ECS 120 Notes

David Doty
based on *Introduction to the Theory of Computation* by Michael Sipser

Based heavily (some parts copied) on *Introduction to the Theory of Computation*, by Michael Sipser. These are notes intended to assist in lecturing from Sipser's book; they are not entirely the original work of the author. Some proofs are also taken from *Automata and Computability* by Dexter Kozen.

Contents

Introduction

What This Course is About

The following is a rough sketch of what to expect from this course:

- In ECS 30/40/60, you programmed computers without studying them formally.
- In ECS 20, you formally studied things that are not computers.
- In ECS 120, you will use the tools of ECS 20 to formally study computers.

Newton's equations of motion tell us that each body of mass obeys certain rules that cannot be broken. For instance, a body cannot accelerate in the opposite direction in which force is being applied to it. Of course, nothing in the world is a rigid body of mass subject to no friction, etc., so Newton's equations of motion do not *exactly* predict anything. But they are a useful *abstraction* of what real matter is like, and many things in the world are close enough to this abstraction that Newton's predictions are reasonably accurate.

The fundamental premise of the theory of computation is that the computer on your desk obeys certain laws, and therefore, certain unbreakable limitations. I often think of the field of computer science outside of theory as being about proving what can be done with a computer, by doing it. Much of research in theoretical computer science is about proving what *cannot* be done with a computer. This can be more difficult, since you cannot simply cite your failure to invent an algorithm for a problem to be a proof that there is no algorithm. But certain important problems cannot be solved with any algorithm, as we will see.

We will draw no distinction between the idea of "formal proof" and more nebulous instructions such as "show your work"/"justify your answer"/"explain". A "proof" of a theorem is an argument that convinces an intelligent person who has never seen the theorem before and cannot see why it is true with having it explained. It does not matter if the argument uses formal notation or not (though formal notation is convenient for achieving brevity), or if it uses induction or contradiction or just a straightforward argument (though it is often easier to think in terms of induction or contradiction). What matters is that there are no holes or counter-arguments that can be thrown at the argument, and that every statement is precise and unambiguous.

Note, however, that one effective technique used by Sipser to prove theorems is to first give a "proof idea", helping you to see how the proof will go. The proof is easier to read because of the proof idea, but the proof idea by itself is not a proof. In fact, I would go so far as to say that the proof by itself is not a very effective proof either, since bare naked details and formalism, without any intuition to reinforce it, do not communicate why the theorem is true any better than the hand-waving proof idea. Both are usually necessary to accomplish the goal of the proof: to communicate why the theorem is true. In this course, in the interest of time, I will often give the intuitive proof idea only verbally, and write only the formal details on the board, since learning to turn the informal intuition into a formal detailed proof is the most difficult part of this course. On your homework, however, you should explicitly write both, to make it easy for the TA to understand your proof and give you full credit.

Three problems

Multivariate polynomials (Diophantine equation)

$x^2 + 2xy - y^3 = 13$. Does this have an integer solution? Yes: $x = 3, y = 2$

How about $x^2 - y^2 = 2$? No.

Task A: write an algorithm that indicates whether a given Diophantine equation has any integer solution.

Fact: Task 1 is impossible.[1]

Task A': write an algorithm that indicates whether a given Diophantine equation has any ~~integer~~ real solution.

Fact: Task A' is possible.[2]

Paths touring a graph

Does the graph $G = (V, E)$ given in Figure 1 have a path that contains each edge exactly once? (*Eulerian path*)

Task B: write an "efficient" algorithm that indicates if a graph as a path containing each edge exactly once

Fact: Task B is possible. (iff connected and even degree)

Task B': write an "efficient" algorithm that indicates if a graph as a path containing each ~~edge~~ node exactly once

Fact: (assuming $\mathsf{P} \neq \mathsf{NP}$) Task B' is impossible.

Balanced parentheses

• `[[]]`	balanced
• `[[]`	unbalanced
• `[][[][[[]][]]]`	balanced
• `[[]][]][[]`	unbalanced

[1] We could imagine trying "all" possible integer solutions, but if there is no integer solution, then we will be trying forever and the algorithm will not halt.

[2] Strange, since there are more potential solutions to search, but the algorithm does not work by trying different solutions.

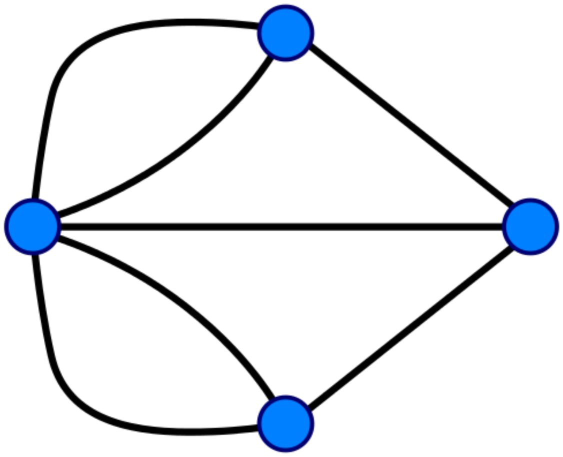

Figure 1: “Königsberg graph”. Licensed under CC BY-SA 3.0 via Commons – `https://commons.wikimedia.org/wiki/File:K%C3%B6nigsberg_graph.svg#/media/File:K%C3%B6nigsberg_graph.svg`

A *regular expression* is an expression that matches some strings and not others. For example

$$(0(0 \cup 1 \cup 2)^*) \cup ((0 \cup 2)^* 1)$$

matches any string of digits that starts with a 0, followed by any number of 0’s, 1’s, and 2’s, or ends with a 1, preceded by any number of 0’s and 2’s.

Task C: write a regular expression that matches a string of parentheses exactly when they are balanced.

Fact: Task C is impossible

Task C': write a regular expression that matches a string of parentheses exactly when every `[` is followed by a `]`.

Answer: (`[` ∪ `]`)*

Rough layout of this course

Computability theory (unit 2): What problems can algorithms solve? (real roots of polynomials, but not integer roots)

Complexity theory (unit 3): What problems can algorithms solve efficiently? (paths visiting every edge, but not every vertex)

Automata theory (unit 1): What problems can algorithms solve with “optimal” efficiency? (at least for finite automata this is a good description; balanced parentheses provably requires non-constant memory)

0.1 Mathematical Background

Reading assignment: Chapter 0 of Sipser. [3]

0.1.1 Implication Statements

Given two boolean statements p and q [4], the *implication* $p \implies q$ is shorthand for "p implies q", or "If p is true, then q is true" [5], p is the *hypothesis*, and q is the *conclusion.* The following statements are related to $p \implies q$:

- the *inverse*: $\neg p \implies \neg q$
- the *converse*: $q \implies p$
- the *contrapositive*: $\neg q \implies \neg p$[6]

If an implication statement $p \implies q$ and its converse $q \implies p$ are both true, then we say p if and only if (iff) q, written $p \iff q$. Proving a "$p \iff q$" theorem usually involves proving $p \implies q$ and $q \implies p$ separately.

0.1.2 Sets

A *set* is a group of objects, called *elements*, with no duplicates.[7] The *cardinality* of a set A is the number of elements it contains, written $|A|$. For example, $\{7, 21, 57\}$ is the set consisting of the integers 7, 21, and 57, with cardinality 3.

For two sets A and B, we write $A \subseteq B$, and say that A is a *subset* of B, if every element of A is also an element of B. A is a *proper subset* of B, written $A \subsetneq B$, if $A \subseteq B$ and $A \neq B$.

We use the following sets throughout the course

- the *natural numbers* $\mathbb{N} = \{0, 1, 2, \ldots\}$
- the *integers* $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- the *rational numbers* $\mathbb{Q} = \left\{ \left. \frac{p}{q} \;\right|\; p \in \mathbb{Z}, q \in \mathbb{Z}, \text{ and } q \neq 0 \right\}$
- the *real numbers* $\mathbb{R}$

[3] This is largely material from ECS 20.

[4] e.g., "Hawaii is west of California", or "The stoplight is green."

[5] e.g., "If the stoplight is green, then my car can go."

[6] The contrapositive of a statement is logically equivalent to the statement itself. For example, it is equivalent to state "If someone is allowed to drink alcohol, then they are at least 21" and "If someone is under 21, then they are not allowed drink alcohol". Hence a statement's converse and inverse are logically equivalent to each other, though not equivalent to the statement itself.

[7] Think of `std::set`.

The unique set with no elements is called the *empty set*, written $\emptyset$.

To define sets symbolically,[8] we use *set-builder notation*: for instance, $\{\ x \in \mathbb{N} \mid x \text{ is odd } \}$ is the set of all odd natural numbers.

We write $\forall x \in A$ as a shorthand for "for all elements x in the set A ...", and $\exists x \in A$ as a shorthand for "there exists an element x in the set A ...". For example, $(\exists n \in \mathbb{N})\ n > 10$ means "there exists a natural number greater than 10".

Given two sets A and B, $A \cup B = \{\ x \mid x \in A \text{ or } x \in B\ \}$ is the *union* of A and B, $A \cap B = \{\ x \mid x \in A \text{ and } x \in B\ \}$ is the *intersection* of A and B, and $A \setminus B = \{\ x \in A \mid x \notin B\ \}$ is the *difference* between A and B (also written $A - B$). $\overline{A} = \{\ x \mid x \notin A\ \}$ is the *complement* of A. [9]

Given a set A, $\mathcal{P}(A) = \{\ S \mid S \subseteq A\ \}$ is the *power set* of A, the set of all subsets of A. For example,

$$\mathcal{P}(\{2,3,5\}) = \{\emptyset, \{2\}, \{3\}, \{5\}, \{2,3\}, \{2,5\}, \{3,5\}, \{2,3,5\}\}.$$

Given any set A, it always holds that $\emptyset, A \in \mathcal{P}(A)$, and that $|\mathcal{P}(A)| = 2^{|A|}$ if $|A| < \infty$. [10] [11]

0.1.3 Sequences and Tuples

A *sequence* is an ordered list of objects [12]. For example, $(7, 21, 57, 21)$ is the sequence of integers 7, then 21, then 57, then 21.

A *tuple* is a finite sequence.[13] $(7, 21, 57)$ is a 3-tuple. A 2-tuple is called a *pair*.

For two sets A and B, the *cross product* of A and B is $A \times B = \{\ (a,b) \mid a \in A \text{ and } b \in B\ \}$. For $k \in \mathbb{N}$, we write $A^k = \underbrace{A \times A \times \ldots \times A}_{k \text{ times}}$ and $A^{\leq k} = \bigcup_{i=0}^{k} A^i$.

For example, $\mathbb{N}^2 = \mathbb{N} \times \mathbb{N}$ is the set of all ordered pairs of natural numbers.

[8] In other words, to express them without listing all of their elements explicitly, which is convenient for large finite sets and necessary for infinite sets.

[9] Usually, if A is understood to be a subset of some larger set U, the "universe" of possible elements, then $\overline{A}$ is understood to be $U \setminus A$. For example if we are dealing only with $\mathbb{N}$, and $A \subseteq \mathbb{N}$, then $\overline{A} = \{\ n \in \mathbb{N} \mid n \notin A\ \}$. In other words, we used "typed" sets, in which case each set we use has some unique superset – such as $\{0,1\}^*$, $\mathbb{N}$, $\mathbb{R}$, $\mathbb{Q}$, the set of all finite automata, etc. – that is considered to contain all the elements of the same type as the elements of the set we are discussing. Otherwise, we would have the awkward situation that for $A \subseteq \mathbb{N}$, $\overline{A}$ would contain not only nonnegative integers that are not in A, but also negative integers, real numbers, strings, functions, stuffed animals, and other objects that are not elements of A.

[10] Why?

[11] Actually, Cantor's theory of infinite set cardinalities makes sense of the claim that $|\mathcal{P}(A)| = 2^{|A|}$ even if A is an infinite set. The furthest we will study this theory in this course is to observe that there are at least two infinite set cardinalities: that of the set of natural numbers, and that of the set of real numbers, which is bigger than the set of natural numbers according to this theory.

[12] Think of `std::vector`.

[13] The closest Java analogy to a tuple, as we will use them in this course, is an object. Each member variable of an object is like an element of the tuple, although Java is different in that each member variable of an object has a name, whereas the only way to distinguish one element of a tuple from another is their position. But when we use tuples, for instance to define a finite automaton as a 5-tuple, we intuitively think of the 5 elements as being like 5 member variables that would be used to define a finite automaton object. And of course, the natural way to implement such an object in C++ by defining a `FiniteAutomaton` class with 5 member variables, which is an easier way to keep track of what each of the 5 elements is supposed to represent than, for instance, using an `void[]` array of length 5.

0.1.4 Functions and Relations

A *function* f that takes an input from set D (the *domain*) and produces an output in set R (the *range*) is written $f : D \to R$.[14] Given $A \subseteq D$, define $f(A) = \{\ f(x) \mid x \in A\ \}$; call this the *image of A under f*.

Given $f : D \to D$, $k \in \mathbb{N}$ and $d \in D$, define $f^k : D \to D$ by $f^k(d) = \underbrace{f(f(\ldots f(}_{k \text{ times}} d)) \ldots))$ to be f composed with itself k times.

If f might not be defined for some values in the domain, we say f is a *partial function.*[15] If f is defined on all values, it is a *total function.*[16]

A function f with a finite domain can be represented with a table. For example, the function $f : \{0, 1, 2, 3\} \to \mathbb{Q}$ defined by $f(n) = \frac{n}{2}$ is represented by the table

n	$f(n)$
0	0
1	$\frac{1}{2}$
2	1
3	$\frac{3}{2}$

If

$$(\forall d_1, d_2 \in D)\ d_1 \neq d_2 \implies f(d_1) \neq f(d_2),$$

then we say f is *1-1* (*one-to-one* or *injective*).[17]

If

$$(\forall r \in R)(\exists d \in D)\ f(d) = r,$$

then we say f is *onto* (*surjective*). Intuitively, f “covers” the range R, in the sense that no element of R is left un-mapped-to by f.

f is a *bijection* (a.k.a. a *1-1 correspondence*) if f is both 1-1 and onto.

A *predicate* is a function whose output is boolean.

Given a set A, a *relation* R on A is a subset of $A \times A$. Intuitively, the elements in R are the ones related to each other. Relations are often written with an operator; for instance, the relation $\leq$ on $\mathbb{N}$ is the set $R = \{\ (n, m) \in \mathbb{N} \times \mathbb{N} \mid (\exists k \in \mathbb{N})\ n + k = m\ \}$.

[14] Think of a static method; `Integer.parseInt`, which takes a `String` and returns the `int` that the `String` represents (if it indeed represents an integer) is like a function with domain `String` and range `int`. `Math.max` is like a function with domain `int` × `int` (since it accepts a pair of `int`s as input) and range `int`.

[15] For instance, `Integer.parseInt` is (strictly) partial, because not all `String`s look like integers, and such `String`s will cause the method to throw a `NumberFormatException`.

[16] Every total function is a partial function, but the converse does not hold for any function that is undefined for at least one value. We will usually assume that functions are total unless explicitly stated otherwise.

[17] Intuitively, f does not map any two points in D to the same point in R. It does not lose information; knowing an output $r \in R$ suffices to identify the input $d \in D$ that produced it (through f).

0.1.5 Strings and Languages

An *alphabet* is any non-empty finite set, whose elements we call *symbols* or *characters*. For example, $\{0, 1\}$ is the binary alphabet, and

$$\{\text{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}\}$$

is the Roman alphabet. [18]

A *string* over an alphabet is a finite sequence of symbols taken from the alphabet. We write strings such as 010001, without the parentheses and commas. If x is a string, $|x|$ denotes the *length* of x.

If Σ is an alphabet, the set of all strings over Σ is denoted Σ^*. For $n \in \mathbb{N}$, $\Sigma^n = \{\ x \in \Sigma^* \mid |x| = n\ \}$ is the number of strings in Σ^* of length n. Similarly $\Sigma^{\leq n} = \{\ x \in \Sigma^* \mid |x| \leq n\ \}$ and $\Sigma^{<n} = \{\ x \in \Sigma^* \mid |x| < n\ \}$.

The string of length 0 is written λ, and in the textbook, ε; in most programming languages it is written `""`.

Note in particular the difference between λ, $\emptyset$, and $\{\lambda\}$.[19]

Given $n, m \in \mathbb{N}$, $x[n\,..\,m]$ is the substring consisting of the nth through mth symbols of x, and $x[n] = x[n\,..\,n]$ is the nth symbol in x.

We write xy (or $x \circ y$ when we would like an explicit operator symbol) to denote the concatenation of x and y, and given $k \in \mathbb{N}$, we write $x^k = \underbrace{xx\ldots x}_{k \text{ times}}$. [20]

Given two strings $x, y \in \Sigma^*$ for some alphabet Σ, x is a *prefix* of y, written $x \sqsubseteq y$, if x is a substring that occurs at the start of y. x is a *suffix* of y if x is a substring that occurs at the end of y.

The *lexicographic ordering* (a.k.a. *military ordering*) of strings over an alphabet is the standard dictionary ordering, except that shorter strings precede longer strings. For example, the lexicographical ordering of $\{0, 1\}^*$ is

$$\lambda, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$$

A *language* (a.k.a. a *decision problem*) is a set of strings. A *class* is a set of languages. [21]

Given two languages $A, B \subseteq \Sigma^*$, let $AB = \{\ ab \mid a \in A \text{ and } b \in B\ \}$ (also denoted $A \circ B$).[22] Similarly, for all $n \in \mathbb{N}$, $A^n = \underbrace{AA\ldots A}_{n \text{ times}}$,[23], $A^{\leq n} = \bigcup_{i=0}^{n} A^i$ and $A^{<n} = A^{\leq n} \setminus A^n$.[24]

[18] We always let the symbols in the alphabet have single-character names.

[19] λ is a string, $\emptyset$ is a set with no elements, and $\{\lambda\}$ is a set with one element. Intuitively, think of the following Java code as defining these three objects

```
String lambda = "";
Set emptySet = new HashSet();
Set<String> setWithLambda = new HashSet<String>();
setWithLambda.add(lambda);
```

[20] Alternatively, define x^k inductively as $x^0 = \lambda$ and $x^k = xx^{k-1}$

[21] These terms are useful because, without them, we would just call everything a "set", and easily forget whether it is a set of strings, a set of set of strings, or even the dreaded set of set of set of strings (they are out there; the arithmetical and polynomial hierarchies are sets – sequences, actually – of classes).

[22] The set of all strings formed by concatenating one string from A to one string from B

[23] The set of all strings formed by concatenating n strings from A.

[24] Note that there is ambiguity, since A^n could also mean the set of all n-tuples of strings from A, which is a

Given a language A, let $A^* = \bigcup_{n=0}^{\infty} A^n$.[25] Note that $A = A^1$ (hence $A \subseteq A^*$).

Examples. Define the languages $A, B \subseteq \{0,1,2\}^*$ as follows: $A = \{0, 11, 222\}$ and $B = \{000, 11, 2\}$. Then

$$\begin{aligned}
AB &= \{0000, 011, 02, 11000, 1111, 112, 222000, 22211, 2222\} \\
A^2 &= \{00, 011, 0222, 110, 1111, 11222, 2220, 22211, 222222\} \\
A^* &= \{\underbrace{\lambda}_{A^0}, \underbrace{0, 11, 222}_{A^1}, \underbrace{00, 011, 0222, 110, 1111, 11222, 2220, 22211, 222222}_{A^2}, \underbrace{000, 0011}_{\text{part of } A^3}, \ldots\}
\end{aligned}$$

LECTURE: end of day 1

0.1.6 Graphs

See the textbook for review.

0.1.7 Boolean Logic

See the textbook for review.

0.2 Proof by Induction

[26]

0.2.1 Proof by Induction on Natural Numbers

Theorem 0.2.1. *For every $n \in \mathbb{N}$, $|\{0,1\}^n| = 2^n$.*

Proof. (by induction on n) [27]

different set. We assume that A^n means n-fold concatenation whenever A is a language. The difference is that in concatenation of strings, boundaries between strings are lost, whereas tuples always have the various elements of the tuple delimited explicitly from the others.

[25] The set of all strings formed by concatenating 0 or more strings from A.

[26] The book discusses proof by construction and proof by contradiction as two alternate proof techniques. These are both simply formalizations of the way humans naturally think about ordinary statements and reasoning. Proof by induction is the only general technique among the three that really is a technique that must be taught, rather than a name for something humans already intuitively understand. Luckily, *you* already understand proof by induction better than most people, since it is merely the "proof" version of the technique of recursion you learned in programming courses.

[27] To start, state in English what the theorem is saying: For every string length n, there are 2^n strings of length n.

Base case: $\{0,1\}^0 = \{\lambda\}$.[28] $|\{\lambda\}| = 1 = 2^0$, so the base case holds.

Inductive case: Assume $|\{0,1\}^{n-1}| = 2^{n-1}$.[29] We must prove that $|\{0,1\}^n| = 2^n$. Note that every $x \in \{0,1\}^{n-1}$ appears as a prefix of *exactly* two *unique* strings in $\{0,1\}^n$, namely $x0$ and $x1$.[30] Then

$$\begin{aligned} |\{0,1\}^n| =& 2 \cdot |\{0,1\}^{n-1}| \\ =& 2 \cdot 2^{n-1} && \text{inductive hypothesis} \\ =& 2^n. \end{aligned}$$

□

Theorem 0.2.2. *For every $n \in \mathbb{Z}^+$, $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$.*

Proof. **Base case ($n = 1$):** $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{1}{1(1+1)} = \frac{1}{2} = \frac{n}{n+1}$, so the base case holds.

Inductive case: Let $n \in \mathbb{Z}^+$ and suppose the theorem holds for n. Then

$$\begin{aligned} \sum_{i=1}^{n+1} \frac{1}{i(i+1)} =& \frac{1}{(n+1)(n+2)} + \sum_{i=1}^{n} \frac{1}{i(i+1)} && \text{pull out last term} \\ =& \frac{1}{(n+1)(n+2)} + \frac{n}{n+1} && \text{inductive hypothesis} \\ =& \frac{1+n(n+2)}{(n+1)(n+2)} \\ =& \frac{n^2+2n+1}{(n+1)(n+2)} \\ =& \frac{(n+1)^2}{(n+1)(n+2)} \\ =& \frac{n+1}{n+2}, \end{aligned}$$

so the inductive case holds.

□

0.2.2 Induction on Other Structures

31

[28] Note that $\{0,1\}^0$ is *not* $\emptyset$; there is always one string of length 0, so the set of such strings is not empty.

[29] Call this the *inductive hypothesis*, the fact we get to assume is true in proving the inductive case.

[30] The fact that they are unique means that if we count two strings in $\{0,1\}^n$ for every one string in $\{0,1\}^{n-1}$, we won't double-count any strings. Hence $|\{0,1\}^n| = 2 \cdot |\{0,1\}^{n-1}|$

[31] Induction is often taught as something that applies only to natural numbers, but one can write recursive algorithms that operate on data structures other than natural numbers. Similarly, it is possible to prove something by induction on something other than a natural number.

Here is an inductive definition of the number of 0's in a binary string x, denoted $\#(0,x)$.[32]

$$\#(0,x) = \begin{cases} 0, & \text{if } x = \lambda; & \text{(base case)} \\ \#(0,w)+1, & \text{if } x = w0 \text{ for some } w \in \{0,1\}^*; & \text{(inductive case)} \\ \#(0,w), & \text{if } x = w1 \text{ for some } w \in \{0,1\}^*. & \text{(inductive case)} \end{cases}$$

To prove a theorem by induction, identify the base case as the "smallest" object[33] for which the theorem holds.[34]

Theorem 0.2.3. *Every binary tree T of depth d has at most 2^d leaves.*

Proof. (by induction on a binary tree T) For T a tree, let $d(T)$ be the depth of T, and $l(T)$ the number of leaves in T.

Base case: Let T be the tree with one node. Then $d(T) = 0$, and $2^0 = 1 = l(T)$.

Inductive case: Let T's root have subtrees T_0 and T_1, at least one of them non-empty. If only one is non-empty (say T_i), then

$$\begin{aligned} l(T) =& l(T_i) & \\ \leq& 2^{d(T_i)} & \text{inductive hypothesis} \\ =& 2^{d(T)-1} & \text{definition of depth} \\ <& 2^{d(T)}. & \end{aligned}$$

If both subtrees are non-empty, then

$$\begin{aligned} l(T) =& l(T_0) + l(T_1) & \\ \leq& 2^{d(T_0)} + 2^{d(T_1)} & \text{ind. hyp.} \\ \leq& \max\{2^{d(T_0)} + 2^{d(T_0)}, 2^{d(T_1)} + 2^{d(T_1)}\} & \\ =& \max\{2^{d(T_0)+1}, 2^{d(T_1)+1}\} & \\ =& 2^{\max\{d(T_0)+1, d(T_1)+1\}} & 2^n \text{ is monotone increasing} \\ =& 2^{d(T)}. & \text{definition of depth} \end{aligned}$$

□

[32] We will do lots of proofs involving induction on strings, but for now we will just give an inductive definition. Get used to breaking down strings and other structures in this way.

[33] In the case of strings, this is the empty string. In the case of trees, this could be the empty tree, or the tree with just one node: the root (just like with natural numbers, the base case might be 0 or 1, depending on the theorem).

[34] The inductive step should then employ the truth of the theorem on some "smaller" object than the target object. In the case of strings, this is typically a substring, often a prefix, of the target string. In the case of trees, a subtree, typically a subtree of the root. Using smaller subtrees than the immediate subtrees of the root, or shorter substrings than a one-bit-shorter prefix, is like using a number smaller than $n-1$ to prove the inductive case for n; this is the difference between *weak induction* (using the truth of the theorem on $n-1$ to prove it for n) and *strong induction* (using the truth of the theorem on *all* $m < n$ to prove it for n)

Chapter 1

Regular Languages

1.1 Finite Automata

Reading assignment: Section 1.1 in Sipser.

See Sipser example of a finite automaton to control an automatic door.

See Figure 1.4 in the textbook. This is a *state diagram* of a finite automaton M_1. it has three *states*: q_1, q_2, and q_3. q_1 is the *start state.* q_2 is the only *accept state.* The arrows are *transitions.* The digits labeling the transitions are *input symbols.* The state M_1 is in at the end of the input determines whether M_1 *accepts* or *rejects.*

If we give the input string 1101 to M_1, the following happens.

1. Start in state q_1
2. Read 1, transition from q_1 to q_2
3. Read 1, transition from q_2 to q_2
4. Read 0, transition from q_2 to q_3
5. Read 1, transition from q_3 to q_2
6. Accept the input because M_1 is in state q_2 at the end of the input.

1.1.1 Formal Definition of a Finite Automaton (Syntax)

To describe how a finite automaton transitions between states, we introduce a *transition function* δ. The goal is to express that if an automaton is in state q, and it reads the symbol 1 (for example), and transitions to state q', then this means $\delta(q, 1) = q'$.

Definition 1.1.1. A *(deterministic) finite automaton (DFA)* is a 5-tuple $(Q, \Sigma, \delta, s, F)$, where

- Q is a non-empty, finite set of *states*,
- Σ is the *input alphabet*,

- $\delta : Q \times \Sigma \to Q$ is the *transition function*,
- $s \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accepting states*.

For example, the DFA M_1 of Figure 1.4 in Sipser is defined $M_1 = (Q, \Sigma, \delta, s, F)$, where

- $Q = \{q_1, q_2, q_3\}$,
- $\Sigma = \{0, 1\}$,
- δ is defined

$$\begin{aligned}
\delta(q_1, 0) &= q_1, \\
\delta(q_1, 1) &= q_2, \\
\delta(q_2, 0) &= q_3, \\
\delta(q_2, 1) &= q_2, \\
\delta(q_3, 0) &= q_2, \\
\delta(q_3, 1) &= q_2,
\end{aligned}$$

or more succinctly, we represent δ by the *transition table*

δ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- q_1 is the start state, and
- $F = \{q_2\}$.[1]

If $M = (Q, \Sigma, \delta, s, F)$ is a DFA, how large is δ?

If $A \subseteq \Sigma^*$ is the set of all strings that M accepts, we say that M *recognizes (accepts,decides)* A, and we write $L(M) = A$. [2]

M_1 recognizes the language

$$L(M_1) = \left\{ w \in \{0,1\}^* \;\middle|\; \begin{array}{l} w \text{ contains at least one 1 and an} \\ \text{even number of 0's follow the last 1} \end{array} \right\}$$

Show DFA simulator and file format.

[1] The diagram and this formal description contain exactly the same information. The diagram is easy for humans to read, and the formal description is easy to work with mathematically, and to program.

[2] If a DFA accepts no strings, what language does it recognize?

Example 1.1.2. See Figure 1.8 in Sipser.

Formally, $M_2 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$, where δ is defined

δ	0	1
q_1	q_1	q_2
q_2	q_1	q_2

[3] $L(M_2) = \{\ w \in \{0, 1\}^* \mid w \text{ ends in a } 1\ \}$

Example 1.1.3. See Figure 1.11 in Sipser.

$L(M_4) = \{\ w \in \{\mathtt{a}, \mathtt{b}\}^+ \mid w[0] = w[|w| - 1]\ \}$

1.1.2 More Examples

Example 1.1.4. Design a DFA that recognizes the language

$$\left\{\ \mathtt{a}^{3n} \ \middle|\ n \in \mathbb{N}\ \right\} = \{\ w \in \{\mathtt{a}\}^* \mid |w| \text{ is a multiple of } 3\ \}.$$

$M = (Q, \Sigma, \delta, s, F)$, where

- $Q = \{0, 1, 2\}$,
- $\Sigma = \{\mathtt{a}\}$,
- $s = 0$,
- $F = \{0\}$, and
- δ is defined

δ	$\mathtt{a}$
0	1
1	2
2	0

Example 1.1.5. Design a DFA that recognizes the language

$$\{\ w \in \{0, 1\}^* \mid w \text{ represents a multiple of } 2 \text{ in binary}\ \}.$$

$M = (Q, \Sigma, \delta, s, F)$, where

- $Q = \{0, 1\}$,
- $\Sigma = \{0, 1\}$,
- $s = 0$,
- $F = \{0\}$, and

[3] Talk through the example 1101.

- δ is defined

δ	0	1
0	0	1
1	0	1

.

Example 1.1.6. Design a DFA that recognizes the language

$$\{\ w \in \{0,1\}^* \mid w \text{ represents a multiple of 3 in binary }\}.$$

$M = (Q, \Sigma, \delta, s, F)$, where

- $Q = \{0, 1, 2\}$,
- $\Sigma = \{0, 1\}$,
- $s = 0$,
- $F = \{0\}$, and
- δ is defined

δ	0	1
0	0	1
1	2	0
2	1	2

.

LECTURE: end of day 2

1.1.3 Formal Definition of Computation by a DFA (Semantics)

Sipser defines DFA acceptance this way: We say $M = (Q, \Sigma, \delta, s, F)$ *accepts* a string $x \in \Sigma^n$ if there is a sequence of states $q_0, q_1, q_2, \ldots, q_n \in Q$ such that

1. $q_0 = s$,
2. $q_{i+1} = \delta(q_i, x[i])$ for $i \in \{0, 1, \ldots, n-1\}$, and
3. $q_n \in F$.

There are other ways to define the same concept. I find the following definition to be more work initially, but easier for proofs later.

Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA. Define the *extended transition function*

$$\widehat{\delta} : Q \times \Sigma^* \to Q$$

[4] for all $q \in Q$, $x \in \Sigma^*$, and $b \in \Sigma$ by the recursion

$$\begin{array}{rcl}\widehat{\delta}(q, \lambda) & = & q, \\ \widehat{\delta}(q, xb) & = & \delta(\widehat{\delta}(q, x), b).\end{array}$$

For example, for the machine M_1,

$$\begin{array}{rcl}\widehat{\delta}(q_1, \lambda) & = & q_1, \\ \widehat{\delta}(q_1, 1) & = & q_2, \\ \widehat{\delta}(q_1, 10) & = & q_3, \\ \widehat{\delta}(q_1, 101) & = & q_2.\end{array}$$

We say M *accepts* a string x if $\widehat{\delta}(s, x) \in F$; otherwise M *rejects* x.

Define the *language recognized by* M as

$$L(M) = \{ \ x \in \Sigma^* \mid M \text{ accepts } x \ \}.$$

We say a language L is *regular* if some DFA recognizes it.

1.1.4 The Regular Operations

So far we have read and "programmed" DFAs that recognize particular regular languages. We now study fundamental properties shared by *all* regular languages.

Example 1.1.7. Design a DFA $M = (Q, \{\mathtt{a}\}, \delta, s, F)$ to recognize the language $\{ \ \mathtt{a}^n \mid n \text{ is divisible by 3 or 5} \ \}$.

- $Q = \{ \ (i, j) \mid 0 \leq i < 3, 0 \leq j < 5 \ \}$
- $s = (0, 0)$
- $\delta((i, j), \mathtt{a}) = (i + 1 \mod 3, j + 1 \mod 5)$
- $F = \{ \ (i, j) \mid i = 0 \text{ or } j = 0 \ \}$

M is essentially simulating two DFAs at once: one that computes congruence mod 3 and the other that computes congruence mod 5. We now generalize this idea.

Definition 1.1.8. Let A and B be languages.

Union: $A \cup B = \{ \ x \mid x \in A \text{ or } x \in B \ \}$[5]

Concatenation: $A \circ B = \{ \ xy \mid x \in A \text{ and } y \in B \ \}$[6] (also written $A \circ B$)

[4] Intuitively, we want $\widehat{\delta}(q, w)$ to mean "M's state after reading the string w."

[5] The normal union operation that makes sense for any two sets, which happen to be languages in this case

[6] The set of all strings formed by concatenating one string from A to one string from B; only makes sense for languages because not all types of objects can be concatenated

(Kleene) Star: $A^* = \bigcup_{n=0}^{\infty} A^n = \{\ x_1 x_2 \ldots x_k \mid k \in \mathbb{N} \text{ and } x_1, x_2, \ldots, x_k \in A\ \}$[7]

Each is an operator on one or two languages, with another language as output.

The regular languages are *closed under* each of these operations.[8] This means that if A, B are regular, $A \cup B$, $A \circ B$, and A^* are all regular also.[9] The proofs of each of these facts are something like the "divisible by 3 or 5" DFA above: we show how to simulate some DFAs (that recognize the languages A and B) with another DFA (that recognizes $A \cup B$, or $A \circ B$, or A^*).

Theorem 1.1.9. *The class of regular languages is closed under* $\cup$.

[10]

Proof. (Product Construction)[11]

[12]

Let $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ be DFAs. We construct the DFA $M = (Q, \Sigma, \delta, s, F)$ to recognize $L(M_1) \cup L(M_2)$ by simulating both M_1 and M_2 in parallel, where

- Q keeps track of the states of both M_1 and M_2:

$$Q = Q_1 \times Q_2\, (= \{\ (r_1, r_2) \mid r_1 \in Q_1 \text{ and } r_2 \in Q_2\ \})$$

- δ simulates moving both M_1 and M_2 one step forward in response to the input symbol. Define δ for all $(r_1, r_2) \in Q$ and all $b \in \Sigma$ as

$$\delta(\ (r_1, r_2)\ ,\ b\ ) = (\ \delta_1(r_1, b)\ ,\ \delta_2(r_2, b)\ )$$

- s ensures both M_1 and M_2 start in their respective start states:

$$s = (s_1, s_2)$$

[7] The set of all strings formed by concatenating 0 or more strings from A. Just like with Σ^*, except now whole strings may be concatenated instead of individual symbols.

[8] Note that this is **not the same** as the property of a subset $A \subseteq \mathbb{R}^d$ being closed in the sense that it contains all of its limit points. The concepts are totally different, even if the same English word "closed" is used in each definition. In particular, one typically does not talk about any class of languages being merely "closed," but rather, closed *with respect to a certain operation* such as union, concatenation, or Kleene star.

[9] The intuition for the terminology is that if you think of the class of regular languages as being like a room, and doing an operation as being like moving from one language (or languages) to another, then moving via union, concatenation, or star won't get you out of the room; the room is "closed" with respect to moving via those operations.

[10] **Proof Idea:** (The *Product Construction*)

We must show that if A_1 and A_2 are regular, then so is $A_1 \cup A_2$. Since A_1 and A_2 are regular, some DFA M_1 recognizes A_1, and some DFA M_2 recognizes A_2. It suffices to show that some DFA M recognizes $A_1 \cup A_2$; i.e., it accepts a string x if and only if at least one of M_1 or M_2 accepts x.

M will simulate M_1 and M_2. If either accepts the input string, then so will M. M must simulate them simultaneously, because if it tried to simulate M_1, then M_2, it could not remember the input to supply it to M_2

[11] The name comes from the cross product used to define the state set.

[12] In this proof, I will write more than normal. In the future I will say out loud the explanation that makes the proof understandable, but refrain from writing it all for the sake of time. On your homework, you should write these explanations, so the TA will understand the proof.

- F must be accepting exactly when either one or the other or both of M_1 and M_2 are in an accepting state:
$$F = \{\ (r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\ \}.$$
[13]

Let $x \in \Sigma^*$. Then

$$\begin{aligned}
x \in L(M) &\iff \widehat{\delta}(s,x) \in F && \text{defn of } L() \\
&\iff \Big(\ \widehat{\delta}_1(s_1,x)\ ,\ \widehat{\delta}_2(s_2,x)\ \Big) \in F && \text{this claim shown below} \\
&\iff \widehat{\delta}_1(s_1,x) \in F_1 \text{ or } \widehat{\delta}_2(s_2,x) \in F_2 && \text{defn of } F \\
&\iff x \in L(M_1) \text{ or } x \in L(M_2) && \text{defn of } L() \\
&\iff x \in L(M_1) \cup L(M_2), && \text{defn of } \cup
\end{aligned}$$

whence $L(M) = L(M_1) \cup L(M_2)$. Now we justify the claim above that $\widehat{\delta}(s,x) = (\widehat{\delta}_1(s_1,x), \widehat{\delta}_2(s_2,x))$ by induction on x.[14] In the base case $x = \lambda$, then

$$\begin{aligned}
\widehat{\delta}(s,\lambda) =& s && \text{base defn of } \widehat{\delta} \\
=& (s_1, s_2) && \text{defn of } s \\
=& \Big(\ \widehat{\delta}_1(s_1,\lambda)\ ,\ \widehat{\delta}_2(s_2,\lambda)\ \Big) && \text{base defn of } \widehat{\delta}_1 \text{ and } \widehat{\delta}_2
\end{aligned}$$

Now inductively assume that for $x \in \Sigma^*$, $\widehat{\delta}(s,x) = (\widehat{\delta}_1(s_1,x), \widehat{\delta}_1(s_2,x))$. Let $b \in \Sigma$. Then

$$\begin{aligned}
\widehat{\delta}(s,xb) =& \delta\Big(\ \widehat{\delta}(s,x)\ , b\Big) && \text{ind. defn of } \widehat{\delta} \\
=& \delta\Big(\ \Big(\widehat{\delta}_1(s_1,x), \widehat{\delta}_2(s_2,x)\Big)\ , b\Big) && \text{ind. hyp.} \\
=& \Big(\ \delta_1\Big(\widehat{\delta}_1(s_1,x), b\Big)\ ,\ \delta_2\Big(\widehat{\delta}_2(s_2,x), b\Big)\ \Big) && \text{defn of } \delta \\
=& \Big(\ \widehat{\delta}_1(s_1,xb)\ ,\ \widehat{\delta}_2(s_2,xb)\ \Big), && \text{ind. defn of } \widehat{\delta}_1 \text{ and } \widehat{\delta}_2
\end{aligned}$$

whence the claim holds for xb.[15] □

Theorem 1.1.10. *The class of regular languages is closed under complement.*

[13] This is *not* the same as $F = F_1 \times F_2$. What would the machine do if we defined $F = F_1 \times F_2$?

[14] Although we do induction manually in this class, it is common in theoretical computer science papers to see a claim such as that above left unproven (often with the flippant remark, "this is easily shown with a trivial induction on x"). This is because claims of the form "what I stated holds for the one-symbol-at-a-time function δ, so it obviously holds for the extended-to-whole-strings function $\widehat{\delta}$" are typically verified by induction. But when an author claims it is "easily verified", they mean that it is easy for someone who has already taken a course like 120, so as to get to the point where they could easily reproduce the induction. In other words, you aren't allowed to claim something is "a trivial induction" until you could produce that induction in your sleep; until that point, you have to prove it explicitly.

[15] Now go back to the claim and verify that the final chain of inequalities actually is what we intended to prove. It is, but it is difficult to remember what claim was intended to be proven after such mental gymnastics.

Proof Idea: Swap the accept and reject states.

Theorem 1.1.11. *The class of regular languages is closed under* $\cap$.

Proof Idea: DeMorgan's Laws.

Proof. Let A, B be regular languages; then

$$A \cap B = \overline{\overline{A} \cup \overline{B}}$$

is the complement of the union of two languages $\overline{A}$ and $\overline{B}$ that are the complements of regular languages. By the union and complement closure properties, this language is also regular. □

Note that we have only proved that the regular languages are closed under a *single* application of $\cup$ or $\cap$ to two regular languages A_1 and A_2. You can use induction to prove that they are closed under *finite* union and intersection; for instance, if $A_1, \ldots, A_k$ are regular, then $\bigcup_{i=1}^{k} A_i$ is regular, for any $k \in \mathbb{N}$.[16]

Finally, we show another nontrivial closure property of the regular languages.

Theorem 1.1.12. *The class of regular languages is closed under* $\circ$.

[17]

We prove this by introducing a tool to help us.

LECTURE: end of day 3

1.2 Nondeterminism

Reading assignment: Section 1.2 in Sipser.

We now reach a central motif in theoretical computer science:

[16] What about infinite union or infinite intersection?

[17] In other words, we need to construct M so that, if M_1 accepts x and M_2 accepts y, then M accepts xy. If we try to reason about this like with the union operation, there is one additional complication. We are no longer simulating the machines on the same input; rather, x precedes y in the string xy. We could say, "build M to simulate M_1 on x, and after the x portion of xy has been read, then simulate M_2 on y."

What is wrong with this idea? When x and y are concatenated to form xy, we no longer know where x ends and y begins. So M would not know when to switch from simulating M_1 to simulating M_2.

We now study nondeterminism as an (apparently) more powerful mechanism for recognizing languages with finite automata. Surprisingly, it turns out it is no more powerful, but is easier think about for certain problems such as this one.

Proof Technique: When you want to prove that an algorithm can solve a problem, brazenly assume your algorithm has a magic ability that you can use to cheat. Then prove that adding adding the magic ability does not improve the fundamental power of the algorithm.[18]

Deterministic finite automata (DFA):	realistic	difficult to program
Nondeterministic finite automata (NFA):	(apparently) unrealistic	easy to program

Example 1.2.1. Design a finite automaton to recognize the language $\{\ x \in \{0,1\}^* \mid x[|x|-3] = 0\ \}$.

See Figure 1.27 in Sipser.
Differences between DFA's and NFA's:

- An NFA state may have any number (including 0) of transition arrows out of a state, for the same input symbol.[19]
- An NFA may change states without reading *any* input (a λ-*transition*).
- If there is a series of choices (when there is a choice) to reach an accept state, then the NFA accepts. If there is no series of choices that leads to an accept state, the NFA rejects.[20]

Example 1.2.2. Design an NFA to recognize the language

$$\{\ x \in \{\mathtt{a}\}^* \mid |x| \text{ is a multiple of 2 or 3 or 5 }\}$$

1.2.1 Formal Definition of an NFA (Syntax)

Definition 1.2.3. A *nondeterministic finite automaton* (*NFA*) is a 5-tuple $(Q, \Sigma, \Delta, s, F)$, where

- Q is a non-empty, finite set of *states*,
- Σ is the *input alphabet*,
- $\Delta : Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q)$ is the *transition function*,
- $s \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accepting states*.

When defining Δ, we assume that if for some $q \in Q$ and $b \in \Sigma \cup \{\lambda\}$, $\Delta(q, b)$ is not explicitly defined, then $\Delta(q, b) = \emptyset$.

Example 1.2.4. See Example 1.38 in Sipser.
The formal description of N_1 is $(Q, \Sigma, \Delta, q_1, F)$, where

[18] This proves it could have been done without the magic ability.

[19] Unlike a DFA, an NFA may attempt to read a symbol a in a state q that has no transition arrow for a; in this case the NFA is interpreted as immediately rejecting the string.

[20] Note that accepting and rejecting are treated asymmetrically; if there is a series of choices that leads to accept and there is another series of choices that leads to reject, then the NFA accepts.

- $Q = \{q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{0, 1\}$,
- Δ is defined

Δ	0	1	λ
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_2\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

- q_1 is the start state, and
- $F = \{q_4\}$.

1.2.2 Formal Definition of Computation by an NFA (Semantics)

An NFA $N = (Q, \Sigma, \Delta, s, F)$ *accepts* a string $x \in \Sigma^*$ if there are sequences $y_0, y_1, y_2, \ldots, y_{m-1} \in \Sigma \cup \{\lambda\}$ and $q_0, q_1, \ldots, q_{m+1} \in Q$ such that

1. $x = y_0 y_1 \ldots y_{m-1}$,
2. $q_0 = s$,
3. $q_{i+1} \in \Delta(q_i, y[i])$ for $i \in \{0, 1, \ldots, m\}$, and
4. $q_m \in F$.

With NFAs, strings appear

- easier to accept, but
- harder to reject.[21]

[21] The trick with NFAs is that it becomes (apparently) easier to accept a string, since multiple paths through the NFA could lead to an accept state, and only one must do so in order to accept. But NFAs aren't magic; you can't simply put accept states and λ-transitions everywhere and claim that there exist paths doing what you want, so the NFA works.

By the same token that makes acceptance easier, rejection becomes more difficult, because you must ensure that *no* path leads to acceptance if the string ought to be rejected. Therefore, the more transitions and accept states you throw in to make accepting easier, that much more difficult does it become to design the NFA to properly reject. The key difference is that the condition "*there exists* a path to an accept state" becomes, when we negate it to define rejection, "*all* paths lead to a reject state". It is of course, more difficult to verify a "for all" claim than a "there exists" claim.

1.2.3 Equivalence of DFAs and NFAs

Theorem 1.2.5. *For every DFA M, there is an NFA N such that $L(N) = L(M)$.*

Proof. A DFA $M = (Q, \Sigma, \delta, s, F)$ is an NFA $N = (Q, \Sigma, \Delta, s, F)$ with no λ-transitions and, for every $q \in Q$ and $a \in \Sigma$, $\Delta(q, a) = \{\delta(q, a)\}$. □

LECTURE: end of day 4

We now show the converse of Theorem 1.2.5.

Theorem 1.2.6. *For every NFA N, there is a DFA D such that $L(N) = L(D)$.*

Show example 1.41, with figures 1.42, 1.43, and 1.44 from textbook.

Proof. (Subset Construction) Let $N = (Q_N, \Sigma, \Delta, s_N, F_N)$ be an NFA with no λ-transitions.[22] Define the DFA $D = (Q_D, \Sigma, \delta, s_D, F_D)$ as follows

- $Q_D = \mathcal{P}(Q_N)$. Each state of D keeps track of a set of states in N, representing the set of all states N could be in after reading some portion of the input.
- For all $R \in Q_D$ (i.e., all $R \subseteq Q_N$) and $b \in \Sigma$,

  $$\delta(R, b) = \bigcup_{q \in R} \Delta(q, b),$$

  If N is in state $q \in R$ after reading some portion of the input, then the states could it be in after reading the next symbol b are all the states in $\Delta(q, b)$; since N could be in any state $q \in R$ before reading b, then we must take the union over all $q \in R$.
- $s_D = \{s_N\}$, After reading no input, N can only be in state s_N.
- $F_D = \{ \ A \subseteq Q_N \ | \ A \cap F_N \neq \emptyset \ \}$. Recall the asymmetric acceptance criterion; we want to accept if there is a way to reach an accept state, i.e., if the set of states N could be in after reading the whole input contains any accept states.

Now we show how to handle the λ-transitions. For any $R \subseteq Q_N$ and define

$$E(R) = \{ \ q \in Q \ | \ q \text{ is reachable from some state in } R \text{ by following 0 or more } \lambda\text{-transitions } \}.$$

Show example picture of some states R and $E(R)$

To account for the λ-transitions, D must be able to simulate

[22] At the end of the proof we explain how to modify the construction to handle them.

1. N following λ-transitions *after* each non-λ-transition, i.e., define
$$\delta(R,b) = E\left(\bigcup_{q\in R} \Delta(q,b)\right).$$
2. N following λ-transitions *before* the first non-λ-transition, i.e., define $s_D = E(\{s_N\})$.

We have constructed a DFA D such that for all $x \in \Sigma^*$, $\widehat{\delta}(s_D, x) \subseteq Q_N$ is the subset of states that N could be in after reading x. By the definition of F_D, D accepts $x \iff \widehat{\delta}(s_D,x) \in F_D \iff \widehat{\delta}(s_D,x) \cap F_N \neq \emptyset$, i.e., if and only if some state $q \in \widehat{\delta}(s_D,x)$ is accepting, which by the definition of NFA acceptance occurs if and only if N accepts x. Thus $L(D) = L(N)$. □

Note that the subset construction uses the power set of Q_N, which is exponentially larger than Q_N. It can be shown (see Kozen's textbook) that there are languages for which this is necessary; the smallest NFA recognizing the language has n states, while the smallest DFA recognizing the language has 2^n states. For example, for any $n \in \mathbb{N}$, the language $\{\ x \in \{0,1\}^* \mid x[|x| - n] = 0\ \}$ has this property.

Corollary 1.2.7. *The class of languages recognized by some NFA is precisely the regular languages.*

Proof. This follows immediately from Theorems 1.2.5 and 1.2.6. □

Theorem 1.2.8. *The class of regular languages is closed under* $\circ$.

Proof. Let $N_1 = (Q_1, \Sigma, \Delta_1, s_1, F_1)$ and $N_2 = (Q_2, \Sigma, \Delta_2, s_2, F_2)$ be NFAs. It suffices to define the NFA $N = (Q, \Sigma, \Delta, s, F)$ such that N recognizes
$$L(N_1) \circ L(N_2) = \{\ xy \in \Sigma^* \mid x \in L(N_1) \text{ and } y \in L(N_2)\ \}.$$

Show example picture from Sipser

Define

- $Q = Q_1 \cup Q_2$,
- $s = s_1$,
- $F = F_2$, and
- Δ is defined for all $q \in Q$ and $b \in \Sigma \cup \{\lambda\}$ by
$$\Delta(q,b) = \begin{cases} \Delta_1(q,b), & \text{if } q \in Q_1 \text{ and } q \notin F_1; \\ \Delta_1(q,b), & \text{if } q \in Q_1 \text{ and } b \neq \lambda; \\ \Delta_2(q,b), & \text{if } q \in Q_2; \\ \Delta_1(q,b) \cup \{s_2\}, & \text{if } q \in F_1 \text{ and } b = \lambda. \end{cases}$$

To see that $L(N_1) \circ L(N_2) \subseteq L(N)$. Let $w \in \Sigma^*$. If there are $x \in L(N_1)$ and $y \in L(N_2)$ such that $w = xy$ (i.e., $w \in L(N_1) \circ L(N_2)$), then there is a sequence of choices of N such that N accepts w (i.e., $q \in L(N)$): follow the choices N_1 makes to accept x, ending in a state in F_1, then execute the λ-transition to state s_2 defined above, then follow the choices N_2 makes to accept y. This shows $L(N_1) \circ L(N_2) \subseteq L(N)$.

To see the reverse containment $L(N) \subseteq L(N_1) \circ L(N_2)$, suppose $w \in L(N)$. Then there is a sequence of choices such that N accepts w. By construction, all paths from $s = s_1$ to some state in $F = F_2$ pass through s_2, so N must reach s_2 after reading some prefix $x \sqsubseteq w$, and the remaining suffix y of w takes N from s_2 to a state in F_2, i.e., $y \in L(N_2)$. By construction, all paths from $s = s_1$ to s_2 go through a state in F_1, and those states are connected to s_2 only by a λ-transition, so x takes N from s_1 to a state in F_2, i.e., $x \in L(N_1)$. Since $w = xy$, this shows that $L(N) \subseteq L(N_1) \circ L(N_2)$.

Thus N recognizes $L(N_1) \circ L(N_2)$. □

LECTURE: end of day 5

Theorem 1.2.9. *The class of regular languages is closed under* *.

Proof. Let $D = (Q_D, \Sigma, \delta, s_D, F_D)$ be an DFA. It suffices to define the NFA $N = (Q_N, \Sigma, \Delta, s_N, F_N)$ such that N recognizes

$$L(D)^* = \bigcup_{k=0}^{\infty} L(D)^k = \{\ x_1x_2\ldots x_k \in \Sigma^* \ \mid\ k \in \mathbb{N} \text{ and, for all } 0 \leq i \leq k,\ x_i \in L(D)\ \}.$$

Show example picture from Sipser

Define

- $Q_N = Q_D \cup \{s_N\}$,
- $F_N = F_D \cup \{s_N\}$, and
- Δ' is defined for all $q \in Q'$ and $b \in \Sigma \cup \{\lambda\}$ by

$$\Delta'(q,b) = \begin{cases} \Delta(q,b), & \text{if } q \in Q_D \text{ and } q \notin F_D; \\ \Delta(q,b), & \text{if } q \in F_D \text{ and } b \neq \lambda; \\ \Delta(q,b) \cup \{s_D\}, & \text{if } q \in F_D \text{ and } b = \lambda; \\ \{s_D\}, & \text{if } q = s_N \text{ and } b = \lambda; \\ \emptyset, & \text{if } q = s_N \text{ and } b \neq \lambda. \end{cases}$$

Clearly N accepts $\lambda \in L(D)^*$, so we consider only nonempty strings.

To see that $L(D)^* \subseteq L(N)$, suppose $w \in L(D)^* \setminus \{\lambda\}$. Then $w = x_1 x_2 \dots x_k$, where each $x_j \in L(D) \setminus \{\lambda\}$. Thus for each $j \in \{1, \dots, k\}$, there is a sequence of states $s_D = q_{j,0}, q_{j,1}, \dots, q_{j,|x_j|} \in Q$, where $q_{j,|x_j|} \in F$, and a sequence $y_{j,0}, \dots, y_{j,|x_j|-1} \in \Sigma$, such that $x_j = y_{j,0} \dots y_{j,|x_j|-1}$, and for each $i \in \{0, \dots, |x_j| - 1\}$, $q_{j,i+1} = \delta(q_{j,i}, y_{j,i})$. The following sequence of states in Q_N testifies that N accepts w:

$$\begin{array}{llll}(s_N, & s_D, q_{1,1}, & \dots, & q_{1,|x_1|}, \\ & s_D, q_{2,1}, & \dots, & q_{2,|x_2|}, \\ & & \dots & \\ & s_D, q_{k,1}, & \dots, & q_{k,|x_k|}).\end{array}$$

Each transition between adjacent states in the sequence is either one of the $\Delta(q_{j,i}, y_{j,i})$ listed above, or is the λ-transition from s_N to s_D or from $q_{j,|x_j|}$ to s_D. Since $q_{k,|x_k|} \in F \subseteq F'$, N' accepts w, i.e., $L(N)^* \subseteq L(N')$.

To see that $L(N') \subseteq L(N)^*$, let $w \in L(N') \setminus \{\lambda\}$. Then there are sequences $s' = q_0, q_1 \dots, q_m \in Q'$ and $y_0, \dots, y_{m-1} \in \Sigma \cup \{\lambda\}$ such that $q_m \in F'$, $w = y_0 y_1 \dots y_{m-1}$, and, for all $i \in \{0, \dots, m-1\}$, $q_{i+1} \in \Delta'(q_i, y[i])$. Since $w \neq \lambda$, $q_m \neq s'$, so $q_m \in F$. Since the start state is $s' = q_0$, which has no outgoing non-λ-transition, the first transition from q_0 to q_1 is the λ-transition s' to s. Suppose there are $k-1$ λ-transitions from a state in F to s in $q_1, \dots, q_m$. Then we can write $q_1, \dots, q_m$ as

$$\begin{array}{llll}(& s_D, q_{1,1}, & \dots, & q_{1,|x_1|}, \\ & s_D, q_{2,1}, & \dots, & q_{2,|x_2|}, \\ & & \dots & \\ & s_D, q_{k,1}, & \dots, & q_{k,|x_k|}).\end{array}$$

where each $q_{j,|x_j|} \in F$ and has a λ-transition to s.[23] For each $j \in \{1, \dots, k)$ and $i \in \{0, |x_j| - 1\}$, let $y_{j,i}$ be the corresponding symbol in $\Sigma \cup \{\lambda\}$ causing the transition from $q_{j,i}$ to $q_{j,i+1}$, and let $x_j = y_{j,0} y_{j,1} \dots y_{j,|x_j|-1}$. Then the sequences $s, q_{j,1} q_{j,2} \dots q_{j,|x_j|}$ and $y_{j,0} y_{j,1} \dots y_{j,m_j-1}$ testify that N accepts x_j, thus $x_j \in L(N)$. Thus $w = x_1 x_2 \dots x_k$, where each $x_j \in L(N)$, so $w \in L(N)^*$, showing $L(N') \subseteq L(N)^*$.

Thus N' recognizes $L(N)^*$. □

1.3 Regular Expressions

Reading assignment: Section 1.3 in Sipser.

A *regular expression* is a pattern that matches some set of strings, used in many programming languages and applications such as `grep`. For example, the regular expression

$$(0 \cup 1)0^*$$

matches any string starting with a 0 or 1, followed by zero or more 0's.[24]

$$(0 \cup 1)^*$$

[23] Perhaps there are no such λ-transitions, in which case $k = 1$.

[24] The notation $(0|1)0^*$ is more common, but we use the $\cup$ symbol in place of | to emphasize the connections to set theory, and because | looks too much like 1.

matches any string consisting of zero or more 0's and 1's; i.e., any binary string.

Let $\Sigma = \{0, 1, 2\}$. Then

$$(0\Sigma^*) \cup (\Sigma^* 1)$$

matches any ternary string that either starts with a 0 or ends with a 1. The above is shorthand for

$$(0(0 \cup 1 \cup 2)^*) \cup ((0 \cup 1 \cup 2)^* 1)$$

since $\Sigma = \{0\} \cup \{1\} \cup \{2\}$.

Each regular expression R defines a language $L(R)$.

1.3.1 Formal Definition of a Regular Expression

Definition 1.3.1. Let Σ be an alphabet.

R is a *regular expression (regex)* if R is

1. b for some $b \in \Sigma$, defining the language $\{b\}$,
2. λ, defining the language $\{\lambda\}$,
3. $\emptyset$, defining the language $\{\}$,
4. $R_1 \cup R_2$, where R_1, R_2 are regex's, defining the language $L(R_1) \cup L(R_2)$,
5. $R_1 R_2$ (or $R_1 \circ R_2$), where R_1, R_2 are regex's, defining the language $L(R_1) \circ L(R_2)$, or
6. R^*, where R is a regex, defining the language $L(R)^*$.

The operators have precedence $* > \circ > \cup$. Parentheses may be used to override this precedence.

We sometimes abuse notation and write R to mean $L(R)$ and rely on context to interpret the meaning.

For convenience, define $R^+ = RR^*$, for each $k \in \mathbb{N}$, let $R^k = \underbrace{RR\ldots R}_{k \text{ times}}$, and given an alphabet $\Sigma = \{a_1, a_2, \ldots, a_k\}$, then Σ is shorthand for the regex $a_1 \cup a_2 \cup \ldots \cup a_k$.

Example 1.3.2. Let Σ be an alphabet.

- $0^*10^* = \{\ w \mid w \text{ contains a single } 1\ \}$.
- $\Sigma^* 1 \Sigma^* = \{\ w \mid w \text{ has at least one } 1\ \}$.
- $\Sigma^* 001 \Sigma^* = \{\ w \mid w \text{ contains the substring } 001\ \}$.
- $1^*(01^+)^* = \{\ w \mid \text{ every } 0 \text{ in } w \text{ is followed by at least one } 1\ \}$.
- $(\Sigma\Sigma)^* = \{\ w \mid |w| \text{ is even }\ \}$.
- $01 \cup 10 = \{01, 10\}$.
- $0\ (0{\cup}1)^*\ 0 \cup 1\ (0{\cup}1)^*\ 1 \cup 0 \cup 1 = \{\ w \in \{0,1\}^* \mid w \text{ starts and ends with the same symbol }\}$

- $(0 \cup \lambda)1^* = 01^* \cup 1^*$.
- $(0 \cup \lambda)(1 \cup \lambda) = \{\lambda, 0, 1, 01\}$.
- $1^*\emptyset = \emptyset$.
- $\emptyset^* = \{\lambda\}$.
- $R \cup \emptyset = R$, where R is any regex.
- $R \circ \lambda = R$, where R is any regex.

Example 1.3.3. Design a regular expression to match C++ `float` literals (e.g., `2`, `3.14`, `-.02`, `0.02`, `.02`, `3.`).

Let $D = 0 \cup 1 \cup \ldots \cup 9$ be a regex recognizing a single decimal digit.

$$(+ \cup - \cup \lambda)(D^+ \cup D^+.D^* \cup D^*.D^+)$$

LECTURE: end of day 6

1.3.2 Equivalence with Finite Automata

Theorem 1.3.4. *A language is regular if and only if some regular expression defines it.*

We prove each direction separately via two lemmas.

Lemma 1.3.5. *If a language is defined by a regular expression, then it is regular.*

Proof. Let R be a regular expression over an alphabet Σ. It suffices to construct an NFA $N = (Q, \Sigma, \Delta, s, F)$ such that $L(R) = L(N)$.

The definition of regex gives us six cases:

1. $R = b$, where $b \in \Sigma$, so $L(R) = \{b\}$, recognized by the NFA $N = (\{q_1, q_2\}, \Sigma, \Delta, q_1, \{q_2\})$, where $\Delta(q_1, b) = \{q_2\}$.
2. $R = \lambda$, so $L(R) = \{\lambda\}$, recognized by the NFA $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$ (no transitions).
3. $R = \emptyset$, so $L(R) = \emptyset$, recognized by the NFA $N = (\{q_1\}, \Sigma, \delta, q_1, \emptyset)$ (no transitions).
4. $R = R_1 \cup R_2$, so $L(R) = L(R_1) \cup L(R_2)$.
5. $R = R_1 \circ R_2$, so $L(R) = L(R_1) \circ L(R_2)$.
6. $R = R_1^*$, so $L(R) = L(R_1)^*$.

For the last three cases, assume inductively that $L(R_1)$ and $L(R_2)$ are regular. Since the regular languages are closed under the operations of $\cup$, $\circ$, and *, R is regular. □

See examples 1.56 and 1.58 in Sipser.

Lemma 1.3.6. *If a language is regular, then it is defined by a regular expression.*

Proof. Let $D = (Q, \Sigma, \delta, s, F)$ be a DFA. It suffices to construct a regex R such that $L(R) = L(D)$.

Given $P \subseteq Q$ and $u, v \in Q$, we will construct a regex R^P_{uv} to define the language

$$L(R^P_{uv}) = \left\{ \, x \in \Sigma^* \;\middle|\; v = \widehat{\delta}(u, x) \text{ and } (\forall i \in \{1, 2, \ldots, |x| - 1\}) \; \widehat{\delta}(u, x[0\,.\,.\,i-1]) \in P \, \right\}.$$

draw picture

That is, $x \in L(R^P_{uv})$ if and only if input x takes D from state u to state v "entirely through P, except possibly at the start and end". The construction of R^P_{uv} is inductive on P.[25]

Base case: Let $P = \emptyset$, and let $b_1, \ldots, b_k \in \Sigma$ be the symbols such that $v = \delta(u, b_i)$. If $u \neq v$, define

$$R^{\emptyset}_{uv} = \begin{cases} b_1 \cup \ldots \cup b_k, & \text{if } k \geq 1; \\ \emptyset, & \text{if } k = 0. \end{cases}$$

If $u = v$, define

$$R^{\emptyset}_{uu} = \begin{cases} b_1 \cup \ldots \cup b_k \cup \lambda, & \text{if } k \geq 1; \\ \lambda, & \text{if } k = 0. \end{cases}$$

The λ represents "traversing" from u to u without reading a symbol, by simply sitting still.

Inductive case: Let $P \subseteq Q$ be nonempty. Choose $q \in P$ arbitrarily, and define

$$R^P_{uv} = R^{P\setminus\{q\}}_{uv} \cup R^{P\setminus\{q\}}_{uq} \left(R^{P\setminus\{q\}}_{qq} \right)^* R^{P\setminus\{q\}}_{qv}.$$

draw picture

That is, either x traverses from u to v through P without ever visiting q, or it visits q one or more times, and *between* each of these visits, traverses within P without visiting q. [26]

To finish the proof, observe that the regex

$$R = R^Q_{sf_1} \cup R^Q_{sf_2} \cup \ldots \cup R^Q_{sf_k},$$

where $F = \{f_1, f_2, \ldots, f_k\}$, defines $L(D)$, since the set of strings accepted by D is precisely those strings that follow a path from the start state s to some accept state f_i, staying within the entire state set Q. □

[25] That is, we assume inductively that the regex's $R^{P'}_{u'v'}$ can be constructed for proper subsets $P' \subsetneq P$ and all pairs of states $u', v' \in Q$, and use these to define R^P_{uv}).

[26] Suppose x follows a path from u to v through P. It either visits state q or not. If not, then the first sub-regex $R^{P\setminus\{q\}}_{uq}$ will match x. If so, then it will get to q, take 0 or more paths through $P \setminus \{q\}$, revisiting q after each one, then proceed within $P \setminus \{q\}$ from q to v. The second sub-regex expresses this. Therefore R^P_{uv} matches x. Since these two sub-regex's express the only two ways for x to follow a path from u to v through P, the converse holds that R^P_{uv} matching x implies x follows a path from u to v through P.

Example: convert the DFA $(\{1,2\},\{\mathtt{a},\mathtt{b}\},\delta,1,\{2\})$ to a regular expression using the given procedure, where

δ	a	b
1	1	2
2	2	2

In the first step of recursion, we choose $q = 2$, i.e., we define $R_{12}^{\{1,2\}}$ inductively in terms of $R_{uv}^{\{1,2\}\setminus\{2\}} = R_{uv}^{\{1\}}$.

$$\begin{aligned}
R_{12}^{\{1,2\}} &= R_{12}^{\{1\}} \cup R_{12}^{\{1\}} \left(R_{22}^{\{1\}}\right)^* R_{22}^{\{1\}} \\
R_{12}^{\{1\}} &= R_{12}^{\emptyset} \cup R_{11}^{\emptyset}(R_{11}^{\emptyset})^* R_{12}^{\emptyset} \\
R_{22}^{\{1\}} &= R_{22}^{\emptyset} \cup R_{21}^{\emptyset}(R_{11}^{\emptyset})^* R_{12}^{\emptyset} \\
R_{12}^{\emptyset} &= \mathtt{b} \\
R_{22}^{\emptyset} &= \mathtt{a} \cup \mathtt{b} \cup \lambda \\
R_{21}^{\emptyset} &= \emptyset \\
R_{11}^{\emptyset} &= \mathtt{a} \cup \lambda
\end{aligned}$$

We use the identity below that for any regex X, $(X \cup \lambda)^* = X^*$ (since concatenating λ any number of times does not alter a string). We also use this observation: for any regex's X and Y, $X \cup XYY^* = XY^*$, since X means X followed by 0 Y's, and XYY^* is X followed by 1 or more Y's, so their union is the same as X followed by 0 or more Y's, i.e., XY^*. Similarly, $X \cup YY^*X = Y^*X$.

Substituting the definitions of $R_{11}^{\emptyset}, R_{12}^{\emptyset}, R_{21}^{\emptyset}, R_{22}^{\emptyset}$, we have

$$\begin{aligned}
R_{12}^{\{1\}} &= \mathtt{b} \cup (\mathtt{a} \cup \lambda)(\mathtt{a} \cup \lambda)^*\mathtt{b} \\
&= (\mathtt{a} \cup \lambda)^*\mathtt{b} && \text{since } X \cup YY^*X = Y^*X \\
&= \mathtt{a}^*\mathtt{b} && \text{since } (X \cup \lambda)^* = X^* \\
R_{22}^{\{1\}} &= (\mathtt{a} \cup \mathtt{b} \cup \lambda) \cup \emptyset(\mathtt{a} \cup \lambda)^*\mathtt{b} \\
&= (\mathtt{a} \cup \mathtt{b} \cup \lambda) && \text{since } \emptyset X = \emptyset
\end{aligned}$$

Substituting the definitions of $R_{12}^{\{1\}}, R_{22}^{\{1\}}$, we have

$$\begin{aligned}
R_{12}^{\{1,2\}} &= R_{12}^{\{1\}} \cup R_{12}^{\{1\}} \left(R_{22}^{\{1\}}\right)^* R_{22}^{\{1\}} \\
&= \mathtt{a}^*\mathtt{b} \cup (\mathtt{a}^*\mathtt{b})(\mathtt{a} \cup \mathtt{b} \cup \lambda)^*(\mathtt{a} \cup \mathtt{b} \cup \lambda) \\
&= \mathtt{a}^*\mathtt{b}(\mathtt{a} \cup \mathtt{b} \cup \lambda)^* && \text{since } X \cup XYY^* = XY^* \\
&= \mathtt{a}^*\mathtt{b}(\mathtt{a} \cup \mathtt{b})^* && \text{since } (X \cup \lambda)^* = X^*
\end{aligned}$$

In other words, $L(R)$ is the set of strings with at least one b.

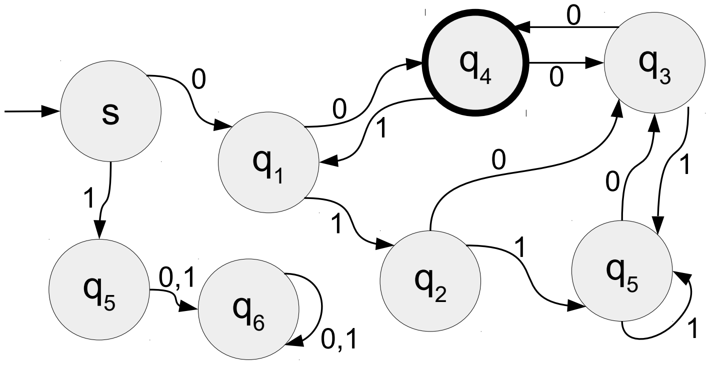

Figure 1.1: A DFA $D = (Q, \Sigma, \delta, s, F)$ to help illustrate the idea of the pumping lemma.

LECTURE: end of day 7

1.4 Nonregular Languages

Reading assignment: Section 1.4 in Sipser.

Consider the language

$$B = \{\ 0^n 1^n \mid n \in \mathbb{N}\ \}.$$

B is not regular.[27]

Recall that given strings a, b, $\#(a, b)$ denotes the number of times that a appears as a substring of b. Consider the languages

$$\begin{aligned} C &= \{\ x \in \{0,1\}^* \mid \#(0,x) = \#(1,x)\ \},\ \text{and} \\ D &= \{\ x \in \{0,1\}^* \mid \#(01,x) = \#(10,x)\ \}. \end{aligned}$$

C is not regular, but D is.

1.4.1 The Pumping Lemma

The pumping lemma is based on the pigeonhole principle. The proof informally states, "If an input to a DFA is long enough, then some state must be visited twice, and the substring between the two visitations can be repeated without changing the DFA's final answer."[28]

See the DFA in Figure 1.1. Consider states reached by reading various strings.

[27] Intuitively, it appears to require unlimited memory to count all the 0's. But we must prove this formally; the next example shows that a language can appear to require unbounded memory while actually being regular.

[28] That is, we "pump" more copies of the substring into the full string. The goal is to pump until we change the membership of the string in the language, at which point we know the DFA cannot decide the language, since it gives the same answer on two strings, one of which is in the language and the other of which is not.

Note that if we reach, for example, q_1, then read the bits 1001, we will return to q_1. Therefore, for any string x such that $\widehat{\delta}(s, x) = q_1$, it follows that $\widehat{\delta}(s, x1001) = q_1$, and $\widehat{\delta}(s, x10011001) = q_1$, etc. i.e., for all $n \in \mathbb{N}$, defining $y = 1001$, $\widehat{\delta}(s, xy^n) = q_1$.

Also notice that if we are in state q_1, and we read the string 1100, we end up in accepting state q_4. Therefore, defining $z = 1100$, $\widehat{\delta}(s, xz) = q_4$, thus D accepts xz. But combined with the previous reasoning, we also have that D accepts $xy^n z$ for all $n \in \mathbb{N}$.

More generally, for any DFA D, any string w with $|w| \geq |Q|$ has to cause D to traverse a cycle. The substring y read along this cycle can either be removed, or more copies added, and the DFA will end in the same state. Calling x the part of w before y and z the part after, we have that whatever is D's answer on $w = xyz$, it is the same on $xy^n z$ for any $n \in \mathbb{N}$, since all of those strings take D to the same state.

Pumping Lemma. *For every DFA D, there is a number $p \in \mathbb{N}$ (the* pumping length*) such that, if w is any string accepted by D of length at least p, then w may be written $w = xyz$, such that*

1. *for each $i \in \mathbb{N}$, D accepts $xy^i z$,*

2. *$|y| > 0$, and*

3. *$|xy| \leq p$.*

Proof. Let $D = (Q, \Sigma, \delta, s, F)$ be a DFA and let $p = |Q|$.

Let $w \in \Sigma^n$ be a string accepted by D, where $n \geq p$. Let $r_0, r_1, \ldots, r_n$ be the sequence of $n + 1$ states M enters while reading w, i.e., $r_i = \widehat{\delta}(s, w[0 \mathinner{..} i - 1])$. Note $r_0 = s$ and $r_n \in F$. Since $n \geq p = |Q|$, two states, say, r_j and r_l, with $j < l \leq p + 1$, must be equal by the pigeonhole principle. Let $x = w[0 \mathinner{..} j - 1]$, $y = w[j \mathinner{..} l - 1]$, and $z = w[l \mathinner{..} n - 1]$.

Since $\widehat{\delta}(r_j, y) = r_l = r_j$, it follows that for all $i \in \mathbb{N}$, $\widehat{\delta}(r_j, y^i) = r_j$. Therefore $\widehat{\delta}(r_0, xy^i z) = r_n = \in F$, whence M accepts $xy^i z$, satisfying (1). Since $j \neq l$, $|y| > 0$, satisfying (2). Finally, $l \leq p + 1$, so $|xy| \leq p$, satisfying (3). □

[29]

Theorem 1.4.1. *The language $B = \{\ 0^n 1^n \mid n \in \mathbb{N}\ \}$ is not regular.*

Proof. Let M be any DFA, with pumping length p. Let $w = 0^p 1^p$. If M rejects w, then we are done since $w \in B$, so assume M accepts w. Then $w = xyz$, where $|y| > 0$, $|xy| \leq p$, and M accepts $xy^i z$ for all $i \in \mathbb{N}$.

Since $|xy| \leq p$, $y \in \{0\}^*$. Since $|y| > 0$ and xyz has an equal number of 0's and 1's, it follows that $xyyz$ has more 0's than 1's, whence $xyyz \notin B$.

But M accepts $xyyz$, so M does not recognize B. □

Theorem 1.4.2. *The language $F = \{\ ww \mid w \in \{0, 1\}^*\ \}$ is not regular.*

[29] The strategy for employing the Pumping Lemma to show a language L is not regular is: fix a DFA M, find a long string in L, long enough that it can be pumped (relative to M), then prove that pumping it "moves it out of the language". This shows M does not recognize L, since M accepts the pumped string, by the Pumping Lemma.

Proof. Let M be any DFA, with pumping length p. Let $w = 0^p10^p1$. If M rejects w, then we are done since $w \in F$, so assume M accepts w. Then $w = xyz$, where $|y| > 0$, $|xy| \leq p$, and M accepts xy^iz for all $i \in \mathbb{N}$.

Since $|xy| \leq p$, $y \in \{0\}^m$ for some $m \in \mathbb{Z}^+$. Hence $xyyz = 0^{p+m}10^p1 \notin F$.

But M accepts $xyyz$, so M does not recognize F. □

Here is a nonregular unary language.

Theorem 1.4.3. *The language* $D = \left\{ \ 1^{n^2} \ \middle| \ n \in \mathbb{N} \ \right\}$ *is not regular.*

Proof. Let M be any DFA, with pumping length p. Let $w = 1^{p^2}$. If M rejects w, then we are done since $w \in D$, so assume M accepts w. Then $w = xyz$, where $|y| > 0$, $|xy| \leq p$, and M accepts xy^iz for all $i \in \mathbb{N}$.

Since $|y| \leq p$, $|xyyz| - |w| \leq p$. Let $u = 1^{(p+1)^2}$ be the next biggest string in D after w. Then

$$\begin{aligned} |u| - |w| &= (p+1)^2 - p^2 \\ &= p^2 + 2p + 1 - p^2 \\ &= 2p + 1 \\ &> p, \end{aligned}$$

whence $|xyyz|$ is strictly between $|w|$ and $|u|$, and hence it is not the length of *any* string in D. So $xyyz \notin D$.

But M accepts $xyyz$, so M does not recognize D. □

In the next example, we use the nonregularity of one language to prove that another language is not regular, without directly employing the Pumping Lemma.

Theorem 1.4.4. *The language* $A = \{ \ w \in \{0,1\}^* \mid \#(0,w) = \#(1,w) \ \}$ *is not regular.*

Proof. Let $B = A \cap \{0^*1^*\} = \{ \ 0^n1^n \mid n \in \mathbb{N} \ \}$. Since $\{0^*1^*\}$ is regular, and the regular languages are closed under intersection, if A were regular, then B would be regular also. But by Theorem 1.4.1, B is not regular. Hence A is not regular.[30] □

LECTURE: end of day 8

---

[30] This is not a proof by contradiction; it is a proof by contrapositive. Since $\{0^*1^*\}$ is regular, and the regular languages are closed under intersection, the implication statement "A is regular $\implies$ B is regular" is true. The contrapositive of this statement (hence equivalent to the statement) is "B is not regular $\implies$ A is not regular", and B is not regular by Theorem 1.4.1; hence, A is not regular. As a matter of style, you should not use a proof by contradiction that does not actually use the assumed falsity of the theorem; in the case of Theorem 1.4.4, we never once needed to assume that A is regular in our arguments, so it would have been useless to first claim, "Assume for the sake of contradiction that A is regular" at the start of the proof, though it is common to see such redundancy in proofs by contradiction.

Chapter 2

Context-Free Languages

Reading assignment: Chapter 2 of Sipser.

2.1 Context-Free Grammars

$$\begin{aligned} A &\rightarrow 0A1 \\ A &\rightarrow B \\ B &\rightarrow \# \end{aligned}$$

A grammar consists of *substitution rules* (*productions*), one on each line. The single symbol on the left is a *variable*, and the string on the right consists of variables and other symbols called *terminals.* One variable is designated as the *start variable*, on the left of the topmost production. The fact that the left side of each production has a single variable means the grammar is *context-free.* We abbreviate two rules with the same left-hand variable as follows: $A \rightarrow 0A1 \mid B$.

A and B are variables, and 0, 1, and # are terminals.

Definition 2.1.1. A *context-free grammar* (*CFG*) is a 4-tuple (V, Σ, R, S), where

- V is a finite alphabet of *variables*,
- Σ is a finite alphabet, disjoint from V, called the *terminals*,
- $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of *rules*, and
- $S \in V$ is the *start symbol.*

A sequence of productions applied to generate a string of all terminals is a *derivation.* For example, a derivation of 000#111 is

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111.$$

We also may represent this derivation with a *parse tree.* See Figure 2.1 in Sipser.

The set of all-terminal strings generated by a CFG G is the *language of the grammar* $L(G)$. Given the example CFG G above, its language is $L(G) = \{\ 0^n\#1^n \mid n \in \mathbb{N}\ \}$. A language generated by a CFG is a *context-free language* (*CFL*).

Below is a small portion of the CFG for the Java language:

$$\begin{aligned} \mathit{AssignmentOperator} &\to \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \\ \mathit{Expression1} &\to \mathit{Expression2} \mid \mathit{Expression2}\ ?\ \mathit{Expression}\ :\ \mathit{Expression1} \\ \mathit{Expression} &\to \mathit{Expression1} \mid \mathit{Expression1}\ \mathit{AssignmentOperator}\ \mathit{Expression1} \end{aligned}$$

2.2 Pushdown Automata

Finite automata cannot recognize the language $L = \{\ 0^n1^n \mid n \in \mathbb{N}\ \}$ because their memory is limited and cannot count 0's. A *pushdown automaton* is an NFA augmented with an infinite stack memory, which enables them to recognize languages such as L.

For an alphabet Σ, let $\Sigma_\lambda = \Sigma \cup \{\lambda\}$.

Definition 2.2.1. A *(nondeterministic) pushdown automaton* (*NPDA*) is a 6-tuple $(Q, \Sigma, \Gamma, \Delta, s, F)$, where

- Q is a finite set of *states*,
- Σ is the *input alphabet*,
- Γ is the *stack alphabet*,
- $\Delta : Q \times \Sigma_\lambda \times \Gamma_\lambda \to \mathcal{P}(Q \times \Gamma_\lambda)$ is the *transition function*,[1]
- $s \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accept states*.

Facts to know about CFLs and NPDAs:

1. A language is context-free if and only if it is recognized by some NPDA.
2. Deterministic PDAs (DPDAs) are strictly weaker than NPDAs, so some CFLs are not recognized by any DPDA.
3. Since a NPDA that does not use its stack is simply an NFA, every regular language is context-free. The converse is not true; for instance, $\{\ 0^n1^n \mid n \in \mathbb{N}\ \}$ is context-free but not regular.

[1] where the NPDA may or may not read an input symbol, may or may not pop the topmost stack symbol, and may or may not push a new stack symbol

Chapter 3

The Church-Turing Thesis

3.1 Turing Machines

Reading assignment: Section 3.1 in Sipser.

A Turing machine is a finite automaton with an unbounded read/write tape memory.

See Figure 3.1 in Sipser.

Differences between finite automata and Turing machines:

- The TM can write on its tape and read from it.
- The read-write tape head can move left or right.
- The tape is unbounded.
- There are special accept and reject states that cause the machine to immediately halt; conversely, the machine will not halt until it reaches one of these states, which may never happen.

Example 3.1.1. Design a Turing machine to test membership in the language

$$A = \{ \ w\#w \mid w \in \{0,1\}^* \ \} .$$

1. Zig-zag across the tape to corresponding positions on either side of the #, testing whether these positions contain the same input symbol. If not, or if there is no #, *reject.* Cross off symbols as they are checked so we know which symbols are left to check.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any remain, *reject*; otherwise, *accept.*

See Figure 3.2 in Sipser.

LECTURE: end of day 9

3.1.1 Formal Definition of a Turing machine

Definition 3.1.2. A *Turing machine (TM)* is a 7-tuple $(Q, \Sigma, \Gamma, \delta, s, q_\texttt{a}, q_\texttt{r})$, where

- Q is a finite set of *states*,
- Σ is the *input alphabet*, assumed not to contain the *blank symbol* $\sqcup$,
- Γ is the *tape alphabet*, where $\sqcup \in \Gamma$ and $\Sigma \subsetneq \Gamma$,
- $s \in Q$ the *start state*,
- $q_\texttt{a} \in Q$ the *accept state*,
- $q_\texttt{r} \in Q$ the *reject state*, where $q_\texttt{a} \neq q_\texttt{r}$, and
- $\delta : (Q \setminus \{q_\texttt{a}, q_\texttt{r}\}) \times \Gamma \to Q \times \Gamma \times \{\texttt{L}, \texttt{R}\}$ is the *transition function.*

Example 3.1.3. We formally describe the TM $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_\texttt{a}, q_\texttt{r})$ described earlier, which decides the language $B = \{\ w\#w \mid w \in \{0,1\}^*\ \}$:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_\texttt{a}, q_\texttt{r}\}$,
- $\Sigma = \{0, 1, \#\}$ and $\Gamma = \{0, 1, \#, \texttt{x}, \sqcup\}$,
- δ is shown in Figure 3.10 of the textbook.

Show TMSimulator with $w;w$ example TM.

3.1.2 Formal Definition of Computation by a Turing Machine

The TM starts with its input written at the leftmost positions on the tape, with $\sqcup$ written everywhere else.

A *configuration* of a TM $M = (Q, \Sigma, \Gamma, \delta, s, q_\texttt{a}, q_\texttt{r})$ is a triple (q, p, w), where

- $q \in Q$ is the *current state*,
- $p \in \mathbb{N}$ is the *tape head position*,
- $w \in \Gamma^*$ is the *tape contents*, the string consisting of the symbols starting at the leftmost position of the tape, until the rightmost non-blank symbol, or the largest position the tape head has scanned, whichever is larger.

Given two configurations C and C', we say C *yields* C', and we write $C \to C'$, if C' is the configuration that the TM will enter immediately after C. Formally, given $C = (q, p, w)$ and $C' = (q', p', w')$, $C \to C'$ if and only if $\delta(q, w[p]) = (q', w'[p], m)$, where

- $w[i] = w'[i]$ for all $i \in \{0, \ldots, |w| - 1\} \setminus \{p\}$
- if $m = \texttt{L}$, then

  - $p' = \max\{0, p-1\}$
  - $|w'| = |w|$

- if $m = \mathtt{R}$, then
  - $p' = p+1$
  - $|w'| = |w|$ if $p' < |w|$, otherwise $|w'| = |w|+1$ and $w'[p'] = \sqcup$.

A configuration (q, p, w) is *accepting* if $q = q_\mathtt{a}$, *rejecting* if $q = q_\mathtt{r}$, and *halting* if it is accepting or rejecting. M *accepts* input $x \in \Sigma^*$ if there is a finite sequence of configurations $C_1, C_2, \ldots, C_k$ such that

1. $C_1 = (s, 0, x)$ (initial/start configuration),
2. for all $i \in \{1, \ldots, k-1\}$, $C_i \to C_{i+1}$, and
3. C_k is accepting.

The *language recognized (accepted) by* M is $L(M) = \{\ x \in \Sigma^* \mid M \text{ accepts } x\ \}$.

Definition 3.1.4. A language is called *Turing-recognizable* (*Turing-acceptable*, *computably enumerable*, *recursively enumerable*, *c.e.*, or *r.e.*) if some TM recognizes it.

A language is *co-Turing-recognizable* (*co-c.e.*) if its complement is c.e.

On any input, a TM may accept, reject, or loop, meaning that it never enters a halting state. If a TM M halts on every input string, then we say it is a *decider*, and that it *decides* the language $L(M)$. We also say that M is *total*, meaning that the function $f : \Sigma^* \to \{\textsc{Accept}, \textsc{Reject}\}$ that it computes is total. f is a partial function if M does not halt on certain inputs, since for those inputs, f is undefined.

Definition 3.1.5. A language is called *Turing-decidable* (*recursive*), or simply *decidable*, if some TM decides it.

3.2 Variants of Turing Machines

Reading assignment: Section 3.2 in Sipser.

Why do we believe Turing machines are an accurate model of computation?

One reason is that the definition is *robust*: we can make all number of changes, including apparent enhancements, without actually adding any computation power to the model. We describe some of these, and show that they have the same capabilities as the Turing machines described in the previous section.[1]

[1] This does not mean that any change whatsoever to the Turing machine model will preserve its abilities; by replacing the tape with a stack, for instance we would reduce the power of the Turing machine to that of a pushdown automaton. Likewise, allowing the start configuration to contain an arbitrary infinite sequence of symbols already written on the tape (instead of $\sqcup$ everywhere but the start, as we defined) would add power to the model; by writing an encoding of an undecidable language, the machine would be able to decide that language. But this would not mean the machine had magic powers; it just means that we cheated by providing the machine (without the machine having to "work for it") an infinite amount of information before the computation even begins.

In a sense, the equivalence of these models is unsurprising to programmers. Programmers are well aware that all general-purpose programming languages can accomplish the same tasks, since an interpreter for a given language can be implemented in any of the other languages.

3.2.1 Multitape Turing Machines

A *multitape Turing machine* has more than one tape, say $k \geq 2$ tapes, each with its own head for reading and writing. Each tape besides the first one is called a *worktape*, each of which starts blank, and the first tape, the *input tape*, starts with the input string written on it in the same fashion as a single-tape TM.

Such a machine's transition function is therefore of the form

$$\delta : (Q \setminus \{q_{\mathtt{a}}, q_{\mathtt{r}}\}) \times \Gamma^k \to Q \times \Gamma^k \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}^k$$

The expression

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, \mathtt{L}, \mathtt{R}, \ldots, \mathtt{L})$$

means that, if the TM is in state q_i and heads 1 through k are reading symbols a_1 through a_k, the machine goes to state q_j, writes symbols b_1 through b_k, and directs each head to move left or right, or to stay put, as specified.

Theorem 3.2.1. *For every multitape TM M, there is a single-tape TM M' such that $L(M) = L(M')$. Furthermore, if M is total, then M' is total.*

See Figure 3.14 in the textbook.

Proof. On input $w = w_1 w_2 \ldots w_n$, M' begins by replacing the string $w_1 w_2 \ldots w_n$ with the string

$$\#\overset{\bullet}{w}_1 w_2 \ldots w_n \# \underbrace{\overset{\bullet}{\sqcup}\#\overset{\bullet}{\sqcup}\# \ldots \overset{\bullet}{\sqcup}\#}_{k-1}$$

on its tape. It represents the contents of each of the k tapes between adjacent # symbols, and the tape head position of each by the position of the character with a • above it.

Whenever a single transition of M occurs, M' must scan the entire tape, using its state to store all the symbols under each • (since there are only a constant k of them, this can be done in its finite state set). This information can be stored in the finite states of M' since there are a fixed number of tapes and therefore a fixed number of •'s. Then, M' will reset its tape to the start, and move over each of the representations of the k tapes of M, moving the • appropriately (either left, right, or stay), and writing a new symbol on the tape at the old location of the •, according M's transition function. Each time the • attempts to move to the right onto the right-side boundary # symbol, the entire single tape contents to the right of the • are shifted right to make room to represent the now-expanded worktape contents.

This allows M' to represent completely represent the configuration of M and to simulate the computation done by M. Note that M' halts if and only if M halts, implying that M' is total if M is total. □

From this point on, we will describe Turing machines in terms of algorithms in pseudocode, knowing that if we can write an algorithm in some programming language to recognize or decide a language, then we can write a Turing machine as well. We will only refer to low-level details of Turing machines when it is convenient to do so; for instance, when simulating an algorithm, it is easier to simulate a Turing machine than a Java program.

A friend of mine who took this class told me he was glad nobody programmed Turing machines anymore, because it's so much harder than a programming language. This is a misunderstanding of why we study Turing machines. No one ever programmed Turing machines; they are a model of computation whose simplicity makes them easy to handle mathematically (and whose definition is intended to model a mathematician sitting at a desk with paper and a pencil), though this same simplicity makes them difficult to program. We generally use Turing machines when we want to prove limitations on algorithms. When we want to design algorithms, there is no reason to use Turing machines instead of pseudocode or a regular programming language.

LECTURE: end of day 10

3.2.2 Nondeterministic Turing Machines

We may define a nondeterministic TM (NTM) in an analogous fashion to an NFA. The transition function for such a machine is of the form

$$\delta : (Q \setminus \{q_{\mathtt{a}}, q_{\mathtt{r}}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\})$$

A NTM accepts a string if any of its computation paths accepts, where a *computation path* is a sequence of configurations corresponding to one particular sequence of nondeterministic choices made during the course of the computation.

Theorem 3.2.2. *For every NTM N, there is a TM M such that $L(N) = L(M)$. Furthermore, if N is total, then M is total.*

Proof. N's possible computation paths form a directed graph, with an edge from configuration C to configuration C' if $C \to C'$.[2] Conduct a breadth-first search of this graph starting at the initial configuration, looking for an accepting configuration. If one is found, *accept.*

If N is total, then all computation paths eventually halt, which implies that this graph is finite. Hence, if no computation path is accepting, M will terminate its search when the entire graph has been explored, and *reject.* □

[2] Nodes with more than one out-neighbor represent nondeterministic choices, and nodes with no out-neighbors represent halting configurations.

3.2.3 Enumerators

An *enumerator* is a TM E with a "printer" attached. It takes no input, and runs forever. Every so often it prints a string.[3] The set of strings printed is the *language enumerated by* E.

Theorem 3.2.3. *A language L is c.e. if and only if it is enumerated by some enumerator.*

This is why such languages are called computably enumerable.

Proof. **($\Longleftarrow$):** Let E be an enumerator that enumerates L. Define $M(w)$ as follows: simulate E. Whenever E prints a string x, *accept* if $x = w$, and keep simulating E otherwise.

Note that if E prints w eventually, then M accepts w, and that otherwise M will not halt. Thus M accepts w if and only if w is in the language enumerated by E.

($\Longrightarrow$): Let M be a TM that recognizes L. We use a standard trick in computability theory known as a *dovetailing* computation. Let $s_1 = \lambda, s_2 = 0, s_3 = 1, s_4 = 00, \ldots$ be the standard lexicographical enumeration of $\{0,1\}^*$.

Define E as follows:

```
for $i = 1, 2, 3, \ldots$:
    for $j = 1, 2, 3, \ldots, i$:
        run $M(s_j)$ for $i$ steps; if it accepts in the first $i$ steps, print $s_j$.
```

Note that for every $i, j \in \mathbb{N}$, M will eventually be allowed to run for i steps on input s_j. Therefore, if $M(s_j)$ ever halts and accepts, E will detect this and print s_i. Furthermore, E prints only strings accepted by M, so E enumerates exactly the strings accepted by M. □

3.3 The Definition of Algorithm

Reading assignment: Section 3.3 in Sipser.

- 1928 - David Hilbert puts forth the Entscheidungsproblem, asking for an algorithm that will, given a mathematical theorem (stated in some formal language, with formal rules of deduction and a set of axioms, such as Peano arithmetic or ZFC), will indicate whether it is true or false

- 1931 - Kurt Gödel proves the incompleteness theorem: for logical systems such as Peano arithmetic or ZFC, there are theorems which are true but cannot be proven in the system.[4]

[3] We could imagine implementing these semantics by having a special "print" worktape. Whenever the TM needs to print a string, it writes the string on the tape, with a $\sqcup$ immediately to the right of the string, and places the tape head on the first symbol of the string (so that one could print λ by placing the tape head on a $\sqcup$ symbol), and enters a special state q_{print}. The set of strings printed is then the set of all strings that were between the tape head and the first $\sqcup$ to the right of the tape head on the printing tape whenever the TM was in the state q_{print}.

[4] Essentially, any sufficiently powerful logical system can express the statement, "This statement is unprovable.", which is either true, hence exhibiting a statement whose truth cannot be proven in the system, or false, meaning the false theorem can be proved, and the system is contradictory.

This leaves open the possibility of an algorithm that decides whether the statement is true or false, even though the correctness of the algorithm cannot be proven.[5]

- At this point in history, it remains the case that no one in the world knows exactly what they mean by the word “algorithm”, or “computable function”.
- 1936 - Alonzo Church proposes λ-calculus (the basis of modern functional languages such as LISP and Haskell) as a candidate for the class of computable functions. He shows that it can compute a large variety of known computable functions, but his arguments are questionable and researchers are not convinced.[6]
- 1936 - Alan Turing, as a first-year graduate student at the University of Cambridge in England, hears of the Entscheidungsproblem taking a graduate class. He submits a paper, “On computable numbers, with an application to the Entscheidungsproblem”, to the London Mathematical Society, describing the Turing machine (he called them a-machines) as a model of computation that captures all the computable functions and formally defines what an algorithm is. He also shows that as a consequence of various undecidability results concerning Turing machines, there is no solution to the Entscheidungsproblem; no algorithm can indicate whether a given theorem is true or false, in sufficiently powerful logical systems.
- Before the paper is accepted, Church’s paper reaches Turing from across the Atlantic. Before final publication, Turing adds an appendix proving that Turing machines compute exactly the same class of functions as the λ-calculus.
- Turing’s paper is accepted, and researchers in the field – Church included – were immediately convinced by Turing’s arguments.

The Church-Turing Thesis. All functions that can be computed in a finite amount of time by a physical object in the universe, can be computed by a Turing machine.

[5] Lest that would provide a proof the truth or falsehood of the statement.

[6] For instance, Emil Post accused Church of attempting to “mask this identification [of computable functions] under a definition.” (Emil L. Post, Finite combinatory processes, Formulation I, The Journal of Symbolic Logic, vol. 1 (1936), pp. 103–105, reprinted in [27], pp. 289–303.)

Chapter 4

Decidability

4.1 Decidable Languages

Reading assignment: Section 4.1 in Sipser.

A decidable language is a language decided by some Turing machine (or equivalently, by some C++ program). Sipser gives extensive examples of decidable languages, all of which are related to finite automata or pushdown automata in some way. We won't do this, for two reasons:

1. It gives the impression that even now that we are studying computability, the entire field is just an elaborate ruse for asking more sophisticated questions about finite-state and pushdown automata. This is not at all the case; almost everything interesting that we know about computability has nothing to do with finite automata or pushdown automata.

2. You have extensive experience with decidable languages, because every time that you wrote a C++ function with a `bool` return type, that algorithm was recognizing some c.e. language: specifically, the set of all input arguments that cause the function to return `true`. If it had no infinite loops, then it was deciding that language.

Because years of programming has already developed in you an intuition for what constitutes a decidable language, we will not spend much time on this. It is sufficient that the last chapter convinced you that Turing machines have the same fundamental computational capabilities as C++ programs, and that therefore all the intuition you have developed in programming, algorithms, and other classes in which you *wrote* algorithms, applies to Turing machines as well.

We now concentrate on the primary utility of Turing machines: using their apparent simplicity to prove *limitations* on the fundamental capabilities of algorithms.

It is common to write algorithms in terms of the data structures they are operating on, even though these data structures must be encoded in binary before delivering them to a computer (or in some alphabet Σ before delivering them to a TM). Given any finite object O, such as a string, graph, tuple, or even a Turing machine, we use the notation $\langle O \rangle$ to denote the encoding of O as a string in the input alphabet of the Turing machine we are using. To encode multiple objects, we use the notation $\langle O_1, O_2, \ldots, O_k \rangle$ to denote the encoding of the objects O_1 through O_k as a single string.

4.2 The Halting Problem

Reading assignment: Section 4.2 in Sipser.

Define the *Halting Problem* (or *Halting Language*)

$$A_{\mathsf{TM}} = \{ \ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \ \} .$$

A_{TM} is sometimes denoted K or $0'$.[1]

Note that A_{TM} is c.e., via the following Turing machine U:

$U =$ "On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Simulate M on input w.

2. If M ever enters its accept state, *accept*. If M ever enters its reject state, *reject*."

U is called the *universal Turing machine*, since it simulates any other Turing machine.[2]

U simulates M exactly, meaning that if M does not halt, neither does M. We will show that though A_{TM} is c.e., A_{TM} is not decidable. The proof technique, surprisingly, is 120 years old.

LECTURE: end of day 11

4.2.1 Diagonalization

Cantor considered the question: given two infinite sets, how can we decide which one is "bigger"? We might say that if $A \subseteq B$, then $|B| \geq |A|$.[3] But this is a weak notion, as the converse does not hold even for finite sets: $\{1, 2\}$ is smaller than $\{3, 4, 5\}$, but $\{1, 2\} \not\subseteq \{3, 4, 5\}$.

Cantor noted that two finite sets have equal size if and only if there is a bijection between them. Extending this slightly, one can say that a finite set A is strictly smaller than B if and only if there is no onto function $f : A \to B$.

Draw picture of 1-1 and onto

For instance, try to find an onto function $f : \{1, 2\} \to \{3, 4, 5\}$: there are only two values of f, $f(1)$ and $f(2)$, but there are three values 3,4,5 that must be mapped to, so at least one of 3, 4, or

[1] Note that to be precise, A_{TM} does not tell us whether $M(w)$ halts; it tells us whether M accepts w. But if we knew whether $M(w)$ halted, then we could determine whether it accepts by running it until it halts, and then reporting the answer. So deciding A_{TM} boils down to deciding whether a given TM on a given input halts, hence the name.

[2] It is obvious to us that such a machine exists, since we have programmed enough to realize that any program is capable of simulating any other, and so we don't need convincing to see that a TM can be written that is capable of simulating any other TM. This is the TM equivalent of, for instance, writing an interpreter for Python in Python. But at the time the universal Turing machine was introduced in Turing's original paper, it was not at all obvious that such a universal simulation algorithm existed, and Turing devoted many pages of his paper to constructing the universal Turing machine in low-level detail.

[3] For instance, we think of the integers as being at least as numerous as the even integers.

5 will be left out (will not be an output of f), so f will not be onto. We conclude the obvious fact that $|\{1,2\}| < |\{3,4,5\}|$.

Since the notion of onto functions is just as well defined for infinite sets as for finite sets, this gives a reasonable notion of how to compare the cardinality of infinite sets.

The following theorem, proved by Cantor in 1891, changed the course of science. It shows that the power set of any set is *strictly* larger than the set itself.

Theorem 4.2.1. *Let X be any set. Then there is no onto function $f : X \to \mathcal{P}(X)$.*

Proof. Let X be any set, and let $f : X \to \mathcal{P}(X)$. It suffices to show that f is not onto.

Define the set

$$D = \{\ a \in X \mid a \not\in f(a)\ \}.$$

Let $a \in X$ be arbitrary. Since $D \in \mathcal{P}(X)$, it suffices to show that $D \neq f(a)$. By the definition of D,

$$a \in D \iff a \not\in f(a),$$

so $D \neq f(a)$. □

Draw picture of metaphorical "diagonal".

The interpretation is that $|X| < |\mathcal{P}(X)|$, even if X is infinite.

For two sets X, Y, we write $|X| < |Y|$ if there is no onto function $f : X \to Y$. We write $|X| \geq |Y|$ if it is not the case that $|X| < |Y|$; i.e., if there *is* an onto function $f : X \to Y$. We write $|X| = |Y|$ if there is a bijection (a 1-1 and onto function) $f : X \to Y$.[4]

We say X is *countable* if $|X| \leq |\mathbb{N}|$;[5] i.e., if X is a finite set or if $|X| = |\mathbb{N}|$.[6] We say X is *uncountable* if it is not countable; i.e., if $|X| > |\mathbb{N}|$.[7]

Stating that a set X is countable is equivalent to saying that its elements can be listed; i.e., that it can be written $X = \{x_0, x_1, x_2, \ldots\}$, where every element of X will appear somewhere in the list.[8]

Observation 4.2.2. $|\mathbb{N}| < |\mathbb{R}|$*; i.e.,* $\mathbb{R}$ *is uncountable.*

Proof. By Theorem 4.2.1, $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})|$, so it suffices to prove that $|\mathcal{P}(\mathbb{N})| \leq |\mathbb{R}|$;[9] i.e., that there is an onto function $f : \mathbb{R} \to \mathcal{P}(\mathbb{N})$.

[4] By a result known as the *Cantor-Bernstein Theorem*, this is equivalent to saying that there is an onto function $f : X \to Y$ and another onto function $g : Y \to X$; i.e., $|X| = |Y|$ if and only if $|X| \leq |Y|$ and $|X| \geq |Y|$.

[5] Some textbooks define countable only for infinite sets, but here we consider finite sets to be countable, so that uncountable will actually be the negation of countable.

[6] It is not difficult to show that for every infinite countable set has the same cardinality as $\mathbb{N}$; i.e., there is no infinite countable set X with $|X| < |\mathbb{N}|$.

[7] The relations $<$, $>$, $\leq$, $\geq$, and $=$ are transitive: for instance, $(|A| \leq |B| \text{ and } |B| \leq |C|) \implies |A| \leq |C|$.

[8] This is because the order in which we list the elements implicitly gives us the bijection between $\mathbb{N}$ and X: $f(0) = x_0, f(1) = x_1$, etc.

[9] Actually they are equal, but we need not show this for the present observation.

Define $f : \mathbb{R} \to \mathcal{P}(\mathbb{N})$ as follows. Each real number $r \in \mathbb{R}$ has an infinite decimal expansion.[10] For all $n \in \mathbb{N}$, let $r_n \in \{0, 1, \ldots, 9\}$ be the n^{th} digit of the decimal expansion of r. Define $f(r) \subseteq \mathbb{N}$ as follows. For all $n \in \mathbb{N}$,

$$n \in f(r) \iff r_n = 0.$$

That is, if the n^{th} digit of r's binary expansion is 0, then n is in the set $f(r)$, and n is not in the set otherwise. Given any set $A \subseteq \mathbb{N}$, there is some some number $r_A \in \mathbb{R}$ whose decimal expansion has 0's exactly at the positions $n \in A$, so $f(r_A) = A$, whence f is onto. □

Continuum Hypothesis. There is no set A such that $|\mathbb{N}| < |A| < |\mathcal{P}(\mathbb{N})|$.

More concretely, this is stating that for every set A, either there is an onto function $f : \mathbb{N} \to A$, or there is an onto function $g : A \to \mathcal{P}(\mathbb{N})$.

Interesting fact: Remember earlier when we stated that Gödel proved that there are true statements that are not provable? The Continuum Hypothesis is a concrete example of a statement that, if it is true, is not provable, nor is its negation. So it will forever remain a hypothesis; we can never hope to prove it either true or false.

Theorem 4.2.1 has immediate consequences for the theory of computing.

Observation 4.2.3. *There is an undecidable language $L \subseteq \{0,1\}^*$.*

Proof. The set of all TM's is countable, as is $\{0,1\}^*$. By Theorem 4.2.1, $\mathcal{P}(\{0,1\}^*)$, the set of all binary languages, is uncountable. Therefore there is a language that is not decided by any TM. □

4.2.2 The Halting Problem is Undecidable

Observation 4.2.3 shows that some undecidable language must exist. However, it would be more satisfying to exhibit a particular undecidable language. In the next theorem, we use the technique of diagonalization directly to show that

$$A_{\mathsf{TM}} = \{ \ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \ \}$$

is undecidable.

Theorem 4.2.4. *A_{TM} is undecidable.*

Proof. Assume for the sake of contradiction that A_{TM} is decidable, via the TM A. Then A accepts $\langle M, w \rangle$ if M accepts w, and A rejects $\langle M, w \rangle$ otherwise.

Define the TM D as follows. On input $\langle M \rangle$ a TM, D runs $A(\langle M, \langle M \rangle \rangle)$,[11] and does the opposite.[12]

[10] This expansion need not be unique, as expansions such 0.03000000... and 0.02999999... both represent the number $\frac{3}{100}$. But whenever this happens, exactly one representation will end in an infinite sequence of 0's, so take this as the "standard" decimal representation of r.

[11] That is, D runs A to determine if M accepts the string that is the binary description of M itself.

[12] In other words, accept if A rejects, and reject if A accepts. Since A is a decider (by our assumption), it will do one of these.

Now consider running D with itself as input. Then

$$\begin{aligned} D \text{ accepts } \langle D\rangle &\iff A \text{ rejects } \langle D, \langle D\rangle\rangle && \text{defn of } D \\ &\iff D \text{ does not accept } \langle D\rangle, && \text{defn of } A \end{aligned}$$

a contradiction. Therefore no such TM A exists. □

It is worth examining the proof of Theorem 4.2.4 to see the diagonalization explicitly. See Figures 4.19, 4.20, and 4.21 in Sipser.

4.2.3 A Non-c.e. Language

Theorem 4.2.5. *A language is decidable if and only if it is c.e. and co-c.e..*

Proof. We prove each direction separately.

(decidable $\implies$ c.e. and co-c.e.): Any decidable language is c.e., and the complement of any decidable language is decidable, hence also c.e.

(c.e. and co-c.e. $\implies$ decidable): Let L be c.e. and co-c.e., let M_1 be a TM recognizing L, and let M_2 be a TM recognizing $\overline{L}$. Define the TM M as follows. On input w, M runs $M_1(w)$ and $M_2(w)$ in parallel. One of them will accept since either $w \in L$ or $w \in \overline{L}$. If M_1 accepts first, then M accepts w, and if M_2 accepts w first, then $M(w)$ rejects. Hence M decides L. □

Corollary 4.2.6. $\overline{A_{\mathsf{TM}}}$ *is not c.e.*

Proof. A_{TM} is c.e. If $\overline{A_{\mathsf{TM}}}$ were c.e., then A_{TM} would be co-c.e. by definition, as well as c.e. via U, and hence decidable by Theorem 4.2.5, contradicting Theorem 4.2.4. □

Note the key fact used in the proof of Corollary 4.2.6 is that the class of c.e. languages is *not* closed under complement. Hence *any* language that is c.e. but not decidable (such as A_{TM}) has a complement that is not c.e..

Chapter 5

Reducibility

Now that we know the halting problem is undecidable, we can use that as a shoehorn to prove other languages are undecidable, without having to repeat the full diagonalization.

5.1 Undecidable Problems from Language Theory (Computability)

Reading assignment: Section 5.1 in Sipser.

Recall

$$A_{\mathsf{TM}} = \{ \ \langle M, w \rangle \mid M \text{ is a TM and accepts } w \ \} .$$

Define

$$HALT_{\mathsf{TM}} = \{ \ \langle M, w \rangle \mid M \text{ is a TM and halts on input } w \ \} .$$

Theorem 5.1.1. *$HALT_{\mathsf{TM}}$ is undecidable.*

Proof. Suppose for the sake of contradiction that $HALT_{\mathsf{TM}}$ is decidable by TM H. Define the TM T as follows on input $\langle M, w \rangle$:

```
1. Run $H(\langle M, w \rangle)$
2. If $H$ rejects $\langle M, w \rangle$, reject.
3. If $H$ accepts $\langle M, w \rangle$:
4.     run $M(w)$
5.     if $M$ accepts $w$, accept.
6.     if $M$ rejects $w$, reject.
```

If H accepts in line (3), M is guaranteed to halt in line (4), so T always halts. But T decides A_{TM}, a contradiction since A_{TM} is undecidable. Hence H does not exist and $HALT_{\mathsf{TM}}$ is undecidable. □

LECTURE: end of day 12

Define

$$E_{\mathsf{TM}} = \{ \ \langle M \rangle \ | \ M \text{ is a TM and } L(M) = \emptyset \ \} .$$

Theorem 5.1.2. *E_{TM} is undecidable.*

Proof. For any TM M and string w, define the TM $N_{M,w}$ as follows:
$N_{M,w}$ on input x:

1. If $x \neq w$, `reject`.
2. If $x = w$, run $M(w)$ and `accept` if M accepts w.

Given M and w, either $L(N_{M,w}) = \{w\} \neq \emptyset$ if M accepts w, and $L(N_{M,w}) = \emptyset$ otherwise.

Suppose for the sake of contradiction that E_{TM} is decidable by TM E. Define the TM A as follows on input $\langle M, w \rangle$:

1. Run $E(\langle N_{M,w} \rangle)$
2. If E accepts $\langle N_{M,w} \rangle$, `reject`
3. If E rejects $\langle N_{M,w} \rangle$, `accept`

Therefore, for TM M and string w,

$$\begin{aligned} M \text{ accepts } w &\iff L(N_{M,w}) \neq \emptyset && \text{defn of } N_{M,w} \\ &\iff E \text{ rejects } N_{M,w} && \text{since } E \text{ decides } E_{\mathsf{TM}} \\ &\iff A \text{ accepts } \langle M, w \rangle . && \text{lines (2) and (3) of A} \end{aligned}$$

Since A always halts, it decides A_{TM}, a contradiction. □

Define the *diagonal halting problem* $K_0 = \{ \ \langle M \rangle \ | \ M \text{ is a TM and } M(\langle M \rangle) \text{ halts } \}$.

Theorem 5.1.3. *K_0 is undecidable.*

First, suppose we knew already that K_0 was undecidable, but not whether $HALT_{\mathsf{TM}}$ was undecidable. Then we could easily use the undecidability of K_0 to prove that $HALT_{\mathsf{TM}}$ is undecidable, using the same pattern we have used already. That is, suppose for the sake of contradiction that $HALT_{\mathsf{TM}}$ was decidable by TM H. Then to decide whether $M(\langle M \rangle)$ halts, we use the decider H to determine whether $\langle M, \langle M \rangle \rangle \in HALT_{\mathsf{TM}}$. That is, an instance of K_0 is a simple special case of an instance of $HALT_{\mathsf{TM}}$,[1] so $HALT_{\mathsf{TM}}$ is at least as difficult to decide as K_0. Proving that K_0 is undecidable means showing that $HALT_{\mathsf{TM}}$ is not actually any *more* difficult to decide than K_0.

[1] Just as, for instance, determining whether two nodes are connected in a graph is a special case of finding the length of the shortest path between them (the length being finite if and only if there is a path), which involves determining whether there is a path as a special case of the more general task of finding the optimal length of the path.

Proof. Suppose for the sake of contradiction that K_0 is decidable by TM K. Define the TM H as follows on input $\langle M, w\rangle$:

1. Define the TM $N_{M,w}$ based on M and w as follows:
   (a) On *every* input x,
   (b) Run $M(w)$ and halt if $M(w)$ does.
2. Run $K(\langle N_{M,w}\rangle)$ and echo its answer.

Given M and w, we define $N_{M,w}$ so that, if $M(w)$ halts, then $N_{M,w}$ halts on all inputs (including the input $\langle N_{M,w}\rangle$), and if $M(w)$ does not halt, then $N_{M,w}$ halts on no inputs (including the input $\langle N_{M,w}\rangle$). Then H decides $HALT_{\mathsf{TM}}$, a contradiction. □

The following theorem generalizes many (but not all) undecidability proofs.

Rice's Theorem. *Given any property $\phi : \mathcal{P}(\{0,1\}^*) \to \{\text{N}, \text{Y}\}$ shared by some, but not all, c.e. languages, the language $\{ \langle M\rangle \mid \phi(L(M)) = \text{Y} \}$ is undecidable.*

For example, if the property ϕ is "$L(M)$ is empty," then Rice's Theorem shows that E_{TM} is undecidable. If ϕ is "$|L(M)| \leq 5$", then Rice's Theorem shows that the language of problem 4 on HW7 is undecidable.

(Draw diagram of c.e. contained in set of all languages, and ϕ cutting both of them.)

Note that Rice's theorem does not state that all properties of *Turing machines* are undecidable (e.g., *"does the machine have 5 states?"* is plenty easy to decide); the theorem only applies to properties of the *language recognized* by the TM.[2]

However, unless explicitly asked to do so, you may not use Rice's Theorem to prove that a language is undecidable.[3]

skipping linear-bounded automata

skipping section 5.2 of Sipser

[2] So it must hold that if one TM has the property, all TM's recognizing the same language must also have the property. A property such as "has 5 states" fails this test, since any machine with 5 states recognizes a language that is also recognized by some machine with more than 5 states.

[3] You may, of course, unravel the proof of Rice's Theorem in the special case of the problem you are solving, to get a proof of the undecidability of the language you are studying. But it is usually easier to do a direct proof.

Chapter 6

Advanced Topics in Computability

Chapter 7

Time Complexity

Computability focuses on which problems are computationally solvable in principle. Computational complexity focuses on which problems are solvable in practice.

7.1 Measuring Complexity

Reading assignment: Section 7.1 in Sipser.

Definition 7.1.1. Let M be a TM, and let $x \in \{0,1\}^*$. Define $\text{time}_M(x)$ to be the number of steps M takes before halting on input x (the number of configurations it visits before halting, so that even a TM that immediately accepts takes 1 step rather than 0), with $\text{time}_M(x) = \infty$ if M does not halt on input x. If M is total, define the *(worst-case) running time* (or *time complexity*) of M to be the function $t : \mathbb{N} \to \mathbb{N}$ defined for all $n \in \mathbb{N}$ by

$$t(n) = \max_{x \in \{0,1\}^n} \text{time}_M(x).$$

We call such a function t a *time bound*.

Definition 7.1.2. Given $f, g : \mathbb{N} \to \mathbb{R}^+$, we write $f = O(g)$ (or $f(n) = O(g(n))$), if there exists $c \in \mathbb{N}$ such that, for all $n \in \mathbb{N}$,

$$f(n) \leq c \cdot g(n).$$

We say g is an *asymptotic upper bound* for f.

Bounds of the form n^c for some constant c are called *polynomial bounds*. Bounds of the form 2^{n^δ} for some real constant $\delta > 0$ are called *exponential bounds*.

Definition 7.1.3. Given $f, g : \mathbb{N} \to \mathbb{R}^+$, we write $f = o(g)$ (or $f(n) = o(g(n))$), if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

One way to see that $f(n) = o(g(n))$ is to find $h(n)$ so that $f(n) \cdot h(n) = g(n)$ for some *unbounded* $h(n)$.

$f = O(g)$ is like saying f "$\leq$" g.

$f = o(g)$ is like saying f "$<$" g.

Based on this analogy, we can write

- $f = \Omega(g)$ if and only if $g = O(f)$ (like saying f "$\geq$" g),
- $f = \omega(g)$ if and only if $g = o(f)$ (like saying f "$>$" g), and
- $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$. (like saying f "$=$" g)

Example 7.1.4. The following are easy to check.

1. $\sqrt{n} = o(n)$.
2. $n = o(n \log \log n)$.
3. $n \log \log n = o(n \log n)$.
4. $n \log n = o(n^2)$.
5. $n^2 = o(n^3)$.
6. For every $f : \mathbb{N} \to \mathbb{R}^+$, $f = O(f)$, but $f \neq o(f)$.

7.1.1 Analyzing Algorithms

Definition 7.1.5. Let $t : \mathbb{N} \to \mathbb{N}$ be a time bound.[1] Define $\mathsf{DTIME}(t) \subseteq \mathcal{P}(\{0,1\}^*)$ to be the set of decision problems

$$\mathsf{DTIME}(t) = \left\{\ L \subseteq \{0,1\}^* \ \middle|\ \text{there is a TM with running time } O(t) \text{ that decides } L\ \right\}.$$

Observation 7.1.6. *For all time bounds $t, T : \mathbb{N} \to \mathbb{N}$ such that $t = O(T)$,*

$$\mathsf{DTIME}(t) \subseteq \mathsf{DTIME}(T).$$

[1] We will prove many time complexity results that actually do not hold for all functions $t : \mathbb{N} \to \mathbb{N}$. Many results hold only for time bounds that are what is known as *time-constructible*, which means, briefly, that $t : \mathbb{N} \to \mathbb{N}$ has the property that the related function $f_t : \{0\}^* \to \{0\}^*$ defined by $f_t(0^n) = 0^{t(n)}$ is computable in time $t(n)$. This is equivalent to requiring that there is a TM that, on input 0^n, halts in exactly $t(n)$ steps. The reason is to require that for every time bound used, there is a TM that can be used as a "clock" to time the number of steps that another TM is using, and to halt that TM if it exceeds the time bound. If the time bound itself is very difficult to compute, then this cannot be done.

All "natural" time bounds we will study, such as n, $n \log n$, n^2, 2^n, etc., are time-constructible. Time-constructibility is an advanced (and boring) issue that we will not dwell on, but it is worth noting that it is possible to define unnatural time bounds relative to which unnatural theorems can be proven, such as a time bound t such that $\mathsf{DTIME}(t(n)) = \mathsf{DTIME}\left(2^{2^{2^{t(n)}}}\right)$. This is an example of what I like to call a "false theorem": it is true, of course, but its truth tells us that our model of reality is incorrect and should be adjusted. Non-time-constructible time bounds do not model anything found in reality.

The following result is the basis for all complexity theory. We will not prove it yet (and perhaps not at all this semester), but it is important because it tells us that complexity classes are worth studying; we aren't just wasting time making up different names for the same class of languages. Informally, it shows that given more time, one can decide more languages.

Theorem 7.1.7 (Deterministic Time Hierarchy Theorem)**.** *Let $t, T : \mathbb{N} \to \mathbb{N}$ be time bounds such that $t(n) \log t(n) = o(T(n))$. Then*

$$\mathsf{DTIME}(t) \subsetneq \mathsf{DTIME}(T).$$

For instance, there is a language decidable in time n^2 that is not decidable in time n, and another language decidable in time n^3 that is not decidable in time n^2, etc.

Sipser discusses three algorithms for deciding the language $\{\ 0^n1^n \mid n \in \mathbb{N}\ \}$, taking time $O(n^2)$ on a one-tape TM, then a better one taking time $O(n \log n)$ on a one-tape TM, then one taking time $O(n)$ on a two-tape TM, on pages 251-253. Read that section, and after proving the next theorem, we will analyze the time complexity of algorithms at a high level (rather than at the level of individual transitions of a TM), assuming, as in 228, that individual lines of code in Java or pseudocode take constant time, so long as those lines are not masking loops or recursion or something that would not take constant time.

Note that since a TM guaranteed to halt in time $t(n)$ is a decider, in complexity theory we are back to the situation we had with finite automata, and unlike computability theory: we can equivalently talk about the language recognized by a TM and the language decided by a TM.

7.1.2 Complexity Relationships Among Models

Theorem 7.1.8. *Let $t : \mathbb{N} \to \mathbb{N}$, where $t(n) \geq n$. Then every $t(n)$ time multitape TM can be simulated in time $O(t(n)^2)$ by a single-tape TM.*

Proof. Recall that a single-tape TM S can simulate a k-tape TM M with tape contents such as (for $k = 3$) $\#01\dot{0}0\#\dot{1}1\sqcup\#00\dot{\sqcup}10\#$. M's tape heads move right by at most $t(n)$ positions, so S's tape contents have length at most $\underbrace{k}_{\text{number of tapes}} \cdot \underbrace{t(n)}_{\text{max size of each of } M\text{'s tape contents}} + \underbrace{k+1}_{\text{number of \#'s}} = O(t(n))$.
Simulating one step of M requires moving S's tape by this length and back, so requires $O(t(n))$ time. Since M takes $t(n)$ steps, S takes $O(t(n)^2)$ steps. □

Definition 7.1.9. Let N be a total NTM. The *running time* of N is the function $t : \mathbb{N} \to \mathbb{N}$, where $t(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.

(See Figure 7.10 in Sipser.)

Theorem 7.1.10. *Let $t : \mathbb{N} \to \mathbb{N}$, where $t(n) \geq n$. Then for every $t(n)$ time NTM N, there is a constant c such that N can be simulated in time $2^{ct(n)}$ by a deterministic TM.*

LECTURE: end of day 13

Proof. We examine the construction in the proof of Theorem 3.2.2, and show that it incurs an exponential overhead.

Let M be the deterministic TM simulating the NTM N, let $x \in \{0,1\}^*$, and let T_x be the BFS tree created when searching the configuration graph of N on input x. On any input length n, every branch of T_x has length at most $t(n)$. There is a constant b, depending on N's transition function but independent of n, such that each node in T_x has at most b children. Every tree with at most b children and depth at most t has at most $b^t = (2^{\log b})^t = 2^{t \log b}$ total nodes. Letting c be sufficiently large, the time required to search this tree is at most $2^{ct(n)}$. □

That is, NTM's can be simulated by deterministic TM's with at most an exponential blowup in the running time.[2] Much of the focus of this chapter is in providing evidence that this exponential blowup is fundamental, and not an artifact of the above proof that can be avoided through a clever trick.

7.2 The Class P

Reading assignment: Section 7.2 in Sipser.

7.2.1 Polynomial Time

- We consider polynomial running times to be "small", and exponential running times to be "large".
- For $n = 1000$, $n^3 = 1$ billion, whereas $2^n >$ number of atoms in the universe
- Usually, exponential time algorithms are encountered when a problem is solved by *brute force search* (e.g., searching all paths in a graph, looking for a Hamiltonian cycle). Polynomial time algorithms are due to someone finding a *shortcut* that allows the solution to be found without searching the whole space.
- All "reasonable" models of computation are *polynomially equivalent*: M_1 (e.g., a TM) can simulate a $t(n)$-time M_2 (e.g., a Java program) in time $p(t(n))$, for some polynomial p (usually not much bigger than $O(n^2)$ or $O(n^3)$).

[2] Compare this to the exponential blowup in states of the subset construction; the reasons are similar, in that n binary choices leads to 2^n possibilities.

- In this course, we generally ignore polynomial differences in running time. Polynomial differences are important, but we simply focus our lens farther out, to see where in complexity theory the *really* big differences lie. In ECS 122A, for instance, a difference of a $\log n$ or $\sqrt{n}$ factor is considered more significant.

- In ignoring polynomial differences, we can make conclusions that apply to any model of computation, since they are polynomially equivalent, rather than having to choose one such model, such as TM's or Java, and stick with it. Our goal is to understand *computation* in general, rather than an individual programming language.

- One objection is that some polynomial running times are not feasible, for instance, n^{1000}. In practice, there are few algorithms with such running times. Nearly every algorithm known to have a polynomial running time has a running time less than n^{10}. Also, when the first polynomial-time algorithm for a problem is discovered, such as the $O(n^{12})$-time algorithm for PRIMES discovered in 2002,[3] it is usually brought down within a few years to a smaller degree, once the initial insight that gets it down to polynomial inspires further research. PRIMES currently is known to have a $O(n^6)$-time algorithm, and this will likely be improved in the future.

Definition 7.2.1.

$$\mathsf{P} = \bigcup_{k=1}^{\infty} \mathsf{DTIME}(n^k).$$

In other words, P is the class of language decidable in polynomial time on a deterministic, one-tape TM.

1. Although $\mathsf{DTIME}(t)$ is different for different models of computation, P is the same class of languages, in any model of computation polynomially equivalent to single-tape TM's (which is all of them worth studying, except possibly for quantum computers, whose status is unknown).

2. P roughly corresponds to the *problems feasibly solvable by a deterministic algorithm.*[4]

7.2.2 Examples of Problems in P

We use pseudocode to describe algorithms, knowing that the running time will be polynomially close to the running time on a single-tape TM.

We must be careful to use reasonable encodings with the encoding function $\langle\cdot\rangle : D \to \{0,1\}^*$, that maps elements of a discrete set D (such as $\mathbb{N}$, Σ^*, or the set of all Java programs) to binary strings. For instance, for $n \in \mathbb{N}$, two possibilities are $\langle n\rangle_1 = 0^n$ (the unary expansion of n) and $\langle n\rangle_2 =$ the binary expansion of n. $|\langle n\rangle_1| \geq 2^{|\langle n\rangle_2|}$, so $\langle n\rangle_1$ is a bad choice. Even doing simple

[3] Here, n is the *size of the input*, i.e., the number of bits needed to represent an integer p to be tested for primality, which is $\approx \log p$.

[4] Here, "deterministic" is intended both to emphasize that P does not take into account nondeterminism, which is an unrealistic model of computation, but also that it does not take into account randomized algorithms, which *is* a realistic model of computation. BPP, then class of languages decidable by polynomial-time randomized algorithms, is actually conjectured to be equal to P, though this has not been proven.

arithmetic would take exponential time in the length of the binary expansion, if we choose the wrong encoding. Alternately, exponential-time algorithms might appear mistakenly to be polynomial time, since the running time is a function of the input size, and exponentially expanding the input lowers the running time artificially, even though the algorithm still takes the same (very large) number of steps. Hence, an analysis showing the very slow algorithm to be technically polynomial would be misinforming.

As another example, a reasonable encoding of a directed graph $G = (V, E)$ with $V = \{0, 1, \ldots, n-1\}$, is via its adjacency matrix, where for $i, j \in \{1, \ldots, n\}$, the $(n \cdot i + j)^{\text{th}}$ bit of $\langle G \rangle$ is 1 if and only if $(i, j) \in E$. For an algorithms course, we would care about the difference between this encoding and an adjacency list, since sparse graphs (those with $|E| \ll |V|^2$) are more efficiently encoded by an adjacency list than an adjacency matrix. But since these two representations differ by at most a linear factor, we ignore the difference.

Define

$$\textsc{Path} = \{ \ \langle G, s, t \rangle \ \mid \ G \text{ is a directed graph with a path from node } s \text{ to node } t \ \}.$$

Theorem 7.2.2. $\textsc{Path} \in \mathsf{P}$.

Proof. Breadth-first search. □

Note in particular that there are an exponential number of simple paths from s to t in the worst case ($(n-2)!$ paths in the complete directed graph with n vertices), but we do not examine them all in a breadth-first search. The BFS takes a shortcut to zero in on one particular path in polynomial time.

Given $x, y \in \mathbb{Z}^+$, we say x and y are *relatively prime* if the largest integer dividing both of them is 1. Define

$$\textsc{RelPrime} = \{ \ \langle x, y \rangle \ \mid \ x \text{ and } y \text{ are relatively prime } \}.$$

Theorem 7.2.3. $\textsc{RelPrime} \in \mathsf{P}$.

Proof. Since the input size is $|\langle x, y \rangle| = O(\log x + \log y)$, we must be careful to use an algorithm that is polynomial in $|\langle x, y \rangle|$, *not* polynomial in x and y themselves, which would be exponential in the input size.

Euclid's algorithm for finding the greatest common divisor of two integers works.

$\text{gcd}((\langle x, y \rangle)) =$

```
1. Repeat until $y = 0$
2.     $x \leftarrow x \mod y$
3.     swap $x$ and $y$
4. output $x$
```

Then R decides $\textsc{RelPrime}$:

$R(\langle x, y \rangle) =$

```
1. If $\text{gcd}(\langle x, y \rangle) = 1$, then accept.
```

2. Otherwise, *reject.*

Each iteration of the loop of gcd cuts the value of x in half, for the following reason. If $y \leq \frac{x}{2}$, then $x \bmod y < y \leq \frac{x}{2}$. If $y > \frac{x}{2}$, then $x \bmod y = x - y < \frac{x}{2}$. Therefore, at most $\log x < |\langle x, y\rangle|$ iterations of the loop execute, and each iteration requires a constant number of polynomial-time computable arithmetic operations, whence R is operates in polynomial time. □

Note that there are an exponential (in $|\langle x, y\rangle|$) number of integers that could potentially be divisors of x and y (namely, all the integers less than $\min\{x, y\}$), but Euclid's algorithm does not check all of them to see if they divide x or y; it uses a shortcut.

7.3 The Class NP

Reading assignment: Section 7.3 in Sipser.

Some problems have polynomial-time deciders. Other problems are not known to have polynomial-time deciders, but given a candidate solution (an informal notion that we will make formal later) to the problem, whether the solution works can be *verified* in polynomial time.

A *Hamiltonian path* in a directed graph G is a path that visits each node exactly once. Define

$$\textsc{HamPath} = \{\ \langle G\rangle\ \mid G \text{ is a directed graph with a Hamiltonian path }\}.$$

HamPath is not known to have a polynomial-time algorithm (and it is generally believed not to have one), but, a related problem, that of *checking* whether a given path is a Hamiltonian path in a given graph, does have a polynomial-time algorithm:

$$\textsc{HamPath}_V = \{\ \langle G, \pi\rangle\ \mid G \text{ is a directed graph with the Hamiltonian path } \pi\ \}.$$

The algorithm simply verifies that each adjacent pair of nodes in π is connected by an edge in G (so that π is a valid path in G), and that each node of G appears exactly once in π (so that π is Hamiltonian).

Another problem with a related polynomial-time verification language is

$$\textsc{Composites} = \left\{\ \langle n\rangle\ \middle|\ n \in \mathbb{Z}^+ \text{ and } n = pq \text{ for some integers } p, q > 1\ \right\},$$

with the related verification language

$$\textsc{Composites}_V = \left\{\ \langle n, p\rangle\ \middle|\ n, p \in \mathbb{Z}^+, p \text{ divides } n, \text{ and } p > 1\ \right\}.$$

Actually, Composites $\in$ P, so sometimes a problem can be decided *and* verified in polynomial time.

Not all problems solvable by exponential brute force necessarily can be verified in polynomial time. For instance, $\overline{\textsc{HamPath}}$ is believed not to be verifiable in polynomial time; it is difficult to imagine what "proof" one could give that a graph has no Hamiltonian path, which could be verified in polynomial time.

We now formalize these notions.

Definition 7.3.1. A *polynomial-time verifier* for a language A is an algorithm V, where there are polynomials p, q such that V runs in time p and

$$A = \left\{ \ x \in \{0,1\}^* \ \middle| \ \left(\exists w \in \{0,1\}^{\leq q(|x|)}\right) \ V \text{ accepts } \langle x, w \rangle \ \right\}.$$

That is, $x \in A$ if and only if there is a "short" string w where $\langle x, w \rangle \in L(V)$ (where "short" means bounded by a polynomial in $|x|$). We call such a string w a *witness* (or a *proof* or *certificate*) that testifies that $x \in A$.

Example 7.3.2. For a graph G with a Hamiltonian path π, π is a witness testifying that G has a Hamiltonian path.

Example 7.3.3. For a composite integer n with a divisor $1 < p < n$, $\langle p \rangle$ is a witness testifying that n is composite. Note that n may have more than one such divisor; this shows that a witness for an element of a polynomially-verifiable language need not be unique.

Definition 7.3.4. NP is the class of languages that have polynomial-time verifiers. We call the language decided by the verifier the *verification language* of the NP language.

For instance, $\textsc{HamPath}_V$ is the verification language of $\textsc{HamPath}$.

NP stands for *nondeterministic polynomial time*, **not** for "non-polynomial time". In fact, $\mathsf{P} \subseteq \mathsf{NP}$, so some problems in NP are solvable in polynomial time. The name comes from an alternate (and in my opinion, less intuitive) characterization of NP in terms of nondeterministic polynomial-time machines.

The following NTM decides $\textsc{HamPath}$ in polynomial time.

N_1 = "On input $\langle G \rangle$, where $G = (V, E)$ is a directed graph,

1. Nondeterministically choose a list of $n = |V|$ vertices $v_1, \ldots, v_n \in V$.
2. If the list contains any repetitions, *reject*.
3. For each $1 \leq i < n$, if $(v_i, v_{i+1}) \notin E$, *reject*.
4. If $\langle G \rangle$ has not been rejected yet, *accept*."

Note what N_1 does: it nondeterministically guesses a witness $(v_1, \ldots, v_n)$ and then runs the verification algorithm with $\langle G \rangle$ and the witness as input.

Theorem 7.3.5. *A language is in* NP *if and only if it is decided by some polynomial-time NTM.*

Proof. skipping □

Definition 7.3.6. $\mathsf{NTIME}(t) = \{ \ L \subseteq \{0,1\}^* \ \mid L \text{ is decided by a } O(t)\text{-time NTM } \}$.

Corollary 7.3.7. $\mathsf{NP} = \bigcup_{k=1}^{\infty} \mathsf{NTIME}(n^k)$.

We will use the verification characterization of NP almost exclusively in this course.[5]

[5] Most proofs showing that a language is in NP that claim to use the NTM characterization actually use an algorithm

7.3.1 Examples of Problems in NP

A *clique* in a graph is a subgraph in which every pair of nodes in the clique is connected by an edge. A k-clique is a clique with k nodes. Define

$$\textsc{Clique} = \{\ \langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\ \}.$$

Theorem 7.3.8. $\textsc{Clique} \in \mathsf{NP}$.

Proof. The following is a polynomial-time verifier for Clique:
On input $\langle\langle G, k\rangle, C\rangle$[6]

1. Test whether C is a set of k nodes from G.
2. Test whether every pair of nodes in C is connected by an edge in G.
3. *accept* iff both tests pass. □

LECTURE: end of day 14

---

midterm

LECTURE: end of day 15

The SubsetSum problem concerns integer arithmetic: given a collection (a multiset) of integers $x_1, \ldots, x_k$ and a target integer t, is there a subcollection that sums to t?

$$\textsc{SubsetSum} = \left\{ \langle S, t\rangle \;\middle|\; \begin{array}{l} S = \{x_1, \ldots, x_k\} \text{ is a collection of} \\ \text{integers and } (\exists S' \subseteq S)\ t = \sum_{y \in S'} y \end{array} \right\}.$$

For example, $\langle\{4, 4, 11, 16, 21, 27\}, 29\rangle \in \textsc{SubsetSum}$ because $4+4+21 = 29$, but $\langle\{4, 11, 16\}, 13\rangle \notin \textsc{SubsetSum}$.

similar to that for HamPath: on input x, nondeterministically guess a witness w, then run the verification algorithm on x and w. In other words, they implicitly use the verification characterization by structuring the algorithm as in the proof of Theorem 7.3.5; the only point of the nondeterminism is to choose a witness at the start of the algorithm, which is then provided to the verification algorithm.

I consider the verification characterization to be cleaner, as it separates these two conceptually different ideas – producing a witness that a string is in the language, and then verifying that it is actually a "truthful" witness – into two separate steps, rather than mingling the production of the witness with the verification algorithm into a single algorithm with nondeterministic choices attempting to guess bits of the witness while simultaneously verifying it. Verifiers also make more explicit what physical reality is being modeled by the class NP. That is, there is no such thing as a nondeterministic polynomial-time machine that can be physically implemented. Verification algorithms, however, can be implemented, without having to introduce a new "magic" ability to guess nondeterministic choices; the model simply states that the witness is provided as an input, without specifying the source of this input.

[6] We pair G and k together to emphasize that they are the input to the original decision problem, and that the witness C is then attached to that input to get an input to the verification problem.

Theorem 7.3.9. $\textsc{SubsetSum} \in \mathsf{NP}$.

Proof. The following is a polynomial-time verifier for $\textsc{SubsetSum}$:

On input $\langle \langle S, t \rangle, S' \rangle$

1. Test whether S' is a collection of integers that sum to t.
2. Test whether S contains all the integers in S'. (i.e., whether $S' \subseteq S$)
3. *accept* iff both tests pass. □

Note that the complements of these languages, $\overline{\textsc{Clique}}$ and $\overline{\textsc{SubsetSum}}$, are not obviously members of NP (and are believed not to be). The class coNP is the class of languages whose complements are in NP, so $\overline{\textsc{Clique}}, \overline{\textsc{SubsetSum}} \in \mathsf{coNP}$. It is not known whether $\mathsf{coNP} = \mathsf{NP}$, but this is believed to be false, although proving that is at least as difficult as proving that $\mathsf{P} \neq \mathsf{NP}$: since $\mathsf{P} = \mathsf{coP}$, if $\mathsf{coNP} \neq \mathsf{NP}$, then $\mathsf{P} \neq \mathsf{NP}$.

7.3.2 The P Versus NP Question

In summary:

- P is the class of languages for which membership can be *decided* quickly.
- NP is the class of languages for which membership can be *verified* quickly.

(Figure 7.26 in Sipser.)

skip rest of this section

The best known method for solving NP problems in general is a brute-force search over all possible witnesses, giving each as input to the verifier. Since we require that each witness for a string of length n has length at most $p(n)$ for some polynomial p, there are at most $2^{p(n)}$ potential witnesses, which implies that every NP problem *can* be solved in *exponential* time by searching through all possible witnesses; that is,

$$\mathsf{NP} \subseteq \mathsf{EXP} = \bigcup_{k=1}^{\infty} \mathsf{DTIME}\left(2^{n^k}\right),$$

This idea is essentially the same as the proof of Theorem 7.1.10.

No one has proven that $\mathsf{NP} \neq \mathsf{EXP}$. It is known that $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP}$, and since the Time Hierarchy Theorem tells us that $\mathsf{P} \subsetneq \mathsf{EXP}$, it is known that at least one of the inclusions $\mathsf{P} \subseteq \mathsf{NP}$ or $\mathsf{NP} \subseteq \mathsf{EXP}$ is proper, though it is not known which one. It is suspected that they both are; i.e., that $\mathsf{P} \subsetneq \mathsf{NP} \subsetneq \mathsf{EXP}$.

We distinguish EXP from *linear-exponential* time:

$$\mathsf{E} = \bigcup_{k=1}^{\infty} \mathsf{DTIME}\left(2^{kn}\right).$$

Note that $\mathsf{E} \subsetneq \mathsf{EXP}$ by the Time Hierarchy Theorem. It is not known whether $\mathsf{NP} \subseteq \mathsf{E}$ or whether $\mathsf{E} \subseteq \mathsf{NP}$, although oddly enough, it is known that $\mathsf{NP} \neq \mathsf{E}$. Probably, they are incomparable (meaning that neither contains the other).

7.4 NP-Completeness

Reading assignment: Section 7.4 in Sipser.

We now come to the only reason that any computer scientist is concerned with the class NP: the NP-complete problems. (More justification of that claim in Section 7.7.1)

Intuitively, a problem is NP-complete if it is in NP, and every problem in NP is reducible to it in polynomial time. This implies that the NP-complete problems are, "within a polynomial error, the hardest problems" in NP. If any NP-complete problem has a polynomial-time algorithm, then all problems in NP do, including all the other NP-complete problems. The contrapositive is more interesting because it is stated in terms of two claims we think are true, rather than two things we think are false:[7] if $\mathsf{P} \neq \mathsf{NP}$, then *no* NP-complete problem is in P.

This gives us a tool by which to prove that a problem is "probably" (so long as $\mathsf{P} \neq \mathsf{NP}$) intractable: show that it is NP-complete. Just as with undecidability, it takes some big insights to prove that one problem is NP-complete, but once this is done, we can use the machinery of reductions to show that other languages are NP-complete, by showing how they can be used to solve a problem already known to be NP-complete. Unlike with undecidability, in which we established that A_{TM} is undecidable before doing anything else, we will first define some problems and show reductions between them, before finally proving that one of them is NP-complete.

The first problem concerns Boolean formulas. We represent True with 1, False with 0, And with $\wedge$, Or with $\vee$, and Not with $\neg$ or with an overbar:

$$\begin{array}{lll} 0 \vee 0 = 0 & 0 \wedge 0 = 0 & \overline{0} = 1 \\ 0 \vee 1 = 1 & 0 \wedge 1 = 0 & \overline{1} = 0 \\ 1 \vee 0 = 1 & 1 \wedge 0 = 0 & \\ 1 \vee 1 = 1 & 1 \wedge 1 = 1 & \end{array}$$

A *Boolean formula* is an expression involving Boolean variables and the three operations $\wedge$, $\vee$, and $\neg$. For example,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

is a Boolean formula. A Boolean formula is *satisfiable* if some assignment w of 0's and 1's to its variables causes the formula to have the value 1. ϕ is satisfiable because assigning $x = 0, y = 1, z = 0$ causes ϕ to evaluate to 1, written $\phi(010) = 1$. We say the assignment *satisfies* ϕ. The *satisfiability problem* is to test whether a given Boolean formula is satisfiable:

$$\textsc{Sat} = \{ \ \langle \phi \rangle \ \mid \phi \text{ is a satisfiable Boolean formula } \}.$$

Note that $\textsc{Sat} \in \mathsf{NP}$, since the language

$$\textsc{Sat}_V = \{ \ \langle \phi, w \rangle \mid \ \phi \text{ is a Boolean formula and } \phi(w) = 1 \ \}$$

is a polynomial-time verification language for Sat; i.e., if ϕ has n input variables, then $\langle \phi \rangle \in \textsc{Sat} \iff (\exists w \in \{0,1\}^n) \ \langle \phi, w \rangle \in \textsc{Sat}_V$.

Theorem 7.4.1 (Cook-Levin Theorem)**.** Sat $\in$ P *if and only if* $\mathsf{P} = \mathsf{NP}$.

This is considered evidence (though not proof) that $\textsc{Sat} \notin \mathsf{P}$.

[7] Statements such as "(Some NP-complete problem is in P) $\implies$ P = NP" are often called, "If pigs could whistle, then donkeys could fly"-theorems.

7.4.1 Polynomial-Time Reducibility

Definition 7.4.2. A function $f : \{0,1\}^* \to \{0,1\}^*$ is *polynomial-time computable* if there is a polynomial-time TM that, on input x, halts with $f(x)$ on its tape, and $\sqcup$'s everywhere else.

Definition 7.4.3. Let $A, B \subseteq \{0,1\}^*$. We say A is *polynomial-time mapping reducible* to B (*polynomial-time many-one reducible*, *polynomial-time reducible*), written $A \leq^{\mathrm{P}}_{\mathrm{m}} B$, if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that, for all $x \in \{0,1\}^*$,

$$x \in A \iff f(x) \in B.$$

f is called a *(polynomial-time) reduction* of A to B.

That is, $\leq^{\mathrm{P}}_{\mathrm{m}}$ is simply the polynomial-time analog of $\leq_{\mathrm{m}}$. We interpret $A \leq^{\mathrm{P}}_{\mathrm{m}} B$ to mean that "B is at least as hard as A, to within a polynomial-time error."

Note that for any mapping reduction f computable in time $p(n)$, and all $x \in \{0,1\}^*$, $|f(x)| \leq p(|x|)$.

Theorem 7.4.4. *If $A \leq^{\mathrm{P}}_{\mathrm{m}} B$ and $B \in \mathsf{P}$, then $A \in \mathsf{P}$.*

Proof. Let M be the algorithm deciding B in time $p(n)$ for some polynomial p, and let f be the reduction from A to B, computable in time $q(n)$ for some polynomial q. Define the algorithm

$N =$ "On input x,

1. Compute $f(x)$.
2. Run M on input $f(x)$ and echo its output."

N correctly decides A because $x \in A \iff f(x) \in B$. Furthermore, on input x, N runs in time $q(|x|) + p(|f(x)|) \leq q(|x|) + p(q(|x|))$. N is then polynomial-time by the closure of polynomials under composition.[8] □

Corollary 7.4.5. *If $A \leq^{\mathrm{P}}_{\mathrm{m}} B$ and $A \notin \mathsf{P}$, then $B \notin \mathsf{P}$.*

Theorem 7.4.4 tells us that if the fastest algorithm for B takes time $p(n)$, then the fastest algorithm for A takes no more than time $q(n) + p(q(n))$, where q is the running time of f; i.e., q is the "polynomial error" when claiming that "A is no harder than B within a polynomial error". Since we ignore polynomial differences when defining P, we conclude that $A \notin \mathsf{P} \implies B \notin \mathsf{P}$.

We now use $\leq^{\mathrm{P}}_{\mathrm{m}}$-reductions to show that Clique is "at least as hard" (modulo polynomial differences in running times) as a useful version of the Sat problem known as 3Sat.

A *literal* is a Boolean variable or negated Boolean variable, such as x or $\overline{x}$. A *clause* is several literals connected with $\vee$'s, such as $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. A *conjunction* is several subformulas connected with $\wedge$'s, such as $(x_1 \wedge (\overline{x_2} \vee x_7) \wedge \overline{x_3} \wedge x_4)$. A Boolean formula ϕ is in *conjunctive normal form*, called a *CNF-formula*, if it consists of a conjunction of clauses,[9] such as

$$\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

[8] i.e., because $r(n) \equiv p(q(n))$ is a polynomial whenever p and q are polynomials.

[9] The obvious dual of CNF is *disjunctive normal form (DNF)*, which is an Or of conjunctions, such as the formula one would derive applying the sum-of-products rule to the truth table of a boolean function, but 3DNF formulas do not have the same nice properties that 3CNF formulas have, so we do not discuss them further.

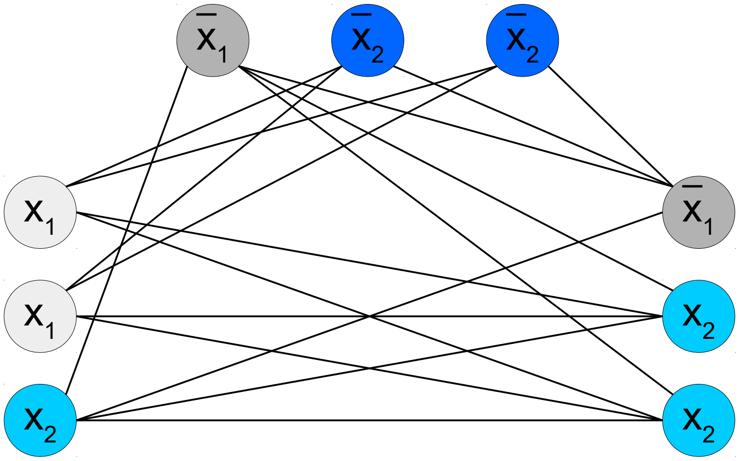

Figure 7.1: Example of the $\leq^{\mathrm{P}}_{\mathrm{m}}$-reduction from 3Sat to Clique when the input is the 3CNF formula $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

ϕ is a *3CNF-formula* if all of its clauses have exactly three literals, such as

$$\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Note that any CNF formula with *at most* 3 literals per clause can be converted easily to 3CNF by duplicating literals; for example, $(x_3 \vee \overline{x_5})$ is equivalent to $(x_3 \vee \overline{x_5} \vee \overline{x_5})$.

Define

$$\text{3Sat} = \{\ \langle \phi \rangle \ \mid \phi \text{ is a satisfiable 3CNF-formula }\}.$$

We will later show that 3Sat is NP-complete, but for now we will simply show that it is reducible to Clique.

Theorem 7.4.6. *3Sat $\leq^{\mathrm{P}}_{\mathrm{m}}$ Clique.*

Proof. Given a 3CNF formula ϕ, we convert it in polynomial time (via reduction f) into a pair $f(\langle \phi \rangle) = \langle G, k \rangle$, a graph G and integer k so that

$$\phi \text{ is satisfiable} \iff G \text{ has a } k\text{-clique}.$$

Let $k = \#$ of clauses in ϕ. Write ϕ as

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k).$$

G will have $3k$ nodes, one for each literal appearing in ϕ.[10]

[10] Note that ϕ could have many more (or fewer) literals than variables, so G may have many more nodes than ϕ has variables. But G will have exactly as many nodes as ϕ has *literals*. Note also that literals can appear more than once, e.g., $(x_1 \vee x_1 \vee x_2)$ has two copies of the literal x_1.

See Figure 7.1 for an example of the reduction.

The nodes of G are organized into k groups of three nodes each, called *triples*. Each triple corresponds to a clause in ϕ.

G has edges between every pair of nodes u and v, unless:

(1) u and v are in the same triple, or

(2) $u = x_i$ and $v = \overline{x_i}$, or vice-versa.

Each of these conditions can be checked in polynomial time, and there are $\binom{|V|}{2} = O(|V|^2)$ such conditions, so this is computable in polynomial time.

LECTURE: end of day 16

We now show that

$$\phi \text{ is satisfiable} \iff G \text{ has a } k\text{-clique}.$$

($\implies$): Suppose that ϕ has a satisfying assignment. Then at least one literal is true in every clause. To construct a k-clique C, select exactly one node from each clause labeled by a true literal (breaking ties arbitrarily); this implies condition (1) is false for every pair of nodes in C. Since x_i and $\overline{x_i}$ cannot both be true, condition (2) is false for every pair of nodes in C. Therefore every pair of nodes in C is connected by an edge; i.e., C is a k-clique.

($\impliedby$): Suppose there is a k-clique C in G. No two of the clique's nodes are in the same triple by condition (1), so each triple contains exactly one node of C. If $x_i \in C$, assign $x_i = 1$ to make the corresponding clause true. If $\overline{x_i} \in C$, assign $x_i = 0$ to make the corresponding clause true. Since no two nodes x_i and $\overline{x_i}$ are part of C by condition (2), this assignment is well-defined (we will not attempt to assign x_i to be both 0 and 1). The assignment makes every clause true, thus satisfies ϕ. □

Theorems 7.4.4 and 7.4.6 tell us that if Clique is decidable in polynomial time, then so is 3Sat. Speaking in terms of what we believe is actually true, Corollary 7.4.5 and Theorem 7.4.6 tell us that if 3Sat is *not* decidable in polynomial time, then neither is Clique.

7.4.2 Definition of NP-Completeness

Definition 7.4.7. A language B is NP-*hard* if, for every $A \in \mathsf{NP}$, $A \leq^{\mathrm{P}}_{\mathrm{m}} B$.

Definition 7.4.8. A language B is NP-*complete* if

1. $B \in \mathsf{NP}$, and
2. B is NP-hard.

Theorem 7.4.9. *If B is* NP*-complete and $B \in \mathsf{P}$, then* $\mathsf{P} = \mathsf{NP}$.

Proof. Assume the hypothesis and let $A \in \mathsf{NP}$. Since B is NP-hard, $A \leq^{\mathrm{P}}_{\mathrm{m}} B$. Since $B \in \mathsf{P}$, by Theorem 7.4.4 (the closure of P under $\leq^{\mathrm{P}}_{\mathrm{m}}$-reductions), $A \in \mathsf{P}$, whence $\mathsf{NP} \subseteq \mathsf{P}$. Since $\mathsf{P} \subseteq \mathsf{NP}$, $\mathsf{P} = \mathsf{NP}$. □

Corollary 7.4.10. *If* $\mathsf{P} \neq \mathsf{NP}$*, then no* NP*-complete problem is in* P*.*

[11]

Theorem 7.4.11. *If* B *is* NP*-hard and* $B \leq^{\mathrm{P}}_{\mathrm{m}} C$*, then* C *is* NP*-hard.*

Proof. Let $A \in \mathsf{NP}$. Then $A \leq^{\mathrm{P}}_{\mathrm{m}} B$ since B is NP-hard. Since $\leq^{\mathrm{P}}_{\mathrm{m}}$ is transitive and $B \leq^{\mathrm{P}}_{\mathrm{m}} C$, it follows that $A \leq^{\mathrm{P}}_{\mathrm{m}} C$. Since $A \in \mathsf{NP}$ was arbitrary, C is NP-hard. □

Corollary 7.4.12. *If* B *is* NP*-complete,* $C \in \mathsf{NP}$*, and* $B \leq^{\mathrm{P}}_{\mathrm{m}} C$*, then* C *is* NP*-complete.*

That is, NP-complete problems can, in many ways, act as "representatives" of the hardness of NP, in the sense that black-box access to an algorithm for solving an NP-complete problem is as good as access to an algorithm for any other problem in NP.

Corollary 7.4.12 is our primary tool for proving a problem is NP-complete: show that some existing NP-complete problem reduces to it. This should remind you of our tool for proving that a language is undecidable: show that some undecidable language reduces to it. We will eventually show that 3SAT is NP-complete; from this, it will immediately follow that CLIQUE is NP-complete by Corollary 7.4.12 and Theorem 7.4.6.

7.4.3 The Cook-Levin Theorem

Theorem 7.4.13. SAT *is* NP*-complete.*[12]

We delay proving this theorem until we have covered some other NP-complete problems.

For the next section, we will assume that the 3SAT problem is also NP-complete.

7.5 Additional NP-Complete Problems

Reading assignment: Section 7.5 in Sipser.

To construct a polynomial-time reduction from 3SAT to another language, we transform the variables and clauses in 3SAT into structures in the other language. These structures are called *gadgets*. For example, to reduce 3SAT to CLIQUE, nodes "simulate" variables and triples of nodes "simulate" clauses. 3SAT is not the only NP-complete language that can be used to show other problems are NP-complete, but its regularity and structure make it convenient for this purpose.

Theorem 7.5.1. CLIQUE *is* NP*-complete.*

Proof. We have already shown that CLIQUE $\in \mathsf{NP}$. 3SAT $\leq^{\mathrm{P}}_{\mathrm{m}}$ CLIQUE by Theorem 7.4.6, and 3SAT is NP-complete, so CLIQUE is NP-hard. □

[11] Since it is generally believed that $\mathsf{P} \neq \mathsf{NP}$, Corollary 7.4.10 implies that showing a problem is NP-complete is evidence of its intractability.

[12] By Theorem 7.4.9, Theorem 7.4.13 implies Theorem 7.4.1.

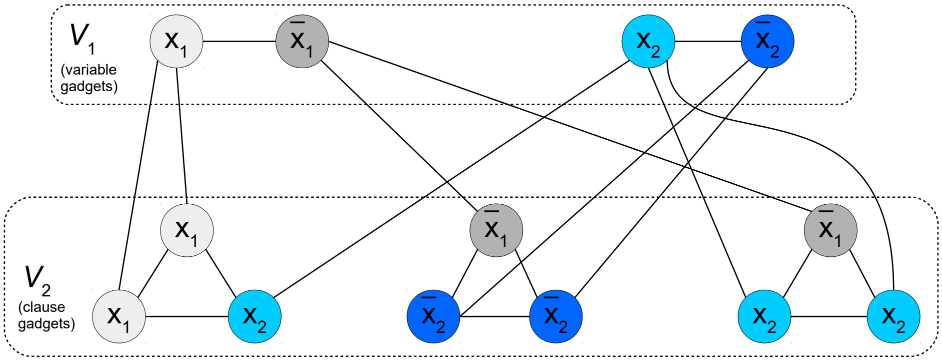

Figure 7.2: An example of the reduction from 3SAT to VERTEXCOVER for the 3CNF formula $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

7.5.1 The Vertex Cover Problem

If G is an undirected graph, a *vertex cover* of G is a subset of nodes, where each edge in G is connected to at least one node in the vertex cover. Define

$$\text{VERTEXCOVER} = \{\ \langle G, k\rangle \mid G \text{ is an undirected graph with a vertex cover of size } k\ \}.$$

Note that adding nodes to a vertex cover cannot remove its ability to touch every edge; hence if $G = (V, E)$ has a vertex cover of size k, then it has a vertex cover of each size k' where $k \leq k' \leq |V|$. Therefore it does not matter whether we say "of size k" or "of size at most k" in the definition of VERTEXCOVER.

Theorem 7.5.2. VERTEXCOVER *is* NP*-complete.*

Proof. We must show that VERTEXCOVER is in NP, and that some NP-complete problem reduces to it.

(in NP): The language

$$\text{VERTEXCOVER}_V = \left\{ \langle\langle G, k\rangle, C\rangle \;\middle|\; \begin{array}{l} G \text{ is an undirected graph and} \\ C \text{ is a vertex cover for } G \text{ of size } k \end{array} \right\}$$

is a verification language for VERTEXCOVER, and it is in P: if $G = (V, E)$, the verifier tests whether $C \subseteq V$, $|C| = k$, and for each $\{u, v\} \in E$, either $u \in C$ or $v \in C$.

(NP-hard): Given a 3CNF-formula ϕ, we show how to (efficiently) transform ϕ into a pair $\langle G, k\rangle$, where $G = (V, E)$ is an undirected graph and $k \in \mathbb{N}$, such that ϕ is satisfiable if and only if G has a vertex cover of size k.

See Figure 7.2 for an example of the reduction.

For each variable x_i in ϕ, add two nodes labeled x_i and $\overline{x_i}$ to V, and connect them by an edge; call this set of gadget nodes V_1. For each literal in each clause, we add a node labeled with the literal's value; call this set of gadget nodes V_2.

Connect nodes u and v by an edge if they are

1. in the same variable gadget,
2. in the same clause gadget, or
3. have the same label.

If ϕ has m input variables and l clauses, then V has $|V_1|+|V_2| = 2m+3l$ nodes. Let $k = m+2l$. Since $|V| = 2m + 3l$ and $|E| \leq |V|^2$, G can be computed in $O(|\langle\phi\rangle|^2)$ time.

LECTURE: end of day 17

Now we show that ϕ is satisfiable $\iff$ G has a vertex cover of size k:

($\implies$): Suppose $(\exists x_1 x_2 \ldots x_m \in \{0,1\}^m)$ $\phi(x_1 x_2 \ldots x_n) = 1$. If $x_i = 1$ (resp. 0), put the node in V_1 labeled x_i (resp. $\overline{x_i}$) in the vertex cover C; then every variable gadget edge is covered. In each clause gadget where this literal appears, if it appears in more than one node, pick one node arbitrarily and put the *other two* nodes of the gadget in C; then all clause gadget edges are covered.[13] All edges between variable and clause gadgets are covered, by the variable node if it was included in C, and by a clause node otherwise, since some other literal satisfies the clause. Since $|C| = m+2l = k$, G has a vertex cover of size k.

($\impliedby$): Suppose G has a vertex cover C with k nodes. Then C contains at least one of each variable node to cover the variable gadget edges, and at least two clause nodes to cover the clause gadget edges. This is $\geq k$ nodes, so C must have exactly one node per variable gadget and exactly two nodes per clause gadget to have exactly k nodes. Let $x_i = 1 \iff$ the variable node labeled $x_i \in C$. Each node in a clause gadget has an external edge to a variable node; since only two nodes of the clause gadget are in C, the external edges of the third clause gadget node is covered by a node from a variable gadget, whence the assignment satisfies the corresponding clause. □

7.5.2 The Subset Sum Problem

Theorem 7.5.3. SUBSETSUM *is* NP*-complete.*

Proof. Theorem 7.3.9 tells us that SUBSETSUM $\in$ NP. We show SUBSETSUM is NP-hard by reduction from 3SAT.

Let ϕ be a 3CNF formula with variables $x_1, \ldots, x_m$ and clauses $c_1, \ldots, c_l$. Construct the pair $\langle S, t\rangle$, where $S = \{y_1, z_1, \ldots, y_m, z_m, g_1, h_1, \ldots, g_l, h_l\}$ is a collection of $2(m+l)$ integers and t is an integer, whose *decimal expansions* are based on ϕ as shown by example:

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_3} \vee x_4)$$

[13] Any two clause gadget nodes will cover all three clause gadget edges.

	x_1	x_2	x_3	x_4	c_1	c_2	c_3
y_1	1	0	0	0	1	0	1
z_1	1	0	0	0	0	1	0
y_2		1	0	0	0	1	0
z_2		1	0	0	1	0	0
y_3			1	0	1	1	0
z_3			1	0	0	0	1
y_4				1	0	0	1
z_4				1	0	0	0
g_1					1	0	0
h_1					1	0	0
g_2						1	0
h_2						1	0
g_3							1
h_3							1
t	1	1	1	1	3	3	3

The upper-left and bottom-right of the table contain exactly one 1 per row as shown, the bottom-left is all empty (leading 0's), and t is m 1's followed by l 3's. The upper-right of the table has 1's to indicate which literals (y_i for x_i and z_i for $\overline{x_i}$) are in which clause. Thus each column in the upper-right has exactly three 1's.

The table has size $O((m+l)^2)$, so the reduction can be computed in time $O(n^2)$, since $m, l \leq n$.

We now show that

$$\phi \text{ is satisfiable} \iff (\exists S' \subseteq S)\ t = \sum_{n \in S'} n$$

($\Longrightarrow$): Suppose $(\exists x_1 x_2 \ldots x_m \in \{0,1\}^m)\ \phi(x_1 x_2 \ldots x_m) = 1$. Select elements $S' \subseteq S$ as follows. For each $1 \leq i \leq m$, $y_i \in S'$ if $x_i = 1$, and $z_i \in S'$ if $x_i = 0$. Since every clause c_j of ϕ is satisfied, for each column c_j, at least one row with a 1 in column c_j is selected in the upper-right. For each $1 \leq j \leq l$, if needed g_j and/or h_j are placed in S' to make column c_j on the right side of the table sum to 3.[14] Then S' sums to t.

($\Longleftarrow$): Suppose $(\exists S' \subseteq S)\ t = \sum_{n \in S'} n$. All the digits in elements of S are either 0 or 1, and each column in the table contains at most five 1's; hence there are no carries when summing S'. To get a 1 in the first m columns, S' must contain exactly one of each pair y_i and z_i. The assignment $x_1 x_2 \ldots x_m \in \{0,1\}^m$ is given by letting $x_i = 1$ if $y_i \in S'$, and letting $x_i = 0$ if $z_i \in S'$. Let $1 \leq j \leq l$; in each column c_j, to sum to 3 in the bottom row, S' contains at least one row with a 1 in column j in the upper-right, since only two 1's are in the lower-right. Hence c_j is satisfied for every j, whence $\phi(x_1 x_2 \ldots x_m) = 1$. □

[14] One or both will be needed if the corresponding clause has some false literals, but at least one true literal must be present to satisfy the clause, so no more than two extra 1's are needed from the bottom-right to sum the whole column to 3.

LECTURE: end of day 18

7.6 Proof of the Cook-Levin Theorem

We prove Theorem 7.4.13 as follows. We first show that the language CircuitSat is NP-complete, where CircuitSat consists of the satisfiable Boolean *circuits.* We then show that CircuitSat $\leq^{\mathrm{P}}_{\mathrm{m}}$ 3Sat. Since every 3CNF formula is a special case of a Boolean formula, it is easy to check that 3Sat $\leq^{\mathrm{P}}_{\mathrm{m}}$ Sat. Thus, we we are done, we will have shown all three of these languages are NP-hard. Since they are all in NP, they are NP-complete.

Definition 7.6.1. A *Boolean circuit* is a collection of *gates* and *inputs* (i.e., nodes in a graph) connected by *wires* (directed edges). Cycles are not permitted (it is a directed acyclic graph). Gates are labeled And, Or (each with in-degree 2), or Not (with in-degree 1), and have unbounded out-degree. One gate is designated the *output gate.*

Given an n-input Boolean circuit γ and a binary input string $x = x_1 x_2 \ldots x_n$, the value $\gamma(x) \in \{0, 1\}$ is the value of the output gate when evaluating γ with the inputs given by each x_i, and the values of the gates are determined by computing the associated Boolean function of its inputs. A circuit γ is *satisfiable* if there is an input string x that satisfies γ; i.e., such that $\gamma(x) = 1$.[15]

Define

$$\textsc{CircuitSat} = \{ \ \langle \gamma \rangle \ | \ \gamma \text{ is a satisfiable Boolean circuit } \} .$$

Theorem 7.6.2. CircuitSat *is* NP*-complete.*

Proof Sketch. CircuitSat $\in$ NP because the language

$$\textsc{CircuitSat}_V = \{ \ \langle \gamma, w \rangle \ | \ \gamma \text{ is a Boolean circuit and } \gamma(w) = 1 \ \}$$

is a polynomial-time verification language for CircuitSat.[16]

Let $A \in \mathsf{NP}$, and let V be an $p(n)$-time-bounded verifier for A with witness length $q(n)$, so that, for all $x \in \{0, 1\}^*$,

$$x \in A \iff \left(\exists w \in \{0,1\}^{q(n)} \right) \ V(x, w) \text{ accepts.}$$

[15] The only difference between a Boolean formula and a Boolean circuit is the unbounded out-degree of the gates. A Boolean formula has out-degree one, so that when expressed as a circuit, have gates that form a *tree* (although note that even in a formula the input variables can appear multiple times and hence have larger out-degree), whereas a Boolean circuit, by allowing unbounded fanout from its gates, allows the use of *shared subformulas.* (For example, "Let $\varphi = (x_1 \wedge \overline{x_2})$ in the formula $\phi = x_1 \vee \varphi \wedge (\overline{x_4} \vee \overline{\varphi})$.") This is a technicality that will be the main obstacle to proving that CircuitSat $\leq^{\mathrm{P}}_{\mathrm{m}}$ 3Sat, but not a difficult one.

[16] To show that CircuitSat is NP-hard, we show how any verification algorithm can be simulated by a circuit, in such a way that the verification algorithm accepts a string if and only if the circuit is satisfiable. The input to the circuit will not be the first input to the verification algorithm, but rather, the *witness.*

To show that $A \le^{\mathrm{P}}_{\mathrm{m}}$ CircuitSat, we transform an input x to A into a circuit γ^V_x such that γ^V_x is satisfiable if and only if there is a $w \in \{0,1\}^{q(n)}$ such that $V(x,w)$ accepts.[17]

Let $V = (Q, \Sigma, \Gamma, \delta, s, q_{\mathtt{a}}, q_{\mathtt{r}})$ be the Turing machine deciding CircuitSat$_V$. V takes two inputs, $x \in \{0,1\}^n$ and the witness $w \in \{0,1\}^{q(n)}$. γ^V_x contains constant gates representing x, and its $q(n)$ input variables represent a potential witness w. We design γ^V_x so that $\gamma^V_x(w) = 1$ if and only if $V(x,w)$ accepts.

We build a subcircuit γ_{local}.[18] γ_{local} has $3m$ inputs and m outputs, where m depends on V – but *not* on x or w – as described below. Assume that each state $q \in Q$ and each symbol $a \in \Gamma$ is represented a binary string,[19] called σ_q and σ_a, respectively.

Let $m = 1 + \lceil \log |Q| \rceil + \lceil \log |\Gamma| \rceil$.[20] Represent V's configuration $C = (q, p, y)$[21] as an element of $\{0,1\}^{p(n) \cdot m}$ as follows. The p^{th} tape cell with symbol a is represented as the string $\sigma(p) = 1\sigma_q\sigma_a$. Every tape cell at positions $p' \ne p$ with symbol b are represented as $\sigma(p') = 0\sigma_s\sigma_b$.[22] Represent the configuration C by the string $\sigma(C) = \sigma(0)\sigma(1)\ldots\sigma(n^k - 1)$.

$\gamma_{\mathrm{local}} : \{0,1\}^{3m} \to \{0,1\}^m$ is defined to take as input $\sigma(k-1)\sigma(k)\sigma(k+1)$,[23] and output the next configuration string for tape cell k. γ_{local} can be implemented by a Boolean circuit whose size depends only on δ, so γ_{local}'s size is a constant depending on V but independent of n.[24]

To construct γ^V_x,[25] attach $p(n) \cdot (m \cdot p(n))$ copies of γ_{local} in a square array, where the t^{th} horizontal row of wires input to a row of γ_{local}'s represents the configuration of V at time step t. The input x to V is provided as input to the first n copies of γ_{local} by constant gates, and the input w to V is provided as input to the next $q(n)$ copies of γ_{local} by the input gates to γ^V_x. Finally, the $3m$ output wires from the final row are collated together into a single output gate that indicates whether the gate representing the tape head position was in state $q_{\mathtt{a}}$, so that the circuit will output 1 if and only if the state of the final configuration is accepting.

Since $V(x)$ runs in time $\le p(n)$ and therefore takes space $\le p(n)$, γ^V_x contains enough gates in the horizontal direction to represent the entire non-␣ portion of the tape of $V(x)$ at every step,

[17] In fact, w will be the satisfying assignment for γ^V_x. The subscript x is intended to emphasize that, while x is an input to V, it is hard-coded into γ^V_x; choosing a different input y for the same verification algorithm V would result in a different circuit γ^V_y.

The key idea will be that circuits can simulate algorithms. We prove this by showing that any Turing machine can be simulated by a circuit, as long as the circuit is large enough to accommodate the running time and space used by the Turing machine.

[18] Many copies of γ_{local} will be hooked together to create γ^V_x.

[19] This is done so that a circuit may process them as inputs.

[20] m is the number of bits required to represent a state and a symbol together, plus the boolean value "the tape head is currently here".

[21] where $q \in Q$ is the current state, $p \in \mathbb{N}$ is the position of the tape head, and $y \in \{0,1\}^{n^k}$ is the string on the tape from position 0 to $n^k - 1$. We may assume that y contains all of the non-␣ symbols that are on the tape, since V runs in time n^k and cannot move the tape head right by more than one tape cell per time step.

[22] That is, only the tape cell string with the tape head actually contains a representation of the current state, and the remaining tape cell strings have filler bits for the space reserved for the state; we have arbitrarily chosen state s to be the filler bits, but this choice is arbitrary. The first bit indicates whether the tape head is on that tape cell or not.

[23] the configuration strings for the three tape cells surrounding tape cell k

[24] It will be the *number of copies of* γ_{local} that are needed to simulate $V(x,w)$ that will depend on n, but we will show that the number of copies needed is polynomial in n.

[25] This is where the proof gets sketchy; to specify the proof in full detail, handling every technicality, takes pages and is not much more informative than the sketch we outline below.

and contains enough gates in the vertical direction to simulate $V(x)$ long enough to get an answer. Since the size of γ_{local} is constant (say, c), the size of the array is at most $cp(n)^2$. n additional constant gates are needed to represent x, and the answer on the top row can be collected into a single output gate in at most $O(p(n))$ gates. Therefore, $|\left\langle \gamma_x^V \right\rangle|$ is polynomial in n, whence γ_x^V is computable from x in polynomial time.

Since $\gamma_x^V(w)$ simulates $V(x,w)$, w satisfies γ_x^V if and only if $V(x,w)$ accepts, whence γ_x^V is satisfiable if and only if there is a witness w testifying that $x \in A$. □

Theorem 7.6.3. 3Sat *is* NP*-complete.*

Proof. 3Sat $\in$ NP for the same reason that Sat $\in$ NP: the language

$$3\textsc{Sat}_V = \{\ \langle \phi, w \rangle \mid \phi \text{ is a 3CNF Boolean formula and } \phi(w) = 1\ \}$$

is a polynomial-time verification language for 3Sat.

To show that 3Sat is NP-hard, we show that CircuitSat $\leq_{\text{m}}^{\text{P}}$ 3Sat.[26]

Let γ be a Boolean circuit with s gates; we design a 3CNF formula ϕ computable from γ in polynomial time, which is satisfiable if and only if γ is.[27] ϕ has all the input variables $x_1, \ldots, x_n$ of γ, and *in addition*, for each gate g_j in γ, ϕ has a variable y_j in ϕ representing the values of the output wire of gate g_j. Assume that y_1 is the output gate of γ.

For each gate g_j, ϕ has a subformula ψ_j that expresses the fact that the gate is "functioning properly" in relating its inputs to its outputs. For each gate g_j, with output y_j and inputs w_j and z_j,[28] define

$$\psi_j = \begin{cases} \begin{array}{ll} & (\overline{w_j} \vee \overline{y_j} \vee \overline{y_j}) \\ \wedge & (w_j \vee y_j \vee y_j) \end{array}, & \text{if } g_j \text{ is a } \textsc{Not} \text{ gate;} \\ \\ \begin{array}{ll} & (\overline{w_j} \vee \overline{z_j} \vee y_j) \\ \wedge & (\overline{w_j} \vee z_j \vee y_j) \\ \wedge & (w_j \vee \overline{z_j} \vee y_j) \\ \wedge & (w_j \vee z_j \vee \overline{y_j}) \end{array}, & \text{if } g_j \text{ is an } \textsc{Or} \text{ gate;} \\ \\ \begin{array}{ll} & (\overline{w_j} \vee \overline{z_j} \vee y_j) \\ \wedge & (\overline{w_j} \vee z_j \vee \overline{y_j}) \\ \wedge & (w_j \vee \overline{z_j} \vee \overline{y_j}) \\ \wedge & (w_j \vee z_j \vee \overline{y_j}) \end{array}, & \text{if } g_j \text{ is an } \textsc{And} \text{ gate.} \end{cases} \tag{7.6.1}$$

[26] The main obstacle to simulating a Boolean circuit with a Boolean formula is that circuits allow unbounded fan-out and formulas do not. The naïve way to handle this would be to make a separate copy of the subformula representing a non-output gate of the circuit, one copy for each output wire. The problem is that this could lead to an exponential increase in the number of copies, as subformulas could be copied an exponential number of times if they are part of larger subformulas that are also copied. Our trick to get around this will actually lead to a formula in 3CNF form.

[27] ϕ is not equivalent to γ: ϕ has more input bits than γ. But it will be the case that ϕ is satisfiable if and only if γ is satisfiable; it will simply require specifying more bits to exhibit a satisfying assignment for ϕ than for γ.

[28] w_j being the only input if g_j is a $\neg$ gate, and each input being either a γ-input variable x_i or a ϕ-input variable y_i representing an internal wire in γ

Observe that, for example, a $\wedge$ gate with inputs a and b and output c is operating correctly if

$$\begin{aligned} & (a \wedge b \implies c) \\ \wedge \quad & (a \wedge \overline{b} \implies \overline{c}) \\ \wedge \quad & (\overline{a} \wedge b \implies \overline{c}) \\ \wedge \quad & (\overline{a} \wedge \overline{b} \implies \overline{c}) \end{aligned}$$

Applying the fact that the statement $p \implies q$ is equivalent to $\overline{p} \vee q$ and DeMorgan's laws gives the expressions in equation (7.6.1).

To express that γ is satisfied, we express that the output gate outputs 1, and that all gates are properly functioning:

$$\phi = (y_1 \vee y_1 \vee y_1) \wedge \bigwedge_{j=1}^{s} \psi_j.$$

The only way to assign values to the various y_j's to satisfy ϕ is for γ to be satisfiable, and *furthermore* for the value of y_j to actually equal the value of the output wire of gate g_j in γ, for each $1 \leq j \leq s$.[29] Thus ϕ is satisfiable if and only if γ is satisfiable. □

LECTURE: end of day 19

[29] That is to say, even if $x_1 x_2 \ldots x_n$ is a satisfying assignment to γ, a satisfying assignment to ϕ includes not only $x_1 x_2 \ldots x_n$, but also the correct values of y_j for each gate j; getting these wrong will fail to satisfy ϕ even if $x_1 x_2 \ldots x_n$ is correct.