky.

But choosing large random values to evaluate p at is a probabilistic algorithm that works.

Theorem 19.10 (Schwartz-Zippel)**.** *Let $p(x_1,\dots,x_n) \in \mathbb{Z}[x_1,\dots,x_n]$ be nonzero, where each variable has degree at most d. Then, if $F \subset \mathbb{Z}$ is a finite set and $a_1,\dots,a_n \in F$ are selected randomly (in the uniform distribution), then*

$$\Pr[p(a_1,\dots,a_n)=0] \le \frac{dn}{|F|}.$$

Proof. Proceed by induction on n; we saw already the idea for $n=1$ (though it's missing a step or two).

In general, assume it's true for $n-1$. Then, we can "factor out" the last variable, writing $p = p_0 + x_n p_1 + \cdots + x_n^d p_d$, where $p_0,\dots,p_d$ are polynomials in $x_1,\dots,x_{n-1}$.[20] Then, if $p(a_1,\dots,a_n)=0$, then there are two possibilities: either $p_i(a_1,\dots,a_{n-1}=0$ for all i, or a_n is a root of the overall polynomial, which eventually leads to the inductive bound. ⊠

Theorem 19.11. ZERO-POLY $\in$ BPP.

Proof idea. Let p be an arithmetic formula, and suppose $|p|=n$. Then, p has $k \le n$ variables, and the degree of each variable is $O(n)$. Then, for all $i=1,\dots,k$, choose r_i randomly from $\{1,\dots,n^2\}$, and of $p(r_1,\dots,r_k)=0$, then claim that $p=0$; otherwise, report that it's nonzero. Then, by Theorem 19.10, this works! ⊠

Note that we have no idea if ZERO-POLY $\in \mathbf{P}$ or not; if we could prove it, then there would be new, nicer lower bounds.

Next question: is BPP $\subseteq \mathbf{NP}$? It seems like it would be reasonable, because probabilistic and nondeterministic Turing machines are cousins, but this is an open question! We want BPP to model efficient computation, but really have no idea. It's true that BPP $\subseteq \mathbf{NP}^{\mathbf{NP}}$, which is a story for CS 254. Another open question is whether $\mathbf{NP} \subseteq$ BPP, or whether BPP $\subseteq \mathbf{P}^{\mathbf{NP}}$.

But we do know that BPP $\subseteq$ PSPACE, because a polynomial-time probabilistic Turing machine can be simulated by trying all possible coin flips in the same space.

We also don't know whether BPP $\stackrel{?}{=}$ EXPTIME... though if this is ever proven, it will probably be negative; the question itself is totally ridiculous. Basically, we *think* $\mathbf{P}$ = BPP, but can't rule out that BPP = EXPTIME!

Here one might be tempted to draw a Venn diagram reflecting inclusions between different classes of languages, $\mathbf{P}$, $\mathbf{NP}$, co$\mathbf{NP}$, $\mathbf{NP}^{\mathbf{NP}}$, co$\mathbf{NP}^{\mathbf{NP}}$, PSPACE, EXPTIME, PSPACE, NEXP (*non*deterministic exponential time, which might be equal yet to BPP), BPP, and so on. But there are so few known strict inclusions that this seems hopeless.

[20] In algebra, this is just using the fact that $\mathbb{Z}[x_1,\dots,x_n] = \mathbb{Z}[x_1,\dots,x_{n-1}][x_n]$.