

# **Theory of Computation**

ECS 120

Lecture Notes

David Doty

Copyright © March 19, 2020, David Doty

No part of this document may be reproduced without the expressed written consent of the author. All rights reserved.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What this course is about	1
1.2	Automata, computability, and complexity	3
1.3	Mathematical background	6
1.3.1	Implication statements	6
1.3.2	Sets	6
1.3.3	Sequences and tuples	7
1.3.4	Functions and relations	8
1.3.5	The pigeonhole principle	9
1.3.6	Combinatorics	9
1.3.7	Graphs	11
1.3.8	Boolean logic	11
1.4	Proof by induction	11
1.4.1	Proof by induction on natural numbers	11
1.4.2	Induction on other structures	12
<b>I</b>	<b>Automata Theory</b>	<b>15</b>
<b>2</b>	<b>String theory</b>	<b>17</b>
2.1	Why to study automata theory	17
2.2	Definitions	18
2.3	Binary numbers	21
<b>3</b>	<b>Deterministic finite automata</b>	<b>23</b>
3.1	Intuitive overview of deterministic finite automata (DFA)	23
3.2	Formal models of computation	24
3.3	Formal definition of a DFA (syntax)	25
3.4	More examples	27
3.5	Formal definition of computation by a DFA (semantics)	28

<b>4</b>	<b>Declarative models of computation</b>	<b>31</b>
4.1	Regular expressions	31
4.1.1	Formal definition of a regex (syntax and semantics)	32
4.2	Context-free grammars	35
4.2.1	Formal definition of a CFG (syntax)	35
4.2.2	Formal definition of computation by a CFG (semantics)	36
4.2.3	Right-regular grammars (RRG)	37
4.3	Nondeterministic finite automata	37
4.3.1	Formal definition of an NFA (syntax)	38
4.3.2	Formal definition of computation by an NFA (semantics)	40
<b>5</b>	<b>Closure of language classes under set operations</b>	<b>43</b>
5.1	Automatic transformation of regex's, NFAs, DFAs	43
5.2	DFA union and intersection (product construction)	46
5.3	NFA union, concatenation, and star constructions	50
5.3.1	Examples	50
5.3.2	Proofs	52
<b>6</b>	<b>Equivalence of models</b>	<b>57</b>
6.1	Equivalence of DFAs and NFAs (subset construction)	57
6.2	Equivalence of RGs and NFAs	60
6.3	Equivalence of regex's and NFAs	62
6.3.1	Every regex-decidable language is NFA-decidable	62
6.3.2	Every NFA-decidable language is regex-decidable	63
6.4	Optional: Equivalence of DFAs and constant memory programs	66
<b>7</b>	<b>Proving problems are not solvable in a model of computation</b>	<b>69</b>
7.1	Some languages are not regular	69
7.2	The pumping lemma for regular languages	70
7.2.1	Using the Pumping Lemma	71
7.2.2	Proof of the Pumping Lemma	74
7.3	Optional: The Pumping Lemma for context-free languages	75
<b>II</b>	<b>Computational Complexity and Computability Theory</b>	<b>79</b>
<b>8</b>	<b>Turing machines</b>	<b>81</b>
8.1	Intuitive idea of Turing machines	81
8.2	Formal definition of a TM (syntax)	83
8.3	Formal definition of computation by a TM (infinite semantics)	84
8.4	Optional: Formal definition of computation by a TM (finite semantics)	85
8.5	Languages recognized/decided by TMs	86
8.6	Variants of TMs	87

8.6.1	Multitape TMs	87
8.6.2	Other variants	89
8.7	TMs versus code	91
8.8	Optional: The Church-Turing Thesis	92
<b>9</b>	<b>Efficient solution of problems: The class P</b>	<b>95</b>
9.1	Asymptotic analysis	95
9.1.1	Analyzing algorithms	98
9.1.2	Optional: Time complexity of simulation of multitape TM with a one-tape TM	99
9.2	Definition of P	100
9.2.1	Polynomial time	100
9.2.2	Examples of problems in P	102
9.2.3	Why identify P with “efficient”?	105
<b>10</b>	<b>Efficient verification of solutions: The class NP</b>	<b>107</b>
10.1	The class NP	109
10.1.1	Definition of NP	109
10.1.2	Decision vs. Search vs. Optimization	110
10.2	Examples of problems in NP	111
10.2.1	The P versus NP question	112
10.3	NP problems are decidable in exponential time	113
10.4	Introduction to NP-Completeness	115
10.5	Polynomial-time reducibility	118
10.6	Definition of NP-completeness	126
10.6.1	The Cook-Levin Theorem	128
10.7	Optional: Additional NP-Complete problems	129
10.7.1	Vertex Cover	129
10.7.2	Subset Sum	131
10.7.3	Hamiltonian path	133
10.8	Optional: Proof of the Cook-Levin Theorem	133
10.9	Optional: A brief history of the P versus NP problem	137
<b>11</b>	<b>Undecidability</b>	<b>141</b>
11.1	The Halting Problem	141
11.1.1	Reducibility	144
11.2	Undecidable problems about algorithm behavior	144
11.2.1	No-input halting problem	144
11.2.2	Accepting a given string	146
11.2.3	Accepting at least one string	147
11.2.4	Rejecting at least one string	148
11.2.5	How to recognize undecidability at a glance	149

11.3	Optional: Enumerators as an alternative definition of Turing-recognizable . .	151
11.3.1	Optional: A non-Turing-recognizable language . . . . .	155
11.4	The Halting Problem is undecidable . . . . .	158
11.4.1	Diagonalization . . . . .	158
11.4.2	The Halting Problem is undecidable . . . . .	162
11.5	Optional: The far-reaching consequences of undecidability . . . . .	164
11.5.1	Gödel's Incompleteness Theorem . . . . .	164
11.5.2	Prediction of physical systems . . . . .	166
<b>A</b>	<b>Reading Python</b>	<b>167</b>
A.1	Installing Python . . . . .	167
A.2	Code from this book . . . . .	167
A.3	Tutorial on reading Python . . . . .	168

# Chapter 1

## Introduction

### 1.1 What this course is about

*What are the fundamental capabilities and limitations of computers?*

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

- In ECS 36a/b/c, you programmed computers without studying them mathematically.
- In ECS 20, you mathematically studied things that are not computers.
- In ECS 120, you will use the tools of ECS 20 to mathematically study computers.

The fundamental premise of the theory of computation is that **the computer on your desk—or in your pocket—obeys certain laws, and therefore, certain unbreakable limitations.** We can reason by analogy with the laws of physics. Newton’s equations of motion tell us that each object with mass obeys certain rules that cannot be broken. For instance, an object cannot accelerate in the opposite direction in which force is being applied to it. Of course, nothing in the world is a rigid frictionless body of mass obeying all the idealized assumptions of classical mechanics, (“the map is not the territory”). So in reality, 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 Newtonian predictions are reasonably accurate.

I often think of the field of computer science outside of theory as being about proving what can be done with a computer... simply 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 claim is an argument that convinces an intelligent person who has never seen the claim**

before and cannot see why it is true with having it explained. It does not matter if the argument uses formal mathematical notation or not (though formal notation is briefer and more straightforward to make precise than English), or if it uses proof by induction or proof by contradiction or just a direct proof (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*.

However, one effective technique to prove theorems (I do this in nearly all of my research papers) is to first give an informal “proof sketch”, intuitively explaining how the proof will go, but shorter than the proof and lacking in potentially confusing (but necessary) details. The proof is easier to read if one first reads the proof sketch, but the proof sketch 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 mathematical notation, without any intuition to help someone understand it, do not communicate why the theorem is true any better than the hand-waving proof sketch. Both are usually necessary to accomplish the goal of the proof: to *help the reader understand* why the theorem is true.

Here’s an example of a formal theorem, and informal proof sketch, and a formal proof:

**Theorem 1.1.1.** *There are infinitely many prime numbers.*

*Proof sketch.* Intuitively, we show that for each finite set of prime numbers, we can find a larger prime number. We do this by doing multiplying together the existing primes and adding 1. It is a new prime because, if it were a multiple of an existing prime, it would be only one greater than another multiple of that prime, a contradiction.

*Proof.* Let  $S = \{p_1, p_2, \dots, p_k\} \subset \mathbb{N}$  be any finite set of primes. It suffices to show that we can find a prime not in  $S$ , implying that  $S$  cannot be all the primes. Since  $S$  is an *arbitrary* finite set of primes, this shows that the set of all prime numbers cannot be finite.

Let  $p = p_1 \cdot p_2 \cdot \dots \cdot p_k$  and let  $n = p + 1$ . Since  $n$  is larger than any number in  $S$  (thus unequal to any of them), it suffices to show  $n$  is also prime. Suppose for the sake of contradiction that  $n$  is not prime; then it has a prime factor  $p_i \neq n$ , where  $p_i \mid n$  (i.e.,  $p_i$  divides  $n$ ). Note that  $p_i \mid p$  as well. Therefore  $n$  and  $p$  are different multiples of  $p_i$ , so their difference  $n - p$  must be at least  $p_i$ . (For example, note that all multiples of 3 are at least 3 apart: 3, 6, 9, 12, 15, ...) But  $n - p = 1$ , which is smaller than any prime, a contradiction.  $\square$

Note that the proof sketch is shorter, uses less formal notation, and skips some details. As a result, it is easy to read, but incomplete. The proof gives all the details, but the proof sketch is a “roadmap” helping to guide the reader through it.

In this course, in the interest of time, I will often give the intuitive proof sketch only verbally, and write only the formal details on the board. 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.



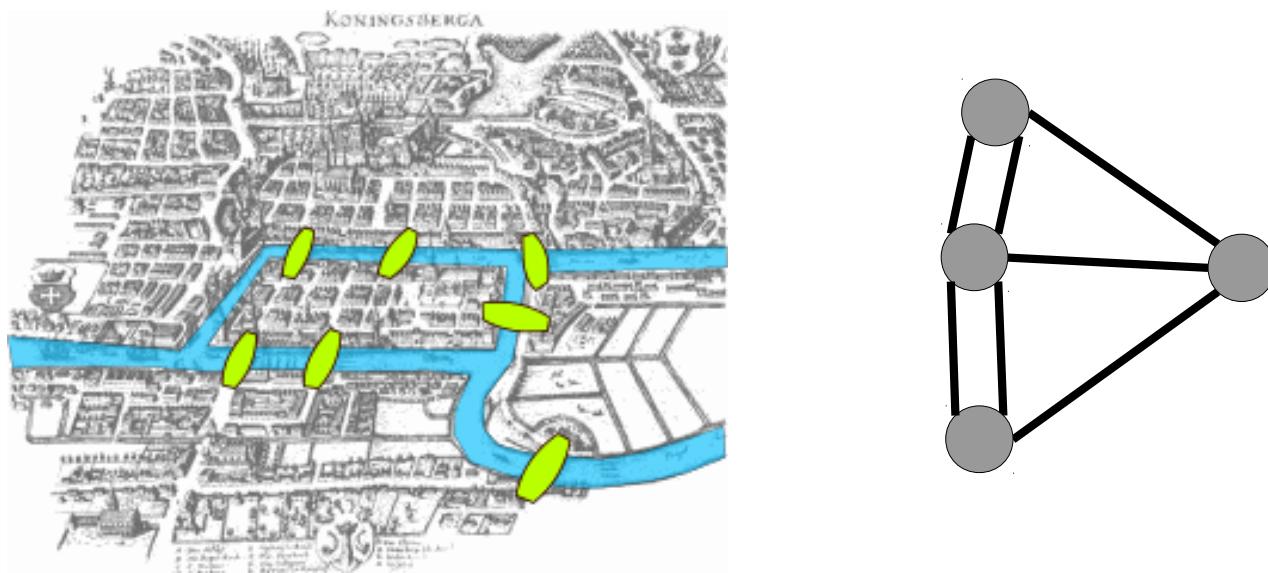


Figure 1.1: “Königsberg graph”. Licensed under CC BY-SA 3.0 via Commons – [https://upload.wikimedia.org/wikipedia/commons/5/5d/Konigsberg\\_bridges.png](https://upload.wikimedia.org/wikipedia/commons/5/5d/Konigsberg_bridges.png)

## 1.2 Automata, computability, and complexity

### Multivariate polynomials

Consider the polynomial equation  $x^2 - y^2 = 2$ . Does this have a solution? Yes:  $x = \sqrt{2}, y = 0$ . What about an *integer* solution? No.

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

Equations involving multivariate polynomials which we ask for a integer solution are called *Diophantine equations*.

**Task A:** write an algorithm that indicates whether a given multivariable polynomial equation has a **real solution**.

**Fact:** Task A is **possible**.

**Task A':** write an algorithm that indicates whether a given multivariable polynomial equation has an **real integer solution**.

**Fact:** Task A' is **impossible**.<sup>1</sup>

### Paths touring a graph

Does the graph  $G = (V, E)$  given in Figure 1.1 have a path that contains each edge exactly once? (*Eulerian path*)

<sup>1</sup>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.

**Task B:** write an “efficient” algorithm that indicates if a graph  $G$  has a path visiting each edge exactly once

**Algorithm:** Check whether  $G$  is connected (by breadth-first search) and every node has even degree.

**Task B':** write an “efficient” algorithm that indicates if a graph  $G$  has a path visiting each edge node exactly once

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

## Counting

A *regular expression (regex)* 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.

If  $x$  is a binary string and  $a$  a symbol, let  $\#_a(x)$  be the number of  $a$ 's in  $x$ . For example,  $\#_0(001200121) = 4$  and  $\#_1(001200121) = 3$ .

**Task C:** write a regex that matches a ternary string  $x$  exactly when  $\#_0(x) + \#_1(x) = 3$ .

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

**Task C':** write a regex that matches a ternary string  $x$  exactly when  $\#_0(x) - \#_1(x) = 3$ .

**Fact:** Task  $C'$  is impossible

## Rough layout of this course

The following are the units of this course:

**Computability theory (unit 3):** What problems can algorithms solve? (finding real roots of multivariate polynomials, but not integer roots)

**Computational complexity theory (unit 2):** What problems can algorithms solve efficiently? (finding paths visiting every edge, but not every node)

**Automata theory (unit 1):** What problems can algorithms solve with “optimal” efficiency (constant space and “real time”, i.e., time = size of input)? (finding whether the sum of  $\#$  of 0's and  $\#$  of 1's equals some constant, but not the difference)

Sorting these by increasing order of the power of the algorithms studied: 1, 2, 3.

Historically, these were discovered in the order 3, 1, 2.

It is most common in a theory course to cover them in order 1, 3, 2.

**We are going to cover them in order 1, 2, 3.**

The reason for swapping the traditional order of units 3 (computability) and 2 (computational complexity) is this: Most of computer science is about writing programs to solve

problems. You think about a problem, you write a program, and you demonstrate that the program solves the problem.<sup>2</sup>

Now, in computability theory, **unit 3, most of the problems studied are questions about the behavior of programs themselves.**<sup>3</sup> But what sort of objects might answer these questions about programs? Other programs! So we imagine writing a program  $P$  that takes as input the source code of another program  $Q$ . This isn't such a crazy idea *yet*... compilers, interpreters (such as the Javascript engine in a web browser), and virtual machines are programs that take as input the source code of other programs.

But in computability theory, the situation complicates quickly. A typical problem involves proving that a certain problem is not solvable by *any* program. Usually this done by writing a program  $P$ , which takes as input another program  $Q$ , and  $P$  outputs a third program  $R$ .<sup>4</sup> It can be quite taxing to keep track of which program is supposed to be doing what. So in the interest of showing no program exists to solve a certain problem, we introduce not one but *three* new programs, all of which are *not* solving that problem. It's quite remarkable that anyone was able to create such a chain of reasoning in the first place to prove limits on the ability of programs. It adds many conceptual layers of abstraction onto what most computer science students are accustomed to doing.

On the other hand, computational complexity theory, **unit 2, has the virtue that most problems are about more mundane objects such as lists of integers, graphs, and Boolean formulas.** In complexity theory, we think about programs that take these objects as input and produce them as output, and there is less danger of getting the program confused with a graph, for instance, because they are two different types of objects.

The ideas in units 2 and 3 are more similar to each other than they are to unit 1. Also, we will spend about half the course on unit 1, and the other half on units 2 and 3. So, these notes are divided into two “parts”: part 1 is unit 1, automata theory, and part 2 is units 2 and 3, computational complexity and computability theory.

---

<sup>2</sup>In algorithms and theory courses and research, to “demonstrate correctness” usually means a proof. In software engineering, it usually means unit tests and code reviews. For critical stuff like spacecraft software, it involves *both*.

<sup>3</sup>The Diophantine equation problem above is a notable exception, but it took 70 years to prove it is unsolvable, and even then it was done by creating a Diophantine equation that essentially “mimics” the behavior of a program so as to connect the existence of its roots to a question about the behavior of certain programs.

<sup>4</sup>Now, this sort of idea, of programs receiving and producing other programs, is not crazy in principle. The C compiler, for example, is itself a program, which takes as input the source code of a program (written in C) and outputs the code of another program (machine instructions, written in the “language” of the host machine’s instruction set). The C preprocessor (which rewrites macros, for instance) takes as input C programs and produces C programs as output.

## 1.3 Mathematical background

### 1.3.1 Implication statements

Given two boolean statements  $p$  and  $q$ <sup>5</sup>, the *implication*  $p \implies q$  is shorthand for “ $p$  implies  $q$ ”, or “If  $p$  is true, then  $q$  is true”<sup>6</sup>,  $p$  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*:  $\neg q \implies \neg p$ <sup>7</sup>

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.

### 1.3.2 Sets

A *set* is a group of objects, called *elements*, with no duplicates.<sup>8</sup> 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*  $\mathbb{N} = \{0, 1, 2, \dots\}$
- the *integers*  $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
- the *rational numbers*  $\mathbb{Q} = \left\{ \frac{p}{q} \mid p \in \mathbb{Z}, q \in \mathbb{N}, q \neq 0 \right\}$
- the *real numbers*  $\mathbb{R}$

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

To define sets symbolically,<sup>9</sup> we use *set-builder notation*: for instance,  $\{x \in \mathbb{N} \mid x \text{ is odd}\}$  is the set of all odd natural numbers.

<sup>5</sup>e.g., “Hawaii is west of California”, or “The stoplight is green.”

<sup>6</sup>e.g., “If the stoplight is green, then my car can go.”

<sup>7</sup>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.

<sup>8</sup>Think of `std::set`.

<sup>9</sup>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.

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  $A$  such that ...”. For example,  $(\exists n \in \mathbb{N}) n > 10$  means “there exists a natural number  $n$  such that  $n$  is 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  $A - B$ ).  $\bar{A} = \{ x \mid x \notin A \}$  is the *complement* of  $A$ .<sup>10</sup>

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  $|A| < \infty$ .<sup>11 12</sup>

### 1.3.3 Sequences and tuples

A *sequence* is an ordered list of objects<sup>13</sup>. For example,  $(7, 21, 57, 21)$  is the sequence of integers 7, then 21, then 57, then 21.

A *tuple* is a finite sequence.<sup>14</sup>  $(7, 21, 57)$  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  $A \times B = \{ (a, b) \mid a \in A \text{ and } b \in B \}$ . Note that  $|A \times B| = |A| \cdot |B|$ . For  $k \in \mathbb{N}$ , we write  $A^k = \underbrace{A \times A \times \dots \times A}_{k \text{ times}}$  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.

<sup>10</sup>Usually, if  $A$  is understood to be a subset of some larger set  $U$ , the “universe” of possible elements, then  $\bar{A}$  is understood to be  $U \setminus A$ . For example if we are dealing only with  $\mathbb{N}$ , and  $A \subseteq \mathbb{N}$ , then  $\bar{A} = \{ n \in \mathbb{N} \mid n \notin A \}$ . 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  $A \subseteq \mathbb{N}$ ,  $\bar{A}$  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  $A$ .

<sup>11</sup>Why?

<sup>12</sup>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.

<sup>13</sup>Think of `std::vector`.

<sup>14</sup>The closest C++ 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 C++ 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 a `void[]` array of length 5.

### 1.3.4 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  $f : D \rightarrow R$ .<sup>15</sup> Given  $A \subseteq D$ , define  $f(A) = \{ f(x) \mid x \in A \}$ ; call this the *image of  $A$  under  $f$* .

Given  $f : D \rightarrow D$ ,  $k \in \mathbb{N}$  and  $d \in D$ , define  $f^k : D \rightarrow D$  by  $f^k(d) = \underbrace{f(f(\dots f(d))\dots)}_{k \text{ times}}$

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*.<sup>16</sup> If  $f$  is defined on all values, it is a *total function*.<sup>17</sup>

A function  $f$  with a finite domain can be represented with a table. For example, the function  $f : \{0, 1, 2, 3\} \rightarrow \mathbb{Q}$  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*).<sup>18</sup>

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  $R = \{ (n, m) \in \mathbb{N} \times \mathbb{N} \mid (\exists k \in \mathbb{N}) n + k = m \}$ .

<sup>15</sup>Most statically typed programming languages like C++ have direct support for functions with declared types for input and output. In Java, these are like static methods; `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`  $\times$  `int` (since it accepts a pair of `ints` as input) and range `int`. The main difference between functions in programming languages and those in mathematics is that in a programming language, a function is really an *algorithm* for computing the output, given in the input, whereas in mathematics the function is just the abstract relationship between input and output, and there may not be any algorithm computing it.

<sup>16</sup>For instance, `Integer.parseInt` is (strictly) partial, because not all `Strings` look like integers, and such `Strings` will cause the method to throw a `NumberFormatException`.

<sup>17</sup>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.

<sup>18</sup>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  $f$ ).

### 1.3.5 The pigeonhole principle

The pigeonhole principle is a very simple concept, but it is surprisingly powerful. It says that if we put  $n$  objects  $a_1, a_2, \dots, a_n$  into fewer than  $n$  boxes, then there is a box with at least two objects. Usually, we think of the boxes as being “properties” or “labels” the objects have: if there are  $n$  objects but fewer than  $n$  labels, then two objects must get the same label.

For example, all people have fewer than 200,000 hairs on their head. There are about 500,000 people in Sacramento. If we think of the people as the objects  $a_1, a_2, \dots, a_{500,000}$ , and we think of the count of hairs as the labels (perhaps person  $a_1$  has 107,235 hairs, person  $a_2$  has 95,300 hairs, etc.) then since there are only 200,000 labels  $0, 1, 2, \dots, 199,999$ , at least two people must have the exact same number of hairs on their head.

Here’s a way of stating the pigeonhole principle using functions. **If I have a set  $O$  of objects and a set  $B$  of boxes, and if  $|O| > |B|$ , then every function  $f : O \rightarrow B$  is not 1-1.** In other words, for every function  $f : O \rightarrow B$ , there are two different objects  $o_1, o_2 \in O$  such that  $f(o_1) = f(o_2)$ , i.e.,  $f$  puts  $o_1$  and  $o_2$  in the same box.<sup>19</sup>

### 1.3.6 Combinatorics

Counting sizes of sets is a tricky art to learn. It’s a whole subfield of mathematics called *combinatorics*, with very deep theorems, but we won’t need anything particularly deep in this course. For simple counting problems, a few basic principles apply.

The first is sometimes called **“The Product Rule”**: **If a set is defined by cross-product (i.e., each element of a set is a tuple), then often multiplying the sizes of the smaller sets works.** For example, there are 4 integers in the set  $A = \{1, 2, 3, 4\}$  and 3 integers in the set  $B = \{1, 2, 3\}$ . How many ways are there to pick one element from  $A$  and one from  $B$ ? There are 12 pairs of integers in the set  $A \times B$ , because to choose a pair  $(a, b) \in A \times B$ , there are 4 ways to choose  $a$  and 3 ways to choose  $b$ , so  $4 \cdot 3$  ways to choose both. Here they all are:

	1	2	3
1	(1, 1)	(1, 2)	(1, 3)
2	(2, 1)	(2, 2)	(2, 3)
3	(3, 1)	(3, 2)	(3, 3)
4	(4, 1)	(4, 2)	(4, 3)

If the tuple is bigger, you just keep multiplying: in the triple  $(a, b, c)$ , if there are 4 ways to choose  $a$ , 3 ways to choose  $b$ , and 7 ways to choose  $c$ , then there are  $4 \cdot 3 \cdot 7 = 84$  possible triples  $(a, b, c)$ .

<sup>19</sup>In fact, we will eventually see (Section 11.4.1) that such reasoning applies even to *infinite* sets. If  $O$  and  $B$  are both infinite, but there is no 1-1 function  $f : O \rightarrow B$ , then any way of assigning objects from  $O$  to “boxes” in  $B$  must assign two objects to the same box. For example, we will see that if  $O$  is the set of all decision problems (defined formally in Section 2.2) we want to solve with algorithms (defined formally as Turing machines in Chapter 8), and  $B$  is the set of all algorithms, then  $|O| > |B|$ , so there is no 1-1 function  $f : O \rightarrow B$ . So any way of assigning each decision problem uniquely to an algorithm that solves it must fail (by assigning two different problems to the same algorithm, which can’t very well solve both of them). Thus some decision problems have no algorithm solving them.



As another example, what's the number of functions  $f : \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, k\}$ ? There are  $k$  ways to choose what  $f(1)$  is, times  $k$  ways to choose what  $f(2)$  is, etc., and there are  $n$  such values to choose. So there are  $\underbrace{k \cdot k \cdot \dots \cdot k}_n = k^n$  ways to choose  $f$ .

The key here is that the element from set is “chosen independently”, meaning it's possible to have any combination of them together, with no extra constraints. Sometimes this doesn't hold. Suppose I define a new set  $C$ , not equal to  $A \times B$ , but defined as the set of all pairs  $(a, b)$ , where  $a \in A$ ,  $b \in B$ , and  $a \leq b$ . Then  $C$  is not equal to  $A \times B$ , but instead  $C$  is a strict subset of  $A \times B$  (meaning there are extra constraints applied so that not all possible pairs in  $A \times B$  are allowed in  $C$ ), because of the last condition that  $a \leq b$ , so you cannot choose  $a$  and  $b$  independently. Here is that set (with 6 elements) for  $A = \{1, 2, 3, 4\}$  and  $B = \{1, 2, 3\}$ :

	1	2	3
1	(1, 1)	(1, 2)	(1, 3)
2		(2, 2)	(2, 3)
3			(3, 3)
4			

In the function example, if we had asked what's the number of *non-decreasing* functions  $f : \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, k\}$ , then the choices are not independent: If we choose  $f(1) = 5$ , then we could not choose  $f(2) = 3$ .

The other basic tools that get used are:

- There are  $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$  permutations of a sequence of  $n$  elements. For example, the  $3! = 3 \cdot 2 \cdot 1 = 6$  permutations of  $\{1, 2, 3\}$  are 123, 132, 213, 231, 312, 321.
- There are  $\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (k+2) \cdot (k+1)}{(n-k) \cdot (n-k-1) \cdot \dots \cdot 2 \cdot 1}$  ways to choose a set of  $k$  elements from a larger set of  $n$  elements. For example, the number of binary strings of length 5 with exactly two 1's is  $\binom{5}{2} = \frac{5 \cdot (5-1)}{2} = 10$ : that's the number of ways to pick 2 positions in the string to be 1, out of 5 possible positions: 00011, 00101, 01001, 10001, 00110, 01010, 10010, 01100, 10100, 11000.
- Split up into sub-cases that can be added together. The next part uses this idea.
- Watch carefully for parts that are dependent, to try to phrase how to choose them in a way that involves only independent choices. For example, suppose I ask how many 4-digit integers have the same first and last digit (e.g., 1471, 5015, 3223). If I say “there are 9 ways to choose the first digit, times 10 ways to choose the second, times 10 ways to choose the third, times 10 ways to choose the fourth, but there is some tricky dependence with the first and fourth”, it's not clear what to do. But we can phrase it like this: there are  $9 \cdot 10 \cdot 10 = 900$  ways to choose the first three digits, and once I've picked the first digit, there is no choice with the fourth. So there are 900 total such numbers.



A slightly trickier example is (where we analyze two different sub-cases): how many 4 digit numbers have their first digit within 1 of their last digit? (for example: 1230, 1231, 1232, 9009, 9008) Well, there's still 900 ways to choose the first three digits, but there's now 3 ways to choose the last digit, unless the first digit is 9, and then there's only 2 ways to choose the last digit. This gives two sub-cases: the first digit is 9 or it isn't.

There are 800 ways to choose the first three digits with the first not equal to 9, times 3 ways to choose the last digit ( $800 \cdot 3 = 2400$ ). There are 100 ways to choose the first three digits with the first equal to 9, times 2 ways to choose the last digit ( $100 \cdot 2 = 200$ ). Combining the sub-cases, there are  $2400 + 200 = 2600$  such numbers.

### 1.3.7 Graphs

See slides.

### 1.3.8 Boolean logic

See slides.

## 1.4 Proof by induction

Proof by induction is a potentially confusing concept, seeming a bit mysterious compared to direct proofs or proofs by contradiction. 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.

### 1.4.1 Proof by induction on natural numbers

**Theorem 1.4.1.** *For every  $n \in \mathbb{N}$ ,  $|\{0, 1\}^n| = 2^n$ .*

*Proof.* (by induction on  $n$ ) <sup>20</sup>

**Base case:**  $\{0, 1\}^0 = \{\varepsilon\}$ .<sup>21</sup>  $|\{\varepsilon\}| = 1 = 2^0$ , so the base case holds.

**Inductive case:** Assume  $|\{0, 1\}^{n-1}| = 2^{n-1}$ .<sup>22</sup> 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  $\{0, 1\}^n$ ,

<sup>20</sup>To start, state in English what the theorem is saying: For every string length  $n$ , there are  $2^n$  strings of length  $n$ .

<sup>21</sup>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.

<sup>22</sup>Call this the *inductive hypothesis*, the fact we get to assume is true in proving the inductive case.

namely  $x_0$  and  $x_1$ .<sup>23</sup> Then

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

□

Of course, there are other (non-induction) ways to see that  $|\{0, 1\}^n| = 2^n$ . For example, using the Product Rule for counting, we can say that there are 2 ways to choose the first bit, times 2 ways to choose the second bit, ..., times 2 ways to choose the last bit, so  $\underbrace{2 \cdot 2 \cdot \dots \cdot 2}_n = 2^n$  ways to choose all of them. This is a sort of “iterative” reasoning that is more cleanly (but also more verbosely and pedantically) captured by the inductive argument above.

**Theorem 1.4.2.** *For every  $n \in \mathbb{N}^+$ ,  $\sum_{i=1}^n \frac{1}{i(i+1)} = \frac{n}{n+1}$ .*

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

**Inductive case:** Let  $n \in \mathbb{N}^+$  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. □

### 1.4.2 Induction on other structures

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.

---

<sup>23</sup>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  $|\{0, 1\}^n| = 2 \cdot |\{0, 1\}^{n-1}|$

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  $\#_0(x)$ .<sup>24</sup>

$$\#_0(x) = \begin{cases} 0, & \text{if } x = \varepsilon; & \text{(base case)} \\ \#_0(w) + 1, & \text{if } x = w0 \text{ for some } w \in \{0, 1\}^*; & \text{(inductive case)} \\ \#_0(w), & \text{if } x = w1 \text{ for some } w \in \{0, 1\}^*. & \text{(inductive case)} \end{cases}$$

To prove a theorem by induction, identify the base case as the “smallest” object<sup>25</sup> for which the theorem holds.<sup>26</sup>

**Theorem 1.4.3.** *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

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

<sup>24</sup>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.

<sup>25</sup>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).

<sup>26</sup>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  $n$ )



Part I

**Automata Theory**



## Chapter 2

# String theory

No, not *that* string theory. In this chapter we cover the basic theoretical definitions and terminology used to talk about the kind of strings found as a data type in programming languages: finite sequences of symbols.

Some people also call this *language theory*, which is an odd name. It was developed at a time when connections were being discovered between linguistics and the theory of computation. As such, many of the terms sound strange to a computer scientist, but make sense if one considers using the theory to model natural languages. Modern-day natural language processing is as much machine learning techniques as linguistics. However, the most elegant application of this theory has been to the development of parsers and compilers for artificial *programming* languages.

It is also sometimes the case that “language theory” refers more broadly to “automata theory”, which is the subject of this whole part of the book, including finite automata, regular expressions, and context-free grammars. But the data that those automata are designed to process is strings, so we necessarily start the study with strings.

### 2.1 Why to study automata theory

It is common—and unfortunate—for modern textbooks on the theory of computation to dismiss automata theory as passé or to skip it entirely. Ironically, automata theory is a victim of its own success. A research field is interesting when there are many questions whose answers are unknown. The P vs. NP question (covered in Chapter 10), the biggest open question in theoretical computer science, still open over 40 years after first being posed, has ignited a huge number of areas of research. It is interesting precisely because it has raised so many questions that seem important, but that we don’t know how to answer.

Automata theory, on the other hand, has fewer fundamental open questions. Precisely because it succeeded in giving clean, elegant answers to questions, it is not as active an area of *current* research. Its well-understood—and often easier to understand—results, which make it suitable for an undergrad course, are precisely what make it less appealing for modern research. Many modern theoretical computer scientists find that they don’t need to use their

knowledge of automata theory to make progress in their subfield of theoretical computer science. Perhaps they bore of teaching automata theory because (at the undergraduate level) it doesn't require the use of deep, difficult mathematics and lengthy, complex proofs.

However, automata theory has had one enormously impactful application outside of theory: *the design of source code parsers for compilers and interpreters*. The centrality of automata theory to parsing, and its ubiquity, means that automata theory single-handedly dwarfs the impact of nearly any other subfield of theoretical computer science, including the very impactful notion of NP-completeness.<sup>1</sup>

Even to those with no interest whatsoever in the theory of computation, this one application alone is reason enough for every computer scientist to develop a solid grasp of automata theory. New programming languages are developed every year. Even if you never work on writing a new fully general-purpose programming language yourself, one fundamental tool, which should be in the toolbox of every software developer, is the ability to create a *domain-specific language* ([https://en.wikipedia.org/wiki/Domain-specific\\_language](https://en.wikipedia.org/wiki/Domain-specific_language)). This requires an understanding of parsers, and parsers are built out of the finite automata, regular expressions, and context-free grammars that we will study in the next few chapters.

And before we study those, we need a common vocabulary to talk about the data they process: strings.

## 2.2 Definitions

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

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

is the Roman alphabet. Symbols in the alphabet are usually single-character identifiers.

A *string* (a.k.a., *word*) over an alphabet is a finite sequence of symbols taken from the alphabet. We write strings such as 010001, without the parentheses and commas.<sup>2</sup> If  $x$  is a string,  $|x|$  denotes the *length* of  $x$ .

If  $\Sigma$  is an alphabet, the set of all strings over  $\Sigma$  is denoted  $\Sigma^*$ . For  $n \in \mathbb{N}$ ,  $\Sigma^n = \{x \in \Sigma^* \mid |x| = n\}$  is the number of strings in  $\Sigma^*$  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  $\varepsilon$ , and in some textbooks,  $\lambda$ . In most programming languages it is written `""`.

Note in particular the difference between  $\varepsilon$ ,  $\emptyset$ , and  $\{\varepsilon\}$ .<sup>3</sup>

<sup>1</sup>Cryptography is arguably equally impactful on applications where security is a concern. However, compilers and interpreters are needed for *all* software, regardless of security. If it weren't for the compilers that enable high-level languages, we'd still be implementing cryptographic software—and all other software—in machine code.

<sup>2</sup>If we used the standard sequence notation, 010001 would be written  $(0, 1, 0, 0, 0, 1)$

<sup>3</sup> $\varepsilon$  is a string,  $\emptyset$  is a set with no elements, and  $\{\varepsilon\}$  is a set with one element. The following Python code defines these three objects

```
epsilon = ""
```



Given  $n, m \in \mathbb{N}$ ,  $x[n..m]$  is the substring consisting of the  $n$ th through  $m$ th symbols of  $x$ , and  $x[n] = x[n..n]$  is the  $n$ th symbol in  $x$ , where  $x[1]$  is the first symbol.

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  $x^k = \underbrace{xx \dots x}_{k \text{ times}}$ .<sup>4</sup> The reverse of  $x$ , where  $|x| = n$ , is the string  $x^R = x[n]x[n-1] \dots x[2]x[1]$ .

Given two strings  $x, y \in \Sigma^*$  for some alphabet  $\Sigma$ ,  $x$  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 *length-lexicographic ordering* (a.k.a. *military ordering*) of strings over an alphabet is the standard dictionary ordering, except that shorter strings precede longer strings.<sup>5</sup> For example, the length-lexicographical ordering of  $\{0, 1\}^*$  is

$\varepsilon, \quad 0, 1, \quad 00, 01, 10, 11, \quad 000, 001, 010, 011, 100, 101, 110, 111, \quad 0000, 0001, \dots$

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

We’ve been speeding through terminology, but it’s worth pausing on these definitions for a moment. We already explained that “language” is an archaic term left over from linguistic theory. But why do we say that a set of strings is also a “decision problem”? A *decision problem is a computational problem with a yes/no answer*. In other words, it’s the kind of problem you are solving when you write a programming language function with a *Boolean return type*. Given some input to the function, the function answers “yes” or “no”. Is a given integer prime or not? Is a given string a properly formatted HTML file? Is a given list of integers  $x, y, z, n$  a counter-example  $x^n + y^n = z^n$  to Fermat’s Last Theorem? These are all decision problems you can solve by writing an appropriate function in your favorite programming language.

But what’s the input to the function? Depending on the programming language, it could be many arguments, and they could all be different types such as integers, floating point numbers, lists of strings, etc. However, every one of these objects, somewhere in memory, is just a sequence of bits. In other words, at a lower level of abstraction, every one of those

---

```
empty_set = set()
set_with_epsilon = {""}
```

In Java, this is

```
String epsilon = "";
Set empty_set = new HashSet();
Set<String> set_with_epsilon = new HashSet<String>();
set_with_epsilon.add(epsilon);
```

In C++, this corresponds to (assuming we’ve imported everything from the standard library `std`)

```
string epsilon("");
set empty_set;
set<string> set_with_epsilon;
set_with_epsilon.insert(epsilon);
```

<sup>4</sup>Alternatively, define  $x^k$  inductively as  $x^0 = \varepsilon$  and  $x^k = xx^{k-1}$

<sup>5</sup>It is also common to call this simply the “lexicographical ordering”. However, this term is ambiguous and is also used to mean the standard alphabetical ordering. In length-lexicographical order,  $1 < 00$ , but in alphabetical order,  $00 < 1$ . We use the term length-lexicographical to avoid confusion.

objects is simply a finite binary string, whose bits are interpreted in a special way. And even if there are many arguments, when they are passed into the function as input, through some mechanism, they are combined into a single string (for instance, with delimiters between them to mark the boundaries).<sup>6</sup> So even though it's not as *convenient for programming*, it is more *convenient for mathematical simplicity* to simply say that **the input to every decision problem is a single finite string**. And since there's only two possible answers, **if we know the subset of strings for which the correct answer is "yes", then we also know the correct answer for every string**.

Thus, the set of input strings (i.e., the language) for which the answer is "yes" is the mathematical object defining the decision problem itself. More formally,  $L \subseteq \Sigma^*$  defines the decision problem: on input  $x$ , if  $x \in L$  then output "yes", and if  $x \notin L$  then output "no". For example, the decision problem of determining whether an integer is prime is represented by the subset  $P \subset \{0, 1\}^*$  of binary strings that encode prime integers in binary:

$$P = \left\{ \begin{array}{cccccccc} 10, & 11, & 101, & 111, & 1011, & 1101, & 10001, & \dots \\ \{ & 2, & 3, & 5, & 7, & 11, & 13, & 17, \dots \end{array} \right\}$$

Now, there's other kinds of computational problems worth solving, where the output is not Boolean. **A solution to a search problem, rather than being merely one bit, is a whole string**. For example, given a set of linear equations, find a solution. **An optimization problem is a special kind of search problem where different solutions have different quantitative "values" and we want to minimize or maximize the value**. For example, given a list of airline ticket prices between cities, find the *cheapest* sequence of flights that visits each city once. It seems like a major restriction to study only decision problems and not these more general types of problems. We will talk in more detail about these kinds of problems in Chapter 9, and we will see that in some senses, we can learn a lot about computational problems generally just by restricting attention to decision problems.

We said that a *class* is a set of languages. This term "class" sounds like just another noun. However, 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 sets of strings, or even the dreaded set of sets of sets of strings (they are out there; the arithmetical and polynomial hierarchies are sets of classes).

Why would we want to think about *sets* of decision problems, instead of just one at a time? Generally, we will define some model of computation, such as finite automata, or Turing machines, or polynomial-time Turing machines. These are sets of "types of computing machines" or "programs" in some programming language. Each machine has some notion of giving a Boolean output for every input, so each machine defines some decision problem that it solves. So we consider classes of decision problems when we want to talk about a

---

<sup>6</sup>Technically this would only be true for pass-by-value. If the language is pass-by-reference, the references/pointers would be concatenated into a single string because they are passed into the function by being pushed sequentially onto the call stack. However, we would still think of the input to the function as including the data in memory to which those pointers are pointing, which may not be contiguous. So it would be more accurate to say that we can conceptually think of the input as something that *could* easily be concatenated into a single string, even if it is not.

concept such as *the decision problems solvable by finite automata* (these are also called the class of *regular languages*) or *the decision problems solvable by Turing machines* (these are also called the class of *decidable languages*) or *the decision problems solvable in polynomial time by Turing machines* (this famous class has a name: P).

Given two languages  $A, B \subseteq \Sigma^*$ , let  $AB = \{ ab \mid a \in A \text{ and } b \in B \}$  (also denoted  $A \circ B$ ).<sup>7</sup> Similarly, for all  $n \in \mathbb{N}$ ,  $A^n = \underbrace{AA \dots A}_{n \text{ times}}$ ,<sup>8</sup>  $A^{\leq n} = A^0 \cup A^1 \cup \dots \cup A^n$  and

$$A^{<n} = A^{\leq n} \setminus A^n.$$

Given a language  $A$ , let  $A^* = \bigcup_{n=0}^{\infty} A^n$ .<sup>10</sup> 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

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

end of lecture 1a

## 2.3 Binary numbers

Since algorithms doing integer arithmetic are generally doing it on binary strings that represent the integers, it is worth recalling how binary representation of integers works. **The binary expansion of a natural number  $n \in \mathbb{N}$  is  $s_n = 0$  if  $n = 0$ , and otherwise, letting  $k = \lfloor \log_2 n \rfloor + 1$ , the binary string  $s_n \in \{0, 1\}^k$  such that (recall  $s_n^R$  is the reverse of  $s_n$ )  $n = \sum_{i=1}^k s_n^R[i] \cdot 2^{i-1}$ .** For example, 10011 is the binary expansion of 19, since

$$1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 2 + 1 = 19.$$

Sometimes, when we want to emphasize that a string is supposed to represent a natural number, we will write the base in a subscript, such as  $10011_2$  or  $19_{10}$ .

It is worth knowing how altering the binary string  $s_n$  affects the number  $n$  it represents. Suppose we append a 0 to the end. What happens when we do this in decimal? It multiplies

<sup>7</sup>The set of all strings formed by concatenating one string from  $A$  to one string from  $B$

<sup>8</sup>The set of all strings formed by concatenating  $n$  strings from  $A$ .

<sup>9</sup>Note that there is ambiguity, since  $A^n$  could also mean the set of all  $n$ -tuples of strings from  $A$ , which is a 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.

<sup>10</sup>The set of all strings formed by concatenating 0 or more strings from  $A$ .

the number by 10. It's the same thing here, but it multiplies it by 2. To see why, note how it alters the sum  $n = \sum_{i=1}^k s^{\mathcal{R}}[i] \cdot 2^{i-1}$ . Let's use the example since it's easier to trace through: it changes

$$1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

to

$$1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

In other words, it added a 0 term to the end (which doesn't change  $n$ ) and it multiplied all other terms by 2 (since each exponent of 2 was increased by 1). More succinctly, if  $s_m = s_n 0$  then  $m = 2n$ . What about  $s_m = s_n 1$ ? (i.e, what if we append a 1 to the end?) Then it's just like above, but the final term is 1 instead of 0. So  $m = 2n + 1$ . What if we remove the final bit of  $s_n$  to get the string  $s_m$ ? This divides by 2, dropping the remainder. (In binary the possible remainders are just 0 and 1.)

## Chapter 3

# Deterministic finite automata

We start with a very restricted model of computation, which (like many models we study) **takes a string input and produces a Boolean output, called *(deterministic) finite automata***. This will be our first example of how to formally model computation with rigorously defined mathematical objects such as tuples, sets, and functions. But, it is not merely a toy model for educational purposes; it is among the models most frequently encountered outside of theoretical computer science. String matching algorithms that implement the “find” feature in text editors and displays are based on finite automata. Syntax highlighting engines in text editors are often specified using finite automata. Many common compression algorithms are implemented as finite automata. **A common use is as a *lexer*: the first step of source code compilation that transforms the raw text into “tokens” such as variable identifiers, numeric literals, comments, string literals, etc.**

### 3.1 Intuitive overview of deterministic finite automata (DFA)

A finite automaton is a mathematical model of a idealized device with essentially no “memory” beyond a finite set of states. A finite automaton starts in a certain state, reads one input symbol at a time in order from left to right. A list of rules called *transitions* indicate, given the current state and symbol, what is the next state the device enters. Each state is labeled with a Boolean output, so at any time the automaton is currently reporting a “yes” or “no” output. The output it reports for the whole string is simply the output of the state the automaton reaches after reading the last symbol of the string.

See Figure 3.1 for an example of one way to represent such a finite automaton  $D_1$ , called a *state transition diagram*.

- *states*:  $a, b, c$
- *input alphabet*:  $0, 1$
- *start state*:  $a$

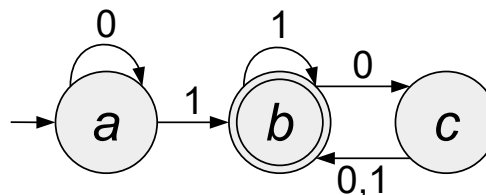


Figure 3.1: A finite automaton  $D_1$ .

- *accept states:  $b$*
- *reject states:  $a, c$*
- *transitions: arrows labeled with input symbols*

$D_1$  *accepts* or *rejects* based on the state it is in after reading the input. If we give the input string 1101 to  $D_1$ , the following happens.

1. Start in state  $a$
2. Read 1, follow transition from  $a$  to  $b$
3. Read 1, follow transition from  $b$  to  $b$
4. Read 0, follow transition from  $b$  to  $c$
5. Read 1, follow transition from  $c$  to  $b$
6. Accept the input because  $D_1$  is in state  $b$  at the end of the input.

$D_1$  also accepts 1, 01, and 010101. In fact all transitions labeled 1 go to  $b$ , so it accepts any string ending in a 1.

Are those *all* the strings it accepts?

$D_1$  also accepts 100, 0100, 110000, and any string that ends with an even number of 0's following the last 1.

## 3.2 Formal models of computation

That example gives some intuition of what we mean by “finite automaton”, but to study them formally, we need a formal mathematical definition, saying what a finite automaton is using only basic discrete mathematical definitions such as integers, strings, sets, sequences, functions, etc. Finite automata are a *model of computation*. In most circumstances, you can take the phrase “model of computation” to be synonymous with *programming language*. Of course, some of them look much different from the programming languages you are used to! But they all involve some way of specifying some *list of rules governing what the program is supposed to do*. So we can say that C++ is a model of computation, and each C++ program is an *instance* of this model.

There are two parts to defining any formal model of computation: *syntax* and *semantics*.

- The *syntax* tells us what an instance of the model *is*.
- The *semantics* tell us what an instance of the model *does*.

For example, I can say that a C++ program is any ASCII string that the gcc compiler can compile with no errors: that's the definition of the *syntax* of C++. <sup>1</sup> But knowing that gcc successfully compiled a program does little to help you understand what will happen when you actually *run the program*. The definition of the semantics of C++ are more involved, telling you what each line of code *does*: what effect does it have on the memory of the computer?

And C++ semantics are complex indeed. For example, `a = b+c;` has the following semantics:

add the integer stored in `b` to the integer stored in `c` and store the result in `a`... unless one of `b` or `c` is a `float` or a `double`, in which case use floating point arithmetic instead, and if the other argument is an `int` it is first converted to a `double`... unless `b` is actually an instance of a class that overloads the `+` operator, in which case call the member function associated with that operator...

And so on. Hopefully this makes it clear why, to learn how to reason formally about computer programs, we start with a much simpler model of computation than C++ or Python, one whose syntax and semantics definitions fit on a page.

### 3.3 Formal definition of a DFA (syntax)

To describe transitions between states, we introduce a *transition function*  $\delta$ . 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 3.3.1.** A (deterministic) finite automaton (DFA) is a 5-tuple  $(Q, \Sigma, \delta, s, F)$ , where

- $Q$  is a non-empty, finite set of states,
- $\Sigma$  is the input alphabet,
- $s \in Q$  is the start state,
- $F \subseteq Q$  is the set of accepting states, and
- $\delta : Q \times \Sigma \rightarrow Q$  is the transition function.

**Example 3.3.2.** The DFA  $D_1$  of Figure 3.1 is defined  $D_1 = (Q, \Sigma, \delta, s, F)$ , where

- $Q = \{a, b, c\}$ ,

---

<sup>1</sup>Of course, that doesn't do much to help you write syntactically valid C++ programs. Another more useful way to define it is that it is any ASCII string that is produced by the C++ grammar (<http://www.nongnu.org/hcb/>). We will cover grammars briefly later in the course, and you will understand better what it means for a string to be produced by a grammar.

- $\Sigma = \{0, 1\}$ ,
- $s = a$ ,
- $F = \{b\}$ , and
- $\delta$  is defined

$$\begin{aligned}
 \delta(a, 0) &= a, \\
 \delta(a, 1) &= b, \\
 \delta(b, 0) &= c, \\
 \delta(b, 1) &= b, \\
 \delta(c, 0) &= b, \\
 \delta(c, 1) &= b,
 \end{aligned}$$

or more succinctly, we represent  $\delta$  by the *transition table*

$\delta$	0	1
$a$	$a$	$b$
$b$	$c$	$b$
$c$	$b$	$b$

The state transition 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.

If  $M = (Q, \Sigma, \delta, s, F)$  is a DFA, how many transitions are in  $\delta$ ; in other words, how many entries are in the transition table?

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

Every DFA decides a language. If it accepts no strings, what language does it decide?  $\emptyset$   
 $D_1$  decides the language

$$L(D_1) = \left\{ w \in \{0, 1\}^* \mid \begin{array}{l} w \text{ contains at least one 1 and an} \\ \text{even number of 0's follow the last 1} \end{array} \right\}$$

Note the terminology here:  $M$  decides a single language  $L(M)$ , but for each string  $x \in \Sigma^*$ ,  $M$  either accepts or rejects  $x$ .  $M$  does *not* decide a string, nor does it accept or reject a language; this misuse of terms mixes up the types.

**Example 3.3.3.** See Figure 3.2.

Formally,  $D_2 = (Q, \Sigma, \delta, s, F)$ , where  $Q = \{a, b\}$ ,  $\Sigma = \{0, 1\}$ ,  $s = a$ ,  $F = \{b\}$ , and  $\delta$  is defined

$\delta$	0	1
$a$	$a$	$b$
$b$	$a$	$b$

What does  $D_2$  do on input 1101?



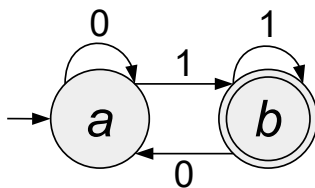


Figure 3.2: A DFA  $D_2$ .  $L(D_2) = \{ w \in \{0,1\}^* \mid w \text{ ends in a } 1 \} = \{1, 01, 11, 001, 011, 101, 111, 0001, \dots\}$

**Example 3.3.4.** Unless we want to talk about the individual states, when specifying a state diagram, it is not necessary to actually give names to the states in the diagram; the start arrow and transition arrows tell us the whole structure of the DFA. Figure 3.3 shows a such

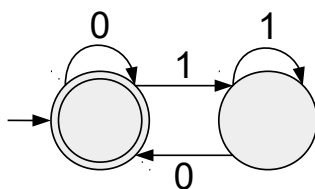


Figure 3.3: A DFA  $D_3$ .  $L(D_3) = \{ w \in \{0,1\}^* \mid w \text{ does not end in a } 1 \}$

a state diagram.

end of lecture 1b

### 3.4 More examples

**Example 3.4.1.** Design a DFA that decides the language

$$\{ a^{3n} \mid n \in \mathbb{N} \} = \{ w \in \{a\}^* \mid |w| \text{ is a multiple of } 3 \} = \{\varepsilon, aaa, aaaaaa, aaaaaaaaa, \dots\}.$$

See Figure 3.5.

**Example 3.4.2.** Design a DFA that decides the language

$$\{ a^{3n+2} \mid n \in \mathbb{N} \} = \{ w \in \{a\}^* \mid |w| \equiv 2 \pmod{3} \} = \{aa, aaaaaa, aaaaaaaaa, \dots\}.$$

Just switch the accept state in Figure 3.5 from 0 to 2.

**Example 3.4.3.** Design a DFA that decides the language

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

For example it should accept 0, 1010, and 11100 and reject 1, 101, and 111.

This is actually  $D_3$  from Figure 3.3.

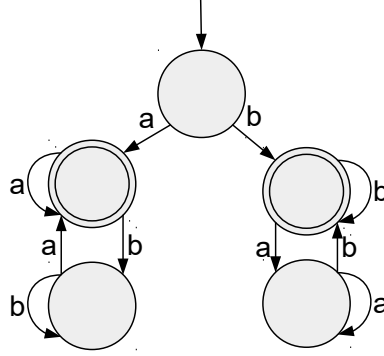


Figure 3.4: A DFA  $D_4$ .  $L(D_4) = \{ w \in \{a, b\}^* \mid |w| > 0 \text{ and } w[1] = w[|w|] \} = \{a, b, aa, bb, aaa, aba, bab, bbb, aaaa, \dots\}$

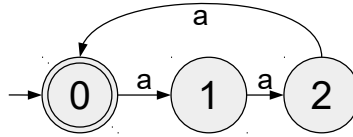


Figure 3.5: A DFA recognizing whether its input string's length is a multiple of 3. Here we give state names that are meaningful: each represents the remainder left when dividing by 3 the number of symbols read so far.

**Example 3.4.4.** Design a DFA that decides the language

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

See Figure 3.6.

Intuitively, we can keep track of whether the bits read so far represent an integer  $n \in \mathbb{N}$  that is of the form  $n = 3k$  for some  $k \in \mathbb{N}$  (i.e.,  $n$  is a multiple of 3, or  $n \equiv 0 \pmod{3}$ ),  $n = 3k + 1$  ( $n \equiv 1 \pmod{3}$ ), or  $n = 3k + 2$  ( $n \equiv 2 \pmod{3}$ ). Appending the bit 0 to the end of  $n$ 's binary expansion multiplies  $n$  by 2, resulting in  $2n$ , whereas appending the bit 1 results in  $2n + 1$ .

For example, consider appending a 1 to a multiple of 3. Since  $n = 3k$ , then  $2n + 1 = 2(3k) + 1 = 3(2k) + 1 \equiv 1 \pmod{3}$ , so the transition function  $\delta$  obeys  $\delta(0, 1) = 1$ . Consider appending a 0 to  $n$  where  $n \equiv 2 \pmod{3}$ . Since  $n = 3k + 2$ , then  $2n = 2(3k + 2) = 6k + 4 = 3(2k + 1) + 1$ , so  $2n \equiv 1 \pmod{3}$ , so  $\delta(2, 0) = 1$ . The other four cases are similar.

### 3.5 Formal definition of computation by a DFA (semantics)

We say  $D = (Q, \Sigma, \delta, s, F)$  *accepts* a string  $x \in \Sigma^n$  if there is a sequence of states  $q_0, q_1, q_2, \dots, q_n \in Q$ , called a *computation sequence of  $D$  on  $x$* , such that

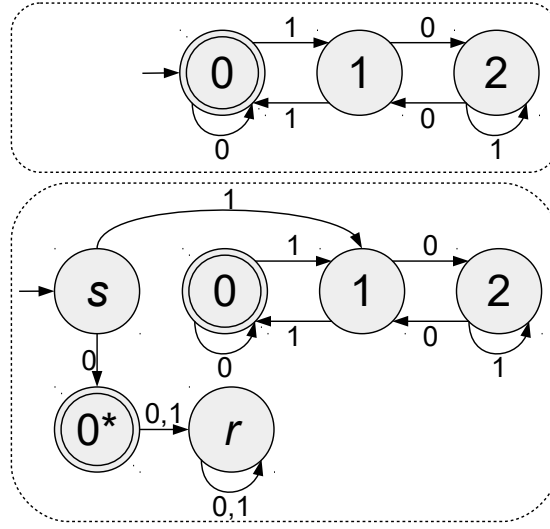


Figure 3.6: Design of a DFA to decide multiples of 3 in binary. If a natural number  $n$  has  $b \in \{0, 1\}$  appended to its binary expansion, then this number is  $2n + b$ . **Top:** a DFA with transitions to implement “when in state  $k$  after reading bits representing  $n$ , if  $n \equiv k \pmod 3$ , then after reading bit  $b$ , so that the bits read so far now represent  $2n + b$ , change to the state  $k'$  such that  $2n + b \equiv k' \pmod 3$ ”. This assumes  $\varepsilon$  represents 0, and that leading 0’s are allowed, e.g., 5 is represented by 101, 0101, 00101, 000101, ... **Bottom:** a modification of top DFA to reject  $\varepsilon$  and any positive integers with leading 0’s.

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

Otherwise,  $D$  rejects  $x$ .

For example, for the DFA in Figure 3.5, for the input  $x = aaaaaa$ , the sequence of states  $(q_0, q_1, q_2, q_3, q_4, q_5, q_6)$  testifying that the DFA accepts  $x$  is  $(0, 1, 2, 0, 1, 2, 0)$ .

Define the *language decided by  $D$*  as  $L(D) = \{ x \in \Sigma^* \mid D \text{ accepts } x \}$ . We say a language  $L$  is *DFA-decidable* if some DFA decides  $L$ .

Using more “computational terminology”, a decision problem is DFA-decidable if it can be solved by an algorithm that uses a constant amount of memory and that runs in exactly  $n$  steps on input strings of length  $n$ .<sup>2</sup>

<sup>2</sup>This is a bit inexact, as I haven’t said what “memory” or “steps” are for general algorithms. But the meaning is clear for DFAs: memory is the set of states and steps correspond to individual transitions. Other reasonable ways of defining space (memory) and time (number of steps) for other models of computation lead to models with equivalent computational power. In fact, the time restriction doesn’t even matter: if you use a constant amount of memory and any amount of time, only regular decision problems can be solved, although proving that is beyond the scope of this course. So for example, if you write a Python function returning a Boolean, and it uses no lists or sets or other unbounded data structures, and if it uses no recursion (a way to use unbounded memory from the function call stack), then the decision problem it is solving is regular, i.e., also solvable by a DFA.

One major goal of this part of the book is to demonstrate that many different models of computation all define exactly the DFA-decidable languages. In fact, the more common term for this class is *regular*. The term “regular language” actually seems a bit inconsistent at this point; it is named for a model studied later called “regular expressions”. But, before we prove that they all define the same class of languages, we need some way to refer to those classes in a way that evokes the model defining the class.

In other words, the *class of regular languages* is the set of all languages decided by some DFA. We will use the term “DFA-decidable” until we actually finish proving that DFA-decidable is the same thing as NFA-decidable, regex-decidable, and RG-decidable, after which we will simply use the term *regular* instead.

end of lecture 1c

## Chapter 4

# Declarative models of computation

Using programming language terminology, DFAs are an *imperative* language, much like C++ or Python: they represent a sequence of instructions telling the machine what to do, based on what input is read. This chapter introduces three models of computation: regular expressions, grammars, and non-deterministic finite automata. In contrast to DFAs, these models are *declarative*: each describes some language, but the model itself gives no indication of instructions for how to “execute” an instance of the model.<sup>1</sup> It’s not obvious how to create an algorithm recognizing a language described by a grammar or regular expression. (Although such algorithms do exist.) Non-deterministic finite automata are closer to DFAs in the sense that they are interpreted to start in some state and transition from state to state based on reading input symbols, but there is a special ability to make non-deterministic choices, and it’s not clear how one would *implement* this ability in a real machine (but we will see how).

### 4.1 Regular expressions

A *regular expression* is a string encoding 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.<sup>2</sup>

In fact, with a small substitution of symbols, the above expression is quite literally an expression of the set of strings it represents, using set notation and string operations we defined in Chapter 2. Simply replace each bit with a singleton set:  $(\{0\} \cup \{1\})\{0\}^*$ , which equals  $\{0, 1\}\{0\}^* = \{0, 1, 00, 10, 000, 100, 0000, 1000, 00000, 10000, \dots\}$

The next regular expression

$(0 \cup 1)^*$

---

<sup>1</sup>In this sense these models are similar to declarative programming languages such as Prolog or SQL. One programs by “declaring” what the computation result should be, without specifying (quite as directly) *how* to compute it.

<sup>2</sup>The notation  $0|10^*$  is more common, but we use the  $\cup$  symbol in place of  $|$  to emphasize the connection to set theory, and because  $|$  looks too much like 1 when hand-written.

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 (or decides, to use consistent terminology with other automata) a language  $L(R)$ .

We now give a formal inductive definition of a regular expression. It has three base cases and three inductive cases.

#### 4.1.1 Formal definition of a regex (syntax and semantics)

**Definition 4.1.1.** Let  $\Delta = \{\cup, (, ), *, \emptyset, \varepsilon\}$  be the alphabet of (*regex*) *control alphabet*. Let  $\Sigma$  be an alphabet such that  $\Sigma \cap \Delta = \emptyset$ , called the *input alphabet*. Let  $\Gamma = \Sigma \cup \Delta$ .

We define regular expressions inductively.  $R \in \Gamma^*$  is a *regular expression (regex)*, deciding language  $L(R) \subseteq \Sigma^*$ , if one of the following cases holds.

1. base cases:

- (a)  $R = b$  for some  $b \in \Sigma$ . Then  $L(R) = \{b\}$ .
- (b)  $R = \varepsilon$ . Then  $L(R) = \{\varepsilon\}$ .
- (c)  $R = \emptyset$ . Then  $L(R) = \{\}$ .

2. inductive cases (let  $R_1, R_2$  be regex's):

- (a)  $R = (R_1) \cup (R_2)$ . Then  $L(R) = L(R_1) \cup L(R_2)$ .
- (b)  $R = (R_1)(R_2)$ . Then  $L(R) = L(R_1) \circ L(R_2)$ .
- (c)  $R = (R_1)^*$ . Then  $L(R) = L(R_1)^*$ .

A language is *regex-decidable* if some regex decides it.

Optionally, we may omit the parentheses in some cases. The operators have precedence  $* > \circ > \cup$ . For example,  $11 \cup 01^*$  is equivalent to  $(1)(1) \cup ((0)(1)^*)$ .<sup>3</sup>

<sup>3</sup>This is similar to why the arithmetic expression  $3 \times 7 + 4 \times 5^6$  is equivalent to  $(3) \times (7) + ((4) \times (5)^6)$ , but not to  $((3 \times 7) + 4) \times (5^6)$  or  $(3 \times ((7 + 4) \times 5))^6$ .

**The cases  $\varepsilon$  and  $\emptyset$ .** Base case (1c)  $R = \emptyset$  is required for the technical reason that, if instead it were omitted, then the easiest decision problem there is, “*just say no*” (i.e., the language  $\emptyset$ ) would not be regex-decidable. However, the case  $R = \emptyset$  is rarely used in practice. The regex simulator (<http://web.cs.ucdavis.edu/~doty/automata/>) has no way to specify it.

Similarly, base case (1b)  $R = \varepsilon$  is rarely used in practice, and there is no syntax in the regex simulator for specifying it explicitly. However, for many of the proofs we do in this course, it will be convenient to have  $\varepsilon$  as its own special case.

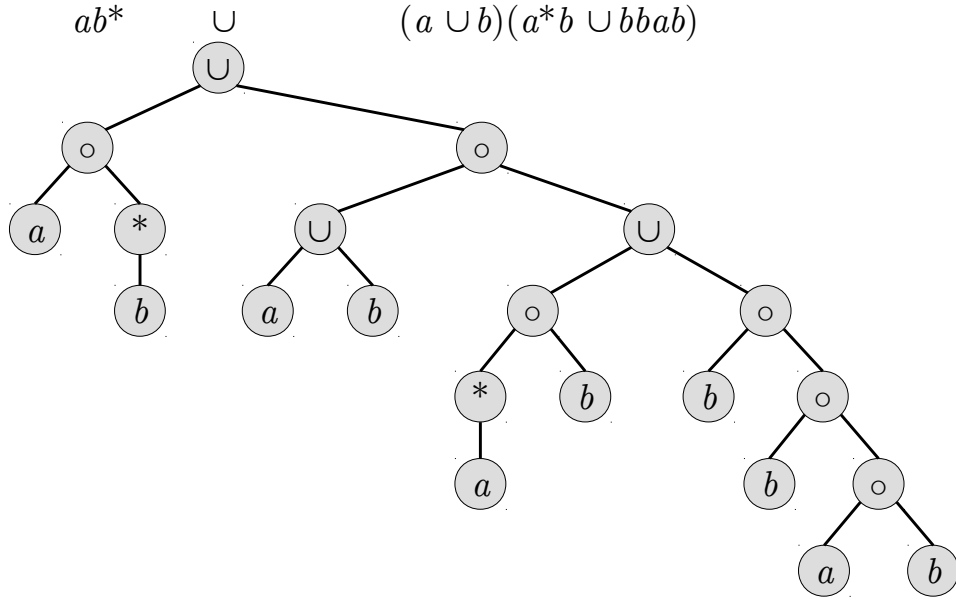


Figure 4.1: Recursive structure of the regular expression  $ab^* \cup (a \cup b)(a^*b \cup bbab)$  viewed as a tree.

Although a regex is merely a string, a more structured way to think of a regex, based on the inductive definition, is as having a tree structure, similar to the parse tree for an arithmetic expression such as  $(7 + 3) \cdot 9 + 11 \cdot 5$ . The base cases are leaves, and the recursive cases are internal nodes, with one child in the case of the unary operation  $*$  and two children in the case of the binary operations  $\circ$  and  $\cup$ . See Figure 4.1 for an example.

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

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

**Example 4.1.2.** Let  $\Sigma$  be an alphabet.

- $0^*10^* = \{ w \mid w \text{ contains a single } 1 \}$ .
- $\Sigma^*1\Sigma^* = \{ w \mid w \text{ has at least one } 1 \}$ .

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

skip in lecture

$$\emptyset^* = \{\varepsilon\}.$$
<sup>4</sup>

There is an algebra of regular expression operations that can be helpful to thinking about when manipulating or simplifying them. One can think of  $\cup$  analogously to addition and  $\circ$  analogously to multiplication. The distributive law holds:  $A(B\cup C)$  is equivalent to  $AB\cup AC$ .<sup>5</sup> In other words, having “a string matching  $A$ , followed by one matching either  $B$  or  $C$ ” is equivalent to having “either a string matching  $A$  followed by one matching  $B$ , or a string matching  $A$  followed by one matching  $C$ ”. One can think of  $\emptyset$  as the additive identity (which is why  $R\cup \emptyset = R$  and  $R\emptyset = \emptyset$ , analogously to  $x+0=x$  and  $x\cdot 0=0$ ), and one can think of  $\varepsilon$  as the multiplicative identity (which is why  $R\varepsilon = R$ , analogously to  $x\cdot 1=x$ ).

**Example 4.1.3.** Design a regex to match C++ `double` literals. For example, the following are valid:

2.0, 3.14, .02, 3., 0.02, -.02, +4.5), 0.00

The following are not valid:

., +, +-2.0, 00.0, 2 (that’s an `int`, not a `double`)

Let  $P = 1\cup \dots \cup 9$  be a regex deciding a single positive decimal digit. Let  $D = 0\cup P$ .

$$(+\cup -\cup \varepsilon)(.D^+\cup (PD^*\cup 0).D^*)$$

<sup>4</sup>The identity  $\emptyset^* = \{\varepsilon\}$  is more a convention than something that can be derived from the definitions. This is similar to the convention for  $0^0$  (the number zero raised to the power zero). Is  $0^0 = 0$ , because 0 times any number is 0? Or is  $0^0 = 1$ , because anything raised to the power 0 is 1? It’s not particularly well defined, so we just say  $0^0 = 1$  by convention. Similarly, it’s not clear whether  $\emptyset^* = \emptyset$  because  $\emptyset$  concatenated to anything is  $\emptyset$ , or whether  $\emptyset^*$  contains  $\varepsilon$  because  $\varepsilon \in A^*$  for all  $A$ . Like in the case of  $0^0$ , we just choose a convention and say that  $\emptyset^* = \{\varepsilon\}$ . In practice, this doesn’t come up much.

<sup>5</sup>As with addition and multiplication, the distributive law fails if we swap the operations:  $A\cup BC$  is *not* equivalent to  $(A\cup B)(A\cup C)$ .



end of lecture 2a

## 4.2 Context-free grammars

Here is an example of a grammar:

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

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 (instead of a multiple-symbol string) 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.

The idea is that we start with a single copy of the start variable. We pick a rule with the start variable and replace the variable with the string on the right side of the rule. This gives a string mixing variables and terminals. We pick some variable in it, pick a rule with that variable on the left side, and again replace the variable with the right side of the rule. Do this until the string is all terminal symbols.

**Example 4.2.1.** Write a grammar generating the language  $\{0^n 1^n \mid n \in \mathbb{N}\}$ .

$$S \rightarrow 0S1 \mid \varepsilon$$

**Example 4.2.2.** Write a grammar generating the language of properly nested parentheses.

$$S \rightarrow (S) \mid SS \mid \varepsilon$$

In theoretical computer science, “context-free grammar” is the common term, whereas in programming languages, “grammar” is more common. The adjective “context-free” distinguishes them from “context-sensitive” grammars, in which the right side of a production rule can have multiple symbols, such as  $AB \rightarrow Axy$ , meaning “replace  $B$  with  $xy$ , but only if there is an  $A$  to the left of  $B$ ”. We won’t study context-sensitive grammars, but it is worth noting that they have greater computational power than context-free grammars.

### 4.2.1 Formal definition of a CFG (syntax)

**Definition 4.2.3.** A *context-free grammar (CFG)* is a 4-tuple  $(\Gamma, \Sigma, S, \rho)$ , where

- $\Gamma$  is a finite alphabet of *variables*,

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

By convention, if only the rules are listed, the variable on the left-hand side of the first rule is the start symbol. Variables are often uppercase letters, but not necessarily: the variables are whatever appears on the left-hand side of rules. It is common in grammars for defining programming languages for the variables to have long names to help readability, such as `AssignmentOperator` or `while_statement`. Terminals are symbols other than uppercase letters, such as lowercase letters, numbers, and other symbols such as parentheses.

#### 4.2.2 Formal definition of computation by a CFG (semantics)

A string in  $(\Sigma \cup \Gamma)^*$  is called an *derived string*. Let  $x, y, z \in (\Sigma \cup \Gamma)^*$  and  $A \in \Gamma$  and let  $A \rightarrow z$  be a production rule. Then we write  $xAy \Rightarrow xzy$  to denote that the rule  $A \rightarrow z$ , applied to derived string  $xAy$ , results in another derived string  $xzy$ . A sequence of derived strings  $x_1, \dots, x_k$ , where  $x_1 = S$  and  $x_k \in \Sigma^*$ , where  $x_i \Rightarrow x_{i+1}$  for  $1 \leq i < k$ , is called a *derivation*. For example, a derivation of `000:111` is

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

We also say that the grammar *accepts* or *produces* the string `000:111`.

For the balanced parentheses grammar  $S \rightarrow (S) \mid SS \mid \varepsilon$ , a derivation of `((()((())))` is

$$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (() (S)) \Rightarrow (() ((S))) \Rightarrow (() ((())))$$

We also may represent this derivation with a *parse tree*, shown in Figure 4.2.

The set of strings accepted by a CFG  $G$  is denoted  $L(G)$ , and we say  $G$  *decides* (a.k.a., *generates*)  $L(G)$ .

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 called *context-free* or *CFG-decidable*.<sup>6</sup>

**Example 4.2.4.** This is a small portion of the grammar for the Python programming language (<https://docs.python.org/3/reference/grammar.html>). The rules use a `:` instead of  $\rightarrow$ , and they allow some syntactic sugar such as regex operators, but the grammar could in principle be specified using only the syntax as defined above.

```
compound_stmt: if_stmt | while_stmt | ...
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
test: or_test ['if' or_test 'else' test] | lambdadef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
```

<sup>6</sup>The common term is “context-free”. We use the phrase “CFG-decidable” to be consistent with our convention of naming language classes after a model of computation defining them.



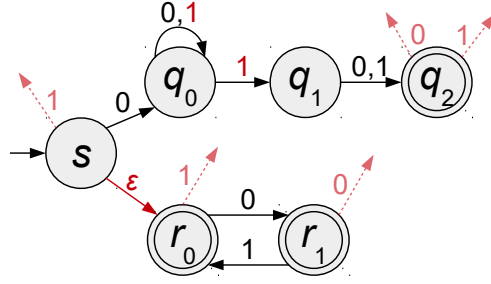


Figure 4.3: A nondeterministic finite automaton (NFA) called  $N_1$ . Differences with a DFA are highlighted in red: 1) There are *two* 1-transitions leaving state  $q_0$ , 2) There is an “ $\varepsilon$ -transition” leaving state  $s$ , 3) There is no 1-transition leaving states  $s, r_0$ , or  $q_2$  and no 0-transition leaving states  $r_1$  or  $q_2$ . Normally “missing” transitions are not shown in NFAs, but we highlight them here to emphasize the difference with a DFA.

**Example 4.3.1.** Design an NFA to decide the language  $\{x \in \{0, 1\}^* \mid x[|x| - 2] = 0\}$ .

See Figure 4.4. This uses nondeterministic transitions (but no  $\varepsilon$ -transitions) to “guess” when the string is 3 bits from the end.

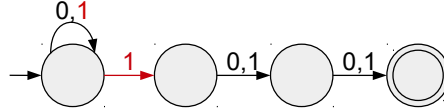


Figure 4.4: An NFA  $N_2$  deciding the language  $\{x \in \{0, 1\}^* \mid x[|x| - 2] = 1\}$ .

#### 4.3.1 Formal definition of an NFA (syntax)

For any alphabet  $\Sigma$ , let  $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ .

**Definition 4.3.2.** A *nondeterministic finite automaton (NFA)* is a 5-tuple  $(Q, \Sigma, \Delta, s, F)$ , where

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

When defining  $\Delta$ , we assume that if for some  $q \in Q$  and  $b \in \Sigma_\varepsilon$ ,  $\Delta(q, b)$  is not explicitly defined, then  $\Delta(q, b) = \emptyset$ .

**Example 4.3.3.** Recall the NFA  $N_1$  in Figure 4.3. Formally,  $N_1 = (Q, \Sigma, \Delta, s, F)$ , where

- $Q = \{s, r_0, r_1, q_0, q_1, q_2\}$ ,
- $\Sigma = \{0, 1\}$ ,
- $F = \{q_2, r_0, r_1\}$ , and
- $\Delta$  is defined

$\Delta$	0	1	$\varepsilon$
$s$	$\{q_0\}$	$\emptyset$	$\{r_0\}$
$r_0$	$\{r_1\}$	$\emptyset$	$\emptyset$
$r_1$	$\emptyset$	$\{r_0\}$	$\emptyset$
$q_0$	$\{q_0\}$	$\{q_0, q_1\}$	$\emptyset$
$q_1$	$\{q_2\}$	$\{q_2\}$	$\emptyset$
$q_2$	$\emptyset$	$\emptyset$	$\emptyset$

A DFA, after reading a string, has exactly one state it can be in. An NFA, since it has many possible paths, could be in many states (or none).

What are all the states  $N_1$  could be in after reading the following strings? (Accepting states underlined.)

$\varepsilon$ :	$s, \underline{r_0}$	10:	$\emptyset$
0:	$q_0, \underline{r_1}$	11:	$\emptyset$
1:	$\emptyset$	000:	$q_0$
00:	$q_0$	010:	$q_0, \underline{q_2}, \underline{r_1}$
01:	$q_0, q_1, \underline{r_0}$	011:	$q_0, q_1, \underline{q_2}$

$N_1$  accepts a string if *any* of states it could be in after reading the string are accepting; it's okay for some of them to be rejecting as long as at least one is accepting. Thus,  $N_1$  accepts  $\varepsilon, 0, 01, 010, 011$  and rejects  $1, 00, 10, 11, 000$ .

**Transition function versus set of transitions.** Sometimes it is easier with NFAs to talk about *sets of transitions*, where each transition is a triple  $(q_f, a, q_t)$ , where  $q_f \in Q$  is the *from-state*,  $q_t \in Q$  is the *to-state*, and  $a \in \Sigma_\varepsilon$  is the *transition label*. We say this is an *a-transition from  $q_f$  to  $q_t$* , and we write  $q_f \xrightarrow{a} q_t$  to denote the transition.

In the transition table above, the from-states are listed on the left column, the transition labels are on the top row, and the to-states are in the table entries. For example,  $N_1$  has transitions  $s \xrightarrow{0} q_0$ ,  $s \xrightarrow{\varepsilon} r_0$ ,  $r_0 \xrightarrow{0} r_1$ ,  $r_1 \xrightarrow{1} r_0$ ,  $q_0 \xrightarrow{0} q_0$ ,  $q_0 \xrightarrow{1} q_0$ ,  $q_0 \xrightarrow{1} q_1$ ,  $q_1 \xrightarrow{0} q_2$ , and  $q_1 \xrightarrow{1} q_2$ . When doing “constructions” in Chapter 5 (algorithms that process an input automaton to produce an output automaton), we will use the language of sets of transitions to talk about adding or removing transitions from the input to produce the output. This is a shorthand for adding or removing to-states from the entries of the table above.

### 4.3.2 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  $y_1, y_2, \dots, y_m \in \Sigma_\varepsilon$  and  $q_0, q_1, \dots, q_m \in Q$  such that

1.  $x = y_1 y_2 \dots y_m$ ,
2.  $q_0 = s$ ,
3.  $q_{i+1} \in \Delta(q_i, y_{i+1})$  for  $i \in \{0, 1, \dots, m-1\}$ , and
4.  $q_m \in F$ .

The two sequences together uniquely identify which transition arrows were followed. We refer to either the sequence of transition arrows, or sometimes the sequence of states followed, as a *computation sequence of  $N$  on  $x$* .<sup>9</sup>

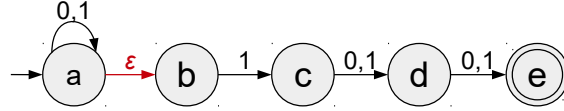


Figure 4.5: An NFA  $N_3$  deciding the language  $\{x \in \{0,1\}^* \mid x[|x| - 2] = 1\}$ .

**Example 4.3.4.** Figure 4.5 shows another NFA deciding if the third-to-last bit is 1, but using an  $\varepsilon$ -transition. This example shows that unlike a DFA, the sequence of states  $q_0, q_1, \dots, q_m$  in the definition of NFA acceptance of a string of length  $n$  can be longer than  $n + 1$ . For example, the string 0100 has  $y_1 = 0, y_2 = \varepsilon, y_3 = 1, y_4 = 0, y_5 = 0$  and  $q_0 = a, q_1 = a, q_2 = b, q_3 = c, q_4 = d, q_5 = e$ .

Note that if  $N$  is actually deterministic (i.e., no  $\varepsilon$ -transitions and  $|\Delta(q, b)| = 1$  for all  $q \in Q$  and  $b \in \Sigma$ ), then the sequence of states leading to an accept state on a given input is *unique*, and it is the same sequence of states as in the definition of DFA acceptance, with  $n + 1$  states to accept a string of length  $n$ .

Compared to DFAs, with NFAs, strings appear

- easier to accept, but
- harder to reject.<sup>10</sup>

<sup>9</sup>Unlike a DFA, even if we know  $x$  and  $q_0, \dots, q_m$ , we cannot uniquely identify which transitions were followed, since it may be possible to take  $\varepsilon$ -transitions at different points while reading  $x$ , yet follow the same sequence of states. (Example?)

<sup>10</sup>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  $\varepsilon$ -transitions everywhere and claim that there exist paths doing what you want.

By the same token that makes acceptance easier, rejection becomes more difficult, because you must ensure that, if the string ought to be rejected, then *no* path leads to an accept state. Therefore, the more transitions and accept

We won't spend time practicing NFA design right now as we did with DFAs. The utility of the non-determinism “feature” of NFAs will become more apparent in Chapter [5](#).

end of lecture 2c

---

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 more difficult to verify a “for all” claim than a “there exists” claim.





## Chapter 5

# Closure of language classes under set operations

So far we have read and “programmed” DFAs, CFGs, regex’s, and NFAs that decide particular languages. We now study fundamental properties shared by *all* languages decided by these models. This will involve thinking at a higher level of abstraction than we’ve done so far. Rather than starting with a concrete problem we want to solve, and designing, for example, a DFA for it, we start with an abstract DFA  $D$  given to us by someone else. We don’t know what  $D$  looks like, other than it is a valid DFA deciding some language (we don’t know the language, but we know it’s called  $L(D)$ ). We have to change it into a second DFA  $D'$ , for instance deciding the complement of  $L(D)$ .

This might be awkward to think about, but all we are doing is describing an algorithm that operates on DFAs. Just as with a sorting algorithm that sorts lists, you can’t make any assumptions about the list. It could be any list, with any number of elements, in any order, and the algorithm has to work on all of them. That’s the same idea we will apply in this chapter: we will describe algorithms that take an automaton as input and produce another automaton as output. We sometimes call these algorithms “constructions” because the output of the algorithm is an automaton like a DFA or a regex that has been constructed. We’re also not going to implement the algorithms in code, although you could. In fact, the autograders implement all of them in code!

### 5.1 Automatic transformation of regex’s, NFAs, DFAs

Let’s start with a simple idea. Suppose we have a DFA  $D$ , and we swap the accept and reject states to get DFA  $D'$ . How are  $L(D)$  and  $L(D')$  related? They behave exactly the same on every input string, except that whenever  $D$  is accepting,  $D'$  is rejecting, and vice versa.  $D$  and  $D'$  give the opposite answer on every single input. Therefore  $L(D') = \overline{L(D)}$ .

This idea works for every single DFA. You take any DFA, defining a DFA-decidable language  $L$ , swap the accept/reject states, and now you have a DFA deciding  $\overline{L}$ . In other words, if a language  $L$  is DFA-decidable, then  $\overline{L}$  is also DFA-decidable. There’s a word for

this concept: the DFA-decidable languages are *closed under complement*.<sup>1</sup>

**Observation 5.1.1.** *The class of DFA-decidable languages is closed under complement.*

At this point you may understand the meaning of the above observation, but not understand why it's worth thinking about. So what if the DFA-decidable languages are closed under complement? Who cares? If we think about it from the point of view of solving problems, understanding these closure properties is often the key to breaking the problem down and solving it.

The problem you are trying to solve, i.e., the language  $L$  you are trying to decide, may not be obviously decidable by any DFA. But if you can spot that  $L$  is simply the complement of another DFA-decidable language, then you know immediately that  $L$  is DFA-decidable. It goes the other way too: in Chapter 7 we will show that certain languages are *not* DFA-decidable. The techniques for proving such *impossibility results* are quite different from showing that a language is DFA-decidable. So you may be asked to show  $L$  is not DFA-decidable, and it's not obvious how to do it. But, if you have already shown that  $\bar{L}$  is not DFA-decidable, then you can immediately conclude that  $L$  cannot be DFA-decidable either, because then otherwise  $\bar{L}$  would be also.

Let's think about the other common set-theoretic operations we have for languages: union, intersection (which are applicable for any kind of set, including languages), concatenation, and Kleene star (which only are applicable to languages). Let  $A$  and  $B$  be languages.

**Union:**  $A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$

**Intersection:**  $A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$

**Concatenation:**  $AB = \{ xy \mid x \in A \text{ and } y \in B \}$ <sup>2</sup> (also written  $A \circ B$ )

**(Kleene) Star:**  $A^* = \bigcup_{n=0}^{\infty} A^n = \{ x_1 x_2 \dots x_k \mid k \in \mathbb{N} \text{ and } x_1, x_2, \dots, x_k \in A \}$ <sup>3</sup>

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

Are the DFA-decidable languages closed under any of these operations? It's not obvious. It turns out they are, but proving this will occupy this chapter and the next. When we are done, we will know not only that DFA-decidable languages are closed under all of these

---

<sup>1</sup>Note that this is **not the same** as the property of a subset  $A \subseteq \mathbb{R}^d$  being closed in the sense studied in a course on real analysis, in that it contains all of its limit points. In particular, one typically does not talk about any class of languages being merely “closed,” but rather, closed *with respect to a certain set operation* such as complement, union, concatenation, or Kleene star. The usage of the word “closed” in real analysis can be seen as a special case of being closed with respect to the operation of taking limits of infinite sequences from the set. The intuition for the terminology is that if you think of the class of DFA-decidable languages as being like a room, and doing an operation as being like moving from one language (or languages) to another, then moving via complement won't get you out of the room; the room is “closed” with respect to that type of move.

<sup>2</sup>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

<sup>3</sup>The set of all strings formed by concatenating 0 or more strings from  $A$ . Just like with  $\Sigma^*$ , except now whole strings may be concatenated instead of individual symbols.

operations, but that so are languages decided by a regex, NFA, or RRG, because all of these models define exactly the same class of languages.

Let's think about other models of computation besides DFAs. Is it obvious that NFA-decidable languages are closed under complement? No it isn't. You might be tempted to swap the accept and reject states, but that won't work! (Homework exercise.)

What about regex-decidability? Recall in the section on regex's, we wrote a regex matching strings that represent C++ `double` literals (for  $P = 1 \cup 2 \cup \dots \cup 9$  and  $D = 0 \cup P$ ):

$$(+ \cup - \cup \varepsilon)(.D^+ \cup (PD^* \cup 0).D^*)$$

Now consider this task: write a regex matching the *complement* of this language, i.e., letting  $D = \{0, 1, \dots, 9\}$ , all strings over  $D \cup \{+, -, .\}$  that are *not* C++ `double` literals.

It's not obvious how to do it, right? **It is true, in fact: the complement of any regex-decidable language is also regex-decidable.** But it's not clear why, right now. For some models of computation it is not even true. For example, **the CFG-decidable languages are not closed under complement.**<sup>4</sup>

Or, consider these two regex's over alphabet  $\Sigma = \{0, 1\}$ :  $R_1 = \Sigma(\Sigma\Sigma)^*$ , matching any odd-length binary string, and  $R_2 = \Sigma^*0011\Sigma^*$ , matching any string containing the substring 0011. The regex  $R_1 \cup R_2$  matches their union: any string that is odd length *or* contains the substring 0011. What about their *intersection*, odd length strings that contain the substring 0011?

We can consider that wherever 0011 appears, since it is even length, for the whole string to be odd length, we must either have an even number of bits before 0011 and an odd number of bits after, or vice versa. Recall that  $(\Sigma\Sigma)^*$  decides even-length strings and  $\Sigma(\Sigma\Sigma)^*$  decides odd-length strings. Thus, this regex works:

$$R_3 = \begin{array}{l} (\Sigma\Sigma)^* 0011 \Sigma(\Sigma\Sigma)^* \\ \cup \Sigma(\Sigma\Sigma)^* 0011 (\Sigma\Sigma)^* \end{array}$$

But this is ad-hoc, requiring us to think about the details of the two languages  $L(R_1)$  and  $L(R_2)$ . Consider changing to  $R_1 = \Sigma\Sigma\Sigma(\Sigma\Sigma\Sigma\Sigma\Sigma)^*$  matching all strings whose length is congruent to 3 mod 5. We could do a similar case analysis of all the combinations of numbers of bits before and after 0011 that would lead the whole length to be congruent to 3 mod 5. But this would be tedious, and it similarly requires us to understand the details of the two languages.

Could we devise a procedure to *automatically* process  $R_1$  and  $R_2$ , based solely on their structure, not requiring any special reasoning specific to their languages, which would tell us how to construct  $R_3$  with  $L(R_3) = L(R_1) \cap L(R_2)$ ?

Similarly, can we devise a way to automatically convert a regex  $R$  into one deciding the complement  $\overline{L(R)}$ ? This task is easy enough with DFAs, but it's not clear for regex's.

Conversely, it is not obvious how to show the DFA-decidable languages are closed under  $\cup$ ,  $\cap$ ,  $\circ$  or  $*$ .

---

<sup>4</sup>The language  $\{0^i 1^j 2^k \mid i \neq j \text{ or } j \neq k\}$  is CFG-decidable, but its complement  $\{0^i 1^i 2^i \mid i \in \mathbb{N}\}$  is not.

So to recap: a DFA can easily be modified to decide the complement. Regex's can trivially be combined for certain operations:  $\cup$ ,  $\circ$ ,  $*$ , because those operations are built right into the definition. NFAs can't even seem to be modified easily to decide the complement. Some closure properties that are easy to prove for one model seem difficult in the other.

But we made the claim that in fact all of these models define the exact same set of decision problems: the regular languages, which are claimed to be closed under *all* of these operations. Thus, if there are DFAs or regex's or NFAs or RRGs for languages  $A, B$ , then there are DFAs and regex's and NFA's and RRG's for *all* of  $A, B, A \cup B, A \cap B, A \circ B, A^*$ , and  $\bar{A}$ .

**Roadmap for next two chapters.** In this chapter, we will show that we can prove some of these closure properties by giving constructions (algorithms) to transform one or two instances of a model into another instance of that same model, deciding a (potentially) different language. For instance, the next section shows how to combine two DFAs  $D_1$  and  $D_2$  into a third DFA  $D$  such that  $L(D) = L(D_1) \cup L(D_2)$ . These constructions are more involved than the very simple “swap the accept/reject” states construction to show DFA-decidability is closed under complement, but they follow the same principle: the constructed automaton “simulates” the given automata in some way. At the end of this chapter, we will have shown that the DFA-decidable languages are closed under complement,  $\cup$ , and  $\cap$ , the NFA-decidable languages are closed under  $\cup$ ,  $\circ$ , and  $*$ , and the regex-decidable languages are closed under  $\cup$ ,  $\circ$ , and  $*$ . The last part about regex's follows directly by the definition of regex's, so we won't actually discuss it again in this chapter.

In Chapter 6, we do something similar, but instead, show constructions that transform an instance of a model into an instance of a different model, deciding the same language. For instance, we will show how to take an arbitrary NFA  $N$ , which decides language  $L(N)$  by definition, and produce a DFA also deciding  $L(N)$ . This will be done between all four models of DFA, NFA, regex, and RRG, showing that all of these models have the same computational power: if one of them decides a language, so do all the others. At that point it will make sense to call the class of languages they decide simply “regular”, without referring specifically to one of those models.

It will follow that, for example, the DFA-decidable languages are closed under  $*$ , although the construction is somewhat indirect:

1. Start with a DFA  $D_1$  deciding language  $A$ . Note that  $D_1$  is also an NFA.
2. Convert  $D_1$  to an NFA  $N$  deciding  $A^*$ .
3. Convert  $N$  to an equivalent DFA  $D_2$  deciding  $A^*$ .

## 5.2 DFA union and intersection (product construction)

**Example 5.2.1.** Let  $L_1 = \{a^n \mid n \equiv 2 \pmod{3}\}$  and  $L_2 = \{a^n \mid n \equiv 1 \pmod{5} \text{ or } n \equiv 4 \pmod{5}\}$ . Design a DFA  $D = (Q, \{a\}, \delta, s, F)$  to decide the language  $L_1 \cup L_2$ .

See Figure 5.1. Informally, we know how to create a DFA  $D_1$  for  $L_1$  and a DFA  $D_2$  for  $L_2$ . To design  $D$ , we want to *simulate* the actions of both  $D_1$  and  $D_2$ , simultaneously, on the input. This is done by giving each state of  $D$  two “fields”, one field to track the state of  $D_1$ , and the second to track the state of  $D_2$ . Each field is updated independently, depending only on the input symbol and the previous value of that field, but not depending on the other field. At any time (including after reading all the input symbols),  $D$  knows both states that  $D_1$  and  $D_2$  would be in, so it knows whether each is accepting. It should accept if either (or both) are accepting.

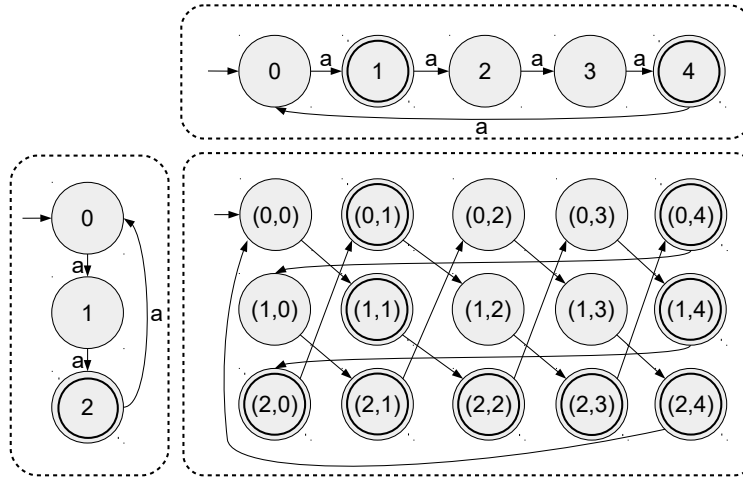


Figure 5.1: **Bottom left:** a DFA deciding the language  $L_3 = \{a^n \mid n \equiv 2 \pmod 3\}$ . **Top right:** a DFA deciding the language  $L_5 = \{a^n \mid n \equiv 1 \pmod 5 \text{ or } n \equiv 4 \pmod 5\}$ . **Bottom right:** a DFA deciding the language  $L_3 \cup L_5$ . For readability, transitions are not labeled, but they should all be labeled with  $a$ . The accept states are row 2 and columns 1 and 4. **To decide  $L_3 \cap L_5$ , we choose the accept states to be  $\{(2, 1), (2, 4)\}$  instead, i.e., where row 2 meets columns 1 and 4.**

Formally,

- $Q = \{ (i, j) \mid 0 \leq i < 3, 0 \leq j < 5 \} = \{0, 1, 2\} \times \{0, 1, 2, 3, 4\}$
- $s = (0, 0)$
- $\delta((i, j), a) = (i + 1 \pmod 3, j + 1 \pmod 5)$
- $F = \{ (i, j) \mid i = 2 \text{ or } j = 1 \text{ or } j = 4 \}$

$D$  is essentially simulating two DFAs at once: one that computes congruence mod 3 and the other that computes congruence mod 5. We call this general idea of one DFA simultaneously simulating two others, by having two parts of its state set, one to remember the state of one DFA and the other to remember the state of the other DFA, the *product construction*, because the larger DFA's state set is the cross product of the smaller two.

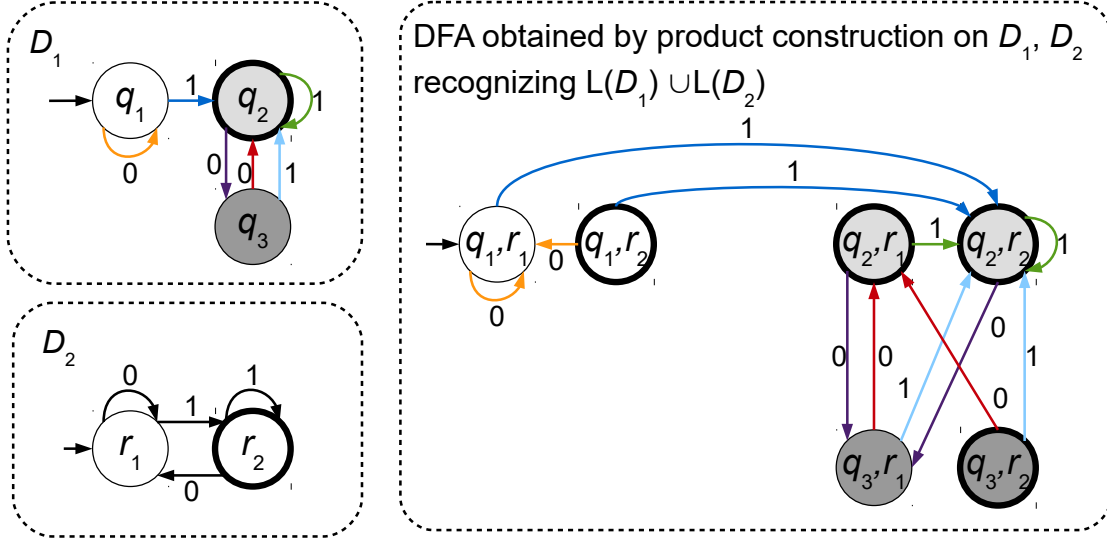


Figure 5.2: The product construction applied to the DFAs  $D_1$  and  $D_2$  from Figures 3.1 and 3.2, to obtain a DFA  $D$  with  $L(D) = L(D_1) \cup L(D_2)$ . We can think of the states of  $D$  as being grouped into three groups representing  $q_1$ ,  $q_2$ , and  $q_3$  respectively, and within each of those groups, there's a state representing  $r_1$  and a state representing  $r_2$ . In this figure, accept states have bold outlines. To decide  $L(D_1) \cap L(D_2)$  instead, we should choose only  $(q_2, r_2)$  to be the accept state.

For an example of the product construction on DFAs with a larger alphabet, as well as a different way to visualize the product construction, see Figure 5.2.

We now generalize this idea.

**Theorem 5.2.2.** *The class of DFA-decidable languages is closed under  $\cup$ .*

5

*Proof.* **(DFA Product Construction for  $\cup$ )**

Let  $D_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$  and  $D_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$  be DFAs. We construct the DFA  $D = (Q, \Sigma, \delta, s, F)$  to decide  $L(D_1) \cup L(D_2)$  by simulating both  $D_1$  and  $D_2$  in parallel, where

- $Q$  keeps track of the states of both  $D_1$  and  $D_2$ :

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

<sup>5</sup>**Proof Idea:** (The DFA Product Construction for  $\cup$ )

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  $D_1$  decides  $A_1$ , and some DFA  $D_2$  decides  $A_2$ . It suffices to show that some DFA  $D$  decides  $A_1 \cup A_2$ ; i.e., it accepts a string  $x$  if and only if at least one of  $D_1$  or  $D_2$  accepts  $x$ .

$D$  will simulate  $D_1$  and  $D_2$ . If either accepts the input string, then so will  $D$ .  $D$  must simulate them simultaneously, because if it tried to simulate  $D_1$ , then  $D_2$ , it could not remember the input to supply it to  $D_2$ .

- $\delta$  simulates moving both  $D_1$  and  $D_2$  one step forward in response to the input symbol. Define  $\delta$  for all  $(r_1, r_2) \in Q$  and all  $b \in \Sigma$  as

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

6

- $s$  ensures both  $D_1$  and  $D_2$  start in their respective start states:

$$s = (s_1, s_2)$$

- $F$  must be accepting exactly when either one, or the other, or both, of  $D_1$  and  $D_2$  are in an accepting state:

$$F = \{ (r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2 \}.$$

7

Then, after reading an input string  $x$  that puts  $D_1$  in state  $r_1$  and  $D_2$  in state  $r_2$ , we have that the state  $q = (r_1, r_2)$  is in  $F$  if and only if  $r_1 \in F_1$  or  $r_2 \in F_2$ , which is true if and only if  $x \in L(D_1)$  or  $x \in L(D_2)$ , i.e.,  $x \in L(D_1) \cup L(D_2)$ .  $\square$

end of lecture 3a

**Theorem 5.2.3.** *The class of DFA-decidable languages is closed under  $\cap$ .*

**Proof Idea:** DeMorgan's Laws.

*Proof.* Let  $A, B$  be DFA-decidable 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 closure under union and complement, this language is also DFA-decidable.  $\square$

*Alternate proof. (DFA Product Construction for  $\cap$ ).* We can modify the product construction for  $\cup$  and define  $F = F_1 \times F_2 = \{(r_1, r_2) \mid r_1 \in F_1 \text{ and } r_2 \in F_2\}$ .  $\square$

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

What about *infinite* union or intersection?

<sup>6</sup>This is difficult to read! Be sure to read it carefully and ensure that the *type* of each object is what you expect.  $Q$  has states that are pairs of states from  $Q_1 \times Q_2$ . Thus,  $\delta$ 's first input, a state from  $Q$ , is a *pair of states*  $(r_1, r_2)$ , where  $r_1 \in Q_1$  and  $r_2 \in Q_2$ . Similarly, the output of  $\delta$  must also be a pair from  $Q_1 \times Q_2$ . However,  $\delta_1$ 's first input, and its output, are just *single* states from  $Q_1$ , and similarly for  $\delta_2$ .

<sup>7</sup>This is *not* the same as  $F = F_1 \times F_2$ . What would the automaton do in that case?



**Closure only goes one way with binary operations.** We now know if  $A$  and  $B$  are DFA-decidable, then  $A \cup B$  is DFA-decidable. What about the reverse claim? Can we say that if  $A \cup B$  is DFA-decidable, then both  $A$  and  $B$  must be DFA-decidable? No: let  $A$  be any DFA-decidable language, and let  $B = \overline{A}$ . Then  $A \cup B = \Sigma^*$ , which is DFA-decidable, but  $A$  is not (neither is  $B$ , otherwise  $A$  would also be by closure under complement).

So now we know that the DFA-decidable languages are closed under complement,  $\cup$ , and  $\cap$ . What about  $\circ$  or  $*$ ? Given two DFAs  $D_1$  and  $D_2$ , it seems difficult to create a DFA  $D$  deciding  $L(D_1) \circ L(D_2)$  using the same ideas as we used in the product construction.

The obvious thing to try is to let  $D$  have all the states of both  $D_1$  and  $D_2$  (i.e., take the union of their states), and on input  $w$ , to start simulating  $D_1$  reading some prefix  $x \sqsubseteq w$ , and then at some point to switch from  $D_1$  to  $D_2$ , and have  $D_2$  process the remaining suffix  $y$  of  $w$ , so that  $w = xy$  and  $D$  accepts if and only if  $D_1$  accepts  $x$  and  $D_2$  accepts  $y$ . But how does  $D$  know when to switch? If  $w$  is the concatenation  $x$  of  $x \in L(D_1)$  and  $y \in L(D_2)$ , there's no delimiter to tell us where  $x$  ends and  $y$  begins. In fact, it might be possible to split  $w$  into  $x$  and  $y$  in multiple ways, for example: if  $L(D_1) = \{0, 00\}$  and  $L(D_2) = \{0, 00\}$ , then  $w = 000$  could be obtained by letting  $x = 0$  and  $y = 00$ , or  $x = 00$  and  $y = 0$ .

## 5.3 NFA union, concatenation, and star constructions

### 5.3.1 Examples

We begin by observing that NFAs can be combined to decide union using a simpler method than the product construction for DFAs. (Unfortunately, unlike the product construction, the NFA union construction only works for union, not intersection.)

**Example 5.3.1.** Design an NFA deciding the language  $\{x \in \{a\}^* \mid |x| \text{ is a multiple of 3 or 5}\}$ .

See Figure 5.3. This uses  $\varepsilon$ -transitions to guess which of the two DFAs to simulate: one that decides multiples of 3, or another that decides multiples of 5.

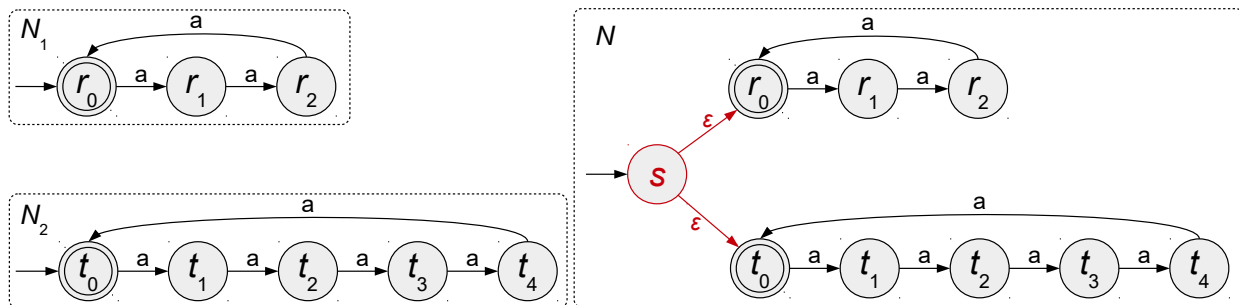


Figure 5.3: Example of the NFA union construction. This NFA decides the language  $L(N) = L(N_1) \cup L(N_2) = \{x \in \{a\}^* \mid |x| \text{ is a multiple of 3 or 5}\}$ , using  $\varepsilon$ -transitions to guess which of the two NFAs to simulate.



Unlike DFAs, there is a simple way to implement concatenation and Kleene star with NFAs as well. We start with an example of concatenation.

**Example 5.3.2.** If  $x \in \{0, 1\}^*$ , let  $n_x \in \mathbb{N}$  be the integer that  $x$  represents in binary (with  $n_\varepsilon = 0$ ).

Design an NFA to decide the language

$$A = \{xy \in \{0, 1\}^* \mid (n_x \equiv 0 \pmod{3} \text{ or } n_x \equiv 1 \pmod{3}) \text{ and } n_y \equiv 2 \pmod{3}\}.$$

For example,  $1000101 \in A$  since  $100_2 = 4 \equiv 1 \pmod{3}$  and  $0101_2 = 5 \equiv 2 \pmod{3}$ . However,  $01 \notin A$  since the possible values for  $x$  and  $y$  are  $(x = \varepsilon, y = 01)$ ,  $(x = 0, y = 1)$ , or  $(x = 01, y = \varepsilon)$ , and in all cases  $n_y \not\equiv 2 \pmod{3}$ .

See Figure 5.4. This simulates the transitions of the DFA from Figure 3.6 *twice*, once on  $x$  and once on  $y$ : it uses  $\varepsilon$ -transitions to guess where  $x$  ends and  $y$  begins (and it only will make that jump if the DFA is currently accepting  $x$ ).

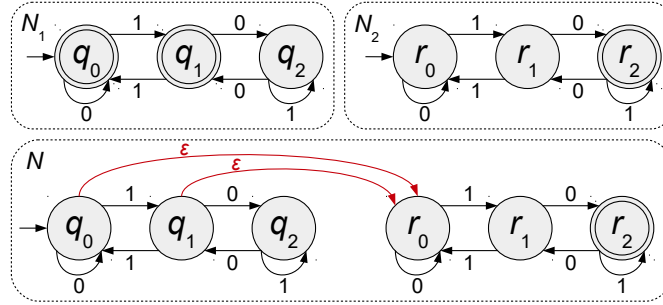


Figure 5.4: Example of the NFA concatenation construction. An NFA  $N$  deciding the language  $\{xy \in \{0, 1\}^* \mid (n_x \equiv 0 \pmod{3} \text{ or } n_x \equiv 1 \pmod{3}) \text{ and } n_y \equiv 2 \pmod{3}\}$ .  $N$  essentially consists of two copies, named  $N_1$  and  $N_2$ , of the DFA from Figure 3.6 (with appropriate accept states) with  $\varepsilon$ -transitions to “guess” when to switch from  $N_1$  to  $N_2$  (but only when  $N_1$  is accepting). Then  $L(N) = L(N_1)L(N_2)$ .

**Example 5.3.3.** Let  $L = \{x \in \{a, b\}^* \mid x \text{ has an odd number of } b\text{'s followed by an } a\}$ . Design an NFA to decide the language  $L^*$ .

$L$  is decided by the NFA  $N$  in Figure 5.5.  $N$  simulates  $D$  over and over in a loop, using  $\varepsilon$ -transitions to guess when to start over (but only starting over if the DFA is currently accepting  $x$ ).

It is important that we did not make the old start state accepting. This changes the semantics of the underlying NFA and could result in a mistake, if positive-length strings reach back to the start state.

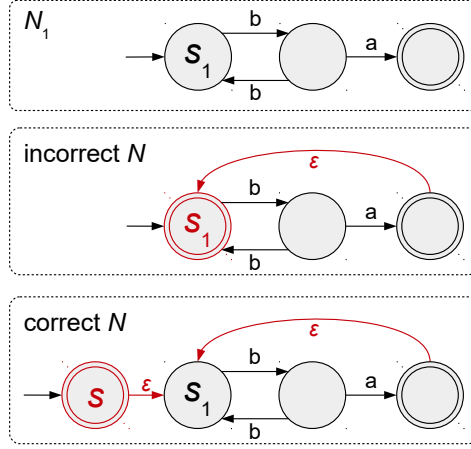


Figure 5.5: Example of the NFA Kleene star construction on an NFA.  $N$  simulates  $N_1$  over and over in a loop, using  $\varepsilon$ -transitions to guess when to start over (but only when  $N_1$  is accepting). The NFA  $N_1$  decides the language described by the regular expression  $\mathbf{b(bb)^*a}$ , i.e., an odd number of  $\mathbf{b}$ 's followed by an  $\mathbf{a}$ .  $L(N) = L(N_1)^*$  is then the language containing  $\varepsilon$  (as any starred language should) and positive-length strings in which all occurrences of an  $\mathbf{a}$  are preceded by an odd number of  $\mathbf{b}$ 's, and the last bit is a  $\mathbf{a}$ . It would be a mistake simply to make the old start state accepting. The incorrect NFA  $N$  shown accepts the strings  $\mathbf{aa}$  and  $\mathbf{aaaa}$ , which do not end in a  $\mathbf{b}$ .

### 5.3.2 Proofs

The above examples show the basic ideas for how to prove that the NFA-decidable languages are closed under union, concatenation, and Kleene star. Now, we actually prove those facts, by showing how to make the ideas work on *any* NFA.

**Theorem 5.3.4.** *The class of NFA-decidable languages is closed under  $\cup$ .*

*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, where  $Q_1 \cap Q_2 = \emptyset$ . Define the NFA  $N$  deciding  $L(N_1) \cup L(N_2)$ .

See Figure 5.3 for an example. Intuitively, on input  $w \in \Sigma^*$ ,  $N$  nondeterministically guesses whether to simulate  $N_1$  or  $N_2$  by taking an  $\varepsilon$ -transition from  $N$ 's start state to the start state for either  $N_1$  or  $N_2$ .  $N$  accepts if the NFA it guessed accepts.

To fully define  $N = (Q, \Sigma, \Delta, s, F)$ :

- $N$  has all the states of  $N_1$  and  $N_2$ , and one extra new state  $s$ , so  $Q = Q_1 \cup Q_2 \cup \{s\}$ , where  $s \notin Q_1 \cup Q_2$ .
- $N$  accepts if the NFA it guessed accepts:  $F = F_1 \cup F_2$ .
- $N$  simulates  $N_1$  or  $N_2$ , depending on the initial guess. To do this, for all  $q \in Q_1 \cup Q_2$  and  $b \in \Sigma_\varepsilon$ ,

$$\Delta(q, b) = \begin{cases} \Delta_1(q, b), & \text{if } q \in Q_1; \\ \Delta_2(q, b), & \text{if } q \in Q_2; \end{cases}$$

Finally,  $N$  must make the initial guess:  $\Delta(s, \varepsilon) = \{s_1, s_2\}$ .

Equivalently, we can use the language of sets of transitions to describe  $\Delta$ :  $\Delta$  has all transitions of  $\Delta_1$  and  $\Delta_2$ , in addition to two new transitions  $s \xrightarrow{\varepsilon} s_1$  and  $s \xrightarrow{\varepsilon} s_2$ .  $\square$

### end of lecture 3b

From now on, we will not define the full transition function  $\Delta$  in NFA constructions such as this. Instead, we will use the equivalent (but easier) terminology of sets of transitions.

**Theorem 5.3.5.** *The class of NFA-decidable 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. Define an NFA  $N$  deciding  $L(N_1) \circ L(N_2) = \{ xy \in \Sigma^* \mid x \in L(N_1) \text{ and } y \in L(N_2) \}$ .

See Figure 5.4 for an example. Intuitively, on input  $w \in \Sigma^*$ ,  $N$  first simulates  $N_1$  for some prefix  $x \sqsubseteq w$ , then  $N_2$  on the remaining suffix  $y$  (so  $w = xy$ ), nondeterministically guessing when to switch, but only when  $N_1$  is accepting.  $N$  accepts if  $N_2$  does, i.e., if  $N_1$  accepts  $x$  and  $N_2$  accepts  $y$ .

To fully define  $N = (Q, \Sigma, \Delta, s, F)$ :

- $N$  has all the states of  $N_1$  and  $N_2$ , so  $Q = Q_1 \cup Q_2$ .
- $N$  starts by simulating  $N_1$ , so  $s = s_1$ .
- $N$  simulates  $N_1$  initially, at some point when  $N_1$  is accepting, guesses where to switch to simulating  $N_2$ . To do this,  $\Delta$  has all transitions of  $\Delta_1$  and  $\Delta_2$ . In addition, for each  $q \in F_1$ ,  $\Delta$  has  $q \xrightarrow{\varepsilon} s_2$ .
- $N$  accepts if  $N_2$  accepts after  $N$  has switched to simulating  $N_2$ , so  $F = F_2$ . (Note that no states in  $F_1$  are accepting in  $N$ .)

**Detailed proof of correctness:** 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  $\varepsilon$ -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  $\varepsilon$ -transition, so  $x$  takes  $N$  from  $s_1$  to a state in  $F_1$ , i.e.,  $x \in L(N_1)$ . Since  $w = xy$ , this shows that  $L(N) \subseteq L(N_1) \circ L(N_2)$ .

Thus  $N$  decides  $L(N_1) \circ L(N_2)$ .  $\square$

**Theorem 5.3.6.** *The class of NFA-decidable languages is closed under  $*$ .*

*Proof.* Let  $N_1 = (Q_1, \Sigma, \Delta_1, s_1, F_1)$  be an NFA. Define an NFA  $N$  deciding  $L(N_1)^*$ .

See Figure 5.5 for an example.  $N$  should accept  $w \in \Sigma^*$  if  $w$  can be broken into several pieces  $x_1, \dots, x_k \in \Sigma^*$  such that  $N_1$  accepts each piece.  $N$  will simulate  $N_1$  repeatedly, nondeterministically guessing when to  $\varepsilon$ -transition from an accept state to the start state, (signifying a switch from  $x_i$  to  $x_{i+1}$ ). Since  $N$  must also accept  $\varepsilon$ , we add one new accepting state to  $N$ , which is  $N$ 's start state, and an  $\varepsilon$ -transition to the original start state.<sup>8</sup>

To fully define  $N = (Q, \Sigma, \Delta, s, F)$ :

- $N$  has the states of  $N_1$  plus the new start state  $s$ , so  $Q = Q_1 \cup \{s\}$ .
- $N$  accepts whenever  $N_1$  does, and to ensure we accept  $\varepsilon$ , the new start state is also accepting:  $F = F_1 \cup \{s\}$ .
- $N$  simulates  $N_1$  repeatedly, guessing when to reset if  $N_1$  when accepting. To do this,  $\Delta$  has all transitions of  $\Delta_1$ , in addition to  $s \xrightarrow{\varepsilon} s_1$  and, for each  $q \in F$ ,  $q \xrightarrow{\varepsilon} s$ .  $\square$

**Detailed proof of correctness:** Clearly  $N$  accepts  $\varepsilon \in L(D)^*$ , so we consider only nonempty strings.

To see that  $L(D)^* \subseteq L(N)$ , suppose  $w \in L(D)^* \setminus \{\varepsilon\}$ . Then  $w = x_1 x_2 \dots x_k$ , where each  $x_j \in L(D) \setminus \{\varepsilon\}$ . 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  $w$ :

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

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  $\varepsilon$ -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.,  $L(D)^* \subseteq L(N)$ .

To see that  $L(N) \subseteq L(D)^*$ , let  $w \in L(N) \setminus \{\varepsilon\}$ . Then there are sequences  $s' = q_0, q_1, \dots, q_m \in Q'$  and  $y_0, \dots, y_{m-1} \in \Sigma_\varepsilon$  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  $w \neq \varepsilon$ ,  $q_m \neq s'$ , so  $q_m \in F$ . Since the start state is  $s' = q_0$ , which has no outgoing non- $\varepsilon$ -transition, the first transition from  $q_0$  to  $q_1$  is the  $\varepsilon$ -transition  $s'$  to  $s$ . Suppose there are  $k-1$   $\varepsilon$ -transitions from a state in  $F$  to  $s$  in

<sup>8</sup>It is tempting simply to make  $N_1$ 's original start state  $s_1$  accepting. However, if  $s_1$  was rejecting before, and if there are existing transitions into  $s_1$ , then this will add new strings to the language accepted by  $N_1$ , which we don't want. Adding a new start state just to handle  $\varepsilon$  solves this problem.

$q_1, \dots, q_m$ . Then we can write  $q_1, \dots, q_m$  as

$$\begin{pmatrix} s_D, q_{1,1}, & \dots, & q_{1,|x_1|}, \\ s_D, q_{2,1}, & \dots, & q_{2,|x_2|}, \\ & \dots & \\ s_D, q_{k,1}, & \dots, & q_{k,|x_k|} \end{pmatrix}.$$

where each  $q_{j,|x_j|} \in F$  and has a  $\varepsilon$ -transition to  $s$ .<sup>9</sup> For each  $j \in \{1, \dots, k\}$  and  $i \in \{0, |x_j| - 1\}$ , let  $y_{j,i}$  be the corresponding symbol in  $\Sigma_\varepsilon$  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  $y_{j,0}y_{j,1} \dots y_{j,|x_j|-1}$  testify that  $D$  accepts  $x_j$ , thus  $x_j \in L(D)$ . Thus  $w = x_1x_2 \dots x_k$ , where each  $x_j \in L(D)$ , so  $w \in L(N)^*$ , showing  $L(N) \subseteq L(D)^*$ .

Thus  $N$  decides  $L(D)^*$ .

end of lecture 3c

---

<sup>9</sup>Perhaps there are no such  $\varepsilon$ -transitions, in which case  $k = 1$ .



## Chapter 6

# Equivalence of models

The previous chapter showed how to convert instances of one model into other instances of the same model, recognizing a different language. In this chapter, we show how to convert an instance of one model into an instance of a *different* model, recognizing the *same* language. This is our primary tool for comparing the power of different models of computation.

For example, if any regex can be simulated by an NFA, then any regex-recognizable language is NFA-recognizable, so NFAs are at least as powerful as regex's. Conversely, if any NFA can be simulated by a regex, then regex's are at least as powerful as NFAs. If both are true, then NFAs and regex's have equivalent computational power. By the end of this chapter, we will have shown that DFAs, NFAs, regex's, and RRGs all have the same computational power: they all recognize the same class of languages, which we call “regular”.

### 6.1 Equivalence of DFAs and NFAs (subset construction)

**Observation 6.1.1.** *If a language is DFA-decidable, then it is NFA-decidable.*

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

The converse is less obvious.

**Theorem 6.1.2.** *If a language is NFA-decidable, then it is DFA-decidable.*

*Proof. (Subset Construction)* Let  $N = (Q_N, \Sigma, \Delta, s_N, F_N)$  be an NFA with no  $\varepsilon$ -transitions.<sup>1</sup> 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.

---

<sup>1</sup>At the end of the proof we explain how to modify the construction to handle them.

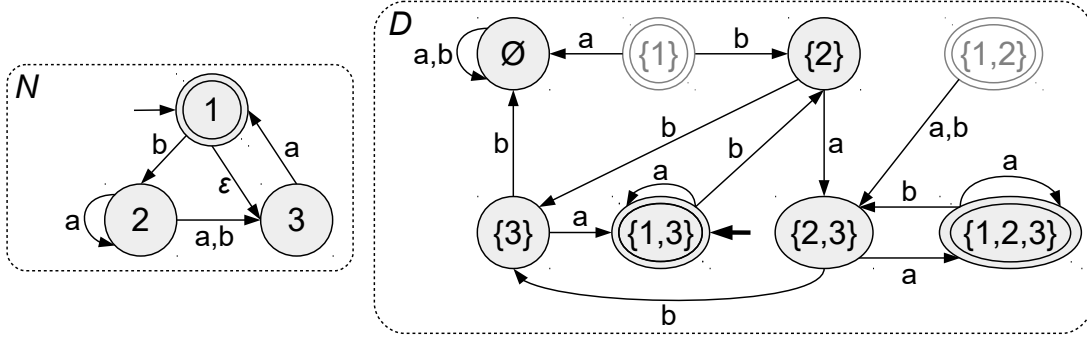


Figure 6.1: Example of the subset construction. We transform an NFA  $N$  into a DFA  $D$ . Each state in  $D$  represents a *subset* of states in  $N$ . The  $D$  states  $\{1\}$  and  $\{1,2\}$  are unreachable from the start state  $\{1,3\}$ , so we could remove them without altering the behavior of  $D$ . However, the official subset construction leaves them in.

- 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  $s_N$ .
- $F_D = \{A \subseteq Q_N \mid A \cap F_N \neq \emptyset\}$ . 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  $\varepsilon$ -transitions. For any  $R \subseteq Q_N$  and define

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

For example, in Figure 6.1, if we let  $R = \{1, 2\}$ , then  $E(R) = \{1, 2, 3\}$ .

An example in which multiple  $\varepsilon$ -transitions can be followed is in Figure 5.5; if we let  $R = \{q_1, q_2\}$ , then  $E(R) = \{q_1, q_2, s_1\}$ .

To account for the  $\varepsilon$ -transitions,  $D$  must be able to simulate

1.  $N$  following  $\varepsilon$ -transitions *after* each input-consuming transition, i.e., define

$$\delta(R, b) = E\left(\bigcup_{q \in R} \Delta(q, b)\right).$$

2.  $N$  following  $\varepsilon$ -transitions *before* the first non- $\varepsilon$ -transition, i.e., define  $s_D = E(\{s_N\})$ .



We have constructed a DFA  $D$  such that the state  $D$  is in after reading string  $x$  is the subset of states that  $N$  could be in after reading  $x$ . So  $D$  accepts  $x$  if and only if at least one of the states  $N$  could be in is accepting, i.e., if and only if  $N$  accepts  $x$ . Thus  $L(D) = L(N)$ .  $\square$

**Corollary 6.1.3.** *A language is DFA-decidable if and only if it is NFA-decidable.*

So from now on, when we call a language “regular”, which up to this point has been a synonym for DFA-decidable, we can also interpret it to mean NFA-decidable.

### end of lecture 4a

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 deciding the language has  $n$  states, while the smallest DFA deciding 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. Thus, if we only care that the number of states is finite, then finite automata have equivalent power whether or not they are non-deterministic. But if we consider the number of states as a *resource* (requiring more memory to implement, for example), then NFAs are more powerful in the sense that for certain problems, they use vastly less resources than the best DFA. This theme will be revisited in the unit on computational complexity.

**Alternate choices that also work.** One thing to note about some of the choices we made about when to simulate  $\varepsilon$ -transitions: we *could* have chosen differently and construct a different, but still correct, DFA. For example, we could simulate  $\varepsilon$ -transitions *before* each input-consuming-transition:

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

and then we could simply let  $s_D = \{s_N\}$ . However, we’d have to remember to also do  $\varepsilon$ -transitions after the last symbol has been read, by asking not only whether any of the states  $N$  could be in is accepting, but also if any accept states are reachable from those states by  $\varepsilon$ -transitions:

$$F_D = \{A \subseteq Q_N \mid A \cap E(F_N) \neq \emptyset\}.$$

There’s nothing wrong with doing the above; it also creates a DFA that decides  $L(N)$ . For example, if we did this in Fig. 6.1, then the  $a$ -transition from  $\{1, 2\}$ , instead of going to  $\{2, 3\}$ , would go to  $\{1, 2, 3\}$ . Although this would be a different DFA, it would be equivalent to that of Fig. 6.1 in the sense of deciding the same language.

We will take the proof of Theorem 6.1.2 to be the “official” subset construction. Particularly for auto-grading, we need to define something to be the “official” construction, and this will be it.

## 6.2 Equivalence of RGs and NFAs

Perhaps you're thinking that context-free grammars also define the regular languages. This turns out not to be true. In Chapter 7, we show that some CFG-decidable languages, such as  $\{0^n 1^n \mid n \in \mathbb{N}\}$ , are not DFA-decidable.

However, there is a one-way implication that is fairly simple to prove:

**Theorem 6.2.1.** *Every DFA-decidable language is RRG-decidable (thus also CFG-decidable).*

*Proof.* Let  $D = (Q, \Sigma, \delta, s, F)$  be a DFA. We construct an RRG  $G = (\Gamma, \Sigma, S, \rho)$  deciding  $L(D)$ . See Figure 6.2 for an example.

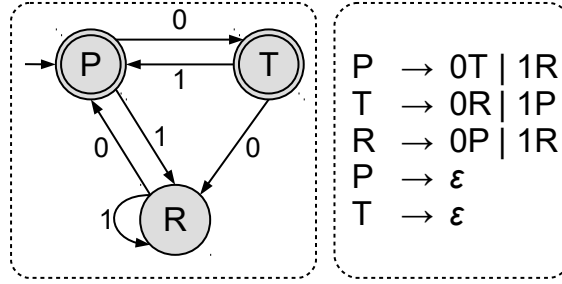


Figure 6.2: **Left:** A DFA  $D$ . **Right:** A right-regular grammar generating  $L(D)$ .

Intuitively, any derived string has exactly 1 or 0 variables.  $G$  has one production rule for each transition in  $D$ , which adds a new symbol to the produced string and possibly changes the variable.  $G$  also has rules to change the variable to  $\varepsilon$  if it represents an accept state. This ensures that any possible string of terminals can be produced, but it is followed by a variable. The variable can be erased only if it represents an accept state.

Formally,  $\Gamma = Q$ ,  $\Sigma$  is the same for each, and  $S = s$ . We have a clash of conventions, lowercase for DFA state names and uppercase for CFG variable names, so we choose uppercase. There is one rule for each transition in  $\delta$ : if  $\delta(A, b) = C$ , then  $G$  has a rule  $A \rightarrow bC$ . There is also one rule for each accept state: if  $A \in F$ , then  $G$  has a rule  $A \rightarrow \varepsilon$ .

**Detailed proof of correctness:** First we show  $L(G) \subseteq L(D)$ . Each rule has at most one variable on the right, so every derived string has at most one variable  $A$ . To eliminate  $A$  and result in an all-terminal string,  $A$  must represent an accepting state, so the string is accepted by  $D$ .

Conversely, to show  $L(D) \subseteq L(G)$ , any string  $x$  accepted by  $D$ , which visits states

$s = A_0, A_1, \dots, A_n \in F$ , can be produced by  $G$  by applying these rules in order:

$$\begin{aligned} A_0 &\rightarrow x[1]A_1 \\ A_1 &\rightarrow x[2]A_2 \\ A_2 &\rightarrow x[3]A_3 \\ &\dots \\ A_{n-1} &\rightarrow x[n]A_n \\ A_n &\rightarrow \varepsilon, \end{aligned}$$

resulting in the derivation  $A_0 \Rightarrow x[1]A_1 \Rightarrow x[1..2]A_2 \Rightarrow x[1..3]A_3 \Rightarrow \dots \Rightarrow x[1..n-1]A_{n-1} \Rightarrow xA_n \Rightarrow x$ . Thus  $L(D) = L(G)$ .  $\square$

In fact, *every* RRG can be interpreted as implementing an NFA with no  $\varepsilon$ -transitions. Thus, RRGs are yet another way to characterize the regular languages, along with DFAs, NFAs, and regex's.

**Theorem 6.2.2.** *Every RRG-decidable language is NFA-decidable.*

*Proof.* Let  $G = (\Gamma, \Sigma, S, \rho)$  be an RRG. We construct an NFA  $N = (Q, \Sigma, \Delta, s, F)$  deciding  $L(G)$ . Let

- $Q = \Gamma$ ,
- $s = S$ ,
- $F = \{A \in \Gamma \mid A \rightarrow \varepsilon \text{ is a rule in } \rho\}$ , and
- for all  $A \in Q$  and  $b \in \Sigma$ ,  $\Delta(A, b) = \{C \mid A \rightarrow bC \text{ is a rule in } \rho\}$ .

Then paths from  $s$  to some state in  $F$  in  $N$  correspond exactly to a sequence of rules in  $G$ , all of which produce a new terminal except the last. Thus for all  $x \in \Sigma^*$ ,  $x \in L(N) \iff x \in L(G)$ .  $\square$

There's an obvious twist on this definition to consider: *left*-regular grammars (LRG), where the non- $\varepsilon$  rules are of the form  $A \rightarrow Cb$  for  $C$  a variable and  $b$  a terminal. As you might suspect, these also define the regular languages. Here's one easy way to see that: if one reverses all the strings on the right side of production rules of a CFG  $G$ , the resulting CFG  $G'$  defines the reverse language, i.e.,  $L(G') = L(G)^{\mathcal{R}}$ . If  $G$  is right-regular (respectively, left-regular), then  $G'$  is left-regular (respectively, right-regular). Since the regular languages are closed under reverse (this hasn't been shown, but it is true), this means  $G'$  is regular if and only if  $G$  is regular.

A CFG is a *regular grammar* (RG) if it is either left-regular or right-regular. Thus, we have the following:

**Theorem 6.2.3.** *A language is regular if and only if it is RG-decidable.*

Be careful here with the definition of RG: either all non- $\varepsilon$  rules are of the form  $A \rightarrow Cb$ , or all non- $\varepsilon$  rules are of the form  $A \rightarrow bC$ . It is *not* saying, for each non- $\varepsilon$  rule in the grammar, it is either of the form  $A \rightarrow Cb$  or  $A \rightarrow bC$ . In other words you cannot *mix* left-regular and right-regular rules in a single grammar. For example, the grammar

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

decides the language  $\{0^n 1^n \mid n \in \mathbb{N}\}$ , which we stated above (and will prove in Chapter 7), is not regular.

end of lecture 4b

### 6.3 Equivalence of regex's and NFAs

**Theorem 6.3.1.** *A language is regular if and only if it is regex-decidable.*

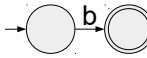


We prove each direction separately via two lemmas.

#### 6.3.1 Every regex-decidable language is NFA-decidable

**Lemma 6.3.2.** *Every regex-decidable language is NFA-decidable.*

*Proof.* Let  $R$  be a regex with alphabet  $\Sigma$ . It suffices to construct an NFA  $N = (Q, \Sigma, \Delta, s, F)$  such that  $L(R) = L(N)$ . An example of this construction is shown in Figure 6.3.

The definition of regex gives us six cases:

1.  $R = b$ , where  $b \in \Sigma$ , so  $L(R) = \{b\}$ , decided by this NFA: 
2.  $R = \varepsilon$ , so  $L(R) = \{\varepsilon\}$ , decided by this NFA: 
3.  $R = \emptyset$ , so  $L(R) = \emptyset$ , decided by this NFA: 
4.  $R = R_1 \cup R_2$ , so  $L(R) = L(R_1) \cup L(R_2)$ .
5.  $R = R_1 R_2$ , 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 NFA-decidable. Since the NFA-decidable languages are closed under the operations of  $\cup$ ,  $\circ$ , and  $*$ ,  $L(R)$  is NFA-decidable. □

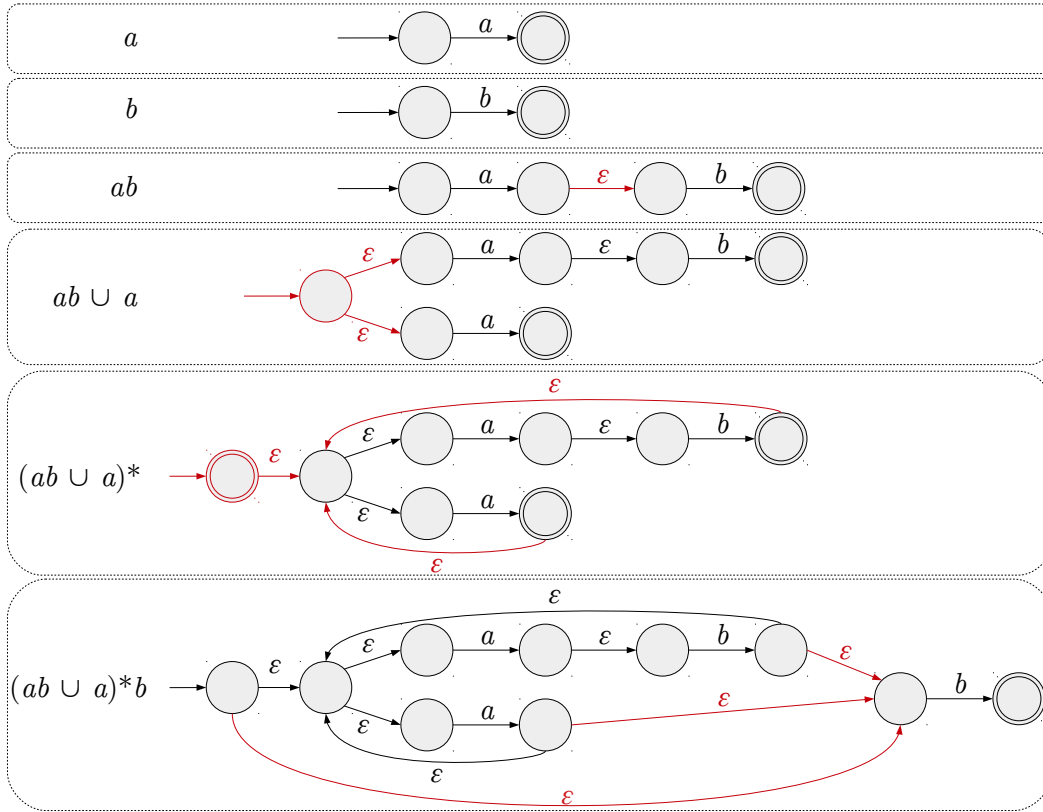


Figure 6.3: Example of converting the regex  $(ab \cup a)^*b$  to an NFA. The recursion is shown “bottom-up”, starting with the base cases for  $a$  and  $b$  and building larger NFAs using the union construction, concatenation construction, and Kleene star construction, as in Figures 5.3, 5.4, and 5.5.

Thus, to convert a regex into an NFA, we replace the base cases in the regex with the simple 1- or 2-state NFAs above, and then we connect them using the constructions for NFA union, concatenation, and Kleene star shown in Section 5.3, shown by example in Figures 5.3, 5.4, and 5.5.

### 6.3.2 Every NFA-decidable language is regex-decidable

First, we observe that NFAs can be converted easily to have a special form that will be useful in the construction.

**Lemma 6.3.3.** *Every NFA-decidable language is decided by an NFA  $N = (Q, \Sigma, \Delta, S, F)$ , such that  $s$  has no transition arrows entering it,  $F = \{a\}$  where  $a \neq s$ , and  $a$  has no transition arrows leaving it.*

**Proof idea:** Add a new start state and a new accept state, with new  $\epsilon$ -transitions to mimic the original behavior. See Fig. 6.4 for an example.

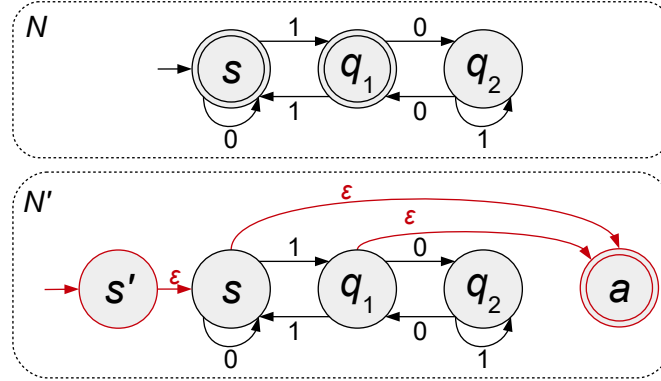


Figure 6.4: Modifying an NFA to ensure it has no transitions entering the start state  $s'$ , and it has a single accept state  $a$  with no transitions leaving  $a$ .

*Proof.* Let  $N = (Q, \Sigma, \Delta, s, F)$  be an NFA. Define the NFA  $N' = (Q', \Sigma, \Delta', s', F')$  as follows. Let  $s', a$  be states not in  $Q$ . Let

- $Q' = Q \cup \{s', a\}$ ,
- $F' = \{a\}$ , and
- $\Delta'$  has all the transitions of  $\Delta$ , and also  $s' \xrightarrow{\epsilon} s$ , and for each  $q \in F$ ,  $q \xrightarrow{\epsilon} a$ .

If a string  $x$  is accepted by  $N$ , ending in accept state  $a \in F$ , then the same computation sequence, with  $s' \xrightarrow{\epsilon} s$  added to the start, and  $q \xrightarrow{\epsilon} a$  added to the end for some  $q \in F$ , is a computation sequence showing that  $N'$  accepts  $x$  as well. Thus  $x \in L(N) \implies x \in L(N')$ , i.e.,  $L(N) \subseteq L(N')$ .

Conversely, every accepting computation sequence of  $N'$  on  $x$  must start with  $s' \xrightarrow{\epsilon} s$  and end with  $q \xrightarrow{\epsilon} a$  for some  $q \in F$ . Removing these two transitions results in an accepting computation sequence of  $N$  on  $x$ , showing  $L(N') \subseteq L(N)$ .  $\square$

Now we show how to convert any NFA  $N$  into a regex  $R$  so that  $L(N) = L(R)$ . First we introduce a generalization of NFAs that will be helpful in the conversion, called *expression automata* (EA). Intuitively, an expression automaton can have its transition arrows labeled with arbitrary regular expressions. A transition from state  $a$  to  $b$  labeled with regex  $X$  is written  $a \xrightarrow{X} b$ . Since  $\epsilon$  and individual alphabet symbols are regular expressions, every NFA is an EA. The idea is that an EA may read any number of symbols from the input while following a transition, as long as the substring read matches the regular expression labeling the transition.

The construction will work like this. We start with an NFA (which is a special type of EA). Then we repeatedly remove one state  $i$  at a time from the EA. When we do, we stitch together the transition arrows coming into and out of  $i$ , combining their regex's so that the EA still decides the same language. When we are done, we will have exactly two states:

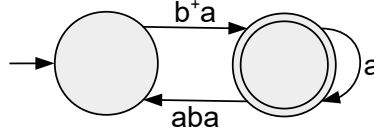


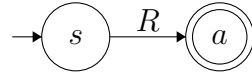
Figure 6.5: Example of an expression automaton (EA). It generalizes NFAs to allow arbitrary regex's on the transitions. This EA accepts the strings  $bba$ ,  $bbaa$ , and  $bbaaababbba$ , and it rejects the strings  $\varepsilon$ ,  $a$ ,  $b$ ,  $baaba$ , and  $bbaabab$ .

a rejecting start state, with a transition going to an accept state. The regex labeling that transition is the output of the construction.

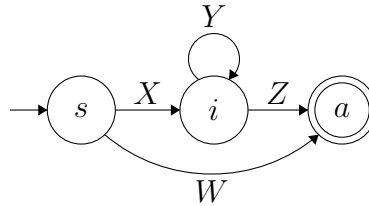
Now, we show the construction that converts any NFA into an equivalent regex.

**Theorem 6.3.4.** *Every NFA-decidable language is regex-decidable.*

*Proof.* Let  $N = (Q, \Sigma, \Delta, s, F)$  be an NFA. By Lemma 6.3.3 we may assume  $N$  has no transitions entering  $s$  and that  $F = \{a\}$  with no transitions leaving  $a$ . We convert  $N$  to an EA  $E$  such that  $L(N) = L(E)$ , where  $E$  is of the following form:



Then  $L(E) = L(R)$ . Intuitively, suppose we have an EA  $E_1$  of the following form.



where  $W, X, Y, Z$  are regex's. Then the following EA  $E_2$  decides the same language and matches the form we seek:



because in both, the strings leading to  $a$  are either of the form  $w \in L(W)$ , or of the form  $xy^kz$  for some  $k \in \mathbb{N}$ , where  $x \in L(X)$ ,  $y \in L(Y)$ , and  $z \in L(Z)$ .

More generally, we may have many transitions entering and leaving the intermediate state  $i$ . We iterate over each pair of states  $q, r$  such that there are transitions  $q \xrightarrow{X} i$  and  $i \xrightarrow{Z} r$ . (We do this even if  $q = r$ , such as  $t$  in Fig. 6.6.) Suppose there are also transitions  $i \xrightarrow{Y} i$  and  $q \xrightarrow{W} r$ . We replace these 4 transitions with  $q \xrightarrow{W \cup XY^*Z} r$ . If  $i \xrightarrow{Y} i$  or  $q \xrightarrow{W} r$  or both were not there, then omit  $Y$  or  $W$  or both (i.e., the regex could be  $W \cup XZ$ ,  $XY^*Z$ , or  $XZ$ ). Figure 6.6 shows an example.

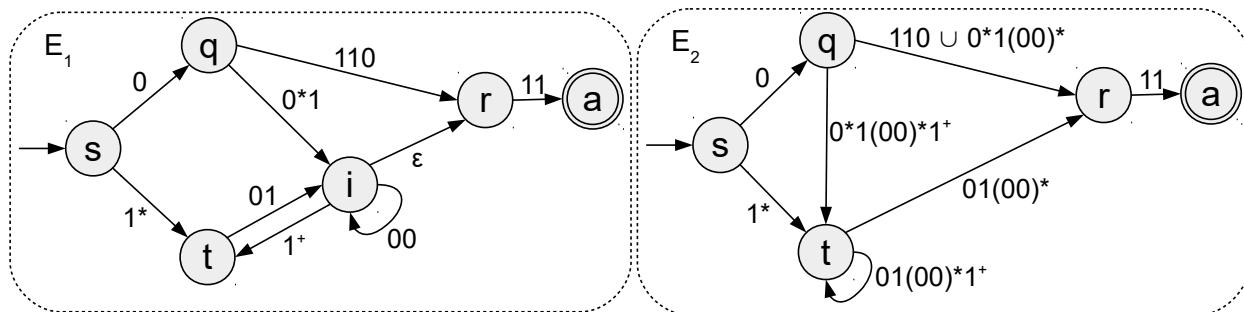


Figure 6.6: Modifying EA  $E_1$  to remove state  $i$ , resulting in EA  $E_2$ . We have only one state removal to show the intuitive idea, because this construction makes very large regexes when all states are removed. The pairs of states between which we add transitions are  $(q, r)$ ,  $(q, t)$ ,  $(t, r)$ ,  $(t, t)$ . Note that since  $E_1$  has transitions between  $t$  and  $i$  in both directions,  $E_2$  has a self-loop on  $t$ . Also note that since there was already a transition  $q \xrightarrow{110} r$ , we must take the union of the regex  $110$  with the new regex  $0^*1(00)^*$  on the new transition  $q \xrightarrow{110 \cup 0^*1(00)^*} r$ .

Repeat this for all states  $i \in Q \setminus \{s, a\}$ , and at that point the EA matches the form described above, with a single regex  $R$  such that  $L(R) = L(N)$ .  $\square$

Now we have done everything we said we'd do in this chapter: we have shown that DFAs and NFAs are equivalent, that regex's are equivalent to both, and that RGs are equivalent to all. We say a language is *regular* if it decided by any of these automata, knowing that an automaton of one type can be converted to an equivalent automaton (meaning, it decides the same language) of any of the other types. In the next chapter, we move on to studying languages that *cannot* be decided by any of these automata.

end of lecture 4c

## 6.4 Optional: Equivalence of DFAs and constant memory programs

We have alluded to the idea that a DFA is a simplified model of a program with constant memory. Actually, we can even think of the program as having (almost) *no memory*. The states in a DFA are analogous to *lines of code* in a program. **if** statements and **while/for** loops transition the program to one line of code or another based on a Boolean test, in the same way that a test on the “current” input symbol transitions the DFA to one state to another.

For example, the DFA in Figure 3.6 is implemented by the following C++ function (the state names have been changed to start with “q” to be valid label identifiers in C++):



```

1  #include <cstdlib>
2  bool dfa(const std::string input_string) {
3      // this is the only variable we will declare
4      std::string::iterator it = input_string.begin();
5  q0:
6      if (it == input_string.end())
7          return true;
8      if (*it == '0') {
9          it++;
10         goto q0;
11     } else {
12         it++;
13         goto q1;
14     }
15  q1:
16     if (it == input_string.end())
17         return false;
18     if (*it == '0') {
19         it++;
20         goto q2;
21     } else {
22         it++;
23         goto q0;
24     }
25  q2:
26     if (it == input_string.end())
27         return false;
28     if (*it == '0') {
29         it++;
30         goto q1;
31     } else {
32         it++;
33         goto q2;
34     }
35 }

```

Now, the semantics of DFA execution take care of some of the logic above, which is why the DFA has only 3 states, yet the C++ program has 35 lines of code. But it is still one block of 10 lines per state.

It is also the case that it is somewhat “cheaty” to say the program has *no* memory, and not just because one iterator variable is needed. Even if no variables were needed, usually the “line of code” is translated, when the program is compiled, to machine code instructions, and the “line” of code becomes an index into these instructions, i.e., an integer, implemented in memory/cache as something called a *program counter*.

The above example illustrates that the following implication is true: if a language is DFA-decidable, then it is solvable by a C++ `bool` function with only a single variable to iterate over the input. The converse implication is true as well, though we will not attempt to prove it here, as the complex semantics of a general-purpose programming language such as C++ would make for a tedious proof.

Similar statements are true about other programming languages such as Python, but

not exactly the same statement, since Python lacks a `goto` statement. Like C/C++, the language of DFAs allows unstructured programming, where one can potentially jump from any state (line of code) to any other. So to truly implement a DFA in a structured language such as Python would require some memory to keep track of the state. For example, the following Python function implements the same DFA as above, using an enumerated type to represent the state:

```
1 from enum import Enum, auto
2 class State(Enum):
3     Q0 = auto()
4     Q1 = auto()
5     Q2 = auto()
6
7 def dfa(input_string) {
8     state = State.Q0
9     for (symbol in input_string):
10         if state is State.Q0:
11             if (symbol == '0'):
12                 state = State.Q0
13             else:
14                 state = State.Q1
15         if state is State.Q1:
16             if (symbol == '0'):
17                 state = State.Q2
18             else:
19                 state = State.Q0
20         if state is State.Q2:
21             if (symbol == '0'):
22                 state = State.Q1
23             else:
24                 state = State.Q2
25     if state is State.Q1:
26         return True
27     else:
28         return False
```

## Chapter 7

# Proving problems are not solvable in a model of computation

### 7.1 Some languages are not regular

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

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

$C$  is not regular.<sup>1</sup>  $C'$  looks like almost the same language, and it appears as though it might not be regular either, since it appears to require counting how many 01's and how many 10's appear in the string, and DFAs cannot count arbitrarily high. However,  $C' = \{x \in \{0, 1\}^* \mid x[1] = x[|x|]\}$  (a regular language decided by the regex  $0 \cup 1 \cup 0\Sigma^*0 \cup 1\Sigma^*1$  for  $\Sigma = \{0, 1\}$ ), since a string that begins and ends with the same bit if and only if it changes bits (switching from 0 to 1, or from 1 to 0) the same number of times in each direction.

These examples show that, to demonstrate no DFA can solve a decision problem, it is insufficient to make noises such as “you need to count arbitrarily high, so you can’t do it with finite states.” Without a formal definition of what it means to “need to count arbitrarily high”, you could make mistakes and think you “need” to count when you really don’t, as with language  $C'$  above.

But we claimed the language  $C$  above is not regular, meaning *no* DFA decides it. Let’s prove it formally.

For  $q \in Q$  and  $x \in \Sigma^*$ , let  $\hat{\delta}(q, x)$  be the state  $D$  is in after reading  $x$ , if starting from state  $q$ . Formally, we can define it recursively:  $\hat{\delta}(q, \varepsilon) = q$  and  $\hat{\delta}(q, xb) = \delta(\hat{\delta}(q, x), b)$  for  $x \in \Sigma^*$  and  $b \in \Sigma$ . Note that if  $\hat{\delta}(q_1, x) = q_2$  and  $\hat{\delta}(q_2, y) = q_3$ , then  $\hat{\delta}(q_1, xy) = q_3$ .

**Theorem 7.1.1.** *Let  $C = \{x \in \{0, 1\}^* \mid \#(0, x) = \#(1, x)\}$ . Then  $C$  is not regular.*

<sup>1</sup>Intuitively, it appears to require unlimited memory to count all the 0’s. But we must prove this formally; the example  $C'$  shows that a language can appear intuitively to require unbounded memory while actually being regular.

*Proof.* Let  $D = (Q, \Sigma, \delta, s, F)$  be a DFA; we show  $L(D) \neq C$ . Let  $p = |Q|$  and  $w = 0^p 1^p$ ; note  $w \in C$ .

Let  $q_0, q_1, \dots, q_n \in Q$  be the computation sequence of  $D$  on  $w$ , with  $q_0 = s$ . If  $q_n \notin F$ , then  $w \notin L(D)$ , so  $L(D) \neq C$  and we are done, so assume  $q_n \in F$ . Since there are more than  $p$  states in  $q_0, q_1, \dots, q_p$ , by the pigeonhole principle, there are  $i, j$  with  $0 \leq i < j \leq p$  where  $q_i = q_j$ . Let  $m = j - i$ , noting  $m > 0$ . Let  $y = w[i + 1..j]$ . Since  $q_i = q_j$ ,  $\widehat{\delta}(q_i, y) = q_i$ .<sup>2</sup>

Let  $x = w[1..i]$ , so  $\widehat{\delta}(q_0, x) = q_i$  and  $z = w[j + 1..|w|]$ , so  $\widehat{\delta}(q_i, z) = q_n$ . Since  $\widehat{\delta}(q_i, y) = q_i$ ,  $\widehat{\delta}(q_i, yy) = q_i$ . Thus  $\widehat{\delta}(q_0, xyy) = q_i$ , and  $\widehat{\delta}(q_0, xyyz) = q_n$ . Since  $q_n \in F$ ,  $xyyz \in L(D)$ .

Since  $w$  starts with  $p$  0's,  $x = 0^i$  and  $y = 0^m$ . But since  $\#(0, xyyz) = p + m > p$  but  $\#(1, xyyz) = p$ ,  $xyyz \notin C$ . Since  $xyyz \in L(D)$ ,  $D$  does not decide  $C$ .  $\square$

Here's another non-regular language. The proof is almost identical; we highlight differences in red.

**Theorem 7.1.2.** *The language  $G = \{ uu \mid u \in \{0, 1\}^* \}$  is not regular.*

*Proof.* Let  $D = (Q, \Sigma, \delta, s, F)$  be a DFA; we show  $L(D) \neq G$ . Let  $p = |Q|$  and  $w = 1^p 0 1^p$ ; note  $w \in G$ .

Let  $q_0, q_1, \dots, q_n \in Q$  be the computation sequence of  $D$  on  $w$ , with  $q_0 = s$ . If  $q_n \notin F$ , then  $w \notin L(D)$ , so  $L(D) \neq G$  and we are done, so assume  $q_n \in F$ . Since there are more than  $p$  states in  $q_0, q_1, \dots, q_p$ , by the pigeonhole principle, there are  $i, j$  with  $0 \leq i < j \leq p$  where  $q_i = q_j$ . Let  $k = j - i$ , noting  $k > 0$ . Let  $y = w[i + 1..j]$ . Since  $q_i = q_j$ ,  $\widehat{\delta}(q_i, y) = q_i$ .

Let  $x = w[1..i]$ , so  $\widehat{\delta}(q_0, x) = q_i$  and  $z = w[j + 1..|w|]$ , so  $\widehat{\delta}(q_i, z) = q_n$ . Since  $\widehat{\delta}(q_i, y) = q_i$ ,  $\widehat{\delta}(q_i, yy) = q_i$ . Thus  $\widehat{\delta}(q_0, xyy) = q_i$ , and  $\widehat{\delta}(q_0, xyyz) = q_n$ . Since  $q_n \in F$ ,  $xyyz \in L(D)$ .

Since  $w$  starts with  $p$  1's and  $j \leq p$ ,  $x = 1^i$  and  $y = 1^k$ .  $xyz$  has a single 0 in each half, but  $xyyz$  both 0's in the right half. Thus  $xyyz$  does not have equal left and right halves, so  $xyyz \notin G$ . Since  $xyyz \in L(D)$ ,  $D$  does not decide  $G$ .  $\square$

When programming, when you find yourself tempted to copy and paste code from one part of a project to another, it is helpful to factor out the common code into a single function that can be called repeatedly. Similarly, when in the course of proving two theorems, you find that they have very similar proofs, it can be helpful to factor out the common parts into a single general-purpose lemma. This guides future proofs and removes redundant parts. We factor out the common parts of the two proofs into a statement called the *Pumping Lemma*.

## 7.2 The pumping lemma for regular languages

The pumping lemma is based on the pigeonhole principle. The proof informally states, "If an input  $w$  to a DFA is long enough, then some state must be visited twice, and the substring  $y$

<sup>2</sup>In other words, the string  $y$  takes  $D$  from state  $q_i$  back to itself. Thus, *another* copy of  $y$  at this point would return to state  $q_i$  again.

between the two visitations can be repeated (or omitted) without changing the DFA's final answer."<sup>3</sup>

This idea is formalized next. It gives names to the substring  $x$  processed before the first instance of the repeated state, and the substring  $z$  processed after the second instance of the repeated state. It also observes that no matter how long the string  $w$ , if the DFA has  $p$  states then some state must be repeated within the *first*  $p$  symbols of  $w$ .

**Pumping Lemma.** *If  $A$  is a regular language, there is a number  $p$  (the pumping length) so that if  $w \in A$  and  $|w| \geq p$ , then  $w$  may be divided into three substrings  $w = xyz$ , satisfying the conditions:*

1. for all  $k \in \mathbb{N}$ ,  $xy^kz \in A$ ,
2.  $|y| > 0$ , and
3.  $|xy| \leq p$ .

### 7.2.1 Using the Pumping Lemma

We prove the Pumping Lemma at the end of the section. It closely resembles the proofs above, however. First we show how to use it to prove languages are not regular.

The strategy for employing the Pumping Lemma to show a language  $L$  is not regular is: assuming  $L$  is regular with pumping length  $p$ , find a long string in  $L$ , long enough that it can be pumped (it has length at least  $p$ , and often longer so that we can make the first  $p$  symbols be equal, which often works), then prove that pumping it “moves it out of the language”. Thus  $D$  does not decide  $L$ , since  $D$  either accepts both strings or rejects both, by the Pumping Lemma.

**Theorem 7.2.1.** *The language  $B = \{ 0^n 1^n \mid n \in \mathbb{N} \}$  is not regular.*

*Proof.* Assume for the sake of contradiction that  $B$  is regular, with pumping length  $p$ . Let  $w = 0^p 1^p$ . Since  $w \in B$  and  $|w| \geq p$ , by the Pumping Lemma,  $w = xyz$ , where  $|y| > 0$ ,  $|xy| \leq p$ , and for all  $k \in \mathbb{N}$ ,  $xy^kz \in B$ .

Since  $|xy| \leq p$  and  $w$  starts with  $p$  0's,  $y$  is all 0's. Then  $xyyz$  has more 0's than 1's.

Thus  $xyyz \notin B$ , contradicting condition (1) of the Pumping Lemma.  $\square$

In the previous subsection, we gave a direct proof of the following theorem.

**Theorem 7.2.2.** *The language  $C = \{w \in \{0,1\}^* \mid \#(0,w) = \#(1,w)\}$  is not regular.*

To help avoid mistakes that are often made when using the Pumping Lemma, let's first see what a bad proof of this theorem looks like.

---

<sup>3</sup>That is, we “pump” more copies of  $y$  into the full string  $w$ . 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.

*Bad proof.* Assume for the sake of contradiction that  $C$  is regular, with pumping length  $p$ . Let  $w = 0^{\lceil p/2 \rceil} 1^{\lceil p/2 \rceil}$ . Since  $w \in C$  and  $|w| \geq p$ , by the Pumping Lemma,  $w = xyz$ , where  $|y| > 0$ ,  $|xy| \leq p$ , and for all  $k \in \mathbb{N}$ ,  $xy^kz \in C$ .

Let  $x = \varepsilon$ ,  $y = 0^{\lceil p/2 \rceil}$ , and  $z = 1^{\lceil p/2 \rceil}$ . Then  $xyyz = 0^{2\lceil p/2 \rceil} 1^{\lceil p/2 \rceil}$ , which has more 0's than 1's.

Thus  $xyyz \notin C$ , contradicting condition (1) of the Pumping Lemma.  $\square$

What's wrong with this proof? It's highlighted in red. We wanted  $y$  to be all 0's so that when we pump a second copy into  $w = xyz$  to create  $xyyz$ , we end up with more 0's than 1's. But just because we *want*  $y$  to be all 0's doesn't mean we get to simply *declare* that it is all 0's. The source of the error is the sentence, "Let  $x = \varepsilon$ ,  $y = 0^{\lceil p/2 \rceil}$ , and  $z = 1^{\lceil p/2 \rceil}$ ." What if those aren't the  $x$ ,  $y$ , and  $z$  that the Pumping Lemma finds? What if  $x = z = \varepsilon$  and  $y = 0^{\lceil p/2 \rceil} 1^{\lceil p/2 \rceil}$ ? This is totally consistent with the Pumping Lemma:  $|y| > 0$  and  $|xy| \leq p$ , just as promised. But for this choice of  $x$  and  $y$ , then  $xy^kz = (0^{\lceil p/2 \rceil} 1^{\lceil p/2 \rceil})^k$ , which has an equal number of 0's and 1's. So  $xy^kz \in C$  for all  $k$ , and we don't get a contradiction.

Think of the Pumping Lemma as a contract: we give it  $w$  of length at least  $p$ , and it gives back  $x$ ,  $y$ , and  $z$ . It is guaranteed to fulfill the exact terms of the contract:  $|y| > 0$ ,  $|xy| \leq p$ , and  $xy^kz \in C$  for all  $k \in \mathbb{N}$ . Nothing more, nothing less.

If we want  $y$  to have some extra property, such as being all 0's, we have to *prove* that  $y$  has that property, that this property *follows from* the conditions  $|y| > 0$  and  $|xy| \leq p$  guaranteed by the Pumping Lemma. We must take care in choosing  $w$  so that *every* choice of  $x$ ,  $y$ , and  $z$  satisfying  $|y| > 0$  and  $|xy| \leq p$  will give us the property we want.

In this case, if we choose  $w$  to be a bit longer, we can use condition (3) to conclude that  $y$  is indeed all 0's.

*Proof of Theorem 7.2.2.* Assume for the sake of contradiction that  $C$  is regular, with pumping length  $p$ . Let  $w = 0^p 1^p$ . Since  $w \in C$  and  $|w| \geq p$ , by the Pumping Lemma,  $w = xyz$ , where  $|y| > 0$ ,  $|xy| \leq p$ , and for all  $k \in \mathbb{N}$ ,  $xy^kz \in C$ .

Since  $|xy| \leq p$  and  $w$  starts with  $p$  0's,  $y$  is all 0's. Then  $xyyz$  has more 0's than 1's.

Thus  $xyyz \notin C$ , contradicting condition (1) of the Pumping Lemma.  $\square$

## end of lecture 5a

Here's an alternate proof that does not directly use the Pumping Lemma. It uses closure properties and appeals to the fact that we already proved that  $B = \{0^n 1^n \mid n \in \mathbb{N}\}$  is not regular.

*Alternate proof of Theorem 7.2.2.* If  $C$  were regular, then by closure of regular languages under  $\cap$ ,  $C \cap L(0^* 1^*) = \{0^n 1^n \mid n \in \mathbb{N}\}$  would be regular,<sup>4</sup> contradicting Theorem 7.2.1.  $\square$

<sup>4</sup>An analogy: if I tell you I have a number  $x$  and an integer  $n$ , and I tell you  $x + n = 3.14$ , what do you know about  $x$ ? We don't know the exact value of  $x$ , but we know it can't be an integer. The integers are closed under addition (the sum of two integers is always an integer), so if I add  $x$  to an integer and get 3.14,  $x$  must be a non-integer.

**Theorem 7.2.3.** *The language  $G = \{ uu \mid u \in \{0,1\}^* \}$  is not regular.*

*Proof.* Assume for the sake of contradiction that  $G$  is regular, with pumping length  $p$ . Let  $w = 1^p 0 1^p 0$ . Since  $w \in G$  and  $|w| \geq p$ , by the Pumping Lemma,  $w = xyz$ , where  $|y| > 0$ ,  $|xy| \leq p$ , and for all  $k \in \mathbb{N}$ ,  $xy^k z \in G$ .

Since  $|xy| \leq p$ ,  $x$  and  $y$  are all 1's. Let  $m = |y| > 0$ . Then  $xyyz = 1^{p+m} 0 1^p 0$ . Since  $p + m > p$ ,  $xyyx \neq uu$  for any  $u$ .

Thus  $xyyz \notin G$ , contradicting condition (1) of the Pumping Lemma.  $\square$

Here is a nonregular unary language.

**Theorem 7.2.4.** *The language  $D = \{ 1^{n^2} \mid n \in \mathbb{N} \}$  is not regular.*

*Proof.* Assume for the sake of contradiction that  $D$  is regular, with pumping length  $p$ . Let  $w = 1^{p^2}$ . Since  $w \in D$  and  $|w| \geq p$ , by the Pumping Lemma,  $w = xyz$ , where  $|y| > 0$ ,  $|xy| \leq p$ , and for all  $k \in \mathbb{N}$ ,  $xy^k z \in D$ .

Let  $w' = 1^{(p+1)^2}$  be the next biggest string in  $D$  after  $w$ ; then no string  $u$  where  $|w| < |u| < |w'|$  is in  $D$ . Then

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

Since  $|y| \leq p$ ,  $|xyyz| - |w| \leq p$ , so  $|xyyz| < |w'|$ . Since  $|y| > 0$ ,  $|w| < |xyyz| < |w'|$ .

Thus  $xyyz \notin D$ , contradicting condition (1) of the Pumping Lemma.  $\square$

The next example shows that “pumping down” (replacing  $xyz$  with  $xz$ ) can be useful.

**Theorem 7.2.5.** *The language  $E = \{0^i 1^j \mid i > j\}$  is not regular.*

*Proof.* Assume for the sake of contradiction that  $E$  is regular, with pumping length  $p$ . Let  $w = 0^{p+1} 1^p$ . Since  $w \in E$  and  $|w| \geq p$ , by the Pumping Lemma,  $w = xyz$ , where  $|y| > 0$ ,  $|xy| \leq p$ , and for all  $k \in \mathbb{N}$ ,  $xy^k z \in E$ .

Since  $|xy| \leq p$ ,  $y$  is all 0's. Let  $m = |y| > 0$ . Then  $xz = 0^{p+1-m} 1^p$ , noting  $p+1-m \leq p$ .

Thus  $xz \notin E$ , contradicting condition (1) of the Pumping Lemma.  $\square$

Finally, let's see a proof where we can't choose the first  $p$  symbols to be the same, since the language disallows it. We have more cases to consider, since we cannot guarantee that, as in the above proofs,  $y$  will be a unary string. Depending on the exact form of  $y$ , different arguments are needed.

**Theorem 7.2.6.** *The language  $H = \{(01)^n (10)^n \mid n \in \mathbb{N}\}$  is not regular.*

For example,  $H$  contains  $\varepsilon, 0110, 01011010, 010101101010$  but not  $0011$  or  $0101$ .

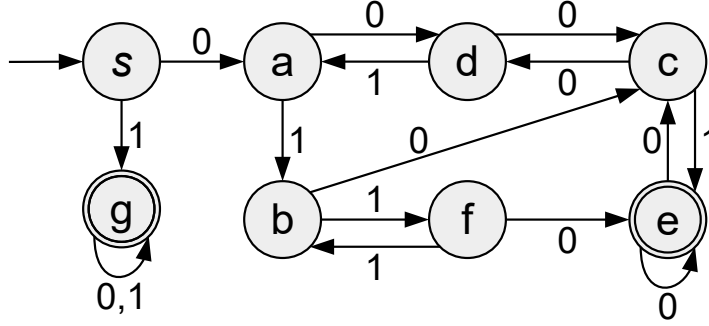


Figure 7.1: A DFA  $D = (Q, \Sigma, \delta, s, F)$  to help illustrate the idea of the pumping lemma. Reading  $x$  reaches  $a$ , then reading  $y$  returns to  $a$ , then reading  $z$  reaches accepting state  $e$ . The strings  $xz$ ,  $xyz$ ,  $xyyz$ ,  $xyyyz$ ,  $\dots$  are all accepted too, since they also end up in state  $e$ . Their paths through  $D$  differ only in how many times they follow the cycle  $(a, b, c, d, a)$ .

*Proof.* Assume for the sake of contradiction that  $H$  is regular, with pumping length  $p$ . Let  $w = (01)^p(10)^p$ . Since  $w \in H$  and  $|w| \geq p$ , by the Pumping Lemma,  $w = xyz$ , where  $|y| > 0$ ,  $|xy| \leq p$ , and for all  $k \in \mathbb{N}$ ,  $xy^kz \in H$ .

Since  $|xy| \leq p$  and  $w$  starts with  $p$  consecutive 01 substrings,  $x$  and  $y$  both occur in this region.<sup>5</sup> Say that a *block* in a string is a substring of length 2 starting at an odd index. For example, the blocks of 01011011 are 01, 01, 10, 11. We have two cases:

$|y|$  is even: Then either  $y = (01)^m$  or  $(10)^m$  for some  $m$ . In either case,  $xyyz$  has more 01 blocks than 10 blocks, so  $xyyz \notin H$ .<sup>6</sup>

$|y|$  is odd: Then  $y$  has a different number of 0's and 1's, so  $xyyz$  does also, so  $xyyz \notin H$ .

Thus  $xyyz \notin H$ , contradicting condition (1) of the Pumping Lemma.  $\square$

### 7.2.2 Proof of the Pumping Lemma

Finally, we prove the Pumping Lemma.

See the DFA  $D$  in Figure 7.1.

Note that if we reach, for example,  $a$ , then read the bits 1001, we will return to  $a$ . Therefore, for any string  $x$  such that  $\widehat{\delta}(s, x) = a$ , such as  $x = 0$ , it follows that  $\widehat{\delta}(s, x1001) = q_1$ , and  $\widehat{\delta}(s, x10011001) = a$ , etc. i.e., for all  $k \in \mathbb{N}$ , defining  $y = 1001$ ,  $\widehat{\delta}(s, xy^k) = a$ .

Also notice that if we are in state  $a$ , and we read the string  $z = 110$ , we end up in accepting state  $e$ . Therefore,  $\widehat{\delta}(s, xz) = e$ , thus  $D$  accepts  $xz$ . But combined with the

<sup>5</sup>If  $y$  were of the form  $(01)^*$ , the rest of the proof could be similar to the previous proofs. The complication is that, although  $y$  appears in the first half of the string  $w$ , which is of the form  $(01)^*$ ,  $y$  is merely a *substring* of it, and so it may not start with 0 or end with 1 or have even length.

<sup>6</sup>This becomes apparent by trying out a few examples of where  $y$  could be.



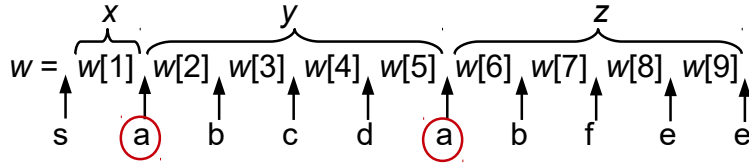
previous reasoning, we also have that  $D$  accepts  $xy^iz$  for all  $n \in \mathbb{N}$ . In the examples  $x = 0, y = 1001, z = 110$ .

Note that the cycle in which a string gets trapped may depend on the string. Any long enough string starting with 1 will repeatedly visit the state  $f$ , but will never visit  $a$ .

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  $D$  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  $xy^kz$  for any  $i \in \mathbb{N}$ , since all of those strings take  $D$  to the same state.

*Proof of Pumping Lemma.* Let  $A$  be a regular language, decided by a DFA  $D = (Q, \Sigma, \delta, s, F)$ . Let  $p = |Q|$ .

Let  $w \in A$  be a string with  $|w| = n \geq p$ . Let  $q_0, q_1, \dots, q_n$  be the computation sequence of  $n + 1$  states of  $D$  on  $w$ . Note  $q_0 = s$  and  $q_n \in F$ . Since  $n + 1 \geq p + 1 > |Q|$ , by the pigeonhole principle, two states in the prefix  $q_0, q_1, \dots, q_p$  must be equal. We call them  $q_i$  and  $q_j$ , with  $i < j \leq p$ . Let  $x = w[1..i]$ ,  $y = w[i+1..j]$ , and  $z = w[j+1..n]$ . For example, with  $i = 1, j = 5$ , and  $q_i = q_j = a$ : (This corresponds to the string  $w = 010011100$  on the DFA in Figure 7.1.)



Since  $\widehat{\delta}(q_i, y) = q_j = q_i$ , for all  $k \in \mathbb{N}$ ,  $\widehat{\delta}(q_j, y^k) = q_j$ .

Therefore  $\widehat{\delta}(q_0, xy^kz) = q_n \in F$ , so  $xy^kz \in L(D)$ , satisfying (1). Since  $j \neq i$ ,  $|y| > 0$ , satisfying (2). Finally,  $j \leq p$ , so  $|xy| \leq p$ , satisfying (3).  $\square$

### 7.3 Optional: The Pumping Lemma for context-free languages

We showed that the language  $\{0^n 1^n \mid n \in \mathbb{N}\}$  is not regular, but it is context-free (CFG-decidable, by the CFG  $S \rightarrow 0S1 \mid \varepsilon$ ). Is every language context-free? It turns out the answer is no; for example  $\{0^n 1^n 2^n \mid n \in \mathbb{N}\}$  is not context-free.

There is a version of the pumping lemma for context-free languages, which helps us prove that certain languages are not context-free, in the same way the original pumping lemma is used to show certain languages are not regular.

As with the pumping lemma for regular languages, we first prove that some languages are not context-free directly, in order to see the proof technique in a more direct and less abstract way, before stating the full lemma.

Intuitively, given any CFG  $G$  with  $k$  variables, any sufficiently large string must have a parse tree of  $G$  with a root-to-leaf path of length  $> k$ . By the pigeonhole principle, along this path, some variable  $A$  must repeat. But what does it mean for a subtree with  $A$  at the

root to also have  $A$  in an lower node? It means that we could replace the smaller subtree with the larger subtree.

Create figure showing this.

Let's use this idea to prove a simple lemma.

**Lemma 7.3.1.** *If  $G = (\Gamma, \Sigma, S, \rho)$  is a CFG with at most  $b \in \mathbb{N}$  symbols on the right-hand side of any rule, then for every string  $x \in L(G)$  with  $|x| > b^{|\Gamma|+1}$ , every parse tree of  $x$  in  $G$  has a root-to-leaf path that repeats some variable  $A \in \Gamma$ .*

*Proof.* Let  $b$  be the most number of symbols on the right-hand side of any rule. Then parse trees of  $G$  have branching factor at most  $b$  (i.e., each node has at most  $b$  children). Any tree with branching factor  $b$  and height  $h$  has at most  $b^h$  leaves, so if the tree has  $> b^h$  leaves, then the tree has height  $> h$ .

Let  $p = b^{|\Gamma|+1}$  and let  $x \in L(G)$  such that  $|x| \geq p$ . Then any parse tree of  $x$  in  $G$  has height  $> |\Gamma| + 1$ , so the longest root-to-leaf path has  $> |\Gamma|$  non-leaf nodes. By the pigeonhole principle, some node must repeat a variable  $A \in \Gamma$ .  $\square$

**Theorem 7.3.2.** *The language  $\{0^n 1^n 2^n \mid n \in \mathbb{N}\}$  is not context free.*

*Proof.* Suppose for the sake of contradiction that the language is context-free, generated by CFG  $G = (\Gamma, \Sigma, \rho, S)$  with at most  $b$  symbols on the right-hand side of any rule. Let  $p = b^{|\Gamma|+1}$  and let  $w = 0^p 1^p 2^p$ . By Lemma 7.3.1, any parse tree of  $w$  in  $G$  must repeat a variable  $A$  on some root-to-leaf path. Let  $x$  and  $y$  respectively be the substrings of  $w$  on the leaves under the lower and upper occurrences of  $A$ , respectively. (Note that  $x$  is a substring of  $y$ .)

TODO: this is easier to visualize with a figure showing the parse tree

Then we can replace the subtree under the upper  $A$  (whose leaves are  $y$ ) with the smaller subtree (whose leaves are  $x$ ), and this remains a valid parse tree, whose leaves are the string  $z$ .

We have three cases:

**$y$  has no 0's:** Then  $y$  also has no 0's, so by replacing  $y$  with  $x$ , we have decreased the number of 1's and/or 2's without decreasing the number of 0's, so  $z$  is not of the form  $0^n 1^n 2^n$  for any  $n \in \mathbb{N}$ . But since  $z$  has a parse tree in  $G$ ,  $z \in L(G)$ , a contradiction.

**$y$  has no 2's:** This is symmetric to the previous case.

**$y$  has both 0's and 2's:** There are three sub-cases:

**$x$  has no 0's:** Then replacing  $y$  with  $x$  decreases the number of 1's and/or 2's without decreasing the number of 0's.

**$x$  has no 2's:** This is symmetric to the previous sub-case.

**$x$  has both 0's and 2's:** Then  $x$  contains all the 1's in  $w$ . Thus, replacing  $y$  with  $x$  decreases the number of 0's and/or 2's without decreasing the number of 1's.  $\square$

Maybe just state formal pumping lemma at end, but do mostly examples. If a parse tree repeats a variable on a root-leaf path, then this can be pumped.



## Part II

# Computational Complexity and Computability Theory



## Chapter 8

# Turing machines

### 8.1 Intuitive idea of Turing machines

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

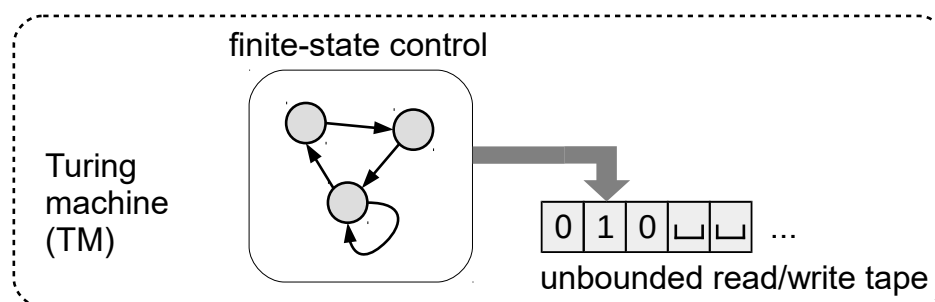


Figure 8.1: Intuitive idea of a Turing machine (TM).

One can think of a finite automaton as taking its input string from a “tape”, in which the finite automaton starts reading the leftmost tape cell, moves to the right by one tape cell each time step, and halts after moving off of the rightmost tape cell. From this perspective, the *differences* between finite automata and TMs are:

- A TM can *write* on its tape; furthermore, it can write symbols that are not part of the input alphabet (in particular, the *blank symbol*  $\sqcup$  is never part of the input alphabet, but appears on the tape).
- The read-write tape head can move right, but also can move left or stay still. This means it can read the same input symbol more than once.
- The tape is unbounded: if the tape head moves off the rightmost tape cell, a new tape cell appears, with a  $\sqcup$  on it.<sup>1</sup>

<sup>1</sup>It is common to describe a TM as having an *infinite* tape, which makes it appear unrealistic. But there is no

- Most states do not accept or reject; there is exactly one accept state and one reject state, and the TM immediately halts upon entering either state. Conversely, the TM will not halt until it reaches one of these states, which may never happen.

Recall that  $w^{\mathcal{R}}$  represents the reverse of  $w$ .

**Example 8.1.1.** Design a Turing machine to test membership in the palindrome language

$$P = \{ w \in \{0,1\}^* \mid w = w^{\mathcal{R}} \}.$$

1. Zig-zag to either side of the string, checking if the leftmost symbol equals the rightmost symbol. If not, *reject*.
2. “Cross off” symbols as they are checked (i.e., replace the symbol with a symbol **x** not in the input alphabet).
3. If we make it to the end without rejecting, *accept*.

This can be implemented in the Turing machine simulator with

```
// This TM recognizes the language { w in {0,1}* | w = w^R }
states = {s,r00,r11,r01,r10,l,lx,qA,qR}
input_alphabet = {0,1}
tape_alphabet_extra = {x,_}
start_state = s
accept_state = qA
reject_state = qR
num_tapes = 1
delta =
    s, 0 -> r00, x, R;
    s, 1 -> r11, x, R;
    r00, 0 -> r00, 0, R;
    r00, 1 -> r01, 1, R;
    r01, 0 -> r00, 0, R;
    r01, 1 -> r01, 1, R;
    r10, 0 -> r10, 0, R;
    r10, 1 -> r11, 1, R;
    r11, 0 -> r10, 0, R;
    r11, 1 -> r11, 1, R;
    r00, _ -> lx, _, L;
    r11, _ -> lx, _, L;
    r00, x -> lx, x, L;
```

---

need to posit anything infinite. Like a list in Python/Java or a vector in C++, the tape is finite, but it can always grow to a larger (yet still finite) size.



```

r11, x  -> lx,  x, L;
lx, 0   -> l,   x, L;
lx, 1   -> l,   x, L;
lx, x   -> qA,  x, S;
l, 0    -> l,   0, L;
l, 1    -> l,   1, L;
l, x    -> s,   x, R;
s, x    -> qA,  x, S;

```

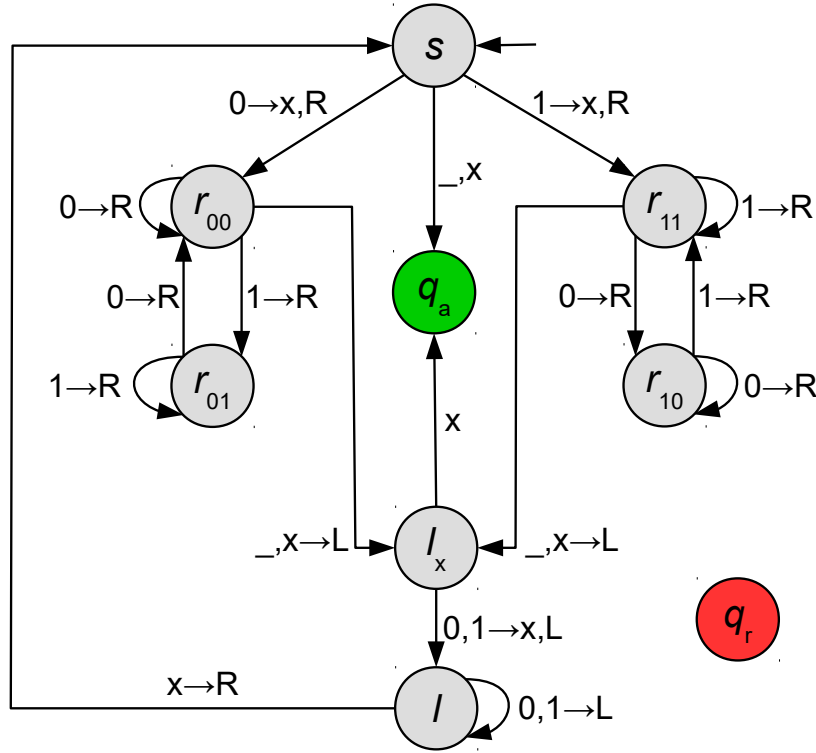


Figure 8.2: TM that decides  $\{w \in \{0,1\}^* \mid w = w^R\}$ . Although state  $q_r$  has no explicit incoming transition, every transition not shown (e.g., reading a  $\sqcup$  in state  $r_{10}$ ) implicitly goes to state  $q_r$ .

## 8.2 Formal definition of a TM (syntax)

**Definition 8.2.1.** A *Turing machine (TM)* is a 7-tuple  $(Q, \Sigma, \Gamma, \delta, s, q_a, q_r)$ , where

- $Q$  is a finite set of *states*,
- $\Sigma$  is the *input alphabet*,

- $\Gamma$  is the *tape alphabet*, where  $\Sigma \subsetneq \Gamma$ <sup>2</sup> and  $\sqcup \in \Gamma \setminus \Sigma$ ,
- $s \in Q$  the *start state*,
- $q_a \in Q$  the *accept state*,
- $q_r \in Q$  the *reject state*, where  $q_a \neq q_r$ , and
- $\delta : (Q \setminus \{q_a, q_r\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$  is the *transition function*.

**Example 8.2.2.** We formally define the TM  $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_a, q_r)$  described earlier, which decides the language  $P = \{w \in \{0, 1\}^* \mid w = w^R\}$ :

- $Q = \{s, r_{00}, r_{01}, r_{10}, r_{11}, l_x, l, q_a, q_r\}$ ,
- $\Sigma = \{0, 1\}$  and  $\Gamma = \{0, 1, x, \sqcup\}$ ,
- $s$  is the start state,  $q_a$  is the accept state,  $q_r$  is the reject state, and
- $\delta$  is shown in Figure 8.2.

end of lecture 5b

### 8.3 Formal definition of computation by a TM (infinite semantics)

This section shows a simple definition of TM semantics that uses an infinite tape. Section 8.4 has a more complex definition that shows TM don't actually require anything to be infinite: a finite tape suffices, but it requires many ugly special cases to define when to grow the tape.

A *configuration* of a TM  $M = (Q, \Sigma, \Gamma, \delta, s, q_a, q_r)$  is a triple  $C = (q, p, w)$ , where

- $q \in Q$  is the *current state*,
- $p \in \mathbb{N}^+$  is the *tape head position*,
- $w \in \Gamma^\infty$  (a one-way infinite sequence of symbols) is the *tape content*.

Let  $C = (q, p, w)$  and  $C' = (q', p', w')$  be configurations.

Informally, we want to talk about  $C'$  being the configuration that the TM will enter immediately after  $C$ .

We identify the moves L, R, S with integers  $-1, +1, 0$ , respectively. We say  $C$  *yields*  $C'$ , and we write  $C \rightarrow C'$ , if and only if  $\delta(q, w[p]) = (q', w'[p], m)$ , where

<sup>2</sup>Since  $\Sigma \subset \Gamma$ , in the simulator, rather than writing down all of  $\Gamma$ , which would be redundant, you only write  $\Gamma \setminus \Sigma$ , i.e., the symbols in the tape alphabet that are not in the input alphabet, calling this `tape_alphabet_extra`. The union of `tape_alphabet_extra` and  $\Sigma = \text{input\_alphabet}$  is the tape alphabet  $\Gamma$ . An underscore `_` represents  $\sqcup$ .

- $p' = \max(1, p + m)$  (move tape head position, but don't move left if it's already 1)
- $w[i] = w'[i]$  for all  $i \in \mathbb{N}^+ \setminus \{p\}$  (tape unchanged away from the tape head)

A configuration  $(q, p, w)$  is *accepting* if  $q = q_a$ , *rejecting* if  $q = q_r$ , and *halting* if it is accepting or rejecting.  $M$  *accepts* (respectively, *rejects*) input  $x \in \Sigma^*$  if there is a finite sequence of configurations  $C_1, C_2, \dots, C_k$  such that

1.  $C_k$  is accepting (respectively, rejecting),
2.  $C_1 = (s, 1, x\sqcup^\infty)$  (initial/start configuration: the TM starts with its input written at the leftmost part on the tape, with  $\sqcup$  written everywhere else.),
3. for all  $i \in \{1, \dots, k-1\}$ ,  $C_i \rightarrow C_{i+1}$ .

If  $M$  neither accepts nor rejects  $x$ , then we say  $M$  *loops* (a.k.a., *does not halt*) on  $x$ .

## 8.4 Optional: Formal definition of computation by a TM (finite semantics)

A *configuration* of a TM  $M = (Q, \Sigma, \Gamma, \delta, s, q_a, q_r)$  is a triple  $C = (q, p, w)$ , where

- $q \in Q$  is the *current state*,
- $p \in \mathbb{N}^+$  is the *tape head position*,
- $w \in \Gamma^*$  is the *tape content*, 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.

Let  $C = (q, p, w)$  and  $C' = (q', p', w')$  be configurations.

Informally, we want to talk about  $C'$  being the configuration that the TM will enter immediately after  $C$ .

We identify the moves L, R, S with integers  $-1, +1, 0$ , respectively. We say  $C$  *yields*  $C'$ , and we write  $C \rightarrow C'$ , if and only if  $\delta(q, w[p]) = (q', w'[p], m)$ , where

- $p' = \max(1, p + m)$  (move tape head position, but don't move left if it's already 1)
- $w[i] = w'[i]$  for all  $i \in \{1, \dots, |w|\} \setminus \{p\}$  (tape unchanged away from the tape head)
- if  $m = \text{L}$  or  $\text{S}$ , then  $|w'| = |w|$  (no need to grow the tape on a left/stay move)
- if  $m = \text{R}$ , then  $|w'| = |w|$  if  $p' < |w|$  (don't grow the tape if tape head was not already on rightmost cell), otherwise  $|w'| = |w| + 1$  and  $w'[p' - 1] = \sqcup$ . (grow the tape and put a  $\sqcup$  in the new position)

A configuration  $(q, p, w)$  is *accepting* if  $q = q_a$ , *rejecting* if  $q = q_r$ , and *halting* if it is accepting or rejecting.  $M$  *accepts* (respectively, *rejects*) input  $x \in \Sigma^*$  if there is a finite sequence of configurations  $C_1, C_2, \dots, C_k$  such that

1.  $C_k$  is accepting (respectively, rejecting),
2.  $C_1 = (s, 1, x)$  if  $x \neq \varepsilon$  and  $C_1 = (s, 0, \sqcup)$  if  $x = \varepsilon$  (initial/start configuration: the TM starts with only its input on the tape, or  $\sqcup$  if input is empty.),
3. for all  $i \in \{1, \dots, k-1\}$ ,  $C_i \rightarrow C_{i+1}$ .

If  $M$  neither accepts nor rejects  $x$ , then we say  $M$  *loops* (a.k.a., *does not halt*) on  $x$ .

## 8.5 Languages recognized/decided by TMs

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

**Definition 8.5.1.** A language is *Turing-recognizable* (a.k.a., *Turing-acceptable*, *computably enumerable*, *recursively enumerable*, *c.e.*, or *r.e.*) if some TM recognizes it.

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

On any input, a TM may accept, reject, or loop (run forever without entering a halting state).

If a language  $L$  is Turing-recognizable, then some Turing machine  $M$  exists so that, for any  $x \in L$ ,  $M$  accepts  $x$ , and for any  $x \notin L$ ,  $M$  either rejects  $x$  or does not halt on input  $x$ . Of course, the last is no good if we want to use  $M$  to actually solve a problem. If  $M$  halts on every input string, we say it is *total*.<sup>3</sup> A total TM  $M$  is also called a *decider*, and we say it *decides* the language  $L(M)$ .

**Definition 8.5.2.** A language is called *Turing-decidable* (*recursive*), or simply *decidable*, if some TM decides it.

We take the formal statement “a problem is decidable” to coincide with the intuitive notion of “the problem is solvable by some algorithm”.

One might ask why we introduce the concept of Turing-recognizability, when Turing-decidability is what we really want from algorithms (they always halt and give an answer). There are a few reasons. First, there is a lot of deep mathematical structure in the notions of Turing-recognizable and co-Turing-recognizable, which has attracted the attention of a lot of mathematicians and logicians who find similarities with classical notions of mathematical logic.

---

<sup>3</sup>The terminology *total* comes from the fact that a function defined on every input in its domain is called a *total function*. We can think of the decision problem solved by any model of computation equivalently as a predicate: a function with range  $\{0, 1\}$   $\phi : \Sigma^* \rightarrow \{0, 1\}$ . If a TM  $M$  halts on all inputs, then  $\phi$  is defined on all inputs, so it is a total function.  $\phi$  is a *partial* function if  $M$  does not halt on certain inputs, since for those inputs,  $\phi$  is undefined.

Another important reason is this: Many decision problems we study later will be of the form, “*given a Turing machine  $M$ , does  $L(M)$  have some property?*” (e.g., is nonempty, is infinite, etc.) Every DFA, regex, NFA, and CFG has some unique language that it decides. With Turing machines, this is true of recognizing but not deciding. That is, every Turing machine  $M$  defines a unique language  $L(M)$  that it recognizes. But if  $M$  is not total, it does not decide any language. So Turing-recognizability gives us a way of making a one-to-one equivalence between machines and languages, in a way that Turing decidability does not. However, in the case of TMs that always halt, the TM decides the same language  $L(M)$  that it recognizes, so asking the question about  $L(M)$  is not a restriction. It simply gives a way for the question to be well-defined on all possible TMs, rather than only on those TMs that always halt.

end of lecture 5c

## 8.6 Variants of TMs

Why do we believe Turing machines are an appropriate 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.<sup>4</sup>

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.

### 8.6.1 Multitape TMs

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.

---

<sup>4</sup>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 deterministic pushdown automaton, which cannot even recognize languages such as  $\{x \mid x = x^R\}$ .

Conversely, allowing the start configuration to contain an arbitrary infinite sequence of symbols already written on the tape (instead of  $\sqcup$  on all but finitely many positions) 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 other words, it would not be the *machine* with the powerful computational ability, but the *programmer* of the machine who is assumed to be able to write down an uncomputable infinite sequence of symbols.

With a single-tape TM, there is rarely any reason to let a tape head stay where it is: a sequence of multiple state transitions that occur while the tape head stays where it is could be replaced with a single transition to the final state in the sequence. However, with multiple tapes, it is often convenient to move only some tape heads but let others stay put, so we make more use of the **S** move. Such a machine's transition function is of the form

$$\delta : (Q \setminus \{q_a, q_r\}) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k$$

For example, the expression

$$\delta(q_i, a_1, a_2, a_3) = (q_j, b_1, b_2, b_3, S, R, L)$$

means that, if the 3-tape TM is in state  $q_i$  and heads 1 through 3 are reading symbols  $a_1, a_2, a_3$ , the machine goes to state  $q_j$ , writes symbols  $b_1, b_2, b_3$ , and directs the first head to stay, the second to move right, and the third to move left.

**Theorem 8.6.1.** *For every multitape TM  $M$ , there is a single-tape TM  $S$  such that  $L(M) = L(S)$ . Furthermore, if  $M$  is total, then  $S$  is total.*

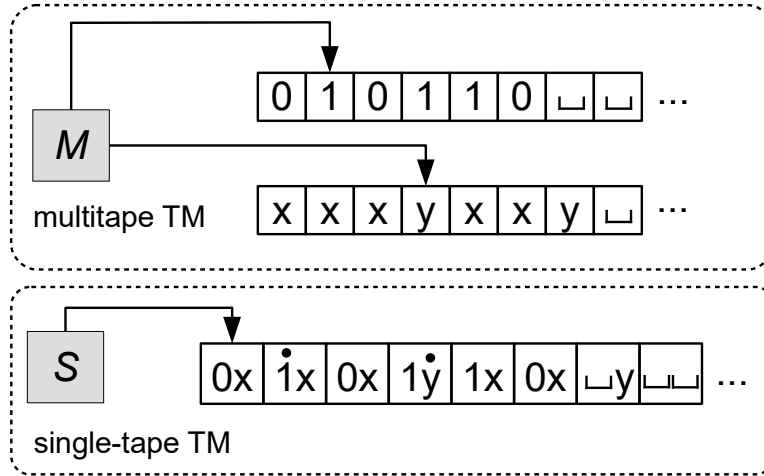


Figure 8.3: Single-tape TM simulating a multitape TM. According to the formalization of  $S$ 's tape alphabet in the proof of Theorem 8.6.1, the symbols on  $S$ 's tape from left to right are  $((0, N), (x, N))$ ,  $((1, H), (x, N))$ ,  $((0, N), (x, N))$ ,  $((1, N), (y, H))$ ,  $((1, N), (x, N))$ ,  $((0, N), (x, N))$ ,  $((\sqcup, N), (y, N))$ ,  $((\sqcup, N), (\sqcup, N))$ , but we write them in the figure in an easier-to-read fashion as a string of symbols with possible dots indicating the presence of one of  $M$ 's tape heads.

*Proof.* The main idea needed for the proof is to describe how any possible configuration  $C_M$  of  $M$  can be represented as a configuration  $C_S$  of  $S$ . Once that is settled, the proof is nearly complete, since we need only convince ourselves that it is possible for  $S$  to update  $C_S$  to represent the next configuration of  $M$ , and also how  $S$  can alter its initial configuration so

that it represents the initial configuration of  $M$ . However, there will not be a 1-1 correspondence of configurations between  $M$  and  $S$ :  $S$  will take many transitions to simulate a single transition of  $M$ , so many consecutive configurations of  $S$  will represent the same configuration of  $M$ .

The idea is shown in Figure 8.3. Say that  $M$  has  $k$  tapes. Each symbol in the tape alphabet of  $S$  actually represents  $k$  symbols from the tape alphabet of  $M$ , as well as  $k$  bits representing “is the  $i$ ’th tape head of  $M$  located here?”, which will be true for exactly one cell. Formally, if  $M$  has tape alphabet  $\Gamma_M$ , we could say  $S$  has tape alphabet  $\Gamma_S = (\Gamma_M \times \{H, N\})^k$  (where  $H$  means “head” and  $N$  means “no head”), so that a symbol  $((a_1, b_1), (a_2, b_2), \dots, (a_k, b_k)) \in \Gamma_S$  on the  $j$ ’th tape cell of  $S$  means that the  $j$ ’th tape cell of the  $i$ ’th tape of  $M$  has symbol  $a_i$ , with  $b_i = H$  if the head of the  $i$ ’th tape is at position  $j$  and  $b_i = N$  otherwise. Figure 8.3 shows an easier-to-read graphical representation, where each symbol of  $\Gamma_S$  is displayed as a length- $k$  string of symbols from  $\Gamma_M$ , with a dot over any symbol being scanned by a tape head of  $M$ .

On input  $w = w[1]w[2] \dots w[n]$ ,  $S$  begins by replacing its input string  $w[1]w[2] \dots w[n]$  with the symbol

$$w[1] \underbrace{\begin{array}{c} \bullet \\ \sqcup \sqcup \dots \sqcup \end{array}}_{k-1}$$

on the first tape cell, and

$$w[i] \underbrace{\begin{array}{c} \sqcup \sqcup \dots \sqcup \end{array}}_{k-1}$$

on the  $i$ ’th tape cell, for  $i \in \{2, \dots, n\}$ .

Whenever a single transition of  $M$  occurs,  $S$  must scan the entire tape, using its state to store all the symbols under each  $\bullet$  (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  $S$  since there are a fixed number of tapes and therefore a fixed number of  $\bullet$ ’s. Then,  $S$  will reset its tape to the start, moving each  $\bullet$  appropriately (either left, right, or stay), and writing new symbols on the tape at the old location of the  $\bullet$ , according to  $M$ ’s transition function.

This allows  $S$  to represent the configuration of  $M$  and to simulate the computation done by  $M$ . Note that  $S$  halts if and only if  $M$  halts, implying that  $S$  is total if  $M$  is total.  $\square$

### 8.6.2 Other variants

One can make all manner of other changes to the Turing machine model, without changing the fundamental computational power of the model:

- two-way infinite tapes instead of one-way infinite
- multiple tape heads on one tape
- 2D tape instead of 1D tape, i.e., the position is an integer  $(x, y)$ -coordinate, not simply a single integer

- random access memory: the machine has a special “address” worktape, where it can write an integer in binary, and then enter a special state, and the machine’s tape head on the main tape will immediately jump to that position without having to visit all the positions in between

The above modifications give *extra* capabilities to the machine. We can also consider certain ways of *reducing* the capabilities of a Turing machine. Surprisingly, all of the following modifications of the Turing machine definition do not alter the fundamental computational power of the model (all of them decide precisely the same class of languages as a Turing machine):

- move-right-or-reset: instead of a single left move, the tape head can be reset to the leftmost position.
- two stacks: instead of a linked list memory (which is a single list that the finite state machine can iterate over in both directions), the finite state machine has two *stacks*: lists in which only the rightmost element can be accessed, and the only way to change the stack is to *pop* (remove) a symbol from the rightmost end, or to *push* a new symbol to the rightmost end.
- one queue: instead of a linked list memory, the finite state machine can control a queue: in which only the rightmost symbol can be read and removed, and the only way to add a new symbol is to push it on the *leftmost* end.
- three counters: instead of a linked list memory, the finite state machine can control three *counters*: each counter is a nonnegative integer whose value can be incremented, decremented, or tested to check whether it is equal to 0. In this case there is no longer any notion of an alphabet of input or tape symbols: the only way to give input to the machine is to put a certain integer in one of its counters initially, and the only information the finite state machine has available, other than its current state, is three bits indicating whether each of its three counters is currently equal to 0 or not. Nonetheless, we can still make sense of doing computation on integers, for example, deciding whether a integer is prime, or replacing an initial  $n$  in a counter with the value  $f(n) = n^2$  or some other computable function of  $n$ . Any surprisingly, any function on integers that a TM can compute, can also be computed by a three-counter machine.

There are, however, ways to reduce the capability of a Turing machine sufficiently that it actually changes the computational power:

- one stack: this is equivalent to a deterministic pushdown automaton, which can decide only context-free languages (and not even all of those)
- two counters: there is a certain clever way that inputs can be encoded so that Turing-powerful computation can be done with only two counters; nonetheless, these machines are less powerful in the strict sense than three-counter machines. For example, although



a three-counter machine can start with a natural number  $n$  in its first counter and halt with the value  $2^n$  in its first counter (i.e., it can compute the function  $f(n) = 2^n$ ), no two-counter machine can do this.

One counter is even less powerful: a single counter is like a stack with a unary alphabet, so it's no more powerful than a one-stack machine! A one-counter machine *can* decide some nonregular languages such as  $\{0^n 1^n \mid n \in \mathbb{N}\}$ , but not something like  $\{w : w^R \mid w \in \{0, 1\}^*\}$ , even though a binary stack suffices for the latter.

- no memory: this is just a finite-state machine, which can decide only regular languages!

## 8.7 TMs versus code

Some of the homework problems involve programming multitape Turing machines. A goal of these exercise is to instill a sense that, although Turing machines look different than popular programming languages, they are in fact equally as powerful, if tedious to program.

From this point on, we will describe Turing machines in terms of algorithms in actual code, in the programming language Python. We know that if we can write an algorithm in Python 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 with another model of computation, it is easier to simulate a Turing machine than a Python program.

A colleague once said 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 except as a mathematical exercise; 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 rarely a reason to use Turing machines instead of pseudocode or a regular programming language.

**Structured data and flat strings.** 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  $\Sigma$  before delivering them to a TM). Given any “discrete” object  $O$ , such as a string, graph, tuple, (or even a Turing machine itself), 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  $\langle O_1, O_2, \dots, O_k \rangle$  to denote the encoding of the objects  $O_1$  through  $O_k$  as a single string.

For example, to encode the graph  $G = (V, E)$ , where  $V = \{v_1, v_2, v_3, v_4\}$  and

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}\},$$

we can use an adjacency matrix

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

encoded as a binary string by concatenating the rows:  $\langle G \rangle = 0110101011010010$ .

## 8.8 Optional: The Church-Turing Thesis

- 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.<sup>5</sup> 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.<sup>6</sup>
- 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  $\lambda$ -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.<sup>7</sup>
- 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  $\alpha$ -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.

---

<sup>5</sup>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.

<sup>6</sup>Lest that would provide a proof the truth or falsehood of the statement.

<sup>7</sup>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.)

- 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  $\lambda$ -calculus.
- Turing's paper is accepted, and researchers in the field – Church included – were immediately convinced by Turing's *physical* arguments: he essentially argued that his model was powerful enough to capture anything any *person* could do as they are sitting at a desk with a pencil and paper, calculating according to fixed instructions.

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

The statement known as the Church-Turing thesis is not a mathematical statement that can be formalized in the same way as a theorem. It is a *physical law*, much like the Laws of Thermodynamics or Maxwell's equations. It is something observed to hold in practice, but in principle it is refutable. In practice, however, attempts to refute it (a crank field known as “hypercomputation”) tend to fail, in the same way that attempts to build a perpetual motion machine (violating the second law of thermodynamics) inevitably fail.

end of lecture 6a

midterm exam

end of lecture 6b



## Chapter 9

# Efficient solution of problems: The class P

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

### 9.1 Asymptotic analysis

Our first concept involves a question: how do we measure the running time of an algorithm? Faster algorithms are better, we can agree, but what does it mean to say one algorithm is faster than another?

**“Clock-on-the-wall” time.** If I run algorithms  $A$  and  $B$  on the same input, and  $A$  gives an answer in 3 seconds, but  $B$  gives an answer in 4 seconds, does that mean  $A$  is better? What if I run them again, and this time  $A$  takes 5 seconds instead? Perhaps the times are different because other processes were running on my computer the second time, making  $A$  appear slower. What if I run  $A$  on an older, slower computer and  $A$  now takes 30 seconds? This way of measuring algorithm efficiency, informally known as “clock-on-the-wall” time, is inherently flawed, because an execution of an algorithm may take different amounts of real time depending on external environmental factors, such as other programs running on the same computer, or running the algorithm on a different computer. However, we want to understand whether the *algorithm itself* is faster than another algorithm. What we really want to know is: if we run two algorithms *in the exact same environment*, which will be faster?

To do this we need a “standard” environment for comparison. This is one elegant usage of the model of Turing machines: they give a simple and clean mathematical way to measure the “number of steps” needed for an algorithm, uncorrupted by complicating environmental factors such as processor clock frequencies or competing operating system processes.

**Definition 9.1.1.** Let  $M$  be a TM, and  $x \in \{0, 1\}^*$ . Define  $\text{time}_M(x)$  to be the number of configurations  $M$  visits on input  $x$  (so a TM that immediately halts takes 1 step, not 0).

This resolves one issue, that of standardizing the definition of “time”. However, note that  $\text{time}_M(x)$  has two variables in it: the Turing machine  $M$  and the input  $x$ . We said we want to measure the running time of  $M$ , but instead we have measured the running time of  $M$  on a particular input  $x$ . Suppose we have two Turing machines  $A$  and  $B$  solving some problem, and we want to determine which is faster. What if for two inputs  $x$  and  $y$ ,  $\text{time}_A(x) < \text{time}_B(x)$  but  $\text{time}_A(y) > \text{time}_B(y)$ , i.e.,  $A$  is faster than  $B$  on  $x$  but slower than  $B$  on  $y$ ? Which algorithm is “better”?

The way we resolve this issue (and it is not the only way to resolve it, but it is a standard simplifying approach) is to ask not merely how fast an algorithm runs on one input, but to ask how quickly its running time *grows* as we increase the *size* of the inputs.

There’s two important points we’ve just made. First, we consider only the *size* of the input and not the exact symbols of the input. For each possible input length  $n$ , we consider the *worst-case* running time over all inputs of length  $n$ . Second, rather than having a single number called the “running time”, we will have a *function*  $t : \mathbb{N} \rightarrow \mathbb{N}$  of  $n$ . Given any possible input length  $n$ ,  $t(n)$  is the most time the algorithm takes on any input of length  $n$ .

**Definition 9.1.2.** If  $M$  is total, define the (*worst-case*) *running time* (or *time complexity*) of  $M$  to be the function  $t : \mathbb{N} \rightarrow \mathbb{N}$  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*.

So we can assign to each total Turing machine  $M$  a unique function  $t$  called its *running time*. But now we have a new problem. We know how to compare two numbers to say which is larger. But how to compare two functions? What if  $t_1(n) > t_2(n)$  but  $t_1(n+1) < t_2(n+1)$ ?

**Asymptotic Analysis.** To resolve this problem, we use a tool called *asymptotic analysis*. This lets us compare two running time functions and say whether the *growth rate* of one is larger than the growth rate of the other, or whether they have the same growth rate.<sup>1</sup>

The main way we compare such functions involves *ignoring multiplicative constants*. This will have two effects:

1. some functions will be equivalent but not equal (different functions can have the same growth rate), and
2. it will greatly simplify analysis of algorithms.

By giving ourselves permission to ignore constants, we will be able to quickly glance at an algorithm and determine its growth rate, without worrying about the precise number of steps it will take.

---

<sup>1</sup>A word of warning: the word “rate” here is a metaphor. There’s no number that is the rate of growth.

**Definition 9.1.3.** Given  $f, g : \mathbb{N} \rightarrow \mathbb{N}^+$ , 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  $f$ .<sup>2</sup>

Bounds of the form  $n^c$  for some constant  $c$  are called *polynomial bounds*. ( $n$ ,  $n^2$ ,  $n^3$ ,  $n^{2.2}$ ,  $n^{1000}$ , etc.) So when we say a function  $f$  is *polynomially bounded*, this means that there exists a  $c$  such that  $f = O(n^c)$ .

Bounds of the form  $2^{n^\delta}$  for some real constant  $\delta > 0$  are called *exponential bounds*. ( $2^n$ ,  $2^{2n}$ ,  $2^{100n}$ ,  $2^{0.01n}$ ,  $2^{n^2}$ ,  $2^{n^{100}}$ ,  $2^{\sqrt{n}}$ , etc.)

So-called “big-O” notation, saying that  $f = O(g)$ , is analogous to saying that  $f$  is “at most” (or “grows no faster than”)  $g$ . “Little-O” notation is a way to capture that  $f$  is “strictly smaller” (or “grows slower than”)  $g$ .

**Definition 9.1.4.** Given  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ , we write  $f = o(g)$  (or  $f(n) = o(g(n))$ ), if  $f = O(g)$  and  $g \neq O(f)$ .

It is straightforward to prove that  $f = o(g)$  if and only if

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

The formal definitions above are “official”, but some shortcuts are handy to remember, and in practice, you will use these shortcuts far more often than you will need to resort to using the above definitions directly.

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  $h(n)$  that grows unboundedly. For example, suppose we want to compare  $n$  and  $n \log n$ : letting  $f(n) = n$ ,  $g(n) = n \log n$ , and  $h(n) = \log n$ , since  $f(n) = g(n) \cdot h(n)$ , and  $h(n)$  grows unboundedly, then  $f(n) = o(g(n))$ , i.e.,  $n = o(n \log n)$ .

Another useful shortcut is to ignore all but the largest terms, and to remove all constants. For example,  $10n^7 + 100n^4 + n^2 + 10n = O(n^7)$ , and  $2^n + n^{100} + 2^n = O(2^n)$ . This is useful when analyzing an algorithm, where you are counting steps from many portions of the algorithm, but only the “slowest part” (contributing the most number of steps) really makes a difference in the final analysis.

Finally, taking the log of anything, or taking the square root of anything (more generally, raising it to a power smaller than 1) makes it strictly smaller, and raising it to a power larger than 1 makes it bigger. So  $\log n = o(n)$ , and  $\sqrt{n} = o(n)$ . Also,  $\log n^4 = o(n^4)$  (actually,  $\log n^4 = 4 \log n = O(\log n)$ ).

The following are used often enough that they are worth memorizing:

- $1 = o(\log n)$ ,
- $\log n = o(\sqrt{n})$  (or more generally  $\log n = o(n^\alpha)$  for any  $\alpha > 0$ , e.g.,  $n^{1/100}$ ),

<sup>2</sup>Often one sees the slightly more complex definition that  $f = O(g)$  if there exists  $c, n_0 \in \mathbb{N}$  such that, for all  $n \geq n_0$ ,  $f(n) \leq c \cdot g(n)$ . (i.e., we require that  $c \cdot g(n)$  exceed  $f$  only on *sufficiently large*  $n$ , instead of *all*  $n$ ). Why are these equivalent? **Hint:**  $f$  and  $g$  are both positive.

- $\sqrt{n} = o(n)$ ,
- $n^c = o(n^k)$  if  $c < k$  For example,  $n = o(n^2)$ ,  $n^2 = o(n^3)$ , and  $n^3 = o(n^{3.01})$ .  $\sqrt{n} = o(n)$  is the special case for  $c = 0.5, k = 1$ .
- $n^k = o(2^{\delta n})$  for any  $k > 0$  and any  $\delta > 0$  (even if  $k$  is huge and  $\delta$  is tiny, e.g.,  $n^{100} = o(2^{0.001n})$ .)

A useful shortcut for comparing two time bounds to see if they have *different* growth rates is to take the log of each. If  $\log f(n) = o(\log g(n))$ , then  $f(n) = o(g(n))$ . This is useful for simplifying expressions. For example, how to compare  $2^{n^2}$  and  $n^n$ ? One has a larger exponent, and the other a larger base, so it's not obvious which grows faster. But taking the log of both, we get  $n^2$  and  $\log n^n = n \log n$ . Since  $n \log n = o(n^2)$ , we know immediately that  $n^n = o(2^{n^2})$ . However, this doesn't work to go the other way: just because  $\log f = \Theta(\log g)$  doesn't imply that  $f = \Theta(g)$ . For example,  $n = \Theta(2n)$ , but  $2^n = o(2^{2n})$ , since  $2^{2n} = (2^n)^2$ .

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

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

Based on this analogy, we introduce notations that are analogous to the three other comparison operators  $\geq$ ,  $>$ , and  $=$ :

- $f = \Omega(g)$  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  $f$  “ $=$ ”  $g$ ; note it is *not* the same as saying they are actually the same function! For example,  $n^2 = \Theta(3n^2)$  but  $n^2 \neq 3n^2$ ; it's saying they *grow* at the same asymptotic rate.)<sup>3</sup>

### 9.1.1 Analyzing algorithms

A major benefit of big- $O$  notation is to make it easier to analyze the complexity of algorithms. Rather than counting the exact number of steps, which would be tedious, we can “throw away constants”. It is often much easier to quickly glance at an algorithm and get a rough estimate of its running time in big- $O$  notation, compared to the much more arduous task of attempting to compute the precise number of steps the algorithms takes on a certain input.

**Definition 9.1.5.** Let  $t : \mathbb{N} \rightarrow \mathbb{N}$  be a time bound.<sup>4</sup> Define  $\text{TIME}(t) \subseteq \mathcal{P}(\{0, 1\}^*)$  to be the

<sup>3</sup>Obviously,  $f = \Theta(g)$  implies that  $f = O(g)$ . But it is very common to see people write  $f = O(g)$  when they really mean  $f = \Theta(g)$ . They are not equivalent, however:  $n = O(n^2)$  but  $n \neq \Theta(n^2)$ .

<sup>4</sup>We will prove many time complexity results that actually do not hold for all functions  $t : \mathbb{N} \rightarrow \mathbb{N}$ . Many results hold only for time bounds that are what is known as *time-constructible*, which means, briefly, that  $t : \mathbb{N} \rightarrow \mathbb{N}$  has the property that the related function  $f_t : \{0\}^* \rightarrow \{0\}^*$  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.



set of decision problems

$$\text{TIME}(t) = \{ L \subseteq \{0,1\}^* \mid \text{there is a TM with running time } O(t) \text{ that decides } L \}.$$

**Observation 9.1.6.** *For all time bounds  $t_1, t_2 : \mathbb{N} \rightarrow \mathbb{N}$  such that  $t_1 = O(t_2)$ ,  $\text{TIME}(t_1) \subseteq \text{TIME}(t_2)$ .*

The following result is the basis for all complexity theory. We will not prove it in this course (take 220 for that), 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 says that given more time, one can decide more languages.

**Theorem 9.1.7** (Deterministic Time Hierarchy Theorem). *Let  $t_1, t_2 : \mathbb{N} \rightarrow \mathbb{N}$  be time bounds such that  $t_1(n) \log t_1(n) = o(t_2(n))$ . Then  $\text{TIME}(t_1) \subsetneq \text{TIME}(t_2)$ .*

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.

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 ECS 36C and ECS 122A, that individual lines of code in C++, Python, or pseudocode take constant time, so long as those lines are not masking loops or recursion or something that would not take constant time (such as a Python list comprehension, which is shorthand for a loop).

### 9.1.2 Optional: Time complexity of simulation of multitape TM with a one-tape TM

Recall the single-tape TM from Section 8.1, deciding whether its input is a palindrome. What is its running time? Recall that it moves the tape head from the left side of the string to the right side, repeatedly, although it replaces the leftmost and rightmost with an  $x$  and only moves over the bits that have not been replaced. So each bit is visited twice each time the tape head makes a pass. Once the bit is replaced with  $x$ , that tape cell is only visited once more. So the  $i$ 'th bit in the left half is then visited  $2i + 1$  times. The time spent in the left half is  $\sum_{i=0}^{n/2} (2i + 1) = n/2 + 1 + 2 \cdot \sum_{i=0}^{n/2} i = n/2 + 1 + 2 \cdot 1/2 \cdot n/2 \cdot (n/2 + 1) = O(n^2)$ . By symmetry, the time spent in the right half of the input is the same, so the whole running time is  $O(n^2)$ .

Observe that a 2-tape TM can decide the palindrome language in  $O(n)$  time: it just needs to copy the input to the second tape in  $n$  steps, put tape head 1 on the left and tape head 2 on the right on  $n$  steps, and then move them in opposite directions for  $n$  steps, comparing

---

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  $\text{TIME}(t(n)) = \text{TIME}\left(2^{2^{t(n)}}\right)$ . 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 symbols they read. Recall also that we showed every multitape TM can be simulated by a single-tape TM. Why does this not prove that single-tape TMs can decide the palindrome language in  $O(n)$  time? It's because that simulation incurs a quadratic slowdown, as the next theorem shows.

**Theorem 9.1.8.** *Let  $t : \mathbb{N} \rightarrow \mathbb{N}$ , 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 the proof of Theorem 8.6.1.  $M$ 's tape heads move right by at most  $t(n)$  positions, so  $S$ 's tape contents have length at most  $t(n)$ . Simulating one step of  $M$  requires moving  $S$ 's tape by this length and back, which is  $O(t(n))$  time. Since  $M$  takes  $t(n)$  steps total, and  $S$  takes  $O(t(n))$  steps to simulate each step of  $M$ ,  $S$  takes  $O(t(n)^2)$  total steps.  $\square$

Note that the single-tape TM  $S$  is a little slower: it takes  $O(t(n)^2)$  to simulate  $t(n)$  steps of  $M$ . Could this be improved to  $O(t(n))$ ? It turns out the answer is no: for example, the palindrome language  $\{w \mid w = w^R\}$  can be solved in time  $O(n)$  on a two-tape TM, but it provably requires  $\Omega(n^2)$  time on a one-tape TM.

The lesson is that although all reasonable models of computation have running time within a polynomial factor of each other, they sometimes are not within a *constant* factor of each other.

end of lecture 6c

## 9.2 Definition of P

### 9.2.1 Polynomial time

**Definition 9.2.1.**

$$P = \bigcup_{k=1}^{\infty} \text{TIME}(n^k).$$

In other words,  $P$  is the class of languages decidable in polynomial time on a deterministic, one-tape TM. But, for reasons we outline below, the easiest way to think of  $P$  is as the class of languages decidable using only a polynomial number of steps in your favorite programming language.

We must be careful to use reasonable encodings with the encoding function  $\langle \cdot \rangle : D \rightarrow \{0, 1\}^*$ , that maps elements of a discrete set  $D$  (such as  $\mathbb{N}$ ,  $\Sigma^*$ , 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 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  $V = \{0, 1, \dots, n-1\}$ , is via its adjacency matrix, where for  $i, j \in \{1, \dots, n\}$ , 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.

To be very technical and precise, the “input size” is the number of bits needed to encode an input. For a graph  $G = (V, E)$ , this means  $|\langle G \rangle|$ , where  $\langle G \rangle \in \{0, 1\}^*$ . For an adjacency matrix encoding, for example, this would be exactly  $n^2$  if  $n = |V|$ . However, for many data structures there is another concept meant by “size”. For example, the “size” of  $G$  could refer to the number of nodes  $|V|$ .

The “size” of a list of strings could mean the number of strings in it, which would be *smaller* than the number of bits needed to write the whole list down. For example, the list  $(010, 1001, 11)$  has 3 elements, but they have 9 bits in total. Furthermore, there’s no obvious way to encode the list with only 9 bits; simply concatenating them to 010100111 could encode other lists, such as  $(0101001, 11)$  or  $(0, 1, 0, 1, 0, 0, 1, 1, 1)$ . So even more bits are needed to encode them in a way that they can be delimited; for instance, by “bit-doubling”: The list above can be encoded as 001100 01 11000011 01 1111, where bits 0 and 1 of the encoded string are represented as 00 and 11, respectively, and 01 is a delimiter to mark the boundary between strings. This increases the length of the string by at most factor 4 (the worst case is a list of single-bit strings).

This frees us to talk about the “size” of an input without worrying too much about whether we mean the number of bits in an encoding of the input, or whether we mean some more intuitive notion of size, such as the number of nodes or edges (or both) in a graph, the number of strings in a list of strings, etc.

In an algorithms course, we would care very much about the differences between these representations, because they could make a larger difference in the analysis of the running time. But in this course, we observe that all of these differences are *within a polynomial factor* of each other.<sup>5</sup> Therefore, if we can find an algorithm that is polynomial time under one encoding, it will be polynomial time under the others as well.

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

So to summarize, we choose to study the class P for two main reasons:

1. Historically, when an “efficient” algorithm is discovered for a problem, it tends to be polynomial time.

---

<sup>5</sup>To say that two time bounds  $t$  and  $s$  are “within a polynomial factor” of each other means that (assuming  $s$  is the smaller of the two)  $t = O(s)$ , or  $t = O(s^2)$ , or  $t = O(s^3)$ , etc. For example, all polynomial time bounds are within a polynomial factor of each other. Letting  $s(n) = 2^n$  and  $t(n) = 2^{3n}$ ,  $t$  and  $s$  are within a polynomial factor because  $t(n) = 2^{3n} = (2^n)^3 = O(s^3)$ .

2. By ignoring polynomial differences, we are able to cast aside detailed differences between models of computation and particular encodings. This gives us a general theory that applies to *all computers and programming languages*.

### 9.2.2 Examples of problems in P

**Paths in graphs.** Define

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

**Theorem 9.2.2.**  $\text{PATH} \in \text{P}$ .

*Proof.* Breadth-first search:

```

1 import collections
2 def path(G,s,t):
3     """G: graph
4         s,t: nodes in G
5     Check if there is a path from node s to node t in graph G."""
6     V,E = G
7     visited = []
8     queue = collections.deque([s])
9     while queue:
10         node = queue.popleft()
11         if node == t:
12             return True
13         if node not in visited:
14             visited.append(node)
15             node_neighbors = [ v for (u,v) in E if u==node ]
16             for neighbor in node_neighbors:
17                 if neighbor not in visited:
18                     queue.append(neighbor)
19     return False

```

How long does this take? Assume we have  $n$  nodes and  $m = O(n^2)$  edges. When a node comes out of `queue`, we add it to `visited` if it is not already in there. Thus the Boolean condition at line 13 is `True` at most once per node in  $V$ . It takes time  $O(n)$  to check whether a node is in the list `visited`. It takes time  $O(m)$  for line (15) to find all the neighbors of a given node. For each iteration of the outer loop, the inner loop at line (16) executes at most  $n$  times. Since this inner loop is executed at most once per node, at most  $n^2$  nodes are ever put in `queue`.

So the loop at line (9) executes  $O(n^2)$  iterations. Putting this all together, it takes time  $O(n^2 \cdot (n + m + n \cdot n)) = O(n^4)$ , which is polynomial time.  $\square$

We can test this out on a few inputs:

```

1 V = [1,2,3,4,5,6,7,8]
2 E = [ (1,2), (2,3), (3,4), (4,5), (6,7), (7,8), (8,6) ]
3 G = (V,E)
4 print("path from {} to {}? {}".format(4, 1, path(G, 4, 1)))
5 print("path from {} to {}? {}".format(1, 4, path(G, 1, 4)))

```

Let's try it on an undirected graph:

```

1 def make_undirected(G):
2     """G: directed graph
3     Makes graph undirected by adding reverse edges."""
4     V,E = G
5     new_edges = []
6     for (u,v) in E:
7         if (v,u) not in E and (v,u) not in new_edges:
8             new_edges.append((v,u))
9     return (V, E+new_edges)
10
11 G = make_undirected(G)
12 print("path from {} to {}? {}".format(4, 1, path(G, 4, 1)))
13 print("path from {} to {}? {}".format(1, 4, path(G, 1, 4)))
14 print("path from {} to {}? {}".format(4, 7, path(G, 4, 7)))

```

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.

Of course, we know from studying algorithms that with an appropriate graph representation, such as an adjacency list, and with a more careful analysis that doesn't make so many worst-case assumptions, this time can be reduced to  $O(n+m)$ . But in computational complexity theory, since we have decided in advance to consider any polynomial running time to be "efficient", we can often be quite lazy in our analysis and choice of data structures and still obtain a polynomial running time.

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

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

**Theorem 9.2.3.**  $\text{RELPRIME} \in \text{P}$ .

*Proof.* Since the input size is  $|\langle x, y \rangle| = O(\log x + \log y)$  (the number of *bits* needed to represent  $x$  and  $y$ ), we must be careful to use an algorithm that is polynomial in  $n = |\langle x, y \rangle|$ , *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.

```

1 def gcd(x,y):
2     """x,y: positive integers
3     Euclid's algorithm for greatest common divisor."""
4     while y>0:
5         x = x % y
6         x,y = y,x
7     return x
8
9 def rel_prime(x,y):
10    return gcd(x,y) == 1

```

```

11
12 print(gcd(24,60), rel_prime(24,60))
13 print(gcd(25,63), rel_prime(25,63))

```

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  $O(1)$  arithmetic operations, each polynomial-time computable, whence `rel_prime` is polynomial time.  $\square$

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.

**Connected graphs.** Recall that an undirected graph is *connected* if every node has a path to every other node. Define

$$\text{CONNECTED} = \{ \langle G \rangle \mid G \text{ is connected undirected graph} \}.$$

**Theorem 9.2.4.**  $\text{CONNECTED} \in \text{P}$ .

*Proof.*  $G = (V, E)$  is connected if, for all  $s, t \in V$ ,  $\langle G, s, t \rangle \in \text{PATH}$ :

```

1 from itertools import combinations as subsets
2 def connected(G):
3     """G: graph
4     Check if G is connected."""
5     V, E = G
6     for (s,t) in subsets(V,2):
7         if not path(G,s,t):
8             return False
9     return True

```

Let  $n = |V|$ . Since  $\text{PATH} \in \text{P}$ , and `connected` calls `path`  $O(n^2)$  times, `connected` takes polynomial time.  $\square$

**Eulerian paths in graphs.** Recall that a *Eulerian path* in a graph visits each edge exactly once. Define

$$\text{EULERIANPATH} = \{ \langle G \rangle \mid G \text{ is a directed graph with an Eulerian path} \}.$$

**Theorem 9.2.5.**  $\text{EULERIANPATH} \in \text{P}$

*Proof.* Euler showed that an undirected graph has an Eulerian path if and only if exactly zero or two vertices have odd degree, and all of its vertices with nonzero degree belong to a single connected component. Thus, we can check the degree of each node to verify the first part, and make a new graph of all the nodes with positive degree, ensuring it is connected, to check the second part.

```

1 def degree(node, E):
2     """node: node in a graph
3     E: set of edges in a graph
4     Return degree of node."""
5     return sum(1 for (u,v) in E if u==node)
6
7 def eulerian_path(G):
8     """G: graph
9     Check if G has an Eulerian path."""
10    V,E = G
11    num_deg_odd = 0
12    V_pos = [] # nodes with positive degree
13    for u in V:
14        deg = degree(u, E)
15        if deg % 2 == 1:
16            num_deg_odd += 1
17        if deg > 0:
18            V_pos.append(u)
19    if num_deg_odd not in [0,2]:
20        return False
21    G_pos = (V_pos,E)
22    return connected(G_pos)

```

Let  $n = |V|$  and  $m = |E|$ . The loop executes  $n = |V|$  iterations, each of which takes  $O(m)$  time to execute `degree`, which iterates over each of the  $m$  edges. So the loop takes  $O(nm)$  time, which is polynomial. It also calls `connected` on a potentially smaller graph than  $G$ , which we showed also takes polynomial time.  $\square$

### 9.2.3 Why identify P with “efficient”?

- 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 C++ program) in time  $p(t(n))$ , for some polynomial  $p$  (usually not much bigger than  $O(n^2)$  or  $O(n^3)$ ).
- Most “reasonable” encodings of inputs as binary strings are polynomially equivalent.
- 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 practice, even a constant factor can be significant; for example the fastest known algorithm for matrix multiplication take time  $O(n^{2.373})$ , but the  $O()$  hides a rather large constant factor. The naïve  $O(n^3)$  algorithm for matrix multiplication, properly implemented to take advantage of pipelining, vectorized instructions, and modern memory caches, tends to outperform the asymptotically faster algorithms even on rather large matrices. So this is a lesson: computational complexity theory should be the first place you check for an efficient algorithm for a problem, but it should not be the *last* place you check.

- 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 C++, 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,<sup>6</sup> it is usually brought down within a few years to polynomial running time with 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.
1. Although  $\text{TIME}(t)$  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*.<sup>7</sup>

end of lecture 7a

<sup>6</sup>Here,  $n \approx \log p$  is the *size of the input*, i.e., the number of bits to represent an integer  $p$  to be tested for primality.

<sup>7</sup>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.



## Chapter 10

# Efficient verification of solutions: The class NP

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

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

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

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 *verifying* whether a given path is a Hamiltonian path in a given graph, does have a polynomial-time algorithm:

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

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

```
1 def ham_path_verify(G, p):
2     """G: graph
3         p: list of nodes
4     Verify that p is a Hamiltonian path in G."""
5     V, E = G
6     # verify each pair of adjacent nodes in p shares an edge
7     for i in range(len(p) - 1):
8         if (p[i], p[i+1]) not in E:
9             return False
10    # verify p and V have same number of nodes
11    if len(p) != len(V):
12        return False
13    # verify each node appears at most once in p
14    if len(set(p)) != len(p):
```

```

15         return False
16     return True

```

Why is this polynomial-time? Let  $n = |V|$ . It executes a `for` loop for  $|p| - 1$  iterations, where  $|p| \leq n$ . each of which checks a pair of nodes for membership in  $E$ , which takes time  $O(|E|) = O(n^2)$ . The comparison of  $|p|$  to  $|V|$  takes  $O(1)$  time, and converting  $p$  from a list to a set in the final `if` statement takes time  $O(n \log n)$  for common set data structures.

Let's try it out on some candidate paths:

```

1  V = [1,2,3,4,5]
2  E = [ (1,2), (2,4), (4,3), (3,5), (3,1) ]
3  G = (V,E)
4
5  p_bad = [1,2,3,4,5]
6  print(ham_path_verify(G, p_bad))
7
8  p_bad2 = [1,2,4,3,1,2,4,3,5]
9  print(ham_path_verify(G, p_bad2))
10
11 p_bad3 = [1,2,4]
12 print(ham_path_verify(G, p_bad3))
13
14 p_good = [1,2,4,3,5]
15 print(ham_path_verify(G, p_good))

```

Another problem with a related polynomial-time verification language is

$$\text{COMPOSITES} = \{ \langle n \rangle \mid n \in \mathbb{N}^+ \text{ and } n = pq \text{ for some integers } p, q > 1 \},$$

with the related verification language

$$\text{COMPOSITES}_V = \{ \langle n, d \rangle \mid n, d \in \mathbb{N}^+, d \text{ divides } n, \text{ and } 1 < d < n \}.$$

Actually,  $\text{COMPOSITES} \in \text{P}$ , so sometimes a problem can be decided *and* verified in polynomial time. The algorithm deciding  $\text{COMPOSITES}$  in polynomial time is a bit more complex (not terribly so, see [https://en.wikipedia.org/wiki/AKS\\_primality\\_test#The\\_algorithm](https://en.wikipedia.org/wiki/AKS_primality_test#The_algorithm)). Here's the verifier, which makes sure that the potential divisor  $d$  1) is of the right size (strictly between 1 and  $n$ ), and 2) is actually a divisor (has remainder 0 when dividing  $n$  by  $d$ ):

```

1 def composite_verify(n,d):
2     """n,d: positive integers
3     Verify that d is a nontrivial divisor of n."""
4     if not 1 < d < n:
5         return False
6     return n % d == 0

```

The first check is  $O(n)$  time (each bit of the integers needs to get compared to check for  $1 < d < n$ ). The next check involving the remainder is  $O(n^2)$  time to do the division by the standard grade-school division algorithm.

Let's try it out on some integers:

```

1 print(composite_verify(15, 3))
2 for d in range(2,17): # 17 is prime so these should all be false
3     print(composite_verify(17, d), end=" ")

```

## 10.1 The class NP

### 10.1.1 Definition of NP

We now formalize these notions.

**Definition 10.1.1.** A *polynomial-time verifier* for a language  $A$  is an algorithm  $V$ , where there are constants  $c, k$  such that  $V$  runs in time  $O(n^c)$  and

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

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  $x \in A$ .

**Definition 10.1.2.** 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.

We note one technicality: sometimes we measure the running time of  $V$  as a function of  $|x|$ , and sometimes we measure it as a function of its entire input length, which is  $|\langle x, w \rangle|$ . Since  $|w|$  is bounded by a polynomial in  $|x|$ , one of these running times is a polynomial if and only if the other is. The exponent in the polynomial might change. For instance, if we have  $n = |x|$  and  $m = |\langle x, w \rangle| = n^3$ , and  $V(x, w)$  runs in time  $m^5$ , then it runs in time  $(n^3)^5 = n^{15}$ .

**Example 10.1.3.** For a graph  $G$  with a Hamiltonian path  $p$ ,  $\langle p \rangle$  is a witness testifying that  $G$  has a Hamiltonian path.

**Example 10.1.4.** For a composite integer  $n$  with a divisor  $1 < d < n$ ,  $\langle d \rangle$  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.

For instance,  $\text{HAMPATH}_V$  is a verification language for  $\text{HAMPATH}$ .

What, *intuitively*, is this class NP? And why should anyone care about it? It is a very natural notion, even if its mathematical definition looks unnatural.

In summary:

- P is the class of decision problems that can be *decided* quickly.
- NP is the class of decision problems that can be *verified* quickly, given a purported solution (the “witness”).

NP is, in fact, a very natural notion, because it is far more common than not that real problems we want to solve have this character that, whether or not we know how to find a solution efficiently, we *know what a correct solution looks like*, i.e., we can verify efficiently whether a purported solution is, in fact, correct.

For instance, it may not be obvious to me, given an encrypted file  $C$ , how to find the decrypted version of it. But if you give me the decrypted version  $P$  and the encryption key  $k$ , I can run the encryption algorithm  $E$  to verify that  $E(P, k) = C$ .

### 10.1.2 Decision vs. Search vs. Optimization

Even problems that don't look like decision problems (such as optimization problems) can be re-cast as decision problems, and when we do this, they tend also to have this feature of being efficiently checkable.

**Optimization.** Suppose I want to find the cheapest flights from Sacramento to Rome. This is an optimization problem. To cast this as a decision problem, we add a new input, a *threshold* value, and ask whether there are flights costing under the threshold: e.g., given two cities  $C_1$  and  $C_2$  and a budget of  $b$  dollars, we ask whether there is a sequence of flights from  $C_1$  to  $C_2$  costing at most  $b$  dollars. It may not be obvious to me how to find an airline route from Sacramento to Rome that costs less than \$1200. But if you give me a sequence of flights with their ticket prices, I can easily check that their total cost is  $\leq$  \$1200, that they start at Sacramento and end at Rome, and that the destination airport of each intermediate leg is the starting airport of the next leg.

**Search.** Not all problems in NP are variants of optimization problems. For example, the problem of determining whether a number is prime or not seems to be “inherently Boolean”: the number is prime or it isn't. Even looking at it like a search problem—given  $n$ , find the prime factorization of  $n$ —doesn't seem to involve any optimization. All integers have a unique prime factorization.

However, there is a sense in which *all* NP problems are search problems. Namely, if a problem is in NP, then it has a polynomial-time verifier that verifies purported witnesses  $w$  for instances  $x$ . The equivalent search problem is then: given  $x$ , find a witness  $w$ , or report that none exists.

For example for the decision problem of determining if a graph  $G$  has a Hamiltonian path, the equivalent search problem is to find a Hamiltonian path of  $G$  if it exists. For the Boolean satisfiability decision problem of determining whether a formula  $\phi$  has a satisfying assignment, the equivalent search problem is to find a satisfying assignment if it exists.

More generally, if a decision problem  $A$  is in NP, then it has a polynomial-time verifier  $V$  such that, for all  $x \in \{0, 1\}^*$ ,  $x \in A \iff (\exists w \in \{0, 1\}^{p(|x|)}) V(x, w)$  accepts. The equivalent search problem is, given  $x$ , find a  $w$  such that  $V(x, w)$  accepts.<sup>1</sup>

For whatever reason, this feature of “efficient checkability” occurs more often than not when it comes to natural decision problems that we care about solving. It seems that most “natural” problems can have their solutions verified efficiently (those problems are in NP). However, what we care about is efficiently finding solutions to problems (showing a problem is in P). The P versus NP question is essentially asking whether we are doomed to fail in some of the latter cases, i.e., whether there are problems whose solutions can be verified

---

<sup>1</sup>We won't prove the following in this course, but it is worth knowing: If  $P = NP$ , i.e., if every NP decision problem can be solved in polynomial time, then every NP *search* problem can also be solved in polynomial time. In other words, although the decision problem looks easier, in fact, the search problem of actually finding a witness is almost as easy as telling whether one exists.

efficiently (which is most of the problems we encounter “in the wild”), but those solutions *cannot* be found efficiently.

NP stands for *nondeterministic polynomial time*, **not** for “non-polynomial time”. In fact,  $P \subseteq NP$ , so some problems in NP are decidable in polynomial time, in addition to being merely verifiable in polynomial time. For instance, we can efficiently find a path  $p$  from  $s$  to  $t$  in a graph  $G$ , and we can also easily verify that  $p$  is actually a path from  $s$  to  $t$  in  $G$ . The name comes from a characterization of NP in terms of nondeterministic polynomial-time machines, which we discuss after the next few examples of more problems in NP.

end of lecture 7b

## 10.2 Examples of problems in NP

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

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

**Theorem 10.2.1.**  $\text{CLIQUE} \in \text{NP}$ .

*Proof.* The following is a verifier for CLIQUE, taking a set  $C$  of nodes as the witness:

```

1 import itertools
2 def clique_verifier(G, k, C):
3     """G: graph
4         k: positive integer
5         C: set of nodes in G.
6     Verify that C is a k-clique in G."""
7     V, E = G
8     # verify C is the correct size
9     if len(C) != k:
10         return False
11     # verify each pair of nodes in C shares an edge
12     for (u,v) in itertools.combinations(C, 2):
13         if (u,v) not in E:
14             return False
15     return True

```

The check for  $|C| = k$  takes  $O(1)$  time. The loop executes  $O(n^2)$  iterations, and each check for  $(u,v) \text{ not in } E$  takes time  $O(|E|) = O(n^2)$ . So the verifier is polynomial-time.  $\square$

Try it out on some examples:

```

1 V = [1,2,3,4,5,6]
2 E = [(1,2), (1,3), (1,4), (2,3), (2,4), (3,4), (4,5), (5,6), (4,6)]
3 G = (V,E)
4 G = make_undirected(G)
5

```

```

6 C = [1,2,3,4]
7 print("{} is {}-clique in G? {}".format(C, clique_v(G, len(C), C)))
8
9 C = [3,4,5]
10 print("{} is {}-clique in G? {}".format(C, clique_v(G, len(C), C)))

```

The SUBSETSUM problem is: given a collection (set with repetition) of integers  $x_1, \dots, x_k$  and a target integer  $t$ , is there a subcollection that sums to  $t$ ?

$$\text{SUBSETSUM} = \left\{ \langle C, t \rangle \mid \begin{array}{l} C = \{x_1, \dots, x_k\} \text{ is a collection of} \\ \text{integers and } (\exists S \subseteq C) t = \sum_{y \in S} y \end{array} \right\}.$$

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

**Theorem 10.2.2.** SUBSETSUM  $\in$  NP.

*Proof.* The following is a verifier for SUBSETSUM:

```

1 def subset_sum_verify(C, t, S):
2     """C,S: collections of integers
3         t: integer
4     Verify that S is a subcollection of C summing to t."""
5     if sum(S) != t: # check sum
6         return False
7     C = list(C) # make copy that we can change
8     for x in S: # ensure S is a subcollection of C
9         if x not in C:
10            return False
11        C.remove(x)
12    return True

```

Let  $n = |C|$ . We have  $|S| \leq |C|$ , and each number in  $C$  can have at most  $n$  bits, so it takes time  $O(n)$  to add them. Thus checking the sum of all numbers in  $S$  takes  $O(n^2)$  time. Making the copy of  $C$  takes  $O(n)$  time. the loop executes  $\leq n$  iterations. Each check for  $x \in C$  in the loop, and each removal, takes time  $O(n)$ . Thus the verifier is polynomial-time.  $\square$

Note that the complements of these languages,  $\overline{\text{CLIQUE}}$  and  $\overline{\text{SUBSETSUM}}$ , 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  $\overline{\text{CLIQUE}}, \overline{\text{SUBSETSUM}} \in \text{coNP}$ . It is not known whether  $\text{coNP} = \text{NP}$ , but this is believed to be false, although proving that is at least as difficult as proving that  $\text{P} \neq \text{NP}$ : since  $\text{P} = \text{coP}$ , if  $\text{coNP} \neq \text{NP}$ , then  $\text{P} \neq \text{NP}$ .

### 10.2.1 The P versus NP question

Exactly one of the two possibilities shown in Figure 10.1 is correct.

The best known method for solving NP problems in general is the one employed in the proof of Theorem 10.3.1 in the next subsection: a brute-force search over all possible witnesses, giving each as input to the verifier.

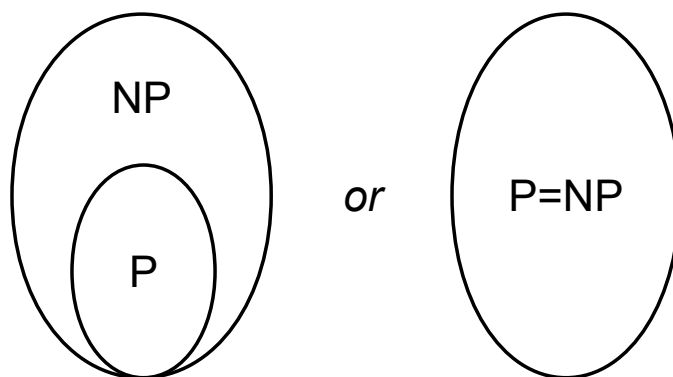


Figure 10.1: We know that  $P \subseteq NP$ , so either  $P \subsetneq NP$ , or  $P = NP$ .

end of lecture 7c

### 10.3 NP problems are decidable in exponential time

Although a problem being in NP doesn't give any hint how to decide it efficiently, it *does* give a way to decide it inefficiently: brute-force search over the (exponentially many) possible witnesses. Much of the focus of this chapter is in providing evidence that this exponential blowup is fundamental, and not an artifact of the next proof that can be avoided through a clever trick.

Let  $\text{EXP} = \bigcup_{k=1}^{\infty} \text{TIME}(2^{n^k})$  be the set of decision problems decidable in exponential time.

**Theorem 10.3.1.**  $NP \subseteq \text{EXP}$ .

In other words, every problem verifiable in polynomial time is decidable in exponential time. *Proof idea:* To solve a problem in NP, iterate over all possible witnesses, and run the verifier on each of them.

*Proof.* Let  $A \in NP$ . Then there are constants  $c, k$  and a  $O(n^c)$ -time verifier  $V_A$  such that, for all  $x \in \{0, 1\}^*$  (letting  $n = |x|$ ),  $x \in A \iff (\exists w \in \{0, 1\}^{\leq n^k}) V(x, w)$  accepts. For example, if  $k = 7$ , then the following algorithm decides if the latter condition is true:

```

1 import itertools
2
3 k = 7 # k is a constant hard-coded into the program
4
5 def binary_strings_of_length(length):
6     """Generate all strings of a given length"""
7     return map(lambda lst: "".join(lst),
8               itertools.product(["0", "1"], repeat=length))
9
10 def A_decider(x):

```

```

11  """Exponential-time algorithm for finding witnesses."""
12  n = len(x)
13  for m in range(n**k + 1): # check lengths m in [0,1,...,n^c]
14      for w in binary_strings_of_length(m):
15          if V_A(x,w):
16              return True
17  return False

```

We now analyze the running time. The two loops iterate over all binary strings of length at most  $n^k$ , of which there are  $\sum_{m=0}^{n^k} 2^m = 2^{n^k+1} - 1 = O(2^{n^k})$ . Each iteration calls  $V_A(x, w)$ , which takes time  $O(n^c)$ . Then the total time is  $O(n^c \cdot 2^{n^k}) = O(2^{2n^k})$ .  $\square$

This is shown by example for `HAMPATH` below. It is a bit more efficient than above: rather than searching all possible witness binary strings up to some length, it uses the fact that any valid witness will encode a list of nodes, that list will have length  $n = |V|$ , and it will be a permutation of the nodes (each node in  $V$  will appear exactly once). (`ham_path_verify` is the same as the verification algorithm we used above, except that the checks intended to ensure that  $p$  is a permutation of  $V$  have been removed for simplicity, since now we are calling `ham_path_verify` only with  $p =$  a permutation of  $V$ )

```

1  import itertools
2  def ham_path_verify(G,p):
3      """G: graph (V,E)
4         p: permutation of V (list of unique nodes of length len(V)
5         Verify that p is a Hamiltonian path in G."""
6      V,E = G
7      # verify each pair of adjacent nodes in p shares an edge
8      for i in range(len(p) - 1):
9          if (p[i],p[i+1]) not in E:
10             return False
11     return True
12
13 def ham_path(G):
14     """G: graph
15     Exponential-time algorithm for finding Hamiltonian paths,
16     which calls the verifier on all potential witnesses."""
17     V,E = G
18     for p in itertools.permutations(V):
19         if ham_path_verify(G,p):
20             return True
21     return False

```

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



## 10.4 Introduction to NP-Completeness

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 10.9)

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 factor, the hardest problems” in NP. If any NP-complete problem has a polynomial-time algorithm, then all problems in NP also have polynomial-time algorithms, including all the other NP-complete problems. In other words, if any NP-complete problem is in P, then  $P = NP$ . 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:<sup>2</sup> if  $P \neq NP$ , 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  $P \neq NP$ ) intractable: show that it 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 such as  $\bar{0}$  to mean  $\neg 0$ ):

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

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

$$\phi = (x \wedge y) \vee (z \wedge \neg y)$$

is a Boolean formula. We may also write negation with an overbar, such as

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

More formally, given a finite set  $V$  of *variables* (we write variables as single letters such as  $a$  and  $b$ , sometimes subscripted, e.g.,  $x_1, \dots, x_n$  for  $n$  variables), a *Boolean formula over  $V$*  is either 0) (base case) a variable  $x \in V$ , or 1) (recursive case 1)  $\neg\phi$ , where  $\phi$  is a Boolean formula over  $V$  (called the *negation* of  $\phi$ ), 2) (recursive case 2)  $\phi \wedge \psi$ , where  $\phi$  and  $\psi$  are Boolean formulas over  $V$ , (called the *conjunction* of  $\phi$  and  $\psi$ ), 3) (recursive case 3)  $\phi \vee \psi$ , where  $\phi$  and  $\psi$  are Boolean formulas over  $V$ , (called the *disjunction* of  $\phi$  and  $\psi$ ).

$\neg$  takes precedence over both  $\wedge$  and  $\vee$ , and  $\wedge$  takes precedence over  $\vee$ . Parentheses may be used to override default precedence.

The following is a simple implementation of Boolean formulas as a Python class using the recursive definition. (Don’t be scared of the next chunk of code; most of it is parsing code to support the ability to create a Boolean formula object from a string representing it, which is simpler than manually creating the objects recursively.)

<sup>2</sup>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.

```

1 class Boolean_formula(object):
2     """Represents a Boolean formula with AND, OR, and NOT operations."""
3     def __init__(self, variable=None, op=None, left=None, right=None):
4         if not ((variable == None and op == "not" and left == None and right != None)
5                 or (variable == None and op == "and" and left != None and right != None)
6                 or (variable == None and op == "or" and left != None and right != None)
7                 or (variable != None and op == left == right == None)):
8             raise ValueError("Must either set variable for base case" +\
9                               "or must set op to 'not', 'and', or 'or'" +\
10                              "and recursive formulas left and right")
11         self.variable = variable
12         self.op = op
13         self.left = left
14         self.right = right
15         if self.variable:
16             self.variables = [variable]
17         elif op == 'not':
18             self.variables = right.variables
19         elif op in ['and', 'or']:
20             self.variables = list(left.variables)
21             self.variables.extend(x for x in right.variables if x not in left.
22                                 variables)
23             self.variables.sort()
24     def evaluate(self, assignment):
25         """Value of this formula with given assignment, a dict mapping variable
26         names to Python booleans.
27
28         Assignment can also be a string of bits, which will be mapped to variables
29         in alphabetical order. Boolean values are interconverted with integers to
30         make for nicer printing (0 and 1 versus True and False)"""
31         if type(assignment) is str:
32             assignment = dict(zip(self.variables, map(int, assignment)))
33         if self.op == None:
34             return assignment[self.variable]
35         elif self.op == 'not':
36             return int(not self.right.evaluate(assignment))
37         elif self.op == 'and':
38             return int(self.left.evaluate(assignment) and self.right.evaluate(assignment))
39         elif self.op == 'or':
40             return int(self.left.evaluate(assignment) or self.right.evaluate(assignment))
41         else:
42             raise ValueError("This shouldn't be reachable")
43     def __repr__(self):
44         if self.variable:
45             return self.variable
46         elif self.op == 'not':
47             return '(not {})'.format(self.right)
48         else:
49             return '({} {} {})'.format(self.left, self.op, self.right)
50

```

```

51
52 def __str__(self):
53     return repr(self)
54
55 @staticmethod
56 def from_string(text):
57     """Convert string that looks like a Python Boolean expression with
58     variables, e.g. "x and y or not z and (a or b)", to Boolean_formula."""
59     # add plenty of whitespace to make it easy to tokenize with string.split()
60     for token in ['and', 'or', 'not', '(', ')']:
61         text = text.replace(token, ' ' + token + ' ')
62     tokens = text.split()
63     val_stack = []
64     op_stack = []
65     for token in tokens:
66         if token in ['and', 'or', 'not']:
67             cur_op = token
68             while len(op_stack) > 0 and not precedence_greater(cur_op, op_stack[-1]):
69                 process_top_op(op_stack, val_stack)
70             op_stack.append(cur_op)
71         elif token == '(':
72             op_stack.append('(')
73         elif token == ')':
74             while op_stack[-1] != '(':
75                 process_top_op(op_stack, val_stack)
76             op_stack.pop()
77         else:
78             val_stack.append(Boolean_formula(variable=token))
79     while len(op_stack) > 0 and not precedence_greater(cur_op, op_stack[-1]):
80         process_top_op(op_stack, val_stack)
81     return val_stack.pop()
82
83 def process_top_op(op_stack, val_stack):
84     """Processes top operator from op_stack, popping one or two values as needed
85     from val_stack, and pushing the result back on the value stack."""
86     op = op_stack.pop()
87     right = val_stack.pop()
88     if op == 'not':
89         val_stack.append(Boolean_formula(op='not', right=right))
90     elif op in ['and', 'or']:
91         left = val_stack.pop()
92         val_stack.append(Boolean_formula(op=op, left=left, right=right))
93
94 def precedence_greater(op1, op2):
95     return (op2 == '(') or (op1=='not' and op2 in ['and', 'or']) or (op1=='and' and
96         op2=='or')

```

The following code builds the formula  $\phi = (x \wedge y) \vee (z \wedge \bar{y})$  and then evaluates it on all  $2^3 = 8$  assignments to its three variables.

```

1 formula = Boolean_formula.from_string("((x and y) or (z and (not y)))")
2 import itertools
3 num_variables = len(formula.variables)
4 for assignment in itertools.product(["0", "1"], repeat=num_variables):

```

```

5     assignment = "".join(assignment)
6     value = formula.evaluate(assignment)
7     print("formula value = {} on assignment {}".format(value, assignment))

```

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.  $\phi = (x \wedge y) \vee (z \wedge \bar{y})$  is satisfiable because assigning  $x = 0, y = 0, z = 1$  causes  $\phi$  to evaluate to 1, written  $\phi(001) = 1$ . We say the assignment *satisfies*  $\phi$ . The *Boolean satisfiability problem* is to test whether a given Boolean formula is satisfiable:

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

Note that  $\text{SAT} \in \text{NP}$ , since the language

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

has a polynomial time decider, i.e., there is a polynomial-time verifier for SAT: if  $\phi$  has  $n$  input variables, then  $\langle \phi \rangle \in \text{SAT} \iff (\exists w \in \{0, 1\}^n) \langle \phi, w \rangle \in \text{SAT}_V$ . The verifier is essentially the method `Boolean_formula.evaluate` in the code above.

Finally, we state the reason for the importance of the SAT (as well as CLIQUE, HAMPATH, SUBSETSUM, and hundreds of other problems that share this characteristic):

**Theorem 10.4.1** (Cook-Levin Theorem).  $\text{SAT} \in \text{P}$  if and only if  $\text{P} = \text{NP}$ .

This is considered evidence (though not proof) that  $\text{SAT} \notin \text{P}$ . The reason the Cook-Levin Theorem is true is that SAT is NP-complete. In the next subsection, we develop the technical machinery needed to prove that a problem is NP-complete: reductions.

end of lecture 8a

## 10.5 Polynomial-time reducibility

Theorem 10.4.1 is proven by showing, in a certain technical sense, that SAT is “at least as hard” as every other problem in NP. The concept of *reductions* is used to formalize what we mean when we say a problem is at least as hard as another. Let’s discuss a bit what we might mean intuitively by this. First, by “hard”, we mean with respect to the running time required to solve the problem. If problem  $A$  can be decided in time  $O(n^3)$ , whereas problem  $B$  cannot be decided in time  $\Omega(n^6)$ , this means that  $B$  is harder than  $A$ : the fastest algorithm for  $A$  is faster than the fastest algorithm for  $B$ .

But, we will also do what we have been doing so far and relax our standards of comparison to ignore polynomial differences. So, suppose that  $B$  is actually decidable in time  $O(n^6)$  and no smaller: there is an  $O(n^6)$  time algorithm for  $B$  and every algorithm deciding  $B$  requires  $\Omega(n^6)$  time. Then, since  $n^6$  is within a polynomial factor of  $n^3$ , because  $(n^3)^2 = n^6$  (i.e.,  $n^6$  “only” quadratically larger than  $n^3$ ), we won’t consider this difference significant, and we

will say that the hardness of  $A$  and  $B$  are close enough to be “equivalent”, since they are within a polynomial factor of each other.<sup>3</sup>

However, suppose there is a third problem  $C$ , and we could prove it requires time  $\Omega(2^n)$  to decide. (We have difficulty proving this for particular natural problems, but the Time Hierarchy Theorem assures us that such problems must exist). Then we will say that  $C$  really is harder than  $A$  or  $B$ , since  $2^n$  is not within a polynomial factor of either  $n^3$  or  $n^6$ . Even composing  $n^6$  with a huge polynomial like  $n^{1000}$  gives the polynomial  $(n^6)^{1000} = n^{6000} = o(2^n)$ .

But, as we said, it is often difficult to take a particular natural problem that we care about and prove that no algorithm with a certain running time can solve it. Obtaining techniques for doing this sort of thing may well be the most important open problem in theoretical computer science, and after decades we still have very few tools to do so. That is to say, it is difficult to pin down the *absolute hardness* of a problem, the running time of the most efficient algorithm for the problem.

What we do have, on the other hand, is a way to compare the *relative hardness* of two problems. What *reducibility* allows to do is to say ... okay, fine, although I don't know what's the *best* algorithm for  $A$ , and I also don't know what's the best algorithm for  $B$ , but what I do know is that, if  $A$  is “reducible” to  $B$ , this means that *any algorithm for  $B$  can be used to solve  $A$  with only a polynomial amount of extra time*. Therefore, *whatever* is the fastest algorithm for  $B$  (even if I don't know what it is, or how fast it is), I know that the fastest algorithm for  $A$  is “almost” as fast (perhaps a polynomial factor slower). So in particular, if  $B$  is decidable in polynomial time, then so is  $A$  (perhaps with a larger exponent in the polynomial).

**Example 10.5.1.** Recall a *clique* in a graph  $G = (V, E)$  is a subset  $C \subseteq V$  such that, for all pairs  $u, v \in C$  of distinct nodes in  $C$ ,  $\{u, v\} \in E$ . An *independent set* in  $G$  is a similar concept:  $S \subseteq V$  is an independent set if, for all  $u, v \in S$ ,  $\{u, v\} \notin E$ , and we say it is a *k-independent set* if it has  $k$  nodes. Suppose we have an algorithm that can decide, on input  $\langle G_c, k \rangle$ , where  $G_c$  has a  $k$ -clique. How can we use this to decide the problem, given input  $\langle G_i, k \rangle$ , whether  $G_i$  has a  $k$ -independent set?

The idea is shown in Figure 10.2. To decide whether  $\langle G_i, k \rangle \in \text{INDSET}$ , we map the graph  $G_i = (V, E)$  to the graph  $G_c = (V, \overline{E})$ , (i.e., for each pair of nodes in  $V$ , add an edge if there wasn't one, and otherwise remove the edge if there was one), and then we give  $\langle G_c, k \rangle$  to the decider for CLIQUE. Since each independent set in  $G_i$  becomes a clique in  $G_c$ , for each  $k$ ,  $G_i$  has a  $k$ -independent set if and only if  $G_c$  has a  $k$ -clique. Thus, the decider for CLIQUE can be used as a subroutine to decide INDSET, by transforming the input  $\langle G_i, k \rangle$  into  $\langle G_c, k \rangle$ , passing the transformed input to the decider for CLIQUE, and then returning its answer.

---

<sup>3</sup>By this metric, *all* polynomial-time algorithms are within a polynomial factor of each other. But there are other polynomially-equivalent running times that are larger than polynomial. For instance, a  $2^n$ -time algorithm is within a polynomial factor of a  $n^5 \cdot 2^n$ -time algorithm, or even a  $4^n$ -time algorithm, since  $4^n = (2^n)^2$ . However, a  $2^n$ -time algorithm is not within a polynomial factor of a  $2^{n^2}$ -time algorithm, since  $2^{n^2}$  is not bounded by a polynomial function of  $2^n$ ; i.e., there is no polynomial  $p$  such that  $2^{n^2} = O(p(2^n))$ . Even if  $p(n) = n^{1000}$ , we still get  $p(2^n) = (2^n)^{1000} = 2^{1000n}$ , which is  $o(2^{n^2})$  because  $1000n = o(n^2)$ .

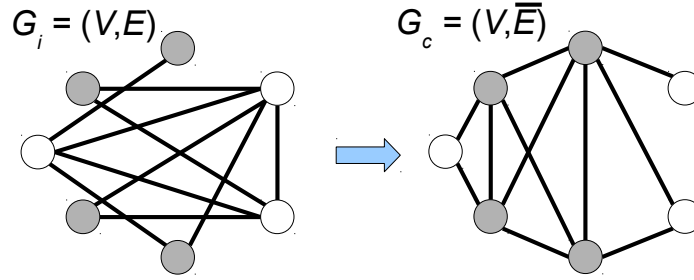


Figure 10.2: A way to “reduce” the problem of finding an independent set of some size in a graph  $G_i$  to finding an clique of the same size. We transform the graph  $G_i$  on the left to  $G_c$  on the right by taking the complement of the set of edges. Each independent set in  $G_i$  (such as the four shaded nodes) becomes a clique in  $G_c$ .

The next definition formalizes this idea.

**Definition 10.5.2.** A function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  is *polynomial-time computable* if there is a polynomial-time that, on input  $x$ , returns  $f(x)$ .

**Definition 10.5.3.** Let  $A, B \subseteq \{0, 1\}^*$ . We say  $A$  is *polynomial-time reducible* to  $B$  (a.k.a., *polynomial-time many-one reducible*, *polynomial-time mapping reducible*), written  $A \leq^P B$ , if there is a polynomial-time computable function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  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  $B$ .

We interpret  $A \leq^P B$  to mean that “ $B$  is at least as hard as  $A$ , to within a polynomial-time factor.”

A pictorial representation of a mapping reduction is shown in Figure 10.3. To implement the mapping reduction of Example 10.5.1, we can use this:

```

1 import itertools
2 def reduction_from_independent_set_to_clique(G):
3     V,E = G
4     Ec = [ {u,v} for (u,v) in itertools.combinations(V,2)
5             if {u,v} not in E and u!=v ]
6     Gc = (V,Ec)
7     return Gc

```

If we had a hypothetical polynomial-time algorithm for CLIQUE, then it could be used to make a polynomial-time algorithm for INDSET, calling the reduction above, in the following way:

```

1 # hypothetical polynomial-time algorithm for Independent-Set, which
2 # calls another hypothetical polynomial-time algorithm A for Clique
3 def independent_set_algorithm(G,k):
4     Gp,kp = reduction_from_clique_to_independent_set(G,k)

```

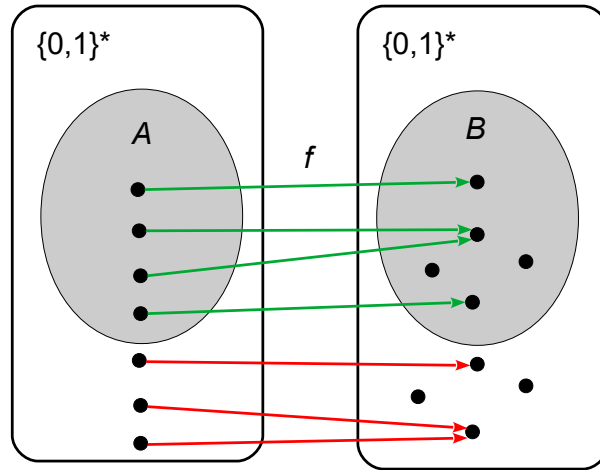


Figure 10.3: A mapping reduction  $f$  that reduces decision problem  $A \subseteq \{0,1\}^*$  to  $B \subseteq \{0,1\}^*$ . The key visual property above is that no arrow points from inside  $A$  to outside  $B$ , nor from outside  $A$  to inside  $B$ . Either both endpoints are in the shaded regions, or both are not.

```

5     return clique_algorithm(Gp, kp)
6
7 def clique_algorithm(G, k):
8     raise NotImplementedError()

```

More formally, defining  $\text{INDSET} = \{\langle G, k \rangle \mid G \text{ has a } k\text{-independent set}\}$ , we have just shown that  $\text{INDSET} \leq^P \text{CLIQUE}$ . In this special case, the reduction above *also* shows that  $\text{CLIQUE} \leq^P \text{INDSET}$ , since one can use an  $\text{INDSET}$ -decider as a subroutine to decide  $\text{CLIQUE}$  in the exact same way. But this is a special case that does not hold in general: a reduction from  $A$  to  $B$  is not in general also a reduction from  $B$  to  $A$ , and it may be that  $A \leq^P B$  but  $B \not\leq^P A$ .<sup>4</sup>

Note that for any reduction  $f$  computable in time  $n^c$  for some constant  $c$ , and all  $x \in \{0,1\}^*$ ,  $|f(x)| \leq |x|^c$ , because if the TM runs for  $|x|^c$  steps, it can write down at most one output bit of  $f(x)$  per step.

The following theorem captures one way to formalize the claim “ $A \leq^P B$  means that  $A$  is no harder than  $B$ ”:

**Theorem 10.5.4.** *Suppose  $A \leq^P B$ . If  $B \in \mathbf{P}$ , then  $A \in \mathbf{P}$ .*

*Proof.* The idea of how to use a polynomial-time mapping reduction is shown in Figure 10.4.

Let  $M_B$  be the algorithm deciding  $B$  in time  $n^k$  for some constant  $k$ , and let  $f$  be the reduction from  $A$  to  $B$ , computable in time  $n^c$  for some constant  $c$ . Define the algorithm

```

1 def f(x):
2     raise NotImplementedError()

```

<sup>4</sup>For instance, we believe this is true of  $\text{NP}$ -complete problems: The statement that  $\mathbf{P} \neq \text{NP}$  is equivalent to the statement that  $\text{PATH} \leq^P \text{CLIQUE}$  but  $\text{CLIQUE} \not\leq^P \text{PATH}$ , since  $\text{PATH} \in \mathbf{P}$  and  $\text{CLIQUE}$  is  $\text{NP}$ -complete.

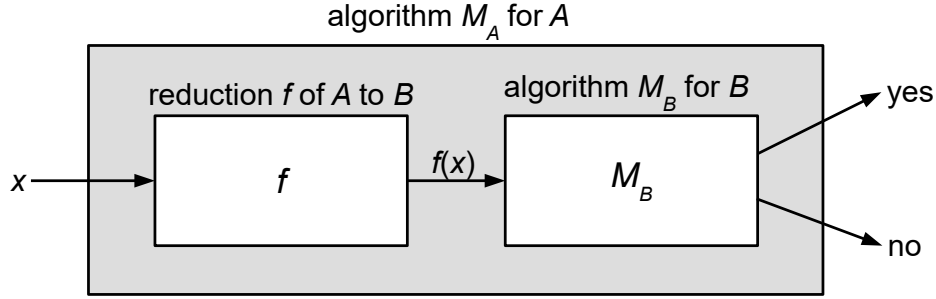


Figure 10.4: How to compose a mapping reduction  $f : \{0,1\}^* \rightarrow \{0,1\}^*$  that reduces decision problem  $A$  to decision problem  $B$ , with an algorithm  $M_B$  deciding  $B$ , to create an algorithm  $M_A$  deciding  $A$ . If  $M_B$  and  $f$  are both computable in polynomial time, then their composition  $M_A$  is computable in polynomial time as well.

```

3      # TODO: code for f, reducing A to B, goes here
4
5  def M_B(y):
6      raise NotImplementedError()
7      # TODO: code for M, deciding B, goes here
8
9  def M_A(x):
10     """Compose reduction f from A to B, with algorithm M for B,
11     to get algorithm N for A."""
12     y = f(x)
13     output = M_B(y)
14     return output

```

$N$  correctly decides  $A$  because  $x \in A \iff f(x) \in B$ . Furthermore, on input  $x$  of length  $n$ , the line  $y = f(x)$  runs in time  $n^c$ , and assuming in the worst case that the length of  $y$  is  $n^c$ , the line  $\text{output} = M_B(y)$  runs in time  $(n^c)^k = n^{ck}$ . Thus  $N$  runs in time at most  $n^c + n^{ck} = n^{c(k+1)}$ .  $\square$

Theorem 10.5.4 tells us that if the fastest algorithm for  $B$  takes time  $t(n)$ , then the fastest algorithm for  $A$  takes no more than time  $p(n) + t(p(n))$ , where  $p$  is the running time of  $f$ ; i.e.,  $p$  is the “polynomial factor” when claiming that “ $A$  is no harder than  $B$  within a polynomial factor”. Since we ignore polynomial differences when defining  $P$ , if it is also true that  $t$  is a polynomial (i.e.,  $B \in P$ ), then we conclude that  $A \in P$  as well, since  $p(n) + t(p(n))$  is bounded by a polynomial in  $n$ .

This is where things begin to get a bit abstract compared to previous definitions. The main way that we use reductions is to show that efficient algorithms do *not* exist for a problem, by invoking the *contrapositive* of Theorem 10.5.4:

**Corollary 10.5.5.** *Suppose  $A \leq^P B$ . If  $A \notin P$ , then  $B \notin P$ .*

In other words, if  $A \leq^P B$ , and if  $A$  is hard to decide, then  $B$  is also hard to decide.



**How to remember which direction reductions go:** The most common mistake made when using a reduction to show that a problem is hard is to switch the order of the problems in the reduction, in other words, to mistakenly attempt to show (for example) that  $\text{HAMPATH} \leq^P \text{SAT}$  by showing that we can transform instances of SAT into instances of HAMPATH (a mistake because we should instead go the other way and transform instances of HAMPATH into instances of SAT).

The  $\leq$  sign is intended to be evocative of “comparing hardness” of the problems. So if we write  $A \leq^P B$ , this means *B is at least as hard as A*, or *A is at least as easy as B*. How do you show a problem is easy? You give an algorithm for solving it! So the way to remember which problem the reduction is helping you solve, is to ask, “which is the easier problem here?” *That’s* the one you are writing an algorithm for. The easier problem is the one on the left of the  $\leq^P$ ,  $A$ . So if the reduction is intended to be used with an *existing* algorithm for one problem  $B$ , in order to show that there is an algorithm for the other problem  $A$ , then then problem  $A$  for which you are *showing an algorithm exists* is the easier one.

Of course, there’s a simpler mnemonic, which doesn’t really convey the *why* of reductions, but that is easy to remember. We always write reductions in the same left-to-right order  $A \leq^P B$ . If you picture the computation of the reduction itself starting with  $x$  on the left, then processing through  $f$  to end up with  $f(x)$  on the right ( $x \xrightarrow{f} f(x)$  as in Figure 10.4), then this will remind you that you should start with an instance  $x$  of  $A$ , and end with an instance  $f(x)$  of  $B$ .

Unlike the previous examples of code implementing these ideas, we cannot give concrete examples of  $M_A$  and  $M_B$  above, because for the types of problems we consider using these reductions, we believe that no efficient algorithms  $M_A$  and  $M_B$  *exist* for the two problems  $A$  and  $B$ . There is, however, a concrete, efficiently computable reduction  $f$  from  $A$  to  $B$  for many important choices of  $A$  and  $B$ . We cover one of these choices next.

We now use  $\leq^P$ -reductions to show that INDSET is “at least as hard” (to within a polynomial factor) as a restricted version of the SAT problem known as 3SAT.

A *literal* is a Boolean variable or negated Boolean variable, such as  $x$  or  $\bar{x}$ . A *clause* is several literals connected with  $\vee$ ’s, such as  $(x \vee \bar{y} \vee \bar{z} \vee x)$ . A *conjunction* is several subformulas connected with  $\wedge$ ’s, such as  $(x \wedge (\bar{y} \vee z) \wedge \bar{z} \wedge w)$ . A Boolean formula  $\phi$  is in *conjunctive normal form*, called a *CNF-formula*, if it consists of a conjunction of clauses,<sup>5</sup> such as

$$\phi = (x \vee \bar{y} \vee \bar{z} \vee w) \wedge (z \vee \bar{w} \vee x) \wedge (y \vee \bar{x}).$$

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

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

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

<sup>5</sup>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.

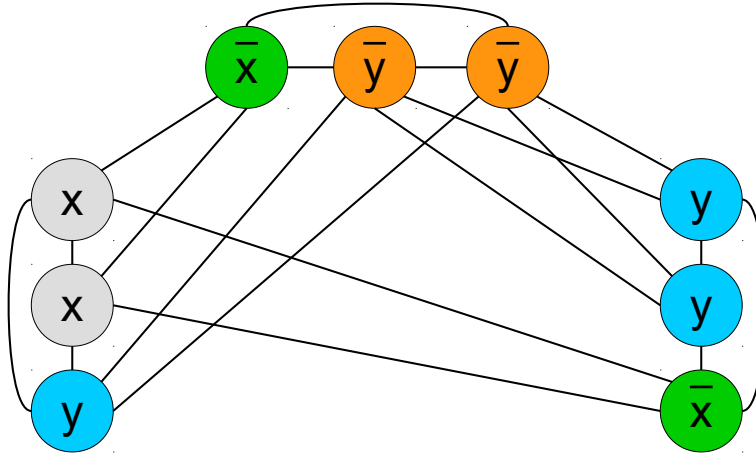


Figure 10.5: Example of the  $\leq^P$ -reduction from 3SAT to INDSET when the input is the 3CNF formula  $\phi = (x \vee x \vee y) \wedge (\bar{x} \vee \bar{y} \vee \bar{y}) \wedge (\bar{x} \vee y \vee y)$ . Observe that  $x = 0, y = 1$  is a satisfying assignment. Furthermore, if we pick exactly one node in each triple, corresponding to a literal that satisfies the clause associated to that triple, it is a  $k$ -independent set: since we are picking the literal that satisfies each clause, we never pick both a literal and its negation, so every node we pick lacks an edge to every other node. In this example, the formula is satisfied by  $x = 0, y = 1$ , and picking nodes  $y$  on the left, one of the  $y$  nodes on the right, and  $\bar{x}$  on the top is a 3-independent set.

Define

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

Theorem 10.8.3 shows that 3SAT is NP-complete. For now we will simply show that it is reducible to INDSET.

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 INDSET, 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 10.5.6.**  $3\text{SAT} \leq^P \text{INDSET}$ .

*Proof.* Given a 3CNF formula  $\phi$ , the reduction must output a pair  $\langle G, k \rangle$ , a graph  $G$  and integer  $k$ , so that

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

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

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

$G$  will have  $3k$  nodes, each labeled with a literal appearing in  $\phi$ .<sup>6</sup>

See Figure 10.5 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  $\phi$ .

$G$  has edge  $\{u, v\}$  if and only if at least one of the following holds:

1.  $u$  and  $v$  are in the same triple, or
2.  $u$  is labeled  $x$  and  $v$  is labeled  $\bar{x}$  for some variable  $x$ , or vice-versa.

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

We now show that

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

( $\implies$ ): Suppose  $\phi(w) = 1$  has a satisfying assignment. Then at least one literal is true in every clause. To construct a  $k$ -independent set  $S$ , select exactly one node from each clause labeled by a true literal (breaking ties arbitrarily). For every  $u, v \in S$  where  $u \neq v$ , condition (1) is false, and since  $x$  and  $\bar{x}$  cannot both be true, condition (2) is false. Therefore  $S$  is a  $k$ -independent set.

( $\impliedby$ ): Suppose there is a  $k$ -independent set  $S$  in  $G$ . For every  $u, v \in S$  where  $u \neq v$ ,  $u$  and  $v$  are in different triples by condition (1). Since there are  $k$  triples and  $|S| = k$ ,  $S$  contains *exactly* one node from each triple. If a node in  $S$  is labeled with literal  $x$ , assign  $x = 1$ . If the node is labeled with literal  $\bar{x}$ , assign  $x = 0$ . Assign other variables arbitrarily (e.g., set them to 0).

Since no pair of nodes in  $S$  are labeled with  $x$  and  $\bar{x}$  by condition (2), this assignment is well-defined (we will not attempt to assign  $x$  to be both 0 and 1). The assignment makes every clause true, thus satisfies  $\phi$ .  $\square$

The reduction can be implemented in Python as

```

1 class CNF(object):
2     """Represents a CNF formula. Each variable is a string (e.g., "x1",
3     and each clause is a tuple of strings, each either a variable
4     or its negation, e.g., ("!x1", "x3", "!x4") """
5     def __init__(self, clauses):
6         self.variables = extract_variables(clauses)
7         self.clauses = clauses
8
9     import itertools
10    def reduce_3sat_to_indset(cnf_formula):
11        nodes_grouped = [(( 'a', idx, clause[0]), ( 'b', idx, clause[1]), ( 'c', idx, clause[2]))

```

<sup>6</sup>Note that  $\phi$  could have many more (or fewer) literals than variables, so  $G$  may have many more nodes than  $\phi$  has variables. But  $G$  will have exactly as many nodes as  $\phi$  has *literals*. Note also that literals can appear more than once, e.g.,  $(x \vee x \vee y)$  has two copies of the literal  $x$ .

```

12     for (clause,idx) in zip(cnf_formula.clauses, itertools.count())
13     V = [ node for node_group in nodes_grouped for node in node_group ]
14     E = []
15     for (u,v) in itertools.combinations(V,2):
16         add_edge = False
17         (_,clause_idx1,lit1),(_,clause_idx2,lit2) = u,v
18         # add edge if nodes are in same clause
19         for clause in cnf_formula.clauses:
20             if clause_idx1 == clause_idx2:
21                 add_edge = True
22             if lit1 == "!" + lit2 or lit2 == "!" + lit1:
23                 add_edge = True
24             if add_edge:
25                 E.append({u,v})
26     k = len(cnf_formula.clauses)
27     return ((V,E), k)

```

Theorems 10.5.4 and 10.5.6 tell us that if INDSET is decidable in polynomial time, then so is 3SAT. In terms of what we believe is actually true, Corollary 10.5.5 and Theorem 10.5.6 tell us that if 3SAT is *not* decidable in polynomial time, then neither is INDSET.

Since INDSET is equivalent to CLIQUE, this also shows that if 3SAT is not decidable in polynomial time, then neither is CLIQUE.

**Reductions are algorithms.** A reduction is just an algorithm. What makes them difficult to understand when first learning the concept is that they are used to relate the difficulty of two problems, but *not* to solve either problem. The reduction above, showing that  $3SAT \leq^P INDSET$ , is an algorithm that takes an instance of 3SAT as input, but it does not solve the problem 3SAT. It also does not solve the problem INDSET. It transforms an instance of 3SAT into an instance of INDSET, while preserving the correct answer, but without ever knowing what that answer is. The job of the reduction is to translate the question about a formula into a question about a graph, *not* to answer either question.

end of lecture 8b

## 10.6 Definition of NP-completeness

**Definition 10.6.1.** A language  $B$  is *NP-hard* if, for every  $A \in NP$ ,  $A \leq^P B$ .

**Definition 10.6.2.** A language  $B$  is *NP-complete* if

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

**Theorem 10.6.3.** If  $B$  is NP-complete and  $B \in P$ , then  $P = NP$ .

*Proof.* Assume the hypothesis. Since  $P \subseteq NP$ , it suffices to show  $NP \subseteq P$ . Let  $A \in NP$ . Since  $B$  is NP-hard,  $A \leq^P B$ . Since  $B \in P$ , by Theorem 10.5.4 (the closure of  $P$  under  $\leq^P$ -reductions),  $A \in P$ . Since  $A$  was arbitrary,  $NP \subseteq P$ .  $\square$

**Corollary 10.6.4.** *If  $P \neq NP$ , then no NP-complete problem is in  $P$ .*

Since it is generally believed that  $P \neq NP$ , Corollary 10.6.4 implies that showing a problem is NP-complete is evidence of its intractability.

The following property of polynomial-time reductions will be useful:

**Observation 10.6.5.**  $\leq^P$  is transitive.

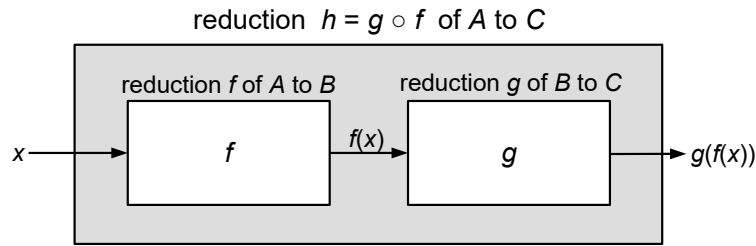


Figure 10.6: Polynomial-time reductions are transitive: simply compose the algorithms.

*Proof.* The idea is shown in Fig. 10.6. Let  $A, B, C \subseteq \{0, 1\}^*$  where  $A \leq^P B$  via reduction  $f$  and  $B \leq^P C$  via reduction  $g$ . Then  $h = g \circ f$  (i.e., for all  $x \in \{0, 1\}^*$ , define  $h(x) = g(f(x))$ ) is a polynomial-time reduction of  $A$  to  $C$ :  $h$  can be computed in polynomial time since both  $f$  and  $g$  can, and for all  $x \in \{0, 1\}^*$ ,  $x \in A \iff f(x) \in B$ , and  $f(x) \in B \iff g(f(x)) \in C$ , so  $x \in A \iff h(x) \in C$ .  $\square$

The following theorem, using the transitivity of reductions proven in Observation 10.6.5, is generally how one shows that a problem  $C$  is NP-hard: find some other NP-hard problem  $B$  and show  $B \leq^P C$ .

**Theorem 10.6.6.** *If  $B$  is NP-hard and  $B \leq^P C$ , then  $C$  is NP-hard.*

*Proof.* Let  $A \in NP$ . Then  $A \leq^P B$  since  $B$  is NP-hard. Since  $\leq^P$  is transitive and  $B \leq^P C$ , it follows that  $A \leq^P C$ . Since  $A \in NP$  was arbitrary,  $C$  is NP-hard.  $\square$

The following Python code shows how one would implement this idea.

```

1 def reduce_A_to_B(x):
2     """ reduction from some NP language A to NP-hard language B """
3     raise NotImplementedError()
4
5 def reduce_B_to_C(x):
6     """ reduction showing B reduces to C """
7     raise NotImplementedError()

```

```

8
9 def C_decider(x):
10     """ hypothetical polynomial-time decider for C """
11     raise NotImplementedError()
12
13 def A_decider(x):
14     """ composition of two reductions to show how a decider for C
15     can be called to decide A """
16     y = reduce_A_to_B(x)
17     z = reduce_B_to_C(y)
18     return C_decider(z)

```

**Corollary 10.6.7.** *If  $B$  is NP-complete,  $C \in \text{NP}$ , and  $B \leq^P C$ , 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 10.6.7 is our primary tool for proving a problem is NP-complete: show that some existing NP-complete problem reduces to it.

### 10.6.1 The Cook-Levin Theorem

We can now restate the Cook-Levin Theorem using the terminology we have developed.

**Theorem 10.6.8.** *SAT and 3SAT are NP-complete.*<sup>7</sup>

We won’t prove Theorem 10.6.8 in this course (take ECS 220 for that).

**Theorem 10.6.9.** *CLIQUE and INDSET are NP-complete.*

*Proof.*  $3\text{SAT} \leq^P \text{INDSET}$  by Theorem 10.5.6, and 3SAT is NP-complete, so INDSET is NP-hard. Recall that we also showed  $\text{INDSET} \leq^P \text{CLIQUE}$  via the reduction  $\langle (V, E), k \rangle \mapsto \langle (V, \bar{E}), k \rangle$ . So CLIQUE is NP-hard as well since the NP-hard problem INDSET reduces to it. We also showed both CLIQUE and INDSET are in NP, so they are NP-complete.  $\square$

end of lecture 8c

Memorial Day

end of lecture 9a

<sup>7</sup>By Theorem 10.6.3, Theorem 10.6.8 implies Theorem 10.4.1. In other words, because SAT is NP-complete, if  $\text{SAT} \in \text{P}$ , then since all  $A \in \text{NP}$  are reducible to SAT,  $A \in \text{P}$  as well, i.e.,  $\text{P} = \text{NP}$ . Conversely, if  $\text{P} = \text{NP}$ , then  $\text{SAT} \in \text{P}$  (and so is every other problem in NP).

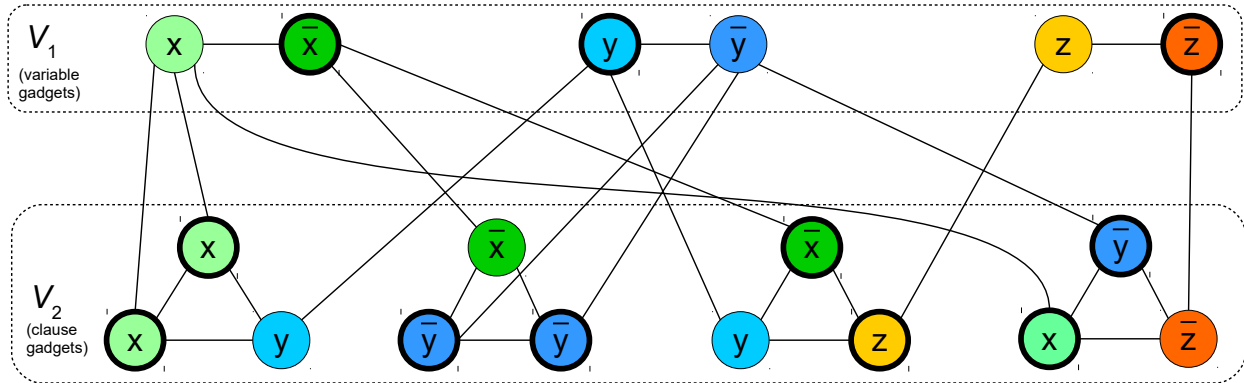


Figure 10.7: An example of the reduction from 3SAT to VERTEXCOVER for the 3CNF formula  $\phi = (x \vee x \vee y) \wedge (\bar{x} \vee \bar{y} \vee \bar{y}) \wedge (\bar{x} \vee y \vee z) \wedge (x \vee \bar{y} \vee \bar{z})$ , with  $m = 3$  variables and  $l = 4$  clauses.  $\phi$  is satisfied by  $x = 0, y = 1, z = 0$ , and we use this assignment to find a vertex cover  $C$  with  $|C| = m + 2l = 11$  as described in the proof; nodes in  $C$  are shown with a bold outline.

## 10.7 Optional: Additional NP-Complete problems

### 10.7.1 Vertex Cover

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 10.7.1.** 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

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

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

See Figure 10.7 for an example of the reduction.

For each variable  $x_i$  in  $\phi$ , add two nodes labeled  $x_i$  and  $\bar{x}_i$  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  $\phi$  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  $O(|\phi|^2)$  time.

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

( $\implies$ ): Suppose  $(\exists x_1 x_2 \dots x_m \in \{0, 1\}^m) \phi(x_1 x_2 \dots x_n) = 1$ . If  $x_i = 1$  (resp. 0), put the node in  $V_1$  labeled  $x_i$  (resp.  $\bar{x}_i$ ) 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.<sup>8</sup> 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.

**Reduction from INDSET:** Instead of reducing from 3SAT, an elegant way to see that VERTEXCOVER is NP-hard is to observe that  $\text{INDSET} \leq^P \text{VERTEXCOVER}$  by a particularly simple reduction  $\langle G, k \rangle \mapsto \langle G, n - k \rangle$ , where  $n$  is the number of nodes.

<sup>8</sup>Any two clause gadget nodes will cover all three clause gadget edges.



This works because  $S$  is an independent set in  $G = (V, E)$  if and only if  $V \setminus S$  is a vertex cover, and if  $|S| = k$  then  $|V \setminus S| = n - k$ . To see the forward direction, note that no pair of nodes in  $S$  has an edge that needs to be covered. Thus, if we pick all nodes in  $V \setminus S$ , they cover all edges between nodes in  $V \setminus S$ , and since  $S$  is an independent set, all remaining edges must be between a node in  $S$  and a node in  $V \setminus S$ , so the set  $V \setminus S$  covers these edges as well. Conversely, if  $V \setminus S$  is a vertex cover, then no pair of nodes in  $S$  can have an edge between them (since it would be uncovered by  $V \setminus S$ ), so  $S$  must be an independent set. In fact, this very same reduction also shows  $\text{VERTEXCOVER} \leq^P \text{INDSET}$ ; they are nearly equivalent problems, in the same way that  $\text{CLIQUE}$  is nearly equivalent to  $\text{INDSET}$  (recall that the reduction from  $\text{INDSET}$  to  $\text{CLIQUE}$  simply takes the complement of  $E$ ).  $\square$

We gave two reductions in the proof; there are many ways to prove a problem is NP-hard, and you just have to find one. The first one reducing 3SAT to VERTEXCOVER was much more complex than the second that used a reduction from INDSET. Why did we go through all that work then, if there was an easier way?

Sometimes the simplest reduction isn't so simple as the second one above. It's good to see an example of a more complex reduction between two objects of different "types" (in this case reducing a Boolean formula question to a graph question), to see how to do them when the correspondence between two problems is not so obvious. But, when you are trying to show a problem is NP-complete, generally the most efficient strategy is to start by looking for existing NP-complete problems about the *same type of object*. Often you find that some existing NP-complete problem is very close to your problem (such as INDSET and VERTEXCOVER), and the reduction is very straightforward. If that doesn't work, then move on to other NP-complete problems. For some reason, 3SAT often works well.

## 10.7.2 Subset Sum

**Theorem 10.7.2.** SUBSETSUM is NP-complete.

*Proof.* Theorem 10.2.2 tells us that SUBSETSUM  $\in$  NP. We show SUBSETSUM is NP-hard by reduction from 3SAT.

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

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

	$x_1$	$x_2$	$x_3$	$x_4$	$c_1$	$c_2$	$c_3$
$y_1$	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  $\bar{x}_i$ ) 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

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

( $\implies$ ): Suppose  $(\exists x_1 x_2 \dots x_m \in \{0, 1\}^m) \phi(x_1 x_2 \dots x_m) = 1$ . 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  $\phi$  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.<sup>9</sup> Then  $S'$  sums to  $t$ .

( $\impliedby$ ): 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  $x_1 x_2 \dots x_m \in \{0, 1\}^m$  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

<sup>9</sup>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.

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  $\phi(x_1x_2 \dots x_m) = 1$ .  $\square$

### 10.7.3 Hamiltonian path

**Theorem 10.7.3.**  $\text{HAMPATH}$  is NP-complete.

*Proof.* We showed  $\text{HAMPATH}$  is in NP earlier (before we had formally defined NP, but we did give a polynomial-time verifier).

To see that  $\text{HAMPATH}$  is NP-hard, we show  $3\text{SAT} \leq^P \text{HAMPATH}$ . Let  $\phi$  be a 3CNF formula with  $n$  variables  $x_1 \dots, x_n$  and  $k$  clauses  $C_1, \dots, C_k$ . Define a graph  $G_\phi = (V, E)$  with  $3k + 3$  nodes.

finish this

$\square$

Finally, we don't prove it in this course, but  $\text{HAMPATH}$  is NP-complete. So are the following variants:

- $\text{HAMPATHST} = \{\langle G, s, t \rangle \mid G \text{ is an directed graph with a Hamiltonian path from } s \text{ to } t\}$
- $\text{UHAMPATH} = \{\langle G \rangle \mid G \text{ is an undirected graph with a Hamiltonian path}\}$
- $\text{UHAMPATHST} = \{\langle G, s, t \rangle \mid G \text{ is an undirected graph with a Hamiltonian path from } s \text{ to } t\}$

## 10.8 Optional: Proof of the Cook-Levin Theorem

We prove Theorem 10.6.8 as follows. We first show that the language  $\text{CIRCUITSAT}$  is NP-complete, where  $\text{CIRCUITSAT}$  consists of the satisfiable Boolean *circuits*. We then show that  $\text{CIRCUITSAT} \leq^P 3\text{SAT}$ . Since every 3CNF formula is a special case of a Boolean formula, it is easy to check that  $3\text{SAT} \leq^P \text{SAT}$ . Thus, 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 10.8.1.** 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*.

What is the difference between a Boolean circuit and a Boolean formula? A formula is a special type of circuit in which all the gates have fan-out 1. A circuit is more general and allows a subexpression to be calculated once, then *shared* among several parts of the circuit by fanning out its value to several downstream gates.

Given an  $n$ -input Boolean circuit  $\gamma$  and a binary input string  $x = x_1x_2 \dots x_n$ , the value  $\gamma(x) \in \{0, 1\}$  is the value of the output gate when evaluating  $\gamma$  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  $\gamma$  is *satisfiable* if there is an input string  $x$  that satisfies  $\gamma$ ; i.e., such that  $\gamma(x) = 1$ .<sup>10</sup>

Define

$$\text{CIRCUITSAT} = \{ \langle \gamma \rangle \mid \gamma \text{ is a satisfiable Boolean circuit} \}.$$

**Theorem 10.8.2.** *CIRCUITSAT is NP-complete.*

*Proof Sketch.*  $\text{CIRCUITSAT} \in \text{NP}$  because the language

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

is a polynomial-time verification language for  $\text{CIRCUITSAT}$ .<sup>11</sup>

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

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

To show that  $A \leq^P \text{CIRCUITSAT}$ , we transform an input  $x$  to  $A$  into a circuit  $\gamma_x^V$  such that  $\gamma_x^V$  is satisfiable if and only if there is a  $w \in \{0, 1\}^{q(n)}$  such that  $V(x, w)$  accepts.<sup>12</sup>

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

We build a subcircuit  $\gamma_{\text{local}}$ .<sup>13</sup>  $\gamma_{\text{local}}$  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,<sup>14</sup> called  $\sigma_q$  and  $\sigma_a$ , respectively.

<sup>10</sup>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  $\text{CIRCUITSAT} \leq^P 3\text{SAT}$ , but not a difficult one.

<sup>11</sup>To show that  $\text{CIRCUITSAT}$  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*.

<sup>12</sup>In fact,  $w$  will be the satisfying assignment for  $\gamma_x^V$ . The subscript  $x$  is intended to emphasize that, while  $x$  is an input to  $V$ , it is hard-coded into  $\gamma_x^V$ ; choosing a different input  $y$  for the same verification algorithm  $V$  would result in a different circuit  $\gamma_y^V$ .

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.

<sup>13</sup>Many copies of  $\gamma_{\text{local}}$  will be hooked together to create  $\gamma_x^V$ .

<sup>14</sup>This is done so that a circuit may process them as inputs.

Let  $m = 1 + \lceil \log |Q| \rceil + \lceil \log |\Gamma| \rceil$ .<sup>15</sup> Represent  $V$ 's configuration  $C = (q, p, y)$ <sup>16</sup> as an element of  $\{0, 1\}^{p(n) \cdot m}$  as follows. The  $p^{\text{th}}$  tape cell with symbol  $a$  is represented as the string  $\sigma(p) = 1\sigma_q\sigma_a$ . Every tape cell at positions  $p' \neq p$  with symbol  $b$  are represented as  $\sigma(p') = 0\sigma_s\sigma_b$ .<sup>17</sup> Represent the configuration  $C$  by the string  $\sigma(C) = \sigma(0)\sigma(1) \dots \sigma(n^k - 1)$ .

$\gamma_{\text{local}} : \{0, 1\}^{3m} \rightarrow \{0, 1\}^m$  is defined to take as input  $\sigma(k-1)\sigma(k)\sigma(k+1)$ ,<sup>18</sup> and output the next configuration string for tape cell  $k$ .  $\gamma_{\text{local}}$  can be implemented by a Boolean circuit whose size depends only on  $\delta$ , so  $\gamma_{\text{local}}$ 's size is a constant depending on  $V$  but independent of  $n$ .<sup>19</sup>

To construct  $\gamma_x^V$ ,<sup>20</sup> attach  $p(n) \cdot (m \cdot p(n))$  copies of  $\gamma_{\text{local}}$  in a square array, where the  $t^{\text{th}}$  horizontal row of wires input to a row of  $\gamma_{\text{local}}$ '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  $\gamma_{\text{local}}$  by constant gates, and the input  $w$  to  $V$  is provided as input to the next  $q(n)$  copies of  $\gamma_{\text{local}}$  by the input gates to  $\gamma_x^V$ . 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  $q_a$ , 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  $\leq p(n)$  and therefore takes space  $\leq p(n)$ ,  $\gamma_x^V$  contains enough gates in the horizontal direction to represent the entire non- $\sqcup$  portion of the tape of  $V(x)$  at every step, and contains enough gates in the vertical direction to simulate  $V(x)$  long enough to get an answer. Since the size of  $\gamma_{\text{local}}$  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,  $|\langle \gamma_x^V \rangle|$  is polynomial in  $n$ , whence  $\gamma_x^V$  is computable from  $x$  in polynomial time.

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

**Theorem 10.8.3.** *3SAT is NP-complete.*

*Proof.*  $3\text{SAT} \in \text{NP}$  for the same reason that  $\text{SAT} \in \text{NP}$ : the language

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

<sup>15</sup> $m$  is the number of bits required to represent a state and a symbol together, plus the boolean value “the tape head is currently here”.

<sup>16</sup>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- $\sqcup$  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.

<sup>17</sup>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.

<sup>18</sup>the configuration strings for the three tape cells surrounding tape cell  $k$

<sup>19</sup>It will be the *number of copies of  $\gamma_{\text{local}}$*  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  $n$ .

<sup>20</sup>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.

is a polynomial-time verification language for 3SAT.

To show that 3SAT is NP-hard, we show that  $\text{CIRCUITSAT} \leq^P \text{3SAT}$ .<sup>21</sup>

Let  $\gamma$  be a Boolean circuit with  $s$  gates; we design a 3CNF formula  $\phi$  computable from  $\gamma$  in polynomial time, which is satisfiable if and only if  $\gamma$  is.<sup>22</sup>  $\phi$  has all the input variables  $x_1, \dots, x_n$  of  $\gamma$ , and *in addition*, for each gate  $g_j$  in  $\gamma$ ,  $\phi$  has a variable  $y_j$  in  $\phi$  representing the values of the output wire of gate  $g_j$ . Assume that  $y_1$  is the output gate of  $\gamma$ .

For each gate  $g_j$ ,  $\phi$  has a subformula  $\psi_j$  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  $z_j$ ,<sup>23</sup> define

$$\psi_j = \begin{cases} \begin{aligned} & (\overline{w_j} \vee \overline{y_j} \vee \overline{y_j}) \\ & \wedge (\overline{w_j} \vee y_j \vee y_j) \end{aligned} & , \text{ if } g_j \text{ is a NOT gate;} \\ \\ \begin{aligned} & (\overline{w_j} \vee \overline{z_j} \vee y_j) \\ & \wedge (\overline{w_j} \vee z_j \vee y_j) \\ & \wedge (\overline{w_j} \vee \overline{z_j} \vee y_j) \\ & \wedge (\overline{w_j} \vee z_j \vee \overline{y_j}) \end{aligned} & , \text{ if } g_j \text{ is an OR gate;} \\ \\ \begin{aligned} & (\overline{w_j} \vee \overline{z_j} \vee y_j) \\ & \wedge (\overline{w_j} \vee z_j \vee \overline{y_j}) \\ & \wedge (\overline{w_j} \vee \overline{z_j} \vee \overline{y_j}) \\ & \wedge (\overline{w_j} \vee z_j \vee y_j) \end{aligned} & , \text{ if } g_j \text{ is an AND gate.} \end{cases} \quad (10.8.1)$$

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

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

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 (10.8.1).

<sup>21</sup>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.

<sup>22</sup> $\phi$  is not equivalent to  $\gamma$ :  $\phi$  has more input bits than  $\gamma$ . But it will be the case that  $\phi$  is satisfiable if and only if  $\gamma$  is satisfiable; it will simply require specifying more bits to exhibit a satisfying assignment for  $\phi$  than for  $\gamma$ .

<sup>23</sup> $w_j$  being the only input if  $g_j$  is a  $\neg$  gate, and each input being either a  $\gamma$ -input variable  $x_i$  or a  $\phi$ -input variable  $y_i$  representing an internal wire in  $\gamma$

To express that  $\gamma$  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  $\phi$  is for  $\gamma$  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  $\gamma$ , for each  $1 \leq j \leq s$ .<sup>24</sup> Thus  $\phi$  is satisfiable if and only if  $\gamma$  is satisfiable.  $\square$

## 10.9 Optional: A brief history of the P versus NP problem

There is a false folklore of the history of the P versus NP problem that is not uncommon to encounter, and the story goes like this: First, computer scientists defined the class P (that part is true) and the class NP (that part is false), and raised this profound question asking whether  $P = NP$ . Then Steven Cook and Leonid Levin discovered the NP-complete problems (that part is true), and this gave everyone the tools to now finally attack the  $P \neq NP$  problem, since by showing that one of the NP-complete problems is in P, this would show that  $P = NP$ .

There is a minor flaw and a major flaw to this story. The minor flaw is that it is widely believed that in fact  $P \neq NP$ . Furthermore, the NP-complete problems are not needed to show this. The existence of a single problem in NP that is not in P suffices to show that  $P \neq NP$ . The problem does not need to be NP-complete for this to work.<sup>25</sup> Cook and Levin were not after a way to resolve P versus NP, which leads to the major flaw in the story: the NP-complete problems came first, not the class NP. The class NP was only defined to capture a large class of problems that SAT is complete *for* (and hopefully large enough to be bigger than P).

No one would care about the class NP if not for the existence of the NP-complete problems. NP is a totally unrealistic model of computation. Showing that a problem is in NP does not automatically lead to a better way to decide it, in any realistic sense.

Beginning in the late 1940s, researchers in the new field of computer science spent a lot of time searching for algorithms for important optimization problems encountered in engineering, physics, and other fields that wanted to make use of the new invention of the computer to automate some of their calculation needs.<sup>26</sup>

<sup>24</sup>That is to say, even if  $x_1x_2 \dots x_n$  is a satisfying assignment to  $\gamma$ , a satisfying assignment to  $\phi$  includes not only  $x_1x_2 \dots x_n$ , but also the correct values of  $y_j$  for each gate  $j$ ; getting these wrong will fail to satisfy  $\phi$  even if  $x_1x_2 \dots x_n$  is correct.

<sup>25</sup>In fact, Andrei Kolmogorov suggested that perhaps the problems that are now known to be NP-complete might contain too much structure to facilitate a proof of intractability, and suggested less natural, but more “random-looking” languages, as candidates for provably intractable problems, whose intractability would imply the intractability of the more natural problems.

<sup>26</sup>Inspired in part by the algorithmic wartime work of some of those same researchers in World War II; Dantzig’s simplex algorithm linear programming was the result of thinking about how to solve problems with linear constraints,



Some of these problems, such as sorting and searching, have obvious efficient algorithms,<sup>27</sup> although clever tricks like divide-and-conquer recursion could reduce the time even further for certain problems such as sorting. Some problems, such as linear programming and maximum flow, seemed to have exponential brute-force algorithms as their only obvious solution, although clever tricks (Dantzig’s simplex algorithm in the case of linear programming, Ford-Fulkerson in the case of network flow)<sup>28</sup> would allow for a much faster algorithm than brute-force search. Still other problems, such as the traveling salesman problem and vertex cover, resisted all efforts to find an efficient exact algorithm.

After decades of effort by hundreds of researchers to solve such problems failed to produce efficient algorithms, many suspected that these problems had no efficient algorithms. The theory of NP-completeness provides an explanation for why these problems are not feasibly solvable. They are important problems regardless of whether they are contained in NP; what makes them most likely intractable is the fact that they are NP-hard. In other words, they are intractable because they are at least as hard as every problem in a class that apparently (though not yet provably) defines a larger class of problems than P. In better-understood models of computation such as finite automata, a simulation of a nondeterministic machine by a deterministic machine necessitates an exponential blowup (in states in the case of finite automata). Our intuition is that, though this has not been proven, the same exponential blowup (in running time now instead of states) is necessary when simulating a NTM with a deterministic TM. The NP-complete problems are believed to be difficult to the extent that this intuition is correct. While not a proof that those problems are intractable, it does imply that if those problems are tractable, then something is seriously wrong with our current understanding of computation, if magical nondeterministic computers could always be simulated by real computers for negligible extra cost.<sup>29</sup>

Prior to the theory of NP-completeness, algorithms researchers – and engineers and scientists in need of those algorithms – were lost in the dark, only able to conclude that a decidable problem was probably too difficult to have a fast algorithm if considerable cost invested in attempting to solve it had failed to produce results. The theory of NP-completeness provides a systematic method of, if not proving, at least *providing evidence* for the intractability of a problem: show that the problem is NP-complete by reducing an existing NP-complete

---

such as: “If transportation with trucks costs \$10/soldier, \$4/ton of food, subject to the constraint that each truck can carry weight subject to the linear tradeoff of 1 ton of food for every 10 soldiers, and each soldier requires 20 pounds of food/week, how can we move the maximum number of soldiers in 30 trucks, for less than \$50,000?”

<sup>27</sup>Though it would not be until the late 1960’s that anyone would even think of polynomial time as the default definition of “efficient”.

<sup>28</sup>Although technically the original versions of both of those methods were exponential time in the worst case, they nonetheless avoided brute-force search in ingenious ways, and led eventually to provably polynomial-time algorithms.

<sup>29</sup>We have deeper reasons for thinking that  $P \neq NP$  besides the hand-waving argument, “C’mon! How could they be equal?!” For example, cryptography as we know it would be impossible, and the discovery of proofs of mathematical theorems could be automated. In a depressing philosophical sense, creativity itself could be automated: as Scott Aaronson said, “*Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett.*”

<http://www.scottaaronson.com/blog/?p=122>



problem to it.<sup>30</sup>

Scott Aaronson’s paper *NP-complete Problems and Physical Reality*<sup>31</sup> contains convincing arguments justifying the intuition that NP contains fundamentally hard problems. Anyone who would considering taking the position that “The lack of a proof that  $P \neq NP$  implies that there is a non-negligible chance that NP-complete problems are tractable”, owes it to themselves to read that paper.

---

<sup>30</sup>For this we probably have Richard Karp to thank more than Cook or Levin, who, one year after the publication of Cook’s 1971 paper showing SAT is NP-complete, published a paper listing 21 famous, important and practical decision problems, involving graphs, job scheduling, circuits, and other combinatorial objects, showing them all to be NP-complete by reducing SAT to them (or reducing them to each other). After that, the floodgates opened, and by the mid-1970’s, hundreds of natural problems had been shown to be NP-complete.

<sup>31</sup><http://arxiv.org/abs/quant-ph/0502072>



# Chapter 11

## Undecidability

This chapter is devoted to proving *absolute limitations* on the fundamental capabilities of algorithms. Certain problems are not solvable by any algorithm; they are *undecidable*. As we mentioned at the end of the chapter on Turing machines, we will use the terms *algorithm* and *Turing machine* interchangeably.

### 11.1 The Halting Problem

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

$$\text{HALTS} = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on input } w \}.$$

HALTS is sometimes denoted  $K$ ,  $H_{\text{TM}}$ , or  $0'$ .

Note that HALTS is Turing-recognizable, via the following algorithm  $H_r$ :

$H_r$  = “On input  $\langle M, w \rangle$ , where  $M$  is an algorithm and  $w$  is a string:

1. Simulate  $M$  on input  $w$ .
2. If  $M$  ever halts, *accept*.”

This can be implemented in Python via

```
1 def halt_recognizer(M_src, w):
2     """M is the source code of a Python function named M,
3     and w is an input to M."""
4     namespace = {}
5     exec(M_src, namespace) # this defines the function M
6     M = namespace["M"] # find function M defined when
7                        # code in M_src executes
8     M(w) # this actually executes the function on input w
9     return True # this is only reached if M(w) halts
```

Here is `halt_recognizer` in action on a few strings representing Python source code.

```

1 M_src = '''
2 def M(w):
3     """Indicates if x has > 5 occurrences of the symbol "a"."""
4     count = 0
5     for c in w:
6         if c == "a":
7             count += 1
8     return count > 5
9 '''
10 w = "abcbabcaaabcaa"
11 print(halt_recognizer(M_src, w)) # prints True since M always halts
12
13 M_src = '''
14 def M(w):
15     """Has a potential infinite loop, depending on input w."""
16     count = 0
17     while count < 5:
18         c = w[0]
19         if c == "a":
20             count += 1
21     return count > 5
22 '''
23 w = "abcbabcaaabcaa"
24 print(halt_recognizer(M_src, w)) # prints True since w[0] == "a"
25
26 # WARNING: will not halt!
27 # After running this, you'll need to kill the interpreter
28 w = "bcabcaaabcaa"
29 print(halt_recognizer(M_src, w)) # won't halt since w[0] != "a"

```

**A note about programs versus source code of programs.** We have taken the convention of representing “programs/algorithms” as Python functions. Above, we showed that there is a Python function `halt_recognizer` that recognizes HALTS taking as input two *strings* `M_src` and `w`. The built-in Python function `exec` gives a way to start from Python source code defining a function and convert it into an actual Python function that can be called. It is worth noting that these are two different types of objects, similarly to the fact that a string  $\langle G \rangle$  representing a graph  $G = (V, E)$  is a different type of object than  $G$  itself.  $G$  is a pair consisting of a set  $V$  and a set  $E$  of pairs of elements from  $V$ , whereas  $\langle G \rangle$  is a sequence of symbols from the alphabet  $\{0, 1\}^*$ . However, the encoding function we defined allows us to interconvert between these two, and it is sometimes more convenient to speak of  $G$  as though it is a graph, and sometimes more convenient to speak of  $G$  as though it is a string representing a graph.

Similarly, Python gives an easy way to interconvert between Python functions that can be called, and strings that contain properly formatted Python source code describing those functions. To convert source code to a function, use `exec` as above. To convert a function to source code, use the `inspect` library. The following code snippet shows how to do this, and even how to do a simple modification on the source code that changes its behavior, which is reflected if the code is “re-compiled” using `exec`.

With this in mind, we can rewrite the above code for `halt_recognizer` to deal directly with Python functions rather than source code, knowing that if we had source code instead, we could simply use `exec` to convert it to a Python function.

```

1 def halt_recognizer(M,w):
2     """M is a Python function, and w is an input to M."""
3     M(w)          # this executes the function M on input w
4     return True   # this is only reached if M(w) halts

1 def M(w):
2     """Indicates if x has > 5 occurrences of the symbol "a"."""
3     count = 0
4     for c in w:
5         if c == "a":
6             count += 1
7     return count > 5

8
9 w = "abcabcaaabcaa"
10 halt_recognizer(M,w) # will halt since M always halts
11
12 def M(w):
13     """Has a potential infinite loop."""
14     count = 0
15     while count < 5:
16         c = w[0]
17         if c == "a":
18             count += 1
19     return count > 5
20
21 w = "abcabcaaabcaa"

```

```

22 halt_recognizer(M,w) # will halt since w[0] == "a"
23
24 # WARNING: will not halt!
25 # After running this, you'll need to shut down the interpreter
26 w = "bcabcaaabcaa"
27 halt_recognizer(M,w) # will not halt since w[0] != "a"

```

We will eventually show that HALTS is not decidable:

**Theorem 11.1.1.** *HALTS is not decidable.*

But before getting to it, we will first use the machinery of reductions, together with Theorem 11.1.1, to show that other problems are undecidable, by showing that if they *could* be decided, then so could HALTS, contradicting Theorem 11.1.1.

### 11.1.1 Reducibility

Although it is standard to use reductions as a formal tool to prove problems are undecidable, we don't formally use them in this chapter. For the curious, they are the same as the (mapping) reductions used for NP-completeness, but without the polynomial-time constraint. We also allow a more general type of reduction (known as a *Turing reduction*), which is closer to what most people would think of as “using one algorithm as a subroutine to write another”.

The problem  $A$  is (Turing) reducible to problem  $B$  if “an algorithm  $M_B$  for  $B$  can be used as a subroutine to create an algorithm  $M_A$  for  $A$ ”. This is formally defined with something called oracle Turing machines, but the concept is intuitively clear enough that we won't go into the details. A mapping reduction  $f$  reducing  $A$  to  $B$  is the special case when the algorithm  $M_A$  is of the form “on input  $x$ , compute  $y = f(x)$  and return  $M_B(y)$ .” In other words,  $M_B$  is called as a subroutine exactly once, at the end of the algorithm, and its answer is returned as the answer for the whole algorithm. With more general reductions, we allow the reduction to change the answer, or to call  $M_B$  more than once.

end of lecture 9b

## 11.2 Undecidable problems about algorithm behavior

### 11.2.1 No-input halting problem

Define the *no-input halting problem*  $\text{HALTS}_\varepsilon = \{ \langle M \rangle \mid M \text{ is a TM and } M(\varepsilon) \text{ halts} \}$ .

First, suppose we knew already that  $\text{HALTS}_\varepsilon$  was undecidable, but not whether HALTS was undecidable. Then we could easily use the undecidability of  $\text{HALTS}_\varepsilon$  to prove that HALTS is undecidable, by showing a reduction from  $\text{HALTS}_\varepsilon$  to HALTS: suppose for the sake of contradiction that HALTS is decidable by TM  $H$ . Then to decide whether  $M(\varepsilon)$  halts, we use call  $H$  with inputs  $\langle M, \varepsilon \rangle$  to decide whether  $M(\varepsilon)$  halts.

That is, an instance  $\langle M \rangle$  of  $\text{HALTS}_\varepsilon$  is a simple special case of an instance  $\langle M, w \rangle$  of  $\text{HALTS}$  (where  $w = \varepsilon$ ), so  $\text{HALTS}$  is at least as difficult to decide as  $\text{HALTS}_\varepsilon$ . Proving that  $\text{HALTS}_\varepsilon$  is undecidable means showing the other direction:  $\text{HALTS}_\varepsilon$  is at least as difficult to decide as  $\text{HALTS}$ .

**Theorem 11.2.1.**  $\text{HALTS}_\varepsilon$  is undecidable.

*Proof.* Suppose for the sake of contradiction that  $\text{HALTS}_\varepsilon$  is decidable by algorithm  $H_\varepsilon$ . Define the algorithm  $H$  deciding  $\text{HALTS}$  as

$H(\langle M, w \rangle)$ :

1) Define the algorithm  $T_{M,w}$  based on  $M$  and  $w$  as:

$T_{M,w}(x)$ :

1) Run  $M(w)$

2) Run  $H_\varepsilon(\langle T_{M,w} \rangle)$  and return its answer.

If  $M(w)$  halts, then  $T_{M,w}$  halts on all inputs (including  $\varepsilon$ ), and if  $M(w)$  does not halt, then  $T_{M,w}$  halts on no inputs (including  $\varepsilon$ ). Then  $H$  decides  $\text{HALTS}$ , a contradiction.  $\square$

The following Python code implements this idea. Note that  $T_{M,w}$  depends on  $M$  and  $w$ , so the program will behave differently if different  $M$  or  $w$  are passed to  $H$ .  $M$  and  $w$  are “hardcoded constants” in  $T_{M,w}$ .

```

1 def H_e(M):
2     """Function that supposedly decides whether M("") halts."""
3     raise NotImplementedError()
4
5 def H(M,w):
6     """Decider for HALTS that works assuming H_e works, giving
7     a contradiction and proving H_e cannot be implemented."""
8     def T_Mw(x):
9         """Halts on every string if M halts on w, and loops on
10         every string if M loops on w."""
11         M(w)
12     return H_e(T_Mw)

```

Pay particular attention to what  $H$  is doing: importantly, it is *not running*  $T_{M,w}$ . It *defines* the algorithm  $T_{M,w}$ . However, it is possible to define an algorithm, meaning, to state what steps would be executed if the algorithm were to run, without actually running it. Every time you are programming, you do this. You write code, but the code doesn’t execute until you run it. Similarly, when a Python function like `H` executes the `def` statement defining the “local function” `T_Mw`, this creates the function, but it does not execute the function. The next line `return H_e(T_Mw)` similarly does not execute `T_Mw`. It merely passes the function object as input to another function.

For example, the following code defines `f`:

```

1 def f(x):
2     print("Returning {} + 1".format(x))
3     return x+1

```

Run it. Nothing happens. It *defined* `f`, but did not *run* `f`. Now run this code:

```

1 def f(x):
2     print("Returning {} + 1".format(x))
3     return x+1
4
5 print("Returned value: {}".format(f(3)))

```

It should print

```

Returning 3 + 1
Returned value: 4

```

Because after defining `f`, it actually runs `f`.

Intuitively, think of the difference between *library* code, versus a program with a `main` function. Library code generally doesn't run when you import the library. For instance, executing `import math` in Python gives you access to all the functions (such as `math.exp`) defined in the `math` library, but doesn't run any of them.

### 11.2.2 Accepting a given string

Define

$$\text{ACCEPTS} = \{\langle M, w \rangle \mid M \text{ is a TM that accepts } w\}.$$

**Theorem 11.2.2.** *ACCEPTS is undecidable.*

*Proof.* Suppose for the sake of contradiction that `ACCEPTS` is decidable by algorithm `A`.

Define the algorithm  $H_\epsilon$  deciding `HALTSε` as:

$H_\epsilon(\langle M \rangle)$ :

1) Define the algorithm  $T_M$  based on  $M$  as:

$T_M(x)$ :

1) Run  $M(\epsilon)$

2) accept

2) Run  $A(\langle T_M \rangle, 011)$  and return its answer.

Since  $T_M$  runs  $M(\epsilon)$ , if  $M(\epsilon)$  loops, then  $T_M$  loops on every input, including 011. Conversely, if  $M(\epsilon)$  halts, then  $T_M$  accepts every input, including 011. Since  $A$  correctly decides `ACCEPTS`,  $A(\langle T_M \rangle, 011)$  accepts  $\iff T_M(011)$  accepts  $\iff M(\epsilon)$  halts. Thus  $H_\epsilon$  decides `HALTSε`, a contradiction.  $\square$

We can implement this in Python with

```

1 def A(M,w):
2     """Function that supposedly decides whether M(w) accepts,
3     where M is a function and w an input to M."""
4     raise NotImplementedError()
5
6 def H_e(M):
7     """Decider for HALTS_epsilon that works assuming A works,
8     giving a contradiction and proving A cannot be implemented."""
9     def T_M(x):

```



```

10     M("")
11     return True
12     return A(T_M, "011")

```

### 11.2.3 Accepting at least one string

Define

$$\text{EMPTY} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}.$$

Here is an example of an undecidability proof that shows how to use a purported decider  $E$  for EMPTY to decide  $\text{HALTS}_\varepsilon$ , but that must negate the Boolean answer given by  $E$ .

**Theorem 11.2.3.** *EMPTY is undecidable.*

*Proof.* Suppose for the sake of contradiction that EMPTY is decidable by algorithm  $E$ . Define the algorithm  $H_\varepsilon$  deciding  $\text{HALTS}_\varepsilon$  as:

$H_\varepsilon(\langle M \rangle)$ :

1) Define the algorithm

$T_M(x)$ :

1) Run  $M(\varepsilon)$

2) accept

2) Run  $E(\langle T_M \rangle)$  and accept if and only if it rejects

Then for algorithm  $M$ ,

$$\begin{array}{lll}
 M(\varepsilon) \text{ halts} & \iff L(T_M) \neq \emptyset & \text{defn of } T_M \\
 & \iff E \text{ rejects } T_M & \text{since } E \text{ decides EMPTY} \\
 & \iff H_\varepsilon \text{ accepts } \langle M \rangle. & \text{line (2) of } H_\varepsilon
 \end{array}$$

Since  $H$  always halts, it decides  $\text{HALTS}_\varepsilon$ , a contradiction. □

The following Python code implements this idea:

```

1  def E(M):
2      """Function that supposedly decides whether L(M) is empty."""
3      raise NotImplementedError()
4
5  def H_e(M):
6      """Decider for HALTS_epsilon that works assuming E works,
7      giving a contradiction and proving E cannot be implemented."""
8      def T_M(x):
9          """Accepts every string if M halts on "", and loops on
10          every string if M loops on ""."""
11          M("")
12          return True
13      return not E(T_M)

```

### 11.2.4 Rejecting at least one string

Define

$$\text{REJ} = \{ \langle M \rangle \mid M \text{ is a TM that rejects at least one input} \}.$$

Here is an example of an undecidability proof that shows how to use a purported decider  $R$  for REJ to decide  $\text{HALTS}_\varepsilon$ , but that must define an algorithm  $T_M$  that has a different behavior on line (2) than the previous proofs (which all had  $T_M$  accept on line (2)).

This illustrates the general format of an undecidability proof for a question about the behavior of programs: reduce the halting problem to the question about the behavior, by defining an algorithm  $T_M$  that runs  $M(\varepsilon)$ , and if  $M(\varepsilon)$  ever halts,  $T_M$ 's next action exhibits the behavior under question (in this example, the behavior is "rejects at least one input"). Thus  $T_M$  exhibits the behavior if and only if  $M(\varepsilon)$  halts.

**Theorem 11.2.4.** *REJ is undecidable.*

*Proof.* Suppose for the sake of contradiction that REJ is decidable by algorithm  $R$ . Define the algorithm  $H_\varepsilon$  deciding  $\text{HALTS}_\varepsilon$  as:

$H_\varepsilon(\langle M \rangle)$ :

1) Define the algorithm

$T_M(x)$ :

1) Run  $M(\varepsilon)$

2) reject

2) Run  $R(\langle T_M \rangle)$  return its answer

Then for algorithm  $M$ ,

$$\begin{array}{lll} M(\varepsilon) \text{ halts} & \iff T_M \text{ rejects at least one input} & \text{defn of } T_M \\ & \iff R \text{ accepts } T_M & \text{since } R \text{ decides REJ} \\ & \iff H_\varepsilon \text{ accepts } \langle M \rangle. & \text{line (2) of } H_\varepsilon \end{array}$$

Since  $H$  always halts, it decides  $\text{HALTS}_\varepsilon$ , a contradiction. □

The following Python code implements this idea:

```

1 def E(M):
2     """Function that supposedly decides whether L(M) is empty."""
3     raise NotImplementedError()
4
5 def H_e(M):
6     """Decider for HALTS_epsilon that works assuming E works,
7     giving a contradiction and proving E cannot be implemented."""
8     def T_M(x):
9         """Rejects every string if M halts on "", and loops on
10        every string if M loops on ""."""
11        M("")
12        return False
13    return R(T_M)

```

**11.2.5 How to recognize undecidability at a glance**

All of the reductions shown above are shown via “effectively” the same reduction from  $\text{HALTS}_\varepsilon$ . To determine if  $M(\varepsilon)$  halts, assuming we have an algorithm  $A$  for deciding some other “behavior” of an algorithm  $T_M$  (such as “ $T_M$  accepts at least one string” or “ $T_M$  rejects the string 011”), design a new algorithm  $T_M$  that

- 1) runs  $M(\varepsilon)$ , and
- 2) if  $M(\varepsilon)$  ever halts, make  $T_M$  show the behavior that  $A$  can supposedly decide.

Then  $T_M$  has that behavior if and only if  $M(\varepsilon)$  halts. This means that essentially any question about the “eventual behavior” of an algorithm is undecidable.

The following are all questions about the eventually behavior of an algorithm, which can be proven undecidable by showing that the halting problem can be reduced to them. This includes questions of the form, given a program  $P$  and input  $x$ :

- Does  $P$  accept  $x$ ?
- Does  $P$  reject  $x$ ?
- Does  $P$  loop on  $x$ ?
- Does  $P(x)$  ever execute line 320?
- Does  $P(x)$  ever call subroutine  $f$ ?
- Does  $P(x)$  return the value 011?
- Does  $P(x)$  ever raise an exception?

Similar questions about general program behavior over all strings are also undecidable, i.e., given a program  $P$ :

- Does  $P$  accept at least one input?
- Does  $P$  accept all inputs?
- Is  $P$  total? (i.e., does it halt on all inputs?)
- Is there an input  $x$  such that  $P(x)$  returns the value 42? (This is undecidable even when we restrict to total programs  $P$ ; so even if we knew in advance that  $P$  halts on all inputs, there is no algorithm that can find an input  $x$  such  $P(x)$  returns the value 42)
- Does some input cause  $P$  to call subroutine  $f$ ?
- Do all inputs cause  $P$  to call subroutine  $f$ ?
- Does  $P$  decide the same language as some other program  $Q$ ?

- Does  $P$  accept any inputs that represent a prime number?
- Does  $P$  decide the language  $\{\langle n \rangle \mid n \text{ is prime}\}$ ?
- Is  $P$  a polynomial-time machine? (this is not decidable even if we are promised that  $P$  is total)
- Is  $P$ 's running time  $O(n^3)$ ?

However, there are certainly questions one can ask about a program  $P$  that are decidable. For example, if the question is about the *syntax* instead of the *behavior*, then the problem can be decidable:

- Does  $P$  have at least 100 lines of code? (Turing machine equivalent of this question would be, “Does  $M$  have at least 100 states?”)
- (Interpreting  $P$  as a TM) Does  $P$  have 2 and 3 in its input alphabet?
- Does  $P$  halt immediately because its first line a return statement not calling another function? (Turing machine equivalent of this question would be, “Does  $P$  halt immediately because its start state is equal to the accept or reject state?”)
- Does  $P$  have a subroutine named  $f$ ?
- (Interpreting  $P$  as a TM) Do  $P$ 's transitions always move the tape head right? (If so, then we could decide whether  $P$  halts on a given input, because it either halts after moving the tape head beyond the input upon reading enough blanks, or repeats a non-halting state while reading blanks and will therefore move the tape head right forever.)
- Does  $P$  “clock itself” to halt after  $n^3$  steps on inputs of length  $n$ , guaranteeing that its running time is  $O(n^3)$ ? One way to do this is, before doing anything else, to write the string  $0^{n^3}$  on a worktape, then reset the tape head to the left, then move that tape head right once per step (in addition to the computation being done with all the other tapes), and immediately halt when that worktape reaches the blank to the right of the last 0.<sup>1</sup>

Furthermore, many questions about the behavior of  $P$  on input  $x$  actually *are* decidable, as long as the question contains a “time bound” that restricts the search, for example, given  $P$  and  $x$ :

- Does  $P(x)$  halt within 20 steps?

---

<sup>1</sup>Note this is not the same as asking if  $P$  has running time  $O(n^3)$ ; it is asking whether this one particular method is employed to guarantee a running time of  $O(n^3)$ . If  $P$  simply had three nested for loops, each counting up to  $n$ , but not “self-clocking” as described, then the answer to this question would be “no” even though  $P$  is an  $O(n^3)$  program.

- Does  $P(x)$  halt within  $n^3$  steps, where  $n = |x|$ ? Note this is not the same as asking whether  $P$  is an  $O(n^3)$ -time program, which is a question about *all* inputs  $x$ .  $P$  could take time  $n^3$  on this  $x$ , but time  $2^{|y|}$  on some other input  $y$ , so not have worst-case running time of  $O(n^3)$ .
- Is 21 the fifth line of code  $P(x)$  executes? (Turing machine equivalent: Is  $q_{21}$  the fifth state  $P(x)$  visits?<sup>2</sup>)
- Does  $P(x)$  (interpreted as a single-tape TM) ever move its tape head beyond position  $n^3$ ? (where  $n = |x|$ ) This one takes a bit more work to see.

Suppose  $P(x)$  does not move its tape head beyond position  $n^3$ . Then  $P(x)$  has a finite number of reachable configurations, specifically,  $|Q| \cdot n^3 \cdot |\Gamma|^{n^3}$ , since there are  $|Q|$  states,  $n^3$  possible tape head positions, and  $|\Gamma|^{n^3}$  possible values for the tape content. If one of these configurations repeats, then  $P$  runs forever. Otherwise, by the pigeonhole principle,  $P(x)$  can visit no more than  $|Q| \cdot n^3 \cdot |\Gamma|^{n^3}$  configurations before halting.

So to decide if  $P(x)$  ever moves its tape head beyond position  $n^3$ , run it until it either moves its tape head beyond position  $n^3$ , so the answer is “yes”, or repeats a configuration so the answer is “no”, since it will now repeat forever the same configurations that don’t move the tape head beyond  $n^3$ . If neither of these happens,  $P(x)$  will halt and the answer is “no”.

end of lecture 9c

### 11.3 Optional: Enumerators as an alternative definition of Turing-recognizable

An *enumerator* is a TM  $E$  with a “printer” attached. It takes no input, and runs forever. Every so often it prints a string.<sup>3</sup> The set of strings printed is the *language enumerated by*  $E$ .

This is a Python implementation of an enumerator that enumerates the prime numbers, which uses a Python trick called *generators* that let one iterate over values without explicitly creating a list of those values (so they can iterate over infinitely many values potentially):

<sup>2</sup>Note that even the following question is decidable: given  $P$ , is  $q_{21}$  the fifth state visited on *every* input? Although there are an infinite number of inputs, the fifth transition can only read some bounded portion of the input, so by searching over the  $2^5 = 32$  possible prefixes of length 5 (the only portion of the input that can be scanned in the first 5 steps), we can answer the question without needing to know the input beyond that prefix.

<sup>3</sup>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  $\varepsilon$  by placing the tape head on a  $\sqcup$  symbol), and enters a special state  $q_{\text{print}}$ . 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  $q_{\text{print}}$ . In the Python code, we implement enumerators with *generators* using the `yield` statement: <https://wiki.python.org/moin/Generators>.

```

1 def is_prime(n):
2     """Check if n is prime."""
3     if n < 2: # 0 and 1 are not primes
4         return False
5     for d in xrange(2, int(n**0.5)+1):
6         if n % d == 0:
7             return False
8     return True
9
10 import itertools
11 def primes_enumerator():
12     """Iterate over all prime numbers."""
13     for n in itertools.count(): #iterates over all natural numbers
14         if is_prime(n):
15             yield n

```

Running the following code prints all the prime numbers:

```

1 for n in primes_enumerator():
2     print(n)

```

Of course, don't run that! If we just want to assure ourselves it works, let's print the first 100 prime numbers:

```

1 # print first 100 numbers returned from primes_enumerator()
2 for (p,_) in zip(primes_enumerator(), range(100)):
3     print(p, end=" ")

```

Recall that a language is Turing-recognizable if there is a TM recognizing it. In other words,  $A$  is Turing-recognizable if there is a TM  $M$  with input alphabet  $\Sigma$  such that, for all  $x \in \Sigma^*$ , if  $x \in A$ , then  $M$  accepts  $x$ , and if  $x \notin A$ , then  $M$  either rejects or loops on  $x$ .

**Theorem 11.3.1.** *A language  $A$  is Turing-recognizable if and only if it is enumerated by some enumerator.*

This is why such languages are sometimes called *computably enumerable*.

*Proof.* ( $\Leftarrow$ ): Let  $E$  be an enumerator that enumerates  $A$ . Define  $M(w)$  as follows: run  $E$ . Whenever  $E$  prints a string  $x$ , *accept* if  $x = w$ , and keep running  $E$  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  $E$ .

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

Define  $E$  as follows:

for  $i = 1, 2, 3, \dots$ :

    for  $j = 1, 2, 3, \dots, i$ :

        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  $s_j$ . Furthermore,  $E$  prints only strings accepted by  $M$ , so  $E$  enumerates exactly the strings accepted by  $M$ .  $\square$

The following Python code implements the “convert enumerator to acceptor” direction:

```

1 def create_acceptor_from_enumerator(enumerator):
2     def acceptor(input_to_acceptor):
3         for enumerated_output in enumerator():
4             if enumerated_output == input_to_acceptor:
5                 return True
6         return False # if it has a finite language it might halt
7     return acceptor

```

So, if we give it the primes enumerator above, it creates a Python function that recognizes prime numbers. (but doesn’t halt on composites!)

```

1 primes_acceptor = create_acceptor_from_enumerator(primes_enumerator)
2 print(primes_acceptor(2)) # prints True
3 print(primes_acceptor(3)) # prints True
4 print(primes_acceptor(5)) # prints True
5 print(primes_acceptor(7)) # prints True
6 print(primes_acceptor(9)) # runs forever

```

Python doesn’t have a way to call a function and run it for only a certain number of steps, so there is no direct Python implementation of the idea of the other direction in the proof of Theorem 11.3.1. We can do something similar, however, using the Python threading library. To take an algorithm  $M$  recognizing  $A$ , and create an enumerator  $E$  enumerating  $A$ , we can run an infinite loop that starts  $A$  on each possible input  $x \in \Sigma^*$  in a separate thread. If  $M(x)$  does not halt then the thread will never terminate, but for any  $M(x)$  that does halt,  $E$  can check whether it accepted and print  $x$  if so.

```

1 import sys, itertools, threading
2
3 # for python 3 include the next line
4 from queue import Queue
5
6 # for python 2 include the next line
7 # from Queue import Queue
8
9 def binary_strings():
10     """Enumerates all binary strings in length-lexicographical order."""
11     for length in itertools.count():
12         for s in itertools.product(["0", "1"], repeat=length):
13             yield "".join(s)
14
15 #WARNING: this runs forever, consuming processor and memory and has to be killed
16 def enumerator_from_recognizer(recognizer, inputs=binary_strings()):
17     """Given recognizer, a function that defines a language by returning True if
18     its input string is in the language and otherwise either returning False
19     or looping, this function enumerates over strings in the language.
20

```

```

21     inputs is iterator yielding valid inputs to recognizer. Default is binary
22     strings in lexicographical order."""
23     # define queue to put outputs (and its associated input) into
24     outputs_queue = Queue()
25     def compute_output_and_add_to_queue(input_string):
26         output = recognizer(input_string)
27         if output == True:
28             outputs_queue.put((input_string, output))
29
30     # run recognizer on all possible inputs and add outputs to outputs_queue
31     recognizer_threads = []
32     def start_all_recognizer_threads():
33         for input_string in inputs:
34             thread = threading.Thread(target=compute_output_and_add_to_queue,
35                                     args=[input_string])
36             thread.daemon = True
37             thread.start()
38             recognizer_threads.append(thread)
39     # if finite number of inputs, wait until all threads have stopped
40     # and then add a new item to the queue to indicate all inputs are done
41     for thread in recognizer_threads:
42         thread.join()
43     outputs_queue.put((None, None))
44
45     starter_thread = threading.Thread(target=start_all_recognizer_threads)
46     starter_thread.daemon = True # ensures thread is killed when main thread stops
47     starter_thread.start()
48
49     # search for True output in outputs_queue
50     while True:
51         (input_string, output) = outputs_queue.get()
52         if output == None:
53             break
54         yield input_string
55
56
57 #WARNING: this runs forever, consuming processor and memory and has to be killed
58 def create_enumerator_from_recognizer(recognizer, inputs=binary_strings()):
59     """Given recognizer, a function that defines a language by returning True
60     if its input string is in the language and otherwise either returning False
61     or looping, it returns an iterator that iterates over strings in the language.
62
63     inputs is iterator yielding valid inputs to recognizer. Default is binary
64     strings in lexicographical order."""
65     def enumerator():
66         return enumerator_from_recognizer(recognizer, inputs)
67     return enumerator

```

We can test it out, but if you actually run this, be sure to kill the process manually. It will run forever and continue to consume resources.

```

1 # for python 2 include the next line
2 # from future_builtins import zip
3

```



```

4 import itertools
5 enumerate_primes_created_from_recognizer = create_enumerator_from_recognizer(
    is_prime, inputs=itertools.count())
6
7 #WARNING: this will run forever and consume processor and memory and have to be
    killed
8 for i,p in zip(itertools.count(), enumerate_primes_created_from_recognizer()):
9     print("{}'th prime is {}".format(i, p))
10    if i > 10:
11        break

```

A more reasonable testing option is to set `inputs` to iterate over a bounded number of inputs:

```

1 import itertools
2 enumerate_primes = create_enumerator_from_recognizer(is_prime, inputs=range(100))
3
4 for i,p in zip(itertools.count(), enumerate_primes()):
5     print("{}'th prime is {}".format(i, p))

```

### 11.3.1 Optional: A non-Turing-recognizable language

The language HALTS is Turing-recognizable, but not decidable. Since it is not decidable, its complement is not decidable. Is its complement  $\overline{\text{HALTS}}$  Turing-recognizable? In other words, is HALTS co-Turing-recognizable?

We will show it is not, using the following theorem.

**Theorem 11.3.2.** *A language is decidable if and only if it is Turing-recognizable and co-Turing-recognizable (i.e., its complement is Turing-recognizable).*

*Proof.* We prove each direction separately.

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

**(Turing-recognizable and co-Turing-recognizable  $\implies$  decidable):** Let  $A$  be Turing-recognizable and co-Turing-recognizable, let  $M_A$  be a TM recognizing  $A$ , and let  $M_{\overline{A}}$  be a TM recognizing  $\overline{A}$ . Define the TM  $M$  as follows. On input  $w$ ,  $M$  runs  $M_A(w)$  and  $M_{\overline{A}}(w)$  in parallel. One of them will accept since either  $w \in A$  or  $w \in \overline{A}$ . If  $M_A$  accepts first, then  $M$  accepts  $w$ , and if  $M_{\overline{A}}$  accepts  $w$  first, then  $M(w)$  rejects. So  $M$  decides  $A$ .  $\square$

The following is a an implementation using the original definition of Turing-recognizable, running two TMs in parallel using Python's threading library.

```

1 # for python 3 include the next line
2 from queue import Queue
3

```

```

4 # for python 2 include the next line
5 # from Queue import Queue
6
7 import threading
8
9 def decider_from_recognizers(recognizer, comp_recognizer, w):
10     """recognizer is function recognizing some language A, and
11     comp_recognizer is function recognizing complement of A."""
12     outputs_queue = Queue()
13
14     def recognizer_add_to_queue():
15         output = recognizer(w)
16         if output == True:
17             outputs_queue.put("w in A")
18
19     def comp_recognizer_add_to_queue():
20         output = comp_recognizer(w)
21         if output == True:
22             outputs_queue.put("w not in A")
23
24     t1 = threading.Thread(target = recognizer_add_to_queue)
25     t2 = threading.Thread(target = comp_recognizer_add_to_queue)
26     t1.daemon = t2.daemon = True
27     t1.start()
28     t2.start()
29
30     # exactly one of the threads will put a message in the queue
31     message = outputs_queue.get()
32     if message == "w in A":
33         return True
34     elif message == "w not in A":
35         return False
36     else:
37         raise AssertionError("should not be reachable")
38
39 def create_decider_from_recognizers(recognizer, comp_recognizer):
40     def decider(w):
41         return decider_from_recognizers(recognizer, comp_recognizer, w)
42     return decider

```

We can also use the enumerator characterization of Turing-recognizable languages to get simpler code for the direction (**Turing-recognizable and co-Turing-recognizable  $\implies$  decidable**). Essentially, run two enumerators in parallel (the syntax for this is very simple in Python), and each time one produces a string, test for equality with  $w$ . Which enumerator produces  $w$  indicates whether to accept or reject. Python actually gives a nice way using the `zip` function to alternate a string produced by one enumerator with a string produced by the other (but only the version of `zip` in Python 3, which returns an iterator; Python 2's `zip` will attempt to build an infinite list and won't halt):

```

1 from future_builtins import zip # only Python 3 version of zip works
2
3 def decider_from_enumerators(enumerator, comp_enumerator, w):

```

```

4  """enumerator is iterator that enumerates some language A, and
5  comp_enumerator is iterator enumerating complement of A."""
6  for (x_yes, x_no) in zip enumerator, comp_enumerator):
7      if w == x_yes:
8          return True
9      if w == x_no:
10         return False

```

The above is shorthand for the following code, which is a more verbose way to use iterators (it also assumes that each enumerator produces an infinite number of strings; otherwise, calling `.next()` raises a `StopIterator` exception if there is no “next” string):<sup>4</sup>

```

1  def decider_from_enumerators(enumerator, comp_enumerator, w):
2      """enumerator is iterator that enumerates some language A, and
3      comp_enumerator is iterator enumerating complement of A."""
4      while True:
5          x_yes = enumerator.next()
6          if w == x_yes:
7              return True
8          x_no = comp_enumerator.next()
9          if w == x_no:
10             return False

```

We can test it as follows. Here, we make recognizers for the problem  $A =$  “is a given number prime?”, and its complement  $\bar{A} =$  “is a given number composite?”, and we intentionally let the recognizers run forever when the answer is no, in order to demonstrate that the decider returned by `create_decider_from_recognizers` still halts.<sup>5</sup>

```

1  def loop():
2      """Loop forever."""
3      while True:
4          pass
5
6  def prime_recognizer(n):
7      """Check if n is prime."""
8      if n < 2: # 0 and 1 are not primes
9          loop()
10     for x in range(2, int(n**0.5)+1):
11         if n % x == 0:
12             loop()
13     return True
14
15 def composite_recognizer(n):
16     if n < 2: # 0 and 1 are not primes
17         return True
18     for x in xrange(2, int(n**0.5)+1):

```

<sup>4</sup>Of course, if that ever happens, for example if `enumerator` runs out of strings, then it means that the language has only a finite number of strings, so if none of them were  $w$ , then  $w$  is not in the language and we should reject.

<sup>5</sup>If you run this from the command line, it will halt as expected. Unfortunately, if you run this code within ipython notebook, the threads continue to run even after the notebook cell has been executed, since the threads run as long as the program that created them is still going, and the “program” is considered the *whole notebook*, not an individual cell. To shut down these threads and keep them from continuing to run and consume resources, you will have to shutdown the ipython notebook.

```

19         if n % x == 0:
20             return True
21     loop()
22
23 #WARNING: will continue to consume resources after it halts
24 # because it starts up threads that run forever
25 prime_decider = create_decider_from_recognizers(prime_recognizer,
26         composite_recognizer)
27 for n in range(2,20):
28     print("{:2} is prime? {}".format(n, prime_decider(n)))

```

**Corollary 11.3.3.**  $\overline{\text{HALTS}}$  is not Turing-recognizable.

*Proof.* HALTS is Turing-recognizable. If  $\overline{\text{HALTS}}$  were Turing-recognizable, then HALTS would be decidable by Theorem 11.3.2, a contradiction.  $\square$

Note the key fact used in the proof of Corollary 11.3.3 is that the class of Turing-recognizable languages is *not* closed under complement. Hence *any* language that is Turing-recognizable but not decidable (such as HALTS) has a complement that is not Turing-recognizable.

## 11.4 The Halting Problem is undecidable

We have used the fact that the Halting Problem is undecidable to prove that several other problems are also undecidable. But how do we prove a problem is undecidable in the first place, if we don't already know of one that we can use with reductions? We now move on to proving directly that the Halting Problem is undecidable, using Turing's original technique used to prove this first in 1936. He actually used a clever technique that was already several decades old at the time, called *diagonalization*.

### 11.4.1 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  $|A| \leq |B|$ .<sup>6</sup> 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  $f : A \rightarrow B$ .

Figure 11.1 shows that there is no onto function  $f : \{1, 2\} \rightarrow \{3, 4, 5\}$ : 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 5 will be left out (will not be an output of  $f$ ), so  $f$  will not be onto. We conclude the obvious fact that  $|\{1, 2\}| < |\{3, 4, 5\}|$ .

<sup>6</sup>For instance, we think of the integers as being at least as numerous as the even integers.

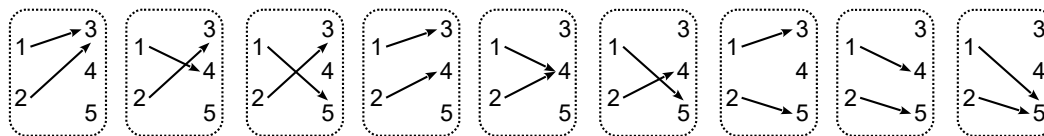


Figure 11.1: All the functions  $f : \{1, 2\} \rightarrow \{3, 4, 5\}$ . None of them are onto.

Conversely, if there *is* an onto function  $f : A \rightarrow B$ , then we can say that  $|A| \geq |B|$ . For instance,  $f(3) = f(4) = 1$  and  $f(5) = 2$  is an onto function  $f : \{3, 4, 5\} \rightarrow \{1, 2\}$ , so we conclude the obvious fact that  $|\{3, 4, 5\}| \geq |\{1, 2\}|$ .

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.

Let's consider two infinite sets:  $\mathbb{N} = \{0, 1, 2, \dots\}$  and  $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ .

What's an onto function from  $\mathbb{Z}$  to  $\mathbb{N}$ ? The absolute value function works:  $f(n) = |n|$ . In fact, for any sets  $A, B$  where  $A \subseteq B$ , we have  $|A| \leq |B|$ , i.e., there is an onto function  $f : B \rightarrow A$ . What is it? One choice is this: pick some fixed element  $a_0 \in A$ . Then define  $f$  for all  $x \in B$  by  $f(x) = x$  if  $x \in A$ , and  $f(x) = a_0$  otherwise. For  $f : \mathbb{Z} \rightarrow \mathbb{N}$ , one possibility is to let  $f(n) = n$  if  $n \geq 0$  and let  $f(n) = 0$  if  $n < 0$ .

How about going the other way? What's an onto function from  $\mathbb{N}$  to  $\mathbb{Z}$ ? This doesn't look quite as easy; it seems like there are "more" integers than nonnegative integers, since  $\mathbb{N} \subsetneq \mathbb{Z}$ . But, there is an onto function:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(2) &= -1 \\ f(3) &= 2 \\ f(4) &= -2 \\ f(5) &= 3 \\ f(6) &= -3 \\ &\dots \end{aligned}$$

Symbolically, we can express  $f$  as  $f(n) = -n/2$  if  $n$  is even and  $\lceil n/2 \rceil$  if  $n$  is odd. But functions don't always need to be expressed symbolically. The partial enumeration above is a perfectly clear way to express the same function.

What about  $\mathbb{N}$  and  $\mathbb{Q}^+$  (the positive rational numbers)? Since  $\mathbb{N}^+ \subset \mathbb{Q}^+$ ,  $|\mathbb{N}| \leq |\mathbb{N}^+| \leq |\mathbb{Q}^+|$ . (Why is the first inequality true?) What about the other way? Is there an onto function  $f : \mathbb{N} \rightarrow \mathbb{Q}^+$ ?

With  $\mathbb{N}$ , there's one nice way to think about onto functions  $f$  with  $\mathbb{N}$  as the domain. We can think of defining  $f : \mathbb{N} \rightarrow \mathbb{Q}^+$  via  $r_0 = f(0)$ ,  $r_1 = f(1)$ ,  $\dots$ , where each  $r_n \in \mathbb{Q}^+$ . In other words, we can describe  $f$  by describing a way to enumerate all the rational numbers in order  $r_0, r_1, \dots$ . What order should we pick?

Each positive rational number  $r$  is defined by a pair of integers  $n, d \in \mathbb{N}^+$ , where  $r = \frac{n}{d}$ . The following order doesn't work:  $r_0 = \frac{1}{1}, r_1 = \frac{1}{2}, r_2 = \frac{1}{3}, r_3 = \frac{1}{4}, \dots, r_{\frac{1}{2}} = \frac{2}{1}, r_{\frac{2}{2}} = \frac{2}{2}, r_{\frac{3}{2}} = \frac{3}{2}, \dots$ . There's infinitely many possible denominators  $d$ , so we can't set  $n = 1$  and iterate through all possible  $d$  before changing  $n$ . We need some way of changing *both*  $n$  and  $d$  to make sure all pairs of positive integers appear in this list.

Here's one way to do it:  $\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{2}{2}, \frac{3}{1}, \frac{1}{4}, \frac{2}{3}, \frac{3}{2}, \frac{4}{1}, \frac{1}{5}, \frac{2}{4}, \frac{3}{3}, \frac{4}{2}, \frac{5}{1}, \frac{1}{6}, \dots$ . In other words, enumerate all  $n, d \in \mathbb{N}^+$  where  $n + d = 2$  (which is just  $n = d = 1$ ), then all  $n, d \in \mathbb{N}^+$  where  $n + d = 3$  (of which there are 2), then all  $n, d \in \mathbb{N}^+$  where  $n + d = 4$  (of which there are 3), then all  $n, d \in \mathbb{N}^+$  where  $n + d = 5$  (of which there are 4), etc. This enumeration shows that  $|\mathbb{N}| \geq |\mathbb{Q}^+|$ .

To see that  $|\mathbb{N}| \geq |\mathbb{Q}|$ , i.e., there's an onto function  $f : \mathbb{N} \rightarrow \mathbb{Q}$ , we can use the same trick as above with  $\mathbb{Z}$ . Let  $g : \mathbb{N} \rightarrow \mathbb{Q}^+$  be the onto function we just defined, and let  $f : \mathbb{N} \rightarrow \mathbb{Q}$  be defined as

$$\begin{aligned} f(0) &= 0 \\ f(1) &= g(0) \\ f(2) &= -g(0) \\ f(3) &= g(1) \\ f(4) &= -g(1) \\ f(5) &= g(2) \\ f(6) &= -g(2) \\ &\dots \end{aligned}$$

Since  $g$  maps onto every positive rational number,  $f$  maps onto every rational number.

What about  $\mathbb{R}$ , the set of real numbers, and  $(0, 1)$ , the set of real numbers strictly between 0 and 1? Since  $(0, 1) \subset \mathbb{R}$ , we know that  $|(0, 1)| \leq |\mathbb{R}|$ .

First, define  $g : (0, 1) \rightarrow \mathbb{R}^+$  as  $g(x) = \frac{1}{x} - 1$ . To see that  $g$  is onto, let  $y \in \mathbb{R}^+$  and note that setting  $x = \frac{1}{y+1}$  means  $g(x) = y$ ; note that  $\frac{1}{y+1} \in (0, 1)$  for any positive real  $y$ . This shows that  $|(0, 1)| \geq |\mathbb{R}^+|$ .

Now we must show  $|\mathbb{R}^+| \geq |\mathbb{R}|$ . Define  $f : \mathbb{R}^+ \rightarrow \mathbb{R}$  as  $f(x) = \log_2 x$ . To see that  $f$  is onto, let  $y \in \mathbb{R}$ . Letting  $x = 2^y$  makes  $f(x) = y$ , noting  $2^y > 0$  for any real  $y$ .

Thus  $|(0, 1)| \geq |\mathbb{R}^+| \geq |\mathbb{R}|$ .

So it seems with many of these infinite sets, we can find an onto function from one to the other. Perhaps  $|A| = |B|$  for all infinite sets? Cantor discovered in 1874 that the answer is no.<sup>7</sup>

The following theorem changed the course of science.

---

<sup>7</sup>He used a different technique in 1874, and in 1891 discovered the technique known as "diagonalization" that we present here.

**Theorem 11.4.1.** *Let  $X$  be any set. Then there is no onto function  $f : X \rightarrow \mathcal{P}(X)$ .*

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

Define the set

$$D = \{ a \in X \mid a \notin f(a) \}.$$

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  $D$ ,

$$a \in D \iff a \notin f(a),$$

so  $D \neq f(a)$ . □

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

This proof works for any set  $X$  at all. In the special case where  $X = \mathbb{N}$ , we can visualize why this technique is called “diagonalization”. Suppose for the sake of contradiction that there is a onto function  $f : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N})$ ; then we can enumerate the subsets of  $\mathbb{N}$  in order  $S_0 = f(0)$ ,  $S_1 = f(1)$ ,  $S_2 = f(2)$ , .... Each set  $S \subseteq \mathbb{N}$  can be represented by an infinite binary sequence  $\chi_S$ , where the  $n$ 'th bit  $\chi_S[n] = 1 \iff n \in S$ . Those sequences are the rows of the following infinite matrix:

	0	1	2	3	...	$k$
$S_0 = \{1, 3, \dots\}$	0	1	0	1	...	
$S_1 = \{0, 1, 2, 3, \dots\}$	1	1	1	1	...	
$S_2 = \{2, \dots\}$	0	0	1	0	...	
$S_3 = \{0, 2, \dots\}$	1	0	1	0	...	
$\vdots$					$\ddots$	
$S_k = D = \{0, 3, \dots\}$	1	0	0	1	...	?

If  $D$  is in the range of  $f$ , then  $S_k = D$  for some  $k \in \mathbb{N}$ . But  $D$  is defined so that  $\chi_D$  is the bitwise negation of the diagonal of the above matrix. So if  $D$  appears as row  $k$ , this gives a contradiction when we ask what is the bit at row  $k$  and column  $k$ .

For two sets  $X, Y$ , we write  $|X| < |Y|$  if there is no onto function  $f : X \rightarrow Y$ . We write  $|X| \geq |Y|$  if it is not the case that  $|X| < |Y|$ ; i.e., if there *is* an onto function  $f : X \rightarrow Y$ . We write  $|X| = |Y|$  if there is a bijection (a 1-1 and onto function)  $f : X \rightarrow Y$ .<sup>8</sup>

We say  $X$  is *countable* if  $|X| \leq |\mathbb{N}|$ ;<sup>9</sup> i.e., if  $X$  is a finite set or if  $|X| = |\mathbb{N}|$ .<sup>10</sup> We say  $X$  is *uncountable* if it is not countable; i.e., if  $|X| > |\mathbb{N}|$ .<sup>11</sup>

<sup>8</sup>By a deep result known as the *Cantor-Bernstein Theorem*, this is equivalent to saying that there is an onto function  $f : X \rightarrow Y$  and another onto function  $g : Y \rightarrow X$ ; i.e.,  $|X| = |Y|$  if and only if  $|X| \leq |Y|$  and  $|X| \geq |Y|$ .

<sup>9</sup>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.

<sup>10</sup>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  $|X| < |\mathbb{N}|$ .

<sup>11</sup>The relations  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ , and  $=$  are transitive: for instance,  $(|A| \leq |B| \text{ and } |B| \leq |C|) \implies |A| \leq |C|$ .

Stating that a set  $X$  is countable is equivalent to saying that its elements can be listed; i.e., that it can be written  $X = \{x_0, x_1, x_2, \dots\}$ , where every element of  $X$  will appear somewhere in the list.<sup>12</sup>

**Observation 11.4.2.**  $|\mathbb{N}| < |\mathbb{R}|$ ; i.e.,  $\mathbb{R}$  is uncountable.

*Proof.* By Theorem 11.4.1,  $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})|$ , so it suffices to prove that  $|\mathcal{P}(\mathbb{N})| \leq |\mathbb{R}|$ ;<sup>13</sup> i.e., that there is an onto function  $f : \mathbb{R} \rightarrow \mathcal{P}(\mathbb{N})$ .

Define  $f : \mathbb{R} \rightarrow \mathcal{P}(\mathbb{N})$  as follows. Each real number  $r \in \mathbb{R}$  has an infinite decimal expansion.<sup>14</sup> For all  $n \in \mathbb{N}$ , let  $r_n \in \{0, 1, \dots, 9\}$  be the  $n^{\text{th}}$  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  $n^{\text{th}}$  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 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.  $\square$

**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  $f : \mathbb{N} \rightarrow A$ , or there is an onto function  $g : A \rightarrow \mathcal{P}(\mathbb{N})$ .

**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 is not provable, nor is its negation.<sup>15</sup> So it will forever remain a hypothesis; we can never hope to prove it either true or false.

Theorem 11.4.1 has immediate consequences for the theory of computing.

**Observation 11.4.3.** There is an undecidable language  $L \subseteq \{0, 1\}^*$ .

*Proof.*  $\{0, 1\}^*$  is countable, as is the set of all TM's. By Theorem 11.4.1,  $\mathcal{P}(\{0, 1\}^*)$ , the set of all binary languages, is uncountable. So some language is not decided by any TM.  $\square$

### 11.4.2 The Halting Problem is undecidable

Observation 11.4.3 shows that some undecidable language must exist. However, it is more challenging to exhibit a particular undecidable language. In the next theorem, we use the technique of diagonalization directly to show that

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

<sup>12</sup>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.

<sup>13</sup>Actually they are equal, but we need not show this for the present observation.

<sup>14</sup>This expansion need not be unique, as expansions such  $0.03000000\dots$  and  $0.02999999\dots$  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  $r$ .

<sup>15</sup>Technically, this isn't quite true. What is true is that either both the Continuum Hypotheses and its negation are both unprovable, or both are provable because *all mathematical statements* are provable, rendering our system of mathematics useless for discovering true theorems.



is undecidable.

**Theorem 11.4.4.** *HALTS is undecidable.*

*Proof.* Assume for the sake of contradiction that HALTS is decidable, by the algorithm  $H$ . Then  $H$  accepts  $\langle M, w \rangle$  if  $M(w)$  halts, and  $H$  rejects  $\langle M, w \rangle$  if  $M(w)$  loops.

Define the TM  $D$  as follows. On input  $\langle M \rangle$  a TM,  $D$  runs  $H(\langle M, M \rangle)$ ,<sup>16</sup> and does the opposite.<sup>17</sup> Now consider running  $D(\langle D \rangle)$ . Does it halt or not? We have

$$\begin{aligned} D(\langle D \rangle) \text{ halts} &\iff H \text{ accepts } \langle D, D \rangle && \text{defn of } H \\ &\iff D(\langle D \rangle) \text{ does not halt,} && \text{defn of } D \end{aligned}$$

a contradiction. Therefore no such algorithm  $H$  exists. □

The following Python code implements this idea:

```

1 def H(M,w):
2     """Function that supposedly decides whether M(w) halts."""
3     raise NotImplementedError()
4
5 def D(M):
6     """ "Diagonalizing" function; given a function M, if M halts
7     when given itself as input, run forever; otherwise halt."""
8     if H(M,M):
9         while True: # loop forever
10             pass
11     else:
12         return # halt
13
14 D(D) # behavior not well-defined! So H cannot be implemented.
```

It is worth examining the proof of Theorem 11.4.4 to see the diagonalization explicitly.

We can give any string as input to a program. Some of those strings represent other programs, and some don't. We just want to imagine what happens when we run a program  $M_i$  on an input  $\langle M_j \rangle$  that represents another program  $M_j$ . If  $M_i$  is not even intended to handle inputs that are programs (for instance if it tests integers for primality, or graphs for Hamiltonian paths), then the behavior of  $M_i$  on inputs representing programs may not be interesting. Nonetheless,  $M_i$  either halts on input  $\langle M_j \rangle$ , or it doesn't.

We then get an infinite matrix describing all these possible behaviors.

<sup>16</sup>That is,  $D$  runs  $H$  to determine if  $M$  halts on the string that is the binary description of  $M$  itself.

<sup>17</sup>In other words, halt if  $H$  rejects, and loop if  $H$  accepts. Since  $H$  is a decider,  $D$  will do one of these.

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\dots$	$\langle D \rangle$
$M_0$	loop	halt	loop	halt		
$M_1$	halt	halt	halt	halt		
$M_2$	loop	loop	halt	loop		
$M_3$	halt	loop	halt	loop		
$\vdots$					$\ddots$	
$D$	halt	loop	loop	halt	$\dots$	?

If the halting problem were decidable, then the program  $D$  would be implementable, and its behavior on each input, described by the row with  $D$  on the left, would be the element-wise opposite of the diagonal of this matrix. But this gives a contradiction since the entry in the diagonal corresponding to  $D$  would then be undefined. Since  $D$  can be implemented if HALTS is decidable, this establishes that HALTS is not decidable.

end of lecture 10a

## 11.5 Optional: The far-reaching consequences of undecidability

We have seen above that because the Halting Problem is undecidable, any other question about the “long-term behavior” of a program is also undecidable. But undecidability has farther-reaching consequences than this. The next is a theorem from mathematical logic, which says something about the limits of knowledge itself.

### 11.5.1 Gödel’s Incompleteness Theorem

Gödel’s Incompleteness Theorem is among the few mathematical theorems to have captured the imagination of a large number of non-mathematicians (the Pythagorean Theorem being another notable example.) It has a formal statement that almost no one other than logicians manage to state correctly. There’s a whole book dedicated to all the ways people misunderstand Gödel’s Incompleteness Theorem: <https://www.crcpress.com/Godels-Theorem-An-Incomplete-Guide-to-Its-Use-and-Abuse/Franzen/p/book/9781568812380>

For the sake of simplicity, we will continue this tradition of incorrectly stating Gödel’s Incompleteness Theorem: very roughly, it says that there is a mathematical statement that is true, but that cannot be proved to be true.<sup>18</sup>

<sup>18</sup>More precisely, it states that either there is a true statement that cannot be proved, OR some other crazy thing

The title of Section 11.5 claims that Gödel’s Incompleteness Theorem is a *consequence* of undecidability. However, Gödel’s Incompleteness Theorem preceded Turing’s proof of the undecidability of the Halting Problem by 5 years. Gödel glimpsed the first outlines of the full theory of undecidability, which was opened wide with Turing’s proof of the undecidability of the Halting Problem. So despite Gödel’s Incompleteness Theorem being *chronologically* first, there’s a certain sense in which undecidability is *conceptually* at the center, with Gödel’s Incompleteness Theorem being one application of these ideas (in fact, the first application, discovered before the full weight of the ideas was truly understood). Also, one can use Turing’s ideas to give a far simpler proof of Gödel’s Theorem than one tends to find in textbooks on mathematical logic (and certainly simpler than Gödel’s original proof).

We prove a slightly weaker statement than what is usually called Gödel’s Incompleteness Theorem; see <http://www.scottaaronson.com/blog/?p=710> for a discussion of the technicalities. What we prove is that there is a mathematical statement  $T$  such that either:

1.  $T$  and  $\neg T$  are both unprovable (thus one is true but not provable), or
2.  $T$  is false but provable.

Before going on, it is worth considering the importance of the second statement. If it were true, it would mean that our formalization of mathematics is useless: if false statements are provable, then proving a statement doesn’t actually inform us whether it is true.<sup>19</sup> The mathematical statement  $T$  will be of the form “a certain Turing machine halts on a certain input.”

We show that if the above were false, then the Halting Problem would be decidable, a contradiction. If the above is false, then for all mathematical statements  $T$ , both of the following hold:

---

happens that renders the whole discussion irrelevant. The most common objection to Gödel’s Theorem is: “Wait, if Gödel’s Theorem proves that a certain statement  $T$  is true but unprovable, isn’t **Gödel’s Theorem itself** a proof of  $T$ ? So how can  $T$  be unprovable?” But Gödel’s Theorem *doesn’t* actually prove  $T$  is true: it proves the logical disjunction “ $T$  is true OR that other crazy thing is true”, so Gödel’s Theorem is not actually a proof of  $T$ ; instead it is a proof of  $(T \vee \text{crazy-thing})$ . Sneak preview: the crazy thing is “*some false statement is provable*”.

<sup>19</sup>Upon hearing that a mathematical system is “useless” if it gets a statement wrong, one might object, “That’s a bit harsh... sure, there’s this one false statement  $T$  that is provable, so the mathematical system misleads us on  $T$ , but maybe the system gets every other statement  $T' \neq T$  correct, in the sense of being able to prove  $T'$  only if  $T'$  is actually true, so the mathematical system is mostly useful despite getting one statement wrong.” It turns out that, with a slight adjustment of the definition of “wrong”, this rosy scenario is impossible: if the system gets one statement wrong, then it gets them *all* wrong.

The stronger statement of Gödel’s Incompleteness Theorem (actually proven by someone named Rosser) replaces the second statement “ $T$  is false but provable” (a condition called *unsoundness*) with “ $T$  and  $\neg T$  are both provable” (a condition called *inconsistency*). Then, if the system were inconsistent, it could prove *every* statement  $S$ ! (including every statement  $S$  and its negation  $\neg S$ , just like  $T$  and  $\neg T$ ). In other words, if there’s even one statement  $T$  where the mathematical theory “can’t decide” whether  $T$  or  $\neg T$  holds, then in fact it can’t make a decision on *any other* statement either. To see this, first convince yourself that  $\neg T \implies (T \implies S)$  is a logical tautology, i.e., holds for every pair of statements  $T$  and  $S$ . (Just write out the truth table for the Boolean formula, for all 4 possible settings of true and false to  $T$  and  $S$ .) Then, let  $S$  be an arbitrary statement and suppose  $T$  and  $\neg T$  are both provable. Apply *modus ponens* twice: since the hypothesis  $\neg T$  is provable, then the conclusion  $(T \implies S)$  is provable, and since the hypothesis  $T$  is provable, then the conclusion  $S$  is provable.

1. at least one of  $T$  or  $\neg T$  is provable, and
2. if  $T$  is provable, then it is true.

To solve the halting problem on instance  $\langle M, w \rangle$ , we begin enumerating *all the proofs there are*. This is a key property of any reasonable mathematical system: one must be able to enumerate the proofs, for instance in length-lexicographical order (formal proofs are, after all, simply finite strings). Upon encountering each new proof, check whether it is a proof of the statement  $T = “M(w) \text{ halts}”$ , or a proof of the statement  $\neg T = “M(w) \text{ does not halt}”$ . (Most of the enumerated proofs will be neither because they prove something else, such as “ $3 < 4$ ” or “ $P \neq NP$ ”.)

By (1) at least one of  $T$  or  $\neg T$  has a proof, so the search for a proof of either  $T$  or  $\neg T$  will eventually terminate. By (2), whichever proof is found is of a true statement, so if we accept or reject  $\langle M, w \rangle$  according to which statement it proved, the answer will be correct. But this would decide the halting problem, a contradiction.

So there is Gödel’s Incompleteness Theorem (slightly weakened from the full version, and still a bit informal since we didn’t get into formal definitions of *proof* or *formal mathematical system*): so long as it is impossible to prove false statements—certainly a requirement of any reasonable system of mathematics—unfortunately it is also impossible to prove some true statements. In other words, our system for proving mathematical statements is necessarily incomplete and unable to help us determine the truth of all possible statements.

### 11.5.2 Prediction of physical systems

DD: write up some self-assembly/chemical reaction network prediction problems that are undecidable

end of lecture 10b

## Appendix A

# Reading Python

### A.1 Installing Python

To install basic Python, go here: <https://www.python.org/downloads/>. I prefer a distribution of Python called Anaconda, which comes with extra nice tools such as the “notebook” discussed below: <https://www.anaconda.com/download/>.

### A.2 Code from this book

The Python code in this book can be found in an ipython notebook:

<https://github.com/dave-doty/UC-Davis-ECS120/blob/master/120.ipynb>

If you just want to read the code without running it, you can look at it directly on GitHub by clicking on the link above.

To try running it yourself, download the 120.ipynb file above. (Go to <https://github.com/dave-doty/UC-Davis-ECS120>, right/optio-click the file 120.ipynb, and select “Save link as...”/“Download linked file as...”)

The code is formatted for Python 3, which has slightly different syntax and semantics than Python 2. If you have Python 2, a few things have to be changed for it to run; these are marked with comments of the form

```
# for python 2 include the next line
```

above a line that needs to be uncommented. Similarly, some lines preceded by

```
# for python 3 include the next line
```

should be commented out.

If you have ipython installed (there are a few ways to do this; I recommend Anaconda: <https://www.continuum.io/downloads>), open a command line in the directory where 120.ipynb is stored and type `jupyter notebook 120.ipynb`

If you don’t have ipython installed, go to <https://try.jupyter.org/>, click the “upload” button, and choose the 120.ipynb file from your local computer. Once it’s uploaded, click on it in your browser.

### A.3 Tutorial on reading Python

This section provides a short tutorial *for experienced programmers* on how to *read* the Python code presented in this book. It is not a tutorial on how to program, or on how to write Python. Instead, it assumes that the reader already has some familiarity with an imperative programming language such as C, C++, C#, Java, Javascript, Go, Rust, Ruby, etc. (i.e., has taken a programming course using such a language), and that the reader is able to follow basic pseudocode conventions such as those used in any undergraduate algorithms/data structures course, using generic programming constructs such as if statements, while/for loops, assignment statements, function calls, basic data structures such as lists, arrays, sets, dictionaries (*a.k.a.* key-value maps, hash tables).

The point of this section is to explain those specific aspects of Python that may be unfamiliar, just well enough to be able to read the code examples given in this book. But Python is called “executable pseudocode” for a reason: it is generally quite easy to read.

If you don’t already have Python installed, you can use <https://repl.it/languages/python> or <https://try.jupyter.org/> to try out snippets of Python code to see what it does. If you aren’t sure what some code does, **don’t guess!** Try running it to see.

**Variables.** Python has no explicit declaration of variables. A variable is created by assigning a value to it for the first time:

```
1 a = 3      # creates variable a and assigns it value 3
2 print(a)
3 print(b)   # causes error because variable b does not yet exist
4 b = 4
```

**Strings.** Python strings can be created with double quotes as in C++ or Java:

```
1 s = "This is a string."
```

There is no separate `char` type. When you would use a `char` in C/C++/Java, in Python you simply use a string of length 1.

Strings can also be created with single-quotes, which is convenient because double quotes can be used inside of single-quoted strings, and vice versa, without escaping them with a backslash:

```
1 s = 'These "double quotes" do not need backslash escaping.'
2 s2 = "These \"double quotes\" need backslash escaping."
3 s3 = "It 'goes the other way' too."
```

Finally, Python has *multiline strings*, which start and end with three quotes instead of just one, that allow newlines. Both types of quote can be used in them without escapes:

```
1 s = """This is the "first" line.
2 This is the 'second' line."""
3 s2 = '''This is the "first" line.
4 This is the 'second' line.'''
```

Without triple quotes, you would need to use `"\n"` to insert newlines as in C/C++/Java, and would need to escape some quotes:

```
1 s3 = "This is the \"first\" line.\nThis is the 'second' line."
```

**Indentation.** Like pseudocode, blocks of Python code are denoted by indentation, *not* curly braces as in C, C++, or Java. Consider the following code, which prints only `c` to the screen:

```
1 if 3 == 4:
2     print("a")
3     print("b")
4 print("c")
```

However, the following code prints `b` followed by `c`.

```
1 if 3 == 4:
2     print("a")
3 print("b")
4 print("c")
```

**Documentation.** Often the first line of a function will be a multiline string not assigned to any variable. This is called a *docstring* and is interpreted as a comment documenting the function:

```
1 def square(x):
2     """Compute the square of x.
3
4     For example, square(5) == 25 and square(3) == 9."""
5     return x*x
```

Since the docstring is not assigned to any variable, it has no effect on the program. Docstrings are used by certain tools to automatically generate documentation for Python libraries: <https://wiki.python.org/moin/DocumentationTools>

**for loops.** The code `for i in range(n):` in Python is similar to `for (int i=0; i<n; i++)` in C/C++/Java: the loop has  $n$  iterations, letting `i` take on the values 0, 1, 2, ...,  $n - 1$ . So the following code prints the numbers from 0 to 9:

```
1 for i in range(10):
2     print(i)
```

But, many times in C/C++ when you would use a loop like `for (int i=0; i<n; i++)`, it would be to iterate over elements of an array or other data structure, i.e.,

```
1 int arr[] = {2,3,5,7,11};
2 for (int i=0; i<5; i++) {
3     int num = arr[i];
4     printf("The current integer is %d", num);
5 }
```

Python has a special syntax for this sort of loop, discussed below, which executes one iteration for every element of a container. This is better to use than to loop over the indices, because it is more readable, and because it eliminates the possible error that the index `i` is out of bounds. For instance, if above loop instead were `for (int i=0; i<=5; i++)`, `i` would go off the end of the array.

**lists and sets.** `v = [1,2,3,4,5]` makes a list with 5 elements. A Python list is like an array in C/C++/Java, but actually more like `std::vector` in C++ or `java.util.ArrayList` in Java, because Python lists can grow and shrink. A list can have duplicates such as `d = [1,2,3,2,3]`, but a set cannot.

`s = set(d)` creates the set `{1,2,3}`, eliminating the duplicates in `d`. Another way to create this set is the line `s = {1,2,3}`. Tuples such as `(2,3,4)` are lists that *cannot* be modified (for example, `l[0] = 5` works for a list `l`, but `t[0] = 5` fails for a tuple `t`).

You can iterate over all elements of a list/tuple/set with a for loop using the `in` keyword:<sup>1</sup>

```
1 for s in ["a","b","cd"]:
2     print(4*s)
```

prints "aaaa", "bbbb", and "cdcdcdcd" to the screen.

**list/set comprehensions.** These are one of the most useful features and one of the best reasons to familiarize yourself with mathematical set builder notation. Many clunky lines of code can be replaced by a simple line that expresses the same idea. Recall that `range(n)` is (something like) a list with the integers from 0 to `n - 1`.<sup>2</sup> The following code creates various other lists/sets from it:

```
1 ints = range(10) # range with elements 0,1,2,3,4,5,6,7,8,9
2 a = [3*n for n in ints] # [0,3,6,9,12,15,18,21,24,27]
3 b = [n for n in ints if n % 2 == 0] # [0,2,4,6,8]
4 c = [n//3 for n in ints] # [0,0,0,1,1,1,2,2,2,3]
5 d = {n//3 for n in ints} # {0,1,2,3}
6 e = {3n for n in ints if n % 2 == 0} # {0,6,12,18,24}
```

`a`, `b`, and `c` are lists, but `d` and `e` are sets, so cannot have duplicates.

In general, if we have some expression `<expr>` (such as `n//3`) and optionally some Boolean expression `<phi>` (such as `n%2==0`), the following code:

```
1 new_lst = [<expr> for n in lst if <phi>]
```

is equivalent to

```
1 new_lst = []
2 for n in lst:
3     if <phi>:
4         new_lst.append(<expr>)
```

<sup>1</sup>The latest C++ standard now supports a similar idea called a “range-based for loop”: <http://en.cppreference.com/w/cpp/language/range-for>.

<sup>2</sup>Technically it returns something called a “range” type in Python 3. The main difference with a list is that the numbers don’t get stored in memory, but we can iterate over a range just like a list, and the numbers will be generated as we need them. In some languages this is called “lazy evaluation”.



Omitting the `if` expression means all of the elements are added, i.e.,

```
1 new_lst = [<expr> for n in lst]
```

is equivalent to

```
1 new_lst = []
2 for n in lst:
3     new_lst.append(<expr>)
```

Sets may be similarly constructed:

```
1 new_set = {<expr> for n in lst if <phi>}
```

is equivalent to

```
1 new_set = set() # need to use set() constructor; {} makes an empty
                  dict, not a set
2 for n in lst:
3     if <phi>:
4         new_set.add(<expr>)
```

For example, `s = {3*n for n in range(10) if n%2==0}` is equivalent to

```
1 s = set()
2 for n in range(10):
3     if n%2==0:
4         s.add(3*n)
```

They both create the set  $\{0, 6, 12, 18, 24\}$ . Note the similarity to the mathematical set-builder notation  $s = \{3n \mid n \in \{0, 1, \dots, 9\}, n \text{ is even}\}$ .

The big advantage of list/set comprehension notation isn't so much that you have to type fewer keystrokes (although you do save a few). The main advantage is *readability*. The notation clearly communicates to anyone fluent in Python what is the purpose of the line of code: to take items from a list/set<sup>3</sup>, perhaps filter some of them out with the `if` keyword, and process the rest using the expression at the beginning.

You may have experience with functional programming, with `map` and `filter` keywords to transform lists. The expression `<expr>` above is like an anonymous function used with `map`, and the Boolean expression `<phi>` is like an anonymous Boolean function used with `filter`. In fact, Python also has the keywords `map` and `filter`, and they work the same way. However, it is conventional to prefer a list comprehension to `map` and/or `filter`, since it is usually more readable. Similarly, `reduce` (also called `foldl/foldr` in functional languages) exists in Python (in `functools` in Python 3), but an explicit `for` loop is usually more readable. See

- <http://www.artima.com/weblogs/viewpost.jsp?thread=98196>
- <https://google.github.io/styleguide/pyguide.html>
- <https://stackoverflow.com/questions/5426754/google-python-style-guide>

---

<sup>3</sup>More generally, any object that can be “iterated over”: lists, sets, tuples, strings, and some other objects.

**itertools.** The `itertools` package is very useful for checking the various kinds of substructures of data structures that are common in algorithms. For example, `itertools.combinations(x,k)` takes all subsequences of length `k` from `x`. For example, this code:

```
1 import itertools
2 x = [2,3,5]
3 for t in itertools.combinations(x, 2):
4     print(t)
```

prints all the ordered pairs of elements from `x`, in the order they originally appear:

```
(2, 3)
(2, 5)
(3, 5)
```

To make the code easier to read, we can also import only the function `combinations`, and change its name to something else (since we use it to get *subsets* of a fixed size, even though for convenience we often use Python lists to represent sets):

```
1 from itertools import combinations as subsets
2 x = [2,3,5]
3 for t in subsets(x, 2):
4     print(t)
```

The above code is more “literate”; it’s straightforward to read the `for` loop in English as “for all `t` that are subsets of `x` of size 2.”

The following code:

```
1 from itertools import combinations as subsets
2 x = [1,2,3,4,5]
3 for t in subsets(x, 3):
4     print(t)
```

prints all ordered triples:

```
(1, 2, 3)
(1, 2, 4)
(1, 2, 5)
(1, 3, 4)
(1, 3, 5)
(1, 4, 5)
(2, 3, 4)
(2, 3, 5)
(2, 4, 5)
(3, 4, 5)
```

Strings such as `"abc"` are not technically lists of characters (in fact Python has no `char` data type; individual characters are the same thing as strings of length 1), but many of the same functions that work on a list also work on a string, as though it were a list of length-1 strings. For example:

```
1 from itertools import combinations as subsets
2 x = "abc"
3 for t in subsets(x, 2):
4     print(t)
```

prints

```
('a', 'b')
('a', 'c')
('b', 'c')
```