CS 154 NOTES

MOOR XU
NOTES FROM A COURSE BY LUCA TREVISAN AND RYAN WILLIAMS

Abstract. These notes were taken during CS 154 (Automata and Complexity Theory) taught by Luca Trevisan and Ryan Williams in Winter 2012 at Stanford University. They were live-TEXed during lectures in `vim` and compiled using `latexmk`. Each lecture gets its own section. The notes are not edited afterward, so there may be typos; please email corrections to `moorxu@stanford.edu`.

1. 1/10

The website for the class is `theory.stanford.edu/~trevisan/cs154`

In the engineering fields, computer science is the field with the most developed idea of what is impossible. There is a mathematical theory for this. In the 1930s, there were some interesting foundational problems in mathematical logic, and impossibility results were obtained. There became a pretty good model for what we now consider a computer. These all came before there was the technology to build them. This is sort of like civil engineering, like the Babylonians. The mathematical theory was ready before the technology was, so the theory was well-developed, and we also knew about what wasn't possible.

Impossibility results and lower bounds are useful in computer science in three ways.

(1) It allows us to know what sort of problems to try. We want to solve tractable problems, and impossible problems can be solved using heuristics and approximations.
(2) Your result might be overly general: a special case might be hard, but there might be approachable examples. What is the actual problem you want to solve in practice? Even that is not always completely clear. For example, there is most than one way to formulate the clustering problem.
(3) You need to have the right definitions, for example in security. We need to model what is an elementary operation, and what types of problems are feasible to solve in a certain amount of time. This tells us whether security systems can be broken.

This is a big challenge. If you study algorithms, you can always do an experimental study. But if you want to prove a result about factoring an n-digit integer, there are infinitely many possible algorithm, and you cannot just experiment. Or you might want to reason about a security algorithm. We can only do this by proving mathematical statements about algorithms. So first we need a mathematical formalization of what is an algorithm. We have three approaches for this.

1. *Automata and regular expressions.* These are mathematical models of algorithms that run in linear time, make one pass through the input, and use a constant number of bits of memory. The disadvantage of this model is that it is very restrictive. However, we can understand it completely. Since we know that they are linear time, the measure of

complexity is memory usage. First, we want to know if a problem can be solved by this type of method, and then we want to know how much memory is necessary and sufficient.

2. *Computability.* This was where computer science was born in the 1930s. Here, we allow any algorithm, with any running time, as long as it returns an answer. There are well-defined problems for which no algorithm can exist, regardless of running time. One such problem is the *halting problem*.

Example 1.1 (Halting problem)**.** Given a problem P and an input x, does P terminate with input x.

There is no general method to solve this problem. The proof is simple, by contradiction. Suppose that such a program exists, and use it to determine whether itself terminates when given itself as input.

After some work on this, we'll be able to look at the following:

a. *Undecidability.*
b. *In every formalization of mathematics, there are true statements that do not have proofs.* In the early 1900s, mathematicians wanted to find a system where any true statement can be proved; this was shown to be impossible in the early 1930s. This can be done simply from undecidability. The basic idea is that this would give us a method to solve the halting problem, by considering statements of the form "this program with this input terminates". But we can't solve the halting problem.
c. *Kolmogorov complexity.* This gives a measure of how random a string is, and puts limits on its compressibility. We will be able to find a probability distribution on mathematical statements, and we can produce statements that are true with high probability without actually having proofs.

This approach to analyzing algorithms seems excessively general, while the previous approach was excessively restrictive. This leads to the last part of the course.

3. *Computational complexity theory.* This studies the existence of "efficient" algorithms for problems.

Definition 1.2. An algorithm is *efficient* if it takes time at most a polynomial function of the input length.

Remark. Observe that n^{100} would be infeasible for most inputs, while $(1.001)^n$ would be tractable for even moderately large inputs. But normal algorithms that we use do not have running times like these.

In the theory of automata, we will be able to prove everything. In computability, we can prove a lot. In computational complexity theory, which is closest to reality, there are a lot of open problems.

We'll see a result known as the *time hierarchy theorem*.

Theorem 1.3 (Time hierarchy theorem)**.** *There are problems solvable in time $O(n^3)$ but not in time $O(n^2)$. Or there are problems solvable in time $O(2^n)$ but not $O(2^{n/2})$. There is a general result.*

Unfortunately, these problems are all very artificial, but we don't actually have strict bounds for real problems.

Then we will look at *NP-completeness*. For many problems, such as the scheduling problem, we can solve them in finite time through a brute-force search. But this is usually

intractable. We want to know if there is a linear or quadratic time algorithm. We would like to prove rigorously that no efficient algorithm exists, but instead, we have results that reduce problems to other problems. Once we have a network of problem reductions, if we can find an efficient algorithm or prove a lower bound for one problem, that result would also apply to these other problems. We don't know what the answer is for any problem, but we know that the answer is the same for this large class of problems.

In this theory, P stands for all problems solvable in polynomial time, and NP stands for all problems involving searching for a solution with a given property.

Both P = NP and P $\neq$ NP lead to many consequences. First, suppose that P = NP. Then there is an algorithm that given an integer n and a property of n-bit strings checkable in time $t(n)$, finds a string with the property in time $(t(n))^{O(1)}$ or determines that none exists. But then given a mathematical problem, such as the Riemann hypothesis, we could probably search for proofs of length up to a million pages. Or we could solve any optimization problem. If we can name a property, then we could make such a property appear in about the same time. But from real life, we know that it is easier to laugh at jokes than to create them, and it is easier to check if homework is correct than to do the homework.

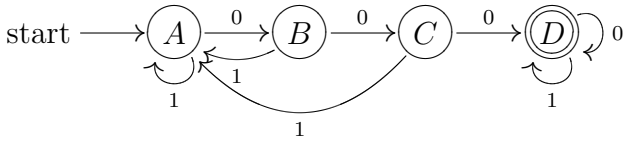
We will also look at the idea of *severely memory bounded* algorithms. Given an n-bit input, we wish to use $O(\log n)$ bits of memory. Effectively, these use no data structures. For example, is a graph strongly connected, or are two vertices of a graph connected? This gives an analog for the theory of NP-completeness.

There is also an application to cryptography, and the idea of *zero knowledge proofs*.

We will begin by recalling the definition of *automata*, which is something that takes its input from a string and has finitely many bits of memory.

First, let's do this graphically.

Example 1.4.



We denote end states where answers are returned by a double circle. This machine checks if there are three zeros in a row. This has states $Q = \{A, B, C, D\}$, and possible inputs $\Sigma = \{0, 1\}$. The starting state is $q_0 = A$. The final state is D, where the machine outputs `yes`. It also has operations $\delta(A, 0) = B$, $\delta(A, 1) = A$, $\delta(B, 0) = C$, ...

In order to define a machine, we need to describe

- the set of possible states Q
- the set of possible inputs Σ, or the *alphabet*
- one-step operation $\delta : Q \times \Sigma \to Q$
- the starting state q_0
- the set of states F that output `yes`.

Definition 1.5. If M is an automaton we define $L(M)$ to be the *language* of M:

$$L(M) = \{x : x \text{ is a string of symbols of } \Sigma \text{ such that } M \text{ outputs } \texttt{YES} \text{ on input } x\}.$$

In general, a language is a set of strings.

In our example above, we would have $L(M) = \{000, 0000, 0001, \dots\}$.

Definition 1.6. A language is *regular* if it is the language of some automaton.

We just showed that the language of all strings with three zeros in a row is regular. As we will see soon, not all languages are regular.

2. 1/12

Today's lecture will be about automata and regular expressions. Last time, we gave the definition of a deterministic finite automaton (DFA). Recall that defining a automaton requires defining the states Q, alphabet Σ, operations $\delta : Q \times \Sigma \to Q$, initial states $q_0 \in Q$, and final states $F \subseteq Q$ corresponding to `yes` answers.

Automata are descriptions of linear time, constant memory algorithms. In contrast, regular expressions describe what a set of strings is like, but does not describe the algorithm. There is an equivalence between automata and regular expressions. The transition between specifications and algorithms is explicitly doable.

Example 2.1. This is an example of a regular expression.

$$(0+1)^*000(0+1)^*$$

Here, the alphabet is $\Sigma = \{0, 1\}$.

Definition 2.2. The regular expression 0 corresponds to the language $\{0\}$, and the regular expression 1 corresponds to the language $\{1\}$.

Definition 2.3. If E_1 and E_2 corresponds to the language L_1 and L_2 respectively, then $E_1 + E_2$ corresponds to the language $L_1 \cup L_2$.

Definition 2.4. If E corresponds to language L, then E^* corresponds to language $L^* = \{x : x = x_1 \cdots x_k, x_i \in L, k \geq 0\}$.

Definition 2.5. $E_1 \cdot E_2$ corresponds to language $L_1 \cdot L_2 = \{x : y \cdot z, y \in L_1, z \in L_2\}$.

Example 2.6. Here are some examples.

regular expression	language it represents
ε	$\{\varepsilon\}$
0	$\{0\}$
1	$\{1\}$
$0+1$	$\{0,1\}$
$(0+1)^*$	$\{\varepsilon, 0, 1, 00, 01, \dots\}$
00	$\{00\}$
000	$\{000\}$
$(0+1)^*000(0+1)^*$	$\{x000y : x \text{ and } y \text{ are arbitrary binary strings}\}$.

We will discuss non-deterministic finite automata (NFA) as well. It seems that NFAs are more powerful than DFAs, but it turns out that for any NFA we can find an equivalent DFA, so they are equivalent notions. But it is much easier to produce an NFA that recognizes a regular expression. Going from NFAs to regular expressions will require that we consider the notion of a GNFA.

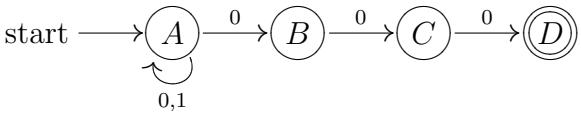
$$\text{DFA} \rightleftarrows \text{NFA} \rightleftarrows \text{regular expressions}$$

Definition 2.7. A *non-deterministic finite automaton* requires defining

- a set of states Q
- an alphabet Σ
- state transitions $\delta : Q \times \Sigma \to 2^Q$
- an initial state $q_0 \in Q$
- final states $F \subseteq Q$ corresponding to `yes` answer.

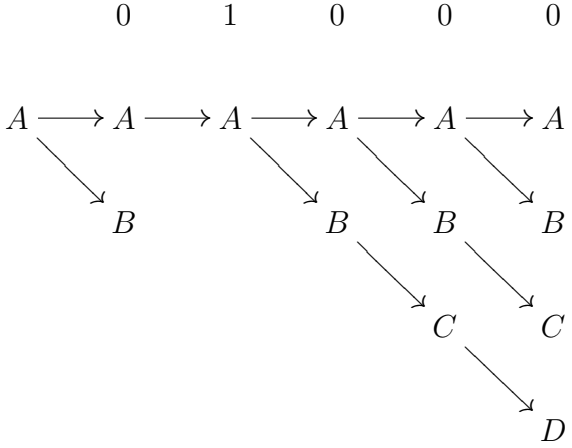
What's different about NFAs are the state transitions: δ returns a subset of Q.

Example 2.8. Suppose L is the language of binary strings that end in 000. The only way to do this with a DFA is to keep a queue that has the last three bits that have been seen. It is impossible to do this using a DFA with fewer than eight states. But there is a much simpler NFA for this problem.



Notice that there is some ambiguity at the first node for how to apply the state transition functions. In fact, the machine effectively chooses one of the possibilities at random.

For example, given the input 01000, what does the machine do? It starts in state A, and goes to either states A or B. At the next step, however, there is no transition from B, so that calculation will die, and the machine will be at A. At the next step, the machine might be in A or B, and after the next step, the machine might be in A or B or C. By the end, the machine could be in four states: A, B, C, or D. This diagram represents the possible state transitions.

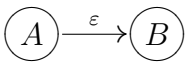


In this example, $\delta(B, 1) = \emptyset$.

Definition 2.9. An NFA *accepts* an input if among the possible states it can reach at the end of the computation, there is a state in F. It *rejects* an input if all reachable states are not in F.

The point is that there are no false positives. One way to think about this is that every time the machine forks the calculation, it makes a copy of itself.

Another feature of NFAs is that some times we see a transition labeled by ε, such as



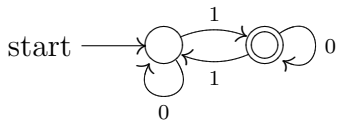
This means that the NFA can move from state A to state B without reading anything from the input. We will show how this is helpful by showing how to convert regular expressions to NFAs.

Theorem 2.10. *Given a regular expression E, we can construct an NFA M such that $L(M)$ is the same as the language described by E.*

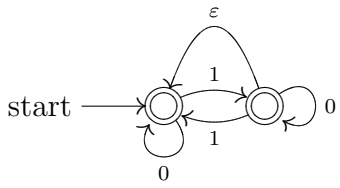
Proof. We will describe an algorithm to do this.

Suppose that we know that the language for E is given by some NFA M. Now, we want to modify M to an NFA that recognizes E^*. This is given as follows. The start state should be accepting, since the empty string is ok. Now, for every state that was accepting in M, add ε-transitions going back to the start state. Then we would be accepting any sequence of any number of strings in E.

To see how things might go wrong, consider

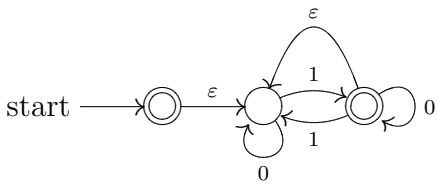


This is an automaton that accepts strings with an odd number of 1s. The star of this would accept the empty string and strings with at least one 1. Applying the description above, we get the automaton



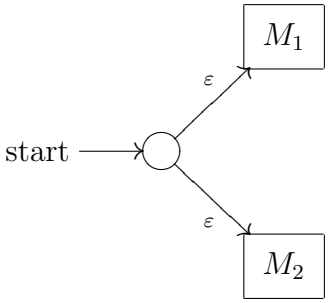
but this accepts 0, so something went wrong. The fact that we made the initial state accepting changed the automaton too much.

The way to fix this is to create a new state that wasn't there before, and making that an accepting initial state. From there, we use an ε-transition to go to what was the initial state. The empty string is still accepting, and we've only added the empty string to the language, and we are not introducing any spurious accepting conditions. In the example above, the correct NFA would be:

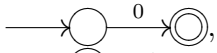
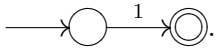


This shows that if we already have an automaton for some expression, we can also build the automaton for the star of that expression.

Next, we discuss unions. If we have machines M_1 and M_2 equivalent to expressions E_1 and E_2 respectively, we want a machine equivalent to the union $E_1 + E_2$. To do this, use



Now, to make $E_1 \cdot E_2$, for each accepting state of M_1, send an ε-transition to the start state of M_2. Then make the only accepting states those of M_2.

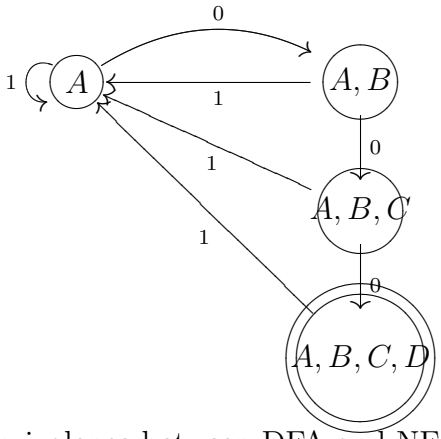
Finally, in the base cases, we have ε corresponding to  , and 0 corresponding to  , and 1 corresponding to  . There's also $\emptyset$ corresponding to  , where everything dies.

Note that $\emptyset + 1 = 1$ and $\emptyset \cdot 1 = \emptyset$. Also, $\varepsilon \cdot 1 = 1$. □

We've shown that from an regular expression, we can get an equivalent NFA. To get that regular expressions are equivalent to DFAs, we need to show that for any NFA there is an equivalent DFA.

Recall the NFA and the tree of transitions in Example 2.8. In this example, we see that the tree becomes rather large. If we want to simulate anything by keeping track of the tree, then after n states and using a machine with k states, we end up with k^n possibilities, which is much too large to efficiently do. In fact, we don't need to keep track of the entire tree. It suffices to know what states are reachable at any step. Now, this is a linear time simulation that makes one pass through the input and only requires a constant amount of memory, which means that this can be simulated by a DFA. Here, we need to store subsets of possible states of the machine.

This yields a DFA whose nodes are subsets of the nodes of the NFA. Then for each state and for each bit, we will look at what are the possible sets that can be reached (ignoring all nodes that cannot actually be reached). For example, for the NFA in Example 2.8, we have



This completely shows the equivalence between DFA and NFA. It remains to go from NFAs to regular expressions, which we will do next time through considering GNFAs.

Definition 2.11. A GNFA is an NFA with transitions labeled by regular expressions.

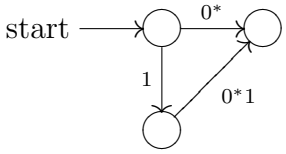
Clearly, every NFA is a generalized NFA.

3. 1/17

Today we will finish the proof of equivalence of DFAs, NFAs, and regular expressions. We then turn to the type of question we are most interested in this class: What are the limitations of this approach, and can we obtain some impossibility results? We will have two results of this sort: the Pumping Lemma and the Myhill-Narode Theorem. Next time, we will discuss the state limitation of DFAs.

We start with the equivalence between DFAs, NFAs, and regular expressions. Recall from last time that it remains to construct regular expressions from NFAs. We will do this through considering GNFAs, which are like NFAs but with transitions labeled by regular expressions.

Example 3.1.

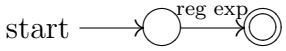


Clearly, any NFA is also a GNFA. The proof of the equivalence requires showing that each GNFA is equivalent to an NFA.

Definition 3.2. A GNFA has transitions labeled by regular expressions, with two restrictions:

- start state has no incoming transitions
- only one state in F, no outgoing transitions

Now, if a GNFA has only two states, then it must have a start state and a final state, and there are only be one transition. The GNFA then looks like

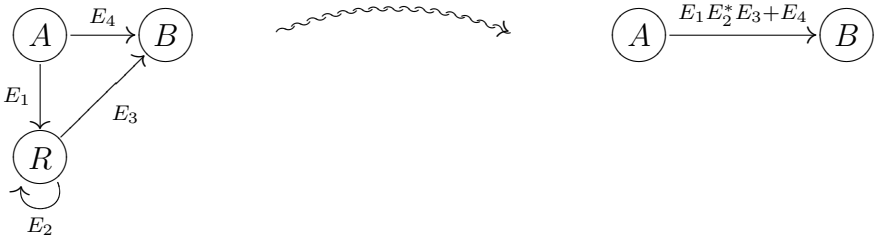


The point here is that a GNFA with two states is a regular expression.

Lemma 3.3. *For every k-state GNFA with $k \geq 3$, there is an equivalent GNFA with $k-1$ states.*

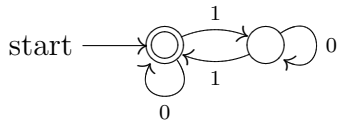
Applying this lemma repeatedly, we can reduce any GNFA to a GNFA with two states, which is just a regular expression.

Proof. Pick one state other than the start state and the accept state. Remove it, and fix the rest of the automaton so the language it accepts does not change. Here is how to fix the automaton.



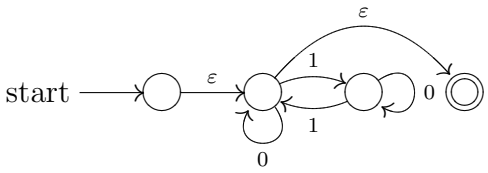
□

Example 3.4. Consider binary strings with an even number of ones. This is a two state machine:

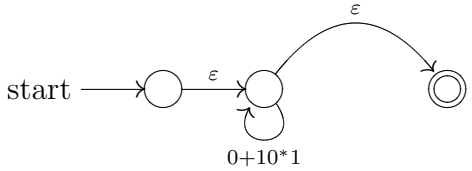


The regular expression is $0^*(0^*10^*10^*)^*$. We can get this by thinking about it, but what is the mechanical way of doing this?

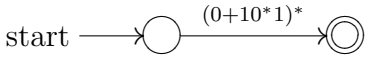
First, we add a new start state and a new final state.



Now we start to remove a state:



Finally, removing one more state gives



This gives us a nice regular expression for this automaton.

Now, we will proceed to impossibility results.

Example 3.5. Consider the language Maj of binary strings with strictly more ones than zeros. Here, $0, 1 \notin \text{Maj}$ and $1, 011, 10111, 10011 \in \text{Maj}$.

We want to show that this language is not regular, and as with many impossibility results, we will use contradiction.

Suppose Maj was regular, and let M be a DFA for Maj. Call k the number of states of this machine. Consider the states that M reaches after reading ε, 0, 00, 000, 0000, ..., up to strings of k zeros. Equivalently, feed the string of k zeros to the machine, and see what states it walks through. In either case, we used $k+1$ strings in a machine with k states, so there must be a state with two strings. This means that there are lengths i and j, such that strings of zeros of length i and j both reach state q.

We can now think of the automaton as having forgotten the difference between i and j zeros; the two situations are indistinguishable. Now, what would the machine do given the inputs $0 \dots 01 \dots 1$ with i zeros and j ones, or j zeros and j ones? The first string is in the language, while the second string is not. However, our hypothetical machine would do precisely the same thing given these inputs; after the zeros, it would be in state q, and then it would process ones without having any memory of what it did before.

This means that either the machine will accept both inputs or reject both inputs; in both cases, it is incorrect. Therefore, the machine can't exist at all, and this language is not regular.

Example 3.6. As a related example, suppose we are interested in a language with more occurrences of 01 than 10. This seems like the same type of problem, needing to use a counter. But this problem can be solved by a regular expression.

Lemma 3.7 (Pumping Lemma)**.** *For any regular language L, there is a P (pumping length) such that for every string $s \in L$ with length $|s| \geq P$, we can write $s = x \cdot y \cdot z$ so that*

- $|y| > 0$
- $x \cdot y^i \cdot z \in L$ *for all* $i \geq 0$
- $|x \cdot y| \leq P$

Proof. Suppose that L is regular, and M is a DFA for L. Let P be the number of states of M. Now, take any string $s \in L$ with $|s| \geq P$. Say $s = s_1 s_2 \dots s_n$, with $n \geq P$.

Consider the computation of the automaton M on input $s_1 \dots s_n$. Before the computation begins, the automaton is in state q_0. After reading each bit, say the automaton is in states $q_0, q_1, q_2, \dots$, with $q_t = \delta(q_{t-1}, s_t)$. Now, look at the states in the first p steps of the computation. The machine goes through $p+1$ states, so the states $q_0, \dots, q_p$ cannot be all different. So there must be two distinct time steps $0 \leq i < j \leq p$ such that $q_i = q_j$.

Now, let $x = s_1 s_2 \dots s_i$, $y = s_{i+1} \dots s_j$, $z = s_{j+1} \dots s_n$. First, observe that $xz \in L$. To see why this is try, consider running the machine on xz as input. This is because after reading x, the machine is in state $q_i = q_j$, and reading z takes the machine to q_n, which is an accepting state. Similarly, $xyyz \in L$ because reading the string y while in state $q_i = q_j$ leaves the machine in state $q_i = q_j$. The point is that the machine forgets how many times it sees the pattern y, so we can include it as many times as we like. In addition, since $i < j$, we have $y \neq \varepsilon$ and $|xy| = j \leq P$. □

This lemma gives a property of regular languages, but we will always use it to show that languages are not regular. The outline is to suppose that a language is regular, and to show that there cannot be a pumping length.

Example 3.8. Recall example 3.5. Here is how we prove that Maj is not a regular language, using the Pumping Lemma 3.7.

Suppose that Maj is regular, and let P be the pumping length of Maj. Consider $s = 0 \cdots 01 \cdots 1 \in$ Maj with P zeros and $P+1$ ones. So we break this up as $s = xyz$ where $y = 0 \cdots 0$ for some number i of zeros. But then $xyyz$ contains $p+i$ zeros and $p+1$ ones, which is not in Maj, contradicting the Pumping Lemma. Therefore, Maj is not regular.

The point is that we needed to find the right string to apply the Pumping Lemma. We'll see a few more examples next time.

4. 1/19

Today, we will prove the Myhill-Nerode Theorem, which will give more tools to show that languages are not regular, and give a lower bound on the number of needed states. Then we will give an algorithm such that given a DFA, it returns another DFA with the minimal number of states.

First, we conclude the review of the Pumping Lemma 3.7 and give a few examples.

Example 4.1. This is the language of matched parentheses. The alphabet is $\Sigma = \{(,)\}$. We define $() \in L$, $E \in L$ implies that $(E) \in L$, and $E_1, E_2 \in L$ implies that $E_1E_2 \in L$.

To prove that this language is not regular, we consider the string $((\cdots())\cdots) \in L$ with p open parentheses and p closed parentheses. Now, no matter how we write $p = xyz$, y will always consist of some open parentheses, so $xyyz$ will be unbalanced. Therefore, the Pumping Lemma 3.7 says that this is not regular.

Example 4.2. Consider $L = \{s : |s| \text{ is a prime number}\}$.

Suppose L is regular, so it has pumping length p. Let n be a prime number $> p$, and consider string s of length $|s| = n$. Take $s = 0 \dots 0$, and write $s = xyz$. If the length of y is l, then xy^iz is of length $n + (i-1)l$. If this language was regular, then there would be n, l such that $n + (i-1)l$ is always prime. But this is not true for $i = n + 1$, so that's a contradiction.

Now, let's look at the other approach.

Example 4.3. Recall Example 3.5 with $\Sigma = \{0, 1\}$ and $\text{Maj} = \{s : s \text{ has more 1s than 0s }\}$.

Suppose Maj is regular, and let M be a DFA for Maj. If M has k states, run M on the strings ε ,0, 00, 000, $\dots$, $00\dots0$ (with k zeros). Here, we pass through $k+1$ different inputs, so there are different $0 \le i < j \le k$ and state q such that M reaches q on input 0^i and 0^j.

Now, $0^i1^j \in L$ and M accepts it. However, $0^j1^j \notin L$, but M still accepts it. The point is that the machine M cannot distinguish between reading 0^i and 0^j.

The idea here is that we are looking for two initial segments, leaving the machine in the same state.

Let's give a definition that captures this idea.

Definition 4.4. Let L be a language. Two strings x and y are indistinguishable (relative to L), written $x \approx_L y$, if for every string z, $xz \in L$ if and only if $yz \in L$. Otherwise, we say that x and y are distinguishable: $x \not\approx_L y$. This means that there exists z such that $xz \in L, yz \notin L$ or $xz \notin L, yz \in L$.

Example 4.5. In the Maj example above, we see that $010 \approx_{\text{Maj}} 100$ because $010z$ has more 1s than 0s if and only if $100z$ has more 1s than 0s.

However, $00 \not\approx_{\text{Maj}} 010$ because if $z = 1$ has $001 \in \text{Maj}$ but $0101 \notin \text{Maj}$.

Example 4.6. Consider $L = (0+1)^*0(0+1)(0+1)$ of strings that are 0 in the third-to-last position. Then $\varepsilon \approx_L 111$ because z has a zero in the 3rd to last position if and only if $111z$ has a zero in the 3rd to last position.

However, $010 \not\approx_L 100$ because taking $z = 0$, we have $0100 \notin L$ but $1000 \in L$.

Proposition 4.7. *Suppose that L is regular, and M is a DFA for L. Suppose that $x \not\approx_L y$ are distinguishable strings. Then M on input x reaches a different state than M on input y.*

Proof. Suppose not. Then M reaches state q on input x and on input y. Here x and y are indistinguishable, so $xz \in L$ and $yz \notin L$ for some z. But from state q, the machine does each computation precisely the same way, so M must reach the same state in the two cases. That's a contradiction. □

Remark. If there are k strings $x_1, \dots, x_k$ such that $x_i \not\approx_L x_j$ for all $i \neq j$. Then every DFA for L has $\ge k$ states. We can show this by applying the proposition above.

Fact 4.8. If there are infinitely many strings $x_1, x_2, \ldots$, all pairwise distinguishable, then L is not regular.

This gives a way to show that languages are not regular. Such proofs are usually very similar to proofs by the Pumping Lemma, but a bit more direct.

Example 4.9. Consider the example 4.1 of matched parentheses. Think about strings (, ((, (((, ..., ((⋯(. If x and y are strings of length i and j of open parentheses, and $z =)\cdots)$ of i closing parentheses, this shows that the language is not regular because x and y are distinguishable for all i and j.

Example 4.10. Consider $L = \{xx : x \text{ is a binary string}\}$. Here, the language is $L = \{\varepsilon, 00, 11, 0000, 0101, 1010, \ldots\}$. This shouldn't be regular because we need to remember half of a string to compare it with the other half. So we want to find infinitely many indistinguishable strings.

Take strings 1, 01, 001, 0001, ... Take two of those $x = 0^i1$ and $y = 0^j1$. Take $z = x$. Then $x = 0^i10^i1 \in L$, but $y = 0^j10^i1 \notin L$, so we've produced an infinite set of distinguishable strings.

Even if a language is regular, we can use this fact 4.8 to give a lower bound on the number of states.

Example 4.11. Consider $L = (0+1)^*0(0+1)(0+1)$. There are 8 distinguishable states: 000, 001, 010, ..., 111, so any DFA for this needs to have at least 8 states.

Note that the property $\approx_L$ of being indistinguishable is an equivalence relation. This means that

(1) $x \approx_L x$,
(2) $x \approx_L y$ if and only if $y \approx_L x$,
(3) $x \approx_L y$ and $y \approx_L z$ imply that $x \approx_L z$.

This means that the strings Σ^* can be decomposed into equivalence classes.

Theorem 4.12 (Myhill-Nerode)**.** *A language L is regular if and only if Σ^* has a finite number of equivalence classes under $\approx_L$. If L is regular, the minimal number of states of a DFA for L is the number of equivalence classes.*

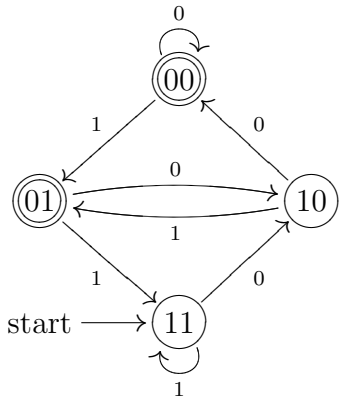
Proof. Taking into account everything that we've said so far (e.g. Fact 4.8), it only remains to prove the following: If there are k equivalence classes of $\approx_L$ then L has a k-state DFA.

Example 4.13. Consider the example of $L = (0+1)^*0(0+1)$. There are four different equivalence classes, corresponding to 00, 01, 10, 11.

Which class does ε belong to? For $\varepsilon z \in L$, we consider $00z$, $01z$, $10z$, $11z$, and observe that ε is in the same class as 11. Similarly, 0 is in the same class as 10, and 1 is in the same class as 11. Now, any longer string is in the same class as its last two bits. So in fact, those four classes are actually all of the equivalence classes of this language. So if this theorem is true, there should be a 4-state DFA. To construct the automaton, follow this procedure:

- allocate one state for each equivalence class.
- the state containing ε is the start state
- states containing strings in L are accept states
- from state $[s]$, reading bit b transitions to state $[sb]$.

In our example, this produces:



To make sure that this DFA makes sense, we need to check that if $x \approx_L y$ and $b \in \Sigma$, and then $xb \approx_L yb$. To see this, if for all z we have $xz \in L$ if and only if $yz \in L$, then for all bz, we have $xbz \in L$ if and only if $ybz \in L$, so $xb \approx_L yb$ is actually a strictly weaker property.

To make sure that the DFA recognizes L, we should check that for all x the DFA on input x reaches the state $[x]$. To prove this, we can proceed by induction on the length of x. On input ε, we end at start state $[\varepsilon]$. For $|x| = n + 1$, take $x = y^n b$. By the inductive hypothesis, y takes the DFA to $[y]$, and after reading b, we see that $[yb] = [x]$. So this is true by induction. □

5. 1/24

A *streaming algorithm* is an algorithm that makes one pass through the data, uses a small amount of memory, and has high efficiency. There are many real-time problems that require this kind of algorithm, such as email messages or router data.

Example 5.1. We want to find the most frequent element of some given list of integers $x_1, x_2, \dots, x_n$.

It is a fact that if we are guaranteed that one number appears a majority ($\geq n/2$) of the time, then there is a $O(\log n)$ memory algorithm that finds it. In fact, if each item is an l-bit string, then the algorithm only needs $l + \log n$ memory.

In general, a simple algorithm uses $O(n(l \log n))$ bits. We will assume that $2^l > n^2$, and we will prove that the memory is $\geq \Omega(nl)$ and $\geq \Omega(n(l + \log n))$.

This is different than from the case of automata, because for automata we have finite memory and only want a yes or no answer. The differences are small enough that we can still handle them.

To get finiteness, consider the most frequent element problem on streams of length n. Now, consider the language $L \subseteq (\{0,1\})^n$ where $\Sigma = \{0,1\}^l$ of streams where $(0, \dots, 0)$ is the most frequent element. This is a problem that we can do.

If we have a streaming algorithm that uses m bits of memory and solves the most frequent element problem, then L_n has a DFA with $\leq 2^m$ states. So if L_n needs $\geq q$ states, then the most frequent element problem needs $\geq \log_2 q$ bits of memory. To do this, we will find lots of distinguishable strings.

First, pick three strings of zeros, and then pick strings $s_1, s_2, \dots, s_{(n-5)/2}$, each of which is used twice. Here, we have selected $n - 2$ items.

We claim that any two strings of this type are distinguishable. So look at two streams of this type. Pick any element where they disagree, say $s_3 \neq t_3$. Then look at

$$(0,\dots,0),(0,\dots,0),(0,\dots,0),s_1,s_1,s_2,s_2,\dots,s_{\frac{n-5}{2}},s_{\frac{n-5}{2}},t_3,t_3$$
$$(0,\dots,0),(0,\dots,0),(0,\dots,0),t_1,t_1,t_2,t_2,\dots,t_{\frac{n-5}{2}},t_{\frac{n-5}{2}},t_3,t_3.$$

So those are all strings that are distinguishable. We've found a distinguishable string for each choice of $\frac{n-5}{2}$ strings in $\{0,1\}^l - \{(0,\dots,0)\}$. We have therefore found

$$\binom{2^l-1}{\frac{n-5}{2}} \geq \left(2\cdot\frac{2^l-1}{e(n-5)}\right)^{\frac{n-5}{2}}$$

distinguishable strings, where we have used $\binom{n}{k} \geq (\frac{n}{ek})^k$. Therefore, every most frequent element streaming algorithm needs

$$\geq \log_2\left(2\cdot\frac{2^l-1}{e(n-5)}\right)^{\frac{n-5}{2}} = \left(\frac{n-5}{2}\right)\log_2\left(\frac{2(2^l-1)}{e(n-5)}\right) \geq \Omega(n\log_2\sqrt{2^l}) = \Omega(nl).$$

Example 5.2. In this problem, we want to find the number of distinct elements. We use the same setting as the previous problem: we have n elements of $\{0,1\}^l$. How many distinct values appear in the sequence $x_1,\dots,x_n$? For example, we might want to count the number of distinct users of a system.

First, we consider an approximation algorithm.

Algorithm 5.3. Consider some hash function $h : \{0,1\}^l \to (0,1]$. (For the analysis, assume that it is a perfect hash function.) Then we take the elements $x_1, x_2, \dots, x_n$ and compute the hash values $h(x_1), h(x_2), \dots$ and stores the minimum m. We only need memory to store m and h, which is $O(\log n)$ memory. It then outputs $\frac{1}{m}$.

Suppose that there are k distinct elements in $x_1,\dots,x_n$. Then $h(x_1),h(x_2),\dots,h(x_n)$ has k distinct inputs, with repetitions. So we have k random real numbers in $(0,1]$ with repetitions. This is equivalent to picking k real numbers in $(0,1]$ and outputting the minimum, and this is approximately $\frac{1}{k}$. Therefore, the output of $\frac{1}{m}$ is approximately k.

If we were not allowed to use randomness, we will show that there is no good streaming algorithm for this problem. Here, we want to compute the number of distinct elements exactly. Look at only n-element strings, and look at the language $L \subset (\{0,1\})^n$ of streams with $\frac{n}{2}$ distinct elements.

Consider all sequences of $\frac{n}{2}$ elements, all distinct, and consider $s_1, s_2, \dots, s_{n/2}$. We claim all strings of this form are distinguishable. Indeed, consider

$$s_1, s_2, \dots, s_{n/2}, s_1, s_2, \dots, s_{n/2}$$
$$t_2, t_2, \dots, t_{n/2}, s_1, s_2, \dots, s_{n/2}.$$

The first case has $n/2$ distinct elements, while the second case has more. Therefore, the number of distinguishable streams is $\binom{2^l}{n/2}$. Therefore, our lower bound is

$$\log_2\binom{2^l}{n/2} \geq \log_2\left(2\cdot\frac{2^l}{n}\right)^{n/2} = \Omega\left(\frac{n}{2}\log_2\frac{2^l}{n}\right) = \Omega(nl).$$

Example 5.4. Suppose we had a deterministic algorithm A for the distinct elements problem that gives a 10% approximation and uses m bits of memory.

It is a fact that it is possible to construct $2^{\Omega(nl)}$ strings in $(\{0,1\}^l)^{n/2}$ such that each string has $n/2$ distinct elements, and each pair of strings in the set has at most $\leq n/10$ elements in common.

To do this, think of a graph where every sequence of $n/2$ distinct elements of $\{0,1\}^l$ is a vertex, and we have an edge between two vertices if the sequence have $\geq n/10$ elements in common. If a graph has N vertices and degree $\leq D$ at each vertex, then there is an independent set of size $\geq N/D+1$. The reason is that we can repeatedly pick a vertex and delete all of its neighbors. In our case, we have $N = \binom{2^l}{n/2}$ and $D \leq \binom{n/2}{n/10}\binom{2^l - n/10}{4/10 \cdot n}$, and this is a calculation that we can now do.

Consider any two string from that set, say

$$s_1, s_2, \ldots, s_{n/2}$$
$$t_1, t_2, \ldots, t_{n/2}.$$

If 2^m is smaller than the number of strings in our set, then there are two different strings such that A has the same internal state after reading each of them. Attach a copy of the first string to both of them. Since A is a deterministic algorithm, it gives the same output to both of them. However, in the first case, the correct answer is $n/2$, which in the second case, the correct answer is $\geq \frac{9}{10}n$. Therefore, the algorithm A could not have given a correct answer in both cases.

Therefore, the amount of memory that such an algorithm A consumes is

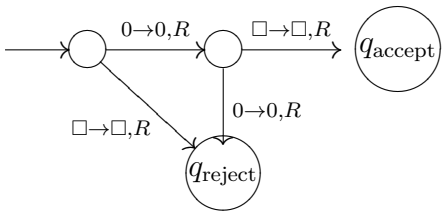
$$m \geq \log(\#\text{ of strings in our set}) \geq \log 2^{\Omega(nl)} = \Omega(nl).$$

6. 1/26

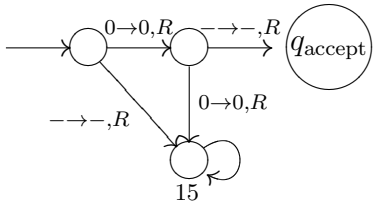
Finite automata are very simple, but we understand them very well. Unfortunately, there's a lot that they can't do. In this part of the course, we will look at Turing machines, and determine what is computable, in general.

To start, a Turing machine (named after Turing) has finite state control, with finitely many states. It is given an infinite tape of input, and it can also write. It can read one bit of the table at a time, and move left or right based on a transition table.

The state diagrams for Turing machines look like those for finite automata, but they have a lot more information:



This recognizes the language $\{0\}$. Alternatively,



Here, instead of rejecting, we infinite loop.

What are differences between Turing machines and DFAs? The Turing machine can both read and write, the head can move left and right, and the input doesn't have to be read entirely, and the computation can continue further after all input has ended. The accept and reject states take immediate effect.

Example 6.1. Consider $L = \{w\#w \mid w \in \{0,1\}^*\}$. We know from the Pumping Lemma 3.7 that no DFA gives this language.

Here is pseudocode for this problem.

(1) If there's no # on the tape, reject.
(2) While there is a bit to the left of #, replace the first bit with X and check if the first bit to the right of the # is identical. (If not, reject.). Replace that bit with an X too.
(3) If there's a bit to the right of #, then reject. Else, accept.

Definition 6.2. A *Turing machine* is a 7-tuple $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$.

- Q is a finite set of states
- Σ is the input alphabet, where $\square \notin \Sigma$.
- Γ is the tape alphabet, where $\square \in \Phi$ and $\Sigma \subseteq \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- q_0 is the start state
- q_{accept} is the accept state
- q_{reject} is the reject state and $q_{\text{accept}} \neq q_{\text{reject}}$

Definition 6.3. Let C_1 and C_2 be configurations of a machine M. We say that C_1 yields C_2 if, after running M in C_1 for one step, M is then in configuration C_2

Example 6.4. Suppose $\delta(q_1, b) = (q_2, c, L)$. Then aaq_1bb yields aq_2acb.

Suppose $\delta(q_1, a) = (q_2, c, R)$. Then $cabq_1a$ yields $cabcq_2\square$.

Definition 6.5. Let $w \in \Sigma^*$ and M be a Turing machine. M accepts w if there are configurations $C_0, C_1, \ldots, C_k$ such that

- $C_0 = q_0w$
- C_i yields C_{i+1} for $i = 0, \ldots, k-1$
- C_k contains the accepting state q_{accept}.

The states $C_0, \ldots, C_k$ is called the accepting computation history. This will be unique for every input.

Definition 6.6. A Turing machine recognizes a language L if M accepts all and only those strings in L.

A language L is called recognizable or recursively enumerable if some Turing machine recognizes L.

Definition 6.7. A language L is called decidable (recursive) if some Turing machine decides it.

It turns out that the set of decidable languages is a proper subset of the recognizable languages.

Definition 6.8. A Turing machine decides a language if M accepts all strings in L *and* rejects all strings not in L.

Example 6.9. Consider $\{0^{2^n} \mid n \geq 0\}$. The pumping lemma says that this is not regular.

Here is pseudocode:

(1) Sweep from left to right, crossing out every other 0
(2) If in stage 1, the tape had only one 0, accept.
(3) If in stage 1, the tape had an odd number of 0s, reject.
(4) Move the head back to the first input symbol.
(5) Go to stage 1.

Why does this work? Every time we return to stage 1, the number of 0s on the tape is halved.

Example 6.10. $C = \{a^i b^j c^k \mid k = i \times j, i, j, k \geq 1\}$.

Pseudocode:

(1) If the input doesn't match $a^*b^*c^*$, reject (e.g. using a DFA).
(2) Move the head back to the leftmost symbol.
(3) Cross off an a, scan to the right until b. Sweep between bs and cs, crossing off one of each until all bs are crossed off. If all cs get crossed off while doing this, reject.
(4) Uncross all the bs. If there's another a left, then repeat stage 3. If all as are crossed out, check if all c are crossed off. If yes, then accept. Else, reject.

For example, we have

$$aabbbccccc \to xabbbccccc \to xayyyzzzccc \to xabbbzzzccc \to xxyyyzzzzzz.$$

One amazing aspect of Turing machine is that we can define many variations on them. But so long as the new model is finite, it will be equivalent to the usual Turing machine.

We can define Turing machines to have multiple tapes. It has a head that can move left or right on each tape. The transitions are then $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$.

Theorem 6.11. *Every multitape Turing machine can be transformed into a single tape Turing machine.*

In fact, the single tape machine only runs polynomially slower than the multitape machine.

Proof. For every tape, we have a special symbol #. On the single tape machine, simply put all of the data on the k tapes onto a single tape, separated by #. Put a mark on each of the k head positions (by doubling the tape alphabet to include marked bits); these are like virtual tape heads. If we move a tape head so far that we run past what we have stored on the single tape, shift everything to the right by one square. □

We can define nondeterministic Turing machines analogously to the case of DFAs. This is a powerful idea when considering efficiency, but with respect to language recognition, this is actually equivalent to Turing machines.

Theorem 6.12. *Every nondeterministic Turing machine N can be transformed into a single tape Turing machine M that recognizes the same language.*

Proof. For all strings $C \in \{Q \cap \Gamma \cap \#\}^*$ is lexicographical order, check if $C = C_0 \# \cdots \# C_k$ where $C_0, \ldots, C_k$ is some accepting computation history for N on w. If so, accept. □

We can encode a Turing machine as a bit string.

$$0^n10^m10^k10^s10^t10^r10^u1\ldots.$$

This means that there are n states, m tape symbols, the first k are input symbols, the start state is s, the accept state is t, the reject state is r, and the blank symbol is u. Then we can encode transitions as $((p,a),(q,b,L)) = 0^p10^a10^q10^b10$ and $((p,a),(q,b,R)) = 0^p10^a10^q10^b11$.

Similarly, we can encode DFAs and NFAs as bit strings. So we can define the following languages:

- $A_{\text{DFA}} = \{(B,w) \mid B \text{ is a DFA that accepts string } w\}$
- $A_{\text{NFA}} = \{(B,w) \mid B \text{ is an NFA that accepts string } w\}$
- $A_{\text{TM}} = \{(B,w) \mid B \text{ is a Turing machine that accepts string } w\}$

(We can encode $(x,y) = 0^{|x|}1xy$.)

Theorem 6.13. *A_{DFA} is decidable.*

This means that there is a Turing machine that can simulate any DFA.

Proof. We directly simulate the DFA on w. The idea is very simple but the details are complicated. □

Theorem 6.14. *A_{NFA} is decidable.*

Proof. Given an NFA, make it a DFA. This is a completely effective procedure, and we can implement it. Now apply the previous result. □

It turns out that A_{TM} is recognizable but not decidable. We'll see more about this later.

Theorem 6.15. *There is a universal Turing machine U that can take as input*

- *the code of an arbitrary Turing machine M*
- *an input string w*

such that $U(m,w)$ accepts if and only if $M(w)$ accepts.

This is a fundamental property: Turing machines can run their own code! Note that DFAs and NFAs do not have this property. That is, A_{DFA} is not a regular language. (This is something that we can prove using the Pumping Lemma 3.7.)

We've seen that we can change the definition of a Turing machine in many different ways without producing a more powerful machine. This led to the Church-Turing thesis: Everyone's intuitive notion of algorithms is given by Turing machines. This thesis is tested every time you write a program that does something!

We've seen Turing machines, and they're nice because they are really simple. The bad side is that implementing things in Turing machines is unpleasant. We get around this by being less formal and writing pseudocode. But it is important to keep in mind: How does one actually implement such a thing?

Theorem 6.16. *A language L is decidable if both it and its complement are recognizable.*

Recall that given $L \subseteq \Sigma^*$, define $\neg L = \Sigma^* - L$.

Proof. We are given a Turing machine M_1 that recognizes A and a Turing machine that recognizes $\neg A$. We want a machine that decides A. To do this, run M_1 and M_2 at the same time, e.g. alternately take steps on each of the two machines. □

Theorem 6.17. *There are languages over* $\{0,1\}$ *that are not decidable.*

Assuming the Church-Turing thesis, this means that there are problems that *no* computing device can solve.

We will prove this using a counting argument. We will show that there is no onto function from the set of all Turing machines to the set of all languages over $\{0,1\}$. That is, every mapping from Turing machines to languages must "miss" some language.

7. 1/31

Today we will look at Turing machines in more depth, and discuss undecidability, recognition, and enumeration.

Recall:

Theorem 7.1. *L is decidable if and only if both L and $\neg L$ are recognizable.*

Proof. We have Turing machines M_1 and M_2 that recognize A and $\neg A$. We then building a machine M that decides A by running M_1 and M_2 on different tapes. If M_1 halts then accept; if M_2 halts then reject. □

The goal of this lecture is to show that there are languages over $\{0,1\}$ that are not decidable. Assuming the Church-Turing these, this means that there are problems that no computing device can solve.

In fact, we can show that there are languages over $\{0,1\}$ that are not recognizable. Here's an outline:

(1) Every recognizable language can be associated with a Turing machine.
(2) Every Turing machine corresponds to a bit string.
(3) So there is a 1-1 mapping from the set of all recognizable languages to $\{0,1\}^*$.
(4) But the set of all languages has a bijection with the power set of $\{0,1\}^*$.
(5) The power set of A is always larger than A, so there must be unrecognizable languages.

Informally, this says that "There are more problems to solve than there are programs to solve them."

Theorem 7.2. *Let L be any set, and let 2^L be the power set of L. There is no onto map from L to 2^L.*

A map $f : A \to B$ is onto if for all $b \in B$ there exists $a \in A$ such that $f(a) = b$.

Proof. Assume, for a contradiction, that there is an onto map $f : L \to 2^L$.

Let $S = \{x \in L : x \notin f(x)\} \in 2^L$. If f is onto, then there is a $y \in L$ where $f(y) = S$. Suppose that $y \in S$. By definition of S, $y \notin f(y) = S$. So $y \notin S$. By definition of S, $y \notin f(y) = S$. This is a contradiction. □

The moral here is that no map from L to the power set 2^L can cover all elements in 2^L. No matter what the set L is, the power set always has larger cardinality.

Theorem 7.3. *There are undecidable (and unrecognizable) languages over* $\{0,1\}$.

Proof. Consider a map from recognizable languages over $\{0,1\}^*$ into Turing machines. Since the Turing machines can be represented by bit strings, we can injectively map it into $\{0,1\}^*$, which we will call L. This means that L covers all recognizable languages.

However, the set of all languages over $\{0,1\}^*$ is the set of strings of 0s and 1s, which is the set of all subsets of L, or 2^L. Therefore, there are unrecognizable languages. □

What property of Turing machines did we use here? All we used was that Turing machines have a finite description (via finite bit strings). This means that all models of finite computation have problems that they cannot solve. Also, the set L is countable, but the set 2^L is not countable.

These issues also appear in set theory. In the early 1900's, logicians were trying to define consistent foundations for mathematics. Suppose that X is the universe of all axioms. Then Frege's Axiom is: Let $P : X \to \{0,1\}$. Then we can define $\{S \in X : P(S) = 1\}$. Russell defined $F = \{S \in X : S \notin S\}$. Suppose $F \in F$. Then by definition, $F \notin F$. So $F \notin F$, and by definition, $F \in F$. This is bad, and it is called Russell's paradox. This means that this naive set theory is inconsistent.

Theorem 7.4. *There is no onto function from the positive integers $\mathbb{Z}^+$ to the real numbers in $(0,1)$.*

Proof. Suppose that f is such a function. Write out a table of values of f. Pick a number r such that the nth digits of r is 1 if the nth digit of $f(n)$ is not 1, and 0 otherwise. This means that $f(n) \neq r$ for all n. So r is never output by f, and we are done. This proof is due to Cantor. □

That's a negative result, but we can be more positive.

Theorem 7.5. *Let $\mathbb{Z}^+ = \{1,2,3,4,\dots\}$. There is a bijection between $\mathbb{Z}^+$ and $\mathbb{Z}^+ \times \mathbb{Z}^+$.*

Proof. To show this, make a table of pairs, and successively look at diagonals. □

The Calkin-Wilf tree gives a bijection between the positive integers and the rational numbers. In this tree, $\frac{a}{b}$ has two children: $\frac{a}{a+b}$ and $\frac{b}{a+b}$. Enumerating the rationals requires doing a breadth first search on the tree. It is interesting that each rational appears precisely once. There is also Stern's diatomic sequence, which also gives such a bijection and has a nice recursive definition.

Now, we give a concrete undecidable problem. Recall that we defined the language $A_{TM} = \{(M,w) : M \text{ is a Turing machine that accepts string } w\}$.

Theorem 7.6. *A_{TM} is recognizable but not decidable.*

Proof. We can define a universal Turing machine can recognizes A_{TM}.

Assume that machine H decides A_{TM}. Then $H(M,w)$ accepts of M accepts w and rejects if M does not accept w. Construct a new Turing machine D as follows: On input M, run H on (M,M) and output the opposite of H.

The $D(D)$ should reject if D accepts D and accept if D does not accept D, which is nonsense. □

Now, we give a more constructive proof of this result.

Proof. Assume that machine H recognizes A_{TM}. Then $H(M,w)$ accepts if M accepts w, and rejects or loops otherwise. Construct a new Turing machine D_H as follows: reject if M accepts M, accept if M rejects M, and loops if M loops on M.

Now, D_H rejects if D_H accepts D_H, and D_H accepts if D_H rejects D_H. Both of these are nonsense, so D_H must loop on D_H. There is no contradiction here: D_H must run forever on D_H.

Given any machine H recognizing A_{TM}, we can effectively construct an instance (D_H, D_H) which does not belong to A_{TM} but H runs forever on the input (D_H, D_H). Therefore, H cannot decide A_{TM}. Given any program that recognizes the acceptance problem, we can efficiently construct an input where the program hangs. □

Corollary 7.7. *$\neg A_{TM}$ is not recognizable!*

Proof. Suppose $\neg A_{TM}$ is recognizable. We know that A_{TM} is recognizable, but then A_{TM} is decidable, which is not true. □

Consider a language $\text{HALT}_{TM} = \{(M, w) : M \text{ is a Turing machine that halts on string } w\}$.

Theorem 7.8. *$HALT_{TM}$ is undecidable.*

Proof. Assume there is H that decides HALT_{TM}. We construct M' that decides A_{TM}.

Define $M'(M, w)$ to run $H(M, w)$. If H rejects then reject. If H accepts then run M on w until it gives an answer, and return that answer. □

Consider a language $\text{EMPTY}_{TM} = \{M : M \text{ is a Turing machine such that } L(M) = \emptyset\}$. Given a program, does it always reject? This is called the Emptiness problem.

Theorem 7.9. *$EMPTY_{TM}$ is undecidable.*

Proof. Assume there is a Turing machine E that decides EMPTY_{TM}. We'll use it to get a decider D for A_{TM}.

Build a Turing machine $D(M, w)$ with the behavior: $M'(x)$ rejects if $x \neq w$, otherwise simulate $M(w)$. Then run $E(M')$. If E accepts, reject. If E rejects, then accept. This machine decides A_{TM}. □

This allows us to say something about the complement of emptiness: $\neg\text{EMPTY}_{TM} = \{M : M \text{ is a Turing machine such that } L(M) \neq \emptyset\}$. Given a program, does it accept some input?

Theorem 7.10. *$\neg EMPTY_{TM}$ is recognizable.*

Proof. Define $M'(M)$ as follows. For all pairs of positive integers (i, t), let x be the ith string is lexicographical order. Determine if $M(x)$ accepts within t steps. If yes then accept.

Then $L(M) \neq \emptyset$ if and only if M' halts and accepts M. □

Corollary 7.11. *$EMPTY_{TM}$ is unrecognizable.*

Proof. Suppose not. Then EMPTY_{TM} and $\neg\text{EMPTY}_{TM}$ are recognizable, so EMPTY_{TM} decidable, which is false. □

One can often show that a language L is undecidable by showing that if L is decidable, then so is A_{TM}. We reduce A_{TM} to the language L, or $A_{TM} \leq L$. For example, we've shown that $A_{TM} \leq \text{HALT}_{TM}$.

Definition 7.12. $f : \Sigma^* \to \Sigma^*$ is a *computable function* if there is a Turing machine M that halts with just $f(w)$ written on its tape, for every input w.

A language A is *mapping reducible* to language B, written $A \leq_M B$ if there is a computable $f : \Sigma^* \to \Sigma^*$ such that for every w, $w \in A$ if and only if $f(w) \in B$.

f is a called a mapping reduction (or many-one reduction) from A to B.

Next time, we'll look for carefully at undecidability and reductions between problems.

8. 2/2

The topic for today is reductions and undecidability.

Recall that last time we had a concrete undecidable problem A_{TM}. This was recognizable but not decidable, and as a corollary, $\neg A_{TM}$ is not recognizable. This was used to prove undecidability of HALT_{TM}.

One can often show that a language L is undecidable by showing that if L is decidable then so is A_{TM}. We reduce A_{TM} to the language L, or $A_{TM} \leq L$ to say that L is harder than A_{TM}.

Definition 8.1. $f : \Sigma^* \to \Sigma^*$ is a computable function if there is a Turing machine M that halts with just $f(w)$ written on its tape, for every input w.

Definition 8.2. A language A is mapping reducible to language B, written $A \leq_m B$, if there is a computable $f : \Sigma^* \to \Sigma^*$ such that for every w, $w \in A$ if and only if $f(w) \in B$.

Theorem 8.3. *If $A \leq_m B$ and $B \leq_m C$ then $A \leq_m C$.*

This $\leq_m$ relation behaves like some sort of partial order.

Proof. Draw a picture and wave your hands at it. □

Theorem 8.4. *If $A \leq_m B$ and B is decidable, then A is decidable.*

Proof. Let M decide B and let f be a reduction from A to B. To decide A, we build a new machine M'. To compute $M'(w)$, it computes $f(w)$, runs M on $f(w)$, and outputs its answer. This works because of the property that $w \in A$ if and only if $f(w) \in B$. □

Theorem 8.5. *If $A \leq_m B$ and B is recognizable, then A recognizable.*

Proof. Let M recognize B and let f be a reduction from A to B. To recognize A, we build a new machine M'. To compute $M'(w)$, it computes $f(w)$, runs M on $f(w)$, and outputs its answer if it ever receives one. This works because of the property that $w \in A$ if and only if $f(w) \in B$. □

Corollary 8.6. *If $A \leq_m B$ and A is undecidable then B is undecidable.*

Corollary 8.7. *If $A \leq_m B$ and A is unrecognizable, then B is unrecognizable.*

All undecidability proofs we've seen can be viewed as constructing an f that reduces A_{TM} to some language.

Theorem 8.8. *We showed $A_{TM} \leq_m HALT_{TM}$.*

Is there a reduction $\text{HALT}_{TM} \leq A_{TM}$? Yes. Define $f(M, w) = (M', w)$ where $M'(w)$ accepts if $M(w)$ ever halts, and loops otherwise. Then $(M, w) \in \text{HALT}_{TM}$ if and only if $(M', w) \in A_{TM}$.

We also had the emptiness problem, and we showed that EMPTY_{TM} is undecidable.

Theorem 8.9. *In fact, $EMPTY_{TM}$ is unrecognizable.*

Proof. Show that $\neg A_{TM} \leq_m \text{EMPTY}_{TM}$.

Define $f(M, w)$ to output a Turing machine M' with the behavior $M'(x)$ is if $x \neq w$ the reject, otherwise run $M(w)$. □

We can also consider the regularity problem. We write $\text{REGULAR}_{TM} = \{M : M \text{ is a Turing machine an}$ Given a program, is it equivalent to some DFA?

Theorem 8.10. *$REGULAR_{TM}$ is not recognizable.*

Proof. We show $\neg A_{TM} \leq_m \text{REGULAR}_{TM}$.

Define $f(M, w)$ to output a Turing machine M' such that $M'(x)$ behaves as follows: if $x = 0^n 1^n$ then simulate $M(w)$, else reject. □

The equivalence problem: $\text{EQ}_{TM} = \{(M, N) : M, N \text{ are Turing machines and } L(M) = L(N)\}$.

Theorem 8.11. *EQ_{TM} is unrecognizable.*

Proof. We reduce EMPTY_{TM} to EQ_{TM}.

Let $M_\emptyset$ to be some Turing machine with no path from start state to accept state. Define $f(M) = (M, M_\emptyset)$. □

Now, consider Post's Correspondence Problem. We call this the PCP game, or "domino solitaire". We have a finite number of dominoes, with a string of each of top and bottom. We have an unbounded stack of these. The goal is to take some subset of these dominoes and line them up, so that the string on the top is the same as the string on the bottom. This is a simple little game. In some cases, we can't possibly have a match. When can we win?

(1) If every top string is longer than the corresponding bottom one, there can't be a match.
(2) If there is a domino with the same string on the top and on the bottom, these is a match.

Problem 8.12 (Post's Correspondence Problem)**.** Given a collection of domino types, can we build up a match?

Let $\text{PCP} = \{P : P \text{ is a set of dominoes with a match }\}$.

Theorem 8.13. *PCP is undecidable.*

We'll do this by considering a variant of the game, called the FPCP game. It is just like the PCP game, except that a match has to start with the first domino type.

Theorem 8.14. *FPCP is undecidable.*

Proof. We will reduce A_{TM} to FPCP.

Recall that an accepting computation history for M on w is a sequence of configurations $C_0, C_1, \ldots, C_k$ where C_0 is the start configuration $q_0 w$, C_k is an accepting configuration, and each configuration C_i yields C_{i+1}. $M(w)$ accepts if and only if such a history exists.

We'll build an instance of FPCP such that a match encodes an accepting computation history. The simple rules of this game are enough to simulate arbitrary Turing machines. We want the string produced by a match to be the accepting computation history, with different steps separated by some special character #.

(1) Given (M, w) we will construct an instance P of FPCP in seven steps. We have a special domino type that must go first, so first put

$$\frac{\#}{\#q_0 w_1 w_2 \dots w_n \#}$$

into P.

(2) If $\delta(q, a) = (p, b, R)$ then add

$$\frac{qa}{bp}.$$

(3) If $\delta(q, a) = (p, b, L)$ then add

$$\frac{cqa}{pcb}$$

for all $c \in \Gamma$.

(4) Add $\frac{a}{a}$ for all $a \in \Gamma$.

(5) Add $\frac{\#}{\#}$ and $\frac{\#}{\square\#}$.

(6) Add

$$\frac{aq_{\text{acc}}}{q_{\text{acc}}}, \frac{q_{\text{acc}}a}{q_{\text{acc}}}$$

for all $a \in \Gamma$, including $\square$. (For Turing machines, this corresponds to the idea that we don't care what happens after we are in an accept state.)

(7) Add

$$\frac{q_{\text{acc}}\#\#}{\#}.$$

When we run this, the bottom is longer than the top, so we are forced to make certain choices to ensure that the top and bottom match.

We've shown that given (M, w) we can effectively construct an instance of FPCP that has a match if and only if M accepts w. So $A_{TM} \leq_m FPCP$.

Can we reduce FPCP to PCP? For $u = u_1 u_2 \dots u_n$, where $u_i \in \Gamma \cup Q \cup \{\#\}$, define

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

For each FPCP instance

$$\frac{t_1}{b_1}\frac{t_2}{b_2}\cdots\frac{t_k}{b_k}$$

we consider the PCP instance

$$\frac{\star t_1}{\star b_1 \star}\frac{\star t_1}{b_1 \star}\frac{\star t_2}{b_2 \star}\cdots\frac{\star t_k}{b_k \star}\frac{\star \diamond}{\diamond}.$$

Even though this is a PCP instance, we can't start with anything besides the first domino. Therefore, we've shown that $A_{TM} \leq_m PCP$ and PCP is undecidable. $\square$

Now, we discuss oracle Turing machines and hierarchies of undecidable problems. Recall that we have languages A_{TM}, HALT_{TM}, and EMPTY_{TM}. Are all of these "equally" hard? How can we even formalize that?

Oracle Turing machines are just like normal Turing machines, except they have access to an oracle. They have a new kind of state, $q_?$. In this state, the oracle gives us answers to

recognition problems, e.g. is (M, w) in A_{TM}. To make this work with our formalism, we go to a state q_{yes} or q_{no} depending on what the oracle says.

Slightly more formally:

Definition 8.15. An *oracle* is a set B to which the Turing machine may ask membership questions and the Turing machine always receives a correct answer in one step.

This makes sense even if B is not decidable!

Definition 8.16. A is *recognizable in* B if there is an oracle Turing machine M with oracle B that recognizes A.

Definition 8.17. A is *decidable in* B if there is an oracle Turing machine M with oracle B that decides A.

Proposition 8.18. *$HALT_{TM}$ is decidable in A_{TM}.*

Proof. On input (M, w), decide if M halts on w as follows. Ask the orcale for A_{TM} if M accepts w. If yes, then accept. Switch the accept and reject states, and query if oracle again. If yes, then accept. Otherwise, reject. □

Proposition 8.19. *A_{TM} is decidable in $HALT_{TM}$.*

Proof. Same as for normal Turing machines. □

We can say that A Turing reduces to B, or $A \leq_T B$.

Theorem 8.20. *If $A \leq_m B$ then $A \leq_T B$.*

Proof. If $A \leq_m B$ then there is a computable function $f : \Sigma^* \to \Sigma^*$ where for every w, $w \in A$ if and only if $f(w) \in B$. We can thus use an oracle for B to decide A. □

Theorem 8.21. *$\neg HALT_{TM} \leq_T HALT_{TM}$.*

Theorem 8.22. *$\neg HALT_{TM} \not\leq_m HALT_{TM}$.*

There has many a lot of work on oracle Turing machines, and there are many hierarchies of problems. As an example, the following problem cannot be decided by a Turing machine with an oracle for the Halting problem. Define

$$\text{SUPERHALT} = \{(M, x) : M \text{ with an oracle for the halting problem, halts on } x\}.$$

We can still use the same diagonalization argument here! Suppose that H decides that SUPERHALT (with an oracle). Define $D(X)$ to be if $H(X, X)$ accepts (with oracle) then loop, else accept.

9. 2/7

This lecture will be about some very powerful tools in computability.

Proposition 9.1. *Define*

$$\text{REVERSE} = \{M : M \textit{ is a TM such that for all } w, M(w) \textit{ accepts iff } M(w^R) \textit{ accepts}\}.$$

This is undecidable.

Proof. Given a machine D for deciding the language REVERSE, we show how to decide A_{TM}.

Construct the machine M_w taking input x. If $x = 01$, it accepts; else, if $x = 10$, it runs $M(w)$. Feed the resulting language $L(M_w)$ into our decider D for REVERSE. Now, if M accepts w, then the language is two strings $\{10, 01\}$ which is closed under reverse, and the decider says yes. But if M does not accept w, then the decider says no; $\{01\}$ is not closed under reverse. □

Proposition 9.2. *Define*

$\{(M, w) : M$ *is a TM that on input* w*, tries to move its head past the left end of the tape*$\}$.

This is undecidable.

Proof. We reduce from A_{TM} to the above language.

On input (M, w), make a Turing machine N that marks the leftmost tape cell, shifts input w over one square, then simulates $M(w)$. If M moves to the marked cell, N moves the head back to the right. If M accepts, N tries to move its head past the left end of the tape.

Therefore, (M, w) is in A_{TM} if and only if (N, w) has this property. □

Proposition 9.3. *Define*

$\{(M, w) : M$ *is a TM that on input* w*, moves its head left at least once, at some point*$\}$.

This is decidable.

Proof. On input (M, w), run the machine for $|Q_M| + |w| + 1$ steps. We accept if M's head moved left at all. Otherwise, we reject.

Why does this work? Suppose we simulate the machine for more than $|w|$ steps, and it never moved left. Then it would be reading blanks. Now, run it for $|Q_M|$ steps. On each step, it's reading a blank and never goes left. At this moment, we're out of possibilities. It must repeat states, and we've ended up in an infinite loop of reading blanks and moving right. □

Let L be a language over Turing machines. Assume that L satisfies the following properties:

Semantic: For any Turing machines M_1 and M_2 where $L(M_1) = L(M_2)$, $M_1 \in L$ if and only if $M_2 \in L$. This means that the language recognized by the machine determines whether the machine is in the language.

Nontrivial: There are Turing machines M_{yes} and M_{no}, where $M_{\text{yes}} \in L$ and $M_{\text{no}} \notin L$.

Then L is undecidable.

Example 9.4. Here are some properties which are semantic:

- M accepts ε
- $L(M) = \{0\}$
- $L(M)$ is nonempty
- $L(M)$ is regular
- M accepts exactly 154 strings.

In general, $L = \{M : P(M) \text{ is true}\}$ is undecidable.

Example 9.5. Here are some nonsemantic properties:

- M halts and rejects ε

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

Theorem 9.6 (Rice's Theorem)**.** *Any semantic nontrivial L over Turing machines is undecidable.*

"Every nontrivial semantic property of Turing machines is undecidable."

Proof. We'll reduce from A_{TM} to the language L. Define $M_\emptyset$ to be a Turing machine that never halts. Assume, WLOG, that $M_\emptyset \notin L$. Let $M_{\text{yes}} \in L$ (such a machine exists, by assumption).

Here's the reduction. On input (M, w), output $M_w(x)$, where if M accepts w and M_{yes} accepts x, then *accept*.

If M accepts w, then $L(M_w) = L(M_{\text{yes}})$. Since $M_{\text{yes}} \in L$, we have $M_w \in L$.

If M does not accept w, then $L(M_w) = L(M_\emptyset) = \emptyset$. Since $M_\emptyset \notin L$, we have $M_w \notin L$. So we have a reduction from A_{TM} to the language L.

If $M_\emptyset \in L$, then we reduce $\neg A_{\text{TM}}$ to L. Define: $M_w(x)$ to be if M accepts w and M_{no} accepts x, then *accept*.

In this case, if M accepts w, then $L(M_w)$ is the same language as $L(M_{\text{no}}) \notin L$. If M does not accept w, then $L(M_w)$ is the same language as $L(M_\emptyset) \in L$. □

One of these is recognizable; the other is not. Which one is which?

(1) $\{M : L(M)$ contains at most 154 strings$\}$
(2) $\{M : L(M)$ contains at least 154 strings$\}$

The second is recognizable. Just enumerate all possible strings and simulate each of them, counting how many are in $L(M)$. If we ever hit 154, we can accept; otherwise, we just keep going.

Is there a generic condition for unrecognizability, in the style of Rice's Theorem for undecidability?

Theorem 9.7 (Rice's Theorem, part II)**.** *Let L be a language over Turing machines. Assume that L is semantic and non-monotone: There are Turing machines M_{yes} and M_{no} where $M_{\text{yes}} \in L$, $M_{\text{no}} \notin L$, and $L(M_{\text{yes}}) \subset L(M_{\text{no}})$. Then L is unrecognizable.*

"Every non-monotone semantic property of Turing machines is unrecognizable."

Example 9.8. Here are monotone properties. This means that if $M_{\text{yes}} \in L$, then for all $T \supseteq L(M_{\text{yes}})$, if $L(M_{\text{no}}) = T$, then $M_{\text{no}} \in L$.

- $L(M)$ is infinite
- $L(M) = \Sigma^*$
- $L(M)$ contains at least 154 strings
- $L(M)$ contains 11111.

As we accept more strings, we don't fall out of having the property.

Example 9.9. Here are non-monotone properties:

- $L(M)$ is finite
- $L(M) = \{0\}$

- $L(M)$ is regular
- $L(M)$ is not regular
- $L(M)$ contains at most 154 strings.

There is some point at which accepting more strings would make us fall out of having the property.

The point is that $L = \{M : P(M) \text{ is true}\}$ is unrecognizable.

Proof. Idea: Give a mapping reduction from $\neg A_{\text{TM}}$ to L.

On input (M, w), output the following machine $M_w(x)$: Run $M_{\text{yes}}(x)$, $M_{\text{no}}(x)$, $M(w)$. If M accepts w and M_{no} accepts x, then *accept*. If M_{yes} accepts x, then *accept*.

If M accepts w, then $L(M_w) = L(M_{\text{no}})$. Since $L(M_{\text{yes}}) \subset L(M_{\text{no}})$, we have $M_w \notin L$

If M does not accept w, then $L(M_w) = L(M_{\text{yes}})$. Since $M_{\text{yes}} \in L$, we have $M_w \in L$.

So $(M, w) \in A_{\text{TM}}$ if and only if $M_w \notin L$. □

Now, we discuss self-reference and the Recursion Theorem.

Theorem 9.10. *There is a computable function $q : \Sigma^* \to \Sigma^*$, where for any string w, $q(w)$ is the description of a Turing machine P_w that on any input, prints out w and then accepts.*

We can use this simple construct to get a self-printing Turing machine. Define a machine B such that given M, it print a machine P_M that prints M, and runs $M(M)$.

What happens when given any input w, it runs machine P_B to get the code of B, and feeds that to B. This prints exactly the same machine: given w, prints P_B, prints B, and runs $B(B)$. This is a machine that prints its own code.

10. 2/9

Theorem 10.1. *There is a computable function $q : \Sigma^* \to \Sigma^*$, where for any string w, $q(w)$ is the description of a Turing machine P_w that on any input, prints out w and then accepts.*

This is a building block we will use for self-reference.

We give a generic construction of a Turing machine that prints its own code. We find a machine B such that given input M, it prints out the following machine: given input w, it runs P_M, prints out M twice, and executes $M(M)$ by simulating on a universal Turing machine.

What happens when we have a machine that given input w, first runs P_B, and feeds its output B to B itself? The output of this machine is the same as the machine itself.

Suppose that in general we want to design a program that prints its own description. How? We want to say something like "Print this sentence." The problem is that there is no programming counterpart for "this". To get self-reference, we want something like this: "Print two copies of the following, the second copy in quotes: "Print two copies of the following, the second copy in quotes:"" The first part of this corresponds to B, and the second part corresponds to P_B.

Now, we can discuss the Recursion Theorem.

Theorem 10.2 (Recursion Theorem)**.** *Let T be a Turing machine that computes a function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. There is a Turing machine R that computes a function $r : \Sigma^* \to \Sigma^*$, where for every string w, $r(w) = t(R, w)$.*

In general, this means that we can assume that a Turing machine has access to its own code. Given any Turing machine t, we have a machine R outputting $t(R, w)$ on input w.

Proof. Suppose Turing machine T takes (a, b) as input and outputs $t(a, b)$.

Consider the machine B that takes input N as a description of a Turing machine, and outputs the machine: given input w, run P_N to get N, and output $N(N, w)$.

Let machine M take two inputs N and w. It feeds N to B and outputs some description Y. Then M runs and outputs $T(Y, w)$.

Here's the machine R: Given input w, run P_M to get machine M. Feed that into machine B to get some machine R. Finally, feed R and w into T to get $t(R, w)$.

We claim that the R output by B in the previous step is actually a description of the entire machine. What does R look like? This is the output of machine B on input M, so it takes input w, runs P_M to get M, and runs $M(M, w)$. But we know what M is, so here's actually machine R: Given input w, run P_M to get M, feed M, w to M, so give M to B to get Y, and output $T(Y, w)$. So this is a description of B.

Let's phrase this in another way:

Define $FOO_x(Y)$ to output x and halt.

Define $BAR(M)$ to output "$N(X)$ that runs FOO_M outputting M, and runs M on (M, x)."

Define $Q(N, x)$ to run $BAR(N)$ outputting E, and runs T on (E, x).

Then $R(X)$ is: Run FOO_Q outputting Q. Run $BAR(Q)$ outputting D. Run T on (D, x).

Let $D(X)$ be: Run FOO_Q outputting Q. Run $BAR(Q)$ outputting E. Run T on (E, x).

Then $D = R$, so $R(x) = T(R, x)$. □

The moral is that a Turing machine can obtain its own description, and compute with it.

Given any computable t, we can get a computable r such that $r(w) = t(R, w)$ where R is a description of r. We can use the operation "Obtain your own description" in Turing machine pseudocode.

Theorem 10.3. *A_{TM} is undecidable.*

Proof using the recursion theorem 10.2. Assume that H decides A_{TM}. Construct machine B such that on input w:

(1) Obtains its own description B.
(2) Runs H on (B, w) and flips the output.

Running B on input w always does the opposite of what H says that it should. □

Another application of the Recursion Theorem is the Fixed Point Theorem:

Theorem 10.4 (Fixed Point Theorem)**.** *Let $t : \Sigma^* \to \Sigma^*$ be any computable function. There is a Turing machine F such that $t(F)$ describes a Turing machine that is equivalent to F.*

Proof. Here is pseudocode for the Turing machine F. On input w:

(1) Obtain the description F.
(2) Let G be the output of $t(F)$ and interpret G as a Turing machine.
(3) Accept w if and only if $G(w)$ accepts.

□

We can use the Fixed Point Theorem to prove Rice's Theorem 9.6. Recall that this says that if L is a language over Turing machines that is nontrivial and semantic, then L is undecidable.

Proof of Rice's Theorem 9.6. Suppose we could decide L. Then define $t(M)$ so that if $M \in L$, output a Turing machine M_{no} such that $M_{\text{no}} \notin L$. Otherwise, output a Turing machine M_{yes} such that $M_{\text{yes}} \in L$.

For all Turing machines M, the function $t(M)$ always outputs a Turing machine that is not equivalent to M by the definition of semantic. This contradicts the fixed-point theorem 10.4! □

Now we will discuss computability in mathematics.

Definition 10.5. A *formal system* describes a formal language for

- writing mathematical statements
- has a definition of what statements are "true"
- a definition of a proof of a statement

Example 10.6. Any Turing machine M defines some formal system.

For example, think of all mathematical statements to be Σ^*. String w represents the statement "M accepts w". The set of true statements is $L(M)$.

A proof that "w is true" is an accepting computation history for M on w.

Definition 10.7. A formal system $\mathcal{F}$ is *consistent* or *sound* if no false statement has a valid proof in $\mathcal{F}$. (Proof implies truth.)

A formal system $\mathcal{F}$ is *complete* if every true statement has a valid proof in $\mathcal{F}$. (Truth implies proof.)

Finding a consistent and complete formal system was a dream of many mathematicians.

Theorem 10.8. *For every "interesting" formal system $\mathcal{F}$:*

- *(Gödel 1931) $\mathcal{F}$ is incomplete. There are true statements that cannot be proved.*
- *(Gödel 1931) The consistency of $\mathcal{F}$ cannot be proved using proofs in $\mathcal{F}$.*
- *(Church-Turing 1936) The problem of checking whether a given statement in $\mathcal{F}$ has a proof is undecidable.*

Definition 10.9. A formal system $\mathcal{F}$ is *interesting* if:

(1) Any mathematical statement describable in English can also be described within $\mathcal{F}$. Given M and w, there is an $S_{M,w}$ in $\mathcal{F}$ such that $S_{M,w}$ is true in $\mathcal{F}$ if and only if M accepts w.

(2) Proofs are convincing: It should be possible to check that a proof of a theorem is correct. Given (S, P), it is decidable if P is a proof of S in $\mathcal{F}$.

(3) If there is a proof of S that's describable in English, then there's a proof describable in $\mathcal{F}$. If M accepts w, then there is a proof in $\mathcal{F}$ on $S_{M,w}$.

Theorem 10.10 (Gödel 1931)**.** *Every consistent $\mathcal{F}$ is* incomplete*: there are true statements that cannot be proved.*

Proof. Let $S_{M,w}$ in $\mathcal{F}$ be true if and only if M accepts w.

Define Turing machine $G(x)$:

(1) Obtain own description G
(2) Construct statement $S' = \neg S_{G,\varepsilon}$
(3) Search for a proof of S' in $\mathcal{F}$ over all finite length strings. Accept if a proof is found.

We claim that S' is true but has no proof in $\mathcal{F}$. If there were such a proof, then it would be true that G accepts ε, which is a contradiction. S' basically says: "There is no proof for me in $\mathcal{F}$." □

Theorem 10.11 (Gödel 1931)**.** *The consistency of $\mathcal{F}$ cannot be proved within any interesing consistent $\mathcal{F}$.*

Proof. Suppose we can prove that "$\mathcal{F}$ is consistent" in $\mathcal{F}$.

We constructed $\neg S_{G,\varepsilon}$, which says that "G does not accept ε", which we showed is true, but has no proof in $\mathcal{F}$.

Here, G accepts ε if and only if there is a proof of $\neg S_{G,\varepsilon}$ within $\mathcal{F}$.

But if there's a proof in $\mathcal{F}$ of "$\mathcal{F}$ is consistent" then there's a proof in $\mathcal{F}$ that $\neg S_{G,\varepsilon}$ is true: "If $S_{G,\varepsilon}$ is true, then there is a proof in $\mathcal{F}$ of $\neg S_{G,\varepsilon}$. But $\mathcal{F}$ is consistent. Therefore $\neg S_{G,\varepsilon}$ is true. But $S_{G,\varepsilon}$ and $\neg S_{G,\varepsilon}$ cannot both be true. Therefore, $\neg S_{G,\varepsilon}$ is true." □

Finally, we want to show that for any interesting formal system, provability is undecidable:

Theorem 10.12 (Church-Turing 1936)**.** *For every interesting $\mathcal{F}$, let* $\text{PROVABLE}_{\mathcal{F}} = \{S :$ *there's a proof in $\mathcal{F}$ of S, or there's a proof in $\mathcal{F}$ of $\neg S$*$\}$*.*

Proof. Suppose $\text{PROVABLE}_{\mathcal{F}}$ is decidable with P. Then we can decide A_{TM} as follows:

On input (M, w), run the Turing machine P on input $S_{M,w}$. If P accepts, go through all possible proof sin $\mathcal{F}$. If you find a proof of $S_{M,w}$ then accept. If you find a proof of $\neg S_{M,w}$ then reject. If P rejects, then reject. □

11. 2/16

Turing machines are a universal notion of computation.

Kolmogorov complexity is motivated by: Is there a universal notion of information? Can we quantify how much information is contained in a string? Given two strings, which has more information? A string $01010101\ldots$ is very simple, but a string of seemingly random bits seems much more complicated.

The idea is: The more we can compress a string, the less information it contains. The amount of information in a string is the shortest way of describing that string. From the Church-Turing thesis, we should use Turing machines to describe inputs.

Definition 11.1. Let $x \in \{0,1\}^*$. The *shortest description of x*, denoted $d(x)$, is the lexicographically shortest string (M, w) such that $M(w)$ halts with x on its tape.

Theorem 11.2. *There is a 1-1 computable function $\langle \cdot, \cdot \rangle : \Sigma^* \times \Sigma^* \to \Sigma^*$ and computable functions $\pi_1, \pi_2 : \Sigma^* \to \Sigma^*$ such that $z = \langle M, w \rangle$ if and only if $\pi_1(z) = M$ and $\pi_2(z) = w$.*

The goal is to encode pairs of strings as another string.

Proof. Let $Z(x_1, x_2, \ldots, x_k) = 0x_1 0x_2 \ldots 0x_k 1$. Then we can define $\langle M, w \rangle = Z(M)w$.

To decode this pair, look at odd positions until we find an odd position with a 1. Then even positions to the left form the first string and everything to the right is the second string. Note that $|\langle M, w \rangle| = 2|M| + |w| + 1$. □

This is pretty good within constant factors. In some cases, we would like to do better.

Example 11.3. Here is a better pairing function. Let $b(n)$ be the binary encoding of n. Again, let $Z(x_1, x_2, \ldots, x_k) = 0x_1 0x_2 \ldots 0x_k 1$. Now, encode via $\langle M, w\rangle = Z(b(|M|))Mw$. For example, $\langle 10110, 101\rangle = 010001110110101$ because $b(|10110|) = 101$. Now, $|\langle M, w\rangle| = 2\log(|M|) + |M| + |w| + 1$, which is much better.

Definition 11.4. The *Kolmogorov complexity of* x, denoted as $K(x)$, is $|d(x)|$.

Theorem 11.5. *There is a* c *so that for all* x *in* $\{0,1\}^*$ *such that* $K(x) \leq |x| + c$.

"The amount of information in x isn't much more than $|x|$."

Proof. Define M so that on any input w, it halts. On any string x, $M(x)$ halts with x on its tape. This implies that $K(x) \leq |\langle M, x\rangle| \leq 2|M| + |x| + 1 \leq c + |x|$. □

Repetitive strings have low information.

Theorem 11.6. *There is a* c *so that for all* x *in* $\{0,1\}^*$ *such that* $K(xx) \leq |x| + c$.

"The amount of information in xx isn't much more than that in x."

Proof. Define N so that on input $\langle M, w\rangle$, let $s = M(w)$. Print ss. Let $\langle M, w\rangle$ be the shortest description of x. Then $\langle N, \langle M, w\rangle\rangle$ is a description of xx. Therefore, $K(xx) \leq |\langle N, \langle M, x\rangle\rangle| \leq 2|N| + K(x) + 1 \leq c + |x|$. □

Corollary 11.7. *There is a fixed* c *so that for all* $n \geq 2$ *and all* $x \in \{0,1\}^*$, $K(x^n) \leq K(x) + c\log n$.

"The information in x^n isn't much more than that in x."

Proof. Define the Turing machine N taking input $\langle n, M, w\rangle$. Let $x = M(w)$. Print x for n times.

If $\langle M, w\rangle$ is the shortest description of x, then $K(x^n) \leq K(\langle N, \langle n, M, x\rangle\rangle) \leq 2|N| + d\log n + K(x) \leq c\log n + K(x)$ for some constants c and d. □

Example 11.8. If $w = (01)^n$ then $K((01)^n) \leq O(\log n)$.

Does the model matter? Turing machines are one programming language. If we use other programming languages, could we get significantly shortest descriptions?

Definition 11.9. An *interpreter* is a semi-computable function $p : \Sigma^* \to \Sigma^*$ that takes programs as input and prints their outputs.

Definition 11.10. Let $x \in \{0,1\}^*$. The *shortest description of* x *under* p (called $d_p(x)$) is the lexicographically shortest string for which $p(d_p(x)) = x$.

Define $K_p(x) = |d_p(x)|$.

Theorem 11.11. *For every interpreter* p*, there is some constant* c *so that for all* $x \in \{0,1\}^*$, $K(x) \leq K_p(x) + c$.

The moral is that using any other programming language would only change $K(x)$ by some constant.

Proof. Define machine M so that on w it outputs $p(w)$. Then $\langle M, d_p(x)\rangle$ is a description of x, and $K(x) \leq |\langle M, d_p(x)\rangle| \leq 2|M| + K_p(x) + 1 \leq c + K_p(x)$. □

What are the limits of compression? There are incompressible strings of every length!

Theorem 11.12. *For all n, there is an $x \in \{0,1\}^n$ such that $K(x) \geq n$.*

Proof. This is a counting argument.

The number of binary strings of length n is 2^n, and the number of descriptions of length $< n$ is at most the number of binary strings of length $< n$, which is $1+2+4+\cdots+2^{n-1} = 2^n - 1$.

Therefore, there's at least one n-bit sting that does not have a description of length $< n$. □

In turns out that most strings are very incompressible.

Theorem 11.13. *For all n and c, $\Pr_{x\in\{0,1\}^n}[K(x) \geq n - c] \geq 1 - \frac{1}{2^c}$.*

Proof. This is another counting argument.

The number of binary strings of length n is 2^n, and the number of descriptions of length $< n - c$ is at most the number of binary strings of length $< n - c$, which is $2^{n-c} - 1$.

So the probability that a *random* x satisfies $K(x) \leq n-c$ is at most $(2^{n-c}-1)/2^n < \frac{1}{2^c}$. □

The problem is that it seems hard to figure if a string is compressible. It is hard to give an algorithm to do optimal compression. Define COMPRESS $= \{(x, c) : K(x) \leq c\}$.

Theorem 11.14. COMPRESS *is undecidable.*

The intuition is that if this were decidable, we could design an algorithm that prints the least incompressible string of length n. But such a string could be succinctly described, by giving the algorithm, and n in binary.

This relates to the Berry paradox: "The smallest integer that cannot be defined in less than thirteen words."

Proof. Assume that COMPRESS has a decider. Define machine M so that on input $x \in \{0,1\}^*$, we interpret x as an integer N. (Then $|x| \leq \log N$.) For all $y \in \{0,1\}^*$ in lexicographical order, if $(y, N) \notin$ COMPRESS then print y and halt.

$M(x)$ prints the shortest string y' with $K(y') > N$. But $\langle M, x\rangle$ describes y' and $|\langle M, x\rangle| \leq c + \log N$. So $N < K(y') \leq c + \log N$, which is a contradiction. □

Theorem 11.15. *A_{TM} is undecidable.*

Proof. We reduce from COMPRESS to A_{TM}.

Given a pair (x, c), construct a Turing machine $M_{x,c}$: Over all pairs $\langle M', w\rangle$ with $|\langle M', w\rangle| \leq c$, simulate each M' on w in parallel. If some M' halts and prints x, then accept.

$K(x) \leq c$ if and only if $M_{x,c}$ accepts ε. □

Recall that a formal system $\mathcal{F}$ is interesting if

(1) Any mathematical statement describable in English can also be described within $\mathcal{F}$.
(2) Proofs are convincing: it should be possible to check that a proof of a theorem is correct.
(3) If there is a proof of S then there's a proof describable in $\mathcal{F}$.

Theorem 11.16. *For every interesting consistent $\mathcal{F}$, there is a t such that "$K(x) > t$" is unprovable in $\mathcal{F}$.*

Proof. Define a Turing machine M that given input k, searches over all strings x and proofs P for a proof P in $\mathcal{F}$ that $K(x) > k$. Output x if found.

Suppose $M(k)$ halts with output x'. Then $K(x') = K(\langle M, k\rangle) \leq c + \log k$ for some c. Because $\mathcal{F}$ is consistent, $K(x') > k$ is true.

But $k < c + \log k$ only holds for finitely many k. Choose t to be greater than all of these k. Then $M(t)$ cannot halt, so "$K(x) > t$" has no proof. □

But for a randomly chosen x of length $t + 100$, we know that "$K(x) > t$" is true with probability at least $1 - 1/2^{100}$. We can randomly generate true statements in $\mathcal{F}$ which have no proof in $\mathcal{F}$ with high probability.

12. 2/21

Complexity theory is the study of what can or cannot be computed with limited computational resources, such as time or space. We start with time complexity.

We measure time complexity by counting the elementary steps required for a machine to halt.

Example 12.1. Consider the language $A = \{0^k1^k \mid k \geq 0\}$. Here is a Turing machine that decides this language:

On input of length n:

(1) Scan across the tape and reject if the string is not of the form 0^i1^j.
(2) Repeat the following if both 0s and 1s remain on the tape. Scan across the tape, crossing off a single 0 and a single 1.
(3) If 0s remain after all 1s have been crossed off, or vice-versa, reject. Otherwise, accept.

Let M be a Turing machine that halts on all inputs.

Definition 12.2. The *running time* or *time complexity* of M is the function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the maximum number of steps taken by M over any input of length n.

Definition 12.3.

$$\begin{aligned}\text{TIME}(t(n)) &= \{L' \mid \text{there is a TM } M \text{ with TIMEcomplexity } O(t(n)) \text{ so that } L' = L(M)\} \\ &= \{L' \mid L' \text{ us a language decided by a TM with } O(t(n)) \text{ running TIME}\}\end{aligned}$$

Example 12.4. We showed that $A = \{0^k1^k \mid k \geq 0\} \in \text{TIME}(n^2)$.

Actually, $A \in \text{TIME}(n \log n)$. Define the machine M taking input w. If w is not of the form 0^*1^*, reject. Repeat the following until all bits of w are crossed out: If the parity of 0s does not equal the parity of 1s, reject. Cross out every other 0. Cross out every other 1.

Theorem 12.5. *$A = \{0^k1^k \mid k \geq 0\}$ can be decided in $O(n)$ time with a two-tape Turing machine.*

Remark. But this cannot be done with a one-tape machine.

Proof. Scan all 0s, copy them to the second tape.

Scan all 1s. For each 1 scanned, cross off a 0 from the second tape. □

Different models of computation can yield different running times for the same language!

Theorem 12.6. *Let $t(n)$ be a function such that $t(n) \geq n$. Then every $t(n)$ time multi-tape Turing machine has an equivalent $O(t(n)^2)$ single tape Turing machine.*

Proof. Our simulation of multitape Turing machines achieved this! □

If you get more time to work, then you can solve strictly more problems.

Theorem 12.7 (Time Hierarchy Theorem)**.** *For all "reasonable" functions $f, g : \mathbb{N} \to \mathbb{N}$ where $f(n) = O(g(n)^{1/3})$, $\text{TIME}(f(n)) \subsetneq \text{TIME}(g(n))$.*

Proof. Make a Turing machine N that "does the opposite" of all $g(n)^{1/2}$ time machines on at least one input, and runs in $O(g(n))$ time.

Define a machine N taking input w: If w does not have the form $M10^i$, reject. Simulate M on w for $f(|w|)$ steps. (This simulation itself uses $f(|w|)^2$.)

If M tries to take more time, reject. Accept w if and only if M rejects w.

Eventually, w is large enough that $f(|w|) < g(|w|)$. Simulation can be carried out using a multitape simulation. We need to compute $f(|w|)$ using only $O(g(|w|))$ time. □

Definition 12.8. A function $f : \mathbb{N} \to \mathbb{N}$ is time-constructible if the function $g : \{0,1\}^* \to \{0,1\}^*$ with $g(1^n)$ defined as "$f(n)$ in binary" is computable in $f(n)$ time.

We also have that for all k and for all $\varepsilon > 0$, we have $\text{TIME}(n^k) \subsetneq \text{TIME}(n^{k+\varepsilon})$.

There is an infinite hierarchy of increasingly more time-consuming problems.

Are there important everyday problems that are high up in the hierarchy? A natural problem that needs exactly n^{10} time? This is an open question.

Definition 12.9.

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

We should have an extended Church-Turing thesis: Everyone's intuitive notion of efficient algorithms corresponds to polynomial-time Turing machines.

This us much more controversial than the Church-Turing thesis. Potential counterexamples include n^{100} time algorithms, quantum algorithms, and randomized algorithms.

Definition 12.10. Nondeterministic Turing machines are just like standard Turing machines, except:

(1) The machine may proceed according to several possibilities.
(2) The machine accepts a string if there exists a path from the start state to the accepting state.

Similarly, we can define accepting computation histories for N on w. Note that these do not have to be unique.

Definition 12.11.

$$\text{NTIME}(t(n)) = \{L \mid L \text{ is decided by a } O(t(n)) \text{ time nondeterministic TM}\}.$$

Note that $\text{TIME}(t(n)) \subseteq \text{NTIME}(t(n))$. Is this a proper containment for all $t(n) = n^k$? This is an open question!

We briefly review Boolean formulas.

Definition 12.12. A *satisfying assignment* is a setting of the variables that makes the formula true.

Example 12.13. For example, $\varphi = (\neg x \wedge y) \vee z$. Then $x = y = z = 1$ is a satisfying assignment for φ.

Definition 12.14. A Boolean formula is *satisfiable* if there exists a satisfying assignment for it.

Example 12.15. $a \wedge b \wedge c \wedge \neg d$ is satisfiable.
$\neg(x \vee y) \wedge x$ is unsatisfiable.

Definition 12.16. SAT $= \{\varphi \mid \varphi$ is a satisfiable Boolean formula$\}$.

Definition 12.17. A 3cnf-formula is of the form

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

3SAT $= \{\varphi \mid \varphi$ is a satisfiable 3-cnf formula$\}$.

Theorem 12.18. 3SAT $\in$ NTIME(n^2).

Proof. On input φ:

(1) Check if the formula is in 3cnf.
(2) For each variable, nondeterministically substitute it with 0 or 1.
(3) Test if the assignment satisfies φ. □

Definition 12.19.

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

Theorem 12.20. *$L \in$ NP if and only if there exists a polynomial time Turing machine V and k such that we can define L as*

$$L = \{x \mid \exists y[|y| \leq |x|^k \text{ and } V(x, y) \text{ accepts}]\}$$

Proof.

(1) If $L = \{x \mid \exists y[|y| \leq |x|^k$ and $V(x, y)$ accepts$]\}$ then $L \in$ NP.
Just nondeterministically guess y and then run $V(x, y)$.
(2) If $L \in$ NP then $L = \{x \mid \exists y[|y| \leq |x|^k$ and $V(x, y)$ accepts$]\}$.
Let N be a nondeterministic polynomial-time Turing machine that decides L. Define $V(x, y)$ to accept if and only if y is an accepting computation history of N on x. □

A language is in NP if and only if there are polynomial-length proofs for membership in the language.
SAT is in NP for this reason.

Example 12.21 (Hamiltonian path problem)**.** Given a directed graph with modes, a *Hamiltonian path* traverses through each node exactly once.
Assume a reasonable encoding of graphs (e.g. the adjacency matrix is reasonable).

HAMPATH $= \{(G, s, t) \mid G$ is a directed graph with a Hamiltonian path from s to $t.\}$

Theorem 12.22. HAMPATH $\in$ NP

13. 2/23

Today we will discuss NP-completeness and Cook's Theorem. NP-complete problems are the hardest problems in NP.

Recall that a language is in NP if and only if there are polynomial-length proofs for membership in the language. For example, SAT is in NP. It is hard to prove that something is unsatisfiable, but proving that something is satisfiable only requires given an example.

Example 13.1 (k-clique problem)**.** Given a graph and a number k, is there a complete subgraph on k nodes? Define

$$\text{CLIQUE} = \{(G, k) : G \text{ is an undirected graph with a } k\text{-clique}\}.$$

Theorem 13.2. CLIQUE $\in$ NP.

This is because a k-clique itself is a proof that (G, k) is in CLIQUE.

The big open question in complexity theory is:

Problem 13.3.

$$\text{P} = \text{NP?}$$

"Can problem solving be efficiently automated?"

What would happen if P = NP?

- Mathematicians may be out of a job. We can define

  $$\text{Short-Provability}_F = \{T \mid T \text{ has a proof in } F \text{ of length} \leq |T|^2\}.$$

  This is in NP, and P=NP would make this easy.
- Cryptography as we know it would probably be impossible, since it depends on one-way functions.
- In principle, every aspect of daily life could be efficiently and globally optimized.

This is all too good to be true. We conjecture that P $\neq$ NP.

Definition 13.4. $f : \Sigma^* \to \Sigma^*$ is a polynomial time computable function if some polynomial-time Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Definition 13.5. Language A is polynomial time reducible to language B, written $A \leq_P B$, if there is a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that $w \in A$ if and only if $f(w) \in B$.

Note that $|f(w)| \leq |w|^k$ for some constant k.

Theorem 13.6. *If $A \leq_P B$ and $B \leq_P C$ then $A \leq_P C$.*

Proof. A polynomial of a polynomial is still a polynomial. □

Theorem 13.7. *If $A \leq_P B$ and $B \in$ P then $A \in$ P.*

Proof. Let M_B is a polynomial-time Turing machine that decides B. Let f be a polynomial-time reduction from A to B.

We build a machine M_A that decides A as follows: On input w, compute $f(w)$, run M_B on $f(w)$, and output its answer. □

Corollary 13.8. *If $A \leq_P B$ and $A \notin$ P then $B \notin$ P.*

Definition 13.9. A language B is NP-complete if:

(1) $B \in \text{NP}$
(2) Every A in NP is polynomial-time reducible to B. That is, $A \leq_P B$.

Suppose B is NP-complete. If $B \in \text{P}$ then $\text{P} = \text{NP}$. Then assuming the conjecture that $\text{P} \neq \text{NP}$, this means that such a problem B is not solvable in $O(n^k)$ time, for any k.

Theorem 13.10 (Cook-Levin)**.** SAT *and* 3SAT *are NP-complete.*

Corollary 13.11. 3SAT *is in* P *if and only if* $\text{P} = \text{NP}$.

Proof. We know that $\text{3SAT} \in \text{NP}$. We will show that every language A in NP is polynomial time reducible to 3SAT.

We give a polynomial time reduction for A to 3SAT. The reduction turns a string w into a 3cnf formula φ such that $w \in A$ if and only if $\varphi \in \text{3SAT}$.

Definition 13.12. A *tableau* for N on w is an $n^k \times n^k$ table whose rows are the configurations of some possible computation of N on input w.

The first row is the starting configuration of some Turing machine, and the next row is some configuration that can be reached in one step (not necessarily unique). A tableau is *accepting* if any row of the tableau is an accepting configuration. A machine N accepts w if and only if there is an accepting tableau for N on w.

Given a string w, our Boolean 3cnf formula φ will describe all of the logical constraints that any accepting tableau for N on w must satisfy. The 3cnf formula φ will be satisfiable if and only if there is an accepting tableau.

Let $C = Q \cup \Gamma \cup \{\#\}$. Each of the $(n^k)^2$ entries of a tableau is a cell. Then $\text{cell}[i, j]$ is the value of the cell at row i and column j, or the jth symbol in the ith configuration. For every i and j $(1 \leq i, j \leq n^k)$ and for every $s \in C$ we have a variable $x_{i,j,s}$. The total number of variables is $|C|n^{2k}$, which is $O(n^{2k})$. These $x_{i,j,s}$ are the variables of φ and represent the contents of the cells. We will have $x_{i,j,s} = 1$ if and only if $\text{cell}[i, j] = s$.

We now design φ so that a satisfying assignment to the variables $x_{i,j,s}$ corresponds to an accepting tableau for N on w (an assignment to the $\text{cell}[i, j]$).

The formula φ will be the and of four cnf formulas: $\varphi = \varphi_{\text{cell}} \wedge \varphi_{\text{start}} \wedge \varphi_{rmaccept} \wedge \varphi_{\text{move}}$.

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

We can define

$$\varphi_{\text{cell}} = \bigwedge_{1 \leq i \leq j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigvee_{s,t \in C, s \neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}) \right) \right],$$

$$\varphi_{\text{start}} = x_{1,1,\#} \wedge x_{1,2,q_0}, \wedge \cdots \wedge x_{1,n^k,\#}$$

$$\varphi_{\text{accept}} = \bigvee_{1 \leq j \leq n^k} x_{n^k,j,q_{\text{accept}}}.$$

φ_{move} checks that every 2×3 window of cells is legal.

Lemma 13.13. *If the top row of the tableau is the start configuration and every window of the tableau is legal, then each row of the table yields the next row of the table.*

Proof. Very similar to the PCP problem's undecidability. □

Write the (i, j) window of a tableau as the tuple $(a_1, \ldots, a_6)$. Then

$$\varphi_{\text{move}} \bigwedge_{\substack{1\le i\le n^k-1\\ 1\le j\le n^k-2}} \left(\bigwedge_{(a_1,\ldots,a_6) \text{ is not legal}} \neg x_{i,j,a_1} \vee \neg x_{i,j+1,a_2}, \vee x_{i,j+2,a_3} \vee x_{i+1,l,a+4} \vee \cdots \right).$$

Now, we need to turn this into a 3cnf. Everything was ands or ors. We just need to make the ors small: $(a_1 \vee \cdots a_t) = (a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee z_2) \wedge (\neg z_2 \vee a_4 \vee z_3) \cdots$.

What's the total length of φ? φ_{cell} has $O(n^{2k})$ clauses. φ_{start} has $O(n^k)$ clauses. φ_{accept} has $O(n^k)$ clauses. φ_{move} has $O(n^{2k})$ clauses. □

Read Luca's notes for an alternative proof of the Cook-Levin theorem. The sketch of this is that for every deterministic n^k time $V(x, y)$, we define CIRCUIT-SAT: Given a logical circuit $C(y)$, is there an input A such that $C(A) = 1$. Show that CIRCUIT-SAT is NP-hard. The $n^k \times n^k$ tableau can be simulated with a logical circuit of $O(n^{2k})$ gates. Reduce CIRCUIT-SAT to 3SAT in polynomial time, and conclude that 3SAT is also NP-hard.

Is 3SAT solvable in $O(n)$ time on a multitape Turing machine? If yes, then not only is P=NP true, but we would have a machine that could crank out short proofs of theorems. This is an open question.

14. 2/28

In the previous lecture, we saw one example of NP-complete problems. Today. we'll see more.

Given a problem in $\Pi \in$ NP, how can we prove it is NP-hard? There is a general recipe:

(1) Take a problem Σ that you know to be NP-hard (e.g. 3SAT).
(2) Prove that $\Sigma \le_P \Pi$.

Then for all $A \in$ NP, $A \le_P \Sigma$ and $\Sigma \le_P \Pi$. We conclude that $A \le_P \Pi$.

Recall the k-clique problem, and define

$$\text{CLIQUE} = \{(G, k) : G \text{ is an undirected graph with a } k\text{-clique}\}.$$

Note that $(G, k) \in$ CLIQUE implies that $(G, k-1) \in$ CLIQUE, for $k \ge 1$.

Theorem 14.1. *CLIQUE is NP-complete.*

Proof. First, observe that CLIQUE $\in$ NP because we can verify it.

Now, we will show that 3SAT $\le_P$ CLIQUE. To do this, we want to transform a 3-cnf formula φ into (G, k) such that $\varphi \in$ 3SAT if and only if $(G, k) \in$ CLIQUE. We want this transformation that can be done in time that is polynomial in the length of φ. How can we encode a logic problem as a graph problem?

Let m be the number of clauses of φ. Set $k = m$ and make a graph G with m clusters of up to 3 nodes each. Each cluster corresponds to a clause of φ. Each node in a cluster is labeled with a literal from the clause. We put edges between all pairs of nodes in different clusters, except pairs of the form $\{x, \neg x\}$. We do not connect any nodes in the same cluster.

A clique is a way to pick a literal for each clause in a consistent way. This then corresponds to a satisfying assignment. If there are k clauses, we look for a k-clique.

We will show that $\varphi \in$ 3SAT if and only if $(G, m) \in$ CLIQUE.

Lemma 14.2. *If $\varphi \in$ 3SAT then $(G, m) \in$ CLIQUE.*

Proof. Given a SAT assignment A of φ, for every clause C there is at least one literal set true by A. Let v_C be the vertex in G corresponding to the first literal of C satisfied by A.

We claim that $S = \{v_C : C \in \varphi\}$ is an m-clique. If there is not an edge between v_C and $v_{C'}$, then v_C and $v_{C'}$ must label inconsistent literals x and $\neg x$. But A cannot satisfy both x and $\neg x$. Therefore, $(v_C, v_{C'}) \in E$ for all $v_C, v_{C'} \in S$, and hence we've proven the claim. □

Lemma 14.3. *If $(G, m) \in$ CLIQUE then $\varphi \in$ 3SAT.*

Proof. Given an m-clique S of G, we'll construct a SAT assignment A of φ.

We claim that S has exactly one node for each cluster. For each variable x of φ, we make an assignment A. Set x to be true if and only if there is a $v \in S$ with label S.

For every clause C, one vertex from the cluster for C is in S. Therefore, A satisfies at least one literal of C, so C is satisfied by A. □

This proves the theorem. □

A related problem is *independent set*. Given a graph $G = (V, E)$ and integer k, is there $S \subseteq V$ such that $|S| \geq k$ and no two vertices in S have an edge.

Proposition 14.4. *CLIQUE $\leq_P$ IS.*

Proof. Given a graph $G = (V, E)$, output $G' = (V, E')$ where $E' = \{(u, v) : (u, v) \notin E\}$.

Note that $(G, k) \in$ CLIQUE if and only if $(G', k) \in$ IS. □

Another problem about graphs is *vertex cover*. This is a set of nodes that covers all edges. Formally, define

$$\text{VERTEX-COVER} = \{(G, k) : G \text{ is an undirected graph with a vertex cover of size } k\}.$$

First, VERTEX-COVER is in NP because it is easy to verify, by looping over all of the edges.

Proposition 14.5. *IS $\leq_P$ VERTEX-COVER*

Proof. We claim that for every graph $G = (V, E)$, $S \subseteq V$ is an independent set if and only if $V \setminus S$ is a vertex cover.

To see this, note that S is independent if and only if for any vertices $u, v \in S$ then $(u, v) \notin E$, or contrapostively, if $(u, v) \in E$ then either $u \notin S$ or $v \notin S$. This means that $V \setminus S$ is a vertex cover.

So therefore $(G, k) \in$ IS if and only if $(G, |V| - k) \in$ VERTEX-COVER. □

Now, consider the *subset sum* problem. Given a set $N = \{a_1, a_2, \ldots, a_n\}$ of positive integers, and a positive integer t, is there some S such that $\sum_{i \in S} a_i = t$?

Define

$$\text{SUBSET-SUM} = \{(N, t) : \text{there exists } S \text{ such that } \sum_{i \in S} a_i = t\}$$

Theorem 14.6. *There is an algorithm for solving SUBSET-SUM in time polynomial in n and t.*

But t can be specified in $\log t$ bits, so this is not an algorithm that runs in polynomial time in the input.

Proof. Use dynamic programming. □

Proposition 14.7. *VERTEX-COVER $\leq_P$ SUBSET-SUM.*

Proof. We want to reduce a graph to a set of numbers.

Given (G, k), let $E = \{e_0, \ldots, e_{m-1}\}$ and $V = \{1, \ldots, n\}$. Our (N, t) will have $|N| = n+m$. For every $e_i \in E$, put in N the integer $b_i = 4^i$. For every $i \in V$, put in N the integer $a_i = 4^m + \sum_{k:\{i,k\}\in E} 4^i$. Set $t = k \cdot 4^m + \sum_{i=0}^{m-1}(2 \cdot 4^i)$.

We claim that $(G, k) \in$ VERTEX-COVER then $(N, t) \in$ SUBSET-SUM. Suppose that $C \subseteq V$ is a vertex cover with k vertices. Let $S = \{a_i : i \in C\} \cup \{b_i : |e_i \cap C| = 1\}$. We claim that $\sum_{x\in S} x = t$. Think of the numbers as being in base 4, as vectors with m components. Just think about it.

We also have to do the other direction. Suppose $S \subseteq V$ and $T \subseteq E$ is such that $\sum_{i\in S} a_i + \sum_{e_i\in T} b_i = t$. We claim that S is a vertex cover of size at most k. As before, just think about it. □

15. 3/1

We have been talking about NP-complete problems. We've seen Cook's theorem, giving reductions from any NP problem to SAT and 3SAT, which then reduce to CLIQUE, INDEPENDENT SET, VERTEX COVER, and from there to SUBSET SUM. Today, we will see reductions to STEINER TREE and PARTITION, and from there to BIN PACKING and MULTI PROCESSOR SCHEDULING.

There are algorithms to solve SAT that can succeed on inputs of several thousand clauses long; they run faster on some inputs and slower on others. There is no algorithm that can run at a fixed time for all inputs though, and finding such an algorithm would mean that all of the problems that we have been considering would be easy to solve.

Note that in addition to the reductions mentioned above, there are also reductions in the other direction, since each of them are NP-complete. Even though the other problems look very different, they are all almost the same, up to polynomial time reduction.

Recall the subset sum problem from last time:

Definition 15.1 (Subset sum)**.** The input is $a_1, a_2, \ldots, a_n$ and t. The goal is to find some subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i\in S} a_i = t$.

This has a nice dynamical programming solution, but it takes exponential time.

There is a special case of subset sum that it still as hard as the general case.

Definition 15.2 (Partition)**.** The input is $a_1, \ldots, a_n$. The goal is to find $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i\in S} a_i = \sum_{i\notin S} a_i$.

We want to show that this is NP-hard, so we write down a reduction.

Proposition 15.3. *Subset sum $\leq_P$ Partition.*

Proof. Given $a_1, \ldots, a_n$, t, we want to construct $a'_1, \ldots, a'_m$ such that there exists S with $\sum_{i \in S} a_i = t$ if and only if there exists S' with $\sum_{i \in S} a'_i = \sum_{i \notin S} a'_i$.

Here's a construction that doesn't work: Let the new elements be $a_1, \ldots, a_n, a_{n+1} = t, a_{n+2} = A - t$ where $A = \sum a_i$. If there is a solution for subset sum on the original input, all of the rest of the items will add up to $A - t$. The old question is: Is there a way to partition the input into sets adding up to t and $A - t$. But since we added two items t and $A - t$, we can complete each set to have size A. This becomes an instance of the partition problem. In the other direction, for every solution of the partition problem, if the two new items are not in the same set, then we clearly get a solution of the subset sum problem (just by removing the extra items). But what if the two new items are put together? This does not solve subset sum! So we don't have a reduction in the reverse direction. This is a common mistake in reductions, and it yields a reduction that doesn't work.

Now, we write down a reduction that does work. The new elements are $a_1, \ldots, a_n, a_{n+1} = A + t$ and $a_{n+2} = 2A - t$. With this definition, if there is a solution for subset sum, we can complete it with the new elements to get a solution for partition. And if we have a partition solution where the two new elements don't appear on the same side, then we remove them to get a solution of subset sum. And this time, the new elements are too big to appear together, so nothing breaks. This gives us the reduction that we wanted. □

Now we discuss the bin packing problem.

Definition 15.4 (Bin packing)**.** The input is an integer B known as the "bin size". Then we have n items, of size $s_1, \ldots, s_n$. The goal is to fit the items in as few bins as possible, so that the size of items in each bin is $\leq B$.

This is the most basic version, with one-dimensional items. To make this optimization problem into a decision problem, add an input integer k. We ask if it is possible to fit the items into at most k bins.

If we have an instance of partition $a_1, \ldots, a_n$, then we can construct an instance of bin packing with $k = 2$ and $B = \frac{1}{2} \sum_{i=1}^{n} a_i$, where the sizes are $a_1, \ldots, a_n$.

There are lots of scheduling problems. As an example, we look at one of the simpler versions.

Definition 15.5 (Scheduling)**.** The input is the number of identical machines m. We also have n tasks, with running time $t_1, \ldots, t_n$; these tasks have no dependencies. The goal is to minimize completion time.

To make this into a decision problem, and an input T. Is it possible to schedule so that the completion time is $\leq T$?

This is exactly the same problem as bin packing. The number of bins is the number of machines. The size of the bins is the completion time, and the sizes of items is the time of the tasks. They are different optimizations, but they look like the same decision problem.

We will finish with the Steiner tree problem, which has a much more interesting reduction from vertex cover.

The original problem, as studied by Steiner, looks like this: Given some points in the plane, we want to connect the points and have minimal total length. It clearly suffices to consider a tree, which is the smallest thing preserving connectivity. But even for four points, the minimal spanning tree is not the right answer. Instead, we want two vertices in the

middle of the rectangle, connecting to the four vertices. This is a solved problem in the plane for three and four points, but it becomes a hard problem when the number of points is large.

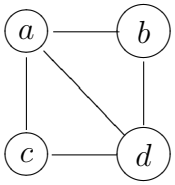
Here is a more abstract version of the problem. In addition to the input points, think about having a grid of extra points that we can use as vertices. Now, we have a symmetric matrix of connection costs.

Definition 15.6 (Steiner Tree)**.** As input, we have a set of points X consisting of required points R and Steiner points S which can also be used. We have a distance function $d(x, y)$ for each $x, y \in X$. The goal is to find a tree $T = (V, E)$ where $R \subseteq V \subseteq X$ and $\sum_{(u,v)\in E} d(u, v)$ is minimized.

We also expect that our distance function is a metric.

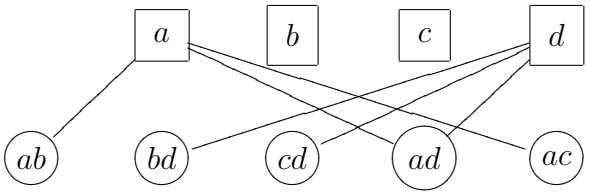
We will now reduce from vertex cover to Steiner tree. In terms of similarity, both have a notion of points, connectivity, and graphs. Also, both problems have hard constraints. In vertex cover, we have pick any subset of vertices, but we must cover all of the edges. In Steiner tree, we can add however many Steiner vertices we like, but we must include all required points. This suggests that edges in the vertex cover problem should become required points in the Steiner tree problem.

Example 15.7. Consider



This gives required points ab, bd, cd, ad, ac. There are also Steiner vertices a, b, c, d. A valid vertex cover is a, d.

We connect all Steiner points to the required points (edge in vertex cover) using that vertex.



We only need the edges corresponding to the Steiner points in the vertex cover, which requires 6 connections.

To make sure this works, we should also add cost 1 connections between Steiner points.

Here is the reduction. Start with an instance of vertex cover $G = (V, E)$. We construct a Steiner tree instance where the $R = E$ and $S = V$. Then $d(u, (u, v)) = 1$, and $d(u, v) = 1$. All other distances must be defined to satisfy the metric property, and are defined by shortest paths; in fact, all distances will be 1, 2, 3.

We claim that G has a vertex cover of size $\leq k$ if and only if the Steiner tree instance has solution of cost $\leq |E| + k - 1$.

Suppose that G has a vertex cover $C \subseteq V$ of size k. Using only Steiner points corresponding to the vertices in the vertex cover, all required points have a connection to one of these vertices, and all relevant Steiner points are connected. That is, the vertices are $E \cup C$, with

all length 1 edges, and it is connected. Therefore, it has a spanning tree. This is a Steiner tree for the network, connecting all required vertices, plus some Steiner points. The number of edges in this tree is $|E| + |C| - 1 = |E| + k - 1$, which is a valid Steiner tree solution.

Now, we have to show that given any Steiner tree solution, we can still get back a vertex cover solution. Suppose we are given a Steiner tree solution of cost $|E| + k - 1$. We want to use only cost 1 connections, so replace every connection of cost ≥ 2 by a path of length c using only cost-1 connections. In doing that, we do not increase the cost of the solution. At the end of this, we get a connected graph that includes all of E and has only length 1 edges, take a spanning tree of this graph, which still has cost $\leq |E| + k - 1$. This tree has at most $|E| + k$ vertices and includes all $|E|$ required vertices, so it has $\leq k$ Steiner vertices. We claim that those k vertices must be a vertex cover. This is because all required points are connected to the rest of the tree, and there are no longer any edges between required points (since that has length > 1). Therefore, all required points must be connected to one of our Steiner points. This shows the desired result.

16. 3/6

Today we will start a new topic: complexity of space (memory) restricted algorithms. Sometimes, memory can be an even more critical resource than time when considering the feasibility of a problem. We will be able to classify problems according to these restrictions, and we will have a sense of completeness like that of NP-completeness.

Definition 16.1. A language L is decidable in space complexity $s(n)$ if there is a Turing machine M that decides L and on every input x of length n accesses at most $s(n)$ cells of the tape.

Just as for time complexity, we want space consumption to grow polynomially as the size of the input. This forms a reasonable notion of efficiency.

Definition 16.2. PSPACE $= \{L : L$ is decidable in space $\leq p(n)$ where p is polynomial$\}$

Fact 16.3. PSPACE contains the class NP. This is because we can enumerate all strings of polynomial length and check if each is in the language. Since verification takes polynomial time, it can also only polynomial space.

Suppose we are given a Boolean formula $(x_1 \vee \overline{x}_2) \wedge (x_2 \vee x_3) \wedge (\overline{x}_3 \vee x_1)$. Two players take turns setting values for the Boolean variables. At the end, the first player wins in the formula is true; otherwise, the second player wins. In fact, in this example, there is a forced win for the first player by setting x_1 to be true. This means that $\exists x_1 \forall x_2 \exists x_3 (x_1 \vee \overline{x}_2) \wedge (x_2 \vee x_3) \wedge (\overline{x}_3 \vee x_1)$.

Problem 16.4. In general, this is called the Quantified Boolean Formula problem. Given input

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \varphi$$

for some Boolean formula φ, the goal is to decide if the quantified formula is true.

Remark. Since we can rename the variables, we can actually process the variables in any order Also, we can also mix $\forall$ and $\exists$ in any order, since we can just introduce extra variables that don't appear in the formula.

Proposition 16.5. *QBF is in PSPACE.*

Proof. The polynomial space algorithm will solve this problem by brute force. It is easiest to think of it as a recursive algorithm V.

Given input $Q_1x_1Q_2x_2\dots Q_nx_n\varphi(\cdots)$ for quantifiers $Q_1,\dots,Q_n$, the algorithm computes

$$b_1 := V(Q_2x_2\dots Q_nx_n\varphi(x_1 \leftarrow F\dots))$$
$$b_2 := V(Q_2x_2\dots Q_nx_n\varphi(x_1 \leftarrow T\dots)).$$

If $Q_1 = \exists$, we accept if and only if $b_1 \vee b_2$. Otherwise, we accept if and only if $b_1 \wedge b_2$.

Here, $s(n) = s(n-1) + p(n)$, so $s(n) = np(n)$ is polynomial. This is extremely time inefficient but is space efficient, because we can reuse space but we cannot reuse time. □

Definition 16.6. $\text{EXP} = \text{TIME}(2^{n^{o(1)}})$.

We claim that PSPACE is contained in EXP.

Fact 16.7. If machine M halts on input x and uses $\le s(n)$ space, then it halts in $\le 2^{O(s(n))}$ steps.

Proof. Consider all configurations of the machine M on input x. A configuration can be represented using $O(s(n)) + O(\log s(n)) + O(1) = O(s(n))$ bits. This means that the total number of configurations is at most $2^{O(s(n))}$. The halting computation of M on input x is a sequence of configurations which must be all different (since otherwise the machine would enter an infinite loop). Therefore, any halting computation is $\le 2^{O(s(n))}$. □

Theorem 16.8. *QBF is PSPACE-complete.*

SAT is like QBF with only existential quantifiers, so this problem is clearly NP-hard.

Proof. Suppose that $L \in \text{PSPACE}$. Then there is a machine M that decides L using space $\le s(n)$ for polynomial s.

Given input x, we want to determine if there is a "path" of length $\le L := 2^{O(s(n))}$ from the starting configuration c_{START} to an accepting configuration c_{ACC}.

We can draw a graph with configurations as nodes and transitions as edges, and so that each node has out-degree 1. We want to say that

$$\exists c_{\text{ACC}}[\text{there is a path of length} \le L \text{ from } c_{\text{START}} \text{ to } c_{\text{ACC}} \text{ and } c_{\text{ACC}} \text{ is accepting}].$$

We can also write that $\exists c_{\text{ACC}}\exists c$ such that there is a path of length $\le L/2$ from c_S to c and a path of length $\le L/2$ from c to c_{ACC} where c_{ACC} is accepting. This process can be iterated.

This isn't quite what we want. At the end, this formula will be enormous. At each step, we halve the length, but we more than double the size of the formula. We only used existential quantifiers, and we should also use the power of universal quantifiers to compress this formula. Note that we are checking two conditions that look very similar to find paths of length $\le L/2$, and we can collapse these with a universal quantifier.

Our formula is now: $\exists c_{\text{ACC}}\exists c\forall(a,b) \in \{(c_{\text{START}},c),(c,c_{\text{ACC}})\}$ [there is a path of length $\le L/2$ from a to b and c_{ACC} is accepting]. We can also rephrase part of this condition as $(a = c_{\text{START}} \wedge b = c) \vee (a = c \wedge b = c_{\text{ACC}})$.

Consider the next level of the recursion to see how this works. Then we have the formula $\exists c_{\text{ACC}}\exists c_1\forall(a_1,b_1)\exists c_2\forall(a_2,b_2)$ [there is a path of length $\le L/4$ from a_2 to $b_2 \wedge c_{\text{ACC}}$ is accepting $\wedge((a_1 = c_S \wedge b_1 = c_1) \vee (a_1 = c_1 \wedge b_1 = c_A)) \wedge ((a_2 = a_1 \wedge b_2 = c_2) \vee (a_2 = c_2 \wedge b_2 = c_1))$].

Here, the number of steps in the construction is $\log_2 L = \log_2 2^{O(s(n))} = O(s(n))$. Each step adds variables corresponding to 3 configurations and Boolean conditions defined by a formula of size $O(s(n))$. The total size of the formula is therefore $O(s^2(n))$. □

Remark. Other games that can be played on a board of size n can also be shown to be PSPACE-complete. The situation with chess is a bit more complicated. If we add a restriction on the number of moves before the game is called a draw, then it is PSPACE-complete. Otherwise, it is actually EXP-complete.

Definition 16.9. NPSPACE is the languages L that are decidable by a non-deterministic TM that uses $\leq p(n)$ space with p polynomial.

This class is never really talked about.

Theorem 16.10 (Savitch)**.** *PSPACE = NPSPACE.*

More precisely, if L is decidable in non-deterministic space $s(n)$ then it is decidable in deterministic space $O(s(n)^2)$.

Proof. Suppose that $L \in$ NPSPACE and M is a non-deterministic Turing machine that decides L and uses $\leq s(n)$ space with s polynomial.

Given some input x, we consider all configurations of machine M on input x. We have the same bound on the number of configurations: $2^{O(s(n))}$. Think of this as a graph G, with configurations as nodes. Now each node can have multiple out-edges. Make an extra vertex c_A, and make all accept states have an edge to there (to make there be only one accepting configuration). We wish to determine if there is a path in G from c_{START} to c_A of length $\leq \ell \leq 2^{O(s(n))}$. Call this procedure $\text{Reach}(c_{\text{START}}, c_A, \ell)$.

Now, we use the same sort of recursion as when we considered QBF. Here is the procedure to compute $\text{Reach}(a, b, \ell)$: If $\ell \geq 2$, then for each configuration c, let $t_1 = \text{Reach}(a, c, \ell/2)$ and $t_2 = \text{Reach}(c, b, \ell/2)$. If $t_1 \wedge t_2$ then halt and accept. Else, reject. If $\ell = 1$, accept if and only if there is a one-step computation from a to b.

What is the space complexity of this procedure? Again, space can be reused. So $s(\ell) = s(\ell/2) + p(n) = (\log \ell)p(n)$. Note that $\log \ell$ is polynomial. □

17. 3/8

Today, we will talk more about the theory of memory-bounded algorithms. PSPACE includes many non-practical algorithms, and today we will discuss the class L, which roughly are algorithms that use no data structures, and the class NL, which roughly are problems solvable by depth first search.

We will show that L $\subseteq$ NL $\subseteq$ P. We will also consider NL-complete problems.

If the input has length n, then you might need space n just to read the input. But some very space-efficient algorithms do not need to do this; for example, an algorithm to determine if a bit string has more zeros than ones.

To capture space-complexity more accurately, we talk about sublinear space complexity. Suppose we have a Turing machine M. One tape just contains the input and is read only. The other tape works the way a tape in a usual Turing machine does, and has both read and write access. In this case, our space complexity will be the number of cells of the work tape that are being used.

Definition 17.1. L is the class of languages solvable with $O(\log n)$ bits of space.

It is still true that if a machine M decides a language using space $\leq s(n)$ then it runs in time $O(n \cdot 2^{O(s(n))}) = O(2^{O(s(n))+\log n})$. Therefore, if a machine uses $O(\log n)$ space, then it runs in polynomial time. This means that $\mathrm{L} \subseteq \mathrm{P}$.

Definition 17.2. NL are the languages that can be solved by a non-deterministic Turing machine that uses $O(\log n)$ space.

This is not a particularly implementable model of computation, but it is a useful way to classify problems.

Example 17.3. s-t connectivity is in NL.

The input is a directed graph $G = (V, E)$ and $s, t \in V$. The goal is to determine if there is a directed path from s to t.

In fact, we claim that STCONN $\in$ NL. To see this consider the following non-deterministic machine.

The input is $G = (V, E)$ and s, t. It computes $n := |V|$ and initializes a pointer $p := s$.

For i from 1 to n:
    Let q be a nondeterministic choice of a neighbor of p.
    If $q = t$, halt and accept.
    Otherwise, let $p = q$.
Reject.

In terms of memory, this is using a constant number of variables, each using $O(\log n)$ bits. This shows that s-t connectivity is in NL. It turns out that other problems using depth first search (such as cycle finding or strongly connected components) also have NL algorithms.

Note that depth first search takes $O(n)$ space if we do not allow non-determinism. Here, using non-determinism allows us to gain something.

Recall Savitch's theorem 16.10 from last time.

Theorem 17.4 (Savitch)**.** *If A is a language solvable by a space $s(n)$ non-deterministic machine ($s(n) \geq \log n$) then A is also solvable by a deterministic Turing machine that uses space $O(s(n)^2)$.*

This shows that STCONN can be decided using $O((\log n)^2)$ bits of memory. So why do we use depth first search, using linear memory? The problem is that we might end up using $n^{O(\log n)}$ time. In fact, this is infeasible to do. We don't know if there is a polynomial time algorithm that uses much less than linear space.

We will show that STCONN is NL-complete. To show this, we can only use reductions using log space. But reductions convert problems into problems. How are we going to write the input? To do this, using a three-tape machine. This has a read-only input tape and a write-only input tape, and a work tape that has read and write access. The space consumed is measured by the work tape.

Equivalently, for $s \geq \log n$, f is computable using space $O(s)$ if the value of the ith bit of $f(x)$ is decidable using space $O(s)$ given x and i.

Definition 17.5. $A \leq_{\log} B$ if there is a function f computable in $O(\log n)$ space such that $x \in A$ if and only if $f(x) \in B$.

Given x, we want to compute $f(x)$ and send it to the machine for B. The problem is that there's no space to store $f(x)$. Think of this as passing $f(x)$ directly to B; suppose that $f(x)$

produces output one bit at a time, and B takes input one bit at a time. Then interleave computing f and running a machine for B. But log space machines can rewind, so f isn't always computed one bit at a time. We need to do something more complicated.

Simulate B, keeping a pointer to the position of the bit of $f(x)$ that the machine wants to read. If B wants to read a particular bit on $f(x)$, run $f(x)$ until we have computed that bit. Then run the machine for B again until it wants to read another bit. At that time, update the pointer, interrupt B, and compute f until we have that bit. In the worst case, the running time is the product of that of f and B, but we only need $O(\log n)$ bits. In general, if the machines for f and B take $s_1(n)$ and $s_2(n)$ bits respectively, the total space usage is $O(\log n) + s_1(n) + s_2(n)$.

Given this discussion, we have

Fact 17.6.

(1) If $A \leq_{\log} B$ and $B \in \mathrm{L}$ then $A \in \mathrm{L}$.
(2) If $A \leq_{\log} B$ and $B \leq_{\log} C$ then $A \leq_{\log} C$.

This means that if A is NL-complete then $A \in \mathrm{L}$ if and only if $\mathrm{L} = \mathrm{NL}$. Also, if A is NL-complete and $B \in \mathrm{NL}$, then $A \leq_{\log} B$ implies that B is NL-complete.

Proposition 17.7. *STCONN is NL-complete.*

Proof. Suppose that $A \in \mathrm{NL}$. Let M be a nondeterministic Turing machine that decides A and uses $O(\log n)$ space. On input x, consider all configurations on M on input x, which is $\leq O(n \cdot 2^{O(s(n))}) = n^{O(1)}$ where $n = |x|$. Construct a graph with vertices as configurations, and with an extra vertex t. The edges are (c, c') if M goes in one step from c to c'. Also, there is (c_{ACC}, t) if c_{ACC} is accepting. Now determine if it is possible to find a path in the graph from $s = c_{\mathrm{START}}$ to t; this translates A into a problem about s-t connectivity. □

This also tells us that $\mathrm{NL} \subseteq \mathrm{P}$ because we can take any language in NL, find a polynomial time reduction to STCONN, and then run depth-first search.

We do not know if NP = coNP, and we suspect that the answer is no. This is because problems in coNP seem hard to verify. However, we know that NL = coNL. There is a clear explanation in Sipser's book.

A recent development from 2005 is that undirected s-t connectivity is in L. This is space-efficient and polynomial time, but the polynomial is very big, so unfortunately this is not practical, but it is still an interesting result.

18. 3/13

Today's topic is randomized algorithms.

Example 18.1. Suppose M_1 and M_2 are $n \times n$ matrices. Normally, matrix multiplication takes time $O(n^3)$, with $O(n^{2.38})$ using clever methods. Here is a $O(n^2)$ randomized algorithm for checking multiplication. Pick a 0-1 bit vector r at random. Test if $M_1 M_2 r = Nr$. If $M_1 M_2 = N$, then $\Pr[M_1 M_2 r = Nr] = 1$. Otherwise, $\Pr[M_1 M_2 r = Nr] \leq \frac{1}{2}$. This is because we are actually looking at $\Pr[Mr = 0]$ where $M = M_1 M_2 - r$, and for any row v of M, we have $\Pr[Mr = 0] \leq \Pr[\langle v, r \rangle = 0] = \Pr[\sum_i v_i r_i = 0] \leq \Pr[r_i = -\sum_{j \neq i} \frac{v_j r_j}{v_i}] \leq \frac{1}{2}$.

If we pick 300 random vectors and test them all, what is the probability of incorrect output? This is $1/2^{300}$ because these are independent random trials. This is a negligible probability.

Here is another problem:

Example 18.2. Let $p(x)$ be a polynomial $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_dx^d$.

Suppose p is hidden in a black box, and we can only see the inputs and outputs. How can we determine if p is identically zero? Simply evaluate p on $d+1$ distinct values! Non-zero degree d polynomials have $\leq d$ roots. The zero polynomial has infinitely many roots.

What if $p(x_1, \ldots, x_n)$ is an multivariate polynomial over $\mathbb{Z}$? Can we determine if p is identically zero? If $p(x_1, \ldots, x_n)$ is a product of m polynomials, each of which has t terms, expanding out the whole thing could take t^m operations. We can try random values for the variables.

Theorem 18.3 (Schwartz-Zippel)**.** *Let $p(x_1, \ldots, x_m)$ be a nonzero polynomial on the variables $x_1, x_2, \ldots, x_m$, where each variable has degree at most d in p. Let F be any subset of integers. If $a_1, \ldots, a_m$ are selected randomly from R, then:* $\Pr[p(a_1, \ldots, a_m) = 0] \leq \frac{md}{|F|}$.

Proof. This is by induction on m. In the base case, when $m = 1$, we have $\Pr[p(a_1) = 0] \leq d/|F|$ because nonzero polynomials of degree d have at most d roots, so at most d elements in F make p zero.

In the inductive step, assume the inequality is true for $m-1$ and prove it is true for m. We'll factor out the last variable x_m. Write $p = p_0 + x_m p_1 + x_m^2 p_2 + \cdots + x_m^d p_d$ where x_m does not occur in any $p_i(x_1, \ldots, x_{m-1})$. If $p(a_1, \ldots, a_m) = 0$, there are only two possibilities:

(1) For all i, $p_i(a_1, \ldots, a_{m-1}) = 0$
(2) Some $p_i(a_1, \ldots, a_{m-1}) \neq 0$, and a_m is a root of the univariate degree d polynomial on x_m that results from evaluating $p_0, \ldots, p_d$ on $(a_1, \ldots, a_{m-1})$.

Note that $\Pr[(1)] \leq \frac{(m-1)d}{|F|}$ and $\Pr[(2)] \leq \frac{d}{|F|}$. Therefore, the probability that we want is $\leq \frac{(m-1)d}{|F|} + \frac{d}{|F|} = \frac{md}{|F|}$. □

Why do we study probabilistic algorithms?

(1) They can be much simpler than deterministic algorithms.
(2) They can be more efficient than deterministic algorithms.
(3) Does randomness make problem much easier to solve? We don't know. The conjecture is that it won't help too much; a deterministic algorithm exists that runs only polynomially slower.

Definition 18.4. A probabilistic Turing machine M is a non-deterministic Turing machine where each non-deterministic step is called a *coin flip* and each non-deterministic step has only two legal next moves (heads or tails).

The probability of branch b is $\Pr[b] = 2^{-k}$ where k is the number of coin flips that occur on branch b. We says that $\Pr[M \text{ accepts } w]$ is the sum of $\Pr[b]$ over all accepting computation histories of M on w.

Definition 18.5. M recognizes language A with error ε if for all strings w, $w \in A$ implies that $\Pr[M \text{ accepts } w] \geq 1 - \varepsilon$ and $w \notin A$ implies that $\Pr[M \text{ doesn't accept } w] \geq 1 - \varepsilon$.

Theorem 18.6. *Let ε be a constant, $0 < \varepsilon < \frac{1}{2}$ and let $p(n)$ be a polynomial. If M_1 has error ε and runs in time $t(n)$, then there is an equivalent machine M_2 such that M_2 has error $2^{-p(n)}$ and runs in time $O(p(n)t(n))$ time.*

Proof Idea. M_2 runs M_1 for $O(p(n))$ independent random trials and records the answer of M_1 each time, and then returns the most common answer (accept or reject). □

Definition 18.7.

BPP $= \{L \mid L$ is recognized by a probabilistic polynomial-time TM with error at most $1/3\}$.

The $1/3$ is an arbitrary choice of number $< 1/2$; once the error is $< 1/2$, we can apply error reduction.

Definition 18.8. An *arithmetic formula* is like a Boolean formula, except it has $+$ and $\times$ instead of OR and AND.

$$\text{ZERO-POLY} = \{p : p \text{ is an arithmetic formula that is identically zero}\}.$$

Theorem 18.9. *ZERO-POLY* $\in$ *BPP.*

Think of this as polynomial identity testing.

Proof Idea. Suppose $|p| = n$. Then p has $\leq n$ variables, and the degree of p is $O(n)$. Assign each variable x_i a large random integer value from the set $\{1, \ldots, n^3\}$. Evaluate the formula on this assignment. Schwartz-Zippel 18.3 implies that this works! □

Is BPP $\subseteq$ NP? Nobody knows. Is BPP $\subseteq$ PSPACE? Yes! Run through all possible sequences of coin flips one at a time (depth-first search through the computation tree), and count up the branches that accept.

Definition 18.10. A language A is in RP (randomized P) if there is a non-deterministic polynomial-time Turing machine M such that for all strings x, we have that $x \notin A$ implies that $\Pr[M(x) \text{ accepts}] = 0$, and $x \in A$ implies that $\Pr[M(x) \text{ accepts}] > 2/3$.

Theorem 18.11. *Language A is in RP if and only if for all k there exists a probabilistic polynomial time Turing machine such that $x \notin A$ implies that $M(x)$ rejects with probability 1, and $x \in A$ implies that $M(x)$ accepts with probability at least $1 - 2^{-|x|^k}$.*

Definition 18.12.

$$\text{NONZERO-POLY} = \{p \mid p \text{ is an arithmetic formula that is not identically zero}\}.$$

Theorem 18.13. *NONZERO-POLY* $\in$ *RP.*

Note that RP $\subseteq$ NP and RP $\subseteq$ BPP.

There is a widespread belief that BPP = P, but we are far from showing that. Maybe BPP = EXP? That would be strange, but we cannot prove that it is not true.

19. 3/15

Today we will talk about security. Usually, we cannot prove security unconditionally; it's about as hard as P=NP. We want to show that no efficient algorithm can make a successful attack. The proof of security will show that any attack is intractible, via a reduction. Note that here, we want to consider average case analyses instead of worst case analyses.

Today, we will discuss:

- zero-knowledge protocols
- identification protocols

- zero knowledge protocols for quadratic reciprocity
- identification protocol based on factorization.

Roughly, we'll see some of the main features of cryptography. Some definitions may seem paradoxical or impossible, but they can actually be realized through some clever ideas.

We begin with zero-knowledge proofs. Someone wishes to convince someone else that a statement is true without communicating any of the details of the proof. It turns out that any NP-type statement has zero-knowledge protocols. This may be complicated, but the case of quadratic reciprocity is simple.

Consider identification protocols. Say we have some websites, e.g. Facebook, Amazon, Wordpress. Each site knows my username and password. When we want to buy stuff on Amazon or check out pictures on Facebook, we want to log in to one of these sites. There are many issues. If I send my password in the clear, anyone will be able to see my password and steal it. So we want network traffic to be included. Or someone could hack into Amazon and steal my password, which is why it should be hashed. Another issue is that we reuse passwords. If I give my password to another site, they could use it to log into my other websites.

Instead, we have a secret key that we keep to ourselves, and a public key that everyone knows. To set up an account, I just give them my public key. This is designed so that even if network traffic is public, we don't leak any information. The point is that given my public key, an attacker cannot do anything. This doesn't protect against a man-in-the-middle attack, however.

The point is that we have some information, like a solution to some NP problem, and we want to use this for identification.

First, we describe quadratic reciprocity. Suppose we have some big $N = pq$ where p and q are prime. Our assumption is that it is computationally hard to find p and q given N. (Here, p and q are big, around 1000 bits.)

Definition 19.1. $\mathbb{Z}_n = \{0, \dots, n-1\}$ with addition mod n.
$\mathbb{Z}_n^* = \{x \in \mathbb{Z}_n : \gcd(x, n) = 1\}$ with multiplication mod n.
x is a *quadratic residue* if there is a square root r such that $r^2 \equiv x \bmod n$.

Note that $|\mathbb{Z}_N^*| = (p-1)(q-1) \approx N - 2\sqrt{N}$. So it is easy to find random elements in $\mathbb{Z}_N^*$.

Example 19.2. Consider $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$. To find the quadratic residues, square every element to see that they are 1 and 4, and each has four square roots.

It turns out that when $N = pq$ for $p, q \geq 3$, there are $\frac{(p-1)(q-1)}{4}$ quadratic residues in $\mathbb{Z}_N^*$, each with four square roots. This can be shown using the Chinese Remainder Theorem. It is believed that determining if an element is a quadratic residue is hard. But given a quadratic residue, then finding square roots is as hard as factoring.

Why? Four square roots means that they come in two pairs. Suppose x is a quadratic residue, and a and b are nontrivially different square roots: $a \not\equiv b \bmod N$, $a \not\equiv -b \bmod N$, $x \equiv a^2 \equiv b^2 \bmod N$. It turns out that given a and b we can factor N. This is because we have $(a+b)(a-b) \equiv a^2 - b^2 \equiv 0 \bmod N$, but $a + b, a - b \neq 0$. So one of these is a multiple of p and the other is a multiple of q. So computing $\gcd(a+b, N)$ and $\gcd(a-b, N)$ gives p and q. Assuming that factoring is hard, finding nontrivially different square roots is just as hard.

Suppose we had a square root algorithm $\mathcal{A}$. Then we can find nontrivially different square roots. Pick a random element a and run $\mathcal{A}$ on $a^2 \bmod N$. If $\mathcal{A}$ finds b where $b \not\equiv \pm a \bmod N$, use a and b to factor N. Otherwise, repeat. How often will this find different square roots? Just think about it – it should happen 50% of the time. For any value of a^2, there are two values of a where we are happy and two where we are not. So we have a reduction. Having a square root algorithm implies having a factoring algorithm.

Here is a zero-knowledge protocol. Suppose we have a user, who will be called the prover. We also have a server, or verifier. The prover knows p and q. The value of $N = pq$ is public.

For this protocol, the server sends us a quadratic residue $r^2 \bmod N$. The user wants to prove that it knows a square root of the first message while giving away no additional information; in particular, we should not give away the value of square root. This should be convincing that we are who we claim to be, since nobody else knows p and q. Our identity is defined as the person who knows how to factor N.

Now, we can state the protocol. There is a prover and a verifier. Everyone knows x and the prover knows some square root r of x, e.g. $x \equiv r^2 \bmod N$. Then the prover picks a random $a \in \mathbb{Z}_N^*$, and sends $y = a^2 x \bmod N$ to the verifier. The verifier sends a random bit $b \in \{0, 1\}$. The prover then responds with $z = a \bmod N$ if $b = 0$, or $z = ar \bmod N$ if $b = 1$. The verifier then checks that $z^2 x = y \bmod N$ if $b = 0$, and $z^2 = y \bmod N$ if $b = 1$. This is repeated some number of times, since it tells us something useful with probability $1/2$.

If x is a quadratic residue and the prover and verifier follow the protocol, then the verifier accepts with probability 1. If x is not a quadratic residue, then no matter what the prover does, the verifier accepts with probability $\leq 1/2$.

To see this, suppose that x is not a quadratic residue. Then there are two possibilities: (1): y is not a quadratic residue. (0): yx^{-1} is not a quadratic residue. The case that y and yx^{-1} are both quadratic residues is impossible, because it means that we can write $y = a^2$ and $yx^{-1} = b^2$, so then $x = y(b^{-1})^2 = (ab^{-1})^2$ is a quadratic residue. In case (1), e.g when $b = 1$, the prover cannot answer. This happens with probability $1/2$.

So this convinces someone that x is a quadratic residue. Why is this zero-knowledge? Someone watching this protocol would see y, a bit b, and then either $\sqrt{y}$ or $\sqrt{y/x}$. If you show this to someone else, it is not really convincing at all, because you could have cooked up such an example yourself, by picking a number and squaring it and possibly multiplying by x.

Definition 19.3. A protocol is (honest verifier) zero knowledge if there an efficient randomized simulator S that on input x outputs a transcript of an interaction between the prover and the verifier that has same distribution as a real interaction.

Here's the simulator for the protocol above: On input x, pick a random bit $b \in \{0, 1\}$. If $b = 1$, pick a random z and output $z^2, 0, z$; if $b = 0$, pick a random z and output $z^2 x, 1, z$.

This assumes that the verifier is following the protocol. We would like that even if the other party deviates from the protocol, it still would not learn anything. So here's the full definition of zero-knowledge:

Definition 19.4. A protocol is zero-knowledge if for every verifier algorithm V' there is a simulator $S_{V'}$ such that $S_{V'}(x)$ has the same distribution as the interaction between the prover and V' on input x.

Let's see that our protocol still fits this definition. Given V', here is our simulator $S_{V'}$: On input x, pick $g = 0$ or $g = 1$. If $g = 1$, pick a random z, set $y = z^2$. If V' given y asks $b = 0$, then this fails. Else, output $y, 1, z$. If $g = 0$, pick a random z, set $y = z^2x$. If V' given y asks $b = 1$, then this fails. Else, output $y, 0, z$.

So now we have a zero-knowledge proof that a given integer is a quadratic residue. In fact, this is also a proof of knowledge: proof that you know a square root. This is what is needed in an identification protocol.

Definition 19.5. A protocol for a problem in NP is a *proof of knowledge* with soundness p if there is an efficient randomized algorithm E (called the knowledge extractor) such that for every prover algorithm P' and input x, if P' makes the verifier accept with probability $> p$ on x then $E(x, P')$ finds a witness for x with probability $\Omega(p)$.

Then anyone who can run the protocol must actually know a square root.

Here's our knowledge extractor: Consider some P' and x makes the verifier accept with probability $> 51\%$. We run the protocol. This means that there is some probability $p > 2\%$ of picking y so that the prover is able to answer both challenges. So our algorithm is the run the first round of the protocol, and continue with both $b = 0$ and $b = 1$. Then there is a certain y such that we know z_1 so that $z_1^2 = y$ and z_0 so that $z_0^2 = yx^{-1}$. But then $x = y(z_0^{-1})^2$, so $x = (z_1 z_0^{-1})^2$, so we have a square root of x. So doing this 50 times on average, we will have a square root of x. So we have a protocol that is both zero-knowledge and a proof of knowledge. This is now a totally secure identification scheme.

E-mail address: moorxu@stanford.edu